CS290D – Advanced Data Mining

Instructor: Xifeng Yan
Computer Science
University of California at Santa Barbara

Recurrent Neural Networks

Lecturer: Semih Yavuz
Computer Science
University of California at Santa Barbara

Source of slides

- Deep Learning for Natural Language Processing Course by Richard Socher - 2015 Stanford
- Using Neural Networks for Modelling and Representing Natural Languages by Mikolov - COLING 2014
- Deep Learning for NLP (without Magic) by Richard Socher - NAACL 2013

Outline

- **Motivation**
- RNN Language Models
- Problems and Tricks
- Applications
  - Other sequence tasks
  - Bidirectional and Deep RNNs

Motivation

Recap: Traditional Language Models

Probabilistic Language Modeling

- Goal: assign a probability to a sentence
  - Machine Translation:
    - P(**high** winds tonight) > P(**large** winds tonight)
  - Spell Correction
    - The office is about fifteen **minuets** from my house
      - P(about fifteen **minutes** from) > P(about fifteen **minuets** from)
  - Speech Recognition
    - P(I saw a van) >> P(eyes awe of an)
  - +Summarization, question answering, etc.

Motivation

Probabilistic Language Modeling

- Goal: compute the probability of a sentence or a sequence of words:

$$P(w_1^m) = P(w_1, w_2, ..., w_m)$$

- How to compute the joint probability?

$$P(a, dog, is, running, in, a, room)$$

- Chain rule:

$$P(w_1, w_2, ..., w_m) = P(w_1)P(w_2 \mid w_1)P(w_3 \mid w_1, w_2)...P(w_m \mid w_1, ...w_{m-1})$$

$$P(a, dog, is, running) =$$

$$P(a)P(dog \mid a)P(is \mid a, dog)P(running \mid a, dog, is)$$

Traditional Language Models

- Probability is usually conditioned on window of n previous words
- An incorrect but necessary Markov assumption!

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

- To estimate probabilities, compute for unigrams and bigrams (conditioning on one/two previous word(s):

$$p(w_2|w_1) = \frac{\text{count}(w_1, w_2)}{\text{count}(w_1)} \qquad p(w_3|w_1, w_2) = \frac{\text{count}(w_1, w_2, w_3)}{\text{count}(w_1, w_2)}$$

Problems with Traditional Language Models

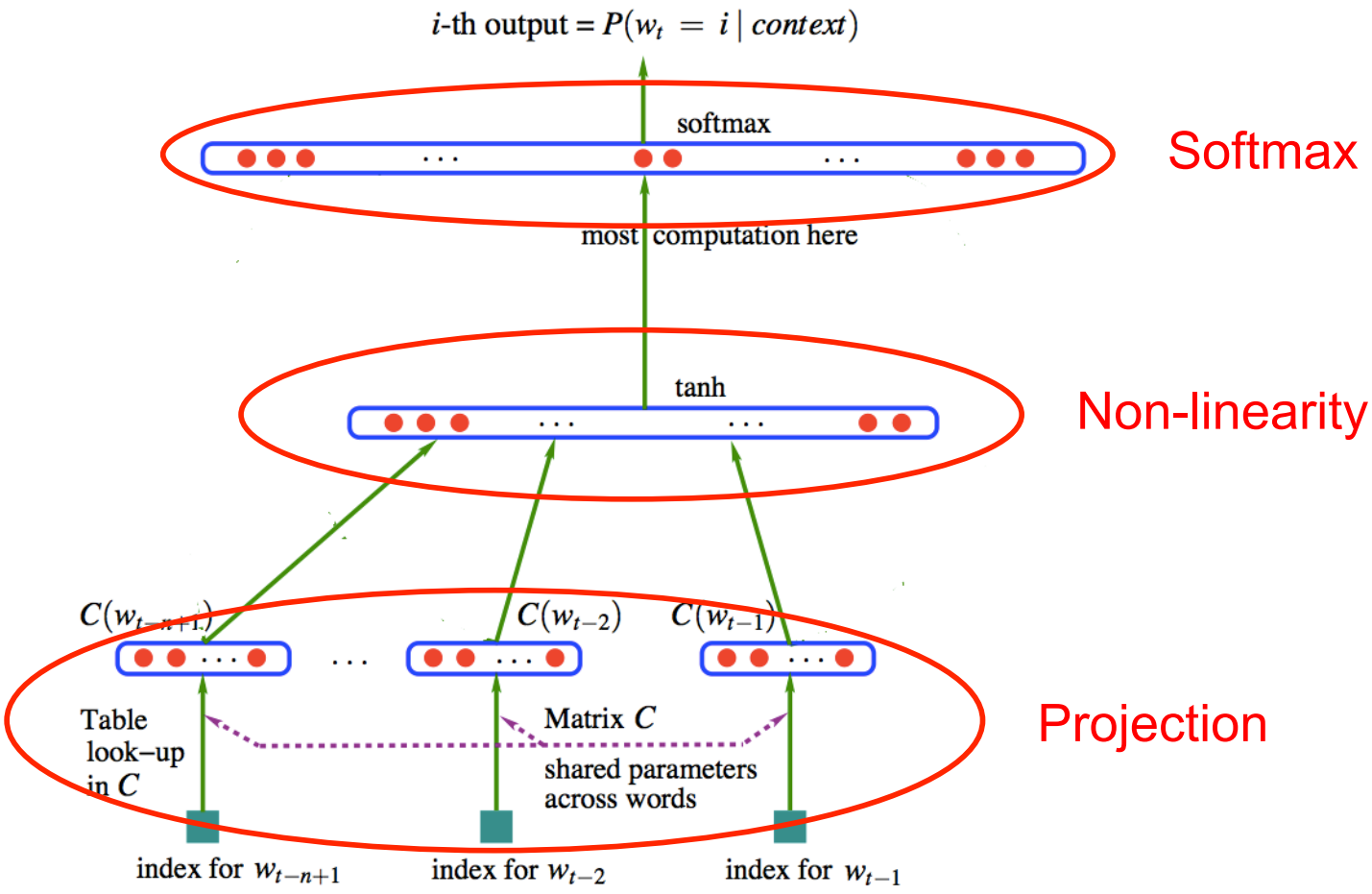
- Performance improves with using higher n-grams counts
- Also, training on a larger corpus leads a better estimation of probabilities
- The number of n-grams grows **exponentially** with the length of context
- It becomes impossible to keep n-gram counts around as the corpus gets larger
- Giant RAM requirement
- So, small context window is inevitable
- Long distance dependencies cannot be supported
- Not utilizing word similarity

Language Modeling with Neural Networks

- Neural Networks address the main problem of traditional language modeling by performing **dimensionality reduction** and **parameter sharing**

