USC Viterbi
School of Engineering
Department of Computer Science

CSCI 544
Applied Natural Language Processing

Mohammad Rostami
USC Computer Science Department

Logistical Notes

- Paper Presentation:

- Selection deadline: 09/26
- Presentation: 09/26 – 10/05

- Syllabus
- PyTorch lecture: HW3

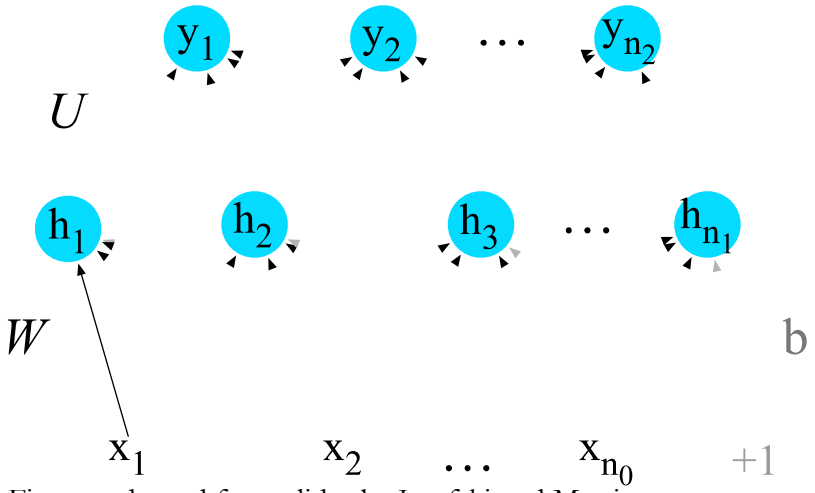
- Coursera: Deep Neural Networks with PyTorch

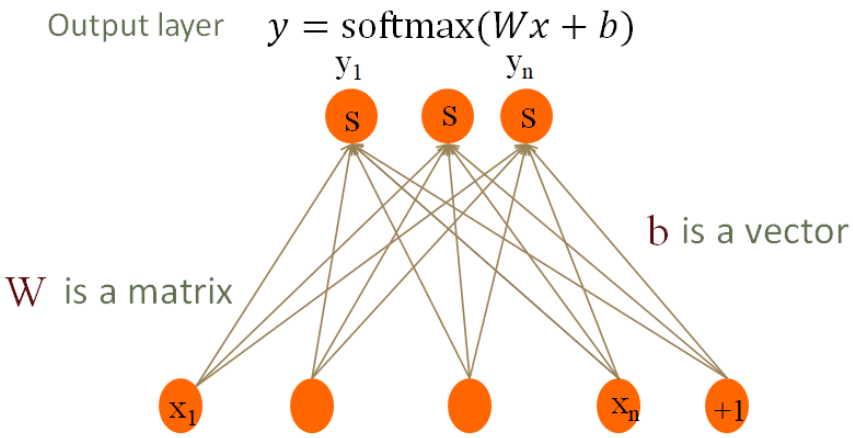
Feedforward Neural Networks

- Is a function approximator where the output depends on a single input

$$y = f(x)$$

- The inputs are assumed to be independent from each other





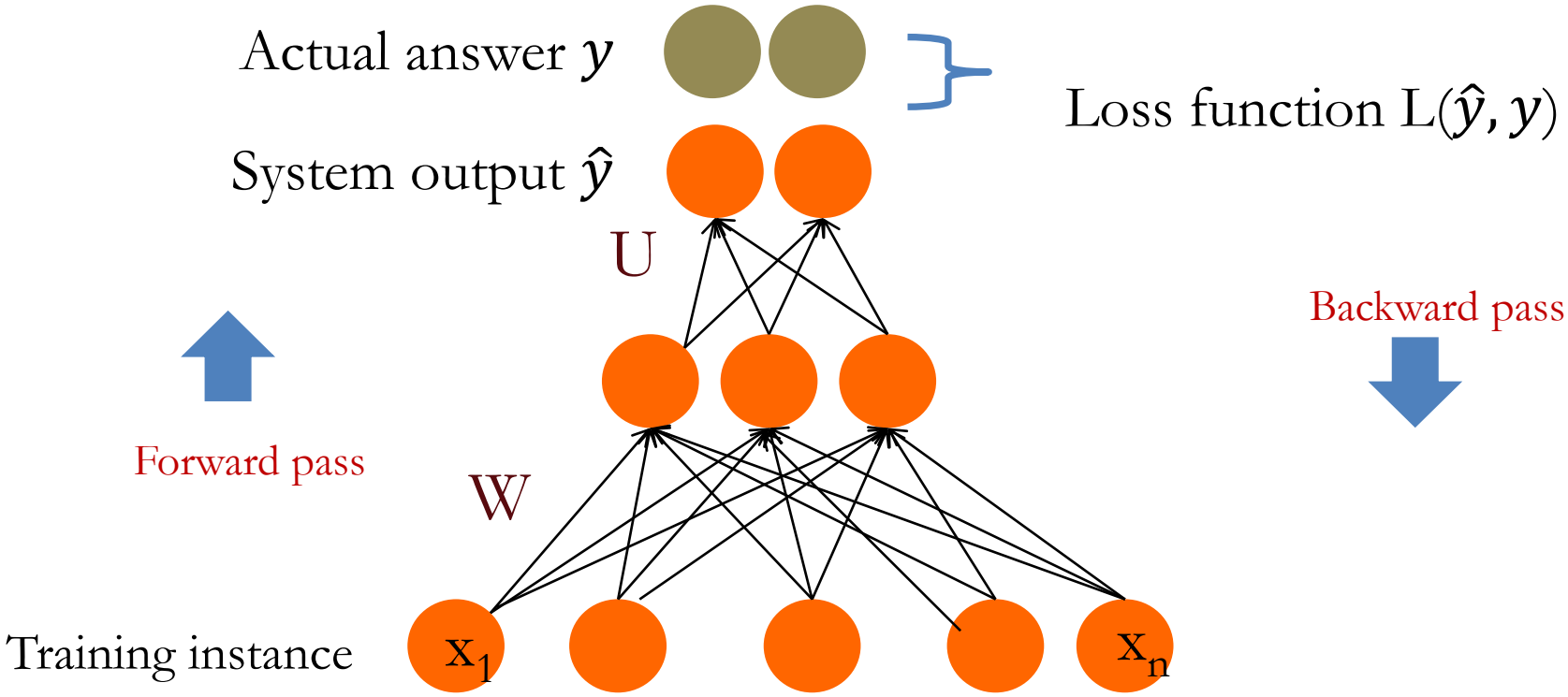
Figures adopted from slides by Jurafski and Martin

