CS447: Natural Language Processing

http://courses.engr.illinois.edu/cs447

Lecture 11: Introduction to RNNs

Julia Hockenmaier

juliahmr@illinois.edu

3324 Siebel Center

Part 1: Recurrent Neural Nets for various NLP tasks

Today's lecture

Part 1: Recurrent Neural Nets for various NLP tasks

Part 2: Practicalities:
Training RNNs
Generating with RNNs
Using RNNs in complex networks

Part 3: Changing the recurrent architecture
to go beyond vanilla RNNs:
LSTMs, GRUs

Recurrent Neural Nets (RNNs)

Feedforward nets can only handle inputs and outputs that have a **fixed size**.

Recurrent Neural Nets (RNNs) handle **variable length sequences** (as **input** and as **output**)

There are 3 main variants of RNNs, which differ in their internal structure:

Basic RNNs (Elman nets),
Long Short-Term Memory cells (LSTMs)
Gated Recurrent Units (GRUs)

RNNs in NLP

RNNS are used for…

… language modeling and generation, including…

… auto-completion and…

… machine translation

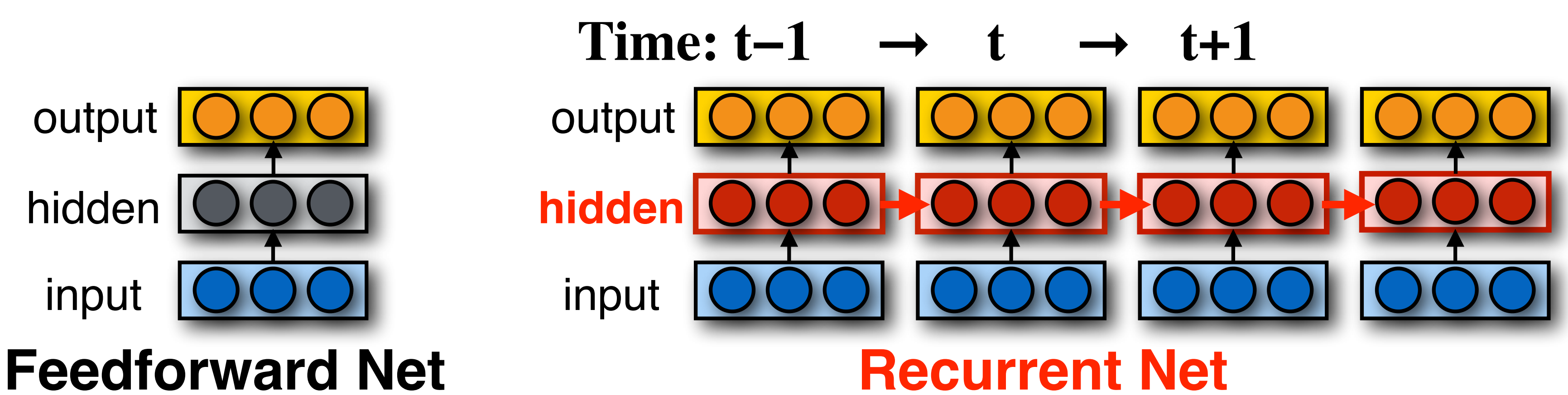
… sequence classification (e.g. sentiment analysis)

… sequence labeling (e.g. POS tagging)

Recurrent neural networks (RNNs)

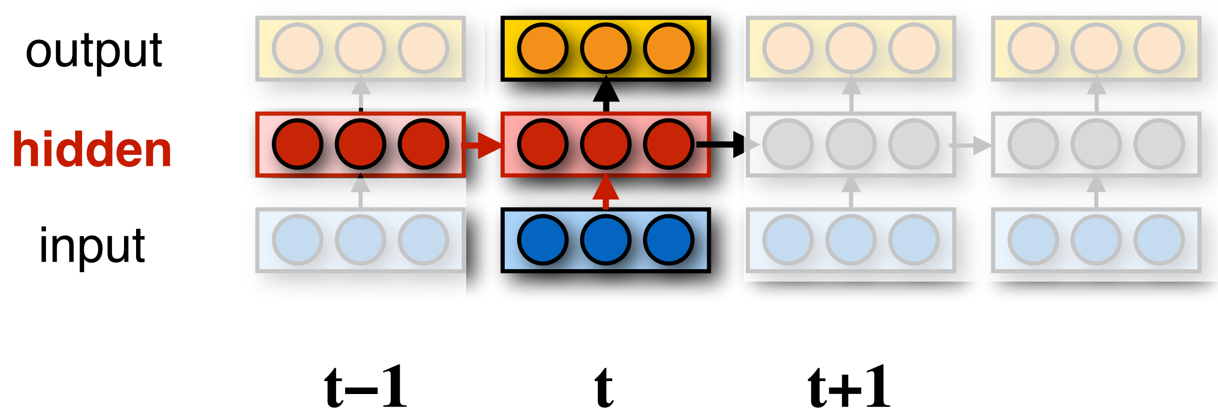
Basic RNN: Generate a **sequence of *T* outputs** by running a variant of a feedforward net T times.

Recurrence:
The **hidden state** computed at the **previous step** ($\mathbf{h}^{(t-1)}$)
is fed into **the hidden state** at the **current step** ($\mathbf{h}^{(t)}$)
With H hidden units, this requires additional H^2 parameters



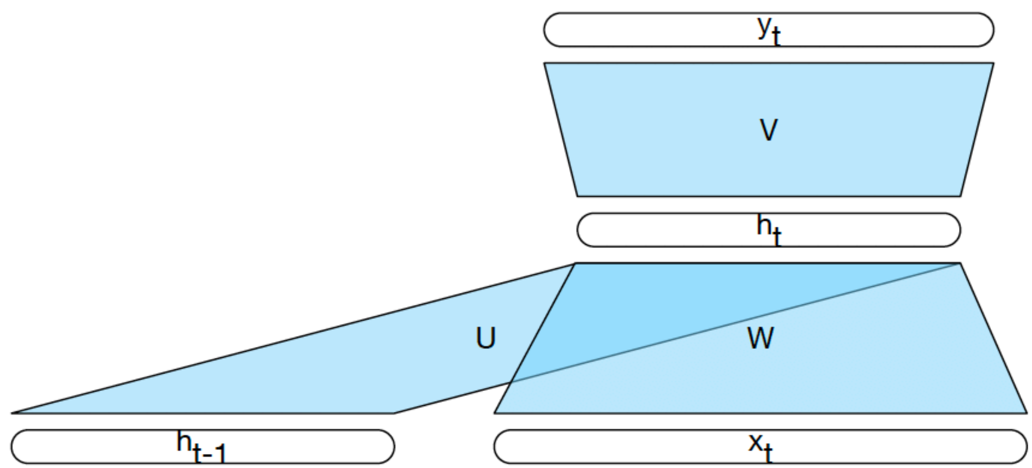
Basic RNNs

Each time step corresponds to a feedforward net where the hidden layer gets its input not just from the layer below but also from the activations of the hidden layer at the previous time step



Basic RNNs

Each time step t corresponds to a **feedforward net** whose **hidden layer h**$^{(t)}$ gets input from the **layer below** (**x**$^{(t)}$) and from the output of the **hidden layer at the previous time step h**$^{(t-1)}$

