Recurrent Neural Network for Language Modeling Task

Tsuyoshi Okita

Ludwig-Maximilian-Universitat Munich

12 December 2016

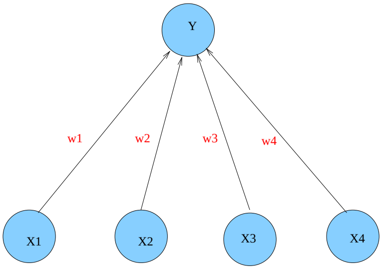
Credit: Slides by Kyunghun Cho / Kevin Duh / Richard Socher

Outline

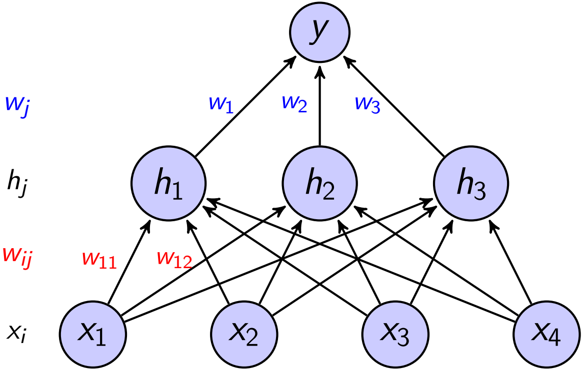
- Review
- Language Modeling Task
- Feedforward Language Model
- Recurrent Neural Network Language Model
- LSTM / GRU
- Vanishing / Exploding Gradient

Logistic Regression (1-layer Neural Networks)

- $f(x) = \sigma(w^T \cdot x)$
  - $\sigma(z) = \frac{1}{(1+\exp(-z))}$: activation function (non linearity)
  - $w(\in \mathcal{R}^d)$: weight



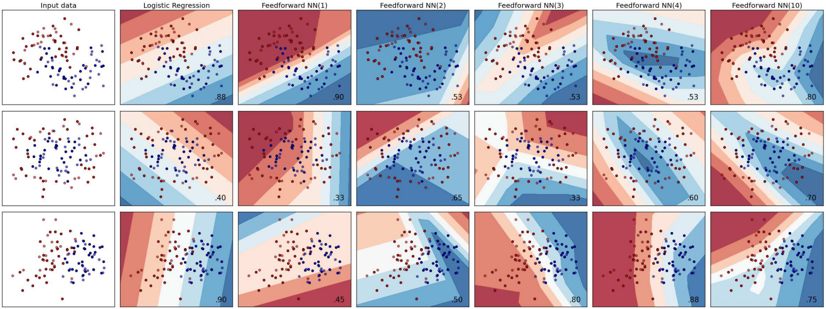
2-layer Neural Networks



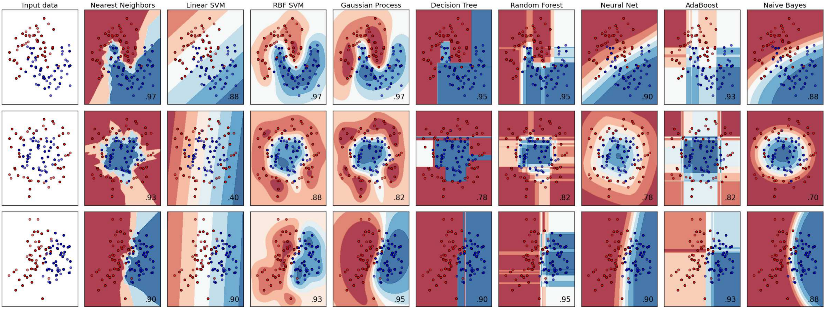
$$f(x) = \sigma(\sum_j w_j \cdot h_j) = \sigma(\sum_j w_j \cdot \sigma(\sum_i w_{ij} x_i))$$

Called Multilayer Perceptron (MLP), but more like multilayer logistic regression

Classification

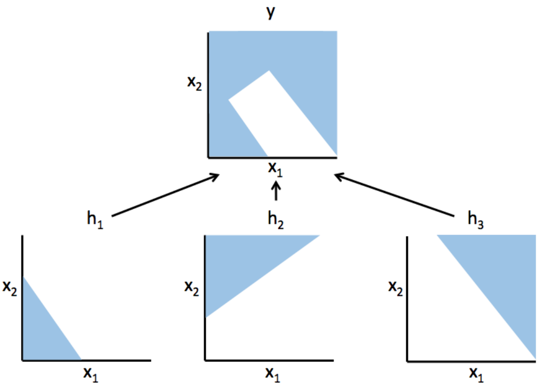


Classification (cont)



Expressive Power of Non-linearity

- A deeper architecture is more expressive than a shallow one given same number of nodes [Bishop, 1995]
  - 1-layer nets only model linear hyperplanes
  - 2-layer nets can model any continuous function (given sufficient nodes)
  - $>$3-layer nets can do so with fewer nodes



Gradient Descent for Logistic Regression

- Assume Squared-Error*
  $Loss(w) = \frac{1}{2} \sum_m (\sigma(w^T x^{(m)}) - y^{(m)})^2$
- Gradient:
  $\nabla_w Loss = \sum_m \left[\sigma(w^T x^{(m)}) - y^{(m)}\right] \sigma'(w^T x^{(m)}) x^{(m)}$
  - Define input into non-linearity $in^{(m)} = w^T x^{(m)}$
  - General form of gradient: $\sum_m Error^{(m)} * \sigma'(in^{(m)}) * x^{(m)}$
  - Derivative of sigmoid $\sigma'(z) = \sigma(z)(1 - \sigma(z))$
- Gradient Descent Algorithm:
  1. Initialize w randomly
  2. Update until convergence: $w \leftarrow w - \gamma(\nabla_w Loss)$
- Stochastic gradient descent (SGD) algorithm:
  1. Initialize w randomly
  2. Update until convergence:
     $w \leftarrow w - \gamma(Error^{(m)} * \sigma'(in^{(m)}) * x^{(m)})$

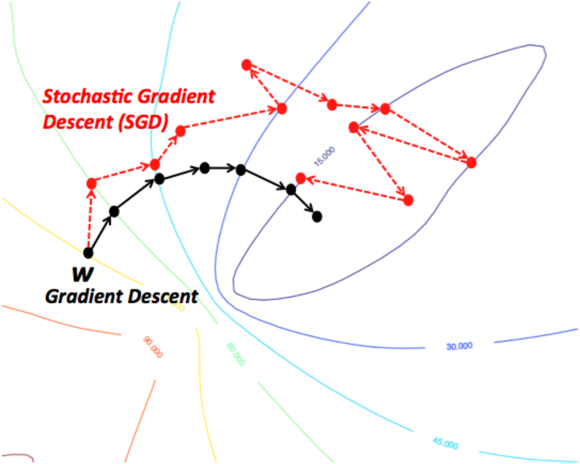
*An alternative is Cross-Entropy loss:
$\sum_m y^{(m)} \log(\sigma(w^T x^{(m)})) + (1 - y^{(m)}) \log(1 - \sigma(w^T x^{(m)}))$

Stochastic Gradient Descent (SGD)

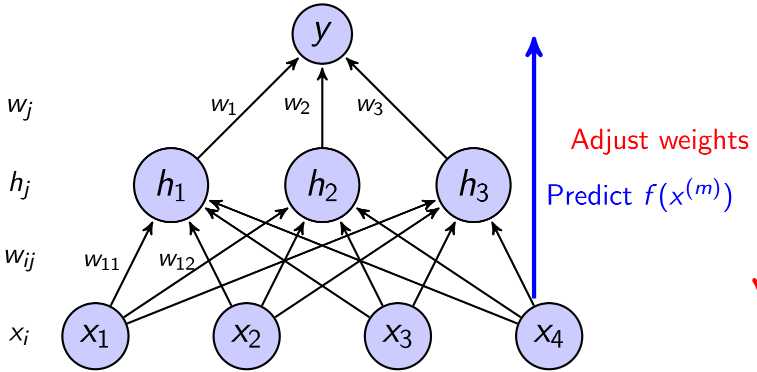
- Gradient Descent Algorithm:
  1. Initialize w randomly
  2. Update until convergence: $w \leftarrow w - \gamma(\nabla_w Loss)$
- Stochastic gradient descent (SGD) algorithm:
  1. Initialize w randomly
  2. Update until convergence:
     $w \leftarrow w - \gamma(\frac{1}{|B|}\sum_{m \in B} Error^{(m)} * \sigma'(in^{(m)}) * x^{(m)})$
     where minibatch B ranges from e.g. 1-100 samples
- Learning rate γ:
  - For convergence, should decrease with each iteration t through samples
  - e.g. $\gamma_t = \frac{1}{\lambda * t}$ or $\gamma_t = \frac{\gamma_0}{1+\gamma_0 * \lambda * t}$

SGD Pictorial View

- Loss objective contour plot:
  $\frac{1}{2}\sum_m(\sigma(w^T x^{(m)}) - y^{(m)})^2 + ||w||$
  - Gradient descent goes in steepest descent direction
  - SGD is noisy descent (but faster per iteration)



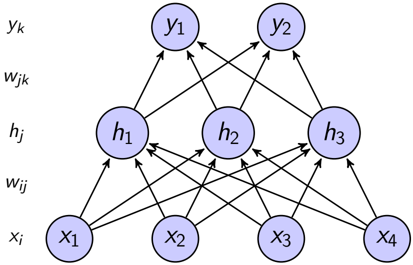
Training Neural Nets: Back-propagation



1. For each sample, compute $f(x^{(m)}) = \sigma(\sum_j w_j \cdot \sigma(\sum_i w_{ij} x_i^{(m)}))$
2. If $f(x^{(m)}) \neq y^{(m)}$, back-propagate error and adjust weights $\{w_{ij}, w_j\}$.

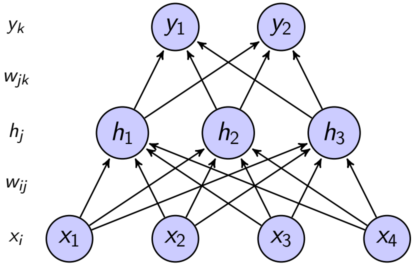
Derivatives of the weights

Assume two outputs (y_1, y_2) per input x,
and loss per sample: $Loss = \sum_k \frac{1}{2} \left[\sigma(in_k) - y_k\right]^2$



Derivatives of the weights

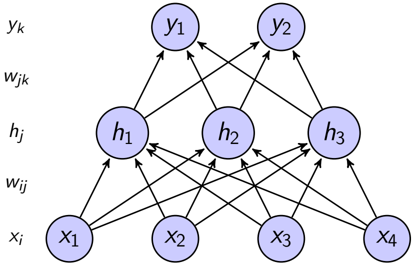
Assume two outputs (y_1, y_2) per input x,
and loss per sample: $Loss = \sum_k \frac{1}{2}\left[\sigma(in_k) - y_k\right]^2$



$$\frac{\partial Loss}{\partial w_{jk}} = \frac{\partial Loss}{\partial in_k}\frac{\partial in_k}{\partial w_{jk}} = \delta_k \frac{\partial(\sum_j w_{jk} h_j)}{\partial w_{jk}} = \delta_k h_j$$

Derivatives of the weights

Assume two outputs (y_1, y_2) per input x,
and loss per sample: $Loss = \sum_k \frac{1}{2}\left[\sigma(in_k) - y_k\right]^2$

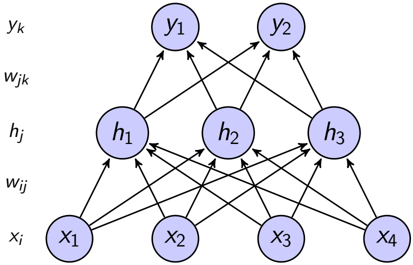


$$\frac{\partial Loss}{\partial w_{jk}} = \frac{\partial Loss}{\partial in_k}\frac{\partial in_k}{\partial w_{jk}} = \delta_k \frac{\partial(\sum_j w_{jk} h_j)}{\partial w_{jk}} = \delta_k h_j$$

$$\frac{\partial Loss}{\partial w_{ij}} = \frac{\partial Loss}{\partial in_j}\frac{\partial in_j}{\partial w_{ij}} = \delta_j \frac{\partial(\sum_j w_{ij} x_i)}{\partial w_{ij}} = \delta_j x_i$$

