CS224d
Deep NLP

Lecture 6:
Neural Tips and Tricks
+
Recurrent Neural Networks

Richard Socher
richard@metamind.io

Overview Today:

- Useful NNet techniques / tips and tricks:
  - Multi-task learning
  - Nonlinearities
  - Finite difference gradient check
  - Momentum, AdaGrad

- Language Models

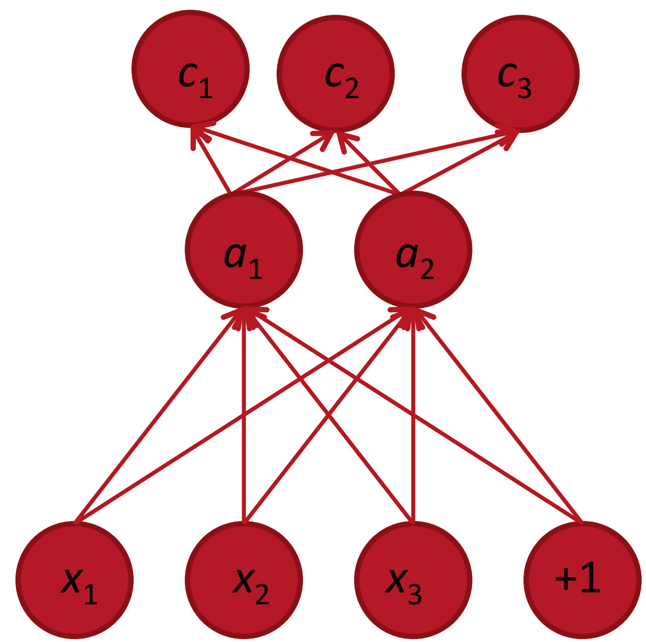
- Recurrent Neural Networks

Deep Learning General Strategy and Tricks

Multi-task learning / Weight sharing

- Similar to neural network from last class but replaces the single scalar score with a *Softmax* classifier

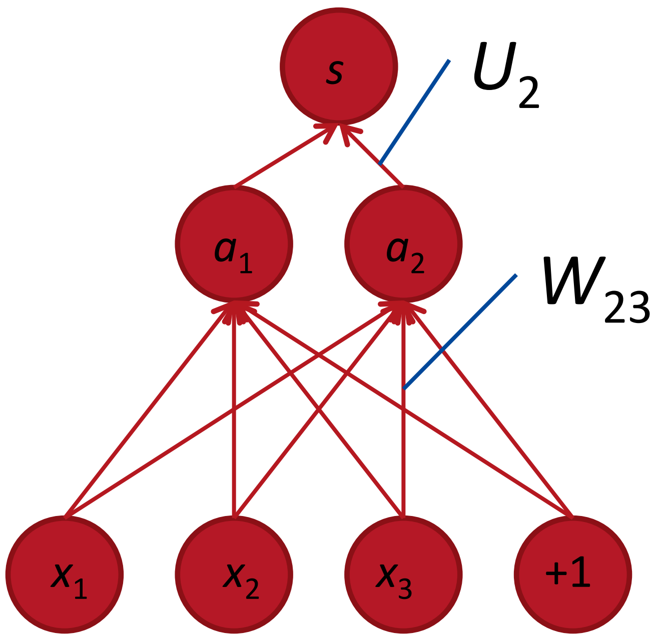
$$\hat{y} = softmax\left(W^{(S)} f(Wx + b)\right)$$

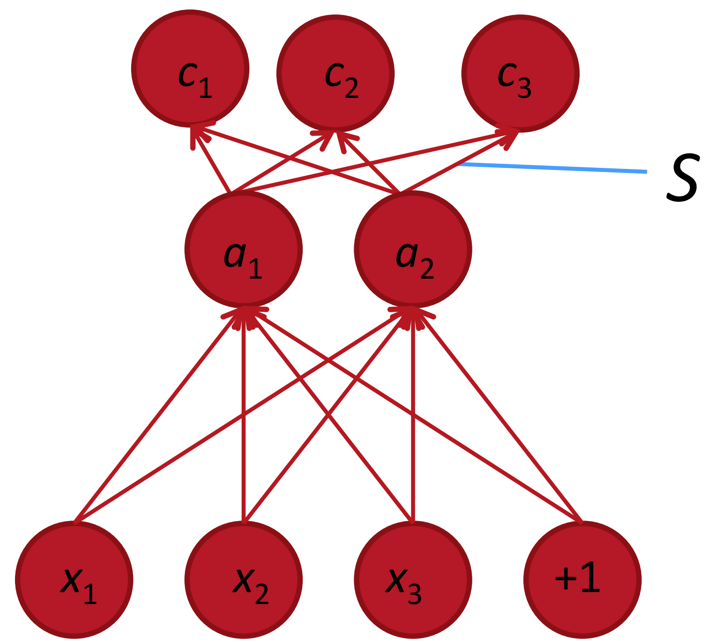


- Training is again done via backpropagation which gives an error similar to the score in the scoring learning model
- NLP (almost) from scratch, Collobert et al. 2011

