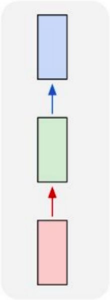
Recurrent Neural Networks and LSTMs

Pawan Goyal

CSE, IIT Kharagpur

June 21st, 2019

Vanilla Neural Networks

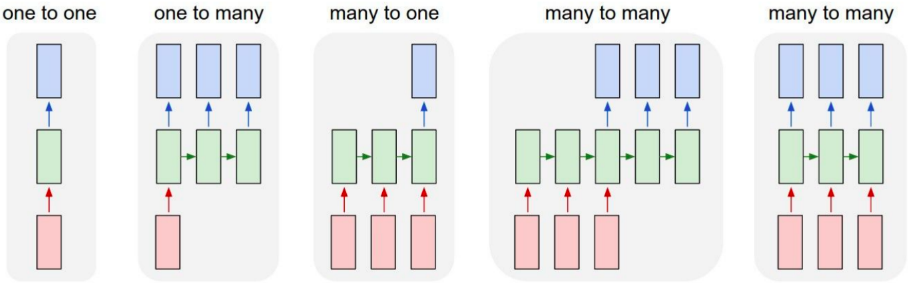


Sequential Data

However, the sequence of data matters for many applications

- Machine translation / text generation
- Speech Recognition
- Stock price prediction

Recurrent Neural Networks: Process Sequences



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

Recurrent Neural Networks (RNNs)

- RNNs are neural networks, specialized for processing a sequence of values $x^{(1)}, \dots, x^{(\tau)}$.
- Can scale to much longer sequences that would be practical for networks without sequence-based specialization
- Most networks can process sequences of variable length

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
- It does so by sharing parameters across different parts of a model

Recurrent Neural Networks

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

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

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

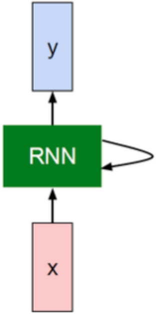
new state

some function
with parameters W

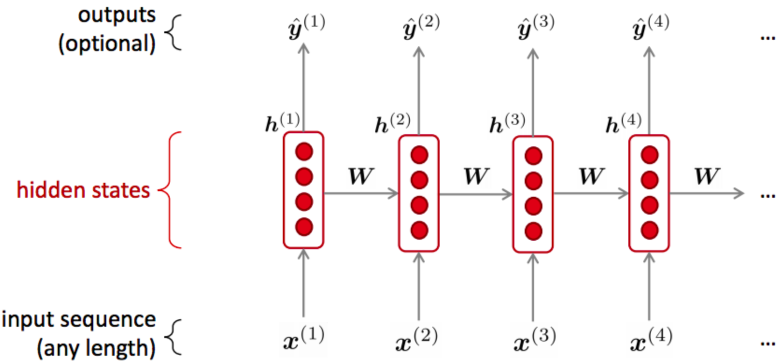
old state

input vector at
some time step

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



Recurrent Neural Networks



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

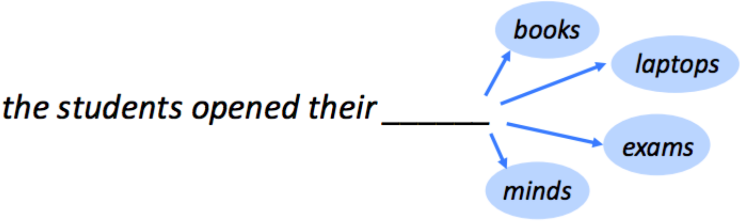
$$
\begin{aligned}
& L(\{x^{(1)},\ldots,x^{(\tau)}\},\{y^{(1)},\ldots,y^{(\tau)}\}) \\
= & \sum_t L^{(t)} \\
= & -\sum_t log\ p_{model}(y^{(t)}|\{x^{(1)},\ldots,x^{(\tau)}\})
\end{aligned}
$$

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 $\hat{y}^{(t)}$

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

An example problem: Language Modeling

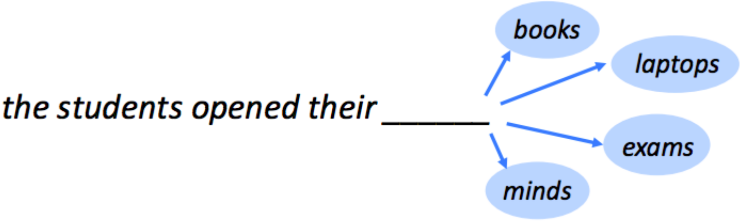
Language Modeling is the task of predicting what word comes next.



- **Goal:** Compute the probability of a sentence or sequence of words:

$$P(W) = P(w_1, w_2, w_3, \ldots, w_n)$$

Language Modeling is the task of predicting what word comes next.



- **Goal:** Compute the probability of a sentence or sequence of words:

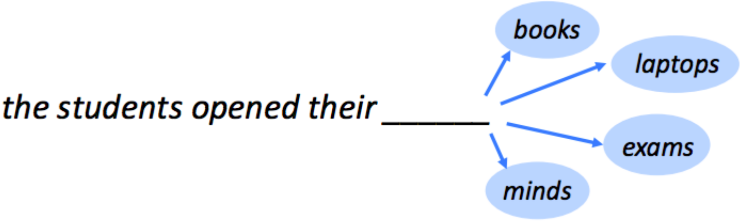
$$P(W) = P(w_1, w_2, w_3, \dots, w_n)$$

- **Related Task:** probability of an upcoming word:

$$P(w_4 | w_1, w_2, w_3)$$

An example problem: Language Modeling

Language Modeling is the task of predicting what word comes next.