Derivatives of the weights

Assume two outputs (y_1, y_2) per input x,
and loss per sample: $Loss = \sum_k \frac{1}{2}\left[\sigma(in_k) - y_k\right]^2$



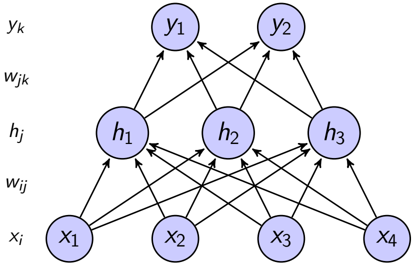
$$\frac{\partial Loss}{\partial w_{jk}} = \frac{\partial Loss}{\partial in_k}\frac{\partial in_k}{\partial w_{jk}} = \delta_k \frac{\partial(\sum_j w_{jk} h_j)}{\partial w_{jk}} = \delta_k h_j$$

$$\frac{\partial Loss}{\partial w_{ij}} = \frac{\partial Loss}{\partial in_j}\frac{\partial in_j}{\partial w_{ij}} = \delta_j \frac{\partial(\sum_j w_{ij} x_i)}{\partial w_{ij}} = \delta_j x_i$$

$$\delta_k = \frac{\partial}{\partial in_k}\left(\sum_k \frac{1}{2}\left[\sigma(in_k) - y_k\right]^2\right) = \left[\sigma(in_k) - y_k\right]\sigma'(in_k)$$

Derivatives of the weights

Assume two outputs (y_1, y_2) per input x,
and loss per sample: $Loss = \sum_k \frac{1}{2}\left[\sigma(in_k) - y_k\right]^2$



$$\frac{\partial Loss}{\partial w_{jk}} = \frac{\partial Loss}{\partial in_k}\frac{\partial in_k}{\partial w_{jk}} = \delta_k \frac{\partial(\sum_j w_{jk} h_j)}{\partial w_{jk}} = \delta_k h_j$$

$$\frac{\partial Loss}{\partial w_{ij}} = \frac{\partial Loss}{\partial in_j}\frac{\partial in_j}{\partial w_{ij}} = \delta_j \frac{\partial(\sum_j w_{ij} x_i)}{\partial w_{ij}} = \delta_j x_i$$

$$\delta_k = \frac{\partial}{\partial in_k}\left(\sum_k \frac{1}{2}\left[\sigma(in_k) - y_k\right]^2\right) = \left[\sigma(in_k) - y_k\right]\sigma'(in_k)$$

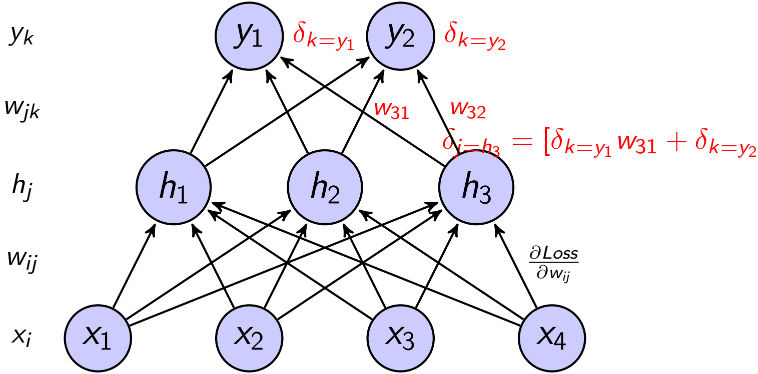
$$\delta_j = \sum_k \frac{\partial Loss}{\partial in_k}\frac{\partial in_k}{\partial in_j} = \sum_k \delta_k \cdot \frac{\partial}{\partial in_j}\left(\sum_j w_{jk}\sigma(in_j)\right) = \left[\sum_k \delta_k w_{jk}\right]\sigma'(in_j)$$

Training Neural Nets: Back-propagation

All updates involve some scaled error from output $*$ input feature:

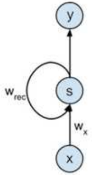
- $\frac{\partial Loss}{\partial w_{jk}} = \delta_k h_j$ where $\delta_k = [\sigma(in_k) - y_k]\, \sigma'(in_k)$
- $\frac{\partial Loss}{\partial w_{ij}} = \delta_j x_i$ where $\delta_j = [\sum_k \delta_k w_{jk}]\, \sigma'(in_j)$

First compute δ_k from final layer, then δ_j for previous layer and iterate.

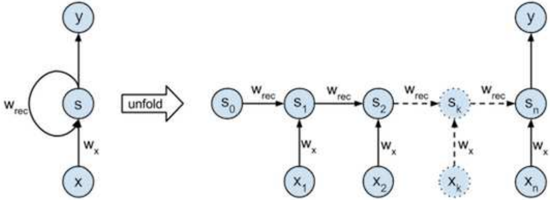


Recurrent Neural Networks (Today's Lecture)

- Recurrent Neural Networks (RNN): a recurrent Layer is defined.



- We want to treat RNN like feed-forward NN!
  - To unfold the recursion.



Outline

- Review
- Language Modeling Task
- Feedforward Language Model
- Recurrent Neural Network Language Model
- LSTM / GRU
- Vanishing / Exploding Gradient

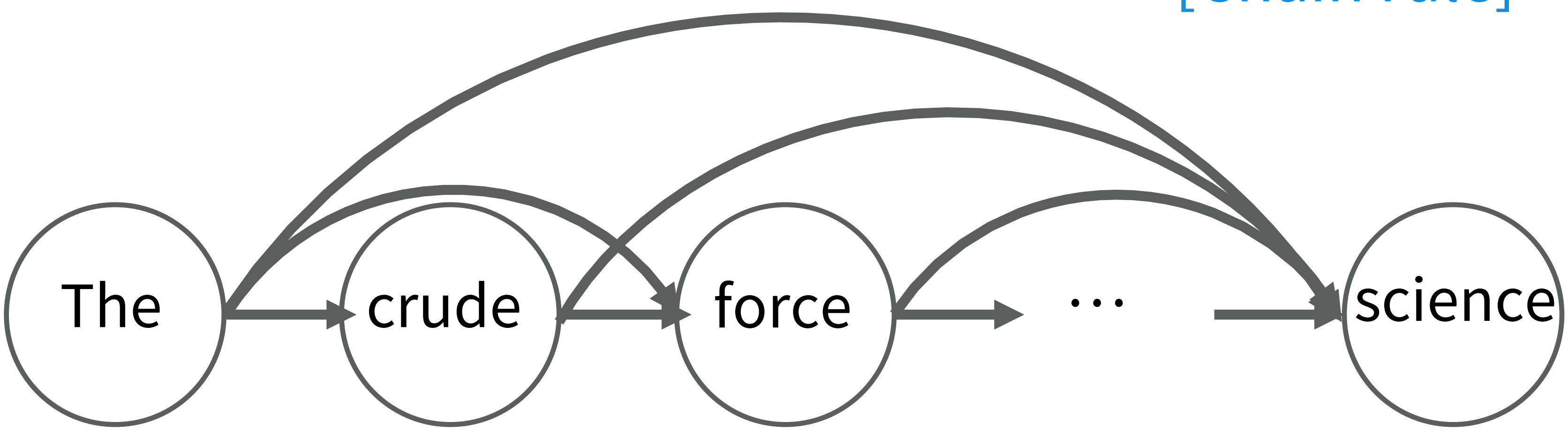
Language Modeling Task

- Language Modeling Task
  - Given a sequence of words (sentences),
  - Obtain P_{LM}: language model
  - Notation: x_t: word in a sentence position t
- language model: probability distribution over sequences of words.
  - $P_{LM}(x_t) = P(x_t | x_{t-1})$ (1-gram language model)
  - $P_{LM}(x_t) = P(x_t | x_{t-1}, x_{t-2})$ (2-gram language model)
  - $P_{LM}(x_t) = P(x_t | x_{t-1}, x_{t-2}, x_{t-3})$ (3-gram language model)
- Used in machine translation, speech recognition, part-of-speech tagging, information retrieval, ...

Language Models: Sentence probabilities

$$p(x_1, x_2, \ldots, x_T) = \prod_{t=1}^{T} p(x_t | x_1, \ldots, x_{t-1})$$

[Chain rule]



There are way too many histories once you're into a sentence a few words! Exponentially many.

Traditional Fix: Markov Assumption

An n^{th} order Markov assumption assumes each word depends only on a short linear history

$$p(x_1, x_2, \ldots, x_T) = \prod_{t=1}^{T} p(x_t | x_1, \ldots, x_{t-1})$$

$$\approx \prod_{t=1}^{T} p(x_t | x_{t-n}, \ldots, x_{t-1})$$

Problems of Traditional Markov Model Assumptions (1): Sparsity

Issue: *Very small window gives bad prediction; statistics for even a modest window are sparse*

Example:

$P(w_0 | w_{-3}, w_{-2}, w_{-1})$ $|V| = 100{,}000 \rightarrow 10^{15}$ contexts

Most have not been seen

The traditional answer is to use various backoff and smoothing techniques, but no good solution

Problems of Traditional Markov Model Assumptions (2): Context

Issue: *Dependency beyond the window is ignored*

Example:

*the same **stump** which had impaled the car of many a guest in the past thirty years and which he refused to have **removed***

Outline

- Review
- Language Modeling Task
- Feedforward Language Model
- Recurrent Neural Network Language Model
- LSTM / GRU
- Vanishing / Exploding Gradient

Neural Language Models

The neural approach [Bengio, Ducharme, Vincent & Jauvin JMLR 2003] represents words as dense distributed vectors so there can be sharing of statistical weight between similar words

Doing just this solves the sparseness problem of conventional n-gram models

Neural (Probabilistic) Language Model

[Bengio, Ducharme, Vincent & Jauvin JMLR 2003]