The Model - Training

- We already know the softmax classifier and how to optimize it
- The interesting twist in deep learning is that the input features x are also learned, similar to learning with a score:

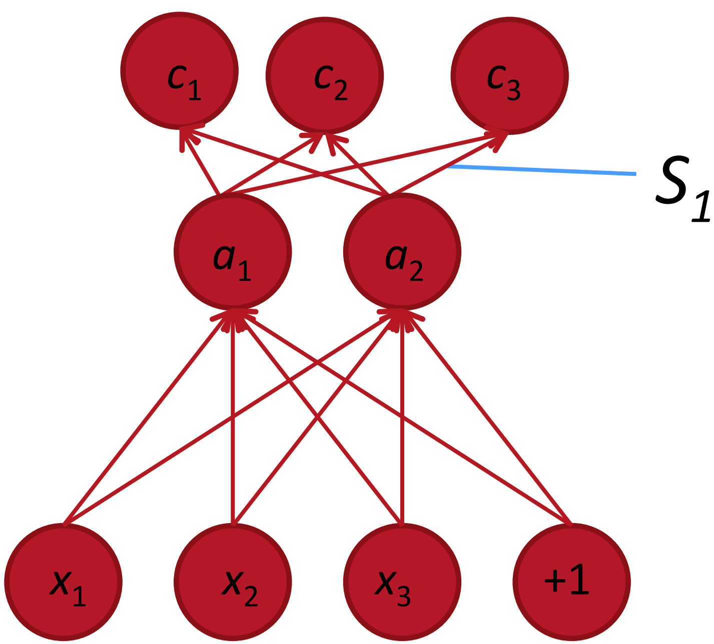


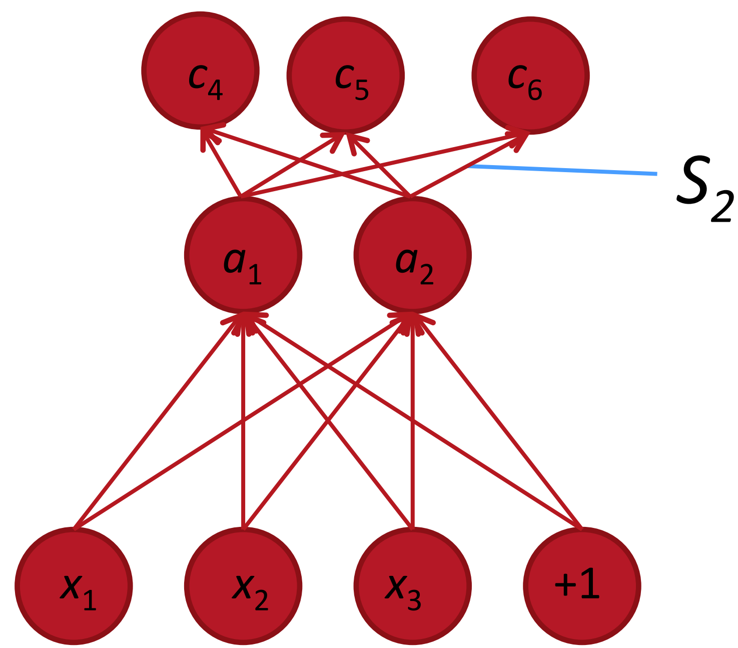


The Model - Training

- Main additional 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

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





The secret sauce 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

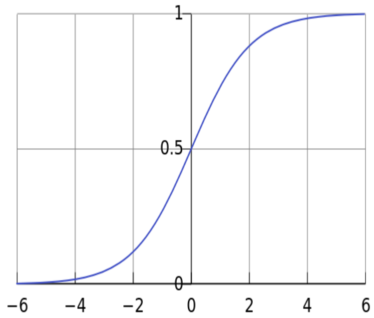
General Strategy for Successful NNets

1. Select network structure appropriate for problem
    1. Structure: Single words, fixed windows, bag of words, recursive vs. recurrent, CNN, sentence based vs. document
    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")

