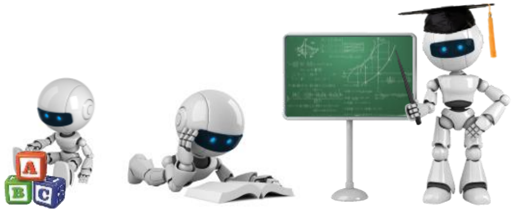
USC Viterbi
School of Engineering
Department of Computer Science

CSCI 544
Applied Natural Language Processing

Mohammad Rostami
USC Computer Science Department



Logistical Notes

- Project Teams: be careful when editing

https://docs.google.com/spreadsheets/d/1_AhFxYLroii4eLyEnzswOnfHLNdZ1H0uzT9HDJQnY_U/edit#gid=0

- Slack channel: please add hashtags to your comments:

- #HW+Number: e.g., HW1
- #Team: looking for team members
- #Q: question
- #Other

- Quiz 2: Thursday and for 10 minutes between 2:05 and 2:20

Word2Vec

- Core idea: find embeddings using a prediction task involving **neighboring words** in a huge real-world corpus.

- Input data can be sets of **successive word-patterns** from meaningful sentences in the corpus, e.g., “one of the most important”.
- Try to build **synthetic** prediction tasks using these patterns, e.g., “(one of the __ important, most)”
- Train a model to solve the prediction task
- Embeddings are found as a **byproduct** of this process

- More specifically:

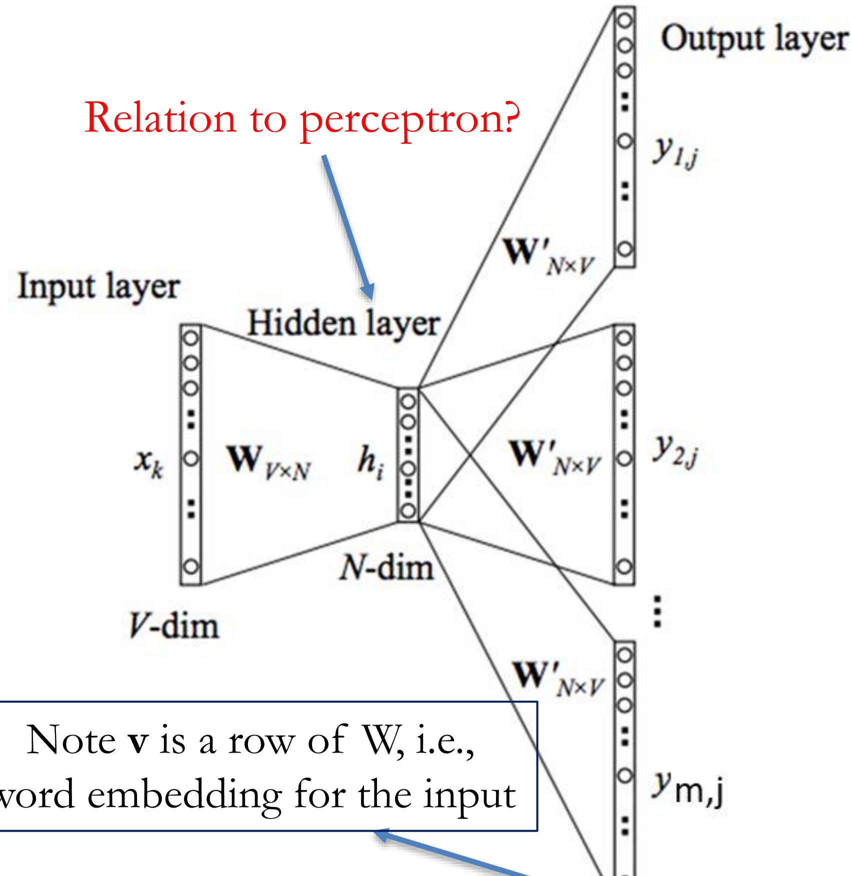
- **We consider a window with the center word w_t and “context words” $w_{t'}$ with a window fixed size, e.g., (t’=t-5, … t-1, t+1, … , t+5).**
- The model is assumed to be a two-layer neural network
- We train the network to predict all w_t given $w_{t'}$ such that $p(w_t|w_{t'})$ is maximized
- We learn embeddings such that the prediction loss is minimized, i.e., if two words occur in close proximity, their representations become similar.