w_{-3}

registration

w_{-2}

is

w_{-1}

now

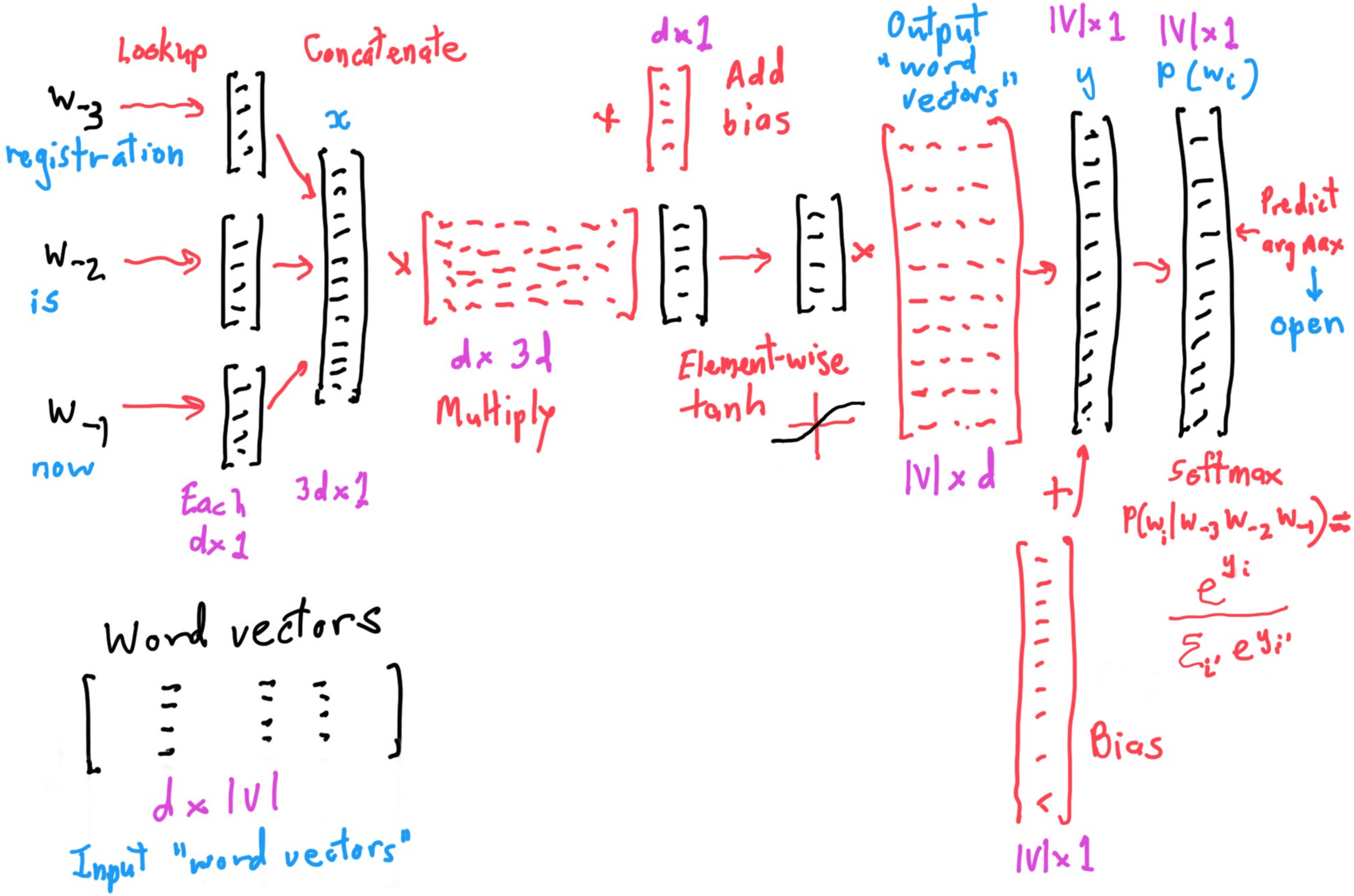
Predict

argmax

open

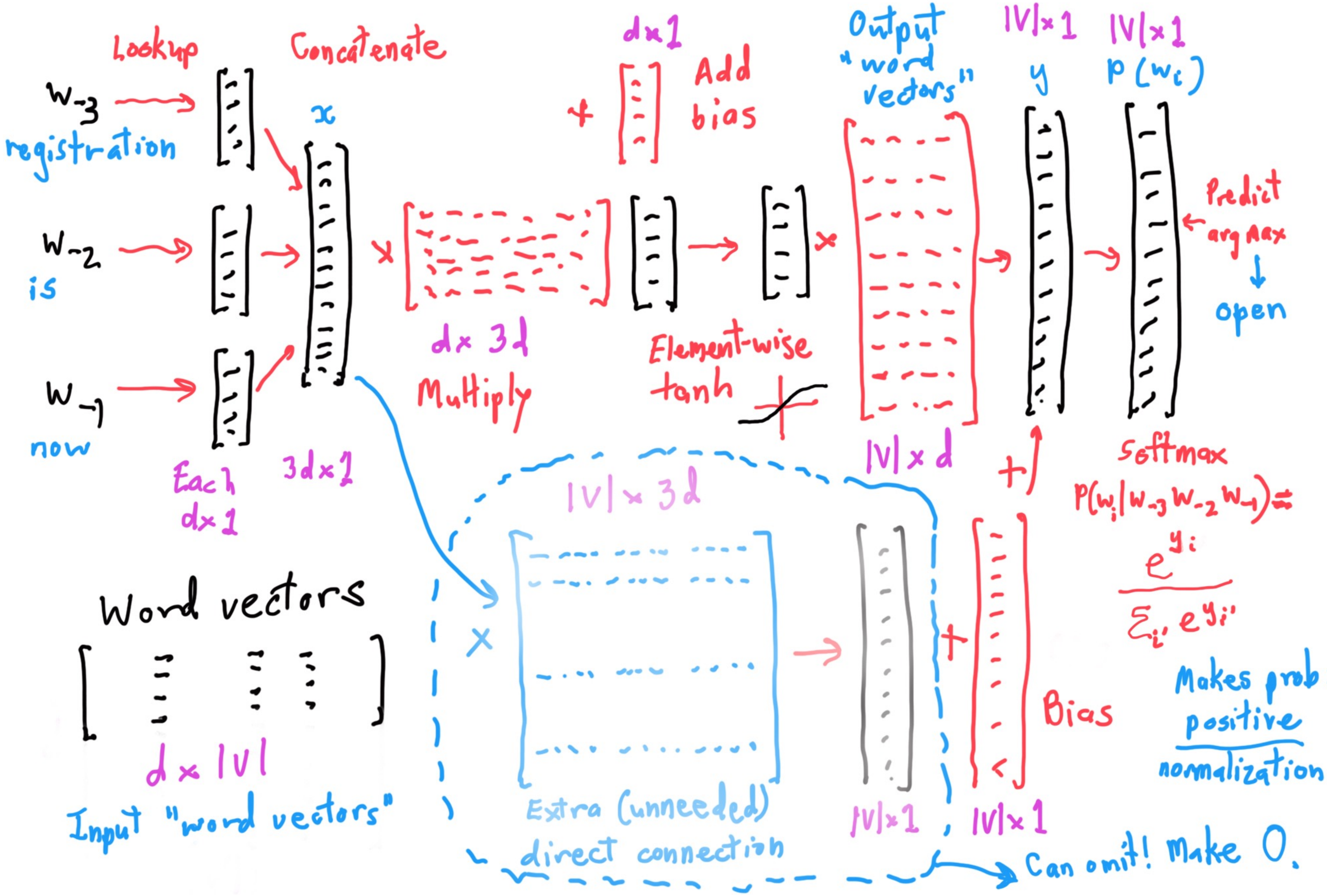
Neural (Probabilistic) Language Model

[Bengio, Ducharme, Vincent & Jauvin JMLR 2003]



Neural (Probabilistic) Language Model

[Bengio, Ducharme, Vincent & Jauvin JMLR 2003]



Outline

- Review
- Language Modeling Task
- Feedforward Language Model
- Recurrent Neural Network Language Model
- LSTM / GRU
- Vanishing / Exploding Gradient

Language Modeling Task

- Language Modeling Task
  - Given a sequence of words (sentences),
  - Obtain P_{LM}: language model
  - Notation: x_t: word in a sentence position t
- language model: probability distribution over sequences of words.
  - $P_{LM}(x_t) = P(x_t|x_{t-1})$ (1-gram language model)
  - $P_{LM}(x_t) = P(x_t|x_{t-1}, x_{t-2})$ (2-gram language model)
  - $P_{LM}(x_t) = P(x_t|x_{t-1}, x_{t-2}, x_{t-3})$ (3-gram language model)
- Used in machine translation, speech recognition, part-of-speech tagging, information retrieval, ...

Problems of Traditional Markov Model Assumptions (2): Context

Issue: *Dependency beyond the window is ignored*

Example:

*the same **stump** which had impaled the car of many a guest in the past thirty years and which he refused to have **removed***

A Non-Markovian Language Model

Can we directly model the true conditional probability?

$$p(x_1, x_2, \ldots, x_T) = \prod_{t=1}^{T} p(x_t | x_1, \ldots, x_{t-1})$$

Can we build a neural language model for this?

1. Feature extraction: $h_t = f(x_1, x_2, \ldots, x_t)$
2. Prediction: $p(x_{t+1} | x_1, \ldots, x_{t-1}) = g(h_t)$

How can f take a variable-length input?

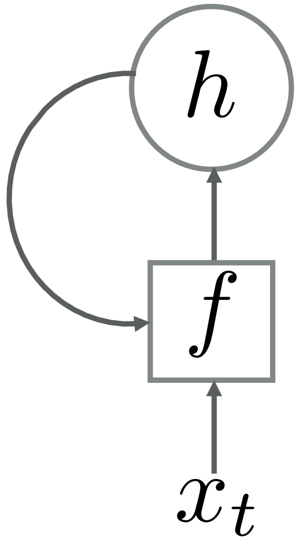
A Non-Markovian Language Model

Can we directly model the true conditional probability?

$$p(x_1, x_2, \ldots, x_T) = \prod_{t=1}^{T} p(x_t | x_1, \ldots, x_{t-1})$$

Recursive construction of f

1. Initialization $h_0 = 0$
2. Recursion $h_t = f(x_t, h_{t-1})$



We call h_t a hidden state or memory

h_t summarizes the history $(x_1, \ldots, x_t)$

A Non-Markovian Language Model

Example: $p(\text{the}, \text{cat}, \text{is}, \text{eating})$

(1) Initialization: $h_0 = 0$

(2) Recursion with Prediction:

$$h_1 = f(h_0, \langle \text{bos} \rangle) \rightarrow p(\text{the}) = g(h_1)$$

$$h_2 = f(h_1, \text{cat}) \rightarrow p(\text{cat}|\text{the}) = g(h_2)$$

$$h_3 = f(h_2, \text{is}) \rightarrow p(\text{is}|\text{the}, \text{cat}) = g(h_3)$$

$$h_4 = f(h_3, \text{eating}) \rightarrow p(\text{eating}|\text{the}, \text{cat}, \text{is}) = g(h_4)$$

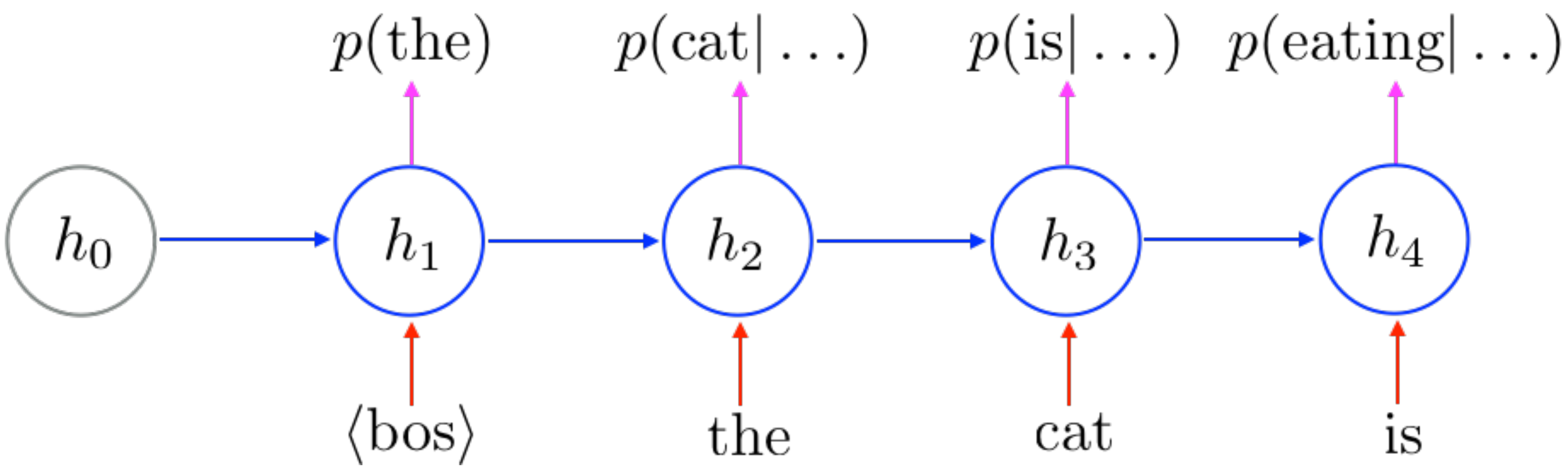
(3) Combination:

$$p(\text{the}, \text{cat}, \text{is}, \text{eating}) = g(h_1)g(h_2)g(h_3)g(h_4)$$

Read, Update and Predict

A Recurrent Neural Network Language Model solves the second problem!