Universal Approximation Theorem

- MLPs can represent a wide range of functions given appropriate values for the weights. (George Cybenko in 1989)
  - Given sufficient layers, i.e., deep nets
  - Given sufficient nodes, i.e., wide nets
- Only existential result: it merely states that approximating most given functions is possible but does not provide the solution.

Training Feedforward Networks

- Backpropagation algorithm: an iterative algorithm to learn network weights using annotated data

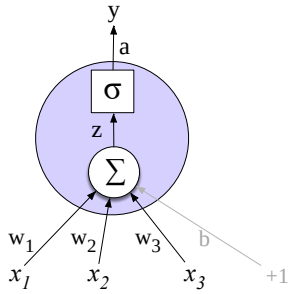


Figures adopted from slides by Jurafski and Martin

Training Feedforward Networks

- For every training data point (x, y)
  - Run *forward* computation to find model estimate $\hat{y}$
  - Run *backward* computation to update weights:
    - For every output node
    - Compute loss L between true y and the estimated $\hat{y}$
    - For every weight w from hidden layer to the output layer

    Update the weight using gradient descent $\frac{d}{dw} L(f(x; w), y)$
    - For all other nodes
    - Assess how much blame it deserves for the current answer



$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial y} \frac{\partial y}{\partial z} \frac{\partial z}{\partial w_i}$$

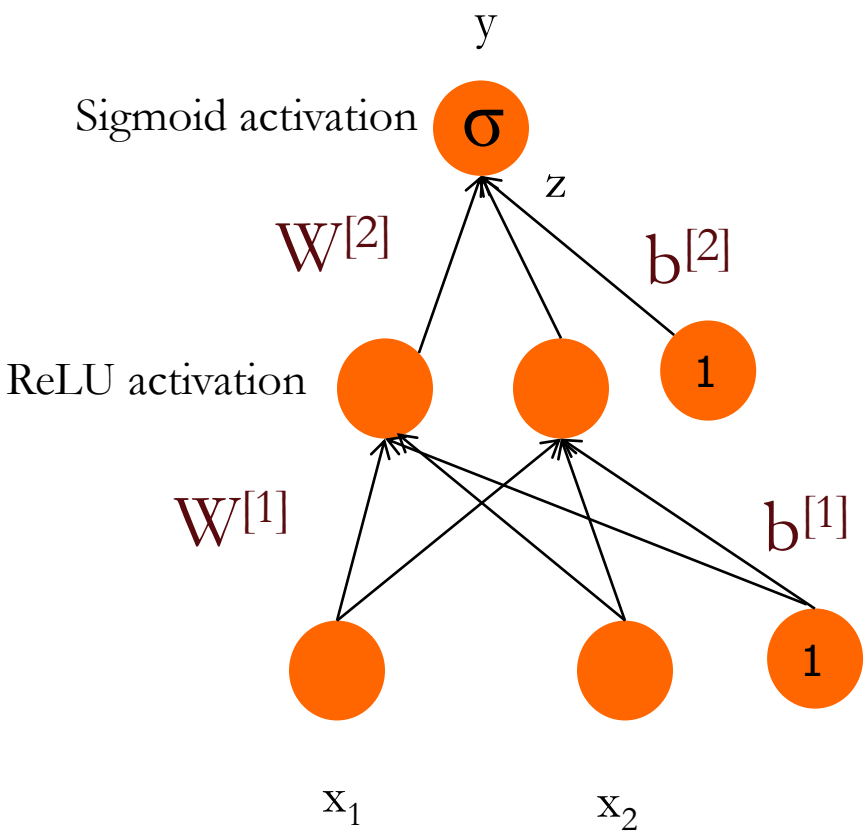
Backpropagation for Two Layer Network Output

$$
\begin{aligned}
z^{[1]} &= W^{[1]}\mathbf{x} + b^{[1]} \\
a^{[1]} &= \text{ReLU}(z^{[1]}) \\
z^{[2]} &= W^{[2]}a^{[1]} + b^{[2]} \\
a^{[2]} &= \sigma(z^{[2]}) \\
\hat{y} &= a^{[2]}
\end{aligned}
$$

