Deep NLP
Recurrent Neural Networks

Richard Socher

richard@metamind.io

Overview: Today

- RNN language models
- Important training problems and tricks
- RNNs for other sequence tasks
- Bidirectional and deep RNNs
- RNN extensions: GRU, LSTM for MT

- Tomorrow: Fun applications and new DMN model

Language Models

A language model computes a probability for a sequence of words: $P(w_1, \ldots, w_T)$

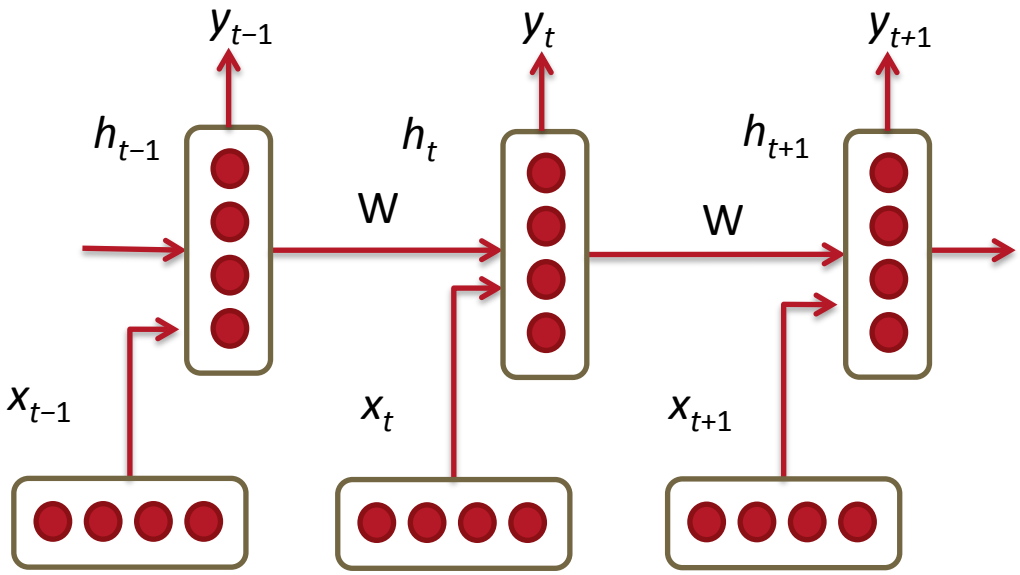
- Useful for machine translation and speech
  - Word choice:
    p(walking home after school) > p(walking house after school)

- Use incorrect but necessary Markov assumptions

$$P(w_1, \ldots, w_m) = \prod_{i=1}^{m} P(w_i \mid w_1, \ldots, w_{i-1}) \approx \prod_{i=1}^{m} P(w_i \mid w_{i-(n-1)}, \ldots, w_{i-1})$$

Recurrent Neural Networks!

- RNNs tie the weights at each time step
- Condition the neural network on all previous words
- RAM requirement only scales with number of words



Recurrent Neural Network language model

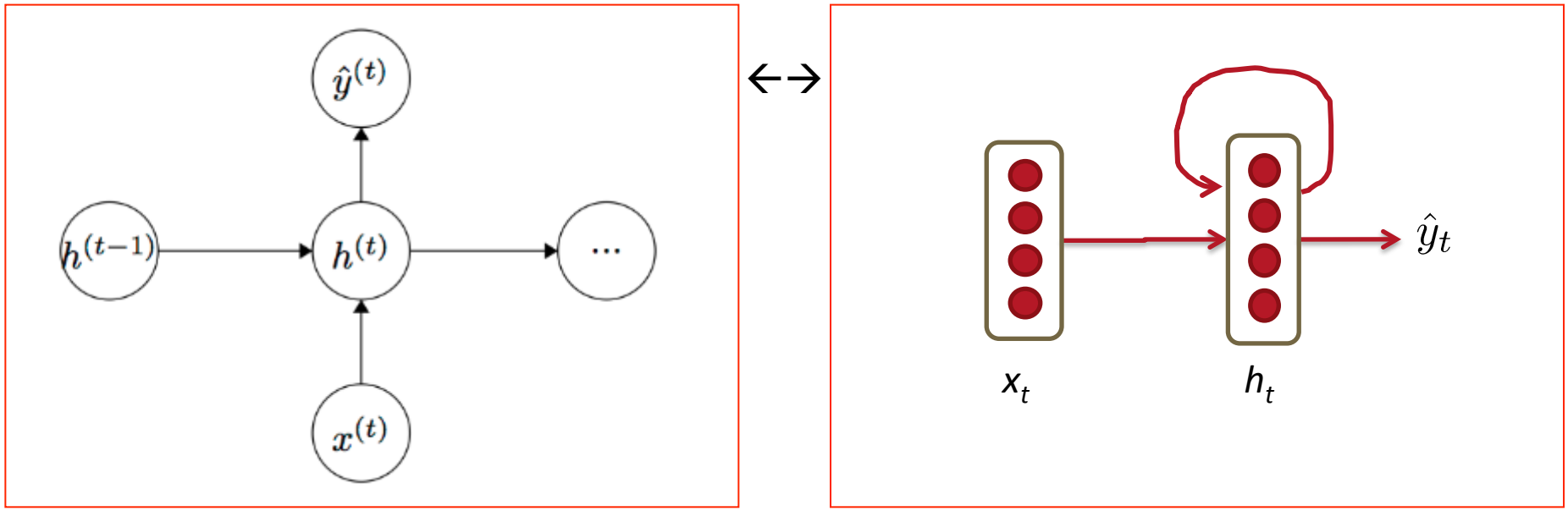
Given list of word **vectors**: $x_1, \ldots, x_{t-1}, x_t, x_{t+1}, \ldots, x_T$

At a single time step:

$$h_t = \sigma\left(W^{(hh)} h_{t-1} + W^{(hx)} x_{[t]}\right)$$

$$\hat{y}_t = \text{softmax}\left(W^{(S)} h_t\right)$$

$$\hat{P}(x_{t+1} = v_j \mid x_t, \ldots, x_1) = \hat{y}_{t,j}$$



Recurrent Neural Network language model

We use the same set of W weights at all time steps!

Everything else is the same:

$$
\begin{aligned}
h_t &= \sigma\left(W^{(hh)} h_{t-1} + W^{(hx)} x_{[t]}\right) \\
\hat{y}_t &= \text{softmax}\left(W^{(S)} h_t\right) \\
\hat{P}(x_{t+1} = v_j \mid x_t, \ldots, x_1) &= \hat{y}_{t,j}
\end{aligned}
$$

$h_0 \in \mathbb{R}^{D_h}$ is some initialization vector for the hidden layer at time step 0

$x_{[t]}$ is the column vector of L at index [t] at time step t

$W^{(hh)} \in \mathbb{R}^{D_h \times D_h}$ $W^{(hx)} \in \mathbb{R}^{D_h \times d}$ $W^{(S)} \in \mathbb{R}^{|V| \times D_h}$

Objective function for language models

$\hat{y} \in \mathbb{R}^{|V|}$ is a probability distribution over the vocabulary

Same cross entropy loss function but predicting words instead of classes

$$J^{(t)}(\theta) = -\sum_{j=1}^{|V|} y_{t,j} \log \hat{y}_{t,j}$$

Recurrent Neural Network language model

Evaluation could just be negative of average log probability over dataset of size (number of words) T:

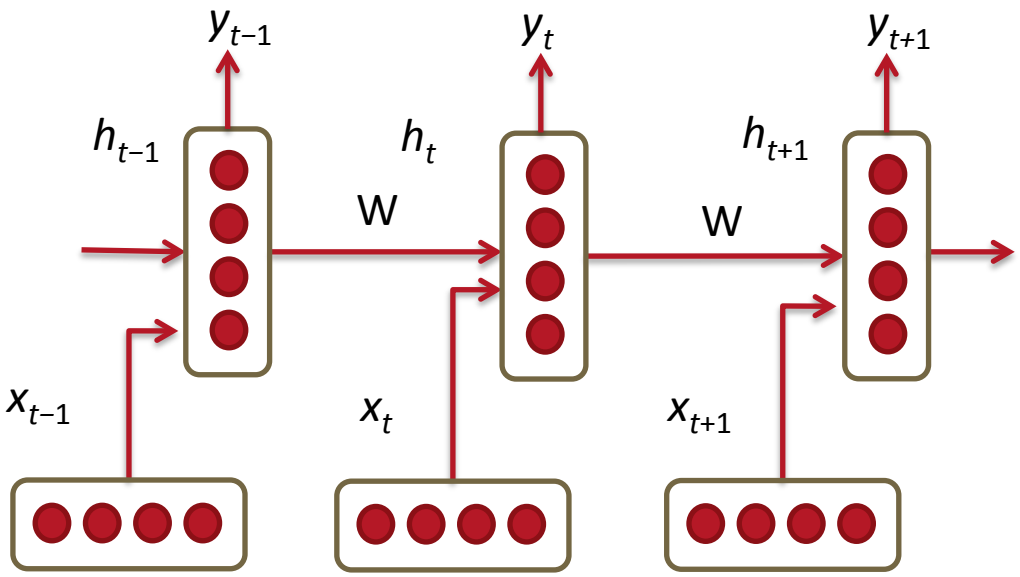
$$J = -\frac{1}{T}\sum_{t=1}^{T}\sum_{j=1}^{|V|} y_{t,j} \log \hat{y}_{t,j}$$

But more common: Perplexity: 2^J

Lower is better!