Example: $p(\text{the}, \text{cat}, \text{is}, \text{eating})$



Read, *Update* and *Predict*

Building a Recurrent Language Model

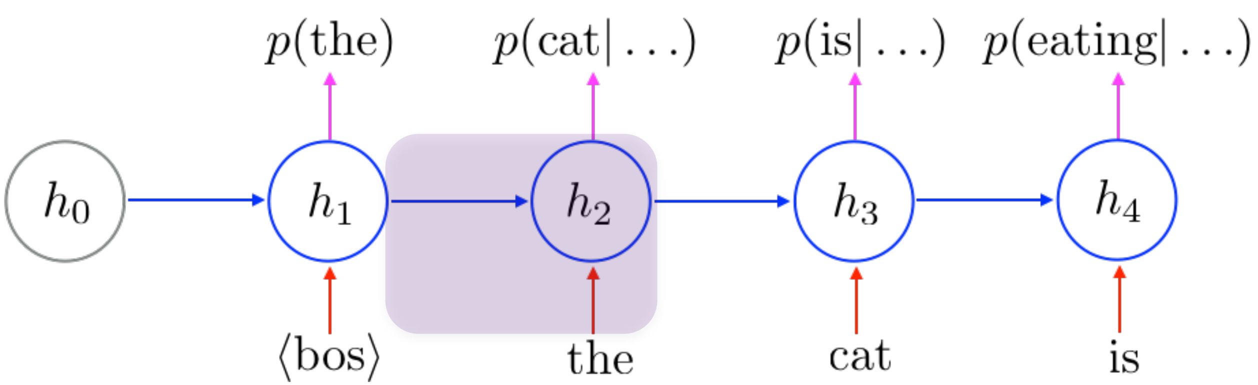
Transition Function $h_t = f(h_{t-1}, x_t)$

Inputs

i. Current word $x_t \in \{1, 2, \ldots, |V|\}$
ii. Previous state $h_{t-1} \in \mathbb{R}^d$

Parameters

i. Input weight matrix $W \in \mathbb{R}^{|V| \times d}$
ii. Transition weight matrix $U \in \mathbb{R}^{d \times d}$
iii. Bias vector $b \in \mathbb{R}^d$



Building a Recurrent Language Model

Transition Function $h_t = f(h_{t-1}, x_t)$

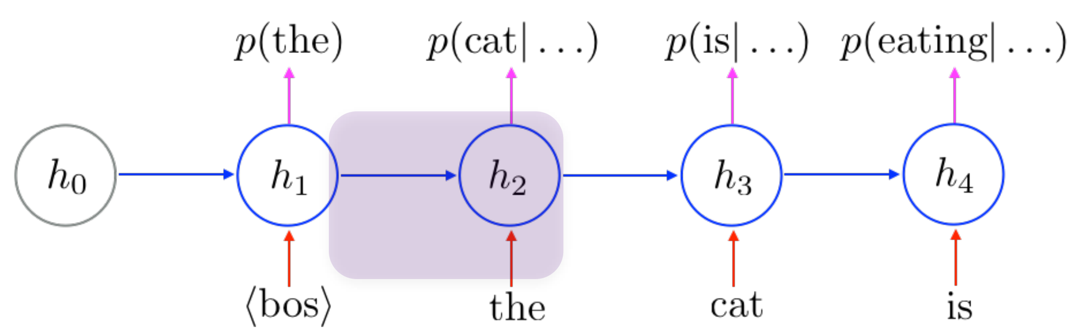
Naïve Transition Function

$$f(h_{t-1}, x_t) = \tanh(W\,[x_t] + U h_{t-1} + b)$$

Element-wise nonlinear transformation

Trainable word vector

Linear transformation of previous state



Building a Recurrent Language Model

Prediction Function $p(x_{t+1} = w | x_{\leq t}) = g_w(h_t)$

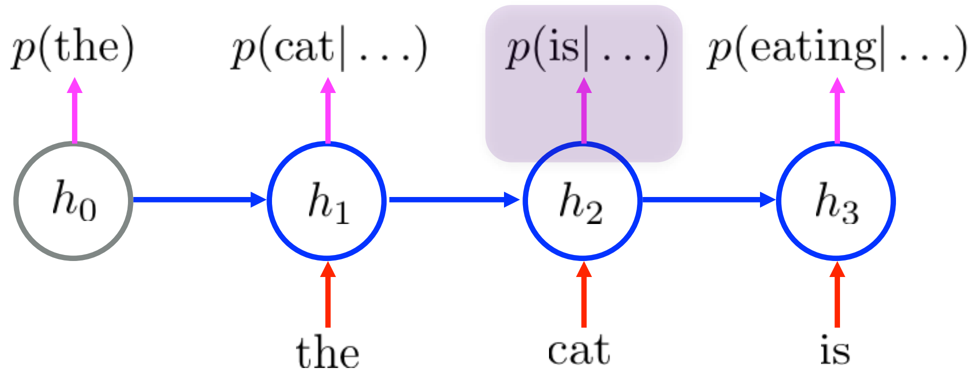
Inputs

i. Current state $h_t \in \mathbb{R}^d$

Parameters

i. Softmax matrix $R \in \mathbb{R}^{|V| \times d}$

ii. Bias vector $c \in \mathbb{R}^{|V|}$



Building a Recurrent Language Model

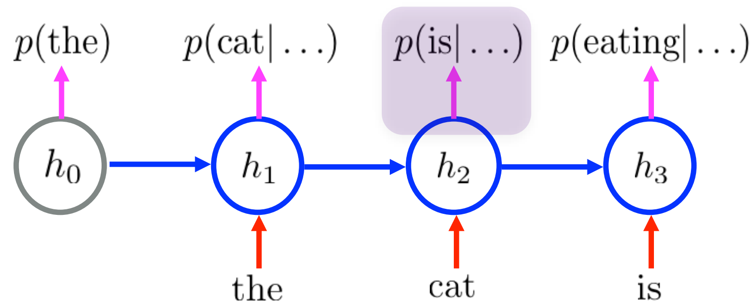
Prediction Function $p(x_{t+1} = w | x_{\leq t}) = g_w(h_t)$

$$p(x_{t+1} = w | x_{\leq t}) = g_w(h_t) = \frac{\exp(R[w]^\top h_t + c_w)}{\sum_{i=1}^{|V|} \exp(R[i]^\top h_t + c_i)}$$

Compatibility between trainable word vector and hidden state

Normalize

Exponentiate



Training a recurrent language model

Having determined the model form, we:

1. Initialize all parameters of the models, including the word representations with small random numbers
2. Define a loss function: how badly we predict actual next words [log loss or cross-entropy loss]
3. Repeatedly attempt to predict each next word
4. Backpropagate our loss to update **all** parameters
5. Just doing this learns good word representations and good prediction functions – it's almost magic

Outline

- Review
- Language Modeling Task
- Feedforward Language Model
- Recurrent Neural Network Language Model
- LSTM / GRU
- Vanishing / Exploding Gradient

Training a Recurrent Language Model

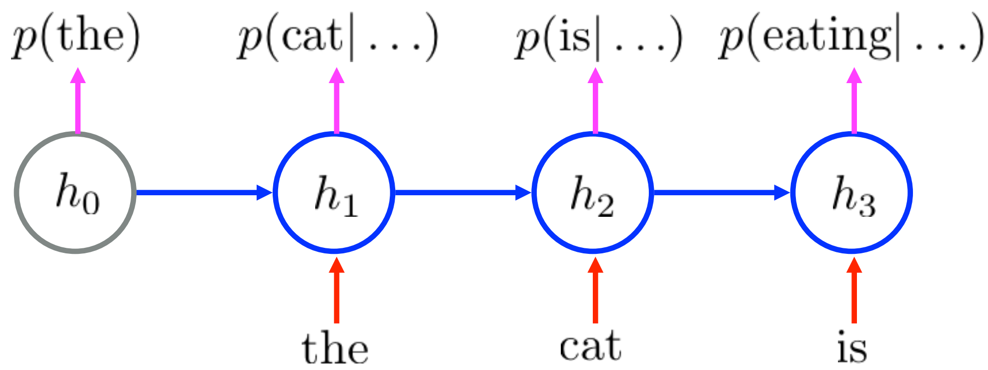
- Log-probability of one training sentence

$$\log p(x_1^n, x_2^n, \ldots, x_{T^n}^n) = \sum_{t=1}^{T^n} \log p(x_t^n | x_1^n, \ldots, x_{t-1}^n)$$

- Training set $D = \{X^1, X^2, \ldots, X^N\}$
- Log-likelihood Functional

$$\mathcal{L}(\theta, D) = \frac{1}{N} \sum_{n=1}^{N} \sum_{t=1}^{T^n} \log p(x_t^n | x_1^n, \ldots, x_{t-1}^n)$$

Minimize $-\mathcal{L}(\theta, D)$ *!!*

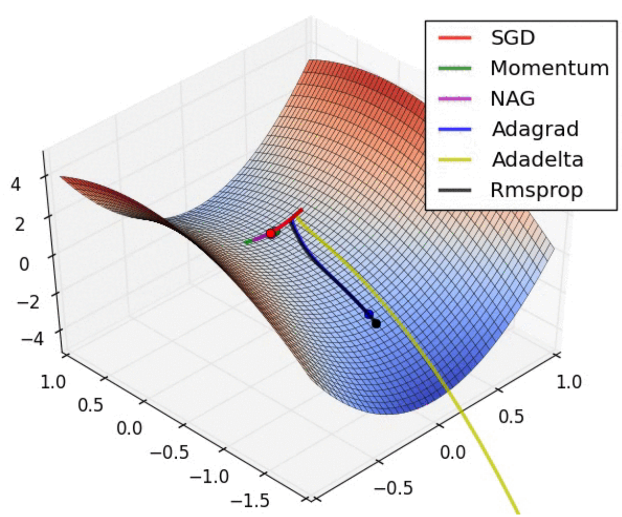


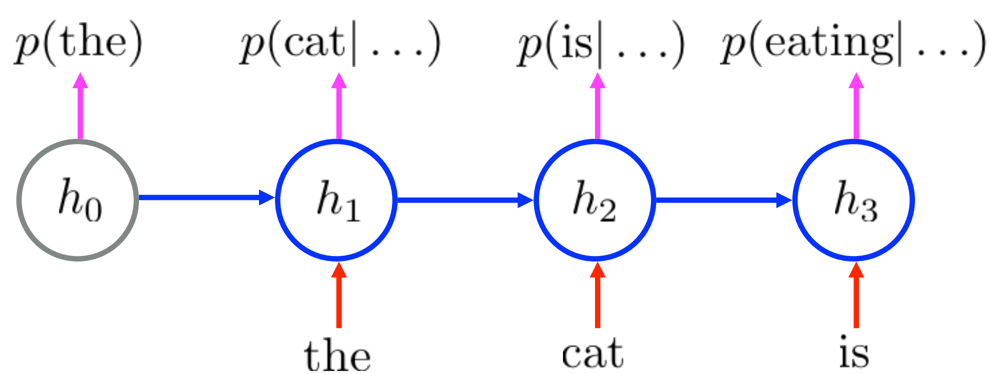
Gradient Descent

- Move slowly in the steepest descent direction

$$\theta \leftarrow \theta - \eta \nabla \mathcal{L}(\theta, D)$$

- Computational cost of a single update: $O(N)$
- Not suitable for a large corpus





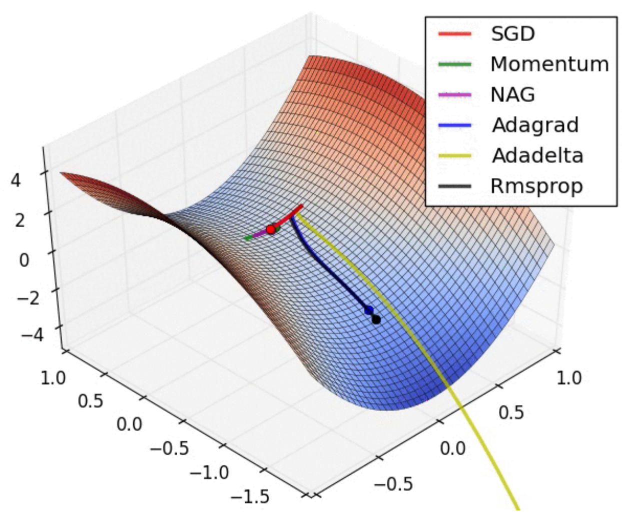
Stochastic Gradient Descent

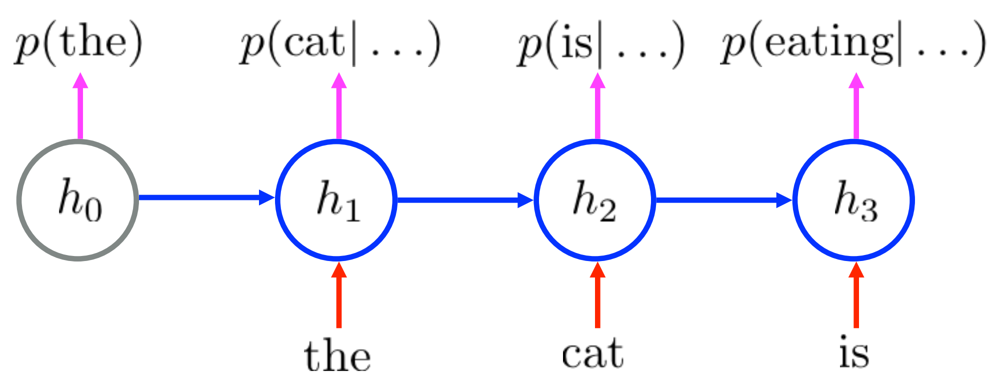
- Estimate the steepest direction with a minibatch

$$\nabla \mathcal{L}(\theta, D) \approx \nabla \mathcal{L}(\theta, \{X^1, \ldots, X^n\})$$