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial z}\frac{\partial z}{\partial w_i}$$

$$\frac{d\,\text{ReLU}(z)}{dz} = \begin{cases} 0 & for \;\; z < 0 \\ 1 & for \;\; z \geq 0 \end{cases}$$

$$\frac{\partial y}{\partial z} = \frac{d\sigma(z)}{dz} = \sigma(z)(1-\sigma(z))$$



Backpropagation for Two Layer Network Output

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial z}\frac{\partial z}{\partial w_i}$$

$$L(\hat{y}, y) = -(\mathrm{y}\log(\hat{y}) + (1-y)\log(1-\hat{y}))$$

$$y = \sigma(z)$$

$$z = \sum w_i h_i + \mathrm{b}$$

$$\begin{aligned} z^{[1]} &= W^{[1]}\mathbf{x} + b^{[1]} \\ a^{[1]} &= \mathrm{ReLU}(z^{[1]}) \\ z^{[2]} &= W^{[2]}a^{[1]} + b^{[2]} \\ a^{[2]} &= \sigma(z^{[2]}) \\ \hat{y} &= a^{[2]} \end{aligned}$$

Notational Simplicity

$$\frac{\partial L}{\partial y} = -\left(\left(y\frac{\partial \log(a)}{\partial a}\right) + (1-\mathrm{y})\frac{\partial \log(1-a)}{\partial a}\right)$$

$$= -\left(\left(y\frac{1}{a}\right) + (1-\mathrm{y})\frac{1}{1-a}(-1)\right) = -\left(\frac{y}{a} + \frac{y-1}{1-a}\right)$$

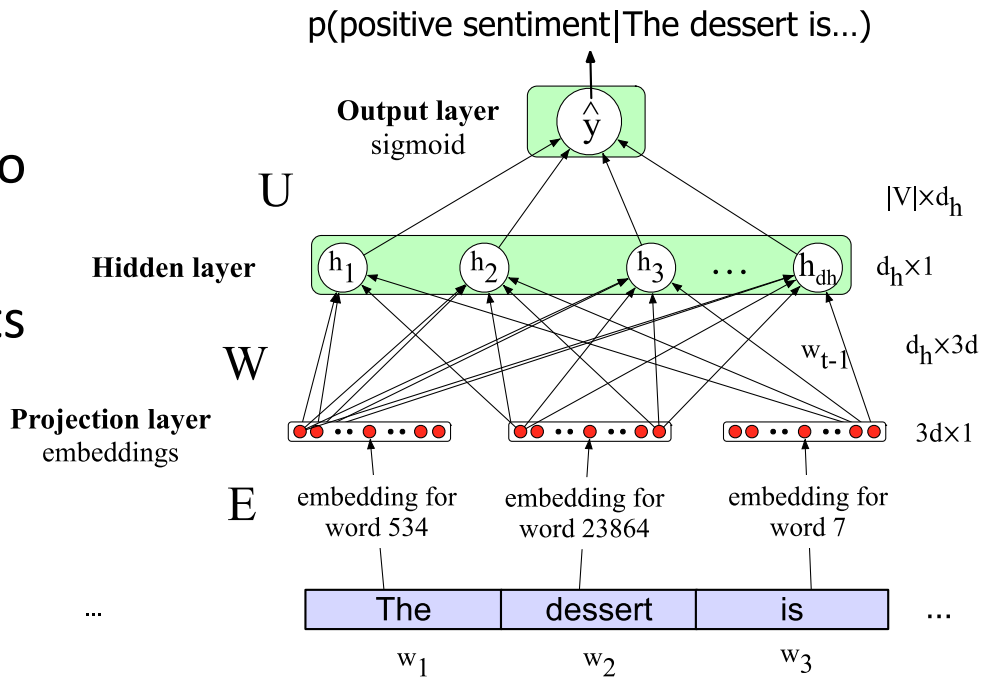
$$\frac{\partial a}{\partial z} = a(1-a) \qquad \frac{\partial z}{\partial w_i} = h_i$$

$$\frac{\partial L}{\partial w_i} = -\left(\frac{y}{a} + \frac{y-1}{1-a}\right)a(1-a)h_i = (\mathrm{a}-\mathrm{y})h_i$$

MLP for NLP Tasks

- Assume a fixed size length

1. Make the input the length of the longest input
   - If shorter then pad with zero embeddings
   - Truncate if you longer inputs are observed at test time
2. Create a single "sentence embedding"
   - Take the mean of all the word embeddings
   - Take the element-wise max of all the word embeddings