Skip-Gram

- Given a center word, we predict the context words:

- Vocabulary size: V
- Input layer: center word in 1-hot form.
- The row k in W_{VxN} is the vector embedding of k-th center word.
- The column k of W'_{NXV} is context vector of the k-th word.
- At output layer y_{ij}, i=1..M is computed:
  1. We use the context word 1-hot vector to choose its column in W'_{NxV}
  2. dot product with h_i for the center word
  3. compute the softmax
  4. **Match** the output one-hot vector
- After optimization, we will have two vectors for each word. We can set the eventual embedding to e the average of these two vectors

Relation to perceptron?



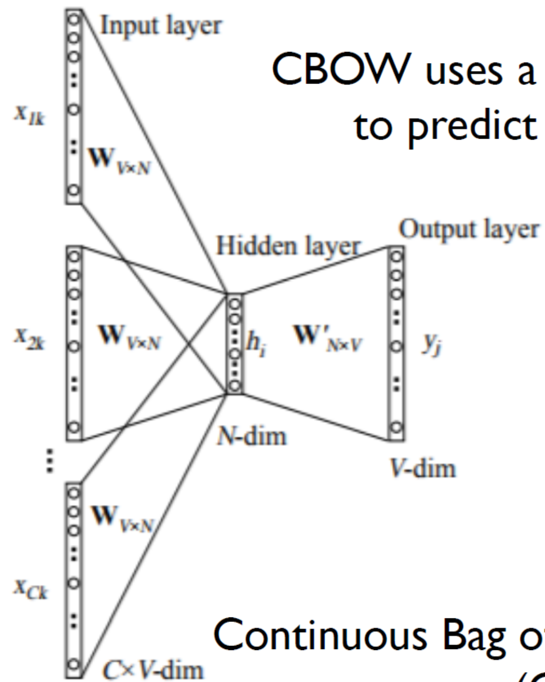
Note **v** is a row of W, i.e., word embedding for the input

$$h_{N\times 1} = W^T_{V\times N} x_{V\times 1} = v$$
$$\hat{z}_j = h^T_{N\times 1} W'[:, j]_{N\times 1} = v^T u$$
$$\hat{y} = \sigma(W'^T_{N\times V} h_{N\times 1})$$

Note **u** is a row of W', i.e., word embedding for the output

Continuous Bag of Words

- Given a context word, we predict the center word:
- Vocabulary size: V
- Input layer: context words in 1-hot form.
- The row k in W_{VxN} is the vector embedding of k-th context word.
- The column k of W'_{NXV} is vector of the k-th center word.
- The output column y_{ij}, i=1..M is computed:
  1. We use the center word 1-hot vector to choose its column in W'_{NxV}
  2. dot product with h_i for the contex word
  3. compute the softmax
- We can set the eventual embedding to e the average of these two vectors
- Skip-gram incorporates non-frequent words better than CBOW



CBOW uses a window of word to predict the middle word

Continuous Bag of words (CBOW)

- Ex: Today is really __ day
Skip-gram: delightful+context and nice+context

Word2Vec Optimization Problem

- Consider an arbitrary order on vocabulary and let v_t be the word vector for center word t and u_t be the word vector for the context word t:

- Is the word vector unique?

Ex: [...like to **eat** lunch and...]