- Until the convergence (w.r.t. a validation set)

$$|\mathcal{L}(\theta, D_{\text{val}}) - \mathcal{L}(\theta - \eta \mathcal{L}(\theta, D), D_{\text{val}})| \leq \epsilon$$



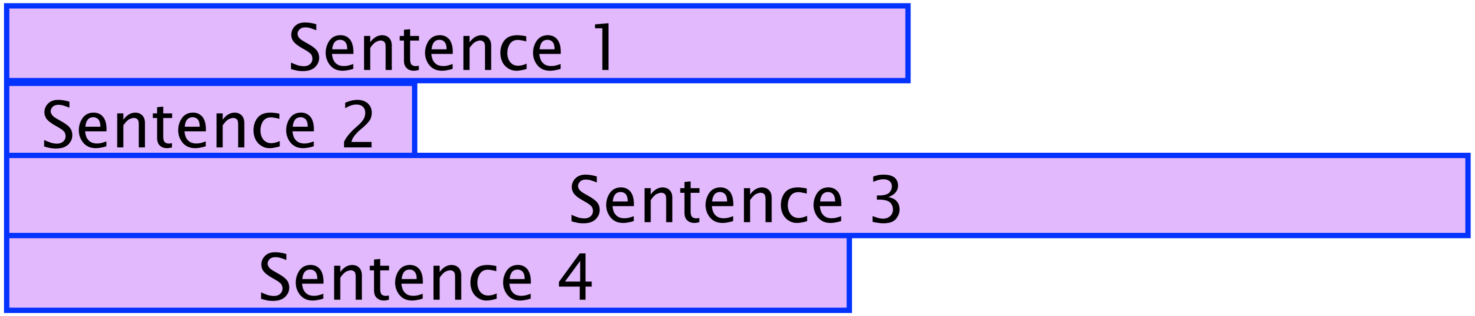


Outline

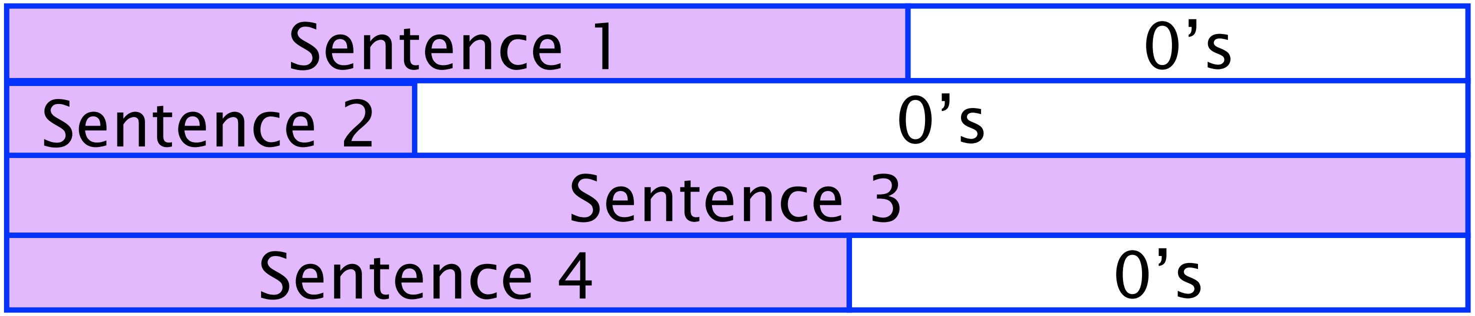
- Review
- Language Modeling Task
- Feedforward Language Model
- Recurrent Neural Network Language Model
- LSTM / GRU
- Vanishing / Exploding Gradient

Stochastic Gradient Descent

- *Not* trivial to build a minibatch

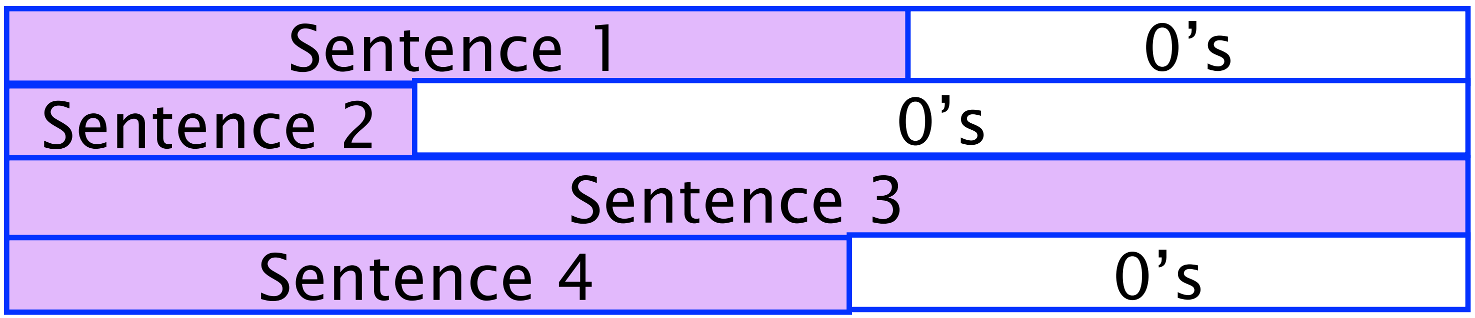


1. Padding and Masking: *suitable for GPU's, but wasteful*
    - *Wasted computation*

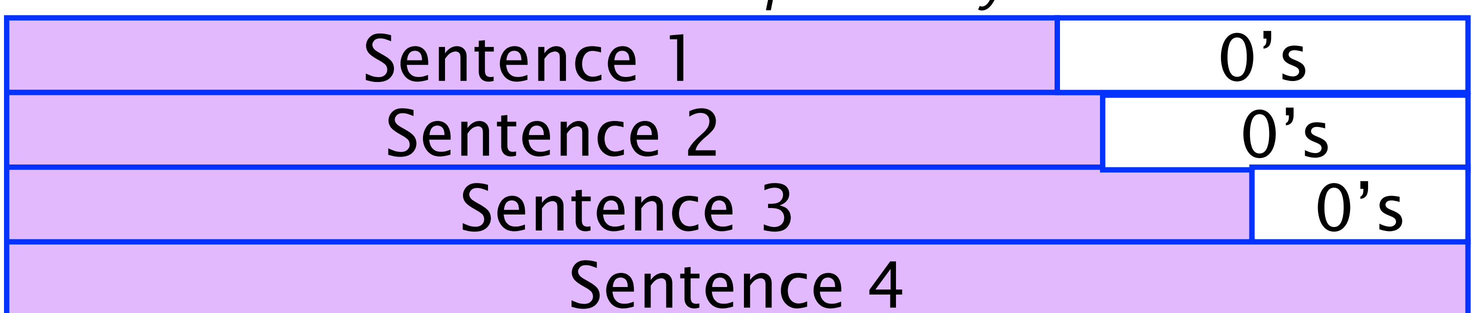


Stochastic Gradient Descent

1. Padding and Masking: *suitable for GPU's, but wasteful*
   - *Wasted computation*



2. Smarter Padding and Masking: *minimize the waste*
   - *Ensure that the length differences are minimal.*
   - *Sort the sentences and sequentially build a minibatch*



Backpropagation through Time

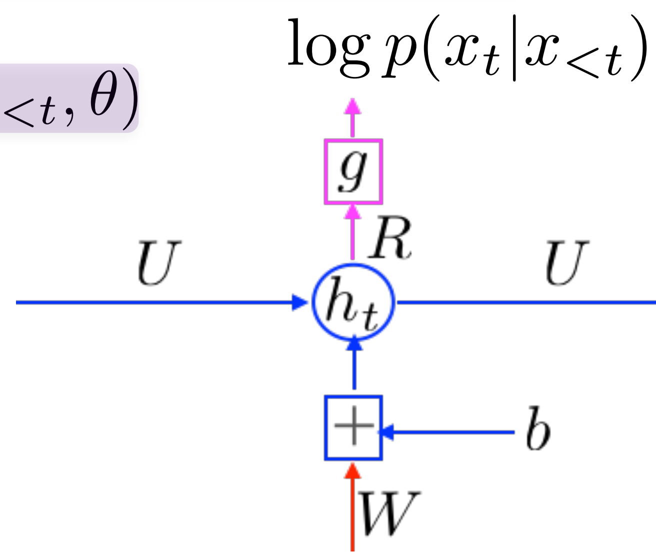
How do we compute $\nabla \mathcal{L}(\theta, D)$ *?*

- Cost as a sum of per-sample cost function

$$\nabla \mathcal{L}(\theta, D) = \sum_{X \in D} \nabla \mathcal{L}(\theta, X)$$

- Per-sample cost as a sum of per-step cost functions

$$\nabla \mathcal{L}(\theta, X) = \sum_{t=1}^{T} \nabla \log p(x_t | x_{<t}, \theta)$$

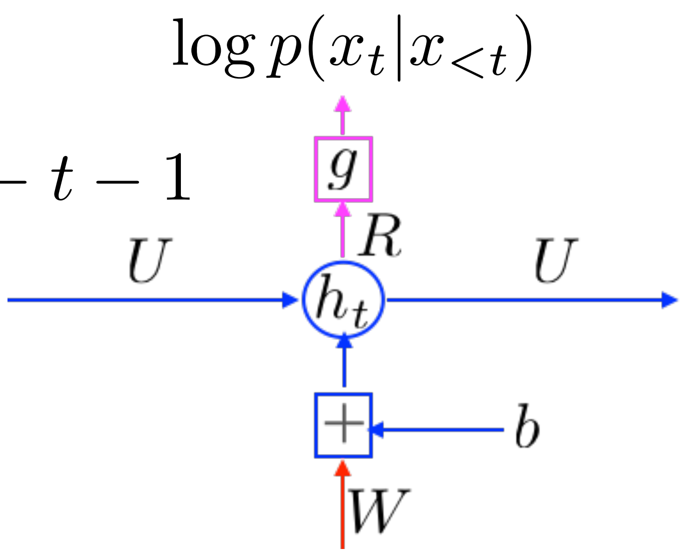


Backpropagation through Time

How do we compute $\nabla \log p(x_t|x_{<t},\theta)$ *?*

- Compute per-step cost function from time $t = T$

1. Cost derivative $\partial \log p(x_t|x_{<t})/\partial g$
2. Gradient w.r.t. R : $\times \partial g/\partial R$
3. Gradient w.r.t. h_t : $\times \partial g/\partial h_t + \partial h_{t+1}/\partial h_t$
4. Gradient w.r.t. U : $\times \partial h_t/\partial U$
5. Gradient w.r.t. b and W :
   $\times \partial h_t/\partial b$ and $\times \partial h_t/\partial W$
6. Accumulate the gradient and $t \leftarrow t-1$



Note: I'm abusing math a lot here!!