- **Goal:** Compute the probability of a sentence or sequence of words:

$$P(W) = P(w_1, w_2, w_3, \ldots, w_n)$$

- **Related Task:** probability of an upcoming word:

$$P(w_4 | w_1, w_2, w_3)$$

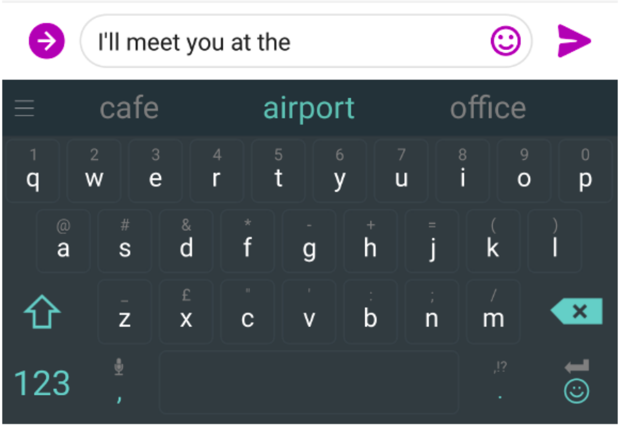
- A model that computes either of these is called a **language model**

Language Modeling

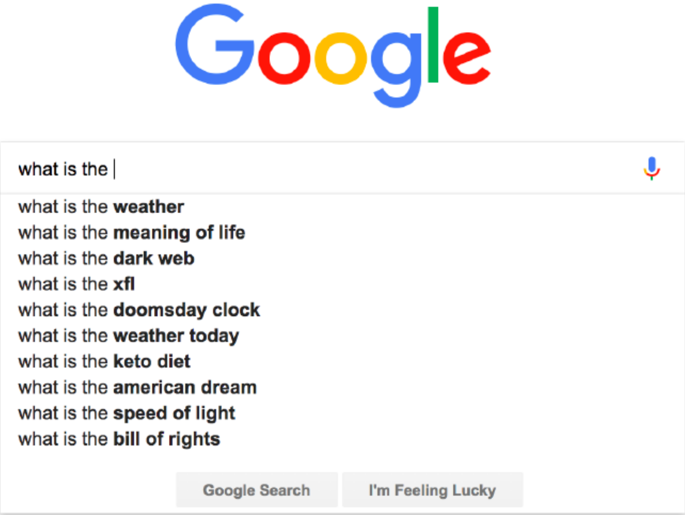
- You can also think of a language model as a system that assigns probability to a piece of text.
- For example, if we have some text $x^{(1)}, \ldots, x^{(T)}$, then the probability of this text (according to the Language Model) is:

$$
\begin{aligned}
P(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(T)}) &= P(\boldsymbol{x}^{(1)}) \times P(\boldsymbol{x}^{(2)} |\, \boldsymbol{x}^{(1)}) \times \cdots \times P(\boldsymbol{x}^{(T)} |\, \boldsymbol{x}^{(T-1)}, \ldots, \boldsymbol{x}^{(1)}) \\
&= \prod_{t=1}^{T} \underbrace{P(\boldsymbol{x}^{(t)} |\, \boldsymbol{x}^{(t-1)}, \ldots, \boldsymbol{x}^{(1)})}_{\text{This is what our LM provides}}
\end{aligned}
$$

You use language models every day!



You use language models every day!



n-gram language models

the students opened their ______

Question: How to learn a Language Model?

Answer (pre- Deep Learning): learn a *n*-gram Language Model!

Definition: A *n*-gram is a chunk of *n* consecutive words.

- unigrams: "the", "students", "opened", "their"
- bigrams: "the students", "students opened", "opened their"
- trigrams: "the students opened", "students opened their"
- 4-grams: "the students opened their"

Idea: Collect statistics about how frequent different n-grams are, and use these to predict next word.

n-gram language models

- First we make a simplifying assumption: $\boldsymbol{x}^{(t+1)}$ depends only on the preceding *n-1* words.

$$P(\boldsymbol{x}^{(t+1)}|\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(1)}) = P(\boldsymbol{x}^{(t+1)}|\overbrace{\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(t-n+2)}}^{n\text{-1 words}})$$

(assumption)

prob of a n-gram

prob of a (n-1)-gram

$$= \frac{\boxed{P(\boldsymbol{x}^{(t+1)},\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(t-n+2)})}}{\boxed{P(\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(t-n+2)})}}$$

(definition of conditional prob)

- **Question:** How do we get these *n*-gram and (*n*-1)-gram probabilities?
- **Answer:** By counting them in some large corpus of text!

$$\approx \frac{\text{count}(\boldsymbol{x}^{(t+1)},\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(t-n+2)})}{\text{count}(\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(t-n+2)})}$$

(statistical approximation)

n-gram language models: Example

Suppose we are learning a 4-gram Language Model.

~~as the proctor started the clock, the~~ $\underbrace{\textit{students opened their}}_{\text{condition on this}}$ ______

discard

$$P(\boldsymbol{w}|\text{students opened their}) = \frac{\text{count(students opened their } \boldsymbol{w})}{\text{count(students opened their)}}$$

For example, suppose that in the corpus:

- "students opened their" occurred 1000 times
- "students opened their books" occurred 400 times
  - → P(books | students opened their) = 0.4
- "students opened their exams" occurred 100 times
  - → P(exams | students opened their) = 0.1

Should we have discarded the "proctor" context?

Sparsity Problems with n-gram Language Model

Sparsity Problem 1

Problem: What if *"students opened their $\boldsymbol{w}$"* never occurred in data? Then $\boldsymbol{w}$ has probability 0!