- We solve for the word vector by maximizing the following likelihood function

$$v_t, u_t = arg\max \log(\Pi_{t=1}^{T} P(c|w_t)) =$$
$$arg\max \log(\Pi_{t=1}^{T} P(c|w_t)) =$$
$$arg\max \log(\Pi_{t=1}^{T} \Pi_{j=-M,\, j\neq 0}^{M} P(w_{t-j}|w_t)) =$$

Important Step

$$arg\max \frac{1}{T} \sum_{t=1}^{T} \sum_{j=-M,\, j\neq 0}^{M} \log(P(w_{t-j}|w_t)) =$$

Word2Vec Optimization Problem

- Conditional Probability modeling

Note **v** denotes word embeddings for center words

$$P(w_{t-j}|w_t) = \frac{e^{u_{t-j}^T v_t}}{\sum_{t=1}^{T} e^{u_{t-j}^T v_t}}$$

Is it an extension of the logistic function?

Computationally expensive!

Note **u** denotes word embeddings for the context words

- The log-likelihood optimization problem

$$u_o, v_c = \arg\min \frac{1}{V} \sum_{w=1}^{V} -u_o^T v_c + \log\left(\sum_{w'=1}^{V} e^{u_o^T v_c}\right)$$

- Can be solved similar to logistic regression objective using numerical optimization techniques, e.g., gradient descent

Word2Vec Optimization Problem

- gradient descent step

$$u_o, v_c = \arg\min \frac{1}{V} \sum_{w=1}^{V} -u_o^T v_c + \log(\sum_{w'=1}^{V} e^{u_o^T v_c})$$

$$v_c^{i+1} = v_c^i - \eta \nabla f(v_c^i)$$

$$\nabla f(v_c^i) = -u_o + \sum_{w=1}^{v} \frac{e^{u_o^T v_c}}{\sum_{w'=1}^{V} e^{u_o^T v_c}} u_o$$

$$\nabla f(v_c^i) = -u_o + \sum_{w=1}^{v} p(v_o|u_c) u_o = -u_o(1 - E(u_o))$$

- Tutorial:

https://www.kaggle.com/pierremegret/gensim-word2vec-tutorial

Negative Sampling

- Word2Vec optimization is a highly computationally intensive problem: the **shallow** network has a large number of weights and we will have billions of pairs
- Because we use one-hot vectors, each training pair (c,o) contributes minimally to updating the weights
- Negative sampling: for each positive pair, we randomly generate negative pairs, for which the network output should be 0.

$$u_o, v_c = \arg\min \sigma(u_o^T v_c) + \sum_{k=1}^{K} E_{v_{w_i} \sim P(w_c)} \log(-\sigma(v_{w_i}^T v_c))$$

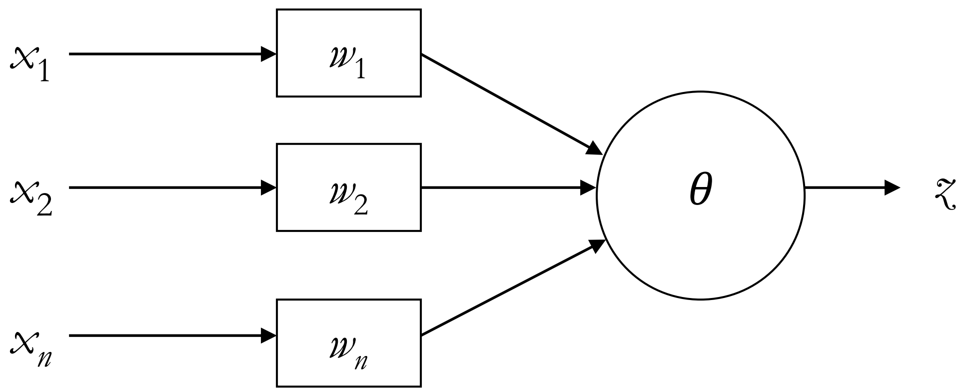
- This is an instance of naïve data augmentation

