UF Herbert Wertheim College of Engineering
UNIVERSITY of FLORIDA

CIS 6930 Special Topics in Large Language Models

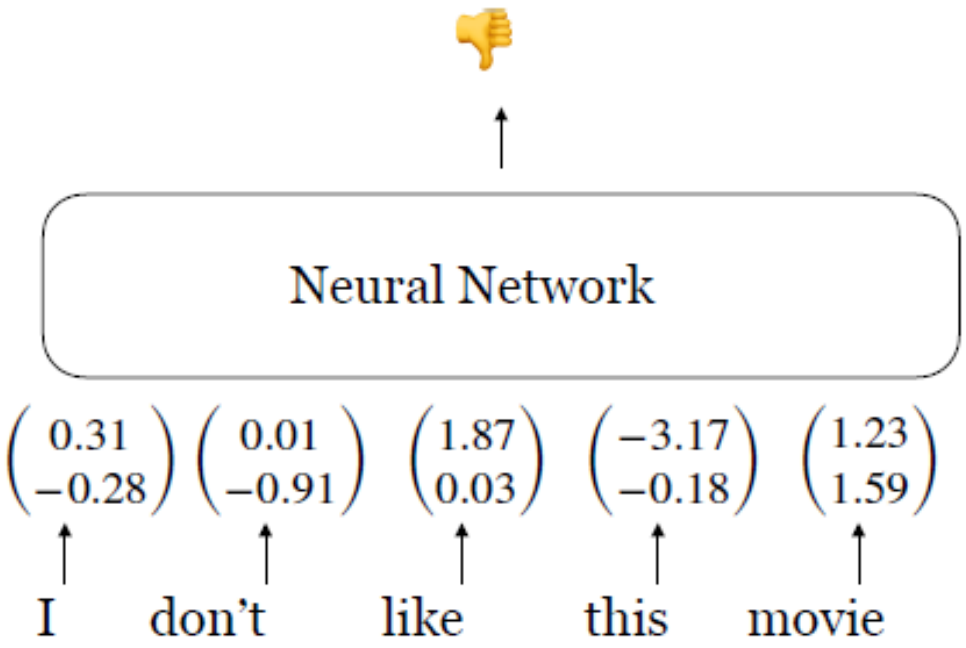
Feedforward Neural Language Models

Neural Networks in NLP

From Word Embedding to Neural Networks

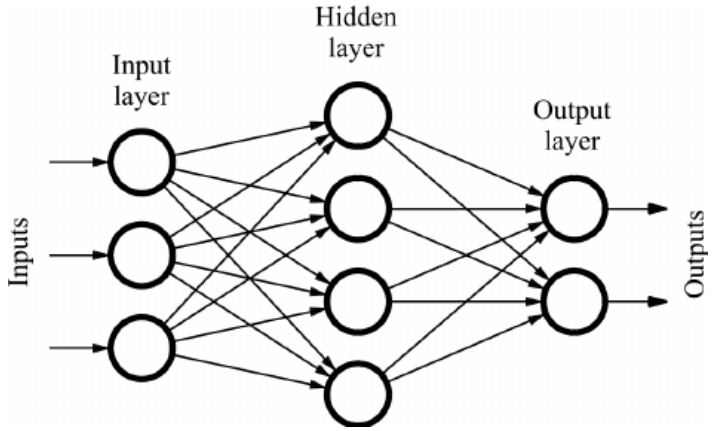
$$v_{\text{cat}} = \begin{pmatrix} -0.224 \\ 0.130 \\ -0.290 \\ 0.276 \end{pmatrix} \quad v_{\text{dog}} = \begin{pmatrix} -0.124 \\ 0.430 \\ -0.200 \\ 0.329 \end{pmatrix}$$

$$v_{\text{the}} = \begin{pmatrix} 0.234 \\ 0.266 \\ 0.239 \\ -0.199 \end{pmatrix} \quad v_{\text{language}} = \begin{pmatrix} 0.290 \\ -0.441 \\ 0.762 \\ 0.982 \end{pmatrix}$$