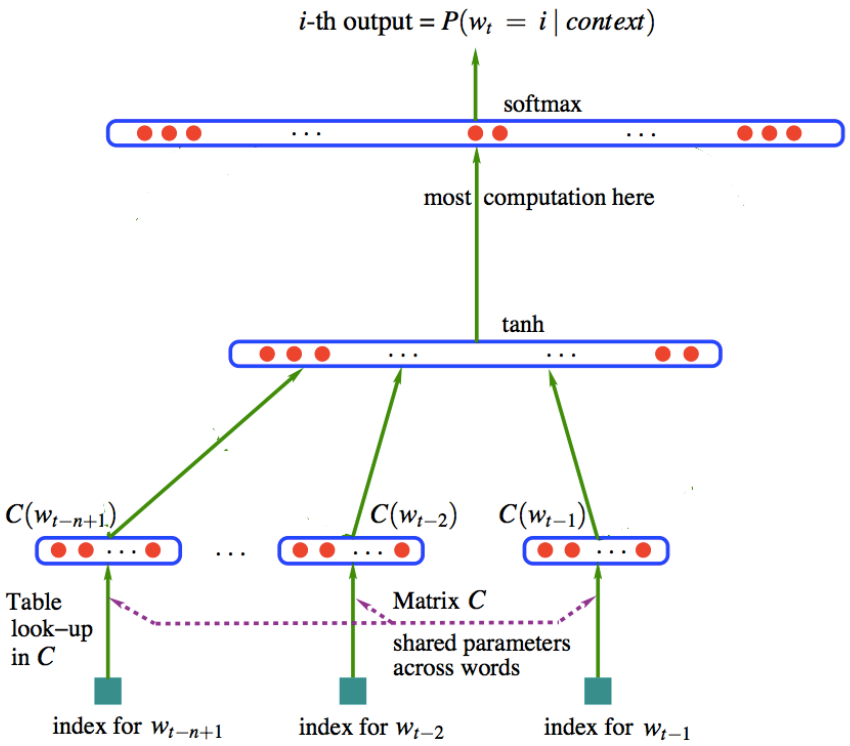
- Two main neural network architectures for language modeling:
  - Feedforward
  - Recurrent

Recap: Feedforward Neural Network LM



Recap: Feedforward Neural Network LM

Neural Language Model



$$P(w_t = i) = \frac{\exp(y_i)}{\sum_{j=1}^{D} \exp(y_j)}$$

softmax

$$y = Uz + b_2$$

output

$$z = \tanh(Hx + b_1)$$

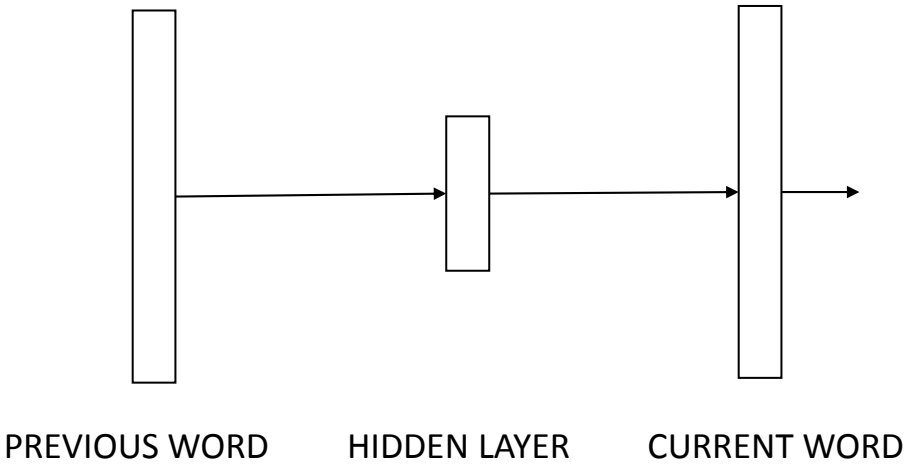
non-linearity

$$x = (Cw_{t-n+1}, Cw_{t-n+2}, \ldots, Cw_{t-1})$$

projection

$$w_{t-n+1}, w_{t-n+2}, \ldots, w_{t-1}$$

Towards Recurrent Neural Network LM



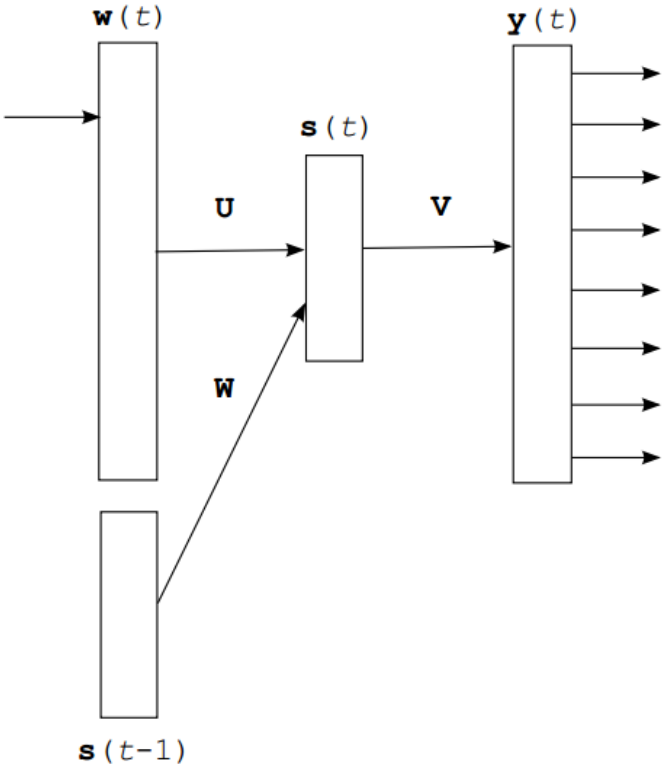
Can we find a better way of representing time than using n-1 previous words as separate inputs ?

Outline

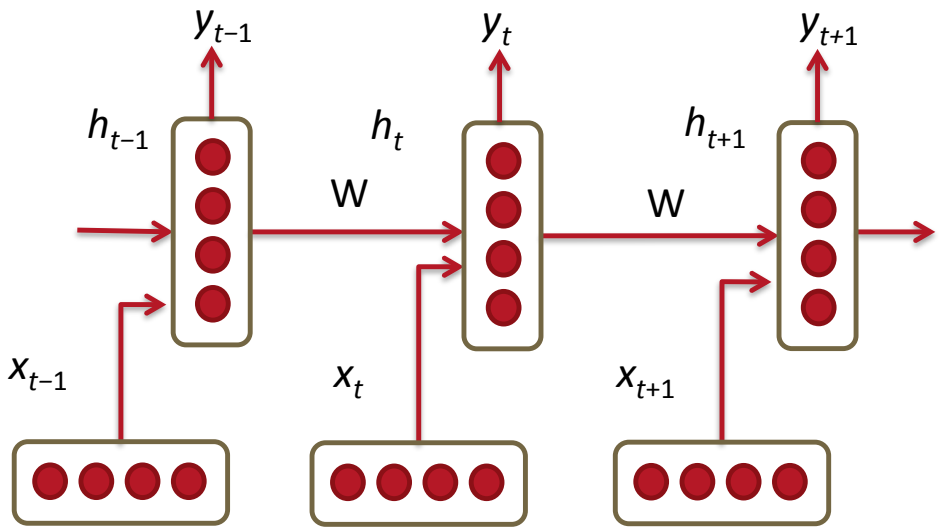
- Motivation
- **RNN Language Models**
- Problems and Tricks
- Applications
  - Other sequence tasks
  - Bidirectional and Deep RNNs

Recurrent Neural Network LM

- RNN is a learner that can operate on sequential data of **variable length**
- History is kept around via the dense **hidden state vectors**.
- Additional weights from the hidden layer in the previous time step
- In theory, the hidden layer can learn to represent unlimited memory



Recurrent Neural Network LM



- RNNs tie the weights at each time step (The same **hidden-hidden** transformation matrix W)
- The decision at the current step is conditioned on all the previous words in Neural Network
- Unlike previous cases, RAM requirement only scales with number of words. NO EXPONENTIAL BLOW-UP!