Neural vs Factorization-Based Embeddings

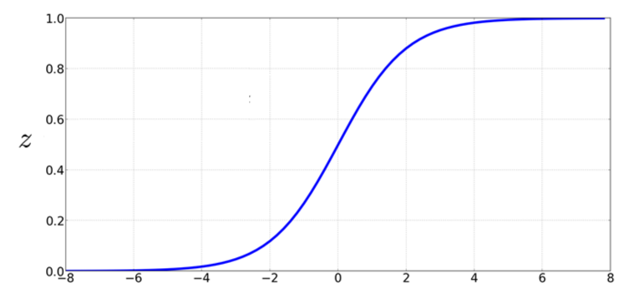
- Comparison is challenging (Levy and Goldberg, NeurIPS 2014):
  - **Hyperparamters**
  - Factorization algorithm
  - Amount of data
- A particular word embedding approach is unlikely to be state-of-the-art for all applications
- Hyperparameters appear to have the largest impact in performance.
- Neural models are less sensitive with respect to hyperparameters, and training data preparation is more straightforward and systematic

Neural Networks

- Perceptron: the neural network learning model in the 1960's
- Simple and limited (single layer models)
- Basic concepts are similar for multi-layer models



$$z = \begin{cases} 1 & \text{if } \sum_{i=1}^{n} x_i\, w_i \geq \theta \\ 0 & \text{if } \sum_{i=1}^{n} x_i\, w_i < \theta \end{cases}$$



$$z = 1/(1 + \exp(-x))$$

XOR Problem

- Can perceptron be used to learn XOR?

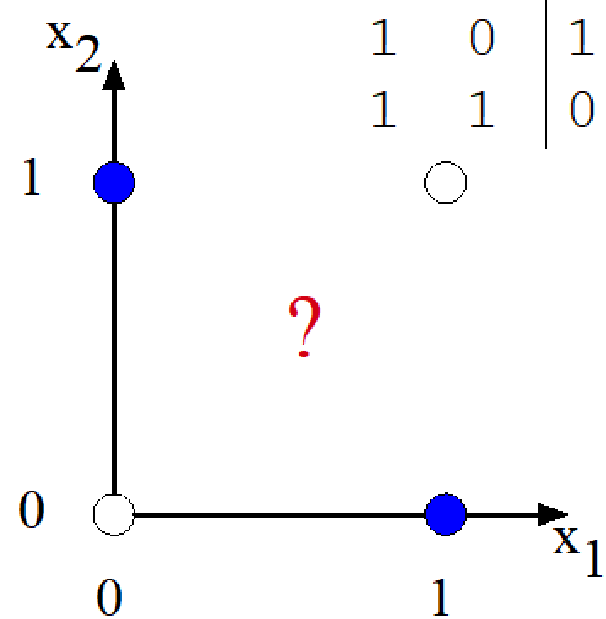
$$ax_1 + bx_2 + c = 0$$

$$\begin{aligned} a\,0 + b\,0 + c &< 0 \\ a\,0 + b\,1 + c &> 0 \\ a\,1 + b\,0 + c &> 0 \\ a\,1 + b\,1 + c &< 0 \end{aligned} \quad \longrightarrow \quad \begin{aligned} c &< 0 \\ b + c &> 0 \\ a + c &> 0 \\ a + b + c &< 0 \end{aligned} \quad \longrightarrow$$

$$\begin{aligned} c &< 0 \\ a + b + 2c &> 0 \\ -a - b - c &> 0 \end{aligned} \quad \longrightarrow \quad \begin{aligned} c &< 0 \\ c &> 0 \end{aligned} \quad !$$