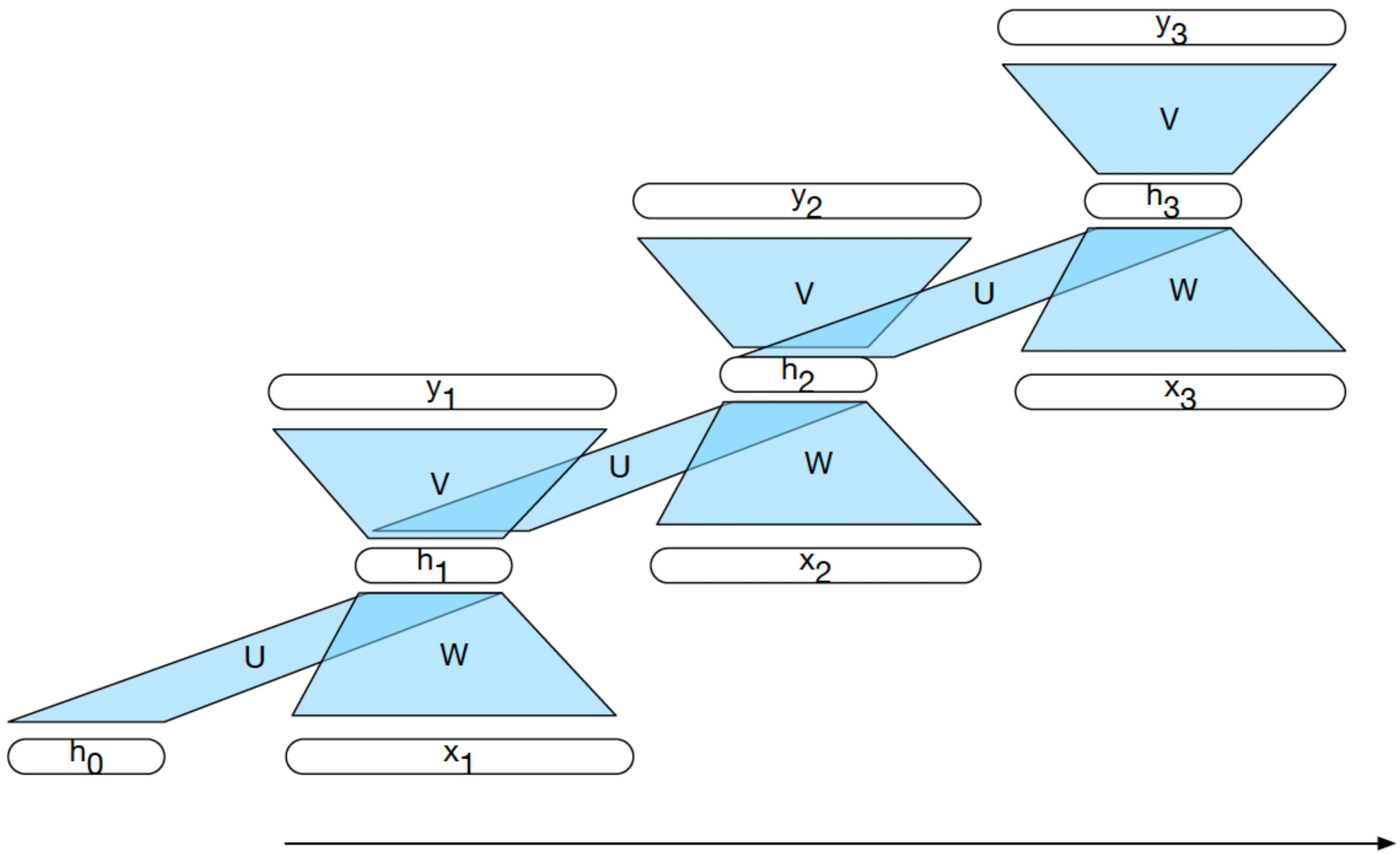


Computing the **vector of hidden states** at time *t*

$$\mathbf{h}^{(t)} = g(\mathbf{U}\mathbf{h}^{(t-1)} + \mathbf{W}\mathbf{x}^{(t)})$$

The ***i*-th element** of **h**$_t$: $h_i^{(t)} = g\left(\sum_j U_{ji} h_j^{(t-1)} + \sum_k W_{ki} x_k^{(t)}\right)$

A basic RNN unrolled in time



RNNs for language modeling

If our vocabulary consists of V words, the output layer (at each time step) has V units, one for each word.

The softmax gives a distribution over the V words for the next word.

To compute the **probability of string** $w^{(0)}w^{(1)}\ldots w^{(n)}w^{(n+1)}$ (where $w^{(0)}$ = <s>, and $w^{(n+1)}$ = <\s>), feed in $w^{(i)}$ as input at time step i and compute

$$\prod_{i=1}^{n+1} P\left(w^{(i)} \mid w^{(0)} \ldots w^{(i-1)}\right)$$

RNNs for language generation

To **generate** $w^{(0)}w^{(1)}\ldots w^{(n)}w^{(n+1)}$
(where $w^{(0)}$ = <s>, and $w^{(n+1)}$ = <\s>)…

…Give $w^{(0)}$ as first input, and

… Choose the next word according to the probability

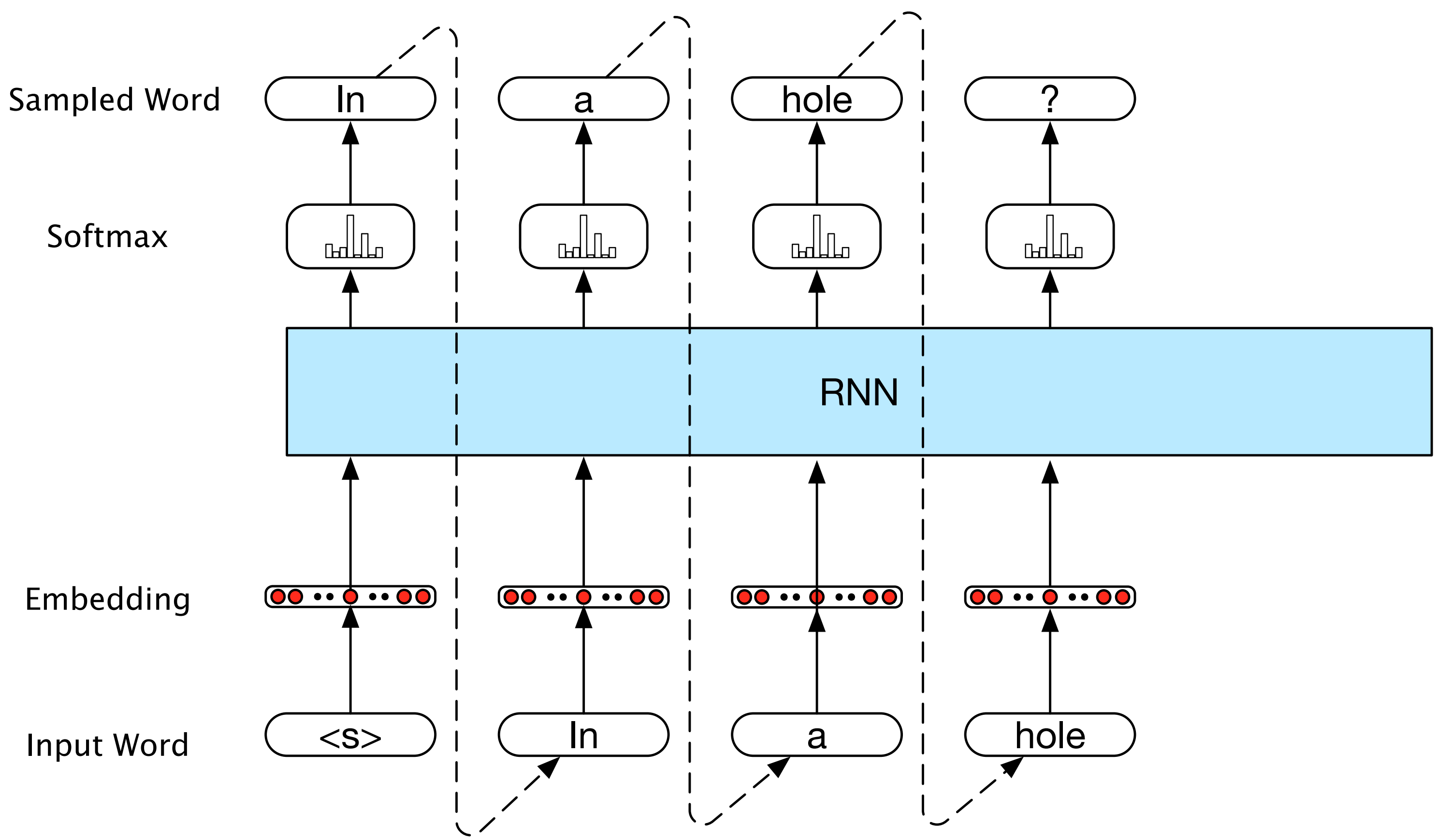
$$P(w^{(i)} \mid w^{(0)}\ldots w^{(i-1)})$$

…Feed the predicted word $w^{(i)}$ in as input
at the next time step.

