CS224d
Deep NLP

Lecture 6:
Neural Tips and Tricks
+
Recurrent Neural Networks

Richard Socher

richard@metamind.io

Overview Today:

- Another explanation of backprop
  (from a tutorial Yoshua, Chris and I did)

- Practical tips and tricks:
  - Multi-task learning
  - Nonlinearities
  - Finite difference gradient check
  - Momentum, AdaGrad

- Language Models
- **First intro to Recurrent Neural Networks**

Backpropagation (Another explanation)

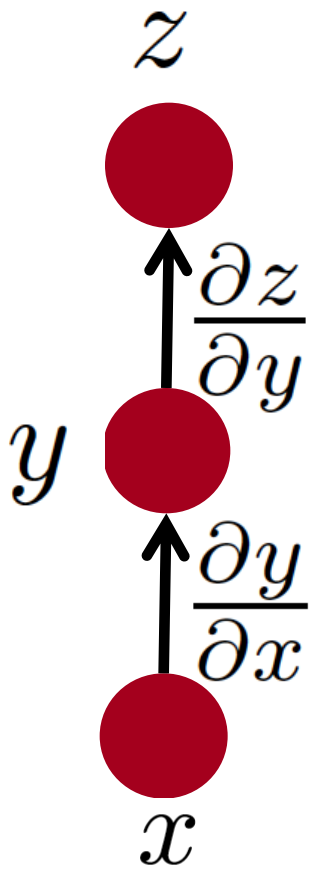
- Compute gradient of example-wise loss wrt parameters

- Simply applying the derivative chain rule wisely

$$z = f(y) \quad y = g(x) \quad \frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$$

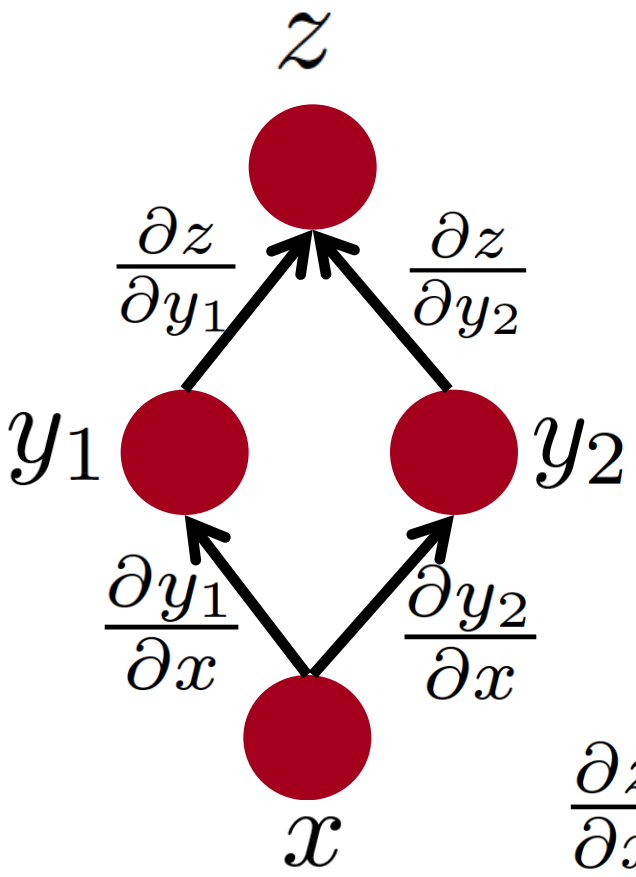
- If computing the loss(example, parameters) is $O(n)$ computation, then so is computing the gradient

Simple Chain Rule



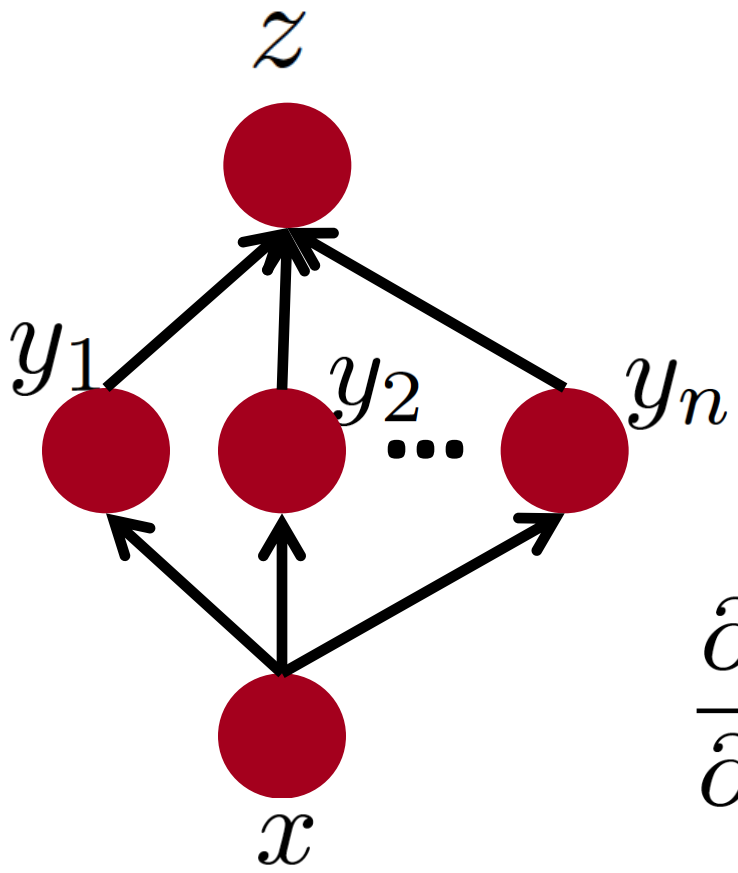
$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$$

Multiple Paths Chain Rule



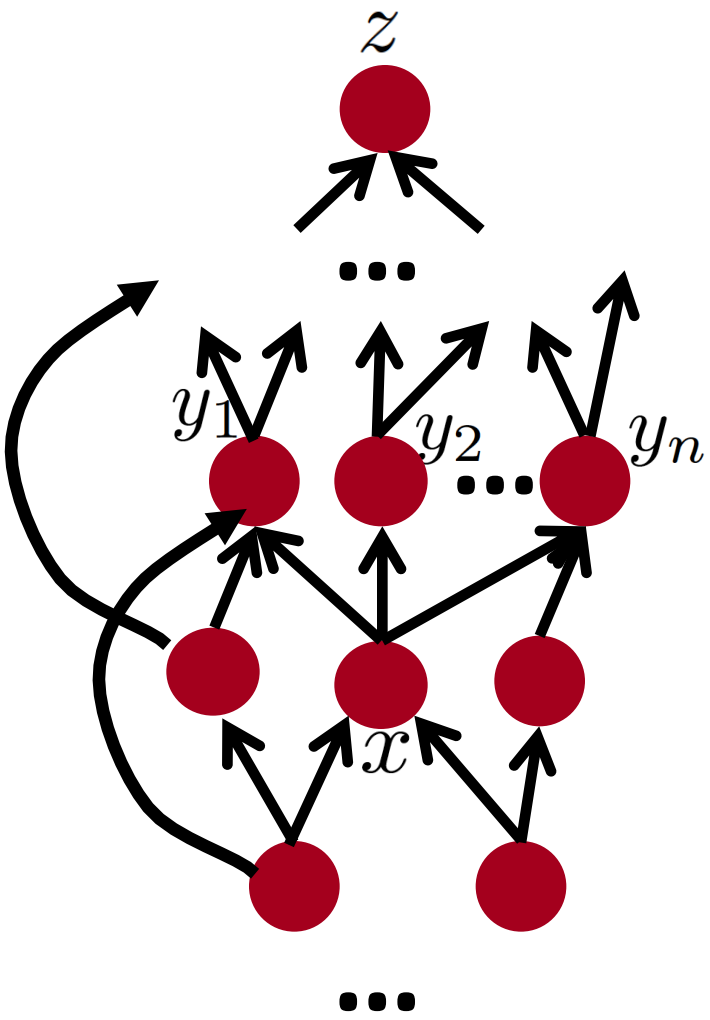
$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y_1}\frac{\partial y_1}{\partial x} + \frac{\partial z}{\partial y_2}\frac{\partial y_2}{\partial x}$$