XOR

x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0

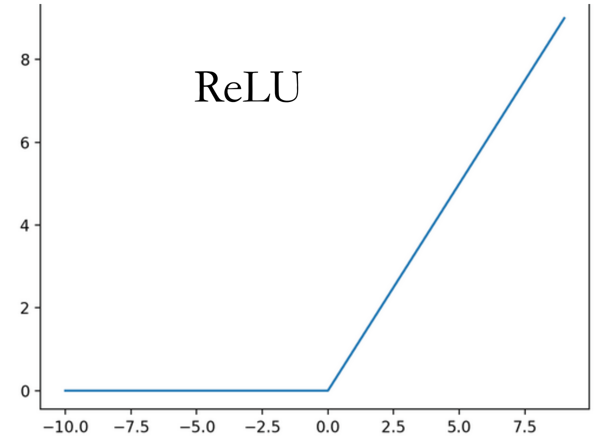


c) x_1 XOR x_2

XOR Problem

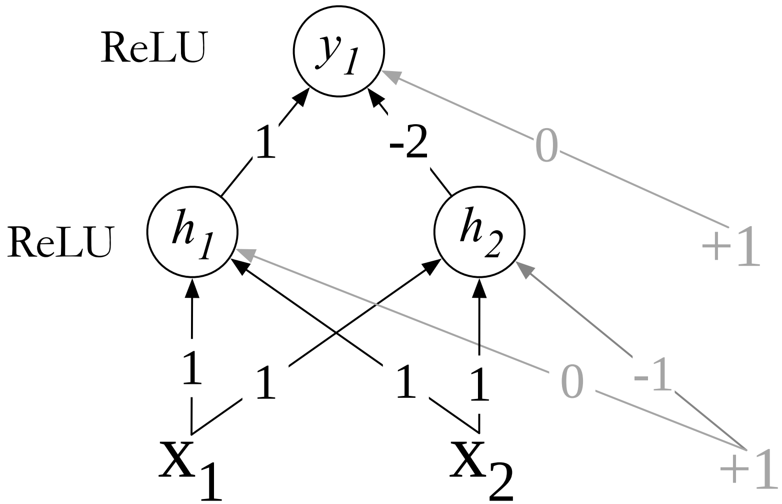
- Idea: XOR function can be computed using two layers perceptron units.

$$y_1 = Relu(Relu(x_2+x_1) - 2Relu(x_2 + x_1 - 1) + 1)$$



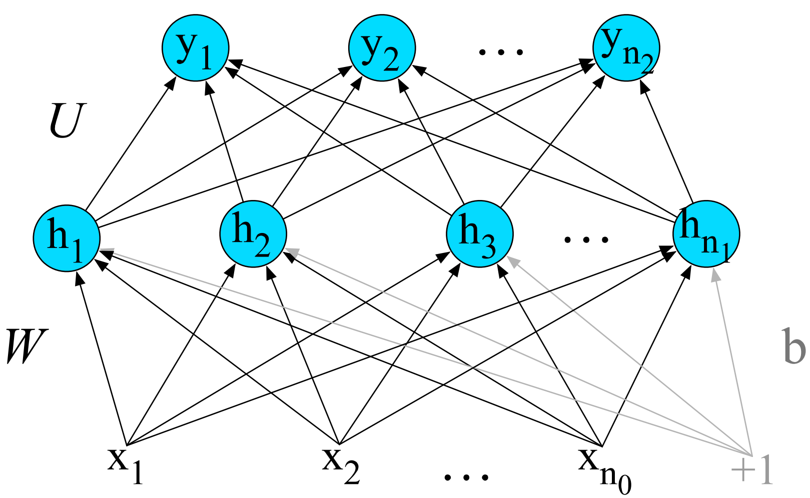
XOR

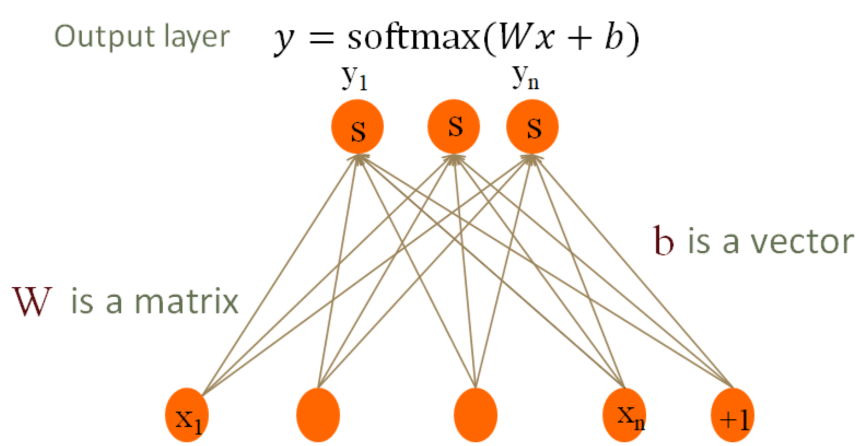
x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0