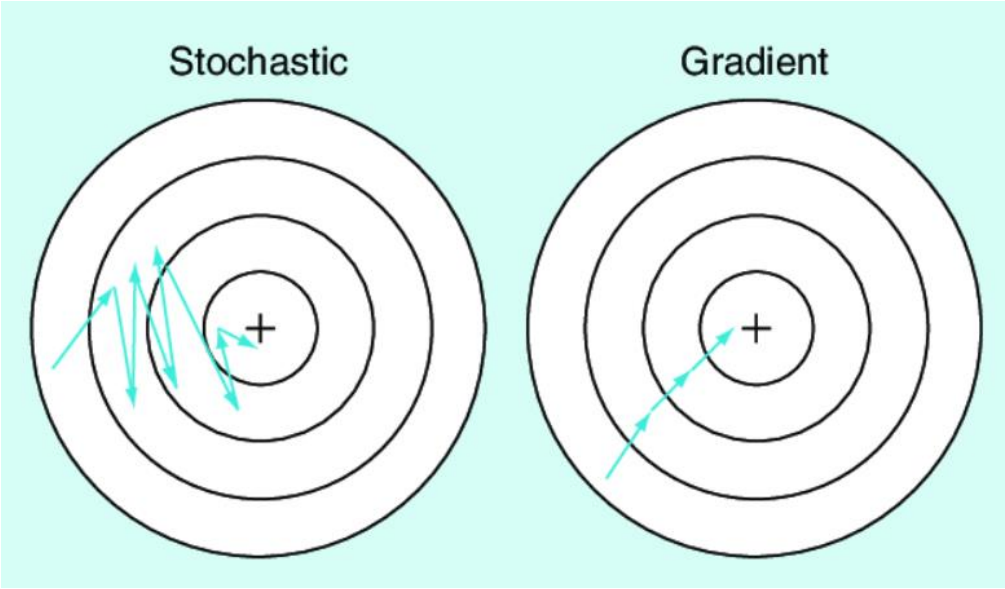


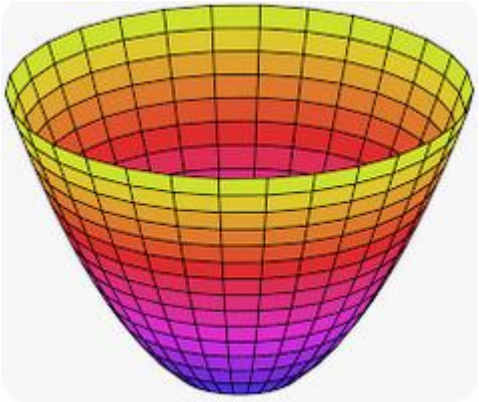
Figures adopted from slides by Jurafski and Martin

Training Feedforward Neural Networks

- Backpropagation is performed using stochastic gradient descent

- We create random batches from the training dataset
- We perform the normal gradient descent using a batch
- Epoch: optimize the cost function using all batches
- Optimize through epochs until convergence





Neural Networks vs SVM Rivalry

- Back Propagation: 1986
- SVM: 1992
- Deep Learning: 2012

- Three key reasons for reemergence of deep learning:

- Computational power: NVIDIA CUDA (2007)
- Annotated datasets: ImageNet (2009)
- ReLU !!!

Feedforward Neural Networks

- Limitations of feedforward neural networks
  - Input size should be fixed
  - All the input instances should have the same length
  - Solutions for this problem are not perfect!
- Language properties:
  - Contextual: “river bank” vs “bank branch”
  - Long-term Dependency: I was born in the US but moved to Italy when I was 2 and grew up there. I moved back to the US when I was 18 for college, so I can speak __ and English.
  - The order of words is important: “this is an informative book, but I do not like it” vs “this is not an informative book, but I like it”

Figures adopted from slides by Jurafski and Martin

Recurrent Neural Networks

- We can consider NL data as sequential data points, where the current word depends on the previous words in the sequence:

1 2 3 4 5 6 7 8 9 10 11

- Ex: Today, I want to play football and then watch a movie.
- Learning representations by back-propagating errors, 1986
- Core Idea: the function approximator can receive the input word by word such that its output depend on the history, i.e., relying on a notion of memory

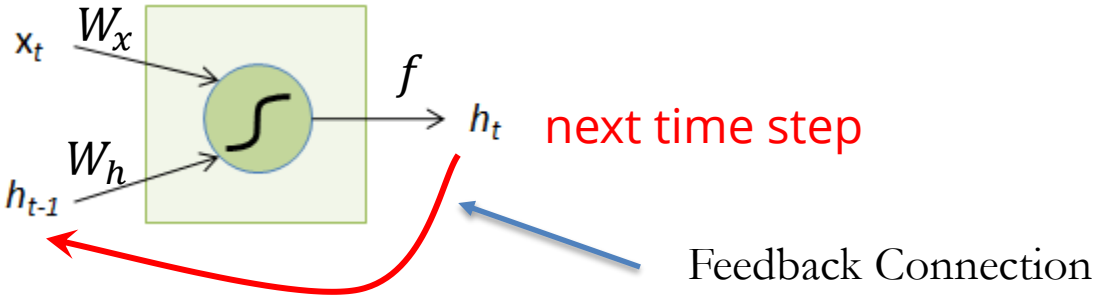
FNN $y = f(x)$

RNN $y_t = f(x_t, h_{t-1})$

Recurrent Neural Networks

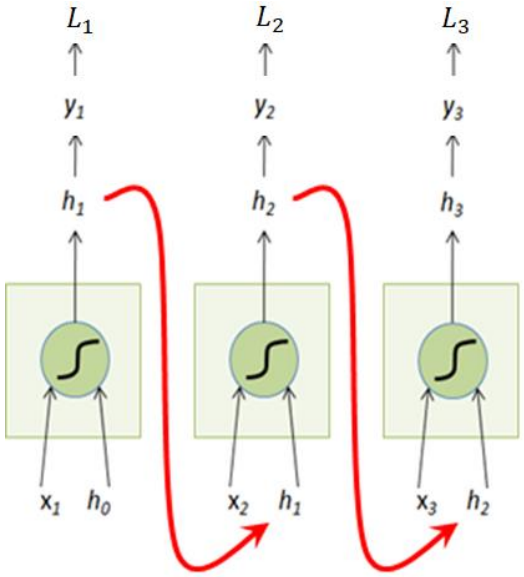
- Equipping perceptron with memory

- x_t: Input at time t
- h_{t-1}: State at time t-1



$$h_t = f(W_x x_t + W_h h_{t-1}), W_x \in \mathbb{R}^{M*N}, W_h \in \mathbb{R}^{H*H}$$