Recurrent Neural Network LM

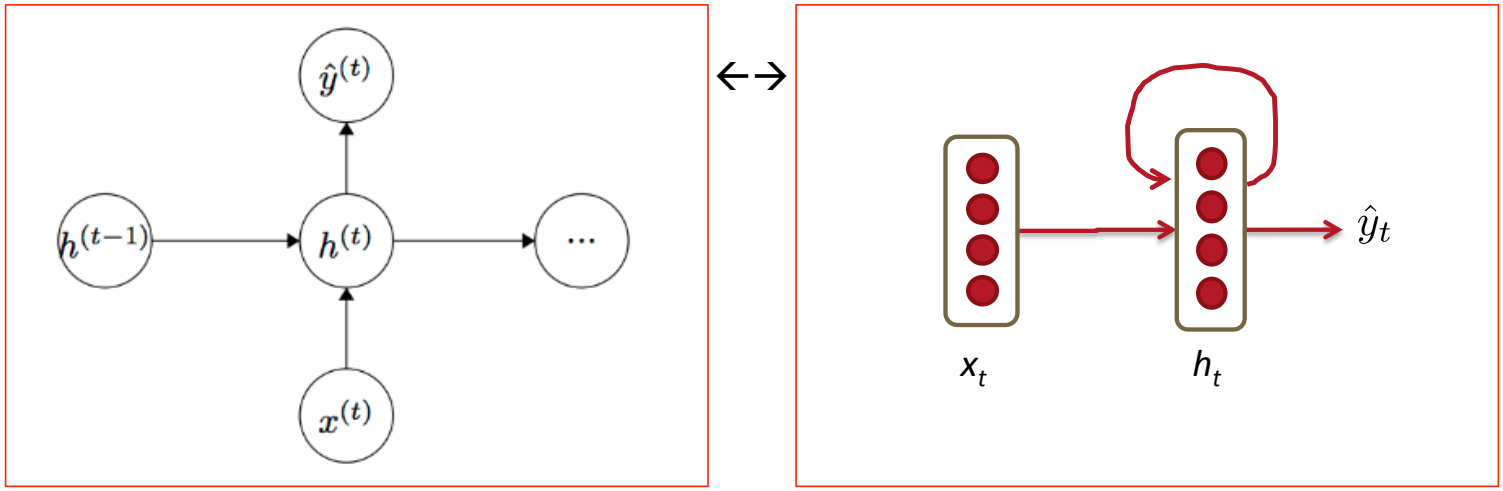
Given list of word **vectors**: $x_1, \ldots, x_{t-1}, x_t, x_{t+1}, \ldots, x_T$

At a single time step:

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

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

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



See the Board!

Recurrent Neural Network LM

- Main Idea: we basically use the same **hidden-hidden** transformation matrix W at all time steps!

- The rest is pretty much the same:

$$
\begin{aligned}
h_t &= \sigma\left(W^{(hh)} h_{t-1} + W^{(hx)} x_{[t]}\right) \\
\hat{y}_t &= \text{softmax}\left(W^{(S)} h_t\right)
\end{aligned}
$$

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

Recurrent Neural Network LM

- Interpretation of the notation:
  - $h_0 \in \mathbb{R}^{D_h}$: Some initialization vector for the hidden layer at time step 0.
  - $x_{[t]}$: d-dimensional column vector of L at index [t] at time step t.
  - $W^{(hh)} \in \mathbb{R}^{D_h \times D_h}$: hidden-to-hidden weight matrix
  - $W^{(hx)} \in \mathbb{R}^{D_h \times d}$: input-to-hidden weight matrix
  - $W^{(S)} \in \mathbb{R}^{|V| \times D_h}$: hidden-to-softmax weight matrix

Recurrent Neural Network LM

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

Loss function at step t: Cross entropy loss

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

Recurrent Neural Network LM

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!

Outline

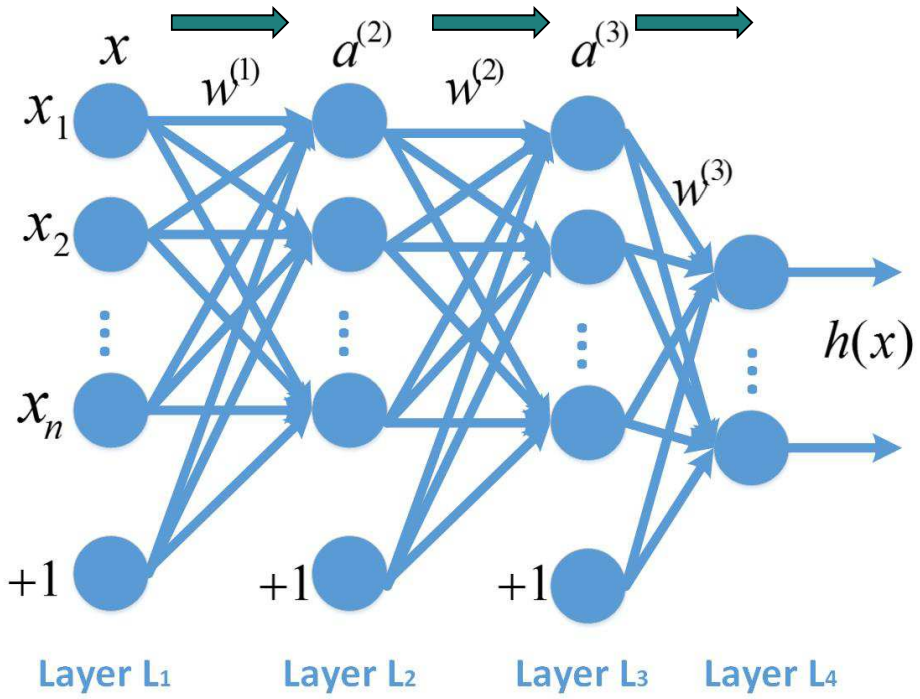
- Motivation
- RNN Language Models
- **Problems and Tricks**
- Applications
  - Other sequence tasks
  - Bidirectional and Deep RNNs

Training of Neural Network Language Models

- **Feedforward** NNLM: The classic SGD + backpropagation
- **Recurrent** NNLM : The same method, but the back propagation part is more difficult to correctly implement!
- The algorithm for computing gradients in RNN is called “Backpropagation Through Time (BPTT)”

Refresher: Backprogagation

- Perform a **feedforward pass**, computing the activations for layers $L2$, $L3$, and so on up to the output layer $h(x)$.



Input and output of 2nd layer:

$$z^{(2)} = w^{(1)}x + b^{(1)}$$

$$a^{(2)} = f(z^{(2)})$$

Input and output of 3rd layer:

$$z^{(3)} = w^{(2)}a^{(2)} + b^{(2)}$$

$$a^{(3)} = f(z^{(3)})$$

Output layer:

$$h(x) = f(w^{(3)}a^{(3)} + \mathrm{b}^{(3)})$$

Refresher: Backprogagation

- Perform a feedforward pass, computing the activations for layers $L2$, $L3$, and so on up to the output layer $h(x)$.