(Partial) Solution: Add small δ to the count for every $\boldsymbol{w} \in V$. This is called *smoothing*.

$$P(\boldsymbol{w}|\text{students opened their}) = \frac{\boxed{\text{count}(\text{students opened their } \boldsymbol{w})}}{\boxed{\text{count}(\text{students opened their})}}$$

Sparsity Problem 2

Problem: What if *"students opened their"* never occurred in data? Then we can't calculate probability for *any* $\boldsymbol{w}$!

(Partial) Solution: Just condition on *"opened their"* instead. This is called *backoff*.

Note: Increasing *n* makes sparsity problems *worse*. Typically we can't have *n* bigger than 5.

Storage Problems with n-gram Language Model

Storage: Need to store count for all *n*-grams you saw in the corpus.

$$P(\boldsymbol{w}|\text{students opened their}) = \frac{\boxed{\text{count}(\text{students opened their } \boldsymbol{w})}}{\text{count}(\text{students opened their})}$$

Increasing *n* or increasing corpus increases model size!

A fixed-window neural language model

~~*as the proctor started the clock*~~ *the students opened their* ______

discard

fixed window

output distribution

$$\hat{\boldsymbol{y}} = \mathrm{softmax}(\boldsymbol{U}\boldsymbol{h} + \boldsymbol{b}_2) \in \mathbb{R}^{|V|}$$

hidden layer

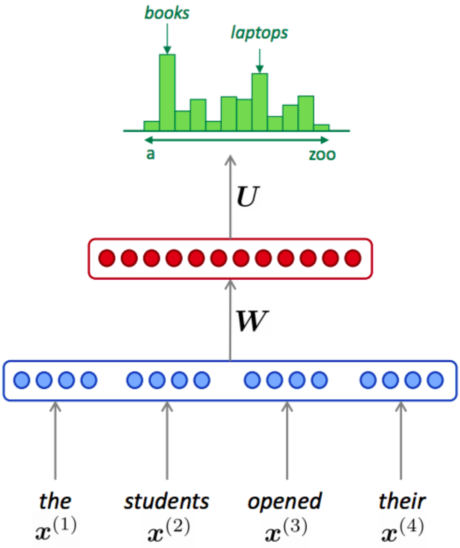
$$\boldsymbol{h} = f(\boldsymbol{W}\boldsymbol{e} + \boldsymbol{b}_1)$$

concatenated word embeddings

$$\boldsymbol{e} = [\boldsymbol{e}^{(1)}; \boldsymbol{e}^{(2)}; \boldsymbol{e}^{(3)}; \boldsymbol{e}^{(4)}]$$

words / one-hot vectors

$$\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(3)}, \boldsymbol{x}^{(4)}$$



A fixed-window neural language model

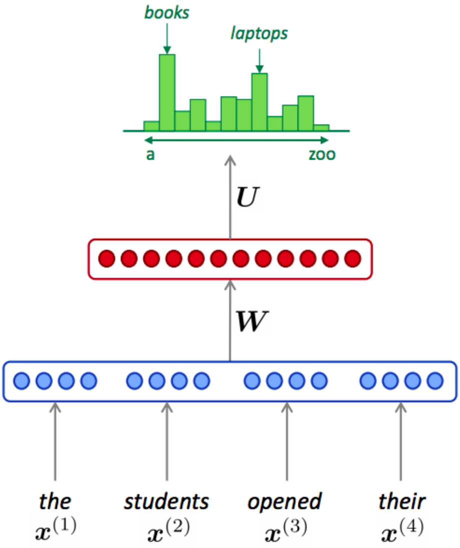
Improvements over n-gram LM:

- No sparsity problem
- Model size is O(n) not O(exp(n))

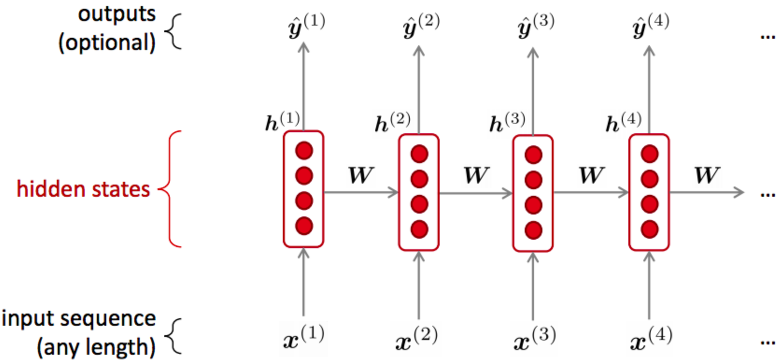
Remaining **problems**:

- Fixed window is too small
- Enlarging window enlarges W
- Window can never be large enough!
- Each $x^{(i)}$ uses different rows of W. We don't share weights across the window.

We need a neural architecture that can process *any length input*



Recurrent Neural Networks



Core Idea

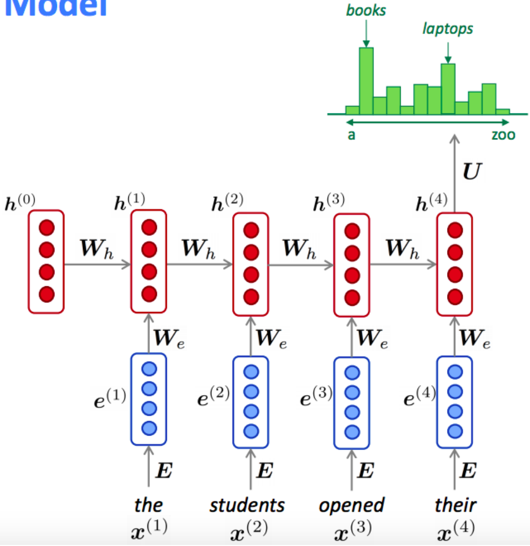
Apply the same weights repeatedly!