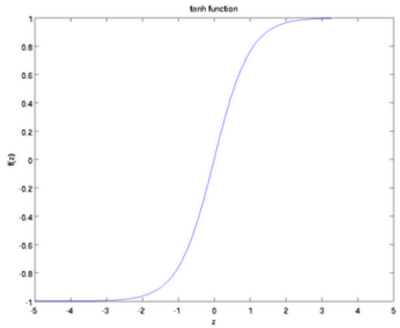
$$f(z) = \frac{1}{1+\exp(-z)}.$$



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

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 often performs well for deep nets

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

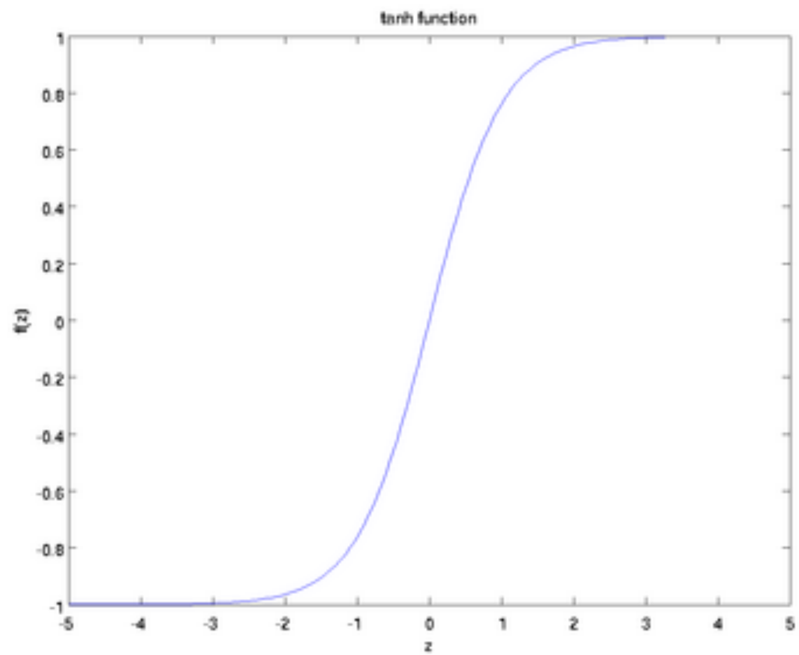
For many models, tanh is the best!

- In comparison to sigmoid:

- At initialization: values close to 0

- Faster convergence in practice

- Like sigmoid: Nice derivative: $f'(z) = 1 - tanh^2(z)$



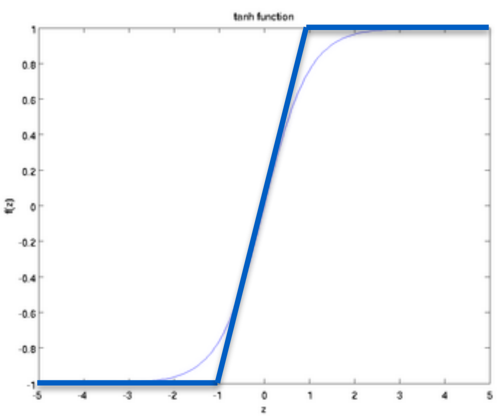
Non-linearities: There are various other choices

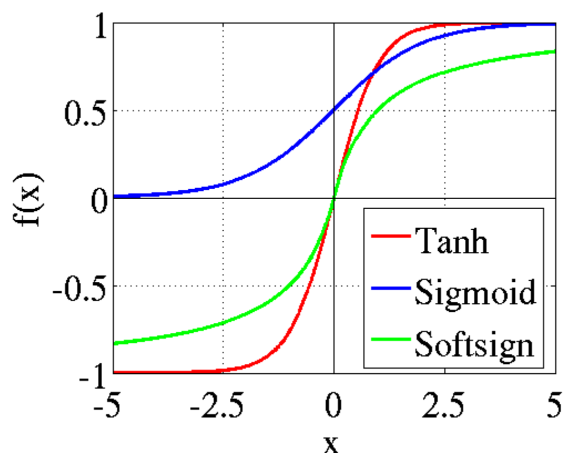
hard tanh | soft sign | **rectified linear (ReLu)**