- Unfolding RNN
- We can make the unit multi-layer



Figures adopted from slides by Arun Mallya

Recurrent Neural Networks

- The weight matrices are shared across time

- multi-output

$$L = L_1 + L_2 + L_3$$

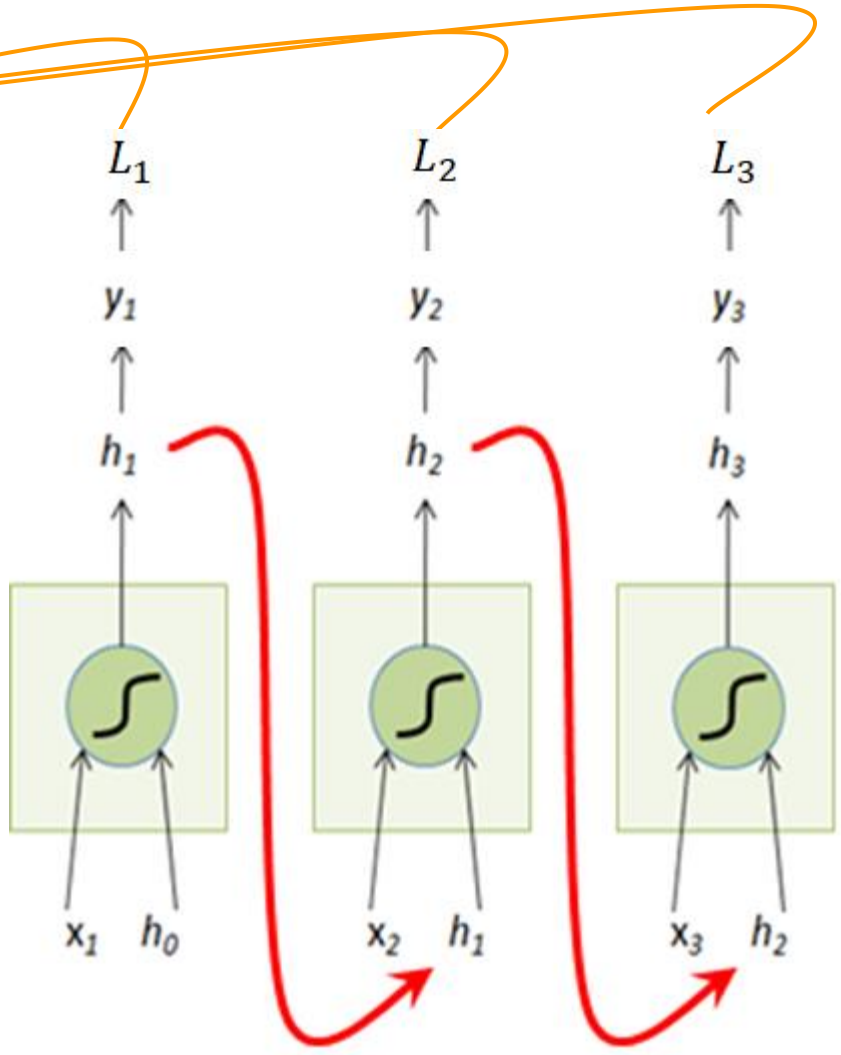
$$L = \sum_{t=1}^{T} L_t$$

$$y_t = f(x_t, h_{t-1})$$

$$L_t = l(y, \hat{y}_t)$$

- Single Output

$$L = L_3$$



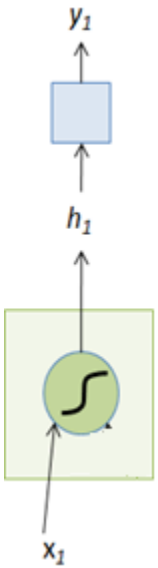
Figures adopted from slides by Arun Mallya

Training Feedforward Networks

- For every training data point (x, y)
  - Run *forward* computation to find model estimate $\hat{y}$
  - Run *backward* computation to update weights:
    - For every output node
    - Compute loss L between true y and the estimated $\hat{y}$
    - For every weight w from hidden layer to the output layer

      Update the weight using gradient descent $\frac{d}{dw} L(f(x; w), y)$
    - For all other nodes
    - Assess how much blame it deserves for the current answer

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial y} \frac{\partial y}{\partial h} \frac{\partial h}{\partial W} \quad , \quad L{=}L(\mathrm{y}(\mathrm{h}(\mathrm{W})))$$