Multiple Paths Chain Rule - General



$$\frac{\partial z}{\partial x} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x}$$

Chain Rule in Flow Graph

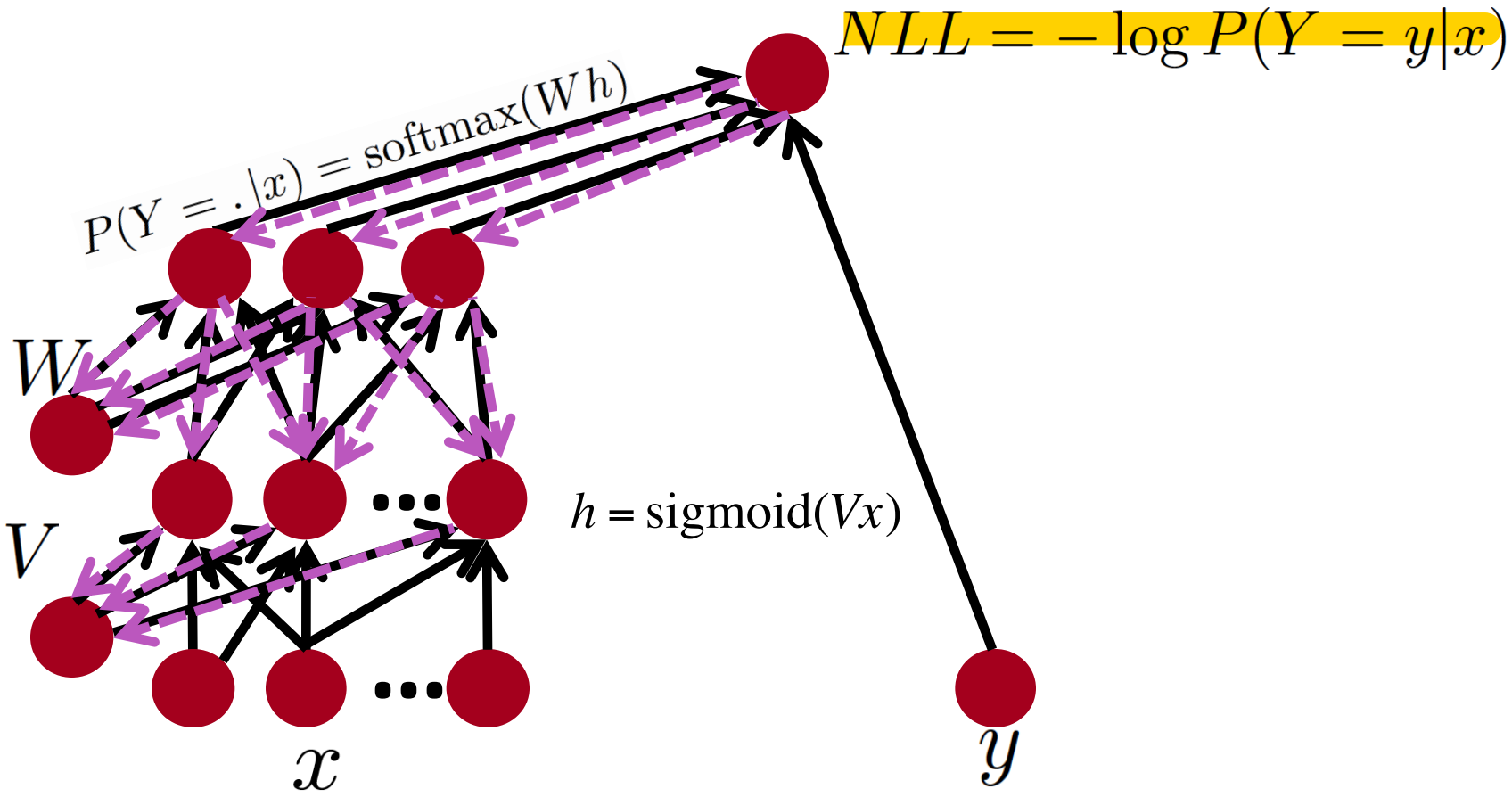


Flow graph: any directed acyclic graph
node = computation result
arc = computation dependency

$\{y_1, y_2, \ldots y_n\}$ = successors of x

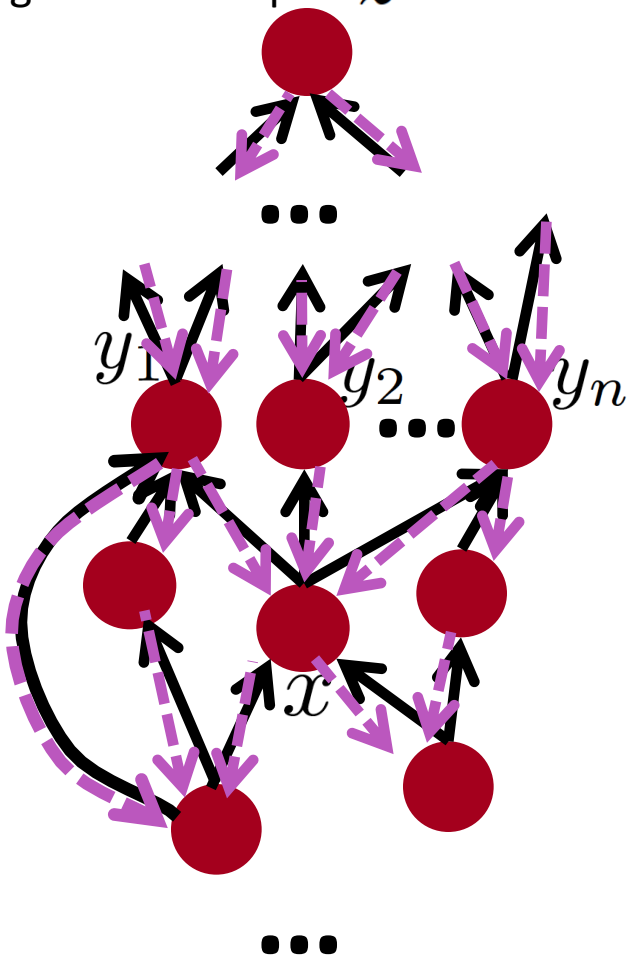
$$\frac{\partial z}{\partial x} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x}$$