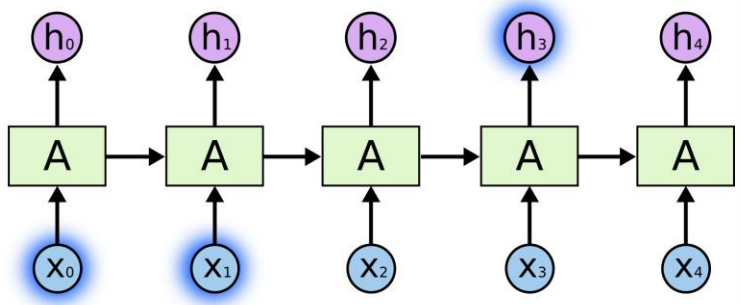


Neural Networks in NLP

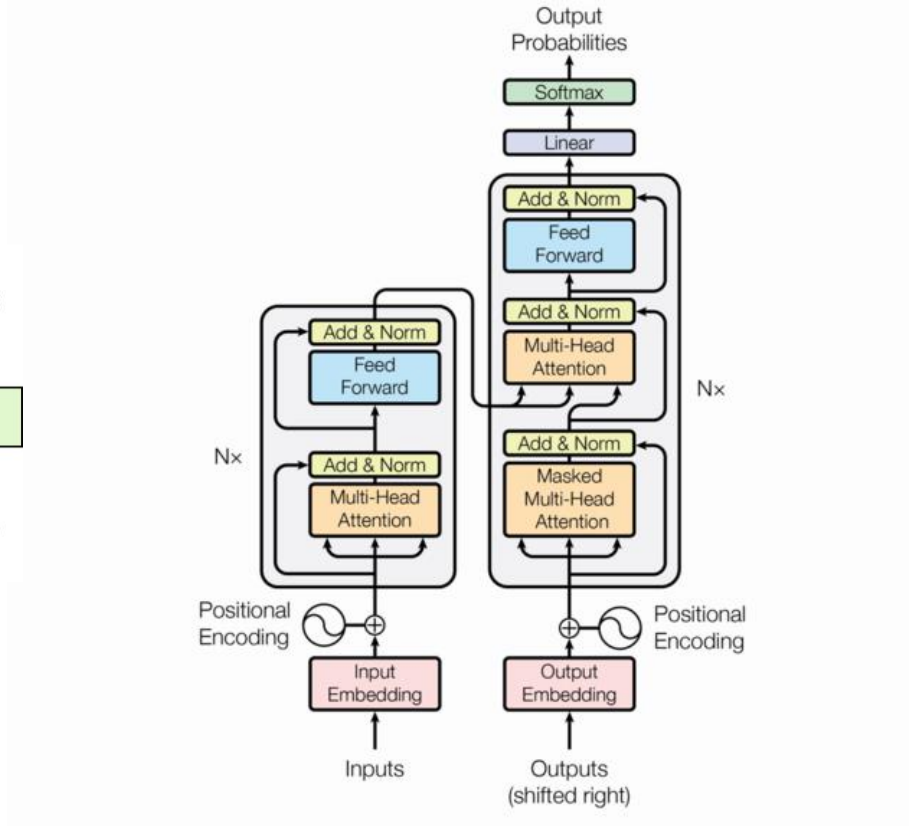
Several Types of Neural Networks in NLP



Feed-forward NNs



Recurrent NNs



Transformers

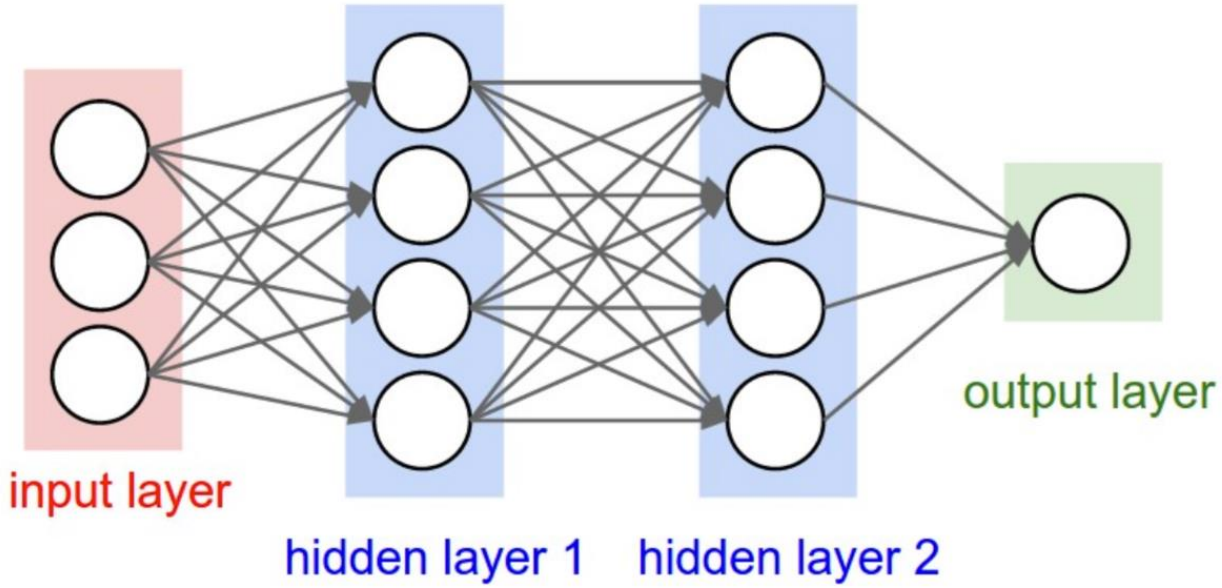
Outline

- Feedforward Neural Networks
- Model 1: Neural “bag-of-words” Models
- Model 2: Feedforward Neural Language Models

Feedforward Neural Networks

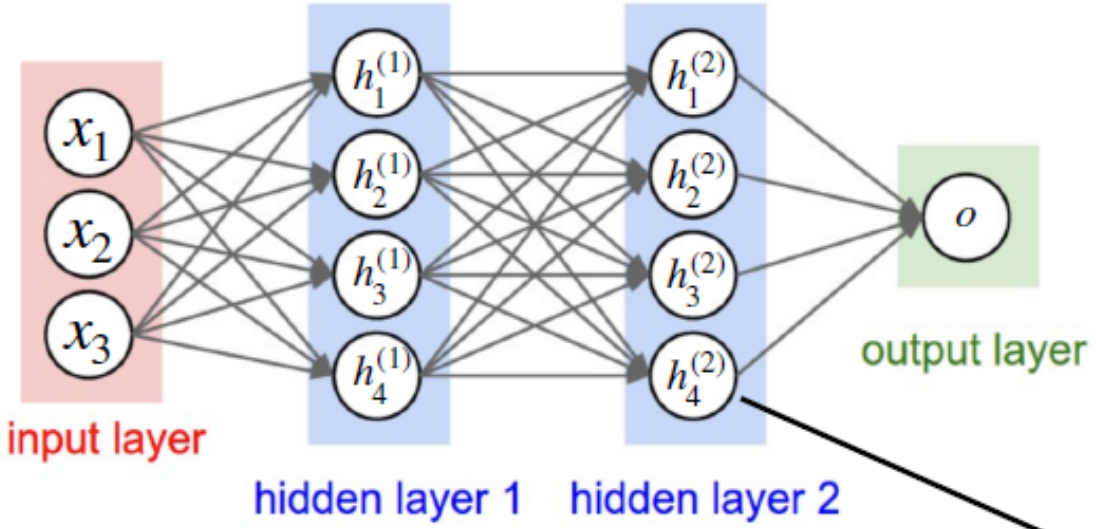
Feed-forward Neural Network

- The units are connected with no cycles
- The outputs from units in each layer are passed to units in the next layer
- No outputs are passed back to lower layers



Fully-connected (FC) layers
All the units from one layer are fully connected to every unites of the next layer

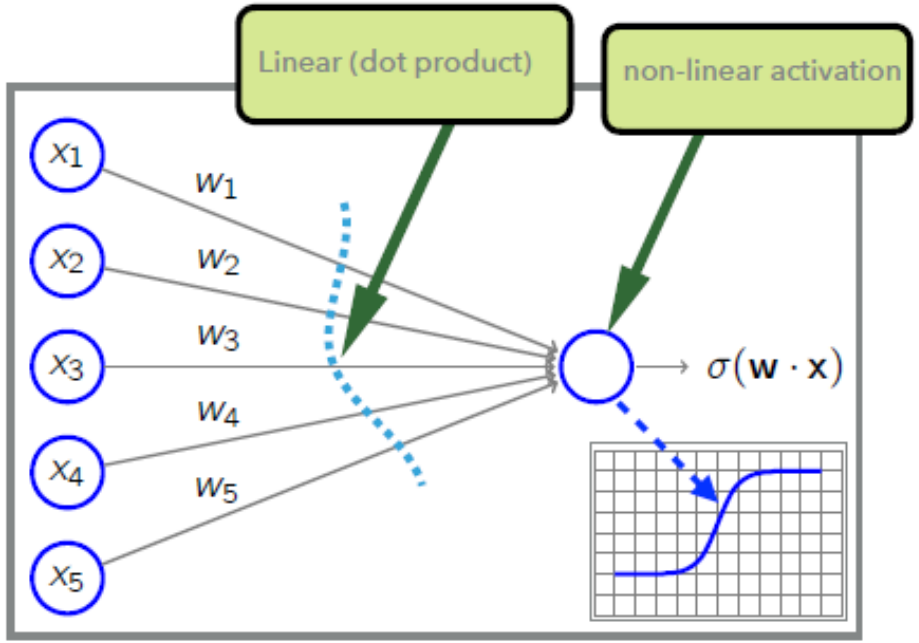
Feed-forward Neural Network



$$h_1^{(1)} = f(w_{1,1}^{(1)}x_1 + w_{1,2}^{(1)}x_2 + w_{1,3}^{(1)}x_3)$$

$$h_3^{(2)} = f(w_{3,1}^{(2)}h_1^{(1)} + w_{3,2}^{(2)}h_2^{(1)} + w_{3,3}^{(2)}h_3^{(1)} + w_{3,4}^{(2)}h_4^{(1)})$$