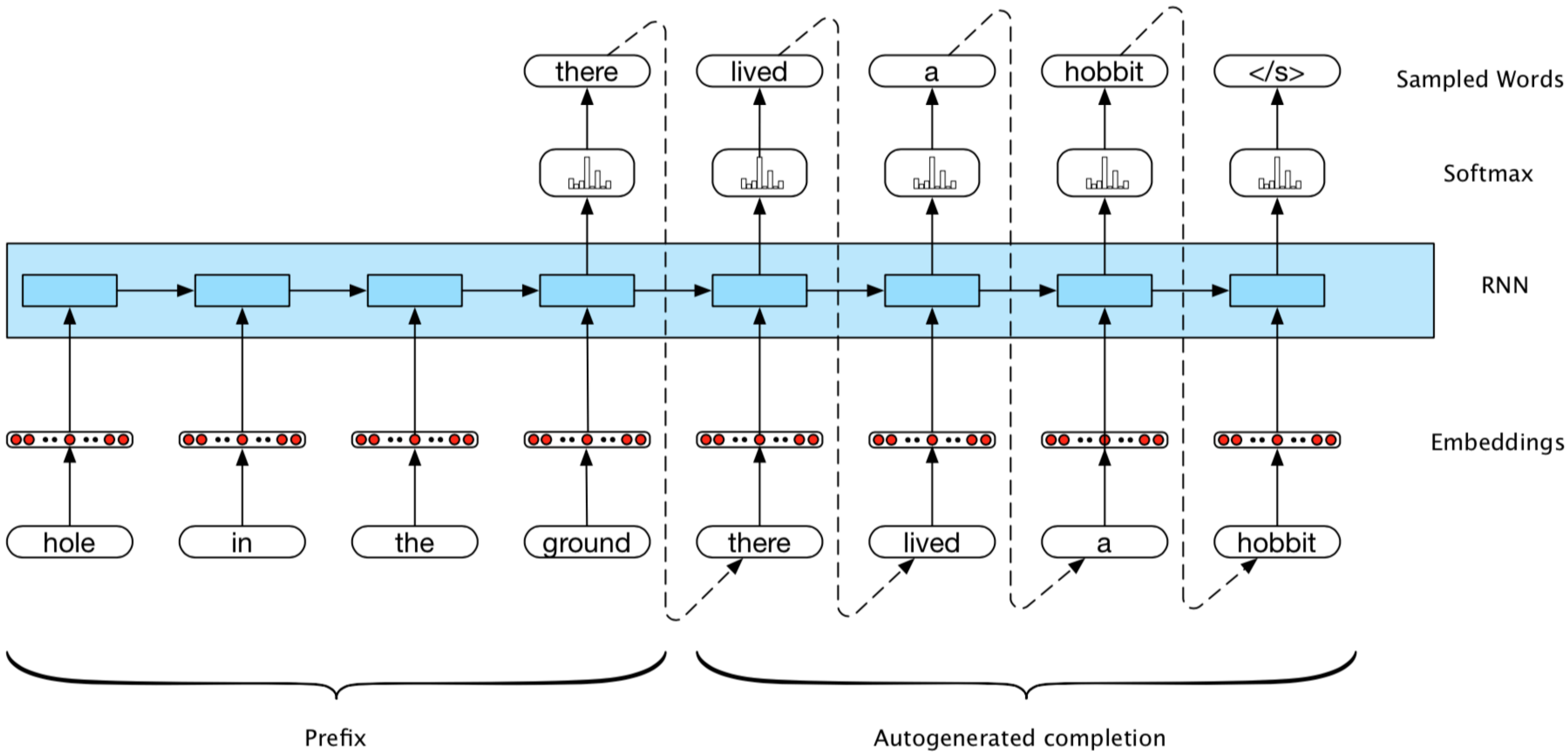
… Repeat until you generate <\s>

RNNs for language generation

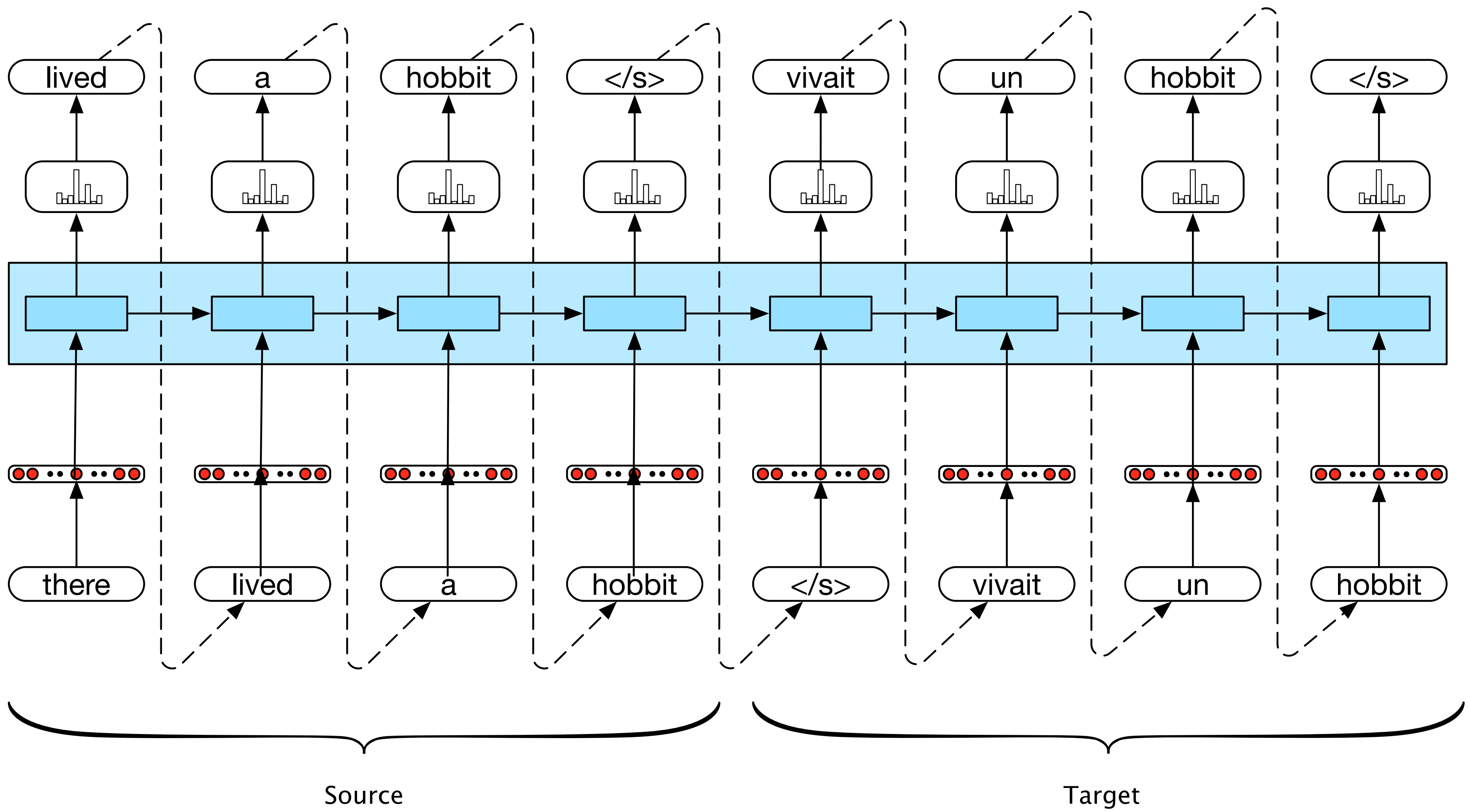
AKA "autoregressive generation"



