CS546: Machine Learning in NLP (Spring 2020)
http://courses.engr.illinois.edu/cs546/

Lecture 6: RNN wrap-up

Julia Hockenmaier
juliahmr@illinois.edu
3324 Siebel Center
Office hours: Monday, 11am—12:30pm

Today's class: RNN architectures

RNNs are among the workhorses of neural NLP:
— Basic RNNs are rarely used
— LSTMs and GRUs are commonly used.
What's the difference between these variants?

RNN odds and ends:
— Character RNNs
— Attention mechanisms (LSTMs/GRUs)

Character RNNs and BPE

Character RNNs:

— Each input element is one character: ’t’,’h’, ‘e’,…

— Can be used to replace word embeddings,
or to compute embeddings for rare/unknown words

(in languages with an alphabet, like English…)
see e.g. http://karpathy.github.io/2015/05/21/rnn-effectiveness/
(in Chinese, RNNs can be used directly on characters without word segmentation; the equivalent of “character RNNs” might be models that decompose characters into radicals/strokes)

Byte Pair Encoding (BPE):

— Learn which character sequences are common in the language (‘ing’, ‘pre’, ‘at’, …)

— Split input into these sequences and learn embeddings for these sequences

Attention mechanisms

Compute a probability distribution $\alpha = (\alpha_{1t}, \ldots, \alpha_{St})$ over the *encoder's* hidden states $\mathbf{h}^{(s)}$ that depends on the *decoder's* current $\mathbf{h}^{(t)}$

$$\alpha_{ts} = \frac{\exp(s(\mathbf{h}^{(t)}, \mathbf{h}^{(s)}))}{\sum_{s'} \exp(s(\mathbf{h}^{(t)}, \mathbf{h}^{(s')}))}$$

Compute a weighted avg. of the encoder's $\mathbf{h}^{(s)}$: $\mathbf{c}^{(t)} = \sum_{s=1..S} \alpha_{ts}\mathbf{h}^{(s)}$

that gets then used with $\mathbf{h}^{(t)}$, e.g. in $\mathbf{o}^{(t)} = \tanh(W_1\mathbf{h}^{(t)} + W_2\mathbf{c}^{(t)})$

— **Hard attention** (degenerate case, non-differentiable):
α is a one-hot vector

— **Soft attention** (general case): α is not a one-hot
- — $s(\mathbf{h}^{(t)}, \mathbf{h}^{(s)}) = \mathbf{h}^{(t)} \cdot \mathbf{h}^{(s)}$ is the dot product (no learned parameters)
- — $s(\mathbf{h}^{(t)}, \mathbf{h}^{(s)}) = (\mathbf{h}^{(t)})^T W \mathbf{h}^{(s)}$ (learn a bilinear matrix W)
- — $s(\mathbf{h}^{(t)}, \mathbf{h}^{(s)}) = \mathbf{v}^T \tanh(W_1\mathbf{h}^{(t)} + W_2\mathbf{h}^{(s)})$ concat. hidden states

Activation functions

Recap: Activation functions

Sigmoid (logistic function):

$\sigma(x) = 1/(1 + e^{-x})$

Returns values bound above and below in the $[0,1]$ range

Hyperbolic tangent:

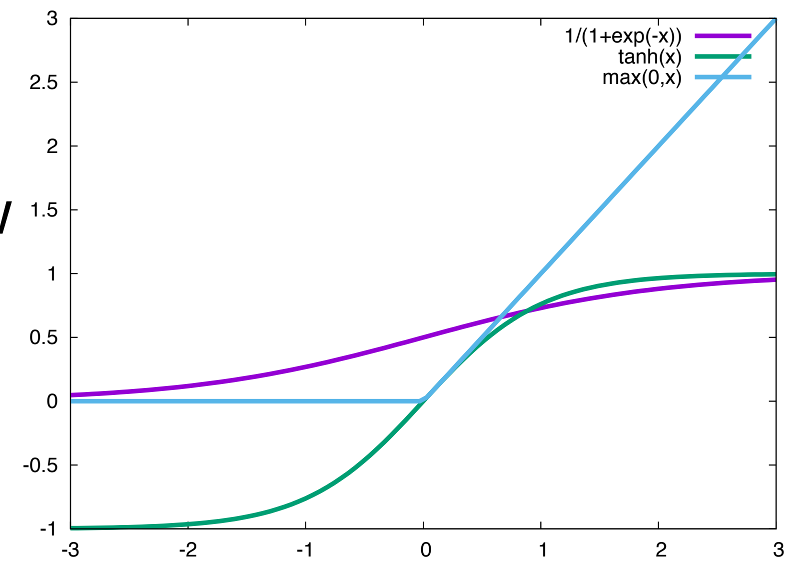
$\tanh(x) = (e^{2x} - 1)/(e^{2x}+1)$

Returns values bound above and below in the $[-1, +1]$ range

Rectified Linear Unit:

$\mathrm{ReLU}(x) = \max(0, x)$

Returns values bound below in the $[0, +\infty]$ range



From RNNs to LSTMs

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 $\mathbf{x}^{(t)}$:

$$\mathbf{h}^{(t)} = g(W_h[\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}] + b_h)$$

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 by Hochreiter and Schmidhuber to overcome this problem.

— They introduce an additional cell state that also gets passed through the network and updated at each time step

— LSTMs define three different gates that read in the previous hidden state and current input to decide how much of the past hidden and cell states to keep.

— This gating mechanism mitigates the vanishing/exploding gradient problems of traditional RNNs

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.:

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

$\mathbf{g}$ 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 $g_i \approx 1,\ v_i \approx u_i$

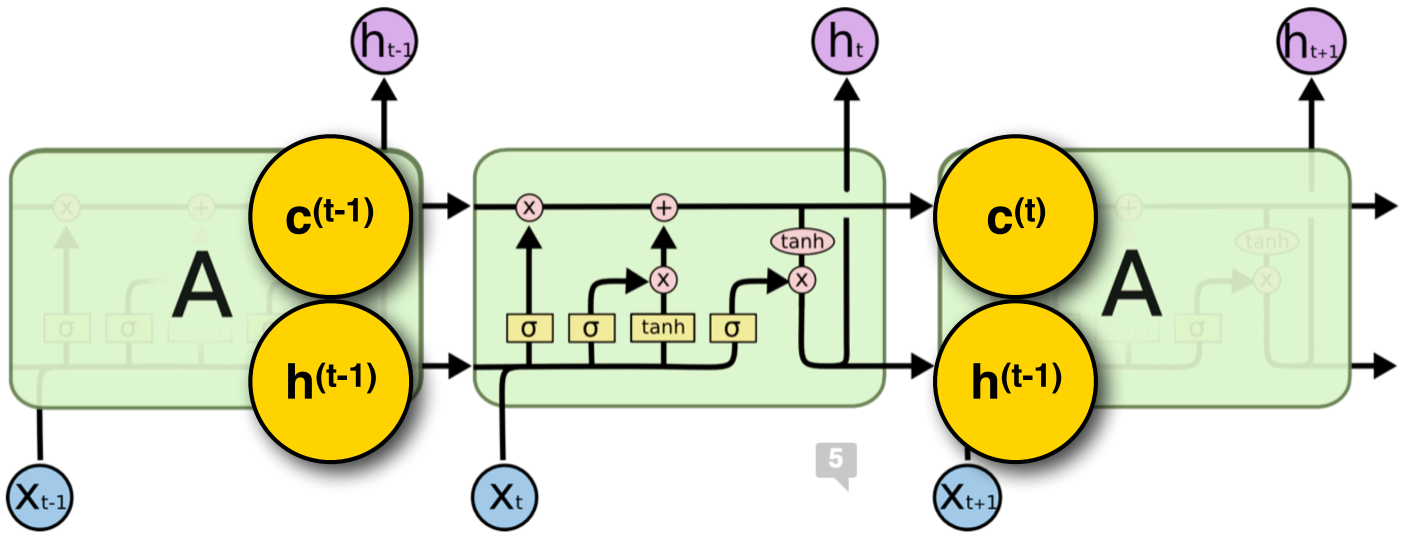
— Each g_i is associated with its own set of trainable parameters and determines how much of u_i to keep or forget

Gates are used to form linear combinations of vectors $\mathbf{u}, \mathbf{v}$:

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

— Addition of two gates: $\mathbf{w} = \mathbf{g_1} \otimes \mathbf{u} + \mathbf{g_2} \otimes \mathbf{v}$

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

1) A new **intermediate cell state** $\tilde{\mathbf{c}}^{(t)}$ that depends on $\mathbf{h}^{(t-1)}$ and $\mathbf{x}^{(t)}$:
$\tilde{\mathbf{c}}^{(t)} = \tanh(W_c[\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}] + b_c)$