non-linearity f: σ, tanh *or ReLU.*

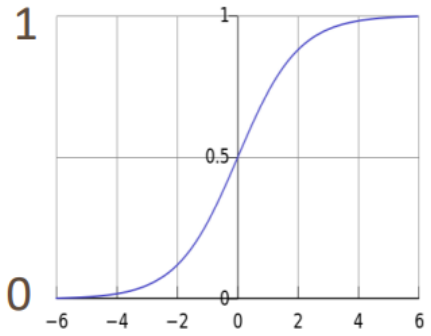


Activation Functions

- For deep neural networks, the first thing to try is ReLU: it trains quickly and performs well due to good gradient backflow
- Sigmoid and tanh functions are still used (e.g. sigmoid to get probability)

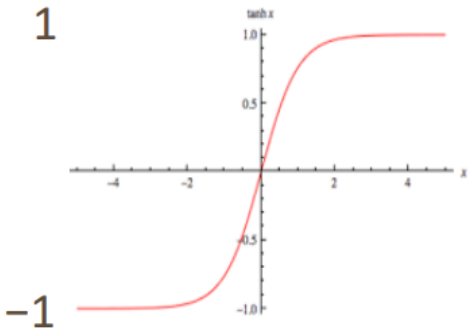
logistic ("sigmoid")

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



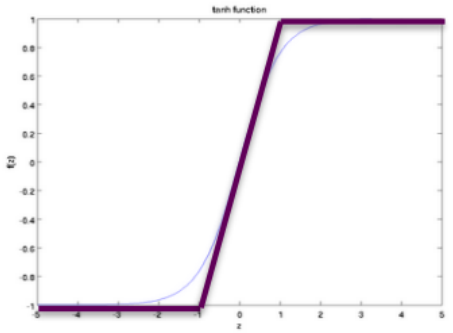
tanh

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



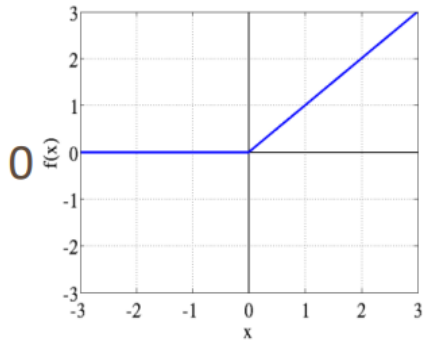
hard tanh

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

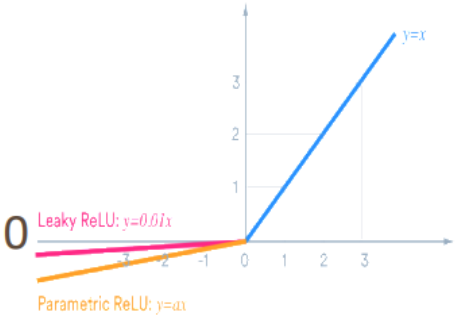


(Rectified Linear Unit)
ReLU

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

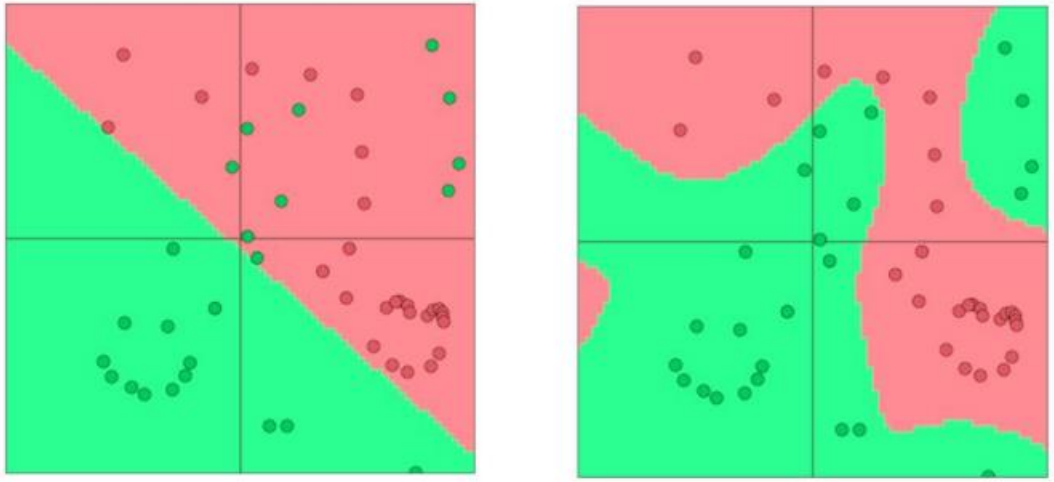


Leaky ReLU /
Parametric ReLU

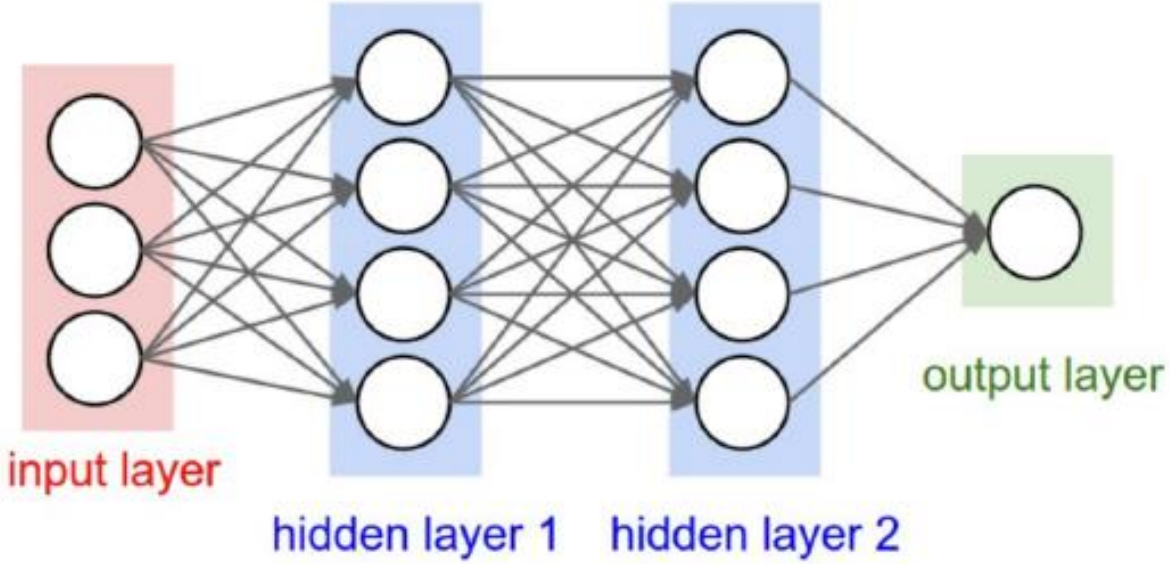


Why Activation Functions

- The purpose is to add **Non-Linearities** into Neural Networks
- Neural networks do function approximation, without non-linearities, neural networks cannot do anything more than a linear transformation
- But with more layers that include non-linearities, neural networks can approximate any **complex** function!