A RNN Language Model

$$\hat{\boldsymbol{y}}^{(4)} = P(\boldsymbol{x}^{(5)}|\text{the students opened their})$$

output distribution

$$\hat{\boldsymbol{y}}^{(t)} = \text{softmax}\left(\boldsymbol{U}\boldsymbol{h}^{(t)} + \boldsymbol{b}_2\right) \in \mathbb{R}^{|V|}$$



hidden states

$$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_e\boldsymbol{e}^{(t)} + \boldsymbol{b}_1\right)$$

$\boldsymbol{h}^{(0)}$ is the initial hidden state

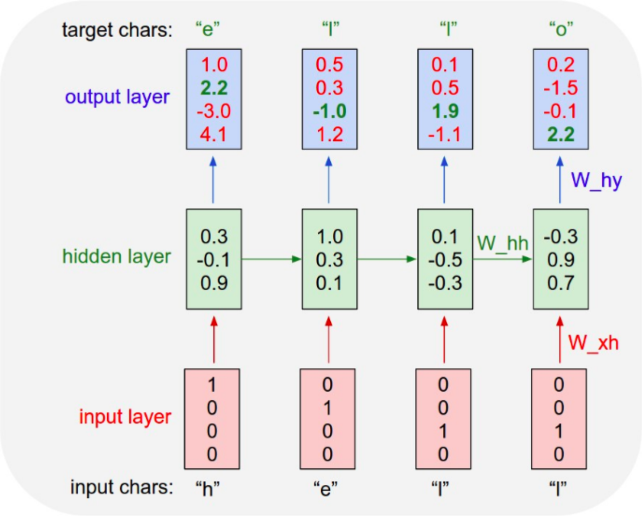
word embeddings

$$\boldsymbol{e}^{(t)} = \boldsymbol{E}\boldsymbol{x}^{(t)}$$

words / one-hot vectors

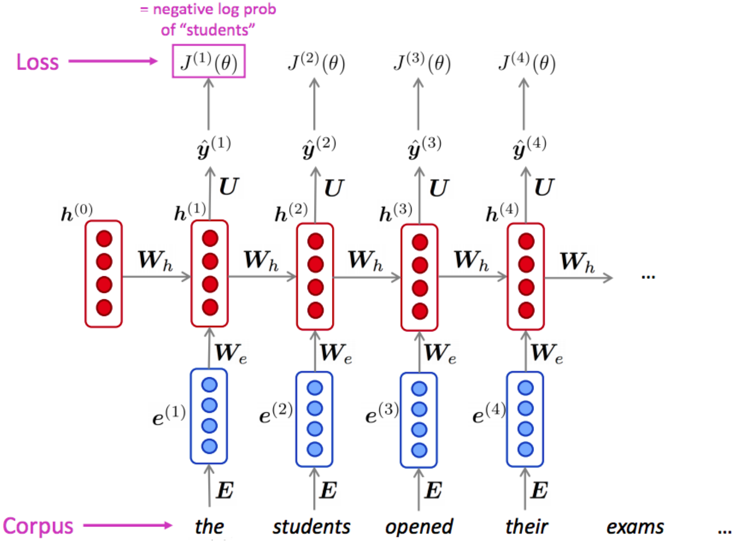
$$\boldsymbol{x}^{(t)} \in \mathbb{R}^{|V|}$$

Example RNN

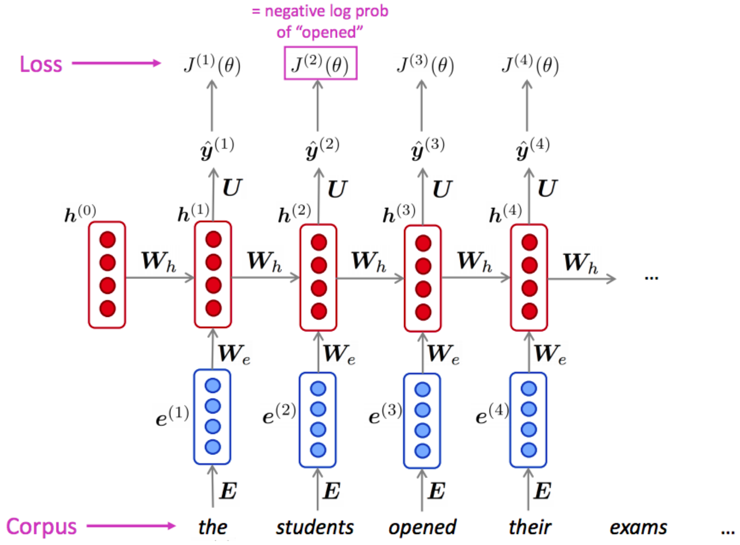


target chars:
"e"
"l"
"l"
"o"
output layer
1.0
2.2
-3.0
4.1
0.5
0.3
-1.0
1.2
0.1
0.5
1.9
-1.1
0.2
-1.5
-0.1
2.2
W_hy
hidden layer
0.3
-0.1
0.9
1.0
0.3
0.1
0.1
-0.5
-0.3
W_hh
-0.3
0.9
0.7
W_xh
input layer
1
0
0
0
0
1
0
0
0
0
1
0
0
0
1
0
input chars:
"h"
"e"
"l"
"l"