Figures adopted from slides by Arun Mallya

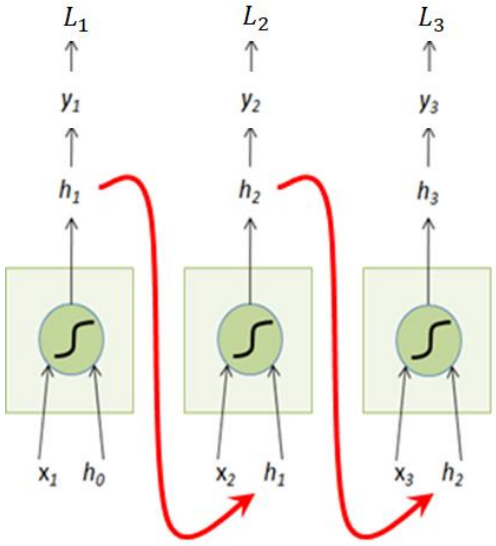
Training RNNs

- For every training data point (x, y)
  - Run *forward* computation to find model estimates $\hat{y}_t$
  - Run *backward* computation to update weights:
    - For every output node
  - Compute loss L between true y and the estimated $\hat{y}_t$
  - For every weight w from hidden layer to the output layer

Update the weight using gradient descent $\frac{d}{dw}\sum_{t=1}^{T} L_t$

$$\frac{\partial L_t}{\partial W} = \frac{\partial L}{\partial y_t}\frac{\partial y_t}{\partial h_t}\frac{\partial h_t}{\partial W} \quad ???$$

$$L_2 = L_2(y_2(h_2(W, h_1(W))))$$



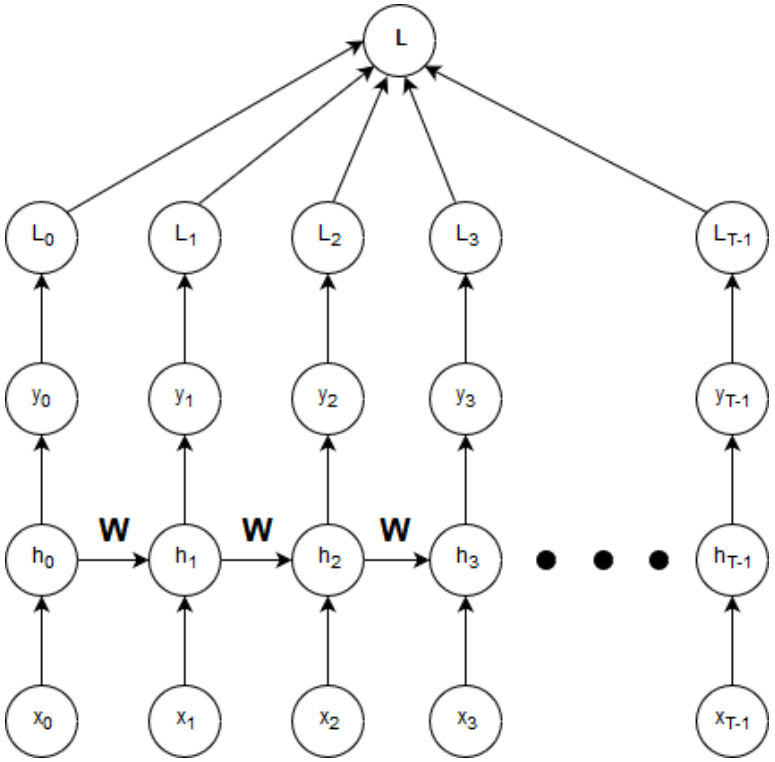
Figures adopted from slides by Arun Mallya

Backpropagation Through Time

- Gradient descent step to update the weights:

$$\mathbf{W} \rightarrow \mathbf{W} - \alpha \frac{\partial L}{\partial \mathbf{W}}$$

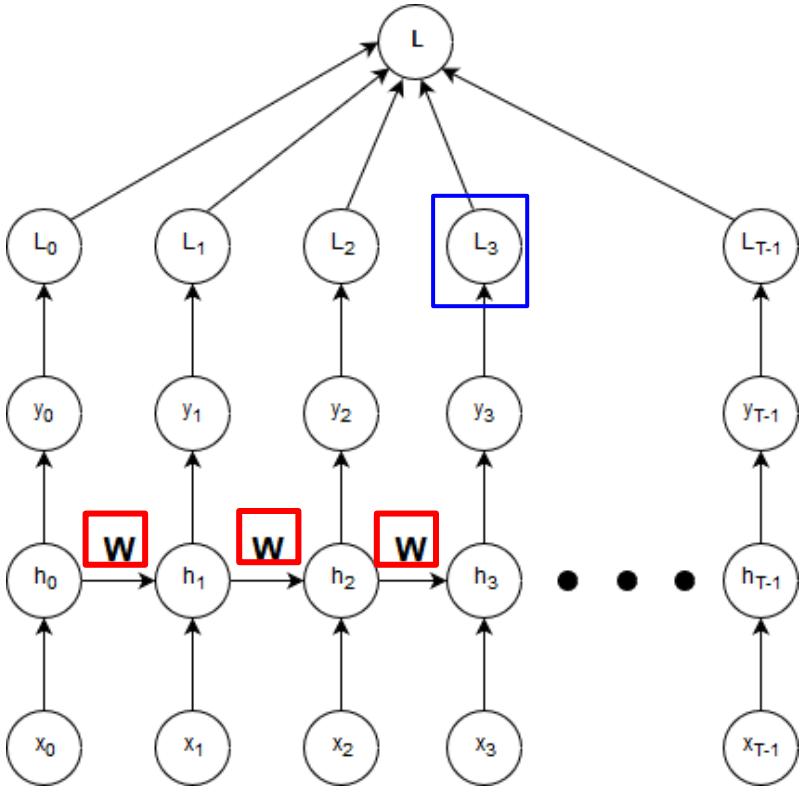
- Issue: **W** occurs each timestep
- **Every** path from **W** to L is one dependency for differentiation
- We need to find all paths from **W** to L
- There is one depenency through L_1
- There are two depenencies through L_2