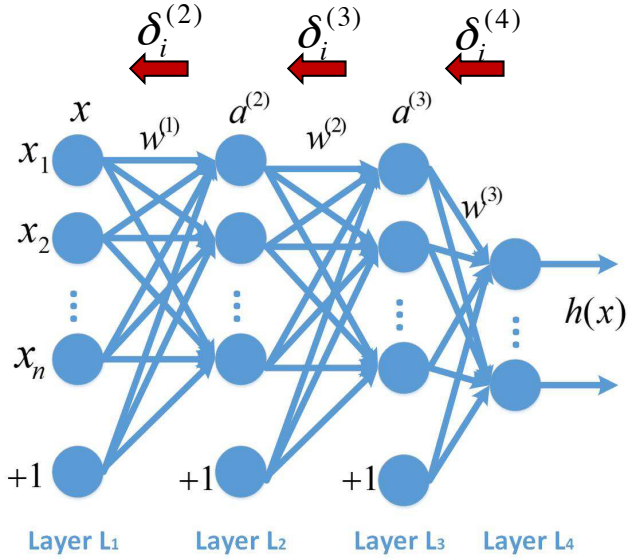
- For each output unit i in the output layer, set

$$\delta_i^{(n_l)} = \frac{\partial}{\partial z_i^{(n_l)}} \frac{1}{2} \left\| y - h(x) \right\|^2 = -(y_i - a_i^{(n_l)}) \cdot f'(z_i^{(n_l)})$$

- For $l = n_l - 1, n_l - 2, \ldots, 2$

  For each node i in layer l, set

$$\delta_i^{(l)} = \left(\sum_{j=1}^{s_l+1} w_{ji}^{(l)} \delta_j^{(l+1)} \right) f'(z_i^{(l)})$$



Refresher: Backprogagation

- Perform a feedforward pass, computing the activations for layers $L2$, $L3$, and so on up to the output layer $h(x)$.

- For each output unit i in the output layer, set

$$\delta_i^{(n_l)} = \frac{\partial}{\partial z_i^{(n_l)}} \frac{1}{2} \left\| y - h(x) \right\|^2 = -(y_i - a_i^{(n_l)}) \cdot f'(z_i^{(n_l)})$$

- For $l = n_l - 1, n_l - 2, \ldots, 2$

  For each node i in layer l, set

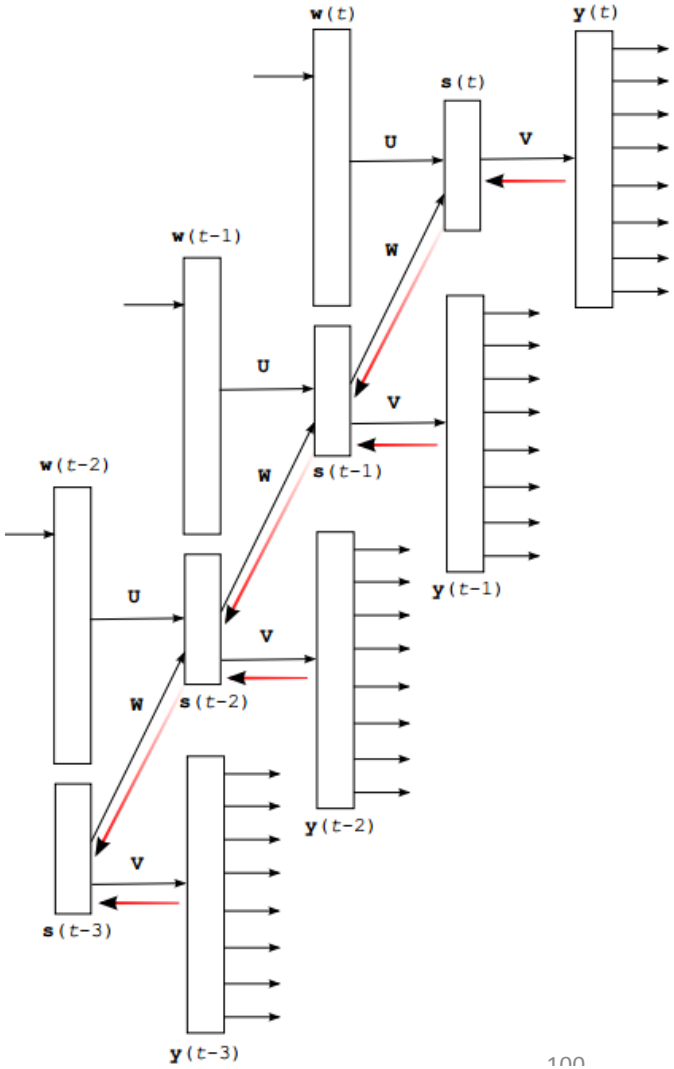
$$\delta_i^{(l)} = (\sum_{j=1}^{s_l+1} w_{ji}^{(l)} \delta_j^{(l+1)}) f'(z_i^{(l)})$$

- Compute the partial derivatives in each layer,

$$\frac{\partial H}{\partial w_{ij}^{(l)}} = a_j^{(l)} \delta_i^{(l+1)} + \lambda \cdot w_{ij}^{(l)} \quad ; \quad \frac{\partial H}{\partial b_i^{(l)}} = \delta_i^{(l+1)}$$

Backpropagation Through Time

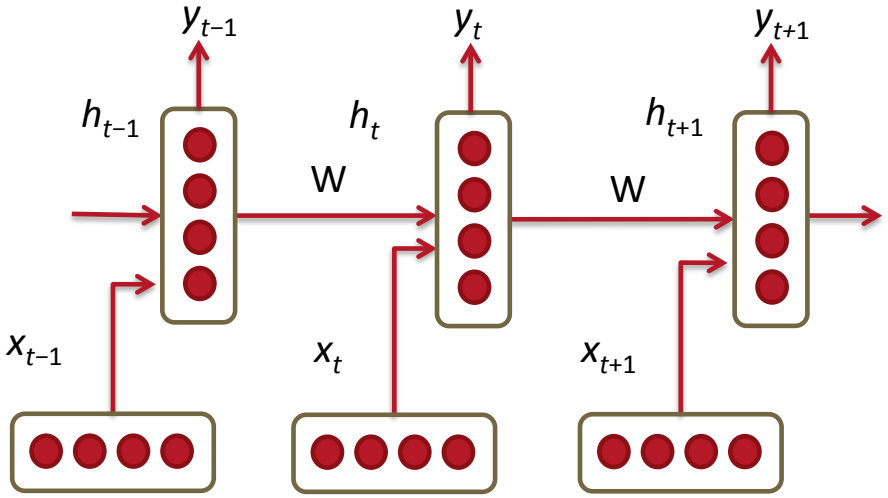
- The **intuition**: we unfold the RNN in time
- Obtain a deep neural network with shared weights **U** and **V**
- Train the unfolded RNN using normal backpropagation + SGD
- In practice, we limit the number of unfolding steps to 5-10



Training RNN is hard!