Feedforward Neural Networks

- An extension to perceptron by forming series connection between layers of parallel perceptrons: input, output, and hidden layers
- The number of layers, the nodes at each layer, and non-linear functions are design parameters

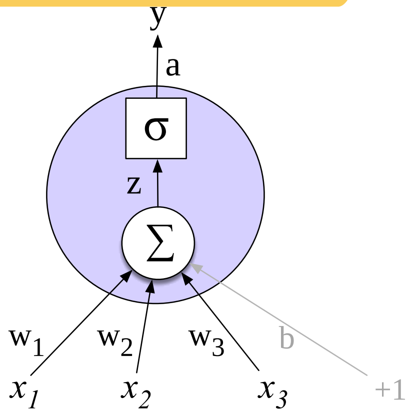




Figures adopted from slides by Jurafski and Martin

Multilayer Perceptron

- In the default setting, a node in hidden layers receives inputs from all nodes in the previous layer and its output is fed to all nodes in the next layer



$$
\begin{aligned}
z^{[1]} &= W^{[1]} a^{[0]} + b^{[1]} \\
a^{[1]} &= g^{[1]}(z^{[1]}) \\
z^{[2]} &= W^{[2]} a^{[1]} + b^{[2]} \\
a^{[2]} &= g^{[2]}(z^{[2]}) \\
\hat{y} &= a^{[2]}
\end{aligned}
$$

for i **in** 1..n

$$
\begin{aligned}
z^{[i]} &= W^{[i]}\, a^{[i-1]} + b^{[i]} \\
a^{[i]} &= g^{[i]}(z^{[i]})
\end{aligned}
$$

$$\hat{y} = a^{[n]}$$

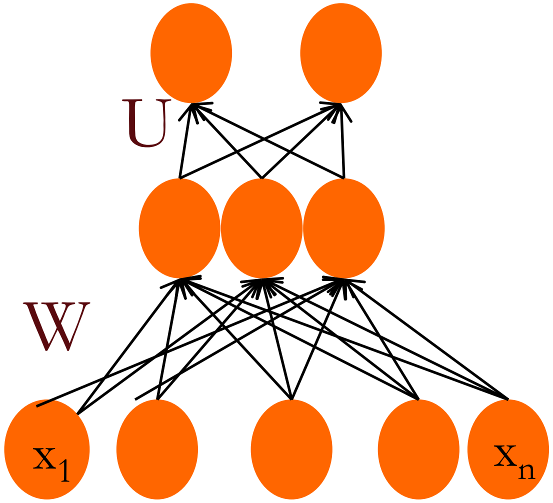
Figures adopted from slides by Jurafski and Martin

Multiclass Outputs

- What if we have more than two output classes?
  - We add one output for each class
  - We use a "softmax layer" at the output to generate a probability distribution
  - We use a proper loss function at the output

$$\text{Loss} = -\sum_{i=1}^{\substack{\text{output}\\ \text{size}}} y_i \cdot \log \hat{y}_i$$