Figures adopted from slides by Arun Mallya

Backpropagation Through Time

$$L = \sum_{t=1}^{T} L_t$$



2D Chain Rule

$$\frac{\partial L_t}{\partial W} = \sum_{k=1}^{t} \frac{\partial L_t}{\partial h_k} \frac{\partial h_k}{\partial W}$$

1D Chain Rule

$$\frac{\partial L_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \frac{\partial h_t}{\partial h_k}$$

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

1D Chain Rule

$$\frac{\partial L_t}{\partial W} = \sum_{k=1}^{t} \frac{\partial L_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \prod_{m=k+1}^{t} \frac{\partial h_m}{\partial h_{m-1}} \frac{\partial h_k}{\partial W}$$

Backpropagation Through Time

$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \sum_{k=1}^{t} \frac{\partial L_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \prod_{m=k+1}^{t} \frac{\partial h_m}{\partial h_{m-1}} \frac{\partial h_k}{\partial W}$$

- Computationally expensive
- Vanishing/Exploding gradients challenge

- Truncated Backpropagation

- Tutorial

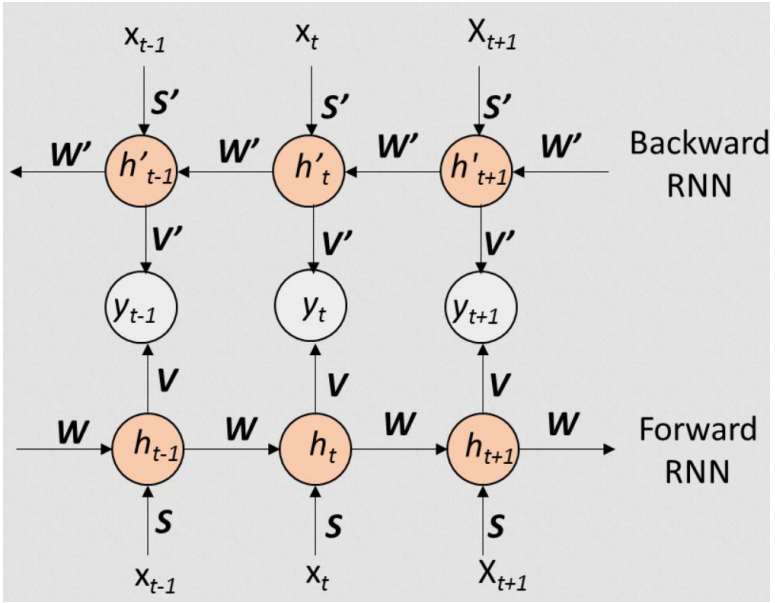
https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html

Bidirectional RNN

- RNN is developed for temporal data (unidirectional)
- Effects in NL are bidirectional:

- Ex: I lived in __ for ten years so I can speak French

- Solution: we can use two RNNs that process the input in opposite directions

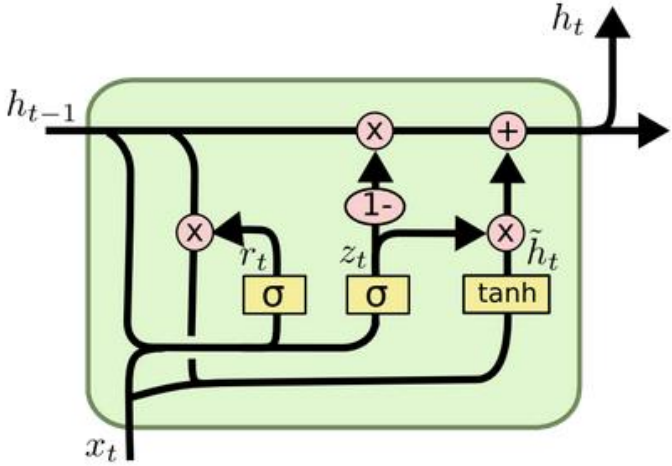


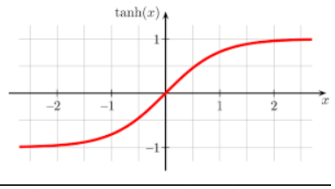
$$h_t = f(W_x x_t + W_h h_{t-1})$$
$$h'_{t'} = f(W'_x x_{t'} + W'_h h'_{t'-1})$$
$$t' = T - t$$
$$y_t = g(h_t, h'_{t'})$$

Gated RNN

- Gated recurrent unit: can learn long-range dependencies

- Control mechanism on information flow
- Gates control information flow
- Resent and Update gates are often close to either 0 or 1 due to using sigmoid
- New gate is used as a preliminary candidate to update the state variable