Matrix Notations



*: f is applied element-wise

$$f([z_1, z_2, z_3]) = [f(z_1), f(z_2), f(z_3)]$$

C: number of classes

d: input dimension, d_1, d_2: hidden dimensions

- Input layer: $\mathbf{x} \in \mathbb{R}^d$
- Hidden layer 1:

$$\mathbf{h}_1 = f(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) \in \mathbb{R}^{d_1}$$

$$\mathbf{W}^{(1)} \in \mathbb{R}^{d_1 \times d}, \mathbf{b}^{(1)} \in \mathbb{R}^{d_1}$$

- Hidden layer 2:

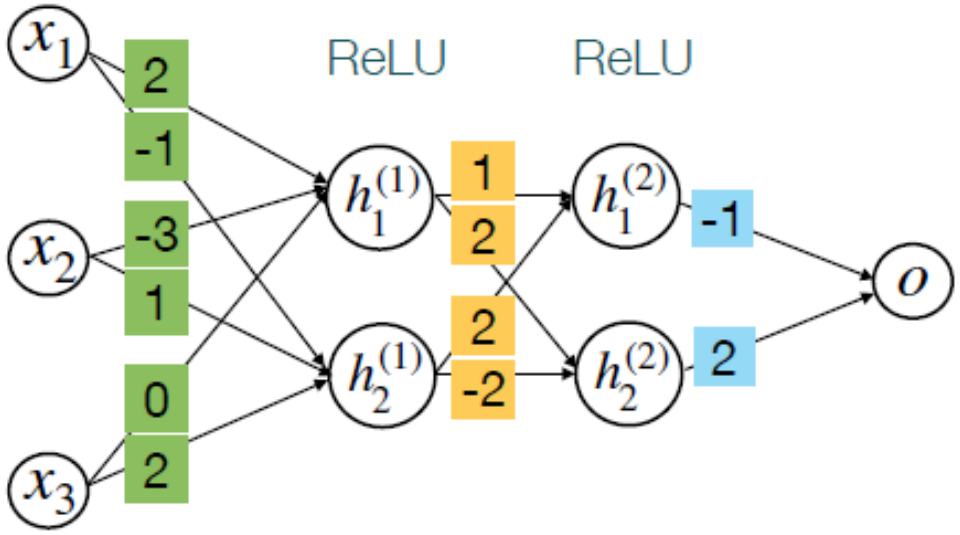
$$\mathbf{h}_2 = f(\mathbf{W}^{(2)}\mathbf{h}_1 + \mathbf{b}^{(2)}) \in \mathbb{R}^{d_2}$$

$$\mathbf{W}^{(2)} \in \mathbb{R}^{d_2 \times d_1}, \mathbf{b}^{(2)} \in \mathbb{R}^{d_2}$$

- Output layer:

$$\mathbf{y} = \mathbf{W}^{(o)}\mathbf{h}_2, \mathbf{W}^{(o)} \in \mathbb{R}^{C \times d_2}$$

Practice – Feedforward Neural Network



(Bias terms omitted in the next few slides)

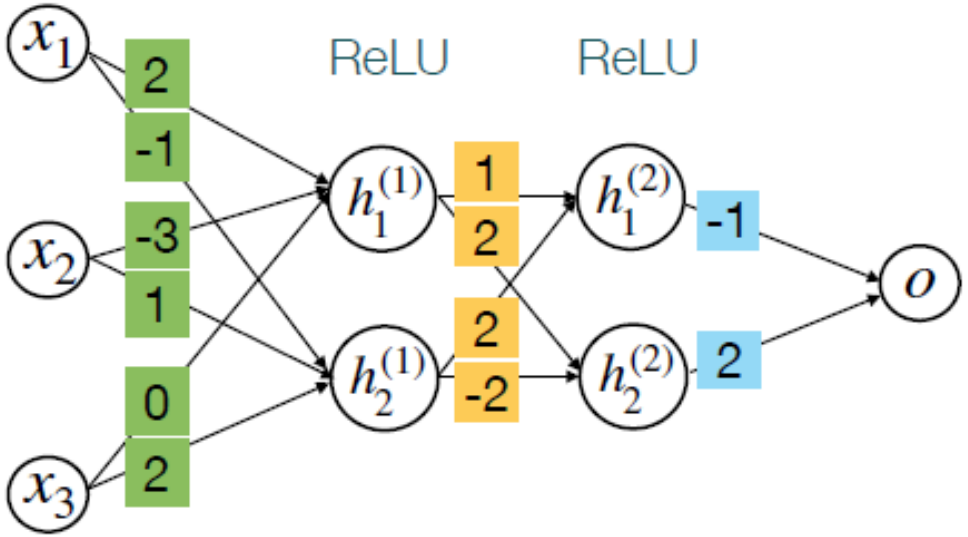
For $x_1 = x_2 = x_3 = 1$, what is the value of $h_1^{(1)}$?

(a) 0 (b) -1 (c) 1 (d) 2

Correct: (a), because of the RELU:

max(2 x 1 + (-3) x 1 + 0 x 1, 0) = max(-1, 0) = 0

Practice – Feedforward Neural Network



(Bias terms omitted in the next few slides)

$$\mathbf{h}^{(1)} = \text{ReLU}(\mathbf{W}^{(1)}\mathbf{x})$$ What is the matrix $\mathbf{W}^{(1)}$?

(a) $\begin{bmatrix} 2 & -1 \\ -3 & 1 \\ 0 & 2 \end{bmatrix}$ (b) $\begin{bmatrix} 2 & -3 & 0 \\ -1 & 1 & 2 \end{bmatrix}$ (c) $\begin{bmatrix} 1 & 2 \\ 2 & -2 \end{bmatrix}$

Correct: (b). $\mathbf{W}^{(1)}$ is a 2 x 3 matrix.

Training Feedforward Neural Networks

- Softmax: applying the softmax function on output logits to get predicted probability

$$\mathbf{y} = \mathbf{W}^{(o)}\mathbf{h}_2, \mathbf{W}^{(o)} \in \mathbb{R}^{C \times d_2}$$

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{y}) \qquad \text{softmax}(\mathbf{y})_k = \frac{\exp(y_k)}{\sum_{j=1}^{C} \exp(y_j)} \qquad \mathbf{y} = [y_1, y_2, \ldots, y_C]$$

- Training loss: maximize the probability of the correct class y or equivalently we can minimize the negative log probability of that class

$$\min_{\mathbf{W}^{(1)}, \mathbf{W}^{(2)}, \mathbf{W}^{(o)}} \; - \sum_{(\mathbf{x}, y) \in D} \log \hat{\mathbf{y}}_y$$

Training Feedforward Neural Networks

- Optimizer: stochastic gradient descent (SGD) to train feedforward NN

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

stepping size or learning rate

Model parameters include W weight matrix and b bias terms between layers

Objective: learn model parameters to minimize loss

- Neural Networks are difficult to optimize. SGD can only converge to local minimum. **Initialization** and **Optimizer** matter a lot!