Training RNNs is hard

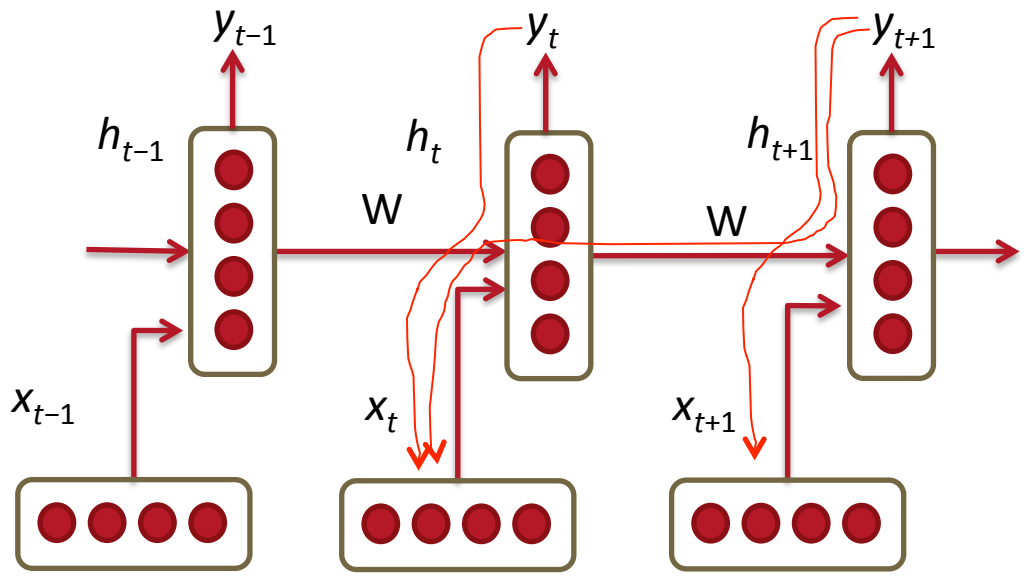
- Multiply the same matrix at each time step during forward prop



- Ideally inputs from many time steps ago can modify output *y*
- Take $\frac{\partial E_2}{\partial W}$ for an example RNN with 2 time steps! Insightful!

The vanishing/exploding gradient problem

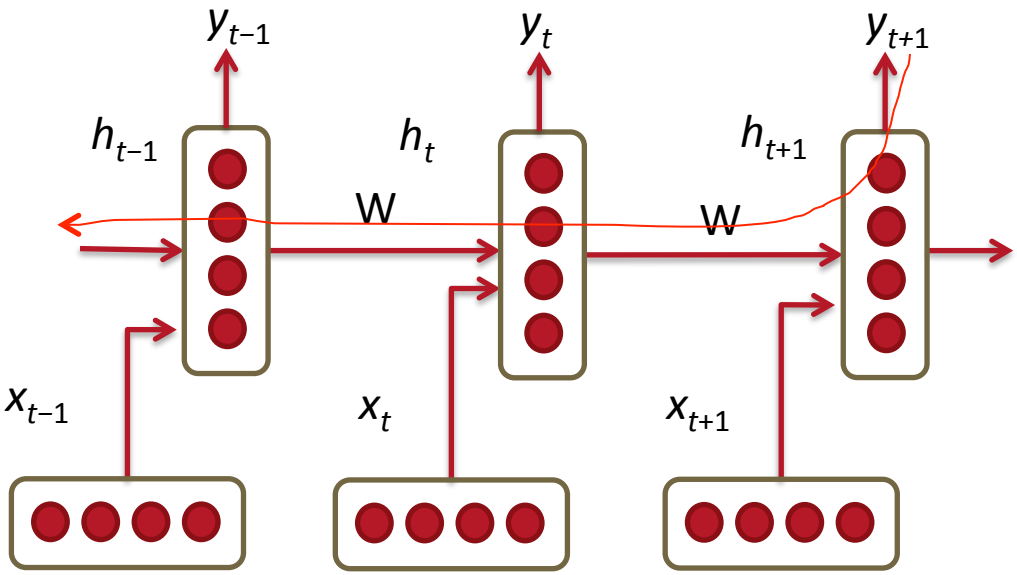
- Multiply the same matrix at each time step during backprop



- Detailed derivations in the appendix of these slides!

Why is the vanishing gradient a problem?

- The error at a time step ideally can tell a previous time step from many steps away to change during backprop



The vanishing gradient problem for language models

- In the case of language modeling or question answering words from time steps far away are not taken into consideration when training to predict the next word

- Example:

  Jane walked into the room. John walked in too. It was late in the day. Jane said hi to ____

Trick for exploding gradient: clipping trick

- The solution first introduced by Mikolov is to clip gradients to a maximum value.

Algorithm 1 Pseudo-code for norm clipping the gradients whenever they explode

$\hat{\mathbf{g}} \leftarrow \frac{\partial \mathcal{E}}{\partial \theta}$
if $\|\hat{\mathbf{g}}\| \geq threshold$ **then**
  $\hat{\mathbf{g}} \leftarrow \frac{threshold}{\|\hat{\mathbf{g}}\|} \hat{\mathbf{g}}$
end if

- Makes a big difference in RNNs.

Gradient clipping intuition

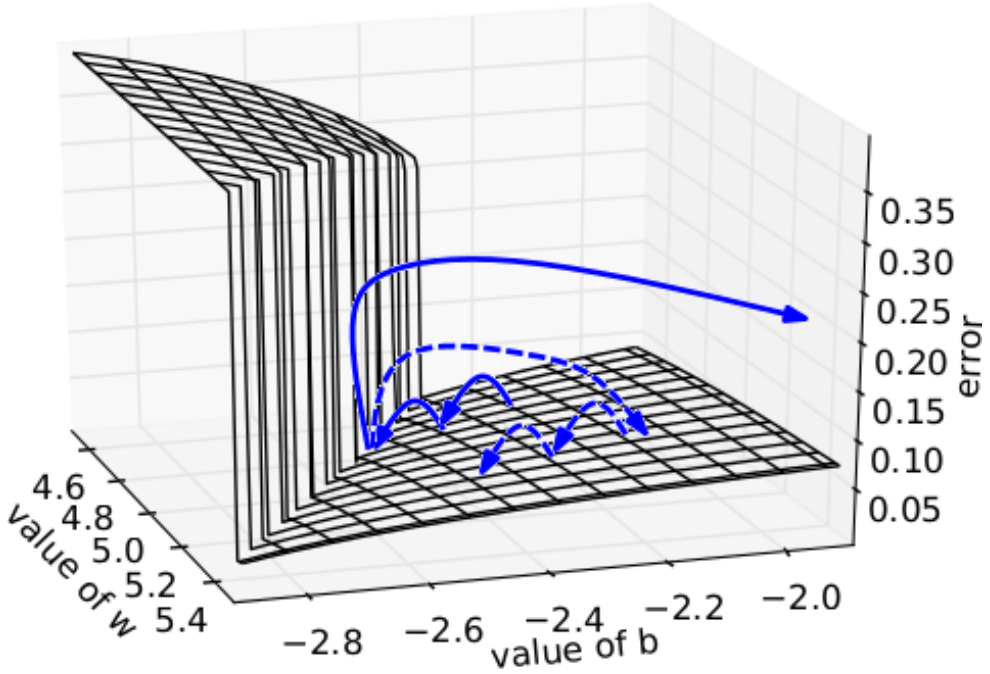
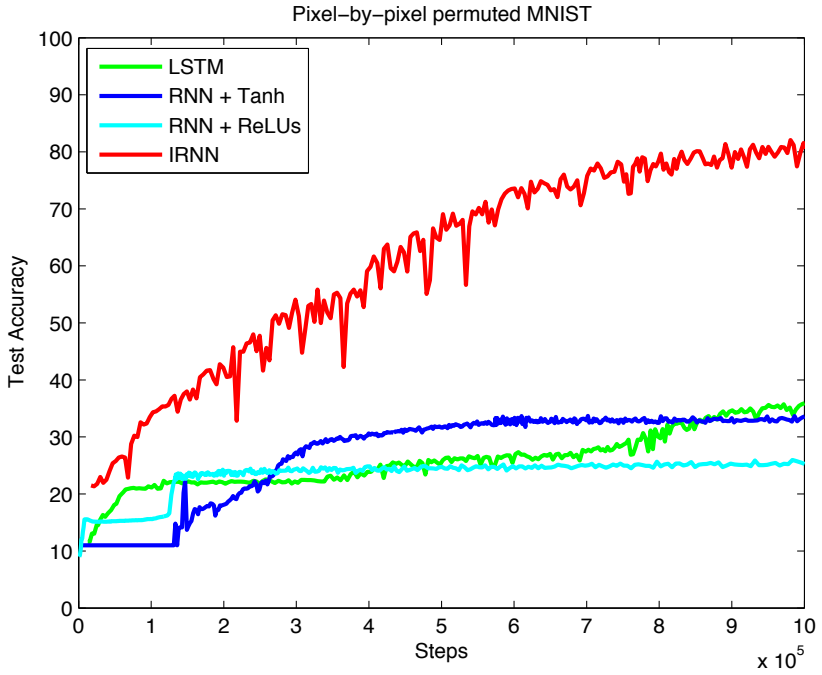