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} \quad 1 \leq i \leq D$$



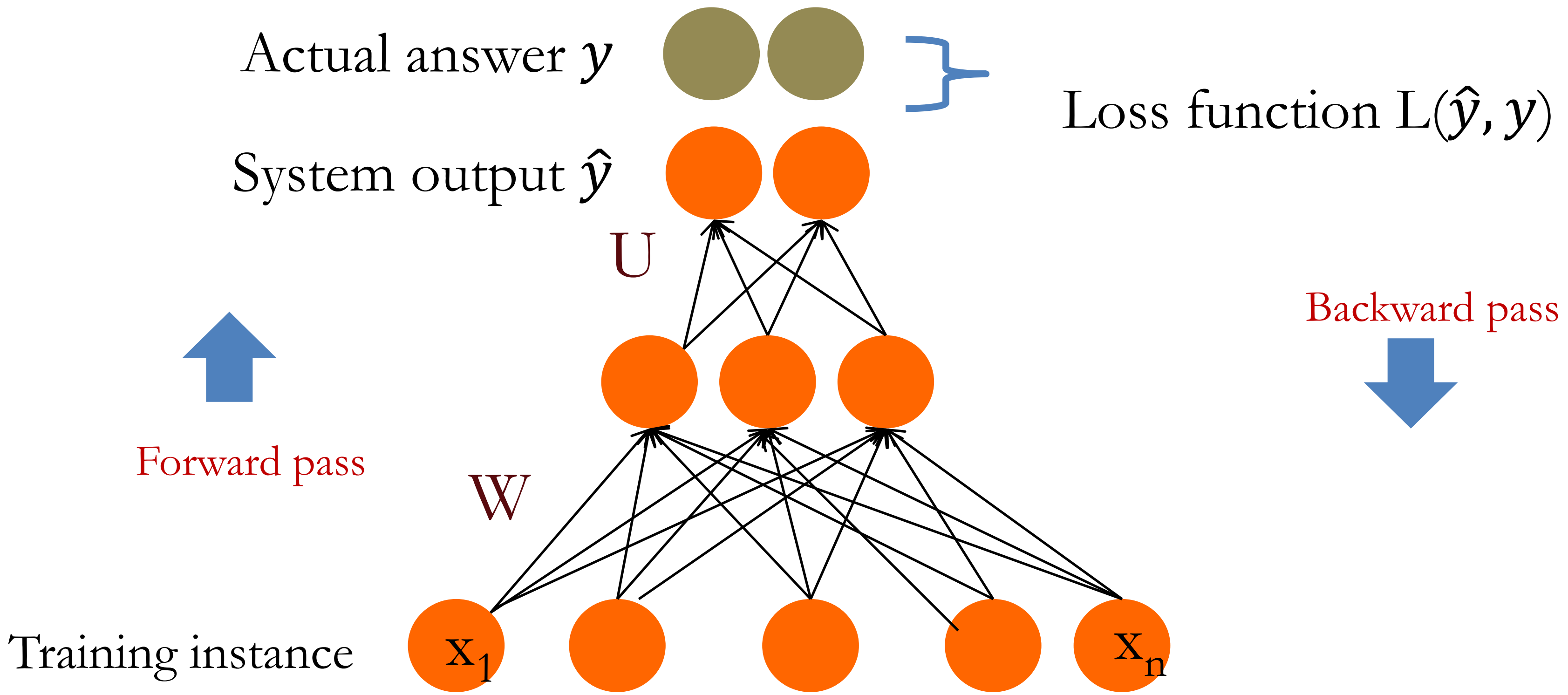
Figures adopted from slides by Jurafski and Martin

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



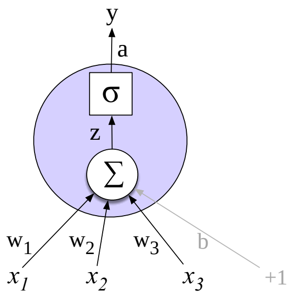
Figures adopted from slides by Jurafski and Martin