Training RNN is hard

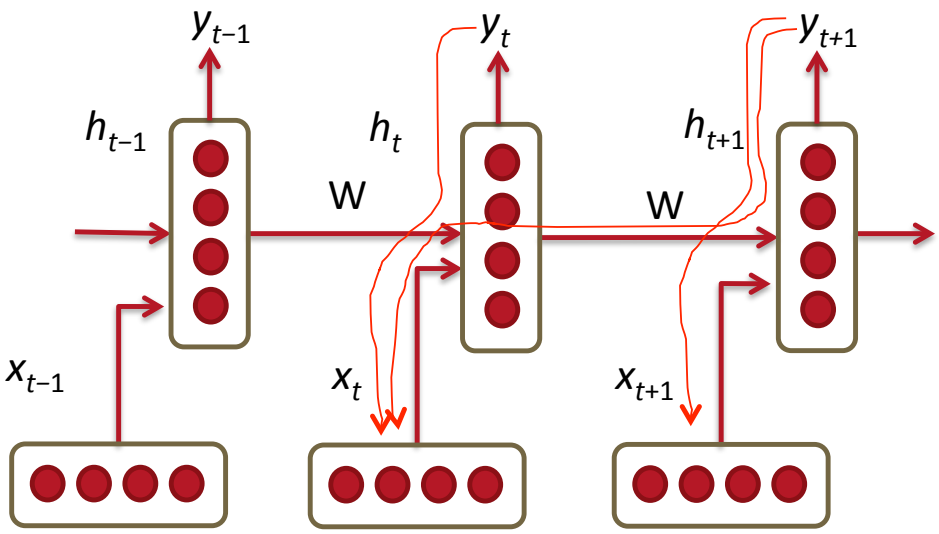
- At each time step during forward prop, we multiply with the same matrix.



- In theory, this model allows **inputs from many time steps ago** to modify output y. However, this makes the training hard!
- Compute $\frac{\partial E_2}{\partial W}$ for an example RNN with 2 time steps for insight.

Problems with Training RNNs (especially with BPTT)

- Multiply the same matrix at each time step during backprop



- Vanishing gradients
- Exploding gradients

The vanishing gradient problem

- **Vanishing Gradients:** As we propagate the gradients back in time, usually the magnitudes of them decrease, and quickly goes to tiny values.

- **What does this mean in practice?**
  - Learning long dependencies is a difficult task!

- There are special architectures to address this problem
  - Long Short-term Memory (LSTM) RNN (Hochreiter & Schmidhuber, 1997)

The exploding gradient problem

- **Exploding Gradients:** The magnitude of gradients sometimes start to **increase exponentially** during back propagation through the shared recurrent weights.

- **What does this mean in practice?**
  - Huge gradients will lead big changes of weights which will cause the network to **forget** almost everything learned so far
  - Although this is a more rare situation, the effect can be much more **catastrophic**
  - If you don't deal with it, you will immediately notice the situation as your gradients will quickly go to a NAN (not a number) in your implementation

The vanishing gradient problem - Details

- Let's try to understand why this problem emerges
- 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:

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

- Apply chain rule:

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

The vanishing gradient problem - Details

- Let's analyze the following from previous slide:

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

- Recall our simpler model:

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

- Some more chain rule:

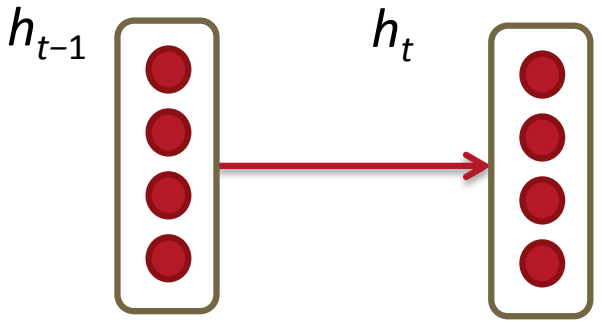
$$\frac{\partial h_t}{\partial h_k} = \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}}$$

- Derivative of vector-valued functions: 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

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



- We need to compute each element $\frac{\partial h_{j,m}}{\partial h_{j-1,n}}$ of Jacobian matrix $\frac{\partial h_j}{\partial h_{j-1}}$
- Remember: $h_t = W f(h_{t-1}) + W^{(hx)} x_{[t]}$
- So: $\frac{\partial h_j}{\partial h_{j-1}} = W^T \text{diag}[f'(h_{j-1})]$
- Finally:

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

The vanishing gradient problem - Details

- Analyzing the norms of the Jacobians, yields:

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

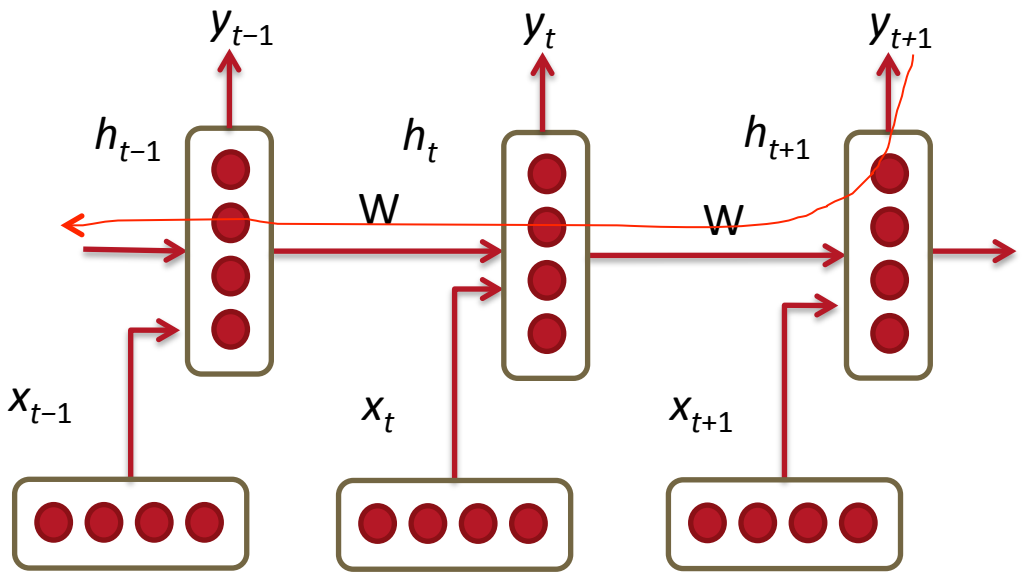
- Where we defined β's as upper bounds of the norms
- The gradient is a product of Jacobian matrices, each associated with a step in the forward computation.

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

- 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**

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 ____

- **In theory,** RNN model should be able to learn to predict the next word as **John**.
- **In practice,** not dealing with vanishing gradients properly would make such learning difficult as the gradient gets very smaller as we go deeper in the network.