Back-Prop in Multi-Layer Net



Back-Prop in General Flow Graph

Single scalar output z

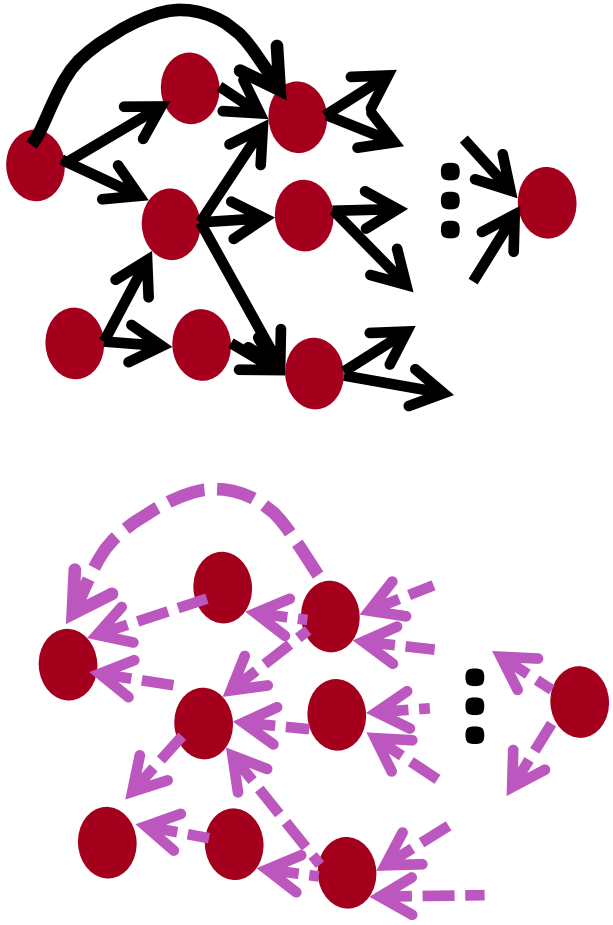


1. Fprop: visit nodes in topo-sort order
   - Compute value of node given predecessors
2. Bprop:
   - initialize output gradient = 1
   - visit nodes in reverse order:
     Compute gradient wrt each node using gradient wrt successors

$\{y_1, y_2, \ldots y_n\}$ = successors of x

$$\frac{\partial z}{\partial x} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x}$$

Automatic Differentiation



- The gradient computation can be **automatically inferred** from the symbolic expression of the fprop.
- Each node type needs to know how to compute its output and how to compute the gradient wrt its inputs given the gradient wrt its output.
- Easy and fast prototyping

Neural Tips and Tricks

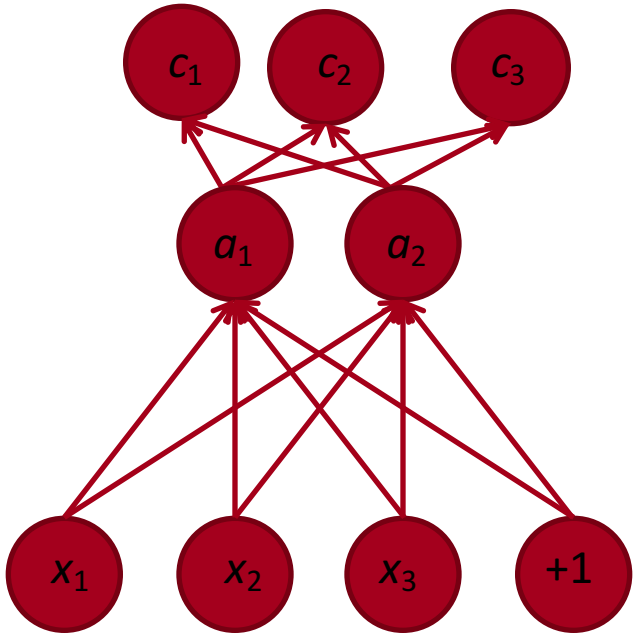
Multi-task learning / Weight sharing

- Base model: Neural network from last class but replaces the single scalar score with a *Softmax* classifier

- Training is again done via backpropagation

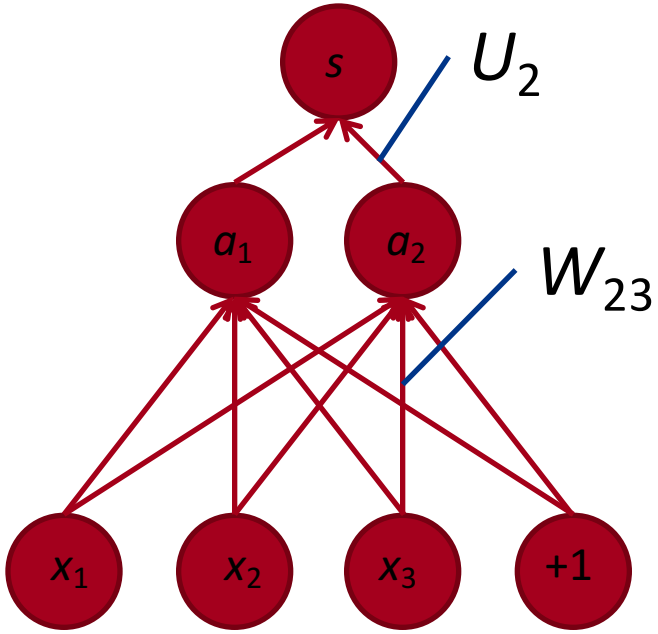
- NLP (almost) from scratch, Collobert et al. 2011

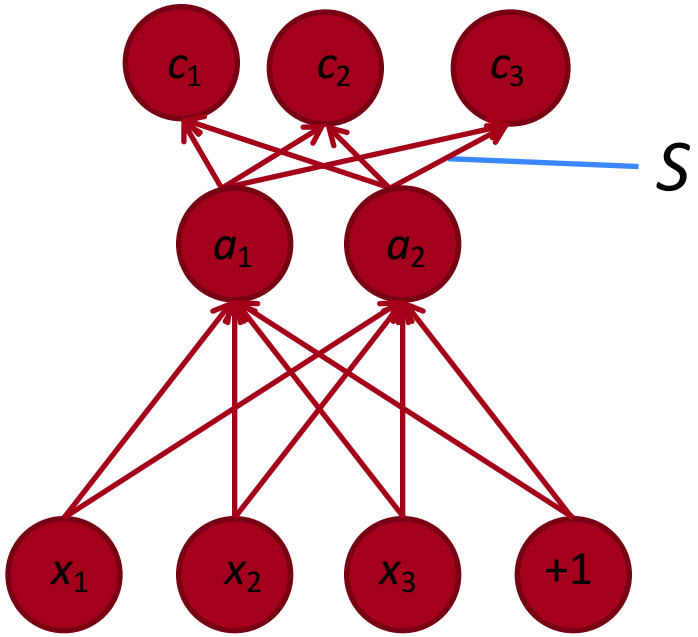
$$\hat{y} = softmax\left(W^{(S)} f(Wx + b)\right)$$



The model

- We already know the softmax classifier and how to optimize it
- The interesting twist in deep learning is that the input features x and their transformations in a hidden layer are also learned.
- Two final layers are possible:

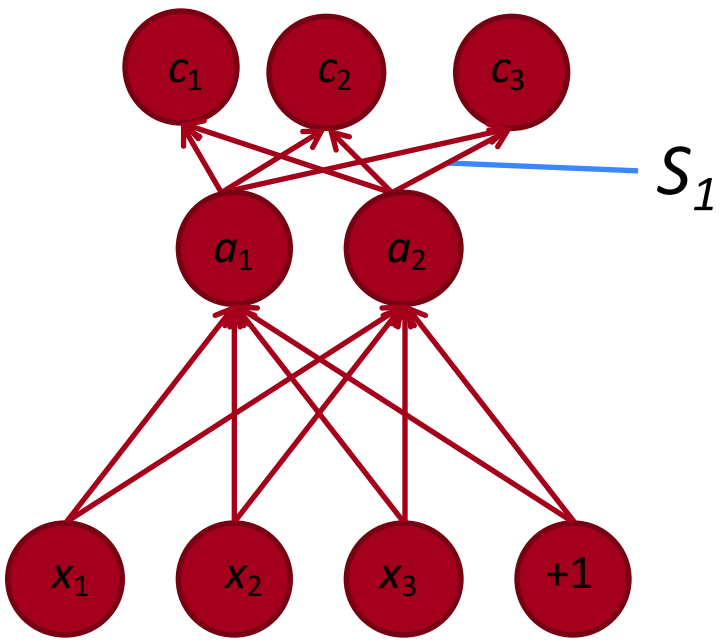


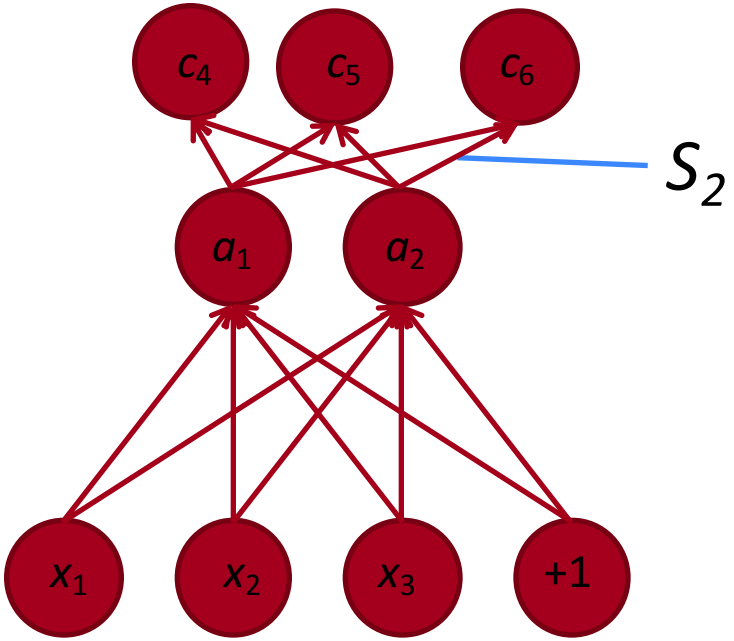


Multitask learning

- Main idea: We can share both the word vectors AND the hidden layer weights. Only the softmax weights are different.
- Cost function is just the sum of two cross entropy errors

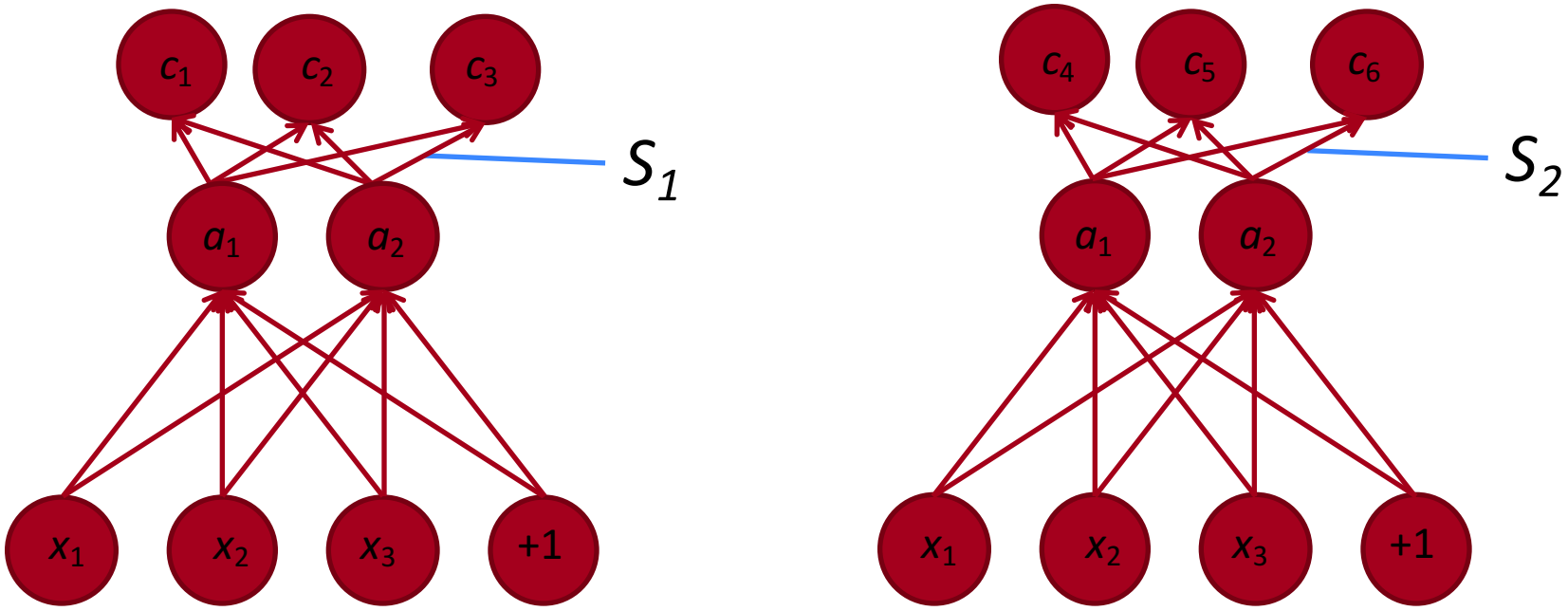
$$\hat{y}^{(1)} = softmax\left(W^{(S_1)} f(Wx + b)\right) \qquad \hat{y}^{(2)} = softmax\left(W^{(S_2)} f(Wx + b)\right)$$





The multitask model - Training

- Example: predict each window's center NER tag and POS tag: (example POS tags: DT, NN, NNP, JJ, JJS (superlative adj), VB,…)
- Efficient implementation: Same forward prop, compute errors on hidden vectors and add $\delta^{total} = \delta^{NER} + \delta^{POS}$



The secret sauce (sometimes) is the unsupervised word vector pre-training on a large text collection

	POS WSJ (acc.)	NER CoNLL (F1)
State-of-the-art*	97.24	89.31
Supervised NN	96.37	81.47
Word vector pre-training followed by supervised NN**	97.20	88.87
+ hand-crafted features***	97.29	89.59

* Representative systems: POS: (Toutanova et al. 2003), NER: (Ando & Zhang 2005)