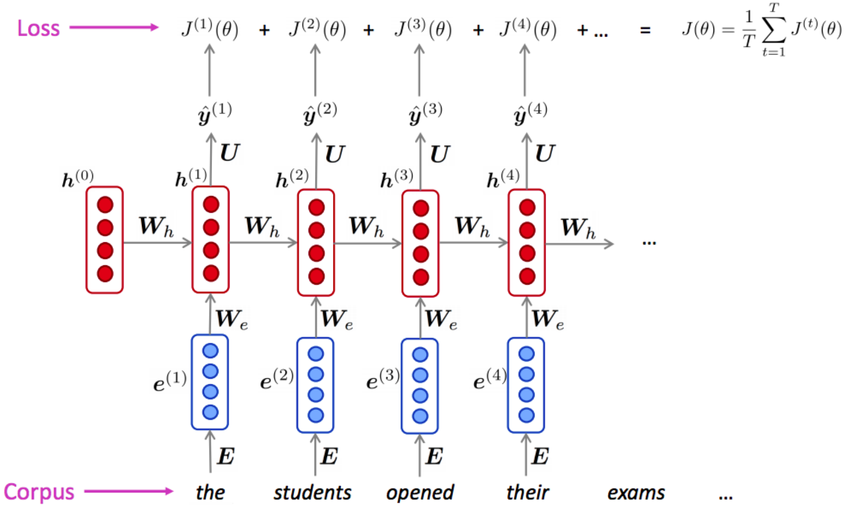
Training a RNN language model



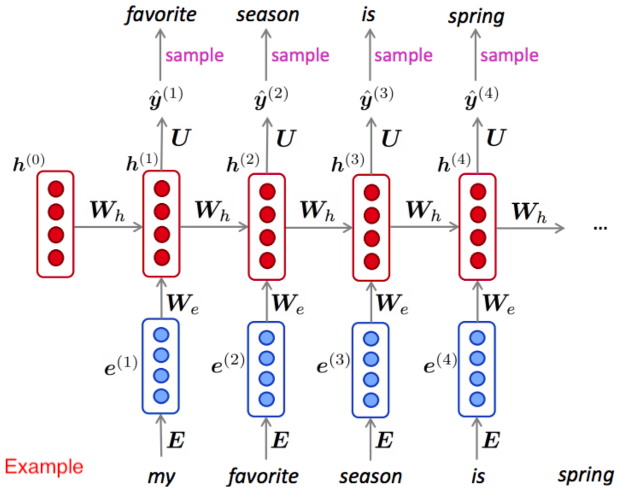
Training a RNN language model



Training a RNN language model



Generating text with a RNN Language Model



Generating text with a RNN Language Model

- You can train a RNN-LM on any kind of text, then generate text in that style.
- RNN-LM trained on Obama speeches:

The United States will step up to the cost of a new challenges of the American people that will share the fact that we created the problem. They were attacked and so that they have to say that all the task of the final days of war that I will not be able to get this done.

Generating text with a RNN Language Model

- You can train a RNN-LM on any kind of text, then generate text in that style.
- RNN-LM trained on *Harry Potter*:

"Sorry," Harry shouted, panicking—"I'll leave those brooms in London, are they?"

"No idea," said Nearly Headless Nick, casting low close by Cedric, carrying the last bit of treacle Charms, from Harry's shoulder, and to answer him the common room perched upon it, four arms held a shining knob from when the spider hadn't felt it seemed. He reached the teams too.

Evaluating Language Models

- The standard evaluation metric for Language Models is perplexity.

$$\text{perplexity} = \underbrace{\prod_{t=1}^{T} \left(\frac{1}{P_{\text{LM}}(\boldsymbol{x}^{(t+1)} \mid \boldsymbol{x}^{(t)}, \ldots, \boldsymbol{x}^{(1)})} \right)}_{\text{Inverse probability of corpus, according to Language Model}}^{1/T}$$

Normalized by number of words

- This is equal to the exponential of the cross-entropy loss $J(\theta)$:

$$= \prod_{t=1}^{T} \left(\frac{1}{\hat{\boldsymbol{y}}^{(t)}_{\boldsymbol{x}_{t+1}}} \right)^{1/T} = \exp\left(\frac{1}{T} \sum_{t=1}^{T} -\log \hat{\boldsymbol{y}}^{(t)}_{\boldsymbol{x}_{t+1}} \right) = \exp(J(\theta))$$

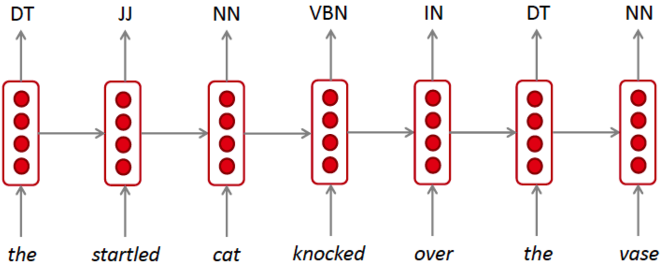
$\boxed{\textbf{Lower} \text{ perplexity is better!}}$

Why should we care about language modeling?

- Language Modeling is a benchmark task that helps us measure our progress on understanding language
- Language Modeling is a subcomponent of many NLP tasks, especially those involving generating text or estimating the probability of text:
  - Predictive typing
  - Speech recognition
  - Handwriting recognition
  - Spelling/grammar correction
  - Authorship identification
  - Machine Translation
  - Summarization