IPython Notebook with vanishing gradient example

- Example of simple and clean NNet implementation
- Comparison of sigmoid and ReLu units
- A little bit of vanishing gradient

Trick for exploding gradient: clipping the gradient

- 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

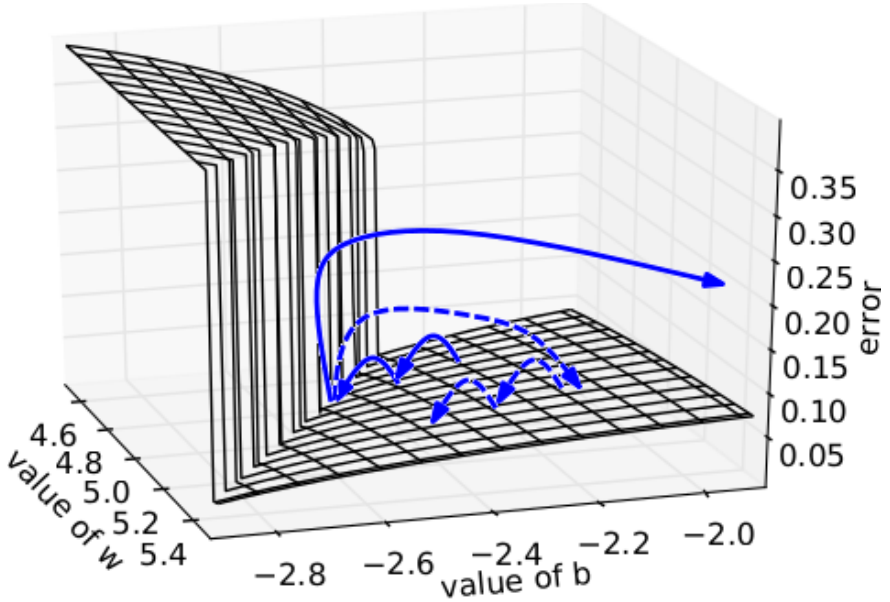
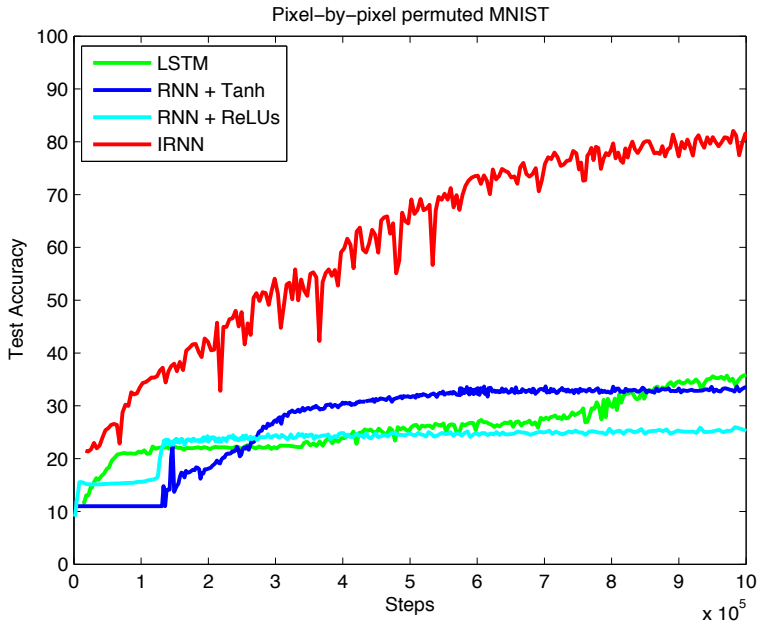


Figure from paper:
On the difficulty of training Recurrent Neural Networks, Pascanu et al. 2013

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

Tricks 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 2 weeks ago (!) 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

- Most computations are at the output layer:
  - At the final prediction step over the entire vocabulary, we have to make a huge computation to obtain the normalization term in softmax
  - The cost is linear to |V|

- Solutions:
  - Negative Sampling (Covered by Yu)
  - **Class-based Softmax**
  - **Hierarchical Softmax**

Problem: Softmax is huge and slow

Trick: Class-based word prediction

- Instead of normalizing probability over all words:
  - Assign each word to single class
  - Normalize over the class layer
  - Normalize over words from within the current class
- Reduces complexity from |V| to about sqrt(|V|)

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

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

Speed-up Experiments with the class-based Softmax

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

The more classes, the better perplexity but also worse speed

Extensions of recurrent neural network language model (Mikolov et al, 2011)

Hierarchical Softmax

- Idea: we can add classes over classes
- Extreme case: binary tree over the whole vocabulary
- Further reduces the complexity of softmax class to about log(|V|)

Hierarchical Probabilistic Neural Network Language Model (Morin & Bengio, 2005)

Hierarchical Softmax - Further Speed-up

- Idea: Use **Huffman encoding** to further increase speed
- Intuition: Frequent words will have short binary codes
- Further reduces the complexity of softmax

Efficient estimation of word representations in vector space (Mikolov, 2013)

How to assign words to classes

- **Frequency binning:** Words are assigned to classes proportionally, while respecting their frequencies.

- The amount of classes is a parameter.

- For example: If we choose 20 classes
  - The words that correspond to the first 5% of the unigram probability distribution would be mapped to class 1
  - The words that correspond to the next 5% of the unigram probability mass would be mapped to class 2, etc.

- Other clustering techniques work too
  - For example: K-means on pre-trained word vectors.

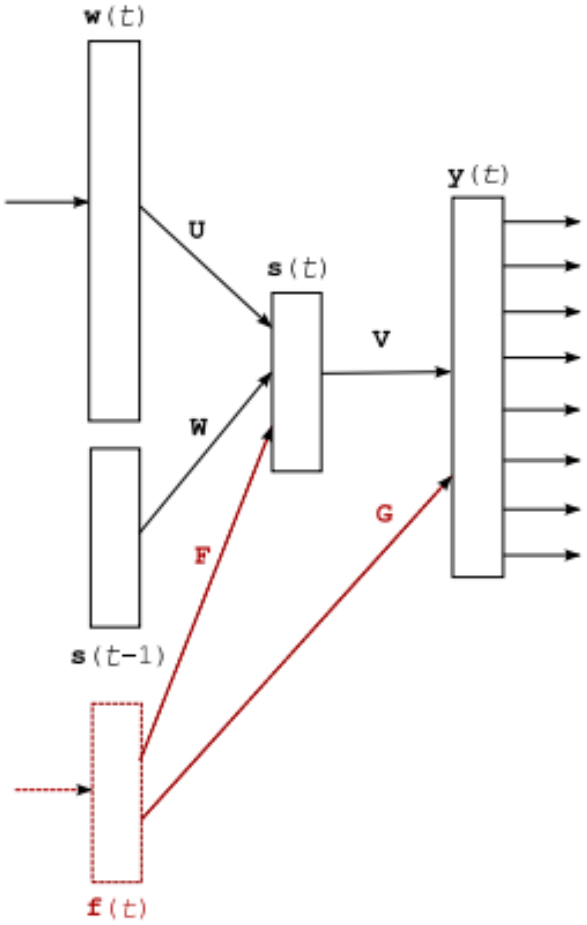
Outline

- Motivation
- RNN Language Models
- **Problems and Tricks**
- **Applications**
  - Other sequence tasks
  - Bidirectional and Deep RNNs

Sequence modeling for other tasks

- Classify
  - NER
  - the sentiment of each word in its context
  - opinionated expressions
- Possibly by incorporating extra features:
  - POS tags
  - external information
  - long context information

represented as additional inputs



Example Application: Opinion Mining with Deep Recurrent Nets

- Classify each word as
  - DES (direct subjective expressions) and ESE (expressive subjective expressions)