** 130,000-word embedding trained on Wikipedia and Reuters with 11 word window, 100 unit hidden layer – then supervised task training

***Features are character suffixes for POS and a gazetteer for NER

Supervised refinement of the unsupervised word representation helps

	POS WSJ (acc.)	NER CoNLL (F1)
Supervised NN	96.37	81.47
NN with Brown clusters	96.92	87.15
Fixed embeddings*	**97.10**	**88.87**
C&W 2011**	**97.29**	**89.59**

* Same architecture as C&W 2011, but word embeddings are kept constant during the supervised training phase

** C&W is unsupervised pre-train + supervised NN + features model of last slide

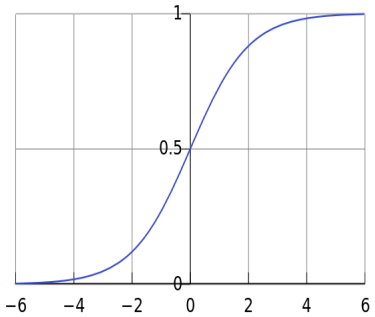
General Strategy for Successful NNets

1. Select network structure appropriate for problem
    1. Structure: Single words, fixed windows, sentence based, document level; bag of words, recursive vs. recurrent, CNN
    2. Nonlinearity
2. Check for implementation bugs with gradient checks
3. Parameter initialization
4. Optimization tricks
5. Check if the model is powerful enough to overfit
    1. If not, change model structure or make model “larger”
    2. If you can overfit: Regularize

Non-linearities: What's used

logistic ("sigmoid")

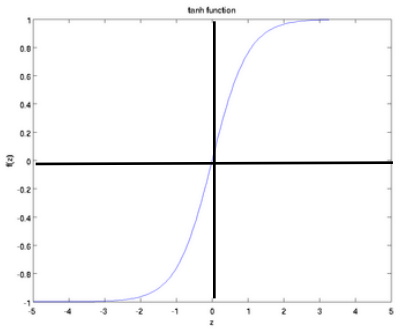
$$f(z) = \frac{1}{1 + \exp(-z)}.$$



$$f'(z) = f(z)(1 - f(z))$$

tanh

$$f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}},$$



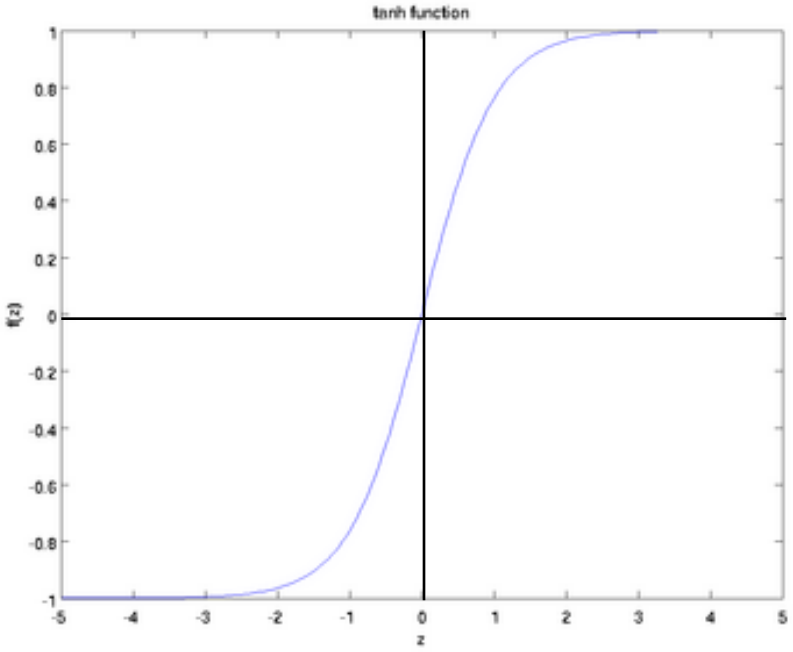
$$f'(z) = 1 - f(z)^2$$

tanh is just a rescaled and shifted sigmoid

$$\tanh(z) = 2\text{logistic}(2z) - 1$$

For many models, tanh is the best!

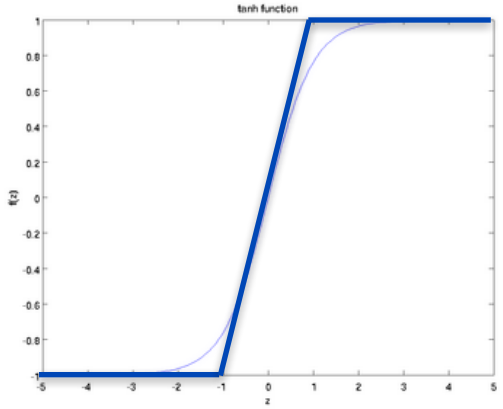
- In comparison to sigmoid:
- At initialization: values close to 0
- Faster convergence in practice
- Like sigmoid: Nice derivative: $f'(z) = 1 - tanh^2(z)$



Non-linearities: There are various other choices

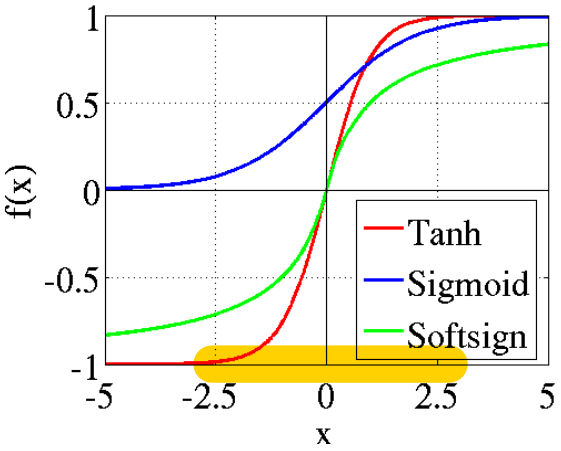
hard tanh

$$\text{HardTanh}(x) = \begin{cases} -1 & \text{if } x < -1 \\ x & \text{if } -1 <= x <= 1 \\ 1 & \text{if } x > 1 \end{cases}$$



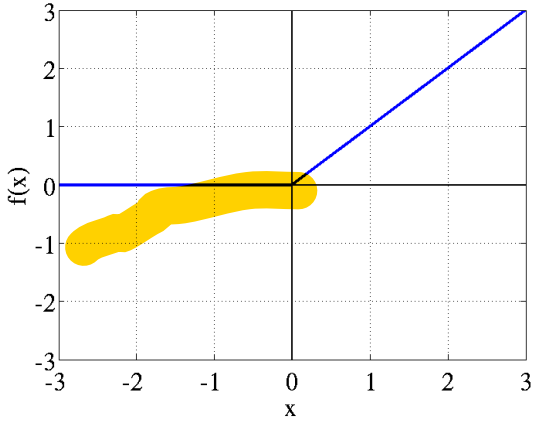
soft sign

$$\text{softsign}(z) = \frac{a}{1+|a|}$$



rectified linear (ReLu)

$$\text{rect}(z) = \max(z, 0)$$



- hard tanh similar but computationally cheaper than tanh and saturates hard.
- Glorot and Bengio, *AISTATS* 2011 discuss softsign and rectifier

MaxOut Network

A recent type of nonlinearity/network

Goodfellow et al. (2013)

Where $f_i(z) = \max_{j \in [1,k]} z_{ij}$

$$z_{ij} = x^T W_{..ij} + b_{ij}$$

This function also becomes a universal approximator when stacked in multiple layers

State of the art on several image datasets

Gradient Checks are Awesome!