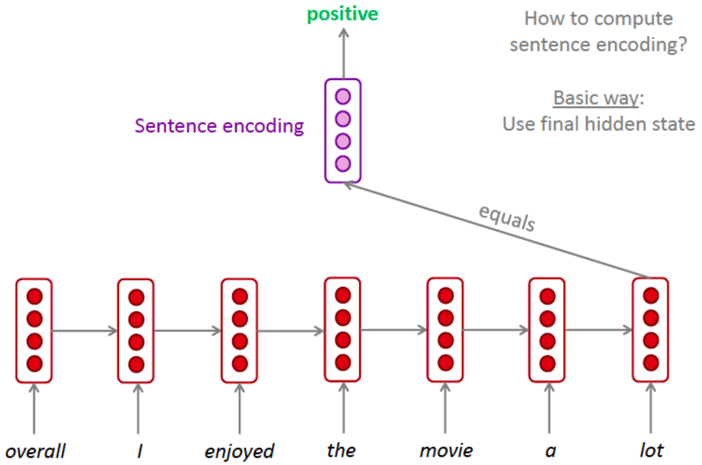
RNNs can be used for tagging

e.g., part-of-speech tagging, named entity recognition



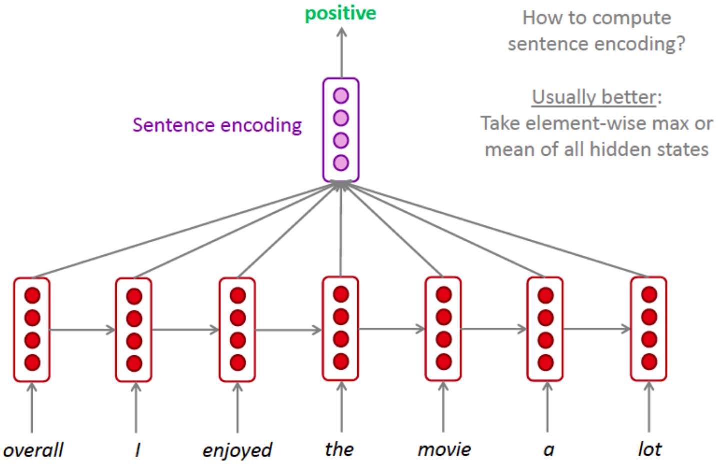
RNNs can be used for sentence classification

e.g., sentiment classification



RNNs can be used for sentence classification

e.g., sentiment classification



Fancy RNNs: A note on terminology

RNN described = "vanilla RNN"

<u>Next</u> You will learn about other RNN flavors

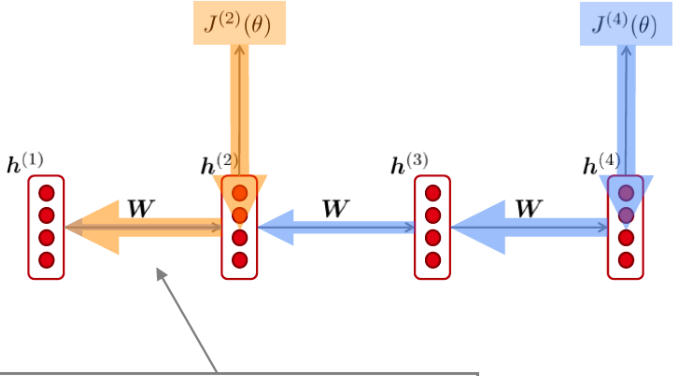
like GRU  and LSTM  and multi-layer RNNs 

<u>By the end</u> You will understand phrases like

stacked bidirectional LSTM with residual connections

51

Vanishing Gradient Problem with RNNs



Gradient signal from faraway is lost because it's much smaller than gradient signal from close-by.

So model weights are only updated only with respect to near effects, not long-term effects.

Effect of vanishing gradient on RNN LM

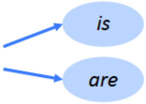
- **LM task:** *When she tried to print her tickets, she found that the printer was out of toner. She went to the stationery store to buy more toner. It was very overpriced. After installing the toner into the printer, she finally printed her* ________

- To learn from this training example, the RNN-LM needs to model the dependency between *"tickets"* on the 7th step and the target word *"tickets"* at the end.

- But if gradient is small, the model can't learn this dependency
  - So the model is unable to predict similar long-distance dependencies at test time

Effect of vanishing gradient on RNN LM

- **LM task:** *The writer of the books* ___



- **Correct answer**: *The writer of the books is planning a sequel*
- **Syntactic recency:** *The writer of the books is* (correct)
- **Sequential recency:** *The writer of the books are* (incorrect)
- Due to vanishing gradient, RNN-LMs are better at learning from sequential recency than syntactic recency, so they make this type of error more often than we'd like [Linzen et al 2016]

How to fix vanishing gradient problem?

- The main problem is that it is too difficult for the RNN to learn to preserve information over many timesteps.
- In a vanilla RNN, the hidden state is constantly being rewritten

$$h^{(t)} = \sigma(W_h h^{(t-1)} + W_x x^{(t)} + b)$$

- How about an RNN with separate memory?

Long Short Term Memory (LSTM)

- On step t, there is a hidden state $h^{(t)}$ and a cell state $c^{(t)}$
  - Both are vectors of length n
  - The cell stores long-term information
  - The LSTM can erase, write and read information from the cell
- The selection of which information is erased/written/read is controlled by three corresponding gates
  - the gates are also vectors of length n
  - On each timestep, each element of the gates can be open (1), close (0) or somehwere in-between.
  - The gates are dynamic: their value is computed based on the current context.