RNN for Autocompletion



An RNN for Machine Translation



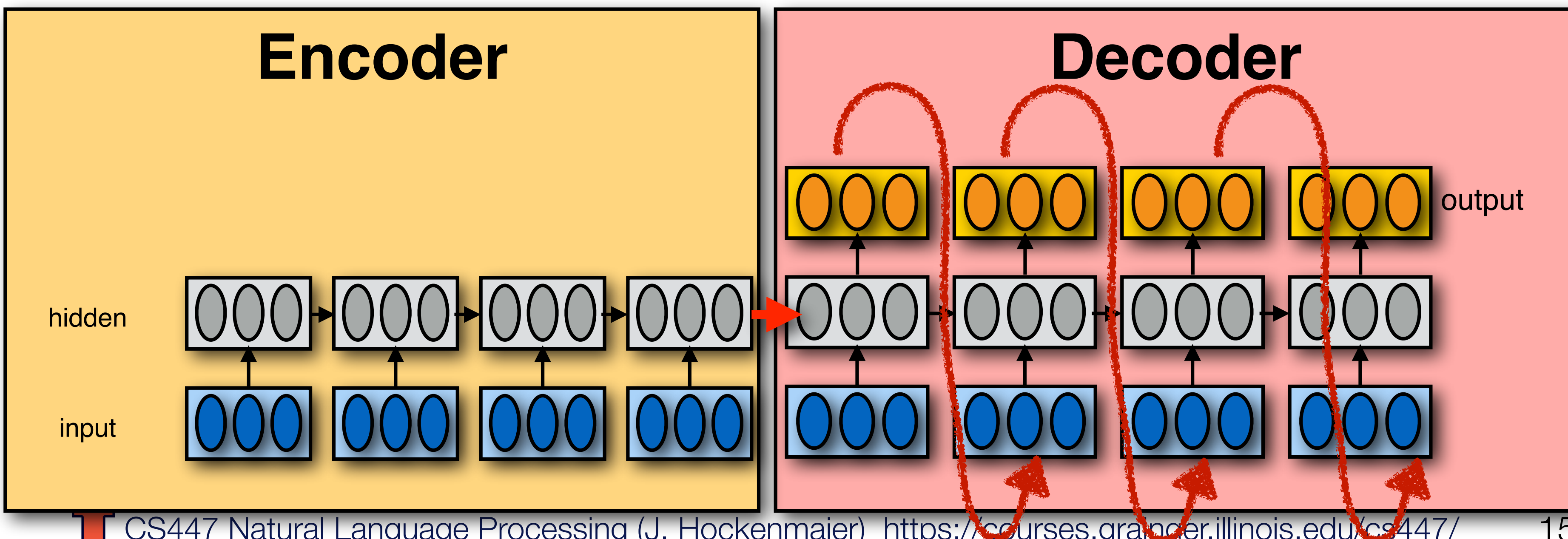
Encoder-Decoder (seq2seq) model

Task: Read an input sequence
and return an output sequence

- Machine translation: translate source into target language
- Dialog system/chatbot: generate a response

Reading the input sequence: **RNN Encoder**

Generating the output sequence: **RNN Decoder**



Encoder-Decoder (seq2seq) model

Encoder RNN:

reads in the input sequence

passes its last hidden state to the initial hidden state of the decoder

Decoder RNN:

generates the output sequence

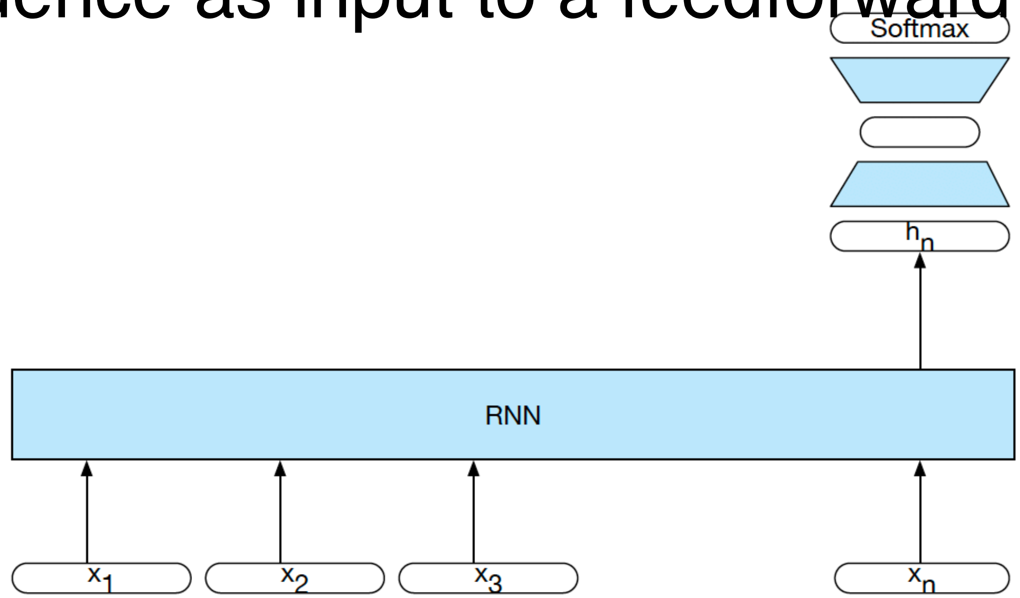
typically uses different parameters from the encoder

may also use different input embeddings

RNNs for sequence classification

If we just want to assign **one label** to the entire sequence, we don't need to produce output at each time step, so we can use a simpler architecture.

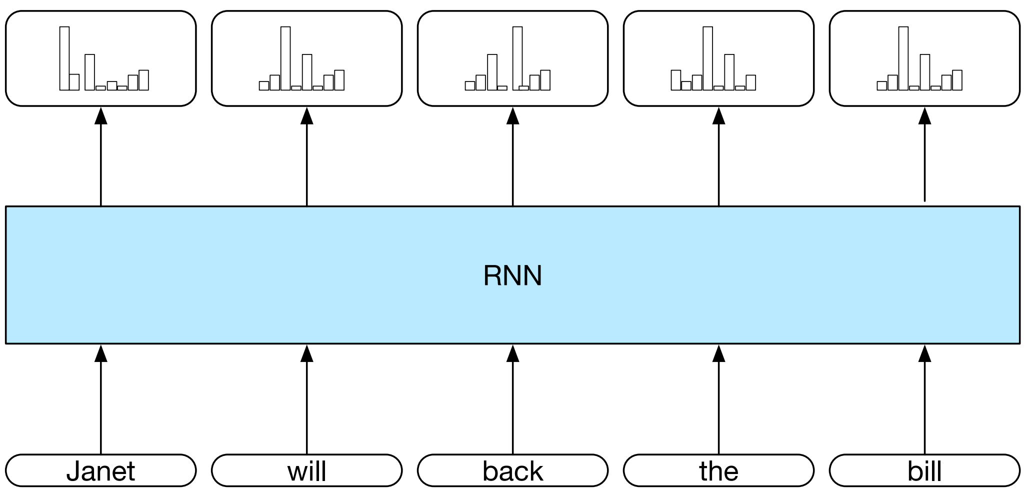
We can use the hidden state of the last word in the sequence as input to a feedforward net:



Basic RNNs for sequence labeling

Sequence labeling (e.g. POS tagging):
Assign **one label to each element** in the sequence.