Figure from paper:
On the difficulty of training Recurrent Neural Networks, Pascanu et al. 2013

- Error surface of a single hidden unit RNN,
- High curvature walls
- Solid lines: standard gradient descent trajectories
- Dashed lines gradients rescaled to fixed size

For vanishing gradients: Initialization + ReLus!

- Initialize $W^{(*)}$'s to identity matrix I and f(z) = $\text{rect}(z) = \max(z, 0)$
- → Huge difference!



- Initialization idea first introduced in *Parsing with Compositional Vector Grammars*, Socher et al. 2013
- New experiments with recurrent neural nets in *A Simple Way to Initialize Recurrent Networks of Rectified Linear* Units, Le et al. 2015

Perplexity Results

KN5 = Count-based language model with Kneser-Ney smoothing & 5-grams

Table 2. *Comparison of different neural network architectures on Penn Corpus (1M words) and Switchboard (4M words).*

	Penn Corpus		Switchboard	
Model	NN	NN+KN	NN	NN+KN
KN5 (baseline)	-	141	-	92.9
feedforward NN	141	118	85.1	77.5
RNN trained by BP	137	113	81.3	75.4
RNN trained by BPTT	123	106	77.5	72.5

Table from paper *Extensions of recurrent neural network language model* by Mikolov et al 2011

Problem: Softmax is huge and slow

Trick: Class-based word prediction

$p(w_t|\text{history}) = p(c_t|\text{history})p(w_t|c_t)$

$= p(c_t|h_t)p(w_t|c_t)$

The more classes,
the better perplexity
but also worse speed:

Table 3. *Perplexities on Penn corpus with factorization of the output layer by the class model. All models have the same basic configuration (200 hidden units and BPTT=5). The Full model is a baseline and does not use classes, but the whole 10K vocabulary.*

Classes	RNN	RNN+KN5	Min/epoch	Sec/test
30	134	112	12.8	8.8
50	136	114	9.8	6.7
100	136	114	9.1	5.6
200	136	113	9.5	6.0
400	134	112	10.9	8.1
1000	131	111	16.1	15.7
2000	128	109	25.3	28.7
4000	127	108	44.4	57.8
6000	127	109	70	96.5
8000	124	107	107	148
Full	123	106	154	212

One last implementation trick

- You only need to pass backwards through your sequence once and accumulate all the deltas from each E_t

Sequence modeling for other tasks

- Classify each word into:
  - NER
  - Entity level sentiment in context
  - opinionated expressions

- Example application and slides from paper *Opinion Mining with Deep Recurrent Nets* by Irsoy and Cardie 2014

Opinion Mining with Deep Recurrent Nets

Goal: Classify each word as

direct subjective expressions (DSEs) and
expressive subjective expressions (ESEs).

DSE: Explicit mentions of private states or speech events expressing private states

ESE: Expressions that indicate sentiment, emotion, etc. without explicitly conveying them.

Example Annotation

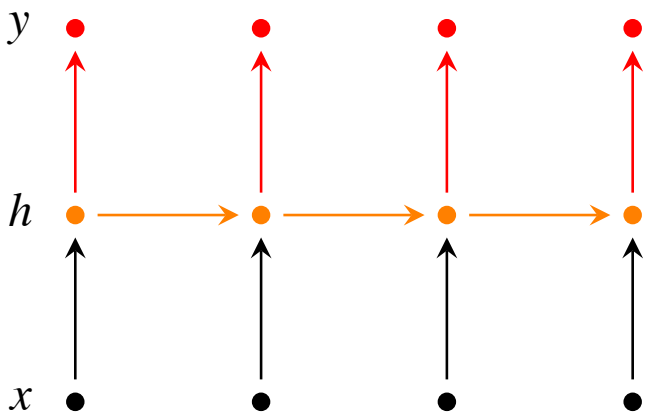
In BIO notation (tags either begin-of-entity (B_*X*) or continuation-of-entity (I_*X*)):
The committee, [as usual]$_{ESE}$, [has refused to make any statements]$_{DSE}$.

The	committee	,	as	usual	,	has
O	O	O	B_ESE	I_ESE	O	B_DSE

refused	to	make	any	statements	.
I_DSE	I_DSE	I_DSE	I_DSE	I_DSE	O

Approach: Recurrent Neural Network

- Notation from paper (so you get used to different ones)

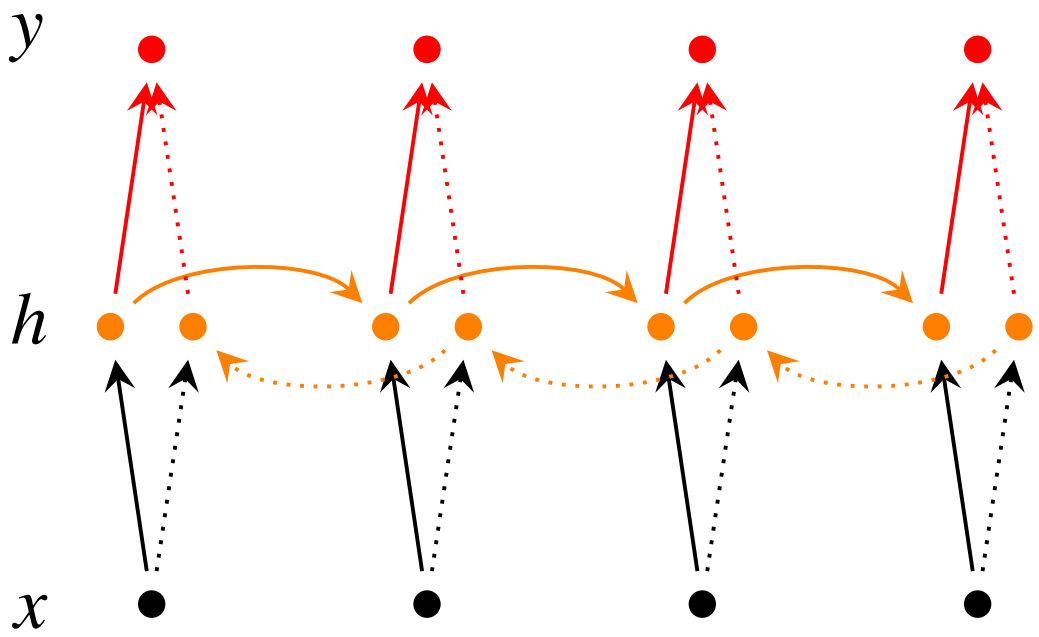


$$h_t = f(Wx_t + Vh_{t-1} + b)$$
$$y_t = g(Uh_t + c)$$

- *x* represents a token (word) as a vector.
- *y* represents the output label (B, I or O) – g = softmax !
- *h* is the memory, computed from the past memory and current word. It summarizes the sentence up to that time.