Training Feedforward NN – Back-propagation

Forward propagation:
from input to output layer

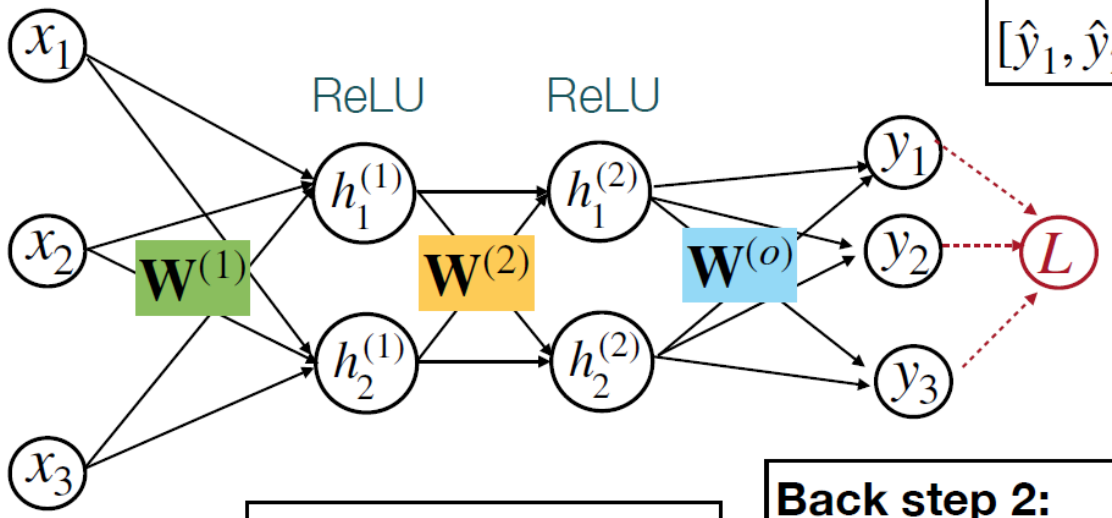
Given: x_1, x_2, x_3 and the class label y (a single training example)

Forward step 1:
Compute $h_1^{(1)}, h_2^{(1)}$

Forward step 2:
Compute $h_1^{(2)}, h_2^{(2)}$

Forward step 3:
Compute y_1, y_2, y_3 and
$[\hat{y}_1, \hat{y}_2, \hat{y}_3] = \text{softmax}[y_1, y_2, y_3]$

Forward step 4:
Compute loss
$L = -\log \hat{\mathbf{y}}_y$



Goal:
$\frac{\partial L}{\partial W^{(1)}}, \frac{\partial L}{\partial W^{(2)}}, \frac{\partial L}{\partial W^{(o)}}$

Back step 4:
Compute $\frac{\partial L}{\partial W^{(1)}}$

Back step 3:
Compute $\frac{\partial L}{\partial h_1^{(1)}}, \frac{\partial L}{\partial h_2^{(1)}}, \frac{\partial L}{\partial W^{(2)}}$

Back step 2:
Compute $\frac{\partial L}{\partial h_1^{(2)}}, \frac{\partial L}{\partial h_2^{(2)}}, \frac{\partial L}{\partial W^{(o)}}$

Back propagation:
from output to input layer

Back step 1:
Compute $\frac{\partial L}{\partial y_1}, \frac{\partial L}{\partial y_2}, \frac{\partial L}{\partial y_3}$

Training Feedforward NN – Back-propagation

```python
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(784, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 10)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

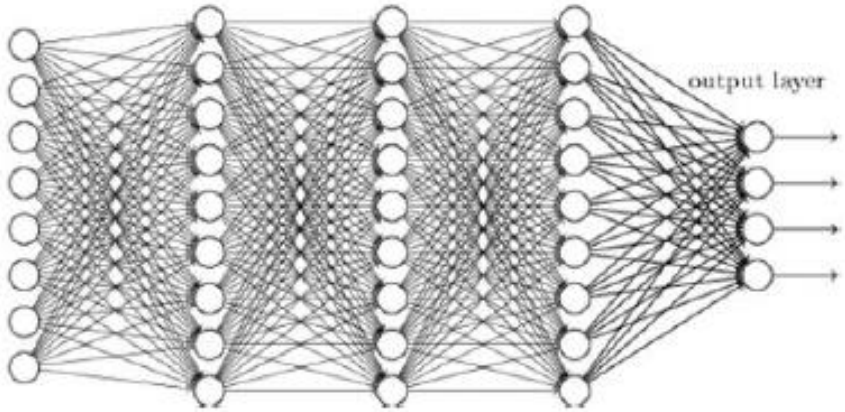
```python
import torch.optim as optim

net = Net()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

```python
outputs = net(inputs)
loss = criterion(outputs, labels)
loss.backward()
optimizer.step()
```

PyTorch did back-propagation for you in this one line of code!

Neural “bag-of-words” Models for Text Classification

Comparison: image vs text inputs



label = “dog”

a sometimes tedious film

i had to look away - this was god awful .

a gorgeous , witty , seductive movie .