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

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

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

Example Annotation

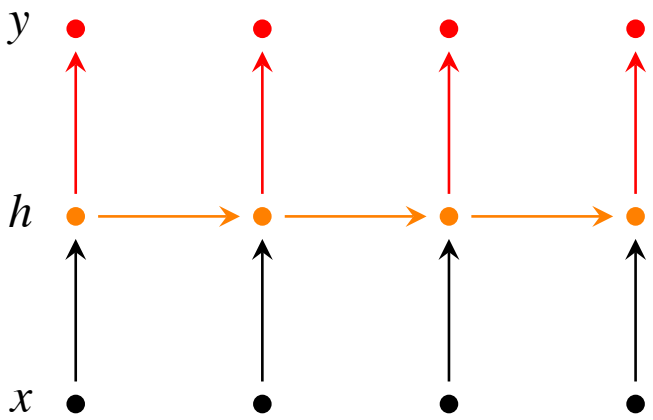
In BIO notation (tags either begin-of-entity (B_*X*) or continuation-of-entity (I_*X*)):
The committee, [as usual]$_{ESE}$, [has refused to make any statements]$_{DSE}$.

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

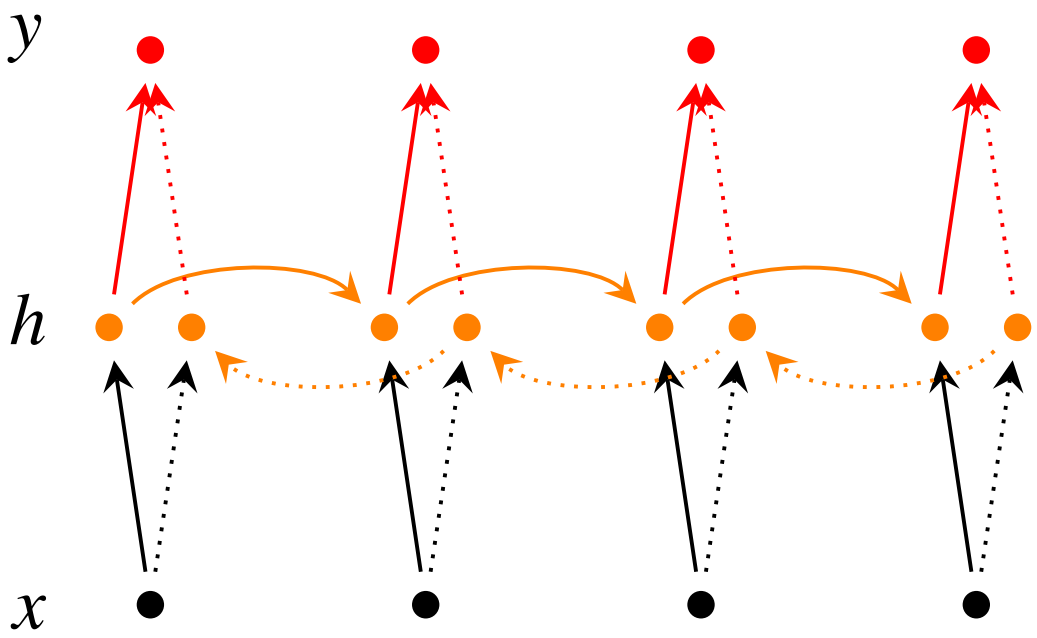


$$h_t = f(Wx_t + Vh_{t-1} + b)$$
$$y_t = g(Uh_t + c)$$

- *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



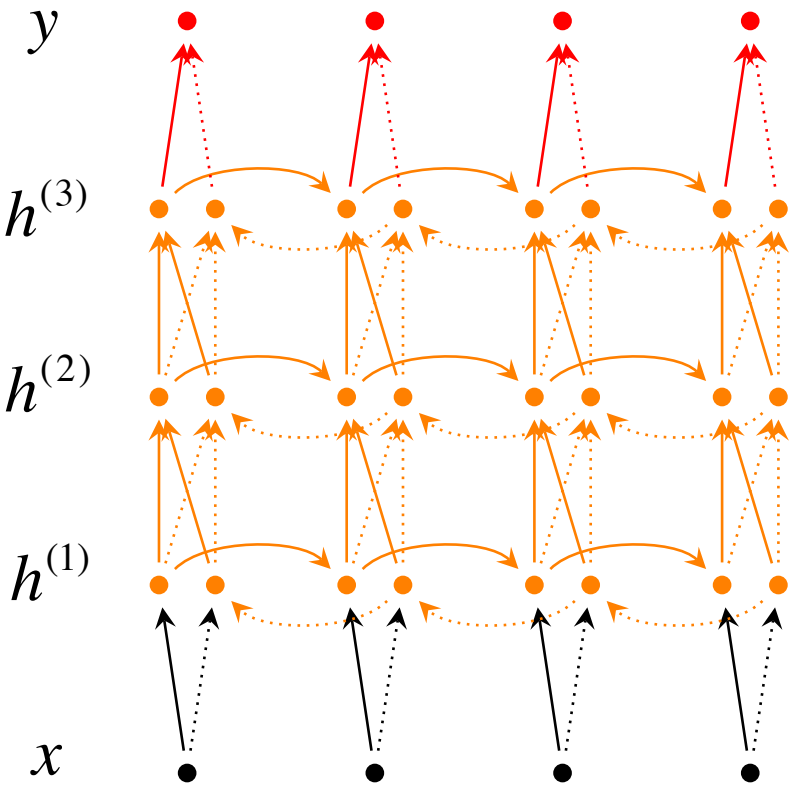
$$\vec{h}_t = f(\vec{W}x_t + \vec{V}\vec{h}_{t-1} + \vec{b})$$

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

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

$h = [\vec{h}; \overleftarrow{h}]$ 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