Bidirectional RNNs

Problem: For classification you want to incorporate information from words both preceding and following



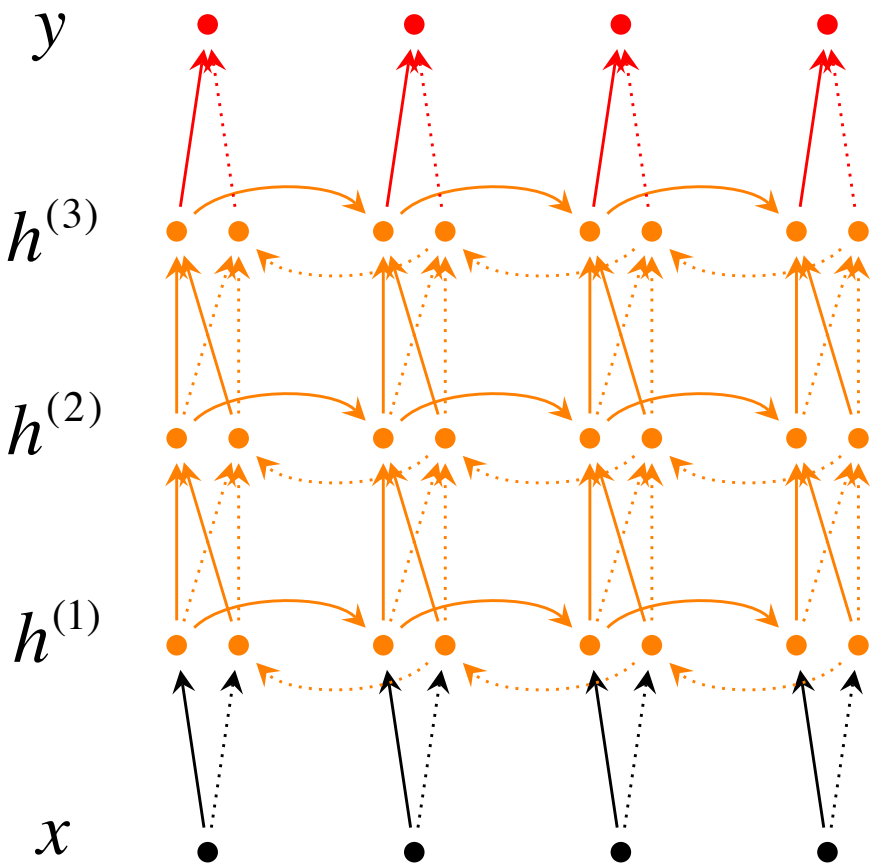
$$\vec{h}_t = f(\vec{W}x_t + \vec{V}\vec{h}_{t-1} + \vec{b})$$

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

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

$h = [\vec{h}; \overleftarrow{h}]$ now represents (summarizes) the past and future around a single token.

Deep Bidirectional RNNs



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

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

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

Each memory layer passes an intermediate sequential representation to the next.

Data

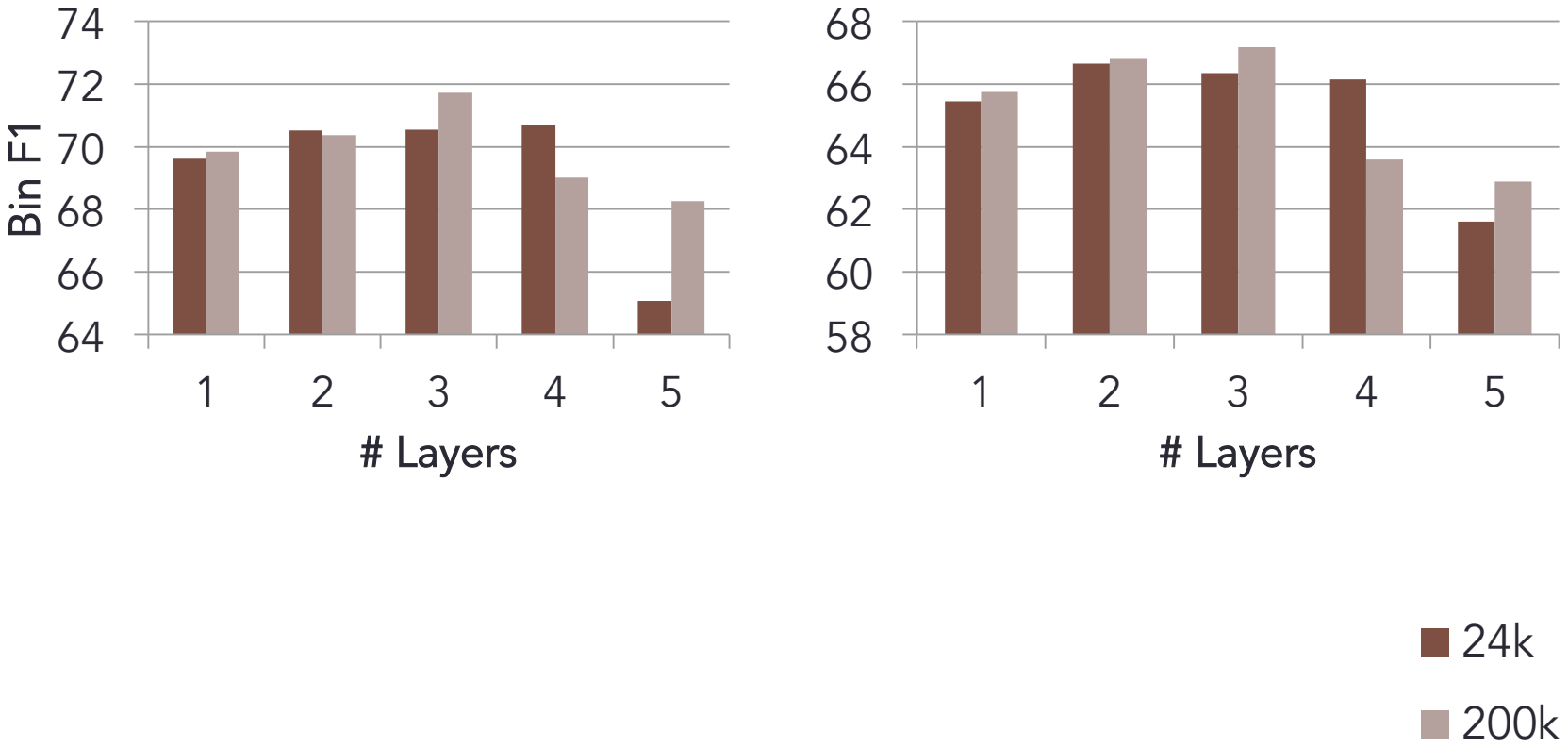
- MPQA 1.2 corpus (Wiebe et al., 2005)
- consists of 535 news articles (11,111 sentences)
- manually labeled with DSE and ESEs at the phrase level
- Evaluation: F1

$$\text{precision} = \frac{tp}{tp + fp}$$

$$\text{recall} = \frac{tp}{tp + fn}$$

$$F1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

Evaluation

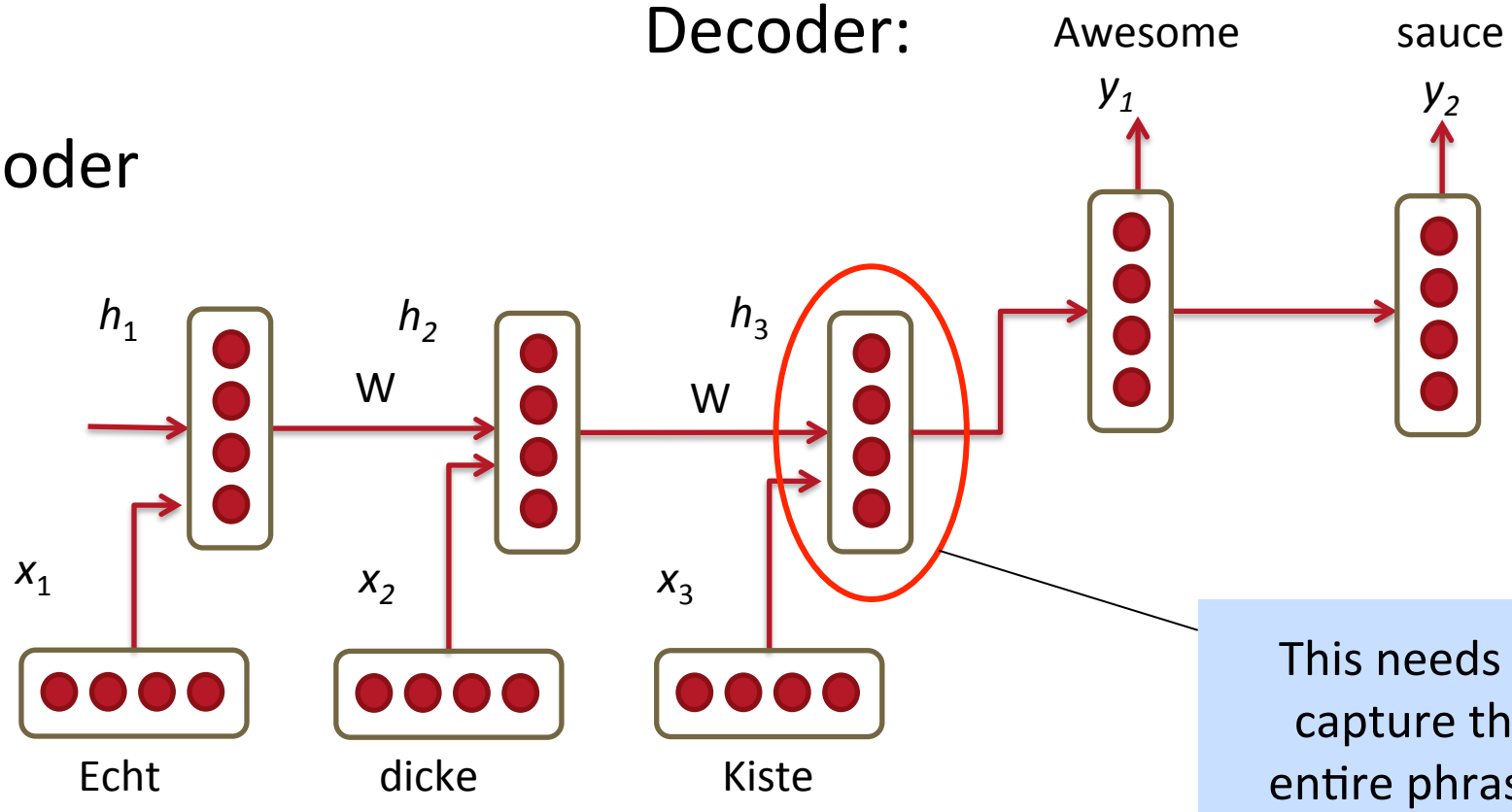

Bin F1
74
72
70
68
66
64
1
2
3
4
5
Layers
68
66
64
62
60
58
1
2
3
4
5
Layers
24k
200k