RNN Architecture:
Each time step has a distribution over output classes



Extension: add a CRF layer to capture dependencies among labels of adjacent tokens.

RNNs for sequence labeling

In sequence labeling, we want to assign a label or tag $t^{(i)}$ to each word $w^{(i)}$

Now the output layer gives a (softmax) distribution over the T possible tags,
and the hidden layer contains information about the previous words and the previous tags.

To compute the probability of a tag sequence $t^{(1)}\ldots t^{(n)}$ for a given string $w^{(1)}\ldots w^{(n)}$, feed in $w^{(i)}$ (and possibly $t^{(i-1)}$) as input at time step i and compute $P(t^{(i)} \mid w^{(1)}\ldots w^{(i-1)}, t^{(1)}\ldots t^{(i-1)})$

Part 2: Recurrent Neural Net Practicalities

RNN Practicalities

This part will discuss how to train and use RNNs.

We will also discuss how to go beyond basic RNNs.

The last part used a simple RNN with one layer to illustrate how RNNs can be used for different NLP tasks.

In practice, more complex architectures are common.

Three complementary ways to extend basic RNNs:

— Using RNNs in **more complex networks** (bidirectional RNNs, stacked RNNs) [This Part]

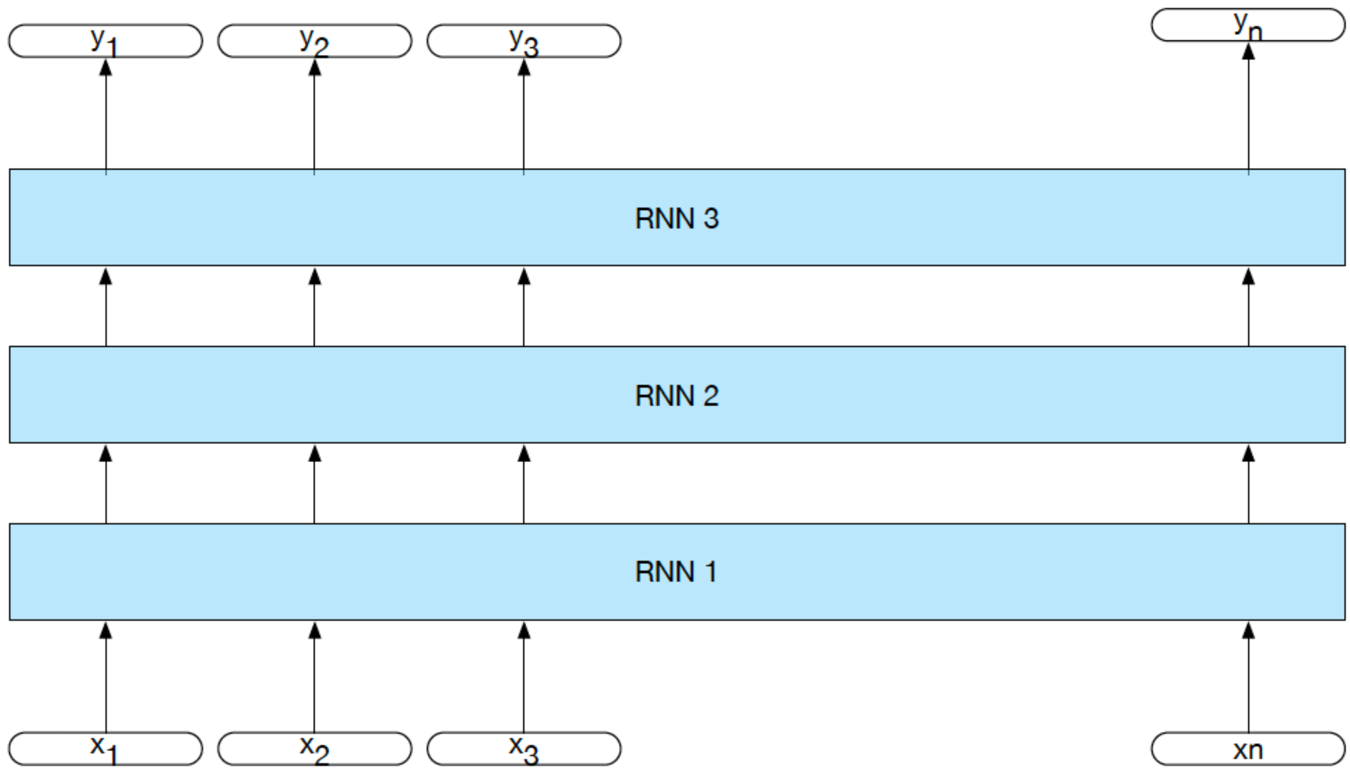
— Modifying the **recurrent** architecture (LSTMs, GRUs) [Part 3]

— Adding **attention** mechanisms [Next Lecture]

Using RNNs in more complex architectures

Stacked RNNs

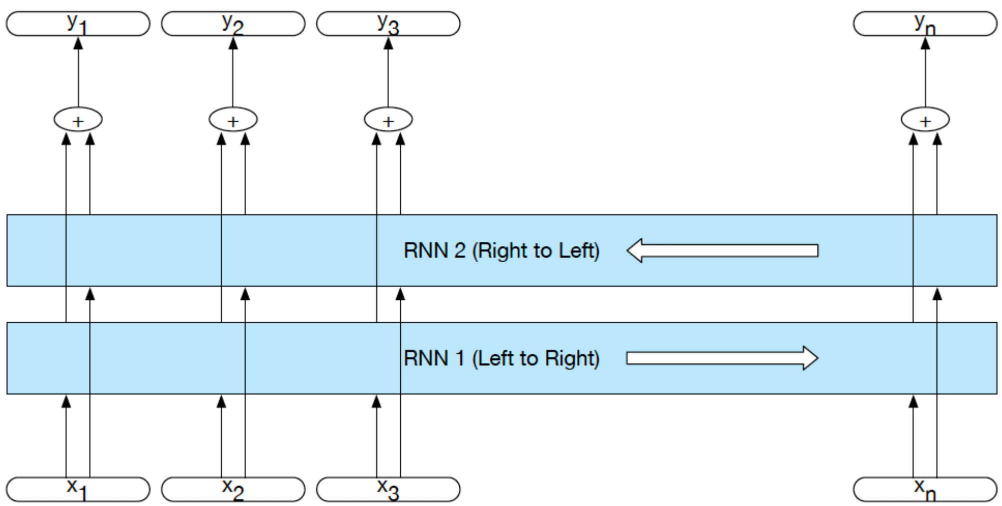
We can create an RNN that has **“vertical” depth** (at each time step) by stacking multiple RNNs:



Bidirectional RNNs

Unless we need to generate a sequence, we can run
***two* RNNs over the input sequence**,
one in the **forward** direction, and one in the **backward** direction.