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



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

Adopted from slides by Jurafski and Martin

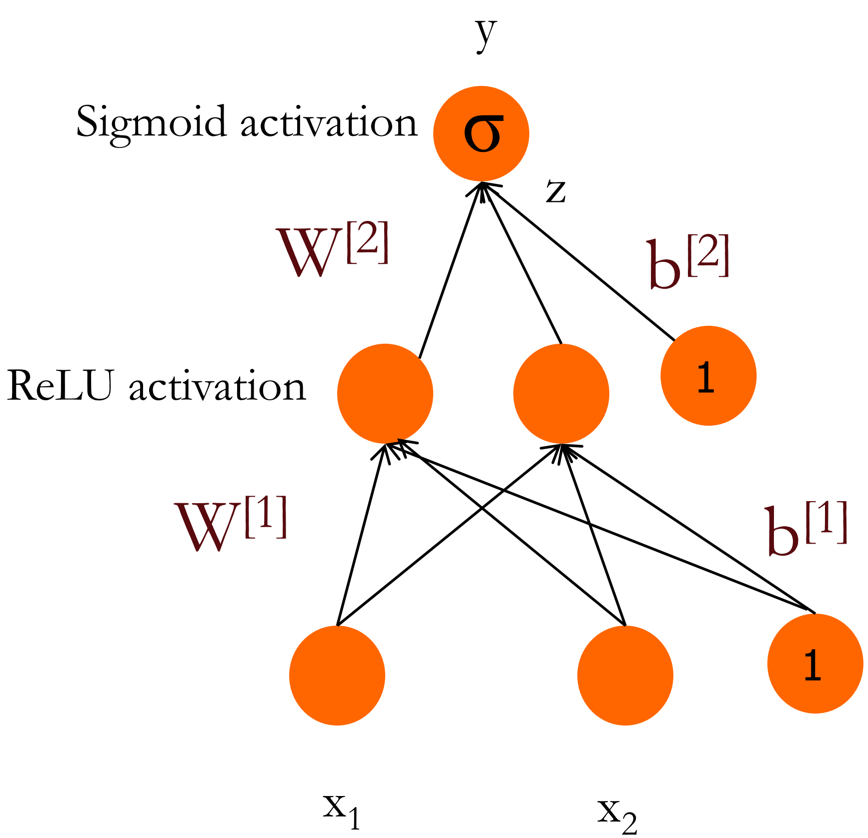
Backpropagation for Two Layer Network Output

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

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

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

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



Backpropagation for Two Layer Network Output

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

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

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

$$y = \sigma(z)$$

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

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

Notational Simplicity

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

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

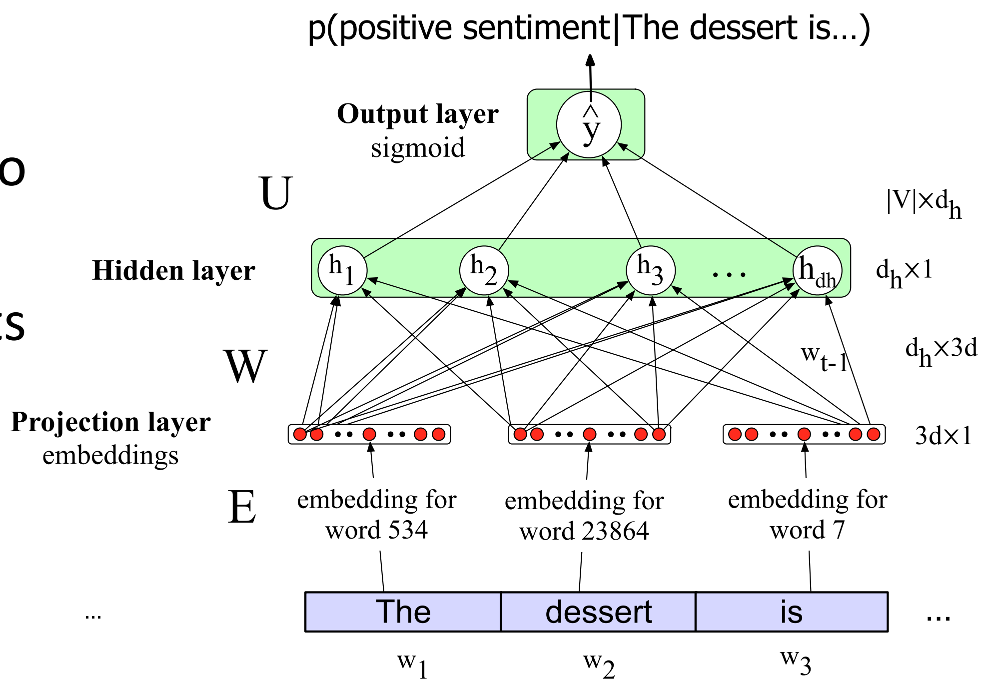
$$\frac{\partial z}{\partial w_i} = h_i$$

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

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

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 !!!