Machine Translation (MT)

- **Traditional MT:**
  - **A lot** of human feature engineering
  - Very complex systems
  - Many different, independent machine learning problems

Deep learning to the rescue! … ?

Maybe, we could translate directly with an RNN?



MT with RNNs – Simplest Model

Encoder: $h_t = \phi(h_{t-1}, x_t) = f\left(W^{(hh)} h_{t-1} + W^{(hx)} x_t\right)$

Decoder: $h_t = \phi(h_{t-1}) = f\left(W^{(hh)} h_{t-1}\right)$

$$y_t = softmax\left(W^{(S)} h_t\right)$$

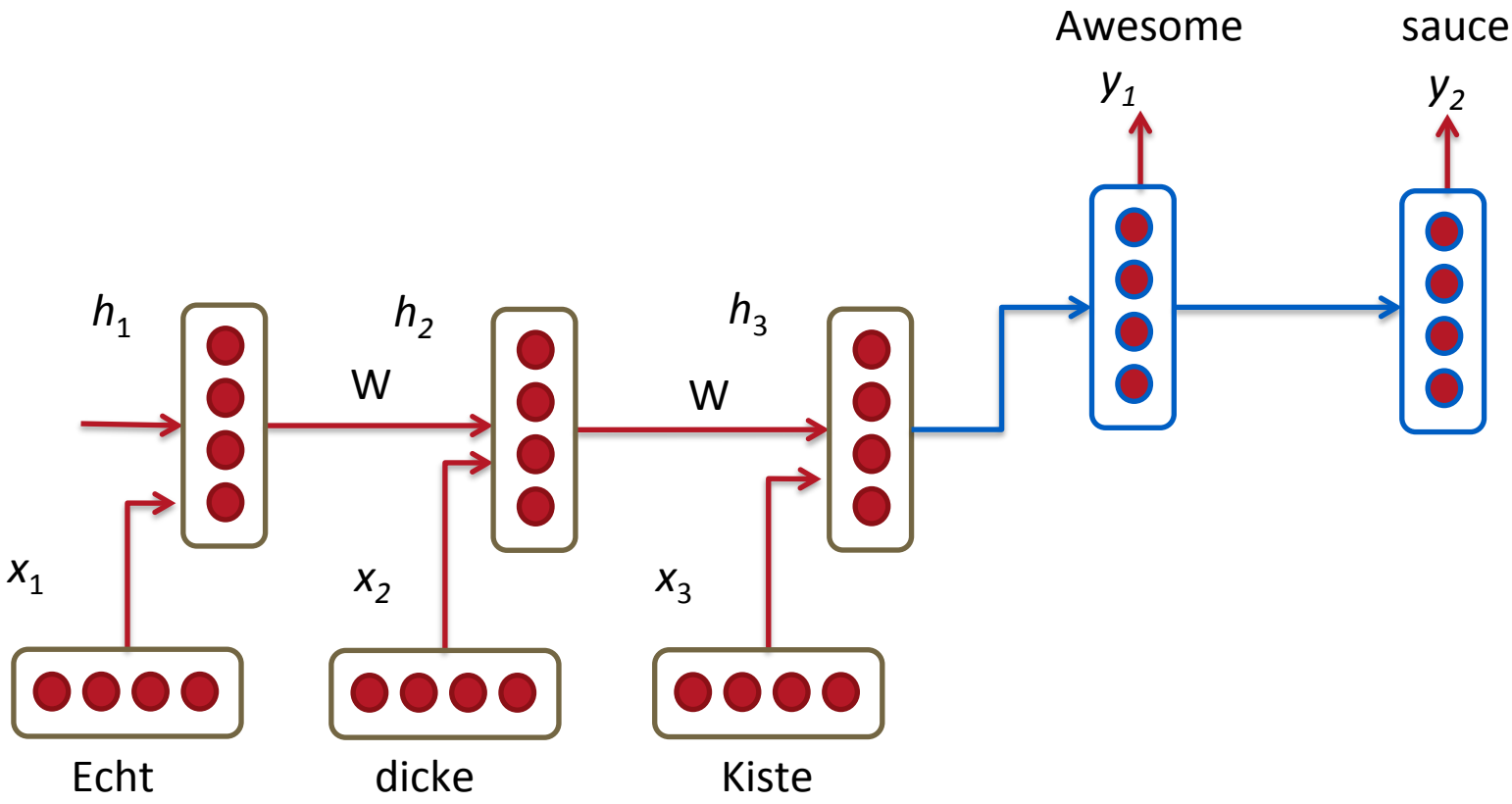
Minimize cross entropy error for all target words conditioned on source words

$$\max_\theta \frac{1}{N} \sum_{n=1}^{N} \log p_\theta(y^{(n)}|x^{(n)})$$

It's not quite that simple ;)

RNN Translation Model Extensions

1. Train different RNN weights for encoding and decoding

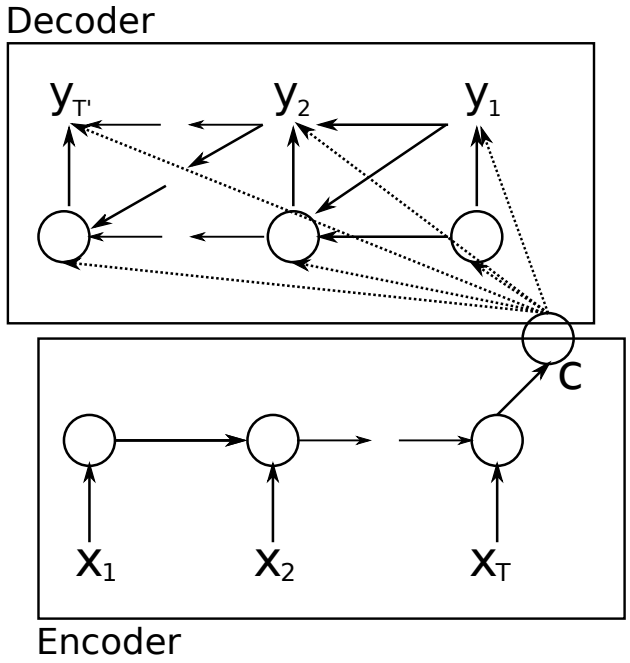


RNN Translation Model Extensions

Notation: Each input of ϕ has its own linear transformation matrix. Simple: $h_t = \phi(h_{t-1}) = f\left(W^{(hh)} h_{t-1}\right)$