2) Three gates (which each depend on $\mathbf{h}^{(t-1)}$ and $\mathbf{x}^{(t)}$)
   a) The **forget gate** $\mathbf{f}^{(t)} = \sigma(W_f[\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}] + b_f)$ decides how much of the last $\mathbf{c}^{(t-1)}$ to remember in the cell state: $\mathbf{f}^{(t)} \otimes \mathbf{c}^{(t-1)}$
   b) The **input gate** $\mathbf{i}^{(t)} = \sigma(W_i[\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}] + b_i)$ decides how much of the intermediate $\tilde{\mathbf{c}}^{(t)}$ to use in the new cell state: $\mathbf{i}^{(t)} \otimes \tilde{\mathbf{c}}^{(t)}$
   c) The **output gate** $\mathbf{o}^{(t)} = \sigma(W_o[\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}] + b_o)$ decides how much of the new $\mathbf{c}^{(t)}$ to use in $\mathbf{h}^{(t)} = \mathbf{o}^{(t)} \otimes \tanh(\mathbf{c}^{(t)})$

3) The new **cell state** $\mathbf{c}^{(t)} = \mathbf{f}^{(t)} \otimes c^{(t-1)} + \mathbf{i}^{(t)} \otimes \tilde{\mathbf{c}}^{(t)}$ 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 $\mathbf{i}^{(t)}$

4) The new **hidden state** $\mathbf{h}^{(t)} = \mathbf{o}^{(t)} \otimes \tanh(\mathbf{c}^{(t)})$

LSTM summary

Based on $\mathbf{c}^{(t-1)}$, $\mathbf{h}^{(t-1)}$, and $\mathbf{x}^{(t)}$, the LSTM computes:

- Intermediate cell state $\tilde{\mathbf{c}}^{(t)} = \tanh(W_c[\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}] + b_c)$
- Forget gate $\mathbf{f}^{(t)} = \sigma(W_f[\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}] + b_f)$
- Input gate $\mathbf{i}^{(t)} = \sigma(W_i[\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}] + b_i)$
- New (final) cell state $\mathbf{c}^{(t)} = \mathbf{f}^{(t)} \otimes c^{(t-1)} + \mathbf{i}^{(t)} \otimes \tilde{\mathbf{c}}^{(t)}$

- Output gate $\mathbf{o}^{(t)} = \sigma(W_o[\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}] + b_o)$
- New hidden state $\mathbf{h}^{(t)} = \mathbf{o}^{(t)} \otimes \tanh(\mathbf{c}^{(t)})$

$\mathbf{c}^{(t)}$ and $\mathbf{h}^{(t)}$ are passed on to the next time step.

Gated Recurrent Units (GRUs)

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

GRU definition

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

— a reset gate $\mathbf{r}^{(t)}$ to determine how much of $\mathbf{h}^{(t-1)}$ to keep in $\tilde{\mathbf{h}}^{(t)}$
$$\mathbf{r}^{(t)} = \sigma(W_r\mathbf{x}^{(t)} + U_r\mathbf{h}^{(t-1)} + b_r)$$

— an intermediate hidden state $\tilde{\mathbf{h}}^{(t)}$ that depends on $\mathbf{x}^{(t)}$ and $\mathbf{r}^{(t)} \otimes \mathbf{h}^{(t-1)}$
$$\tilde{\mathbf{h}}^{(t)} = \phi(W_h\mathbf{x}^{(t)} + U_h(\mathbf{r}^{(t)} \otimes \mathbf{h}^{(t-1)}) + b_r)$$

— an update gate $\mathbf{z}^{(t)}$ to determine how much of $\mathbf{h}^{(t-1)}$ to keep in $\mathbf{h}^{(t)}$
$$\mathbf{z}^{(t)} = \sigma(W_z\mathbf{x}^{(t)} + U_z\mathbf{h}^{(t-1)} + b_r)$$

— 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 update gate $\mathbf{z}^{(t)}$
$$\mathbf{h}^{(t)} = \mathbf{z}^{(t)} \otimes \mathbf{h}^{(t-1)} + (\mathbf{1} - \mathbf{z}^{(t)}) \otimes \tilde{\mathbf{h}}^{(t)}$$

Expressive power of RNN, LSTM, GRU

Weiss, Goldberg, Yahav (2018)
On the Practical Computational Power
of Finite Precision RNNs for Language Recognition
https://www.aclweb.org/anthology/P18-2117.pdf

Models

Basic RNNs:

Simple (Elman) SRNN: $\mathbf{h}^{(t)} = \tanh(W\mathbf{x}^{(t)} + U\mathbf{h}^{(t-1)} + b)$