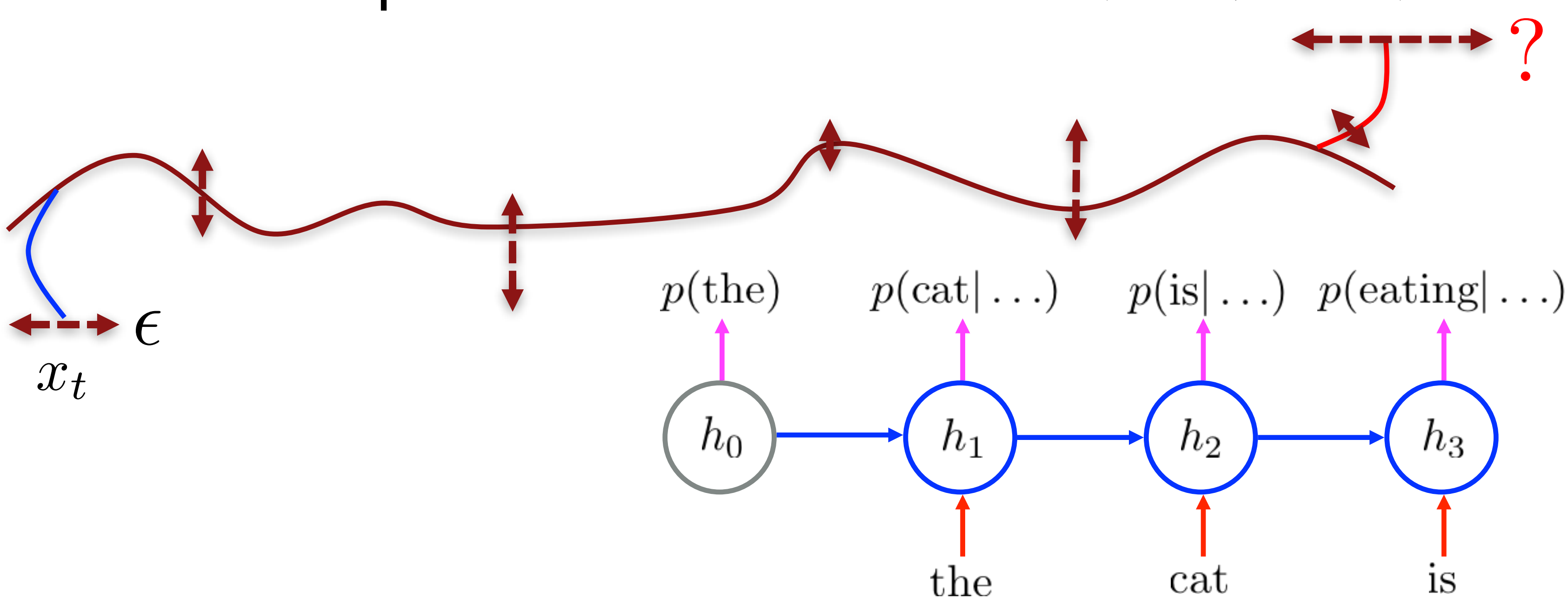
Backpropagation through Time

Intuitively, what's happening here?

1. Measure the influence of the past on the future

$$\frac{\partial \log p(x_{t+n}|x_{<t+n})}{\partial h_t} = \frac{\partial \log p(x_{t+n}|x_{<t+n})}{\partial g} \frac{\partial g}{\partial h_{t+n}} \frac{\partial h_{t+n}}{\partial h_{t+n-1}} \cdots \frac{\partial h_{t+1}}{\partial h_t}$$

2. How does the perturbation at t affect $p(x_{t+n}|x_{<t+n})$?



Backpropagation through Time

Intuitively, what's happening here?

1. Measure the influence of the past on the future

$$\frac{\partial \log p(x_{t+n}|x_{<t+n})}{\partial h_t} = \frac{\partial \log p(x_{t+n}|x_{<t+n})}{\partial g} \frac{\partial g}{\partial h_{t+n}} \frac{\partial h_{t+n}}{\partial h_{t+n-1}} \cdots \frac{\partial h_{t+1}}{\partial h_t}$$

2. How does the perturbation at t affect $p(x_{t+n}|x_{<t+n})$?

3. Change the parameters to maximize $p(x_{t+n}|x_{<t+n})$

Backpropagation through Time

Intuitively, what's happening here?

1. Measure the influence of the past on the future

$$\frac{\partial \log p(x_{t+n}|x_{<t+n})}{\partial h_t} = \frac{\partial \log p(x_{t+n}|x_{<t+n})}{\partial g} \frac{\partial g}{\partial h_{t+n}} \frac{\partial h_{t+n}}{\partial h_{t+n-1}} \cdots \frac{\partial h_{t+1}}{\partial h_t}$$

2. With a naïve transition function

$$f(h_{t-1}, x_{t-1}) = \tanh(W\left[x_{t-1}\right] + U h_{t-1} + b)$$

We get $\frac{\partial J_{t+n}}{\partial h_t} = \frac{\partial J_{t+n}}{\partial g} \frac{\partial g}{\partial h_{t+N}} \underbrace{\prod_{n=1}^{N} U^\top \mathrm{diag}\left(\frac{\partial \tanh(a_{t+n})}{\partial a_{t+n}}\right)}_{}$

Problematic!

[Bengio, IEEE 1994]

Backpropagation through Time

Gradient either ***vanishes*** *or* ***explodes***

- What happens?

$$\frac{\partial J_{t+n}}{\partial h_t} = \frac{\partial J_{t+n}}{\partial g} \frac{\partial g}{\partial h_{t+N}} \underbrace{\prod_{n=1}^{N} U^\top \mathrm{diag}\left(\frac{\partial \tanh(a_{t+n})}{\partial a_{t+n}}\right)}$$

1. The gradient *likely* explodes if

$$e_{\max} \geq \frac{1}{\max \tanh'(x)} = 1$$

2. The gradient *likely* vanishes if

$$e_{\max} < \frac{1}{\max \tanh'(x)} = 1,$$ where $e_{\max}$: largest eigenvalue of U

[Bengio, Simard, Frasconi, TNN1994; Hochreiter, Bengio, Frasconi, Schmidhuber, 2001]

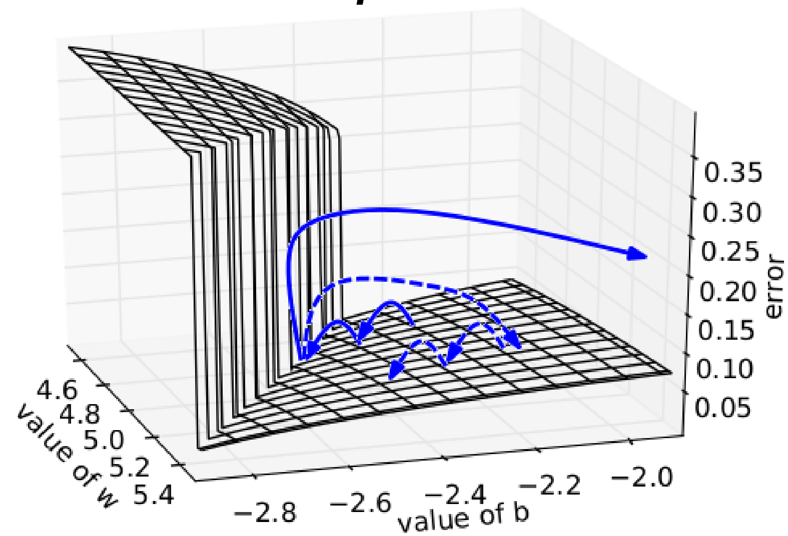
Vanishing/Exploding Gradient (Intuition Only)

- long-term dependencies
  - Suppose that the backpropagation involves repeated multiplication of matrix W.
  - After t steps, this becomes W^t.
  - Suppose W allows eigendecomposition, $W = V \text{diag}(\lambda) V^{-1}$.
  - Then $W^t = (V \text{diag}(\lambda) V^{-1})^t = V \text{diag}(\lambda)^t V^{-1}$.
- When eigenvalues which are greater than 1, this will explode.
- When eigenvalues which are less than 1, this will vanish.
  - Exploding gradients: this makes learning unstable.
  - Vanishing gradients: it is difficult to know which direction the parameters should move to improve the cost function

Backpropagation through Time

Addressing Exploding Gradient

- *"when gradients explode so does the curvature along v, leading to a wall in the error surface"*



- Gradient Clipping
  1. Norm clipping

$$\tilde{\nabla} \leftarrow \begin{cases} \frac{c}{\|\nabla\|}\nabla & \text{,if } \|\nabla\| \geq c \\ \nabla & \text{,otherwise} \end{cases}$$

  2. Element-wise clipping

$$\nabla_i \leftarrow \min(c, |\nabla_i|)\text{sgn}(\nabla_i), \text{ for all } i \in \{1, \ldots, \dim \nabla\}$$

[Pascanu, Mikolov, Bengio, ICML 2013]

Backpropagation through Time

Vanishing gradient is super-problematic

- When we only observe

$$\left\| \frac{\partial h_{t+N}}{\partial h_t} \right\| = \left\| \prod_{n=1}^{N} U^\top \mathrm{diag}\left(\frac{\partial \tanh(a_{t+n})}{\partial a_{t+n}} \right) \right\| \to 0 \ ,$$

- We cannot tell whether
  1. No dependency between *t* and *t*+*n* in data, or
  2. Wrong configuration of parameters:

$$e_{\max}(U) < \frac{1}{\max \tanh'(x)}$$

Outline

- Review
- Language Modeling Task
- Feedforward Language Model
- Recurrent Neural Network Language Model
- LSTM / GRU
- Vanishing / Exploding Gradient

GRU/LSTM Ideas

- Model that operates at multiple time scales
  - some parts of the model operate at fine-grained time scales and can handle small details.
  - other parts operate at coarse time scales and transfer information from the distant past to the present more efficiently.
- Strategies
  - addition of skip connections across time
  - "leaky units" which integrate signals with different time constraints
  - removal of some of the connections used to model fine-grained time sclae.

Gated Recurrent Unit

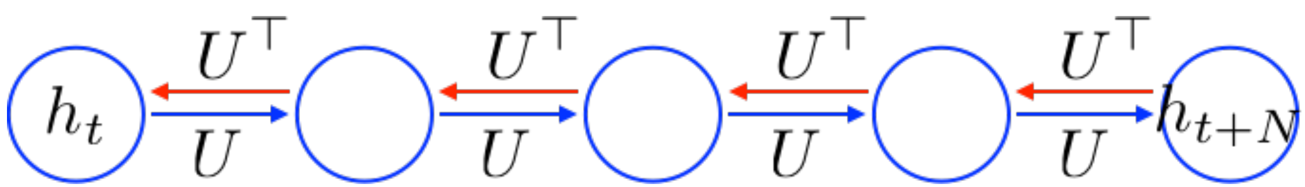
- Is the problem with the naïve transition function?

$$f(h_{t-1}, x_t) = \tanh(W\,[x_t] + U h_{t-1} + b)$$

- With it, the temporal derivative is

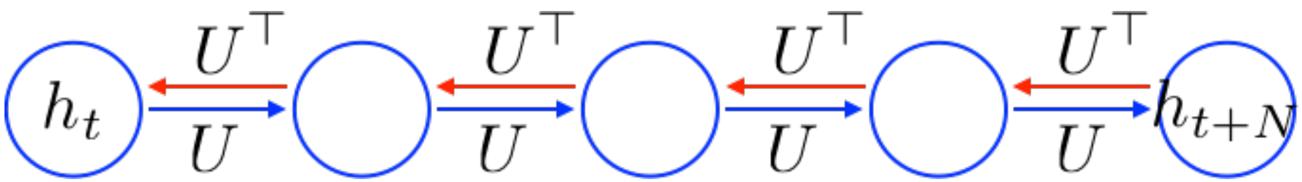
$$\frac{\partial h_{t+1}}{\partial h_t} = U^\top \frac{\partial \tanh(a)}{\partial a}$$

- It implies that the error must be backpropagated through all the intermediate nodes:

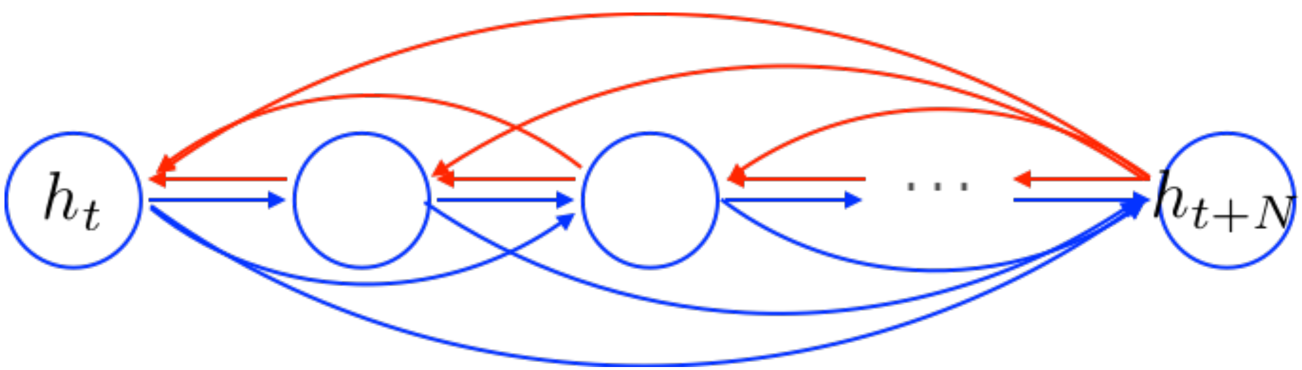


Gated Recurrent Unit

- It implies that the error must backpropagate through all the intermediate nodes:

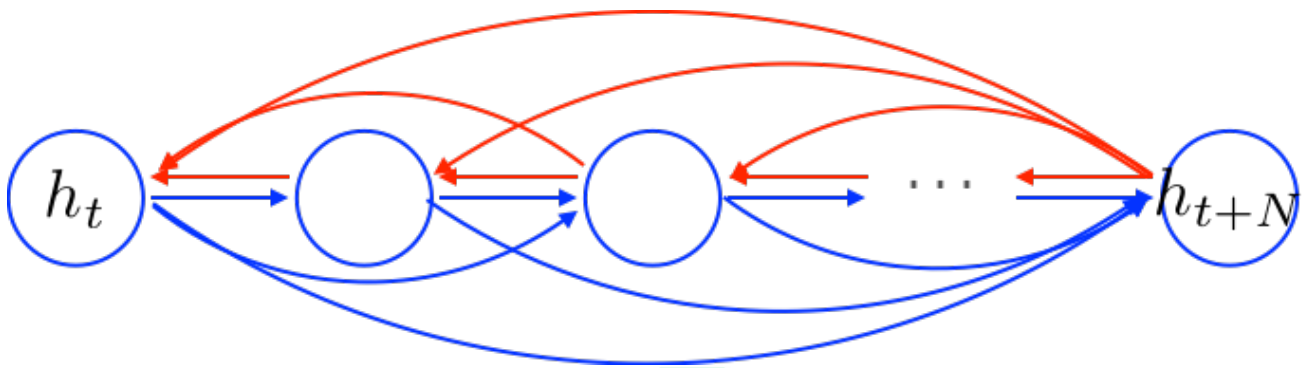


- Perhaps we can create shortcut connections.



Gated Recurrent Unit

- Perhaps we can create *adaptive* shortcut connections.



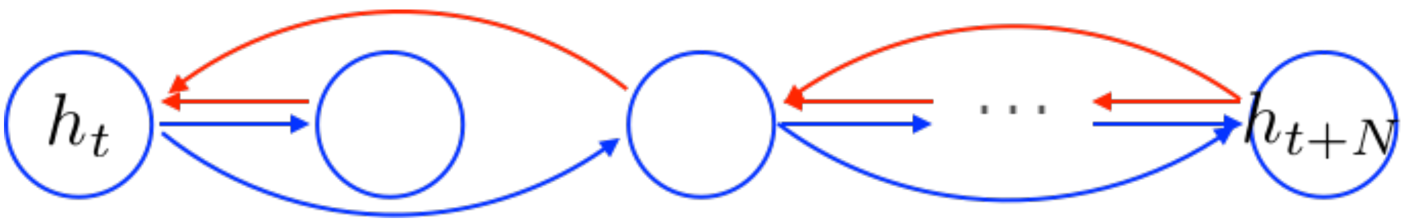
$$f(h_{t-1}, x_t) = u_t \odot \tilde{h}_t + (1 + u_t) \odot h_{t-1}$$

- Candidate Update $\tilde{h}_t = \tanh(W[x_t] + Uh_{t-1} + b)$
- Update gate $u_t = \sigma(W_u[x_t] + U_u h_{t-1} + b_u)$

$\odot$: element-wise multiplication

Gated Recurrent Unit

- Let the net prune unnecessary connections *adaptively*.

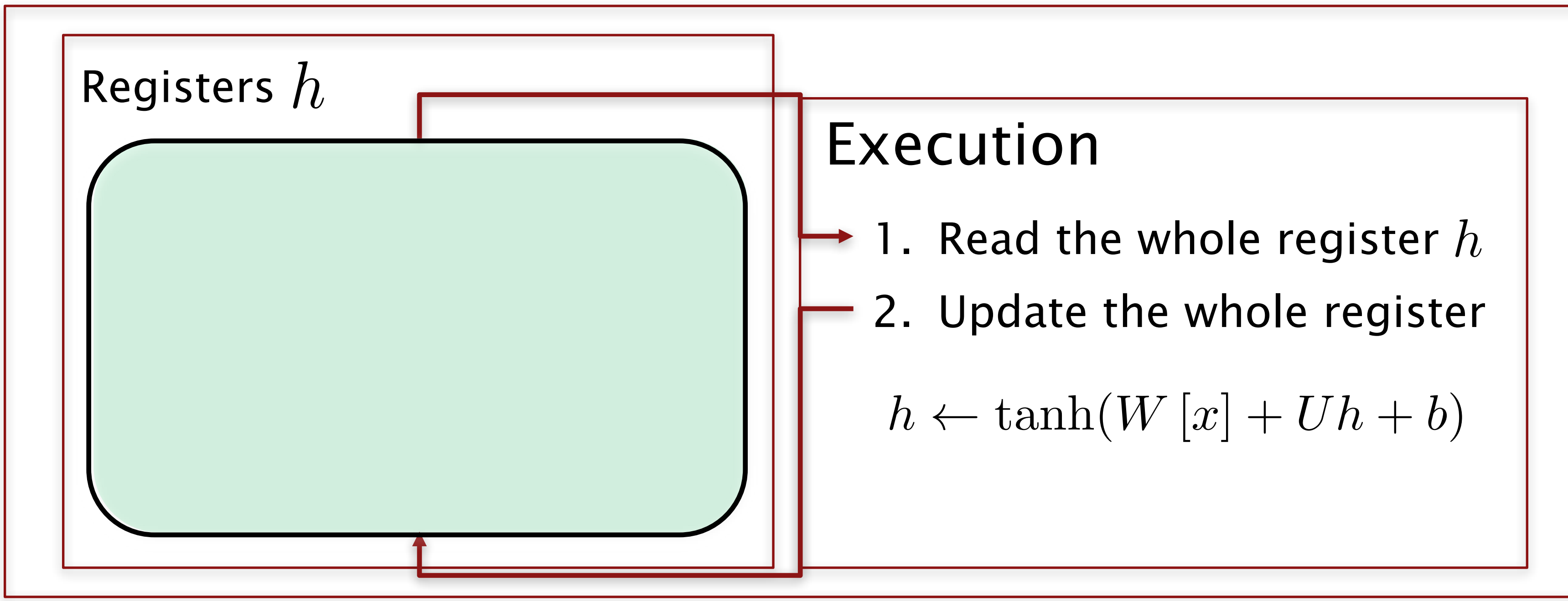


$$f(h_{t-1}, x_t) = u_t \odot \tilde{h}_t + (1 + u_t) \odot h_{t-1}$$

- Candidate Update $\tilde{h}_t = \tanh(W[x_t] + U(r_t \odot h_{t-1}) + b)$
- Reset gate $r_t = \sigma(W_r[x_t] + U_r h_{t-1} + b_r)$
- Update gate $u_t = \sigma(W_u[x_t] + U_u h_{t-1} + b_u)$

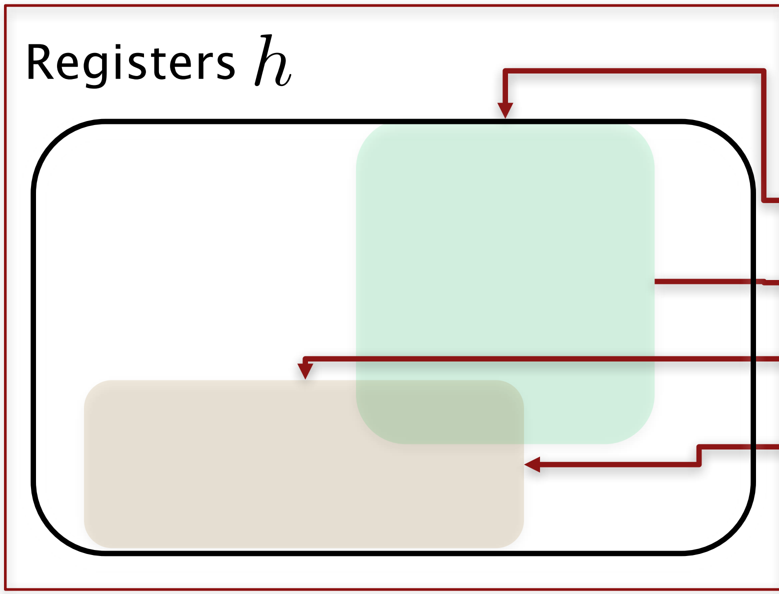
Gated Recurrent Unit

tanh-RNN ….



Gated Recurrent Unit

GRU …



Execution

1. Select a readable subset r
2. Read the subset $r \odot h$
3. Select a writable subset u
4. Update the subset

$$h \leftarrow u \odot \tilde{h} + (1 - u_t) \odot h$$

Clearly gated recurrent units are much more realistic.

Gated Recurrent Unit

Two most widely used gated recurrent units

Gated Recurrent Unit

[Cho et al., EMNLP2014; Chung, Gulcehre, Cho, Bengio, DLUFL2014]

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

$$\tilde{h} = \tanh(W[x_t] + U(r_t \odot h_{t-1}) + b)$$

$$u_t = \sigma(W_u[x_t] + U_u h_{t-1} + b_u)$$

$$r_t = \sigma(W_r[x_t] + U_r h_{t-1} + b_r)$$

Long Short-Term Memory

[Hochreiter&Schmidhuber, NC1999; Gers, Thesis2001]

$$h_t = o_t \odot \tanh(c_t)$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$

$$\tilde{c}_t = \tanh(W_c[x_t] + U_c h_{t-1} + b_c)$$

$$o_t = \sigma(W_o[x_t] + U_o h_{t-1} + b_o)$$

$$i_t = \sigma(W_i[x_t] + U_i h_{t-1} + b_i)$$

$$f_t = \sigma(W_f[x_t] + U_f h_{t-1} + b_f)$$

Training an RNN

A few well-established + my personal wisdoms

1. Use LSTM or GRU: *makes your life so much simpler*
2. Initialize recurrent matrices to be orthogonal
3. Initialize other matrices with a sensible scale
4. Use adaptive learning rate algorithms: *Adam, Adadelta, …*
5. Clip the norm of the gradient: “*1” seems to be a reasonable threshold when used together with adam or adadelta.*
6. *Be patient!*

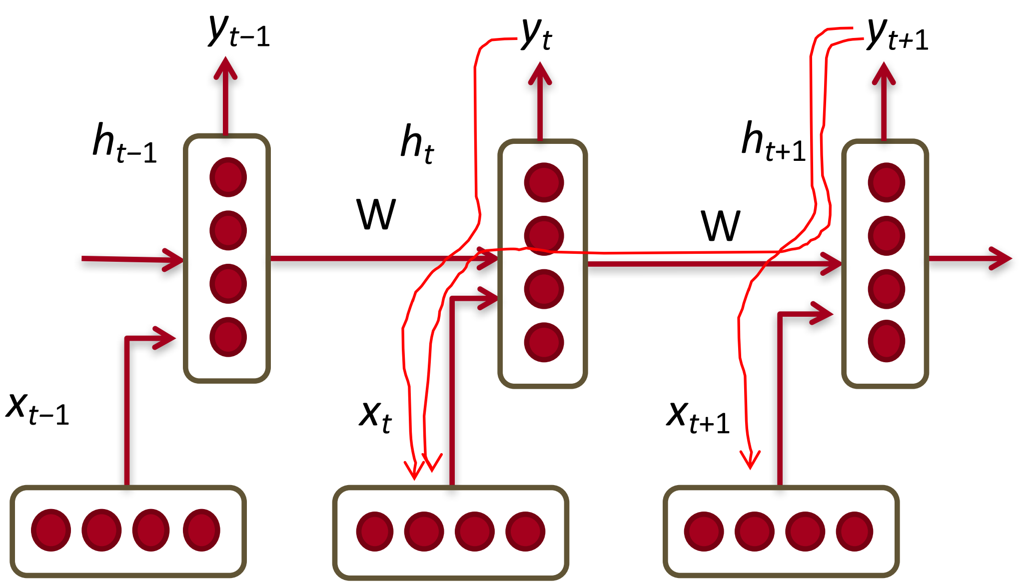
[Saxe et al., ICLR2014;
Ba, Kingma, ICLR2015;
Zeiler, arXiv2012;
Pascanu et al., ICML2013]

