Midterm Review

COS 484

Catherine Cheng, Simon Park
3/3/2025

Today's Topics

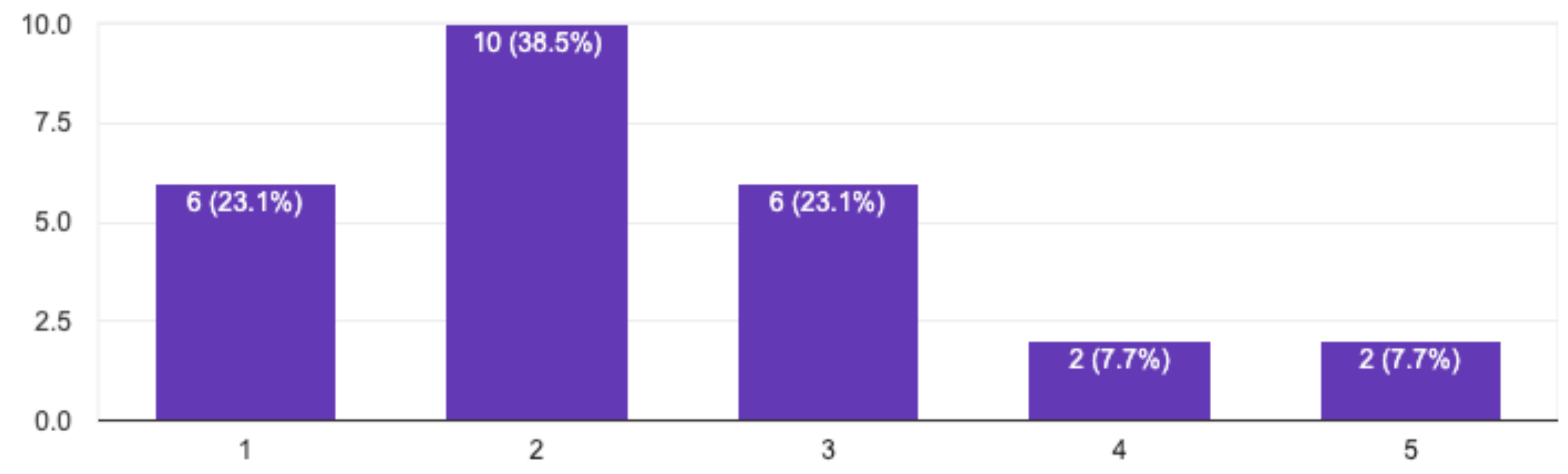
DRUM ROLL

Today's Topics

How confident do you feel about **Predict-based Word Embeddings (word2vec / skip-gram)** (Lecture 5)

Copy chart

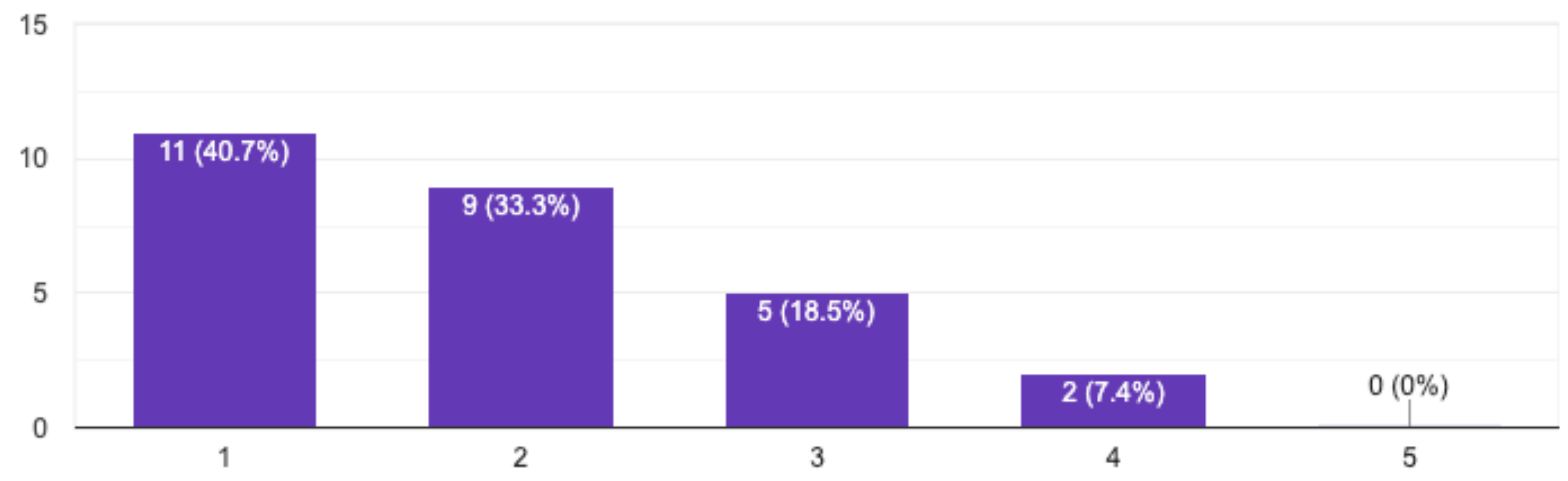
26 responses



How confident do you feel about **Sequence Models (MEMM / CRF)** (Lecture 6-7)