IRNN: $\mathbf{h}^{(t)} = ReLU(W\mathbf{x}^{(t)} + U\mathbf{h}^{(t-1)} + b)$

Gated RNNs (GRUs and LSTMs)

Gates $\mathbf{g}_k^{(t)} = \sigma(W_k\mathbf{x}^{(t)} + U_k\mathbf{h}^{(t-1)} + b_k)$: each element is a probability

NB: a gate can return $\mathbf{0}$ or $\mathbf{1}$ by setting its matrices to 0 and b=0 or b=1

GRU with gates $\mathbf{r}^{(t)}, \mathbf{z}^{(t)}$

hidden state $\tilde{\mathbf{h}}^{(t)} = \tanh(W_h\mathbf{x}^{(t)} + U_h(\mathbf{r}^{(t)} \otimes \mathbf{h}^{(t-1)}) + b_r)$

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

NB: GRU reduces to SRNN with $\mathbf{r} = \mathbf{1},\ \mathbf{z} = \mathbf{0}$

LSTM with gates $\mathbf{f}^{(t)}, \mathbf{i}^{(t)}, \mathbf{o}^{(t)}$,

memory cell $\tilde{\mathbf{c}}^{(t)} = \tanh(W_c\mathbf{x}^{(t)} + U_c\mathbf{h}^{(t-1)} + b_c)$

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

hidden state $\mathbf{h}^{(t)} = \mathbf{o}^{(t)} \otimes \phi(\mathbf{c}^{(t)})$ for ϕ = identity or $\tanh$

NB: LSTM reduces to SRNN with $\mathbf{f} = \mathbf{0}, \mathbf{i} = \mathbf{1}, \mathbf{o} = \mathbf{1}$

Simplified k-Counter Machines (SKCM)

A finite-state automaton with *k* counters

Depending on the input, in each step, each counter can be:
— incremented (INC) by a fixed amount
— decremented (DEC) by a fixed amoung
— or left as is

State transitions and accept/reject decisions
can compare each counter to 0 (COMP0)

SKCMs can recognize $a^n b^n$ (context-free) and $a^n b^n c^n$ (context-sensitive), but not palindromes ($S \rightarrow x \mid aSa \mid bSb$) (also context-free)

LSTMs and Counting

LSTMs can be used to implement an SKCM:

— *k* dimensions of the memory cell $\mathbf{c}^{(t)}$ are counters

— Non-counting steps:

Set $i_j^{(t)}=0$, $f_j^{(t)}=1$ to leave counter unmodified:

$$c_j^{(t)} = 1 \cdot c_j^{(t-1)} + 0 \cdot \tilde{c}_j^{(t)} = c_j^{(t-1)}$$

— Counting steps:

Set $i_j^{(t)}=1$, $f_j^{(t)}=1$ to increment/decrement cell:

$$c_j^{(t)} = 1 \cdot c_j^{(t-1)} + 1 \cdot \tilde{c}_j^{(t)} = c_j^{(t-1)} + \tilde{c}_j^{(t)}$$

— Reset counter to 0:

Set $i_j^{(t)}=0$, $f_j^{(t)}=0$ to increment/decrement cell:

$$c_j^{(t)} = 0 \cdot c_j^{(t-1)} + 0 \cdot \tilde{c}_j^{(t)} = 0$$

— Comparing counters to 0:

$h_j^{(t)} = o_j^{(t)} c_j^{(t)}$ and $h_j^{(t)} = o_j^{(t)} \tanh(c_j^{(t)})$ are both 0 iff $c_j^{(t)} = 0$

Simple RNNs and Counting

Update:

$$h_i^{(t)} = \tanh\left(\sum_j W_{ij} x_j^{(t)} + \sum_j U_{ij} h_j^{(t-1)} + b_i \right)$$

The tanh() activation function means the activation lies within [-1,+1]
With finite precision, counting can only be achieved within a narrow range (and will be unstable)

Simple RNNs have poor generalization capabilities for counting

IRNNs and counting

Update:

$$\mathbf{h}^{(t)} = ReLU(W\mathbf{x}^{(t)} + U\mathbf{h}^{(t-1)} + b)$$
$$= \max(0,\ W\mathbf{x}^{(t)} + U\mathbf{h}^{(t-1)} + b)$$

The ReLU maps all negative numbers to 0,
but leaves positive numbers unchanged

Finite-precision IRNNs can perform unbounded counting by representing each counter as *two* dimensions:
— INC increments one dimension
— DEC increments the other dimension
— COMP0 compares their difference to 0.