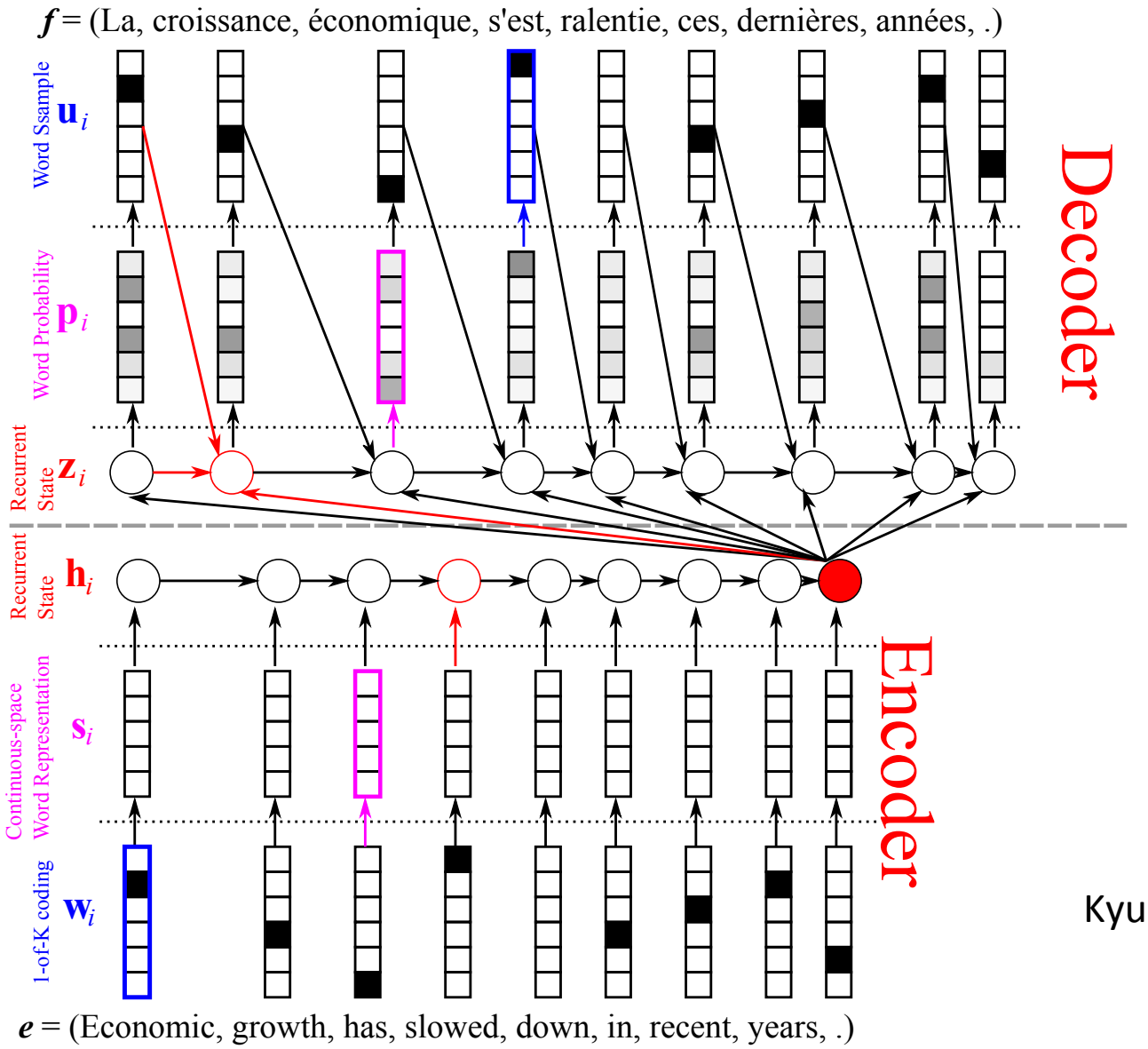
2. Compute every hidden state in decoder from
   - Previous hidden state (standard)
   - Last hidden vector of encoder $c=h_T$
   - Previous predicted output word y_{t-1}

$$h_{D,t} = \phi_D(h_{t-1}, c, y_{t-1})$$



Cho et al. 2014

Different picture, same idea

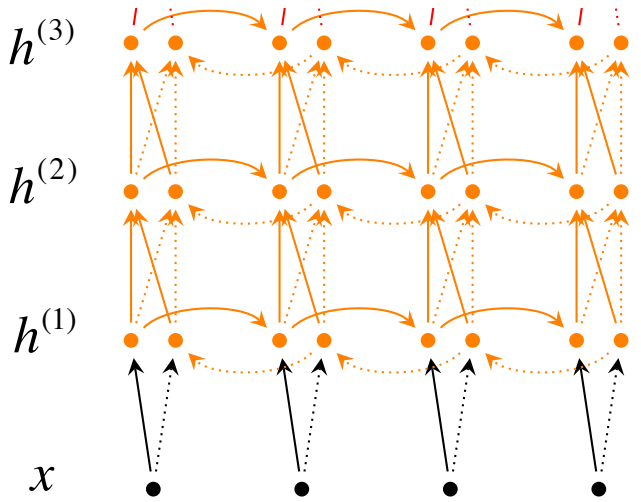


Kyunghyun Cho et al. 2014

RNN Translation Model Extensions

3. Train stacked/deep RNNs with multiple layers

4. Potentially train bidirectional encoder



5. Train input sequence in reverse order for easier optimization problem: Instead of A B C → X Y, train with C B A → X Y

6. Main Improvement: Better Units

- More complex hidden unit computation in recurrence!
- Gated Recurrent Units (GRU)
  introduced by Cho et al. 2014
- Main ideas:
  - keep around memories to capture long distance dependencies
  - allow error messages to flow at different strengths depending on the inputs

GRUs

- Standard RNN computes hidden layer at next time step directly: $h_t = f\left(W^{(hh)}h_{t-1} + W^{(hx)}x_t\right)$
- GRU first computes an update **gate** (another layer) based on current input word vector and hidden state

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

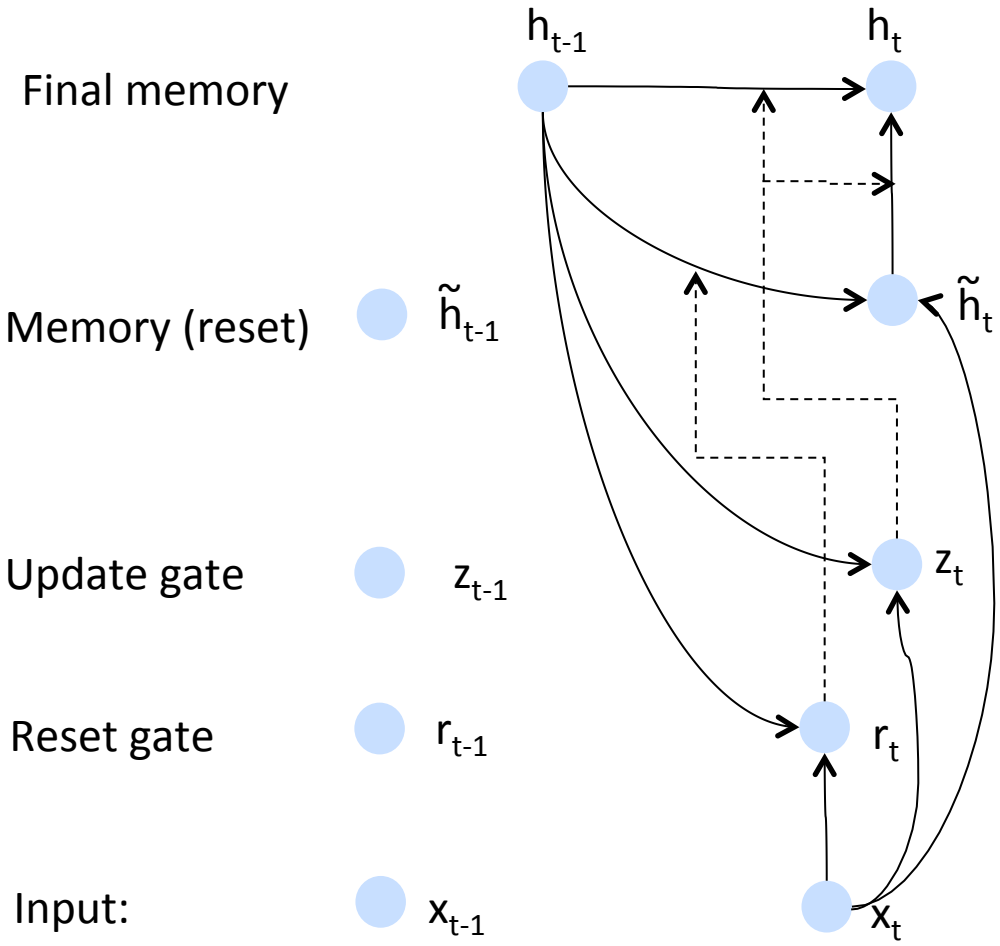
- Compute reset gate similarly but with different weights

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

GRUs

- Update gate $$z_t = \sigma\left(W^{(z)}x_t + U^{(z)}h_{t-1}\right)$$
- Reset gate $$r_t = \sigma\left(W^{(r)}x_t + U^{(r)}h_{t-1}\right)$$
- New memory content: $$\tilde{h}_t = \tanh\left(Wx_t + r_t \circ Uh_{t-1}\right)$$
  If reset gate unit is ~0, then this ignores previous memory and only stores the new word information
- Final memory at time step combines current and previous time steps: $$h_t = z_t \circ h_{t-1} + (1 - z_t) \circ \tilde{h}_t$$