Copy chart

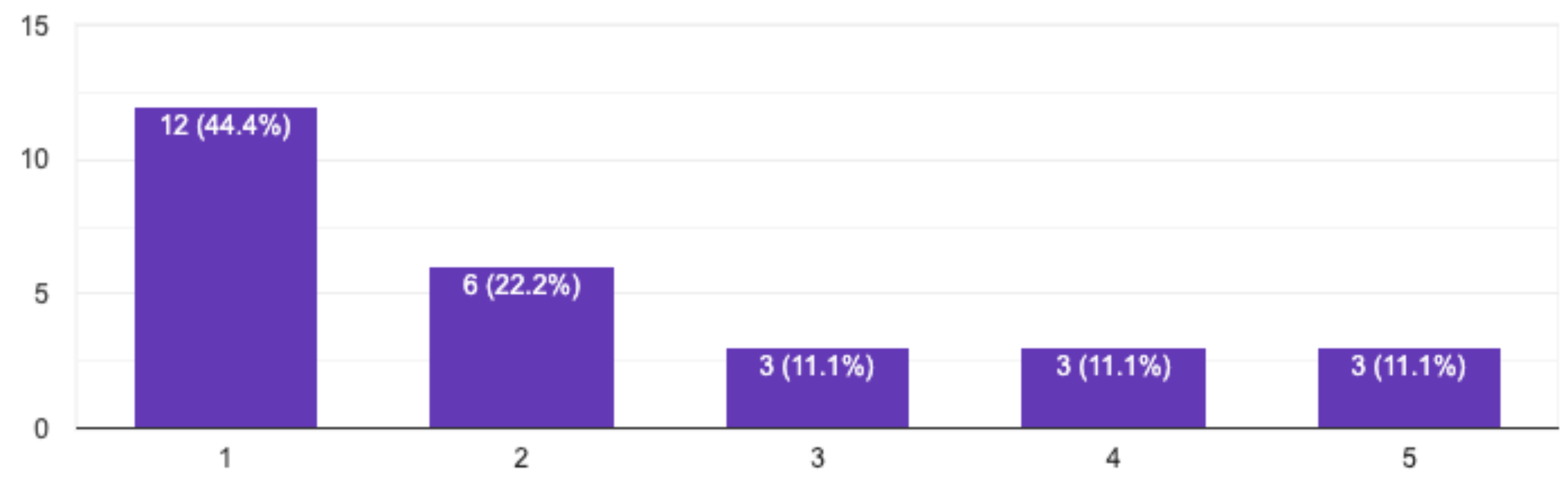
27 responses



How confident do you feel about **Decoding Strategies for Sequence Models (Greedy / Viterbi / Beam Search)** (Lecture 6-7)