But: IRNNs are difficult to train because
they have exploding gradients. So they don't work well.

GRUs and counting

Updates

$$\tilde{\mathbf{h}}^{(t)} = \tanh(W_h \mathbf{x}^{(t)} + U_h(\mathbf{r}^{(t)} \otimes \mathbf{h}^{(t-1)}) + b_r)$$
$$\mathbf{h}^{(t)} = \mathbf{z}^{(t)} \otimes \mathbf{c}^{(t-1)} + (\mathbf{1} - \mathbf{z}^{(t)}) \otimes \tilde{\mathbf{h}}^{(t-1)}$$

Finite-precision GRUs cannot implement unbounded counting because the tanh squashing and linear interpolation restrict hidden state values to [-1,1]

GRUs can learn counting up to a finite bound seen in training, but won't generalize beyond that.
Counting requires setting gates and hidden states to precise non-saturated values that are difficult to find

Summary

— Simple RNN and GRU cannot represent unbounded counting (mostly because they use $\tanh$ and linear interpolation)
—IRNN and LSTM can represent unbounded counting

Claims about other LSTM variants
— **Coupling the input and forget gates**
by setting $\mathbf{i} = (\mathbf{1} - \mathbf{f})$ also removes their counting ability
—**"Peephole connections"** where gates read cell states
'essentially' uses identity as activation function, and allows comparing counters in a stable way

Peephole connections: feed cell states into gates

$$\mathbf{f}^{(t)} = \sigma(W_f \mathbf{x}^{(t)} + U_f \mathbf{h}^{(t-1)} + V_f \mathbf{c}^{(t-1)} + b_f)$$
$$\mathbf{i}^{(t)} = \sigma(W_i \mathbf{x}^{(t)} + U_i \mathbf{h}^{(t-1)} + V_i \mathbf{c}^{(t-1)} + b_i)$$
$$\mathbf{o}^{(t)} = \sigma(W_o \mathbf{x}^{(t)} + U_o \mathbf{h}^{(t-1)} + V_o \mathbf{c}^{(t)} + b_o)$$

Experiments

Setup:

—Train models to recognize strings in a language (binary classification: accept if input string is in the language, reject otherwise)

—Each model has one layer, and a hidden size of 10

—Training on $a^n b^n$ up to *n*=100, on $a^n b^n c^n$ up to *n*=50

Results:

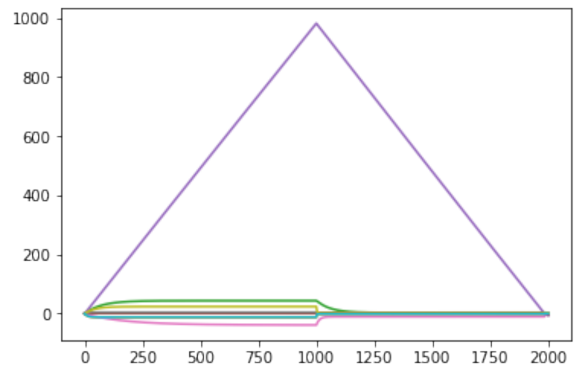
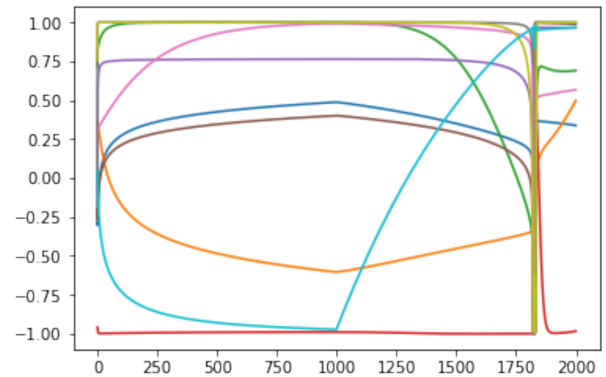
— Counting mechanisms are not precise; fail for very large *n*

— But LSTMs can be trained to recognize $a^n b^n$ and $a^n b^n c^n$ for much greater *n* than seen during training.

— These trained LSTMs do use per-dimension counters

— GRUs can also be trained to recognize $a^n b^n$ and $a^n b^n c^n$ but without counting dimensions, and much poorer generalization (they fail even on some training examples)

LSTM vs GRU: activations

	LSTM	GRU
a^nb^n models on $a^{1000}b^{1000}$		
$a^nb^nc^n$ models on $a^{100}b^{100}c^{100}$