We have a sequence of inputs $\boldsymbol{x}^{(t)}$, and we will compute a sequence of hidden states $\boldsymbol{h}^{(t)}$ and cell states $\boldsymbol{c}^{(t)}$. On timestep *t*:

Sigmoid function: all gate values are between 0 and 1

Forget gate: controls what is kept vs forgotten, from previous cell state

Input gate: controls what parts of the new cell content are written to cell

Output gate: controls what parts of cell are output to hidden state

$$\boldsymbol{f}^{(t)} = \boxed{\sigma}\left(\boldsymbol{W}_f \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_f \boldsymbol{x}^{(t)} + \boldsymbol{b}_f\right)$$

$$\boldsymbol{i}^{(t)} = \boxed{\sigma}\left(\boldsymbol{W}_i \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_i \boldsymbol{x}^{(t)} + \boldsymbol{b}_i\right)$$

$$\boldsymbol{o}^{(t)} = \boxed{\sigma}\left(\boldsymbol{W}_o \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_o \boldsymbol{x}^{(t)} + \boldsymbol{b}_o\right)$$

New cell content: this is the new content to be written to the cell

Cell state: erase ("forget") some content from last cell state, and write ("input") some new cell content

Hidden state: read ("output") some content from the cell

$$\tilde{\boldsymbol{c}}^{(t)} = \tanh\left(\boldsymbol{W}_c \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_c \boldsymbol{x}^{(t)} + \boldsymbol{b}_c\right)$$

$$\boldsymbol{c}^{(t)} = \boldsymbol{f}^{(t)} \circ \boldsymbol{c}^{(t-1)} + \boldsymbol{i}^{(t)} \circ \tilde{\boldsymbol{c}}^{(t)}$$

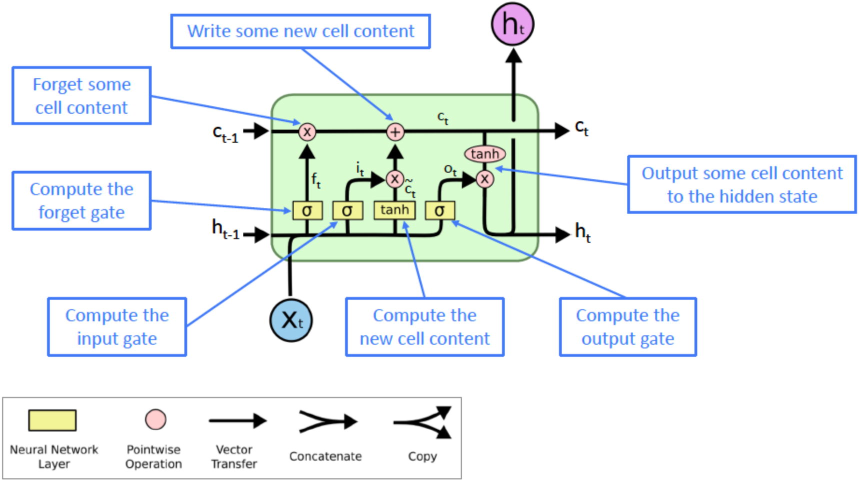
$$\boldsymbol{h}^{(t)} = \boldsymbol{o}^{(t)} \circ \tanh \boldsymbol{c}^{(t)}$$

All these are vectors of same length *n*

Gates are applied using element-wise product

Long Short Term Memory (LSTM)

You can think of the LSTM equations visually like this:



How does LSTM solve vanishing gradients?

The LSTM architecture makes it easier for the RNN to preserve information over many timesteps

- e.g., if the forget gate is set to remember everything on every timestep, then the info in the cell is preserved indefinitely
- By contrast, it is harder for vanilla RNN to learn a recurrent weight matrix W_h that preserves info in hidden state

Gated Recurrent Units (GRU)

- Proposed by Cho et al. in 2014 as a simpler alternative to the LSTM.
- On each timestep t we have input $\boldsymbol{x}^{(t)}$ and hidden state $\boldsymbol{h}^{(t)}$ (no cell state).

Update gate: controls what parts of hidden state are updated vs preserved

$$\boldsymbol{u}^{(t)} = \sigma\left(\boldsymbol{W}_u \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_u \boldsymbol{x}^{(t)} + \boldsymbol{b}_u\right)$$

Reset gate: controls what parts of previous hidden state are used to compute new content

$$\boldsymbol{r}^{(t)} = \sigma\left(\boldsymbol{W}_r \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_r \boldsymbol{x}^{(t)} + \boldsymbol{b}_r\right)$$

New hidden state content: reset gate selects useful parts of prev hidden state. Use this and current input to compute new hidden content.

$$\tilde{\boldsymbol{h}}^{(t)} = \tanh\left(\boldsymbol{W}_h(\boldsymbol{r}^{(t)} \circ \boldsymbol{h}^{(t-1)}) + \boldsymbol{U}_h \boldsymbol{x}^{(t)} + \boldsymbol{b}_h\right)$$

Hidden state: update gate simultaneously controls what is kept from previous hidden state, and what is updated to new hidden state content

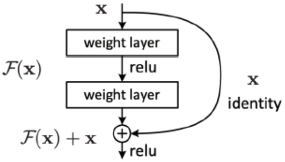
$$\boldsymbol{h}^{(t)} = (1 - \boldsymbol{u}^{(t)}) \circ \boldsymbol{h}^{(t-1)} + \boldsymbol{u}^{(t)} \circ \tilde{\boldsymbol{h}}^{(t)}$$

How does this solve vanishing gradient?
Like LSTM, GRU makes it easier to retain info long-term (e.g. by setting update gate to 0)

Residual connections for vanishing gradient problem

In general, vanishing gradient is a problem for all neural architectures

- Due to chain rule / choice of nonlinearity function, gradient can become vanishingly small as it backpropagates
- Thus lower layers are learnt very slowly
- Lot of new architectures add more direct (residual) connections, allowing the gradients to flow



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

Bidirectional RNNs

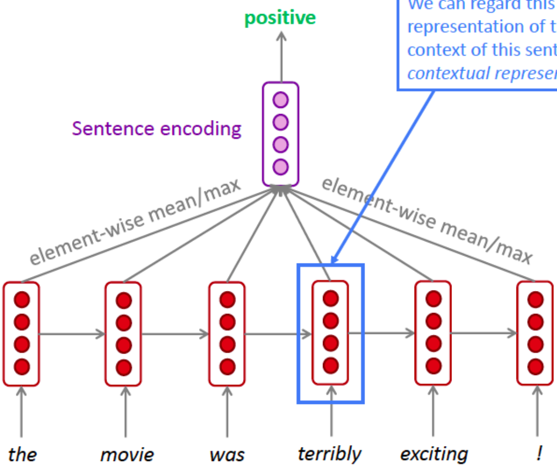
What we have seen till now?