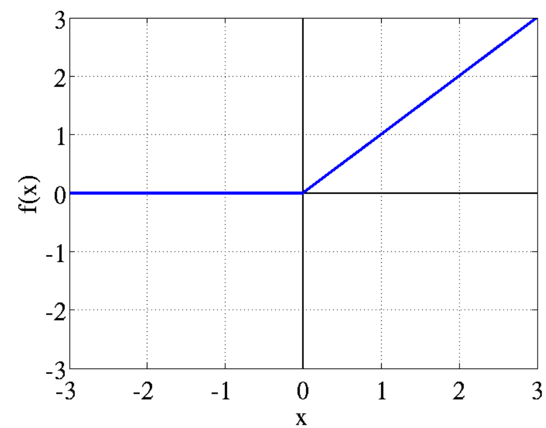
$$\mathrm{HardTanh}(x) = \begin{cases} -1 & \text{if } x < -1 \\ x & \text{if } -1 <= x <= 1 \\ 1 & \text{if } x > 1 \end{cases}$$

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

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







- 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 $$f_i(z) = \max_{j \in [1,k]} z_{ij}$$
$$z_{ij} = x^T W_{\cdot\cdot ij} + b_{ij}$$

This function too is a universal approximator

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

Gradient Checks are Awesome!

- If you gradient fails and you don't know why?
- What now?
- 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 two softmax units

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, θ = parameter vector, α = 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].

Learning Rates

- Simplest recipe: keep it fixed and use the same for all parameters.
- Collobert scales them by the inverse of square root of the fan-in of each neuron
- Better results can generally be obtained by allowing learning rates to decrease, typically in $O(1/t)$ because of theoretical convergence guarantees, e.g., $\epsilon_t = \frac{\epsilon_0 \tau}{\max(t, \tau)}$ with hyper-parameters ε_0 and τ
- Better yet: No hand-set learning rates by using L-BFGS or AdaGrad (Duchi, Hazan, & Singer 2011)

Adagrad

- Standard SGD, fixed α: $\theta^{new} = \theta^{old} - \alpha \nabla_\theta J_t(\theta)$
- Instead: Adaptive learning rates!
- 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: $\theta_{t,i} = \theta_{t-1,i} - \frac{\alpha}{\sqrt{\sum_{\tau=1}^{t} g_{\tau,i}^2}} g_{t,i}$

Long-Term Dependencies and Clipping Trick

- In very deep networks such as recurrent networks, the gradient is a product of Jacobian matrices, each associated with a step in the forward computation. This can become very small or very large quickly [Bengio et al 1994], and the locality assumption of gradient descent breaks down.

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

- The solution first introduced by Mikolov is to clip gradients to a maximum value. Makes a big difference in RNNs.

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
- 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
    - L1 or L2 weight decay
    - Sparsity regularization
  - How to efficiently search for hyper-parameter configurations
    - Short answer: **Random hyperparameter search** (!)

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

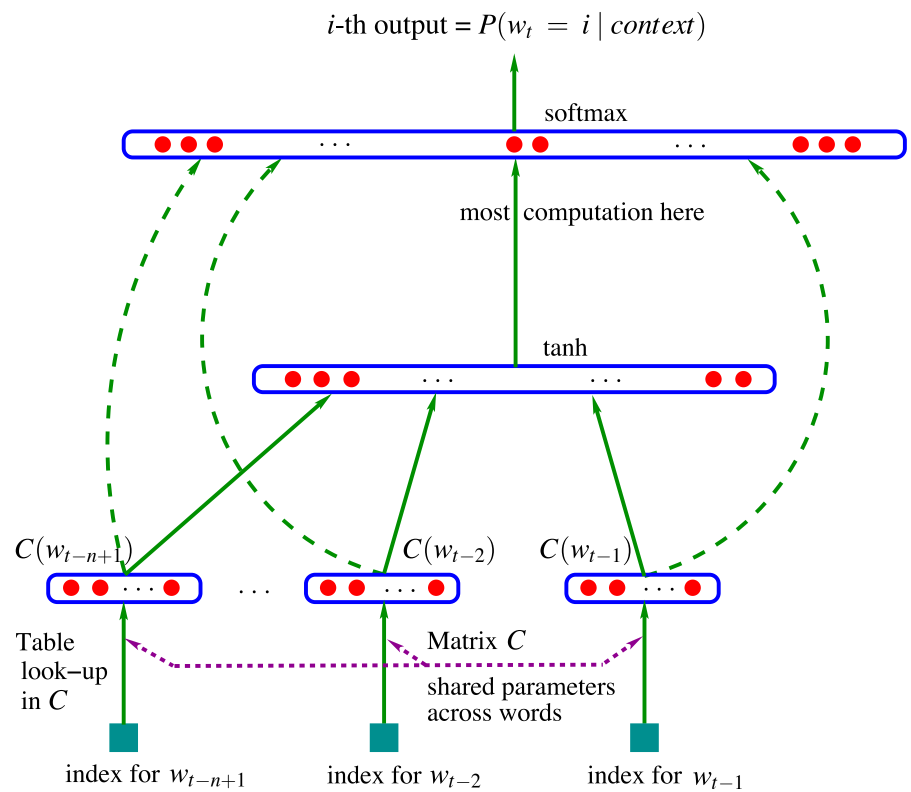
$$\hat{y} = softmax\left(W^{(2)} f\left(W^{(1)}x + b^{(1)}\right) + W^{(3)}x + b^{(3)}\right)$$