- Allow you to know that there are no bugs in your neural network implementation!
- Steps:
  1. Implement your gradient
  2. Implement a finite difference computation by looping through the parameters of your network, adding and subtracting a small epsilon (~10^{-4}) and estimate derivatives

$$f'(\theta) \approx \frac{J\left(\theta^{(i+)}\right) - J\left(\theta^{(i-)}\right)}{2\epsilon} \qquad \theta^{(i+)} = \theta + \epsilon \times e_i$$

  3. Compare the two and make sure they are almost the same

Using gradient checks and model simplification

- If you gradient fails and you don't know why?
- What now? **Create a very tiny synthetic model and dataset**
- Simplify your model until you have no bug!
- Example: Start from simplest model then go to what you want:
  - Only softmax on fixed input
  - Backprop into word vectors and softmax
  - Add single unit single hidden layer
  - Add multi unit single layer
  - Add bias
  - Add second layer single unit, add multiple units, bias
  - Add one softmax on top, then two softmax layers

General Strategy

1. Select appropriate Network Structure
    1. Structure: Single words, fixed windows vs Recursive Sentence Based vs Bag of words
    2. Nonlinearity
2. Check for implementation bugs with gradient check
3. Parameter initialization
4. Optimization tricks
5. Check if the model is powerful enough to overfit
    1. If not, change model structure or make model “larger”
    2. If you can overfit: Regularize

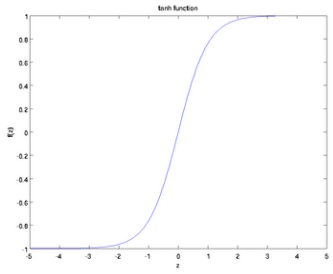
Parameter Initialization

- Initialize hidden layer biases to 0 and output (or reconstruction) biases to optimal value if weights were 0 (e.g., mean target or inverse sigmoid of mean target).

- Initialize weights ~ Uniform(−*r*, *r*), *r* inversely proportional to fan-in (previous layer size) and fan-out (next layer size):

$$\sqrt{6/(\text{fan-in} + \text{fan-out})}$$

for tanh units, and 4x bigger for sigmoid units [Glorot AISTATS 2010]



Stochastic Gradient Descent (SGD)

- Gradient descent uses total gradient over all examples per update, SGD updates after only 1 or few examples:

$$\theta^{new} = \theta^{old} - \alpha \nabla_\theta J_t(\theta)$$

- J_t = loss function at current example, μ = parameter vector, ® = learning rate.
- Ordinary gradient descent as a batch method is very slow, should never be used. Use 2nd order batch method such as L-BFGS.
- On large datasets, SGD usually wins over all batch methods. On smaller datasets L-BFGS or Conjugate Gradients win. Large-batch L-BFGS extends the reach of L-BFGS [Le et al. ICML 2011].

Mini-batch Stochastic Gradient Descent (SGD)

- Gradient descent uses total gradient over all examples per update, SGD updates after only 1 example
- Most commonly used now: Mini batches

- Size of each mini batch B: 20 to 1000:

$$\theta^{new} = \theta^{old} - \alpha \nabla_{\theta} J_{t:t+B}(\theta)$$

- Helps parallelizing any model by computing gradients for multiple elements of the batch in parallel

Improvement over SGD: Momentum

- Idea: Add a fraction v of previous update to current one
- When the gradient keeps pointing in the same direction, this will increase the size of the steps taken towards the minimum
- Reduce global learning rate ® when using a lot of momentum

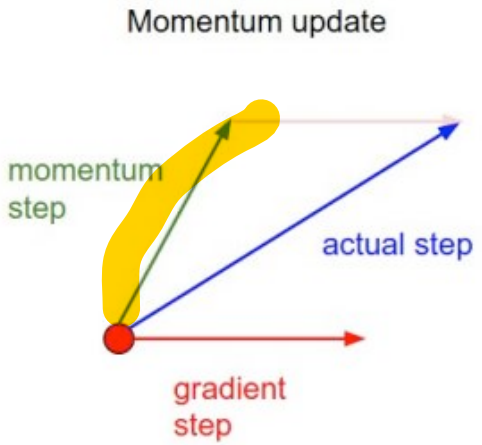
- Update rule:

$$v = \mu v - \alpha \nabla_\theta J_t(\theta)$$

$$\theta^{new} = \theta^{old} + v$$

- v is initialized at 0
- Common: ^{1} = 0.9
- Momentum often increased after some epochs (0.5 → 0.99)

Intuition Momentum



- Adds friction (*momentum ~ misnomer)*
- Parameters build up velocity in direction of consistent gradient
- Simple convex function optimization dynamics
  without momentum with momentum:

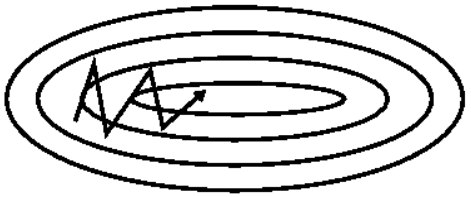


Figure from https://www.willamette.edu/~gorr/classes/cs449/momrate.html

Learning Rates

- Simplest recipe: keep it fixed and use the same for all parameters. Standard: $\theta^{new} = \theta^{old} - \alpha \nabla_\theta J_t(\theta)$

- Better results by allowing learning rates to decrease Options:
  - Reduce by 0.5 when validation error stops improving
  - Reduction by $O(1/t)$ because of theoretical convergence guarantees, e.g.: $\alpha = \frac{\alpha_0 \tau}{\max(t, \tau)}$
    with hyper-parameters ε_0 and τ and t is iteration numbers
  - Better yet: No hand-set learning of rates by using AdaGrad →

Adagrad

- Adaptive learning rates for each parameter!
- Related paper: Adaptive Subgradient Methods for Online Learning and Stochastic Optimization, Duchi et al. 2010

- Learning rate is adapting differently for each parameter and rare parameters get larger updates than frequently occurring parameters. Word vectors!

- Let $g_{t,i} = \frac{\partial}{\partial \theta_i^t} J_t(\theta)$, then: $\theta_{t,i} = \theta_{t-1,i} - \frac{\alpha}{\sqrt{\sum_{\tau=1}^{t} g_{\tau,i}^2}} g_{t,i}$

General Strategy

1. Select appropriate Network Structure
    1. Structure: Single words, fixed windows vs Recursive Sentence Based vs Bag of words
    2. Nonlinearity
2. Check for implementation bugs with gradient check
3. Parameter initialization
4. Optimization tricks
5. Check if the model is powerful enough to overfit
    1. If not, change model structure or make model “larger”
    2. If you can overfit: Regularize

Assuming you found the right network structure, implemented it correctly, optimize it properly and you can make your model overfit on your training data.