Attempt at a clean illustration



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

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

$$\tilde{h}_t = \tanh\left(Wx_t + r_t \circ Uh_{t-1}\right)$$

$$h_t = z_t \circ h_{t-1} + (1 - z_t) \circ \tilde{h}_t$$

GRU intuition

- If reset is close to 0, ignore previous hidden state → Allows model to drop information that is irrelevant in the future

$$z_t = \sigma\left(W^{(z)}x_t + U^{(z)}h_{t-1}\right)$$
$$r_t = \sigma\left(W^{(r)}x_t + U^{(r)}h_{t-1}\right)$$
$$\tilde{h}_t = \tanh\left(Wx_t + r_t \circ Uh_{t-1}\right)$$
$$h_t = z_t \circ h_{t-1} + (1 - z_t) \circ \tilde{h}_t$$

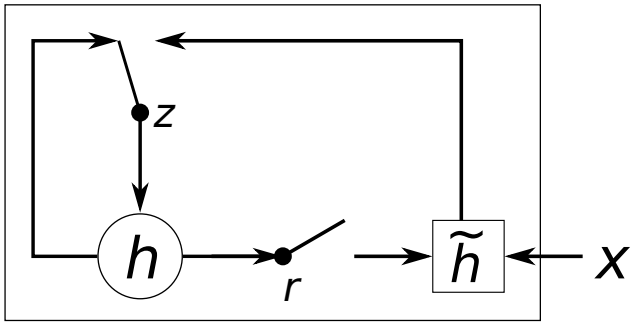
- Update gate z controls how much of past state should matter now.
  - If z close to 1, then we can copy information in that unit through many time steps! **Less vanishing gradient**!
- Units with short-term dependencies often have reset gates very active

GRU intuition

- Units with long term dependencies have active update gates z

$$z_t = \sigma\left(W^{(z)}x_t + U^{(z)}h_{t-1}\right)$$
$$r_t = \sigma\left(W^{(r)}x_t + U^{(r)}h_{t-1}\right)$$
$$\tilde{h}_t = \tanh\left(Wx_t + r_t \circ Uh_{t-1}\right)$$
$$h_t = z_t \circ h_{t-1} + (1 - z_t) \circ \tilde{h}_t$$

- Illustration:

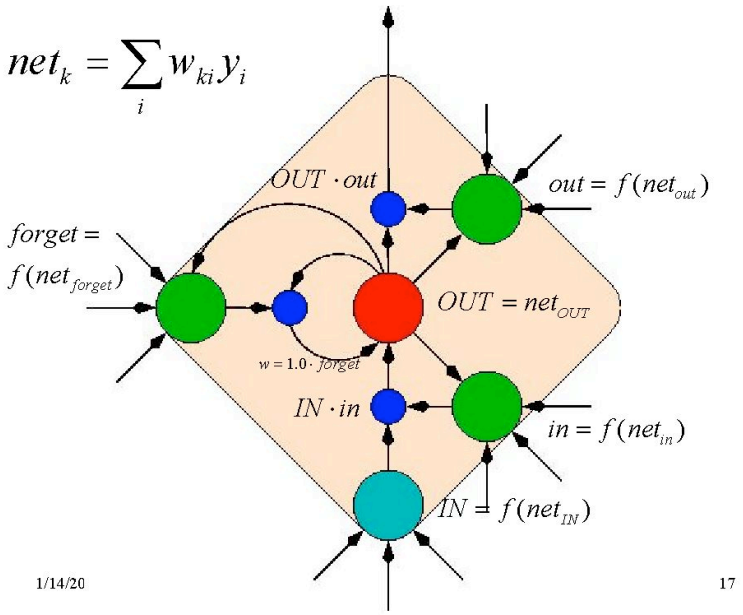


- Derivative of $\frac{\partial}{\partial x_1} x_1 x_2$? → rest is same chain rule, but implement with **modularization** or automatic differentiation

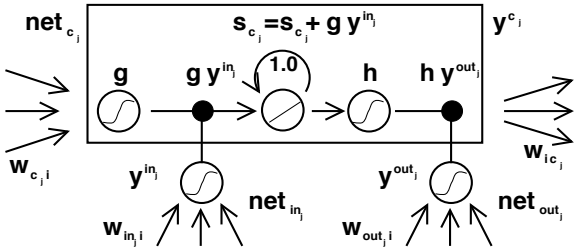
Long-short-term-memories (LSTMs)

- We can make the units even more complex
- Allow each time step to modify
  - Input gate (current cell matters) $i_t = \sigma\left(W^{(i)}x_t + U^{(i)}h_{t-1}\right)$
  - Forget (gate 0, forget past) $f_t = \sigma\left(W^{(f)}x_t + U^{(f)}h_{t-1}\right)$
  - Output (how much cell is exposed) $o_t = \sigma\left(W^{(o)}x_t + U^{(o)}h_{t-1}\right)$
  - New memory cell $\tilde{c}_t = \tanh\left(W^{(c)}x_t + U^{(c)}h_{t-1}\right)$
- Final memory cell: $c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t$
- Final hidden state: $h_t = o_t \circ \tanh(c_t)$

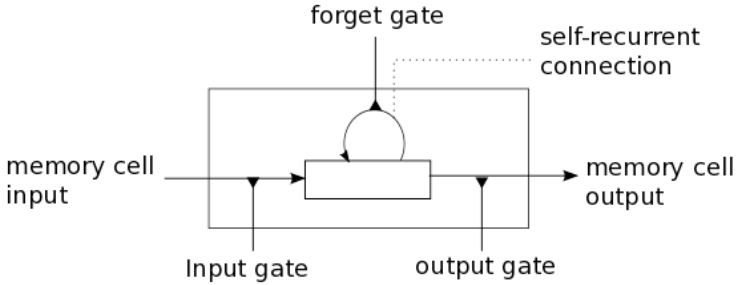
Illustrations a bit overwhelming ;)



http://people.idsia.ch/~juergen/lstm/sld017.htm



Long Short-Term Memory by Hochreiter and Schmidhuber (1997)



http://deeplearning.net/tutorial/lstm.html

Intuition: memory cells can keep information intact, unless inputs makes them forget it or overwrite it with new input.
Cell can decide to output this information or just store it

LSTMs are currently very hip!

- En vogue default model for most sequence labeling tasks
- Very powerful, especially when stacked and made even deeper (each hidden layer is already computed by a deep internal network)
- Most useful if you have lots and lots of data

Deep LSTMs don't outperform traditional MT yet

Method	test BLEU score (ntst14)
Bahdanau et al. [2]	28.45
Baseline System [29]	33.30
Single forward LSTM, beam size 12	26.17
Single reversed LSTM, beam size 12	30.59
Ensemble of 5 reversed LSTMs, beam size 1	33.00
Ensemble of 2 reversed LSTMs, beam size 12	33.27
Ensemble of 5 reversed LSTMs, beam size 2	34.50
Ensemble of 5 reversed LSTMs, beam size 12	**34.81**

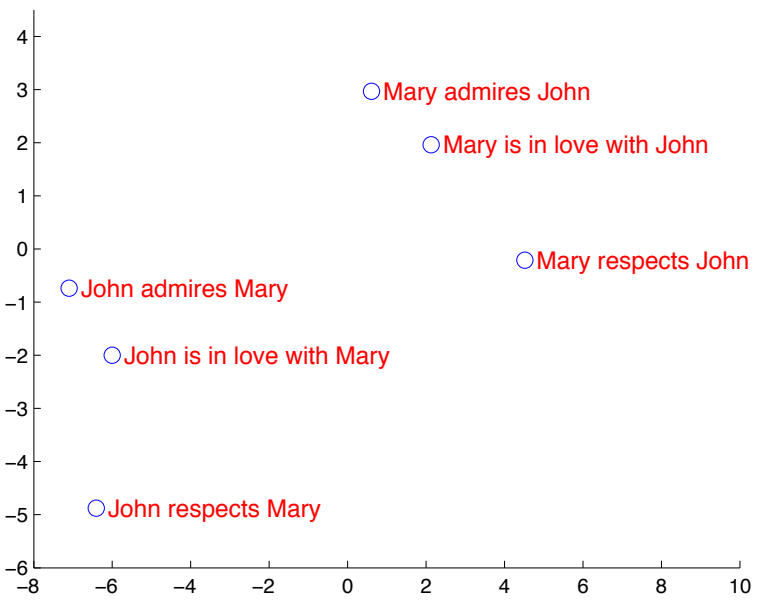
Table 1: The performance of the LSTM on WMT'14 English to French test set (ntst14). Note that an ensemble of 5 LSTMs with a beam of size 2 is cheaper than of a single LSTM with a beam of size 12.