Their hidden states will capture **different context** information



To obtain a **single hidden state** at time t: $\mathbf{h}_{\mathbf{bi}}^{(\mathbf{t})} = \mathbf{h}_{\mathbf{fw}}^{(\mathbf{t})} \oplus \mathbf{h}_{\mathbf{bw}}^{(\mathbf{t})}$

where $\oplus$ is typically concatenation

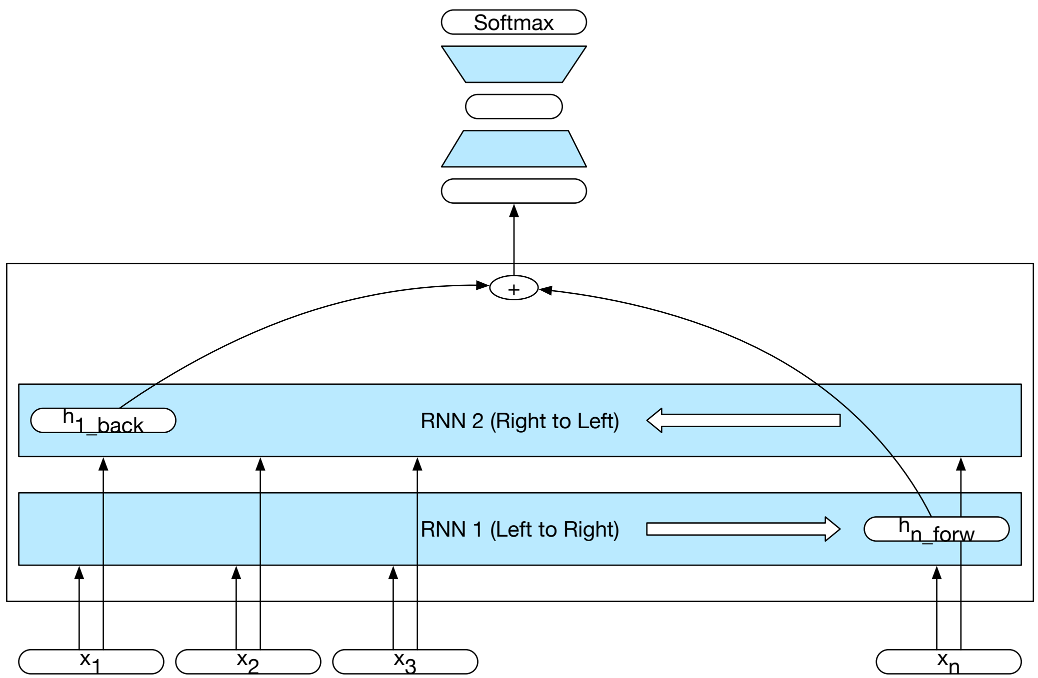
Bidirectional RNNs for sequence classification

Combine…

…the forward RNN's hidden state for the last word, and

…the backward RNN's hidden state for the first word

into a single vector



Training and Generating Sequences with RNNs

How to generate with an RNN

Greedy decoding:

Always pick the word with the highest probability

(if you start from <s>, this only generates a single sentence)

Sampling:

Sample a word according to the given distribution

Beam search decoding:

Keep a number of hypotheses after each time step

— **Fixed-width beam**: keep the top *k* hypotheses

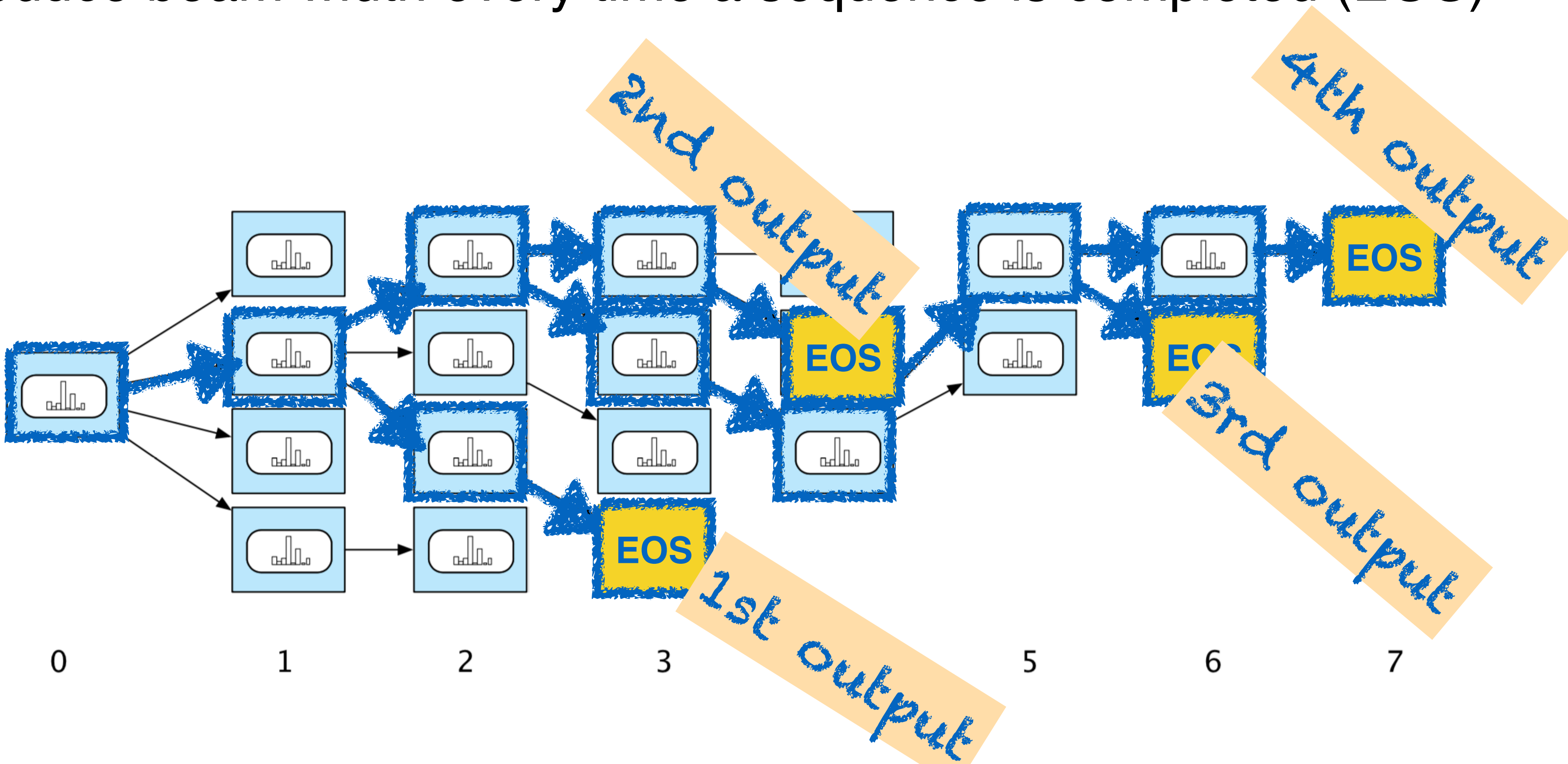
— **Variable-width beam**: keep all hypotheses whose score is with a certain factor of the best score

Beam Decoding (fixed width k=4)

Keep the *k* best options around at each time step.

Operate breadth-first: keep the *k* best next hypotheses among the best continuations for each of the current *k hypotheses*.

Reduce beam width every time a sequence is completed (EOS)



Training RNNs for generation

Maximum likelihood estimation (MLE):

Given training samples $w^{(1)}w^{(2)}\ldots w^{(T)}$, find the parameters $\theta*$ that assign **the largest probability to these training samples:**

$$\theta* = \text{argmax}_\theta P_\theta(w^{(1)}w^{(2)}\ldots w^{(T)}) = \text{argmax}_\theta \prod_{t=1..T} P_\theta(w^{(t)} | w^{(1)}\ldots w^{(t-1)})$$

Since $P_\theta(w^{(1)}w^{(2)}\ldots w^{(T)})$ is factored into $P_\theta(w^{(t)} | w^{(1)}\ldots w^{(t-1)})$, we can train models to assign a higher probability to the word $w^{(t)}$ that occurs in the training data after $w^{(1)}\ldots w^{(t-1)}$ than any other word $w_i \in V$:

$$\forall_{i=1\ldots|V|} P_\theta\left(w^{(t)} \mid w^{(1)}\ldots w^{(t-1)}\right) \geq P_\theta\left(w_i \mid w^{(1)}\ldots w^{(t-1)}\right)$$

This is also called **teacher forcing**.