Now, it’s time to regularize

Prevent Overfitting: Model Size and Regularization

- Simple first step: Reduce model size by lowering number of units and layers and other parameters
- Standard L1 or L2 regularization on weights
- Early Stopping: Use parameters that gave best validation error
- Sparsity constraints on hidden activations, e.g., add to cost:

$$KL\left(1/N \sum_{n=1}^{N} a_i^{(n)} \| 0.0001\right)$$

Prevent Feature Co-adaptation

Dropout (Hinton et al. 2012)

- Training time: at each instance of evaluation (in online SGD-training), randomly set 50% of the inputs to each neuron to 0
- Test time: halve the model weights (now twice as many)
- This prevents feature co-adaptation: A feature cannot only be useful in the presence of particular other features
- In a single layer: A kind of middle-ground between Naïve Bayes (where all feature weights are set independently) and logistic regression models (where weights are set in the context of all others)
- Can be thought of as a form of model bagging
- It also acts as a strong regularizer

Deep Learning Tricks of the Trade

- Y. Bengio (2012), "Practical Recommendations for Gradient-Based Training of Deep Architectures"

  - Unsupervised pre-training
  - Stochastic gradient descent and setting learning rates
  - Main hyper-parameters
    - Learning rate schedule & early stopping, Minibatches, Parameter initialization, Number of hidden units, regularization (= weight decay)
  - How to efficiently search for hyper-parameter configurations
    - Short answer: **Random hyperparameter search** (!)

- Some more advanced and recent tricks in later lectures

Language Models

Language Models

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

Probability is usually conditioned on window of n previous words :

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

Very useful for a lot of tasks:

Can be used to determine whether a sequence is a good / grammatical translation or speech utterance

Original neural language model

A Neural Probabilistic Language Model, Bengio et al. 2003

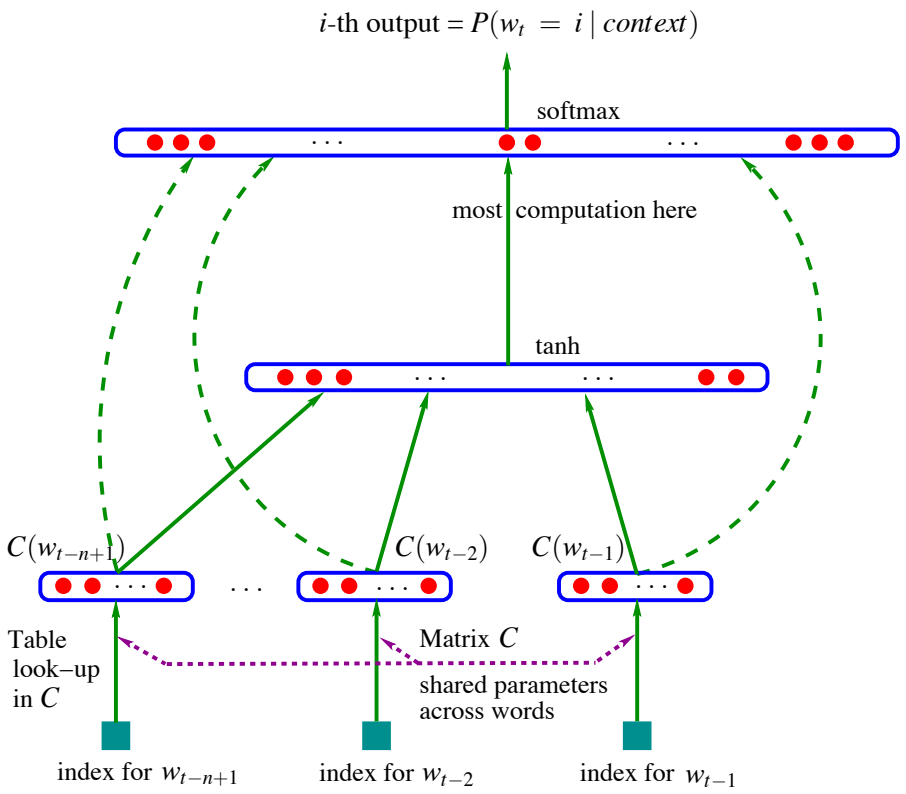
$$\hat{y} = softmax\left(W^{(2)} f\left(W^{(1)} x + b^{(1)}\right) + W^{(3)} x + b^{(3)}\right)$$

Original equations:

$$y = b + Wx + U\tanh(d + Hx)$$

$$\hat{P}(w_t | w_{t-1}, \cdots w_{t-n+1}) = \frac{e^{y_{w_t}}}{\sum_i e^{y_i}}.$$

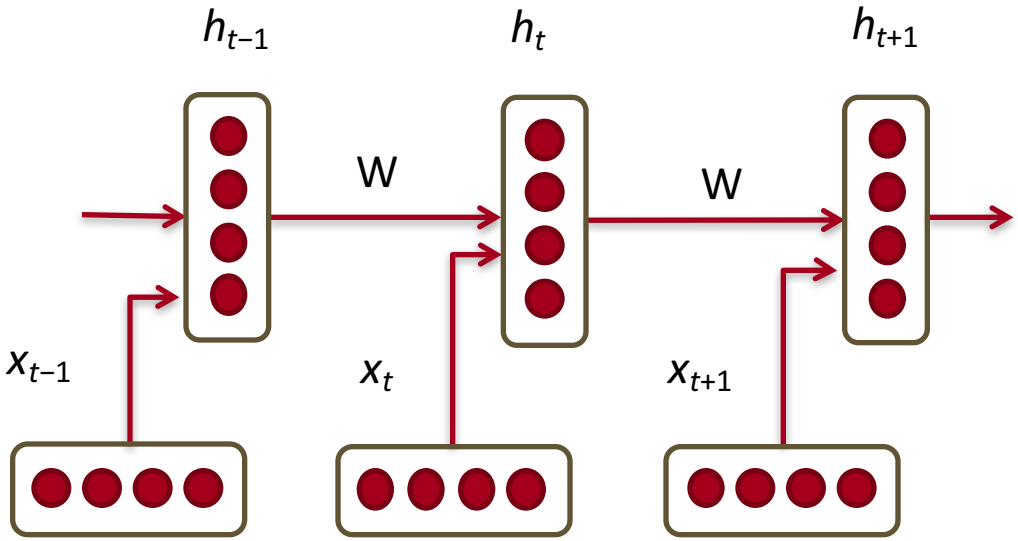
Problem: Fixed window of context for conditioning :(



Recurrent Neural Networks

Recurrent Neural Networks!