Original equations:

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

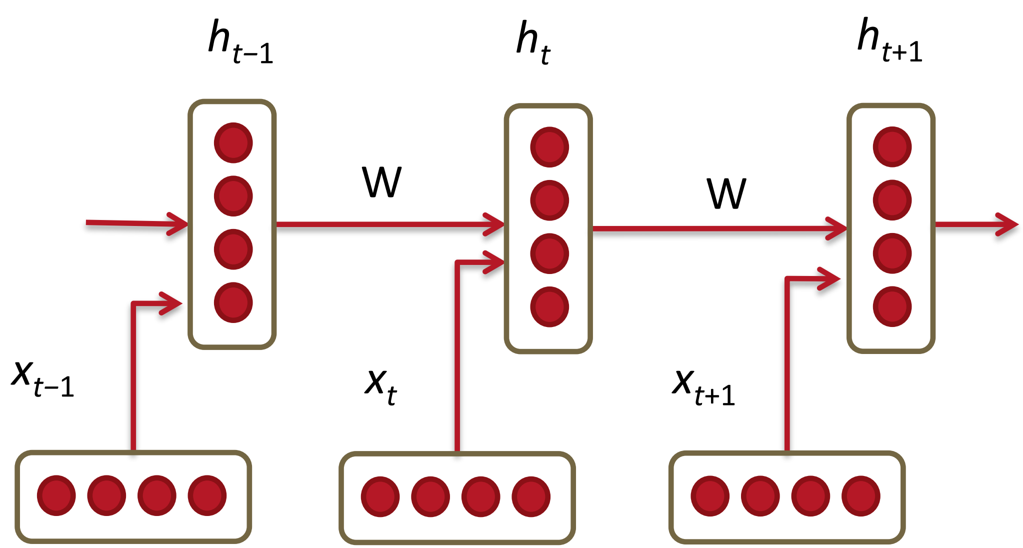
$$\hat{P}(w_t | w_{t-1}, \cdots w_{t-n+1}) = \frac{e^{y_{w_t}}}{\sum_i e^{y_i}}.$$

Problem: Fixed window of context for conditioning :(



Recurrent Neural Networks!

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

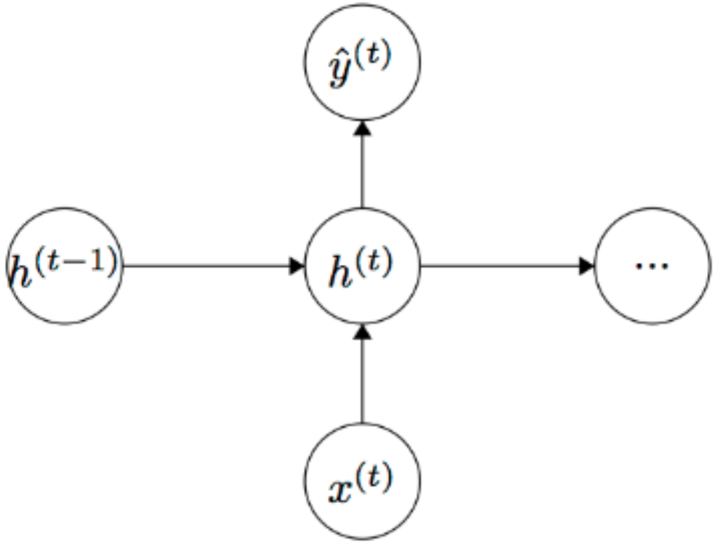


Recurrent Neural Network language model

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

At a single time step:

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



Recurrent Neural Network language model

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

Everything else is the same:

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

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

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

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

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

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