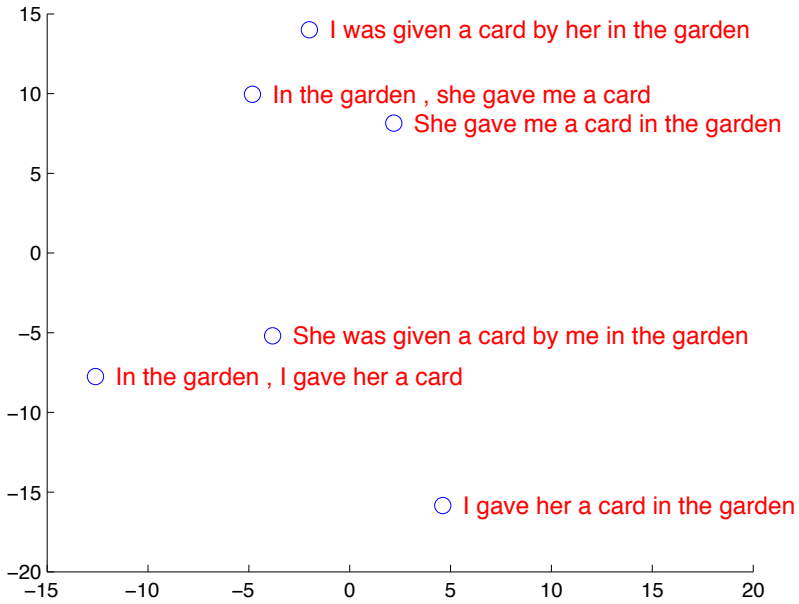
Method	test BLEU score (ntst14)
Baseline System [29]	33.30
Cho et al. [5]	34.54
Best WMT'14 result [9]	**37.0**
Rescoring the baseline 1000-best with a single forward LSTM	35.61
Rescoring the baseline 1000-best with a single reversed LSTM	35.85
Rescoring the baseline 1000-best with an ensemble of 5 reversed LSTMs	**36.5**
Oracle Rescoring of the Baseline 1000-best lists	~45

Sequence to Sequence Learning by Sutskever et al. 2014

Deep LSTM for Machine Translation

PCA of vectors from last time step hidden layer

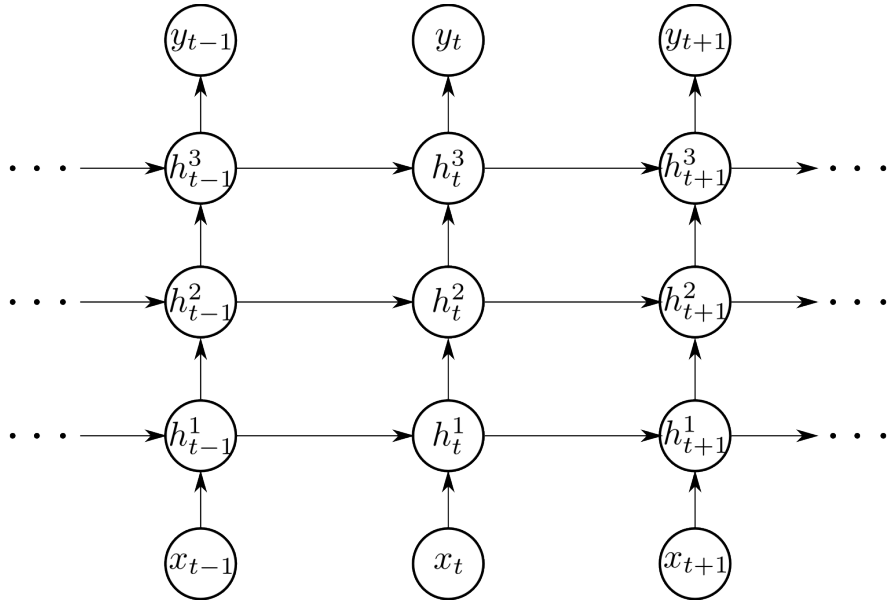




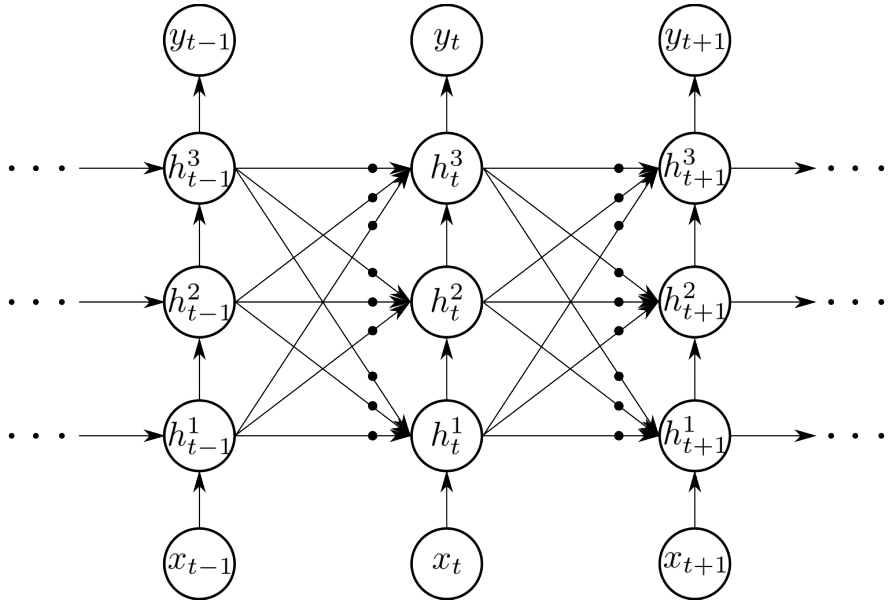
Sequence to Sequence Learning by Sutskever et al. 2014

Further Improvements: More Gates!

Gated Feedback Recurrent Neural Networks, Chung et al. 2015



(a) Conventional stacked RNN



(b) Gated Feedback RNN

Summary

- Recurrent Neural Networks are powerful
- Gated Recurrent Units even better
- LSTMs maybe even better (jury still out)

- A lot of ongoing work right now

- Next lecture: Putting it all together for fun applications and **dynamic memory networks**

The vanishing gradient problem - Details

- Similar but simpler RNN formulation:

$$\begin{aligned} h_t &= W f(h_{t-1}) + W^{(hx)} x_{[t]} \\ \hat{y}_t &= W^{(S)} f(h_t) \end{aligned}$$

- Total error is the sum of each error at time steps t

$$\frac{\partial E}{\partial W} = \sum_{t=1}^{T} \frac{\partial E_t}{\partial W}$$

- Hardcore chain rule application:

$$\frac{\partial E_t}{\partial W} = \sum_{k=1}^{t} \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial W}$$

The vanishing gradient problem - Details

- Similar to backprop but less efficient formulation
- Useful for analysis, we'll look at:

$$\frac{\partial E_t}{\partial W} = \sum_{k=1}^{t} \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial W}$$

- Remember: $h_t = W f(h_{t-1}) + W^{(hx)} x_{[t]}$
- More chain rule, remember:

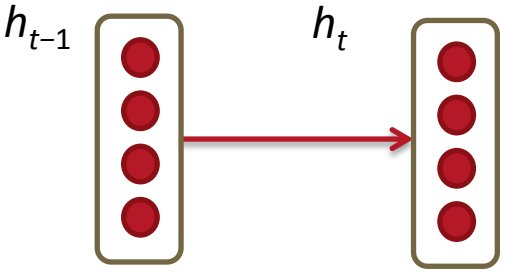
$$\frac{\partial h_t}{\partial h_k} = \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}}$$

- Each partial is a Jacobian:

$$\frac{d\mathbf{f}}{d\mathbf{x}} = \left[\frac{\partial \mathbf{f}}{\partial x_1} \quad \cdots \quad \frac{\partial \mathbf{f}}{\partial x_n} \right] = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

The vanishing gradient problem - Details

- From previous slide: $\frac{\partial h_t}{\partial h_k} = \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}}$



- Remember: $h_t = W f(h_{t-1}) + W^{(hx)} x_{[t]}$

- To compute Jacobian, derive each element of matrix: $\frac{\partial h_{j,m}}{\partial h_{j-1,n}}$

$$\frac{\partial h_t}{\partial h_k} = \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} = \prod_{j=k+1}^{t} W^T \mathrm{diag}[f'(h_{j-1})]$$

- Where: $\mathrm{diag}(z) = \begin{pmatrix} z_1 & & & \\ & z_2 & & 0 \\ & & \ddots & \\ & 0 & & z_{n-1} \\ & & & & z_n \end{pmatrix}$

Check at home that you understand the diag matrix formulation

The vanishing gradient problem - Details

- Analyzing the norms of the Jacobians, yields:

$$\left\| \frac{\partial h_j}{\partial h_{j-1}} \right\| \leq \|W^T\| \|\text{diag}[f'(h_{j-1})]\| \leq \beta_W \beta_h$$

- Where we defined β's as upper bounds of the norms
- The gradient is a product of Jacobian matrices, each associated with a step in the forward computation.

$$\left\| \frac{\partial h_t}{\partial h_k} \right\| = \left\| \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} \right\| \leq (\beta_W \beta_h)^{t-k}$$

- This can become very small or very large quickly [Bengio et al 1994], and the locality assumption of gradient descent breaks down. → **Vanishing or exploding gradient**