Solution: Condition the neural network on all previous words and tie the weights at each time step



Recurrent Neural Network language model

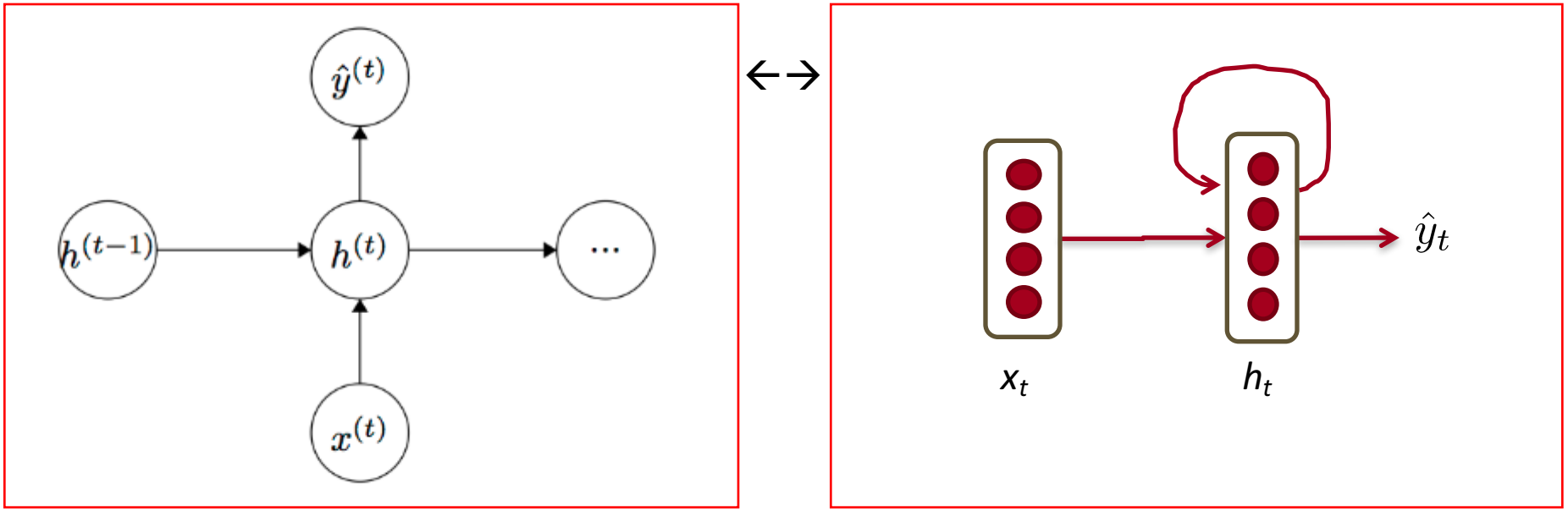
Given list of word **vectors**: $x_1, \ldots, x_{t-1}, x_t, x_{t+1}, \ldots, x_T$

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

Main idea: we use the same set of W weights at all time steps!

Everything else is the same:

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

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

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

$h_0 \in \mathbb{R}^{D_h}$ 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

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

Recurrent Neural Network language model

$\hat{y} \in \mathbb{R}^{|V|}$ 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 commonly: Perplexity: 2^J

Lower is better!

Training RNNs is hard

- The gradient is a product of Jacobian matrices, each associated with a step in the forward computation.

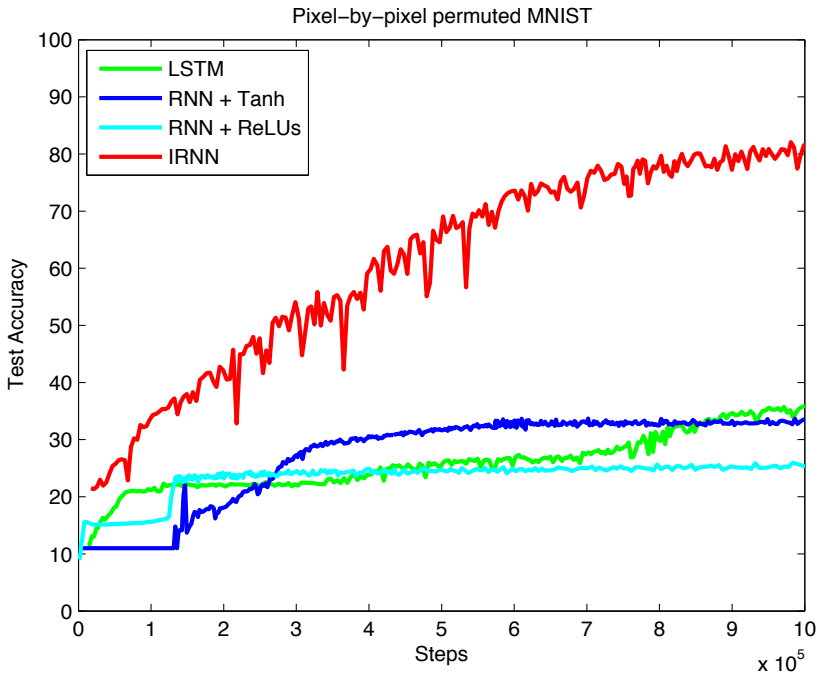
- Multiply the same matrix at each time step during backprop

$$\delta^{(l)} = \left((W^{(l)})^T \delta^{(l+1)}\right) \circ f'(z^{(l)}),$$

- 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**

Initialization trick for RNNs!

- Initialize $W^{(hh)}$ to be the 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 last week (!) in *A Simple Way to Initialize Recurrent Networks of Rectified Linear* Units, Le et al. 2015

Long-Term dependencies and clipping trick

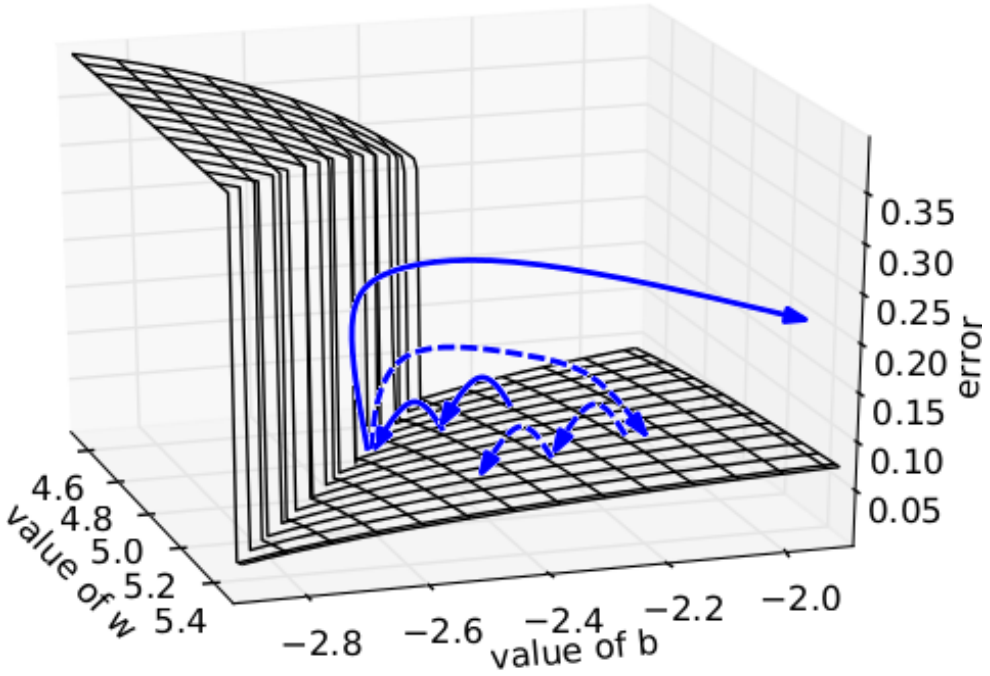
- The solution first introduced by Mikolov is to clip gradients to a maximum value.

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



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

Summary

Tips and tricks to become a deep neural net ninjia

Introduction to Recurrent Neural Network

Next week:

→ Lecture on TensorFlow for practical implementations, PSet and project

→ More RNN details and variants (LSTMs and GRUs)

→ Exciting times!