Reset Gate $r_t = \sigma(W_{ir}x_t + W_{hr}h_{(t-1)})$

Update Gate $z_t = \sigma(W_{iz}x_t + W_{hz}h_{(t-1)})$

New Gate $\tilde{h}_t = \tanh(W_{in}x_t + r_t \odot (W_{hn}h_{(t-1)}))$

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

Hadamard Product

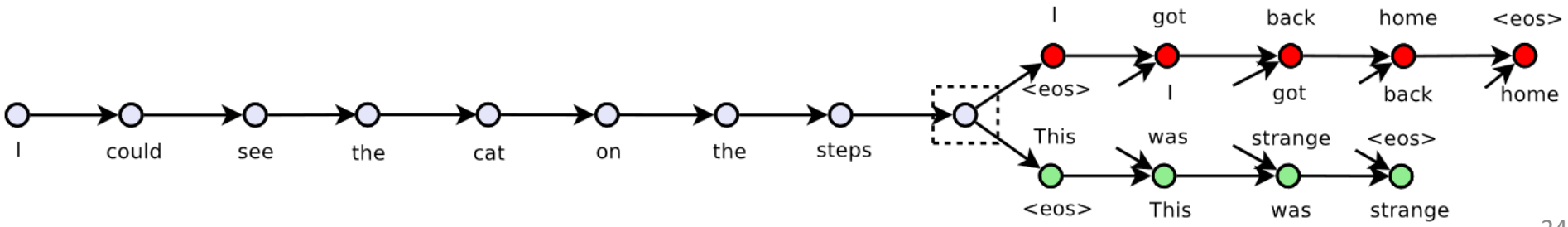
Language Processing Hierarchy Levels

- Hierarchy levels:
- Documents
- **Sentences**
- Phrase
- Words
- The hierarchy is determined based on application
- Ex: Document-level representation is more natural for document classification, but word representation makes more sense to do semantic inference tasks
- The success of Word2Vec led exploring possibility of extending a similar approach to structures in other hierarchies that have similar relationships

- Information Retrieval

Skip-Thought Vectors

- An encoder-decoder model:
- Encoder: maps a sentence into a vector
- Decoder: conditions on this vector to generate surrounding sentences
- Architecture: RNN encoder with GRU activations, RNN decoder with conditioned GRU
- Benefit: Skip-thoughts sentence representations lead to robust performance across sentence-level tasks
- retrieval, Q&A answering



Skip-Thought Vectors

- Encoder Structure:

- Let w_i^j be the word j in sentence I, where N is the number of words in that sentence
- At each time step, the encoder produces a hidden state h_i^t as the representation of the sequence $w_i^1, \ldots, w_i^t$
- The hidden state h_i^N represents the full sentence

$$
\begin{aligned}
\mathbf{r}^t &= \sigma(\mathbf{W}_r \mathbf{x}^t + \mathbf{U}_r \mathbf{h}^{t-1}) \\
\mathbf{z}^t &= \sigma(\mathbf{W}_z \mathbf{x}^t + \mathbf{U}_z \mathbf{h}^{t-1}) \\
\bar{\mathbf{h}}^t &= \tanh(\mathbf{W} \mathbf{x}^t + \mathbf{U}(\mathbf{r}^t \odot \mathbf{h}^{t-1})) \\
\mathbf{h}^t &= (1 - \mathbf{z}^t) \odot \mathbf{h}^{t-1} + \mathbf{z}^t \odot \bar{\mathbf{h}}^t
\end{aligned}
$$

Skip-Thought Vectors

- Decoder Structure:

- The matrices C_z, C_r, and C that are used to bias the update gate, reset gate and hidden state computation by the sentence vector
- Separate decoders are used for previous and next sentences
- Parameters for each decoder are separated

$$
\begin{aligned}
\mathbf{r}^t &= \sigma(\mathbf{W}_r^d \mathbf{x}^{t-1} + \mathbf{U}_r^d \mathbf{h}^{t-1} + \mathbf{C}_r \mathbf{h}_i) \\
\mathbf{z}^t &= \sigma(\mathbf{W}_z^d \mathbf{x}^{t-1} + \mathbf{U}_z^d \mathbf{h}^{t-1} + \mathbf{C}_z \mathbf{h}_i) \\
\bar{\mathbf{h}}^t &= \tanh(\mathbf{W}^d \mathbf{x}^{t-1} + \mathbf{U}^d (\mathbf{r}^t \odot \mathbf{h}^{t-1}) + \mathbf{C}\mathbf{h}_i) \\
\mathbf{h}_{i+1}^t &= (1 - \mathbf{z}^t) \odot \mathbf{h}^{t-1} + \mathbf{z}^t \odot \bar{\mathbf{h}}^t
\end{aligned}
$$

Given $\mathbf{h}_{i+1}^t$, the probability of word w_{i+1}^t given the previous $t-1$ words and the encoder vector is

$$P(w_{i+1}^t | w_{i+1}^{<t}, \mathbf{h}_i) \propto \exp(\mathbf{v}_{w_{i+1}^t} \mathbf{h}_{i+1}^t)$$

Skip-Thought Vectors

- Objective function: Given the sentence tuple (s_{i-1}, s_i, s_{i+1}) the objective function is the sum of the log-probabilities for the forward and backward sentences conditioned on the encoder representation:

$$\sum_t \log P(w_{i+1}^t | w_{i+1}^{<t}, \mathbf{h}_i) + \sum_t \log P(w_{i-1}^t | w_{i-1}^{<t}, \mathbf{h}_i)$$