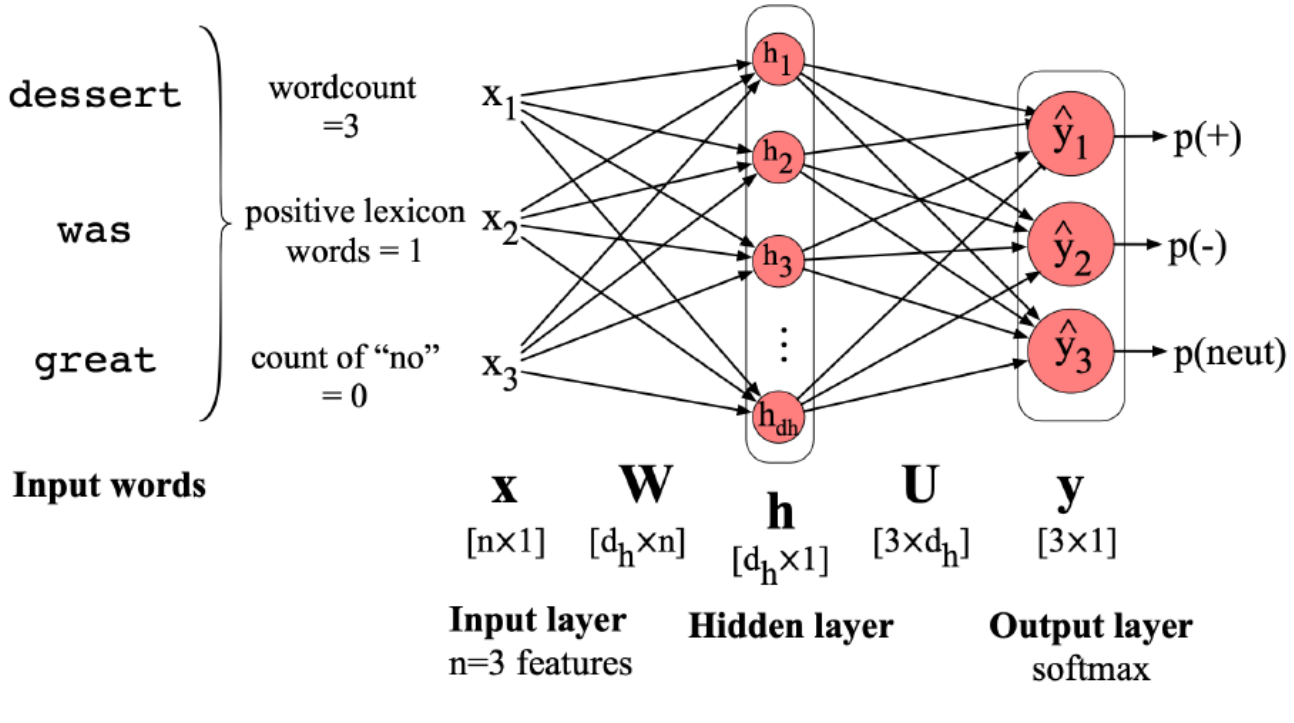
label = positive

- Images: fixed-size input, continuous values
- Text: **variable-length** input, discrete words

Solution 1: feature vector as input

- Input: $w_1, w_2, \ldots, w_K \in V$
- Output: $y \in C$

- Input: dessert was great
- Output: positive C = {positive, negative, neutral}



(each $\mathbf{x}_i$ is a hand-designed feature)

- $\mathbf{x} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n]$
- $\mathbf{h} = \text{ReLU}(\mathbf{W}\mathbf{x} + \mathbf{b})$
- $\mathbf{y} = \mathbf{U}\mathbf{h}$
- $\hat{\mathbf{y}} = \text{softmax}(\mathbf{y})$

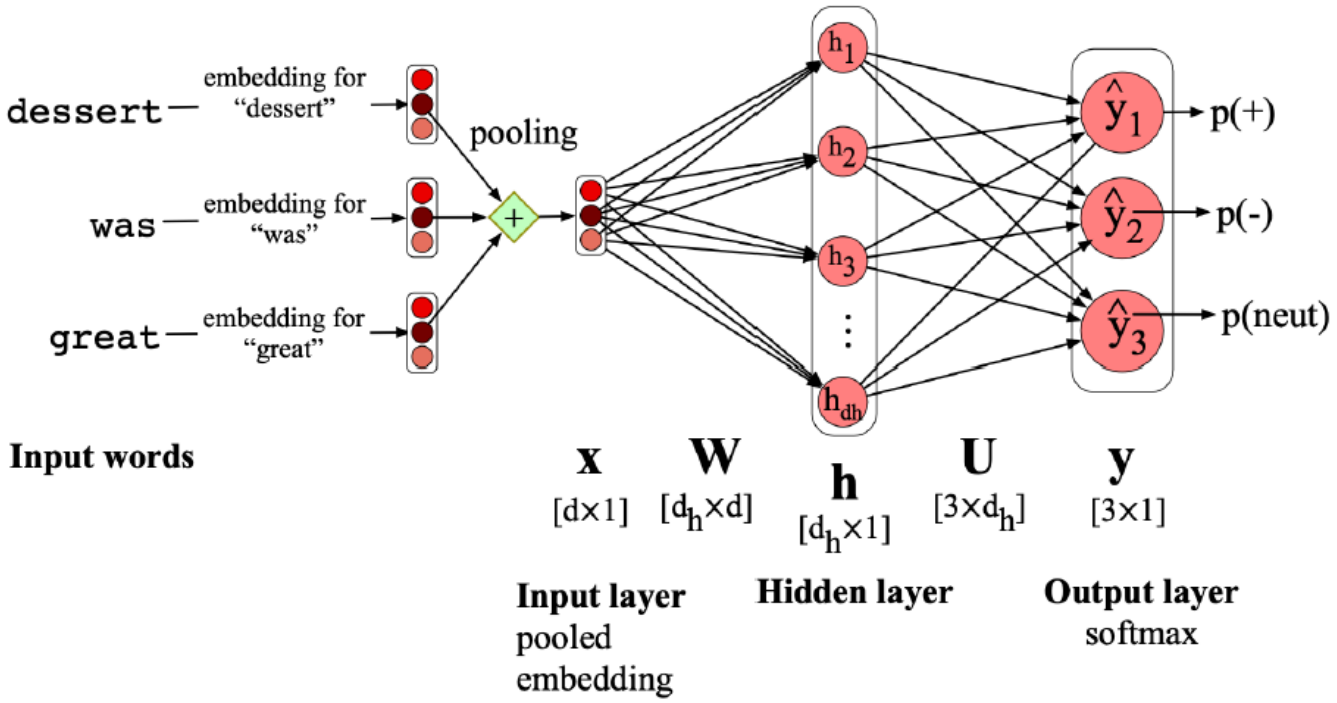
Deep learning has the promise to learn good features automatically..

Solution 2: pooling on word embeddings

- Aggregate all the word embeddings into a vector via pooling function

$$\mathbf{x}_{\text{mean}} = \frac{1}{K}\sum_{i=1}^{K} e(w_i)$$

pooling: sum, mean or max



- $\mathbf{x} = \frac{1}{K}\sum_{i=1}^{K} e(w_i)$
- $\mathbf{h} = \text{ReLU}(\mathbf{W}\mathbf{x} + \mathbf{b})$
- $\mathbf{y} = \mathbf{U}\mathbf{h}$
- $\hat{\mathbf{y}} = \text{softmax}(\mathbf{y})$

Important note: each input has a different K

Neural “bag-of-words” Model

Pro:

- This provides a simple and flexible way to handle variable-length input
- Learns feature representation (word embedding) automatically from data
- It can generalize to similar inputs through word embeddings

Cons:

- The model throw away any sequential information of the text

Neural "bag-of-words" Model – Training

- $\mathbf{x} = \frac{1}{K}\sum_{i=1}^{K} e(w_i)$
- $\mathbf{h} = \text{ReLU}(\mathbf{W}\mathbf{x} + \mathbf{b})$
- $\mathbf{y} = \mathbf{U}\mathbf{h}$
- $\hat{\mathbf{y}} = \text{softmax}(\mathbf{y})$

- Training data: $\{(d^{(1)}, y^{(1)}), \ldots, (d^{(m)}, y^{(m)})\}$
- Parameters: $\{\mathbf{W}, \mathbf{b}, \mathbf{U}\}$
- Optimize these parameters using stochastic gradient descent!
- Word embeddings can be treated as parameters too!