- 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., sentiment analysis.

Bidirectional RNNs: motivation for sentiment classification

Task: Sentiment Classification



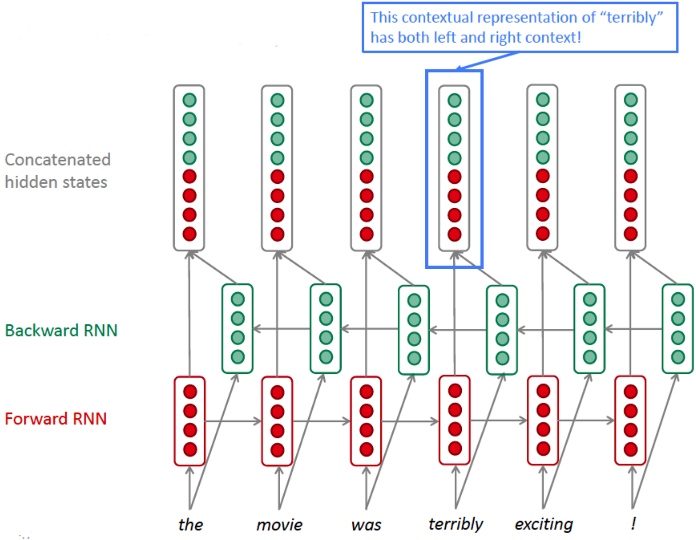
We can regard this hidden state as a representation of the word *"terribly"* in the context of this sentence. We call this a *contextual representation.*

These contextual representations only contain information about the *left* context (e.g. *"the movie was"*).

What about *right* context?

In this example, *"exciting"* is in the right context and this modifies the meaning of *"terribly"* (from negative to positive)

Bidirectional RNNs



Bidirectional RNNs

On timestep *t*:

This is a general notation to mean "compute one forward step of the RNN" – it could be a vanilla, LSTM or GRU computation.

Forward RNN $\overrightarrow{\boldsymbol{h}}^{(t)} = \boxed{\mathrm{RNN}_{\mathrm{FW}}}(\overrightarrow{\boldsymbol{h}}^{(t-1)}, \boldsymbol{x}^{(t)})$

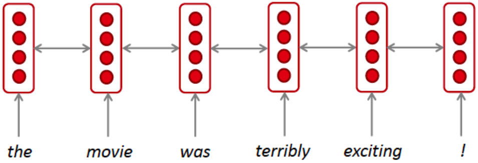
Backward RNN $\overleftarrow{\boldsymbol{h}}^{(t)} = \mathrm{RNN}_{\mathrm{BW}}(\overleftarrow{\boldsymbol{h}}^{(t+1)}, \boldsymbol{x}^{(t)})$

Generally, these two RNNs have separate weights

Concatenated hidden states $\boxed{\boldsymbol{h}^{(t)}} = [\overrightarrow{\boldsymbol{h}}^{(t)}; \overleftarrow{\boldsymbol{h}}^{(t)}]$

We regard this as "the hidden state" of a bidirectional RNN. This is what we pass on to the next parts of the network.

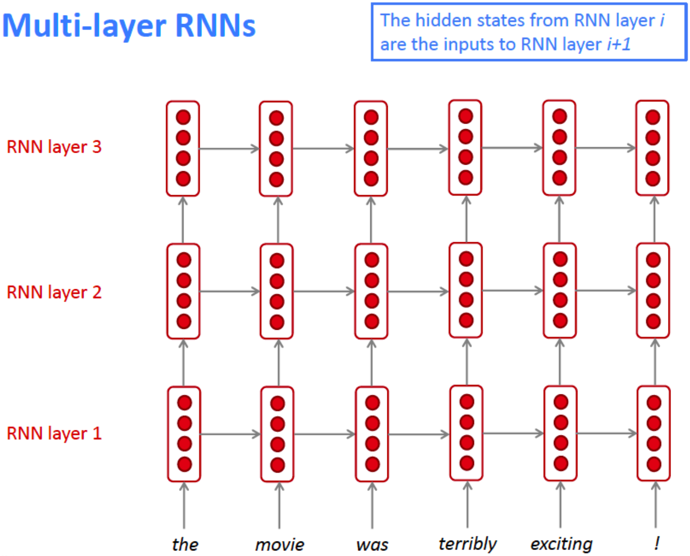
Bidirectional RNNs: simplified diagram



The two-way arrows indicate bidirectionality and the depicted hidden states are assumed to be the concatenated forwards+backwards states.

Bi-RNNs are only applicable if you have access to the entire input sequence

- thus not applicable to Language Modeling where only the left context is available


Multi-layer RNNs
The hidden states from RNN layer i are the inputs to RNN layer i+1
RNN layer 3
RNN layer 2
RNN layer 1
the
movie
was
terribly
exciting
!

Summary

- RNNs (LSTMs) are architectures for sequence processing, many applications not only in Natural Language Processing, but also Speech, and time series data.
- Bottleneck issues: A single hidden state captures lot of information, so many advanced models make use of attention mechanism.
- Many of these slides have been adapted from CS 224n. Other material is from the Deep Learning book.