Teacher forcing

Each training sequence $w^{(1)}w^{(2)}\ldots w^{(T)}$ turns into T training items:

Give $w^{(1)}w^{(2)}\ldots w^{(t-1)}$ as input to the RNN, and train it to maximize the probability of $w^{(t)}$

(as you would in standard classification,
or when training an n-gram language model).

Problems with teacher forcing

Exposure bias:

When we *train* an RNN for sequence generation, the prefix $y^{(1)} \ldots y^{(t-1)}$ that we condition on comes from the original data

When we *use* an RNN for sequence generation, the prefix $y^{(1)} \ldots y^{(t-1)}$ that we condition on is also generated by the RNN,

— The model is run on data that may look quite different from the data it was trained on.

— The model is not trained to predict the best next token within a generated sequence, or to predict the best sequence

— Errors at earlier time-steps propagate through the sequence.

Remedies

Minimum risk training:

(Shen et al. 2016, https://www.aclweb.org/anthology/P16-1159.pdf)

— define a loss function (e.g. negative BLEU) to compare generated sequences against gold sequences

—Minimize risk (expected loss on training data) such that candidates outputs with a smaller loss (higher BLEU score) have higher probability.

Reinforcement learning-based approaches:

(Ranzato et al. 2016 https://arxiv.org/pdf/1511.06732.pdf)

— use BLEU as a reward (i.e. like MRT)

— perhaps pre-train model first with standard teacher forcing.

GAN-based approaches ("professor forcing")

(Goyal et al. 2016, http://papers.nips.cc/paper/6099-professor-forcing-a-new-algorithm-for-training-recurrent-networks.pdf)

— combine standard RNN with an adversarial model that aims to distinguish original from generated sequences

Part 3: RNN Variants

RNN variants: LSTMs, GRUs

Long Short-Term Memory networks (**LSTMs**)
are RNNs with a more complex recurrent architecture

Gated Recurrent Units (GRUs)
are a simplification of LSTMs

Both contain "**Gates**" to control how much of the input or previous hidden state to forget or remember

From RNNs to LSTMs

In **Vanilla (Elman) RNNs**, the current hidden state $\mathbf{h}^{(t)}$ is a nonlinear function of the previous hidden state $\mathbf{h}^{(t-1)}$ and the current input $\mathbf{x}^{(t)}$:

$$\mathbf{h}^{(t)} = g\left(\mathbf{U}\mathbf{h}^{(t-1)} + \mathbf{W}\mathbf{x}^{(t)} + b_h\right)$$

With g=**tanh** (the original definition):
⇒ Models suffer from the *vanishing gradient* problem: they can't be trained effectively on long sequences.

With g=**ReLU**
⇒ Models suffer from the *exploding gradient* problem: they can't be trained effectively on long sequences.

From RNNs to LSTMs

LSTMs (Long Short-Term Memory networks)
were introduced to overcome the vanishing gradient problem.
Hochreiter and Schmidhuber, Neural Computation 9(8), 1997
https://www.bioinf.jku.at/publications/older/2604.pdf

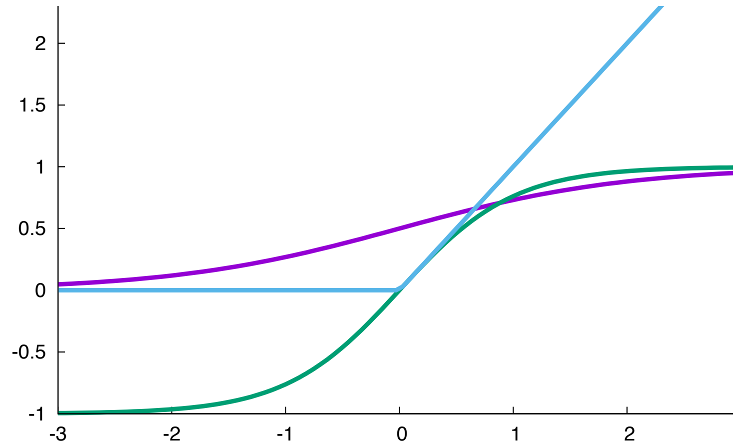
Like RNNs, LSTMs contain a **hidden state** that gets passed through the network and updated at each time step

LSTMs contain an additional **cell state** that also gets passed through the network and updated at each time step

LSTMs contain three different **gates** (input/forget/output) that read in the previous hidden state and current input to decide how much of the past hidden and cell states to keep.

These gates mitigate the vanishing/exploding gradient problem

Recap: Activation functions



Hyperbolic Tangent: $\tanh(x) = \dfrac{\exp(2x) - 1}{\exp(2x) + 1} \in [-1, +1]$

Rectified Linear Unit: $\mathrm{ReLU(x) = max(0, x)} \in [0, +\infty]$

Sigmoid (logistic function): $\sigma(x) = \dfrac{1}{1 + \exp(-x)} \in [0,1]$

RNN variants: LSTMs, GRUs

Long Short-Term Memory networks (**LSTMs**) are RNNs with a more complex recurrent architecture

Gated Recurrent Units (GRUs) are a simplification of LSTMs

Both contain "**Gates**" to control how much of the input or past hidden state to forget or remember

A **gate** performs **element-wise multiplication** of

a) a *d*-dimensional **sigmoid** layer ***g*** (all elements between 0 and 1), and

b) a *d*-dimensional **input** vector ***u***

Result: *d*-dimensional **output** vector ***v*** which is like the input ***u***, but **elements** where $g_i \approx 0$ are (partially) **"forgotten"**

Gating mechanisms

Gates are trainable layers with a **sigmoid** activation function
often determined by the current input $\mathbf{x}^{(t)}$ and the (last) hidden state $\mathbf{h}^{(t-1)}$ eg.:

$$\mathbf{g}_k^{(t)} = \sigma(\mathbf{W}_k \mathbf{x}^{(t)} + \mathbf{U}_k \mathbf{h}^{(t-1)} + b_k)$$

$\mathbf{g}$ is a vector of (Bernoulli) probabilities ($\forall i : 0 \leq g_i \leq 1$)

Unlike traditional (0,1) gates, neural gates are differentiable (we can train them)

$\mathbf{g}$ is combined with another vector $\mathbf{u}$ (of the same dimensionality)
by **element-wise multiplication** (Hadamard product): $\mathbf{v} = \mathbf{g} \otimes \mathbf{u}$