$$\mathbf{E} \in \mathbb{R}^{|V| \times d}$$

Neural “bag-of-words” Model – Training

- Common practice: initialize word embedding E using static pre-trained word embeddings (e.g. word2vec), and optimize them using SGD
- When the training data is small, don’t treat E as parameters
- When the training data is very large, initialization does not matter much, can use random initialization for word embeddings

Feedforward Neural Language Models

N-gram vs neural language models

Language models: Given $x_1, x_2, \ldots, x_n \in V$, the goal is to model:

$$P(x_1, x_2, \ldots, x_n) = \prod_{i=1}^{n} P(x_i \mid x_1, \ldots, x_{i-1})$$

Bigram: $$P(x_1, x_2, \ldots, x_n) = \prod_{i=1}^{n} P(x_i \mid x_{i-1})$$

Trigram: $$P(x_1, x_2, \ldots, x_n) = \prod_{i=1}^{n} P(x_i \mid x_{i-2}, x_{i-1})$$

Maximum likelihood estimate:

$$P(\text{sat}|\text{the cat}) = \frac{\text{count}(\text{the cat sat})}{\text{count}(\text{the cat})}$$

Limitations: cannot handle long histories

N-gram vs neural language models

- The larger n, the number of possible n-grams grow exponentially
- The larger n, the frequency counts for a specific n-gram is more sparse

$$P(w \mid \text{students opened their}) = \frac{\text{count}(\text{students opened their } w)}{\text{count}(\text{students opened their})}$$

- A lot of contexts are similar and simply counting their frequency cannot reflect their sematic similarity

I am a **good** _____ count(I am a good *w*)

I am a **great** _____ count(I am a great *w*)

e(good) $\approx$ **e(great)**

Neural LM mitigates these issues

Feedforward Neural Language Models

- Key Idea: instead of estimating raw probabilities, let's use a **feedforward neural network** to fit **the probabilistic distribution of language**

$$P(w \mid \text{I am a good})$$

$$P(w \mid \text{I am a great})$$

- Feedforward neural language models approximate the probability based on the previous m (e.g. 5) words – m is a hyper-parameter

$$P(x_1, x_2, \ldots, x_n) \approx \prod_{i=1}^{n} P(x_i \mid x_{i-m+1}, \ldots, x_{i-1})$$

Feedforward Neural Language Models

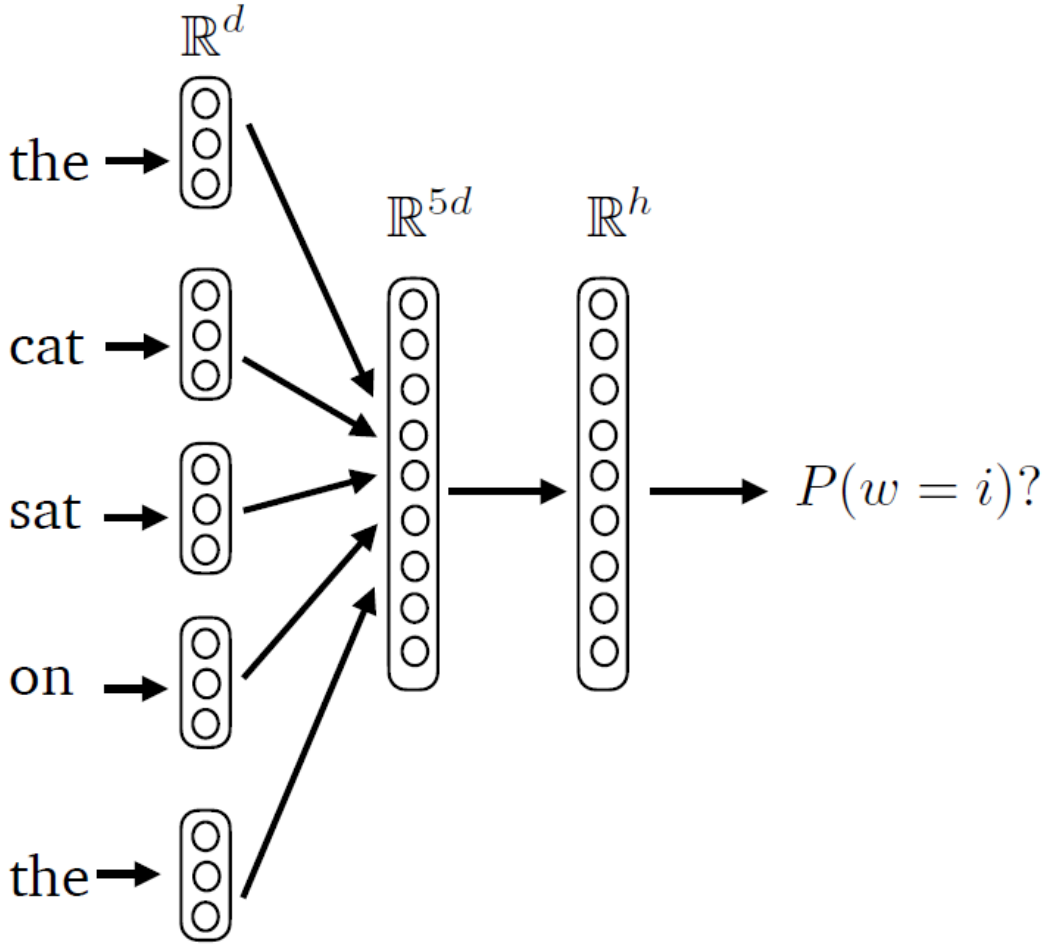
$$P(x_1, x_2, \ldots, x_n) \approx \prod_{i=1}^{n} P(x_i \mid x_{i-m+1}, \ldots, x_{i-1})$$

P(mat | the cat sat on the) = ?

d: word embedding size

h: hidden size

It is a |V|-way classification problem!



Feedforward Neural Language Models

P(mat | the cat sat on the) = ? d: word embedding size h: hidden size

Q: why concat instead of taking the average?