Outline

- Review
- Language Modeling Task
- Feedforward Language Model
- Recurrent Neural Network Language Model
- LSTM / GRU
- Vanishing / Exploding Gradient

The vanishing/exploding gradient problem

- Multiply the same matrix at each time step during backprop



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:

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

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

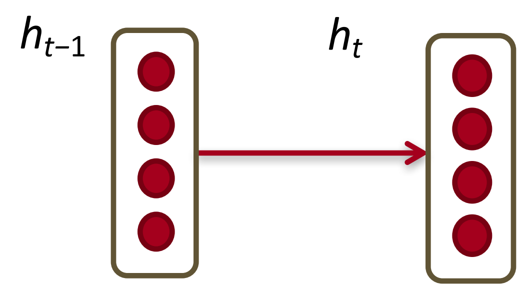
$$\frac{\partial h_t}{\partial h_k} = \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}}$$

- Each partial is a Jacobian:

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

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

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

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

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

Check at home that you understand the diag matrix formulation

The vanishing gradient problem - Details

- Analyzing the norms of the Jacobians, yields:

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

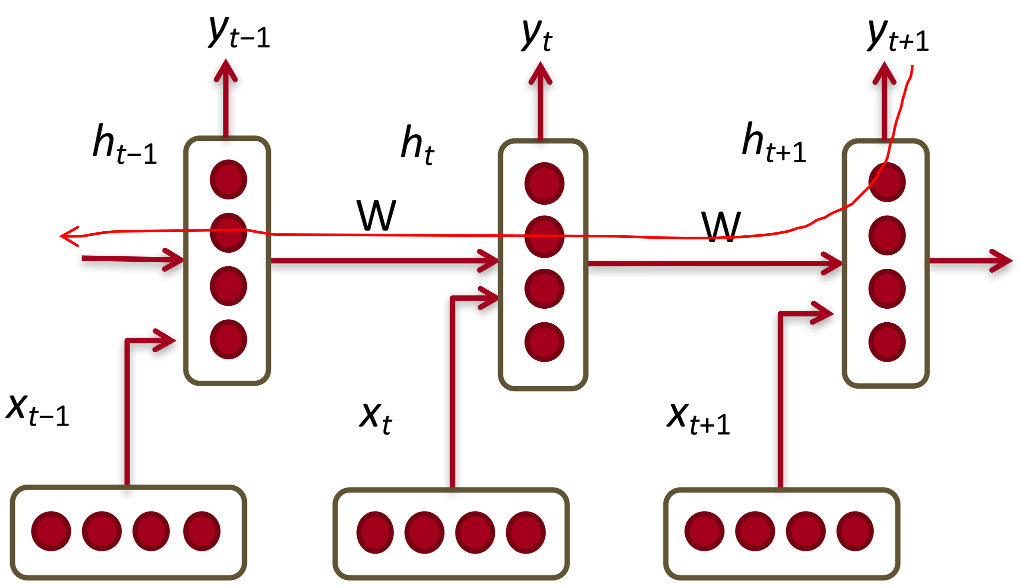
- Where we defined ¯'s as upper bounds of the norms
- The gradient is a product of Jacobian matrices, each associated with a step in the forward computation.

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

- 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:

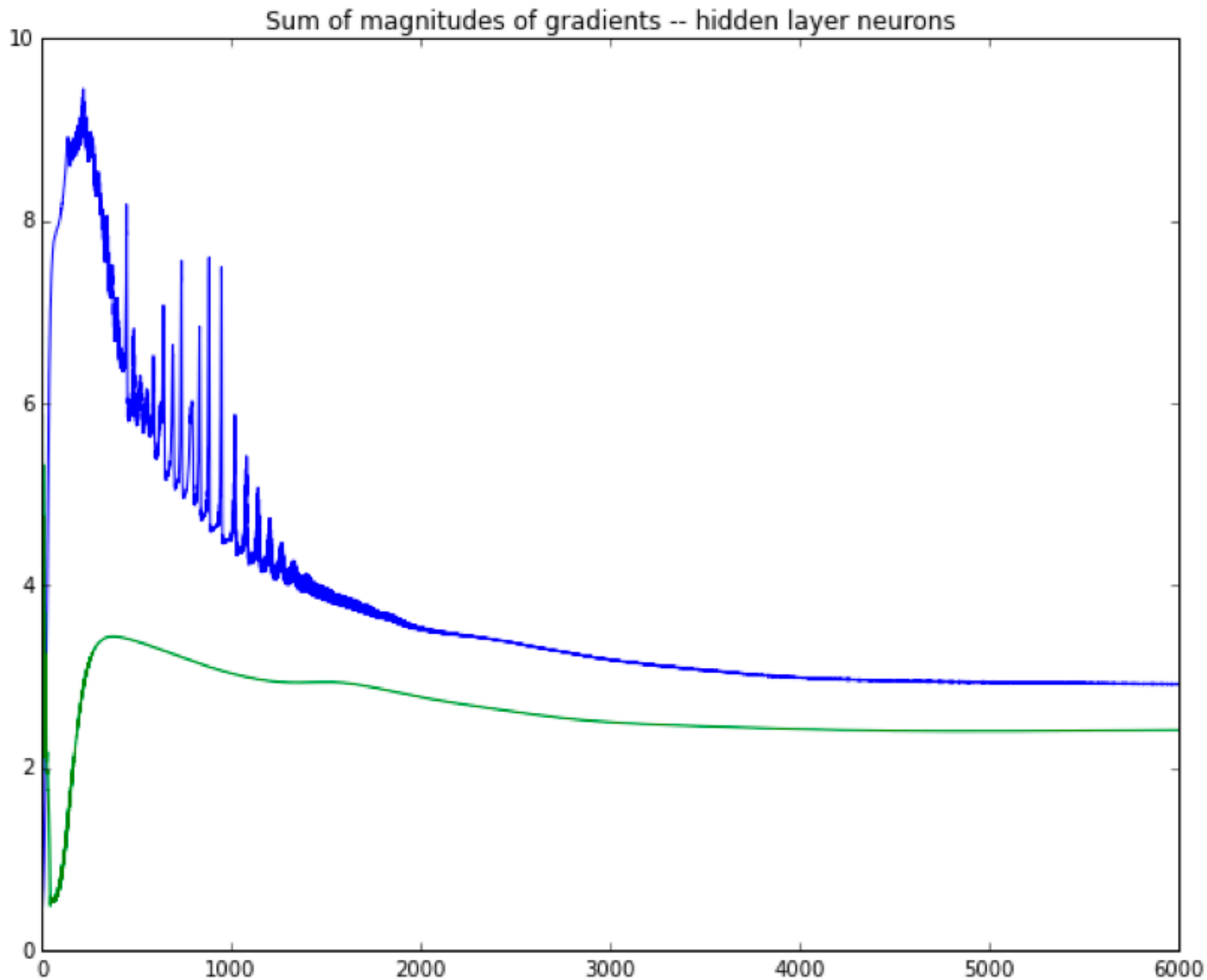
  Jane walked into the room. John walked in too. It was late in the day. Jane said hi to ____

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

In [21]:
```
plt.plot(np.array(relu_array[:6000]),color='blue')
plt.plot(np.array(sigm_array[:6000]),color='green')
plt.title('Sum of magnitudes of gradients -- hidden layer neurons')
```

Out[21]: <matplotlib.text.Text at 0x10a331310>



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

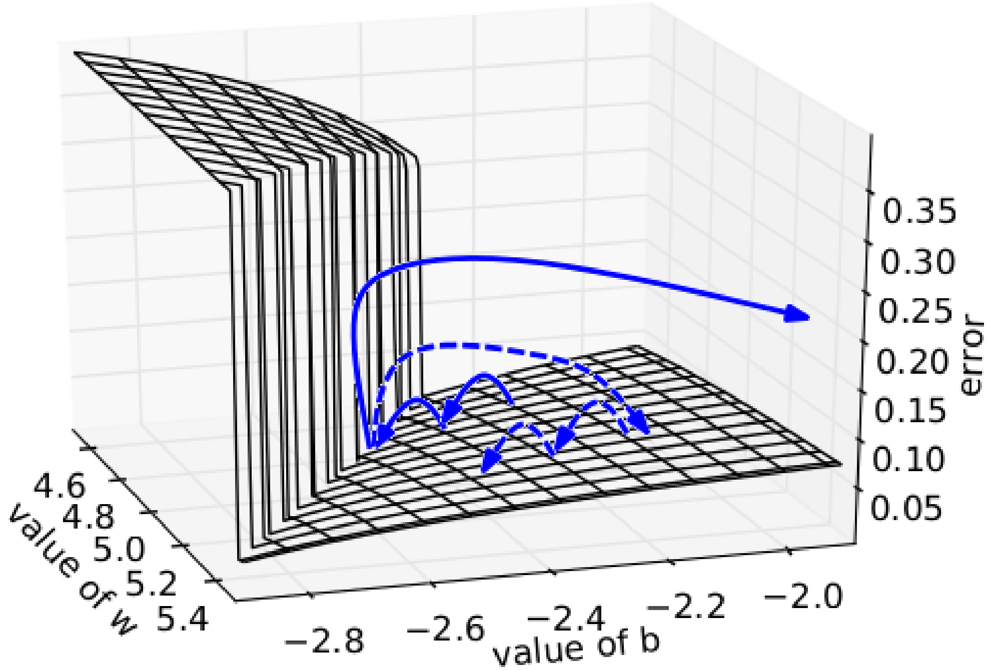


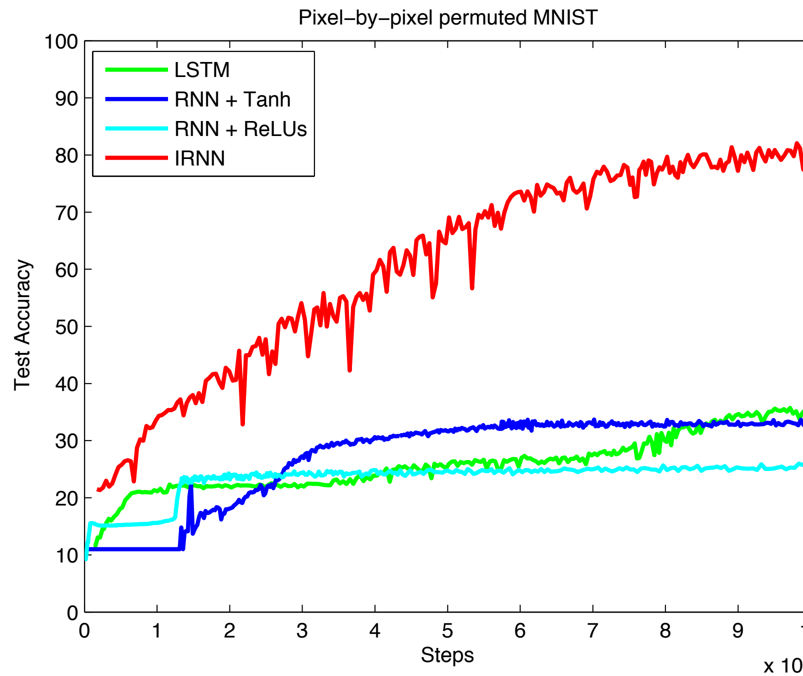
Figure from paper:
On the difficulty of training Recurrent Neural Networks, Pascanu et al. 2013

- 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 2 weeks ago (!) in *A Simple Way to Initialize Recurrent Networks of Rectified Linear* Units, Le et al. 2015

Conclusion

- Language Modeling Task
- Feedforward Language Model
- Recurrent Neural Network Language Model
- LSTM / GRU
- Vanishing / Exploding Gradient

- Thank you for your attention!!

References

- Richard Socher's ipython code for vanishing gradient problem: `http://cs224d.stanford.edu/notebooks/vanishing_grad_example`
- Various optionsation algorithms (Alec Radford): `http://www.denizyuret.com/2015/03/alec-radfords-animations-f`