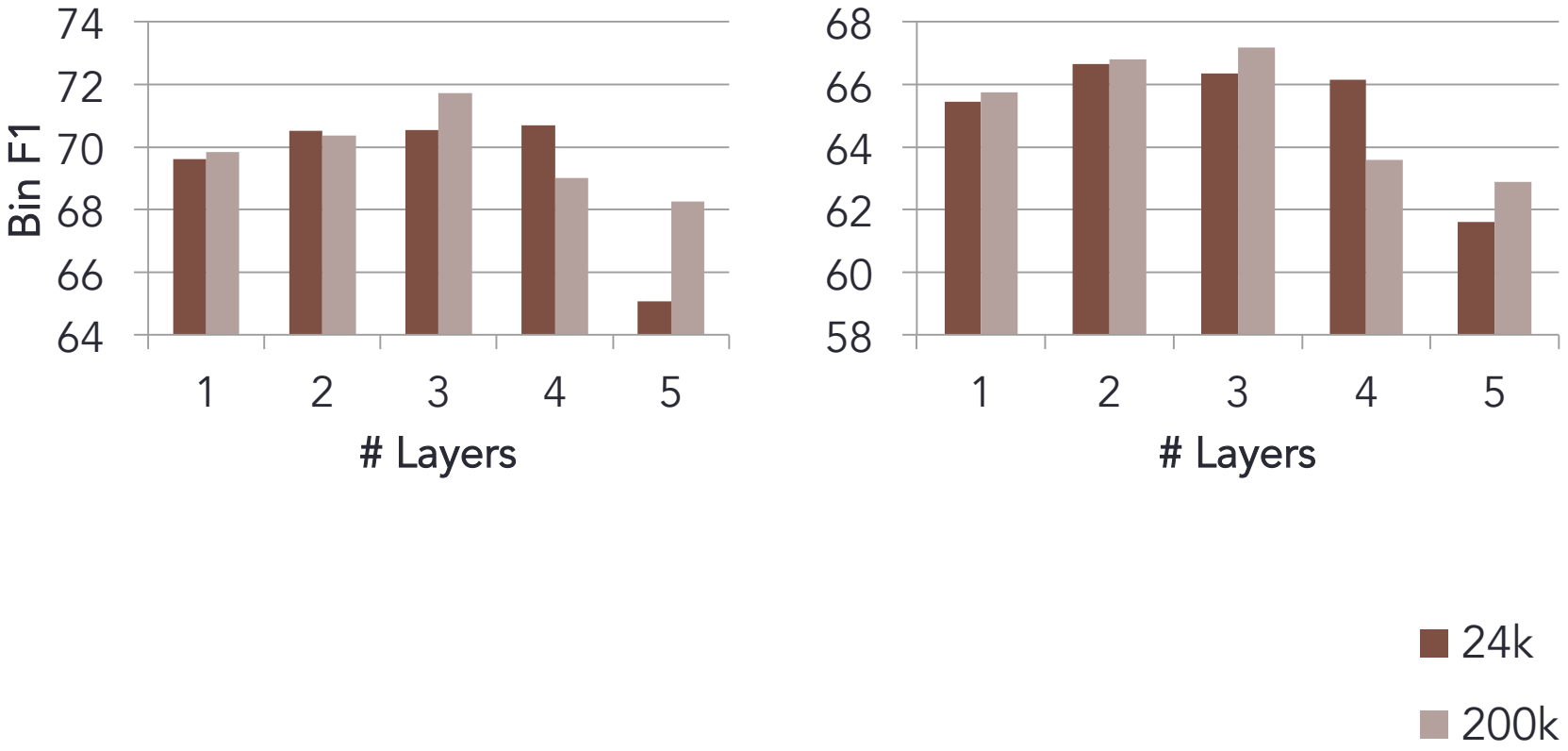
- MPQA 1.2 corpus (Wiebe et al., 2005)
- consists of 535 news articles (11,111 sentences)
- manually labeled with DSE and ESEs at the phrase level
- Evaluation: F1

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

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

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

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

References

1. Recurrent Neural Network Based Language Model (Mikolov *et al.* INTERSPEECH 2010)

2. Extensions of Recurrent Neural Network Based Language Model (Mikolov *et al.* ICAASP 2011)

3. Opinion Mining with Deep Recurrent Neural Networks (Irsoy *et al.* EMNLP 2014)

Outline

- Motivation
- RNN Language Models
- Problems and Tricks
- Applications
  - Other sequence tasks
  - Bidirectional and Deep RNNs
- **Advanced RNNs**
  - Gated Recurrent Units

Gated Recurrent Units (GRUs)

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

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

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

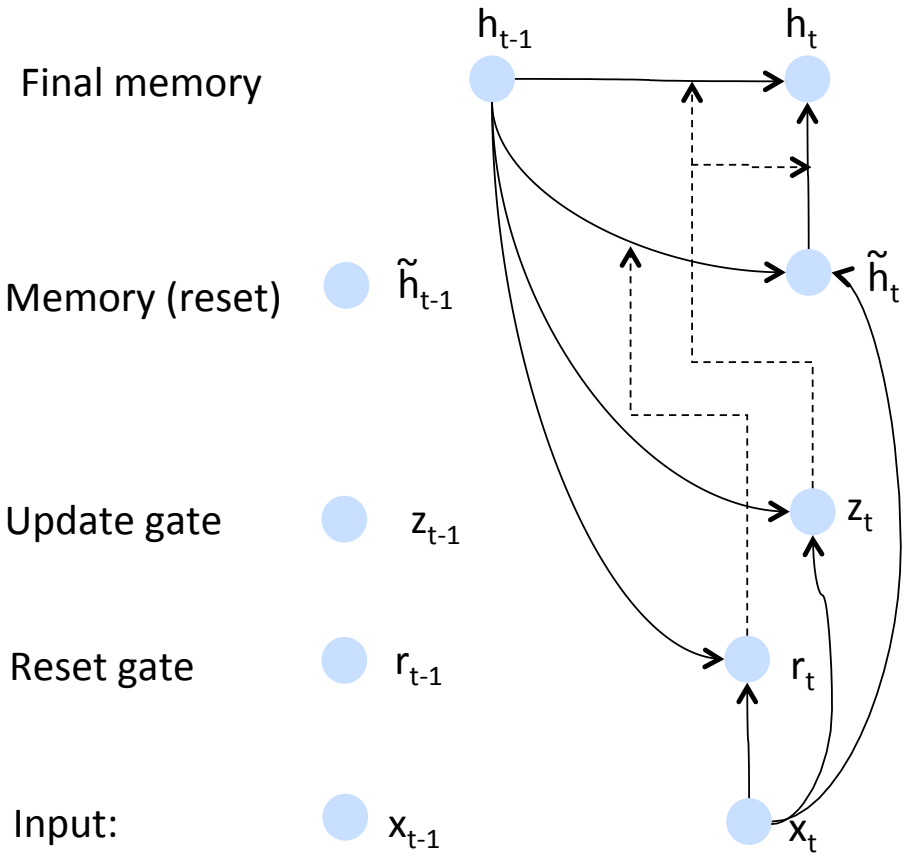
- Compute reset gate similarly but with different weights

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

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

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

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

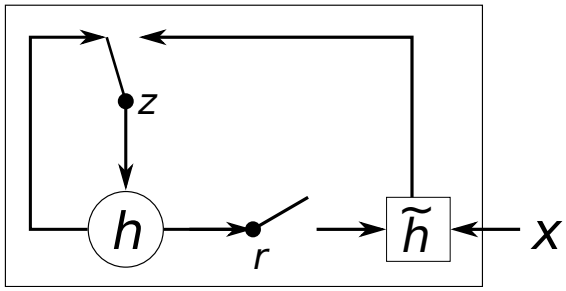
- Update gate z controls how much of past state should matter now.
  - If z close to 1, then we can copy information in that unit through many time steps! **Less vanishing gradient**!
- Units with short-term dependencies often have reset gates very active

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

Summary

- Recurrent Neural Networks are powerful
- A lot of ongoing work right now
- Gated Recurrent Units even better
- LSTMs maybe even better (jury still out)

Note: Pointers to LSTMS (not covered in this class, but important) in further reading list!

Further Readings:

1. Emprical Evaluations of Gated Recurrent Neural Networks on Sequence Modeling (Chung *et al.* 2014)

2. Gated Feedback Recurrent Neural Networks (Chung *et al.* 2015)

3. Long Short-Term Memory by (Hochreiter *et al. 1997)*

4. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks (Tai *et al. 2015*)