If $g_i \approx 0$, $v_i \approx 0$, and if $g_i \approx 1$, $v_i \approx u_i$

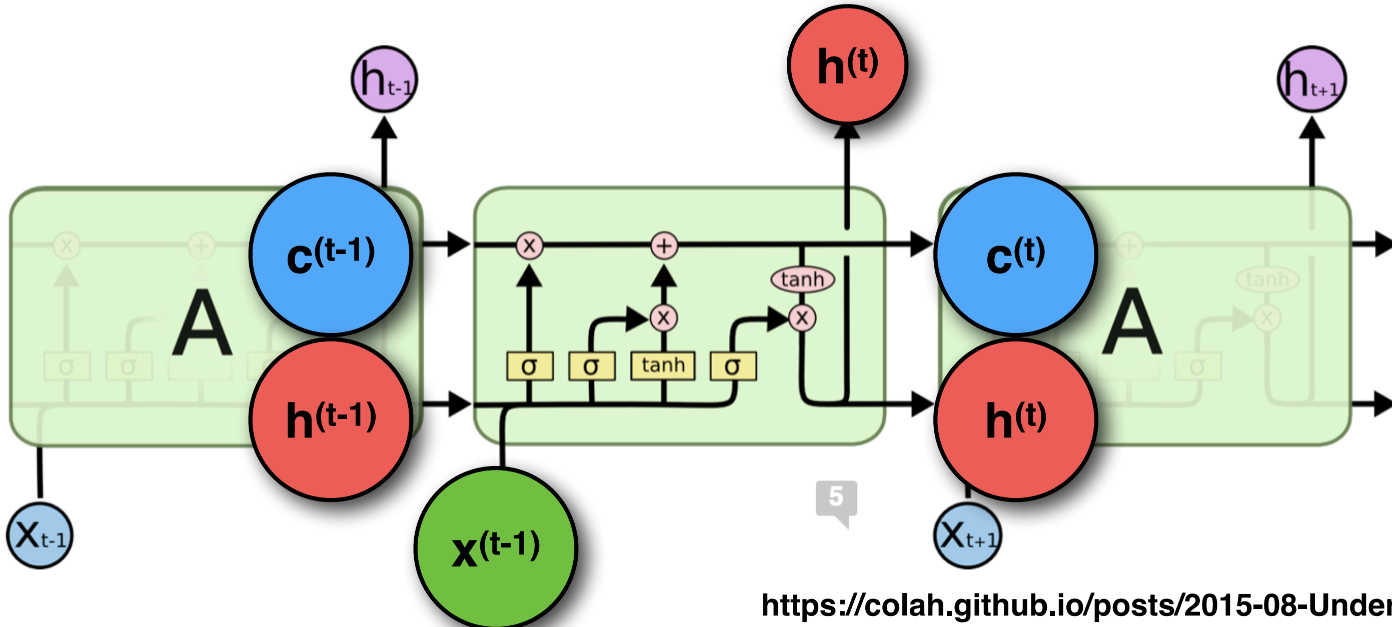
Each g_i has its own set of trainable parameters to determine how much of u_i to keep

Gates can also be used to form
linear combinations of two input vectors $\mathbf{t}, \mathbf{u}$:

– **Addition of two independent gates**: $\mathbf{v} = \mathbf{g}_1 \otimes \mathbf{t} + \mathbf{g}_2 \otimes \mathbf{u}$

– **Linear interpolation (coupled** gates): $\mathbf{v} = \mathbf{g} \otimes \mathbf{t} + (1 - \mathbf{g}) \otimes \mathbf{u}$

Long Short-Term Memory Networks (LSTMs)



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

At time t, the LSTM cell reads in

— a c-dimensional ***previous*** **cell state vector** $\mathbf{c}^{(t-1)}$

— an h-dimensional ***previous*** **hidden state vector** $\mathbf{h}^{(t-1)}$

— a d-dimensional ***current*** **input vector** $\mathbf{x}^{(t)}$

At time t, the LSTM cell returns

— a c-dimensional ***new*** **cell state vector** $\mathbf{c}^{(t)}$

— an h-dimensional ***new*** **hidden state vector** $\mathbf{h}^{(t)}$

(which may also be passed to an **output layer**)

LSTM operations

Based on the previous cell state $\mathbf{c}^{(t-1)}$, previous hidden state $\mathbf{h}^{(t-1)}$ and the current input $\mathbf{x}^{(t)}$, the LSTM computes:

… A new **intermediate cell state** $\tilde{\mathbf{c}}^{(t)}$ that depends on $\mathbf{h}^{(t-1)}$ and $\mathbf{x}^{(t)}$**:**

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

… **Three gates** $\mathbf{f}^{(t)}, \mathbf{i}^{(t)}, \mathbf{o}^{(t)}$, which each depend on $\mathbf{h}^{(t-1)}$ and $\mathbf{x}^{(t)}$:

- The **forget gate** $\mathbf{f}^{(t)} = \sigma\left(\mathbf{W}_f\mathbf{x}^{(t)} + \mathbf{U}_f\mathbf{h}^{(t-1)} + b_f\right)$ decides how much of the **last** $\mathbf{c}^{(t-1)}$ to remember in the new cell state: $\mathbf{f}^{(t)} \otimes \mathbf{c}^{(t-1)}$
- The **input gate** $\mathbf{i}^{(t)} = \sigma\left(\mathbf{W}_i\mathbf{x}^{(t)} + \mathbf{U}_i\mathbf{h}^{(t-1)} + b_i\right)$ decides how much of the **intermediate** $\tilde{\mathbf{c}}^{(t)}$ to use in the new cell state: $\mathbf{i}^{(t)} \otimes \tilde{\mathbf{c}}^{(t)}$
- The **output gate** $\mathbf{o}^{(t)} = \sigma\left(\mathbf{W}_o\mathbf{x}^{(t)} + \mathbf{U}_o\mathbf{h}^{(t-1)} + b_o\right)$ decides how much of the **new** $\mathbf{c}^{(t)}$ to use in the next hidden state: $\mathbf{h}^{(t)} = \mathbf{o}^{(t)} \otimes \mathbf{c}^{(t)}$

The new **cell state** $\mathbf{c}^{(t)} = \tanh\left(\mathbf{f}^{(t)} \otimes \mathbf{c}^{(t-1)} + \mathbf{i}^{(t)} \otimes \tilde{\mathbf{c}}^{(t)}\right)$ is a **linear combination of cell states** $\mathbf{c}^{(t-1)}$ and $\tilde{\mathbf{c}}^{(t)}$ that depends on forget gate $\mathbf{f}^{(t)}$ and input gate $\mathbf{i}^{(t)}$

The new **hidden state** $\mathbf{h}^{(t)} = \mathbf{o}^{(t)} \otimes \mathbf{c}^{(t)}$ depends on $\mathbf{c}^{(t)}$ and the output gate $\mathbf{o}^{(t)}$

Gated Recurrent Units (GRUs)

Based on $\mathbf{h}^{(t-1)}$ and $\mathbf{x}^{(t)}$, a GRU computes:

— a **reset gate** $\mathbf{r}^{(t)}$ to determine how much of $\mathbf{h}^{(t-1)}$ to keep in $\tilde{\mathbf{h}}^{(t)}$

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

— an intermediate **hidden state** $\tilde{\mathbf{h}}^{(t)}$ that depends on $\mathbf{x}^{(t)}$ and $\mathbf{r}^{(t)} \otimes \mathbf{h}^{(t-1)}$

$$\tilde{\mathbf{h}}^{(t)} = \phi\left(\mathbf{W}_h\mathbf{x}^{(t)} + \mathbf{U}_h(\mathbf{r}^{(t)} \otimes \mathbf{h}^{(t-1)}) + b_r\right) \quad [\phi = \tanh \text{ or ReLU}]$$

— an **update gate** $\mathbf{z}^{(t)}$ to determine how much of $\mathbf{h}^{(t-1)}$ to keep in $\mathbf{h}^{(t)}$

$$\mathbf{z}^{(t)} = \sigma\left(\mathbf{W}_z\mathbf{x}^{(t)} + \mathbf{U}_z\mathbf{h}^{(t-1)} + b_r\right)$$

— a **new hidden state** $\mathbf{h}^{(t)}$ as a linear interpolation of $\mathbf{h}^{(t-1)}$ and $\tilde{\mathbf{h}}^{(t)}$ with weights determined by the (coupled) update gate $\mathbf{z}^{(t)}$

$$\mathbf{h}^{(t)} = \mathbf{z}^{(t)} \otimes \mathbf{h}^{(t-1)} + \left(\mathbf{1} - \mathbf{z}^{(t)}\right) \otimes \tilde{\mathbf{h}}^{(t)}$$

Cho et al. (2014) Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation
https://arxiv.org/pdf/1406.1078.pdf

LSTMs vs GRUs

LSTMs are more expressive than GRUs
and basic RNNs (they're better at learning long-range dependencies)

But GRUs are easier to train than LSTMs
(useful when training data is limited)