- Input layer (m= 5):

$$\mathbf{x} = [e(\text{the}); e(\text{cat}); e(\text{sat}); e(\text{on}); e(\text{the})] \in \mathbb{R}^{md}$$

- Hidden layer:

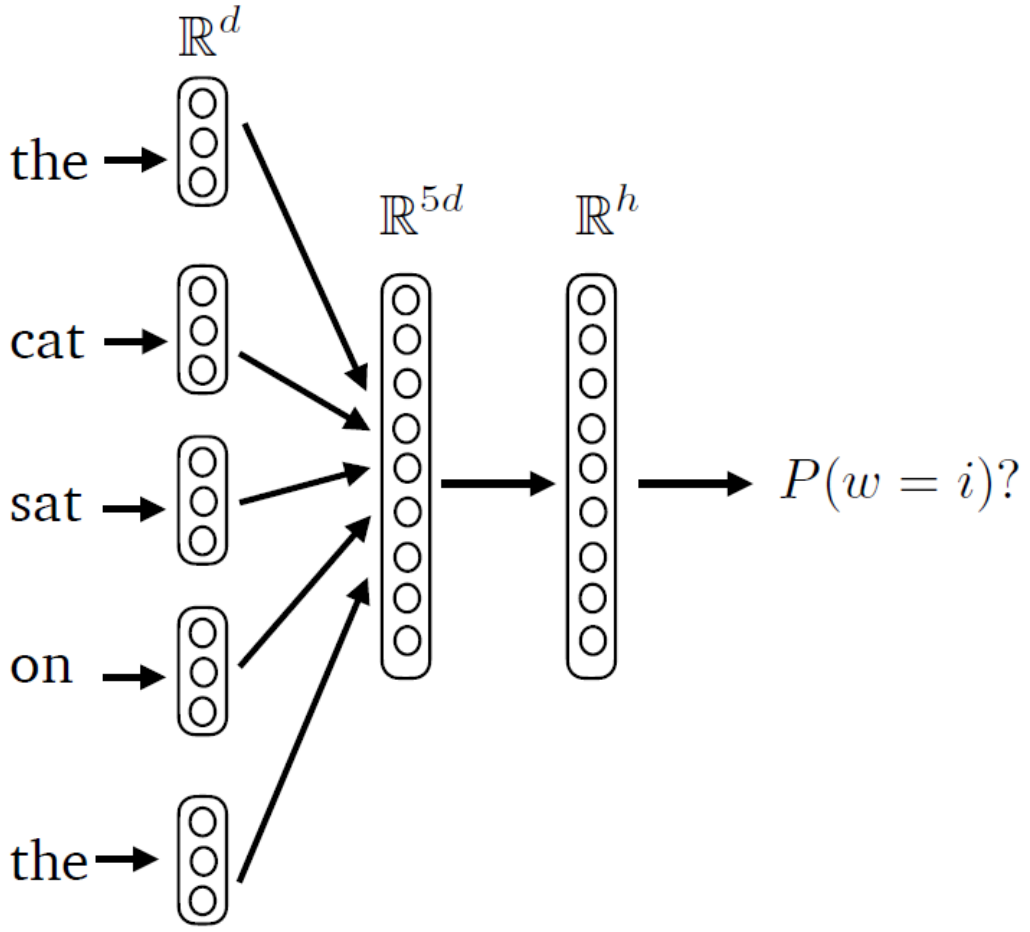
$$\mathbf{h} = \tanh(\mathbf{W}\mathbf{x} + \mathbf{b}) \in \mathbb{R}^{h}$$

- Output layer

$$\mathbf{z} = \mathbf{U}\mathbf{h} \in \mathbb{R}^{|V|}$$

$$P(w = i \mid \text{the cat sat on the})$$

$$= \text{softmax}_i(\mathbf{z}) = \frac{e^{z_i}}{\sum_k e^{z_k}}$$



Feedforward Neural Language Models

d: word embedding size, h: hidden size

(a) $\mathbf{W} \in \mathbb{R}^{h \times d}, \mathbf{U} \in \mathbb{R}^{|V| \times h}$

(b) $\mathbf{W} \in \mathbb{R}^{h \times 5d}, \mathbf{U} \in \mathbb{R}^{|V| \times h}$

(c) $\mathbf{W} \in \mathbb{R}^{h \times 5d}, \mathbf{U} \in \mathbb{R}^{|V| \times d}$

(d) $\mathbf{W} \in \mathbb{R}^{h \times d}, \mathbf{U} \in \mathbb{R}^{d \times h}$

Correct: (b)

- Input layer (m= 5):

$$\mathbf{x} = [e(\text{the}); e(\text{cat}); e(\text{sat}); e(\text{on}); e(\text{the})] \in \mathbb{R}^{md}$$

- Hidden layer:

$$\mathbf{h} = \tanh(\mathbf{W}\mathbf{x} + b) \in \mathbb{R}^{h}$$

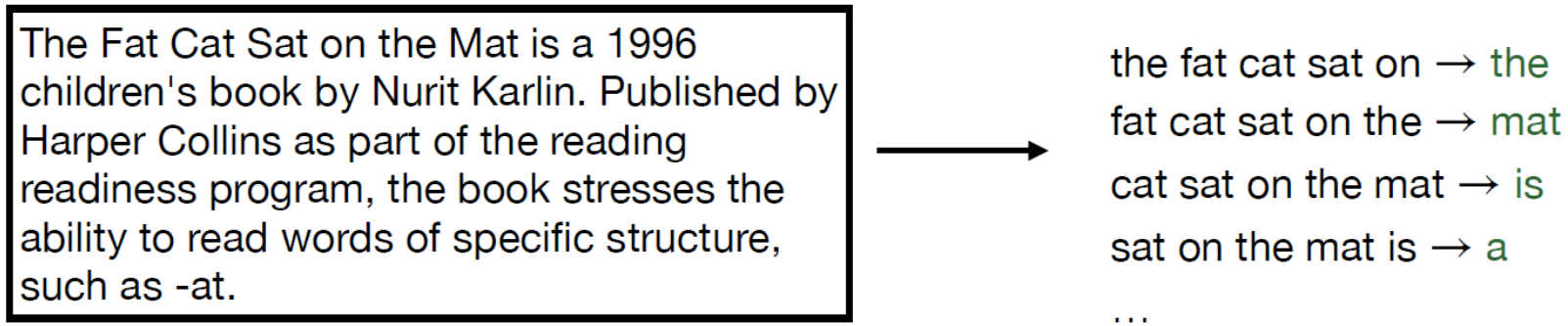
- Output layer

$$\mathbf{z} = \mathbf{U}\mathbf{h} \in \mathbb{R}^{|V|}$$

$$P(w = i \mid \text{the cat sat on the}) = \text{softmax}_i(\mathbf{z}) = \frac{e^{z_i}}{\sum_k e^{z_k}}$$

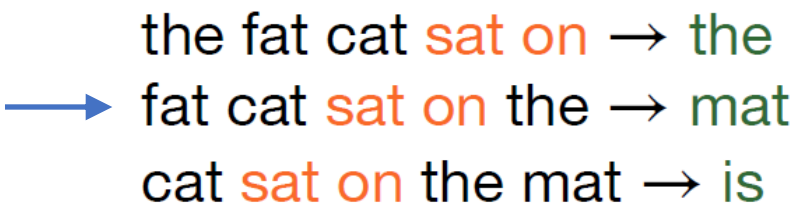
Feedforward Neural Language Models

- How to train this model? Use a lot of raw text to create training examples and run gradient-descent optimization



- Limitations: (1) W linearly scales with the context size m (2) The model learns separate patterns for different positions

"sat on" corresponds to different parameters in W



- Better solutions: Recurrent NNs, Transformers


Thank you!
UF
Herbert Wertheim
College of Engineering
UNIVERSITY of FLORIDA
LEADING THE CHARGE, CHARGING AHEAD