Copy chart

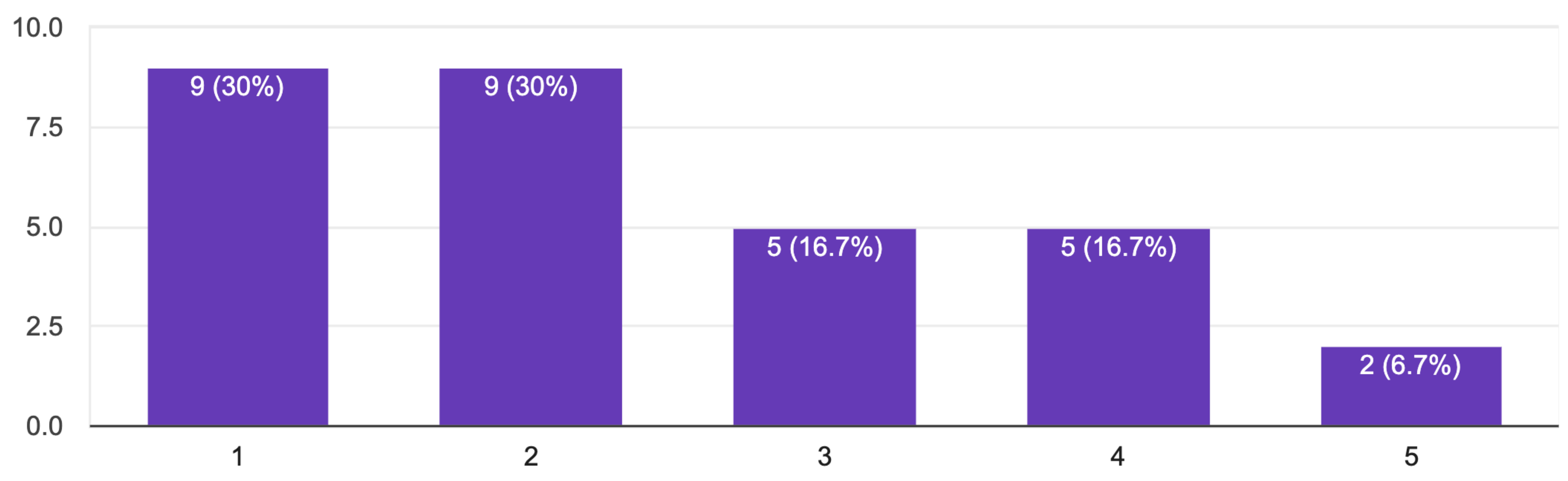
27 responses



How confident do you feel about **Neural Networks for NLP (FFNN / RNN)** (Lecture 8-9)

Copy chart

30 responses



Today's Topics

1. Predict-Based Word Embeddings (20 min)
2. Sequence Models (MEMM / CRF) (20 min)
3. Decoding Strategies for Sequence Models (20 min)
4. Neural Networks for NLP (FFNN / RNN) (10 min)
5. Free-for-all Q&A

Predict-Based Word Embeddings

Slides borrowed from Precept 3

Overview - Word Embeddings

- Represent words as vectors
  - e.g., apple -> [0.1, 0.2, 0.5]
  - Encode semantic information
  - Useful for downstream NLP tasks

QUESTION: what are good word vectors

IDEA: words that occur **near each other** should have **similar directions**

Overview - ML pipeline

- Initialize
- Loop over data:
  - Compute logits for all possible options
  - Normalize with softmax
  - Compute loss (negative log likelihood)
  - Compute gradient of loss w.r.t. each parameter
  - Update parameter via GD

Overview - ML pipeline

- Initialize
- Loop over data:
  - Compute logits for all possible options
  - Normalize with softmax
  - Compute loss (negative log likelihood)
  - Compute gradient of loss w.r.t. each parameter
  - Update parameter via GD

Word2vec - Computing Logits / Probability

- Given corpus, dictionary V, and desired dimension d
- Train an ML model with the following $2d\left|V\right|$ parameters:
  - **two embedding vectors** of dimension d for each word
    - **u** when the word is a target word
    - **v** when the word is a context word
- Given words **t** and **c**, probability that **c** appears in the context of **t** is determined by $\mathbf{u_t} \cdot \mathbf{v_c}$ (large when same direction / small when opposite)
- After softmax normalization, $\mathbb{P}[\mathbf{c} \,|\, \mathbf{t}] = \dfrac{\exp(\mathbf{u_t} \cdot \mathbf{v_c})}{\sum_{\mathbf{c'}} \exp(\mathbf{u_t} \cdot \mathbf{v_{c'}})}$

Overview - ML pipeline

- Initialize
- Loop over data:
  - Compute logits for all possible options
  - Normalize with softmax
  - Compute loss (negative log likelihood)
  - Compute gradient of loss w.r.t. each parameter
  - Update parameter via GD

Word2vec - Computing Loss

- Given a sequence of words $w_1, w_2, \cdots, w_T$ and context window size m
- For $t = 1,2,\cdots, T$,
  - Consider w_t a target word
  - For each $-m \le j \le m, j \neq 0$, consider w_{t+j} a context word
  - Compute probability of the (target, context) pair $\mathbb{P}[w_{t+j} | w_t]$
  - Compute loss (negative log likelihood), assuming all context words happen independently

$$L_t = -\log \prod_{-m \le j \le m, j \neq 0} \mathbb{P}[w_{t+j} | w_t] = -\sum_{-m \le j \le m, j \neq 0} \log \mathbb{P}[w_{t+j} | w_t]$$

Word2vec - Computing Loss

- Loss at position t

$$L_t = -\sum_{-m \leq j \leq m, j \neq 0} \log \mathbb{P}[w_{t+j} | w_t]$$

- Average loss across all position

$$L = -\frac{1}{T} \sum_{t=1}^{T} \sum_{-m \leq j \leq m, j \neq 0} \log \mathbb{P}[w_{t+j} | w_t]$$

Overview - ML pipeline

- Initialize
- Loop over data:
  - Compute logits for all possible options
  - Normalize with softmax
  - Compute loss (negative log likelihood)
  - Compute gradient of loss w.r.t. each parameter
  - Update parameter via GD

Word2vec - Updating Parameters

- Gradient update via

$$\mathbf{u} \leftarrow \mathbf{u} - \eta \frac{\partial L}{\partial \mathbf{u}} \qquad \mathbf{v} \leftarrow \mathbf{v} - \eta \frac{\partial L}{\partial \mathbf{v}}$$

Where if we let $L_{t,c} = -\log \frac{\exp(\mathbf{u_t} \cdot \mathbf{v_c})}{\sum_{\mathbf{c'}} \exp(\mathbf{u_t} \cdot \mathbf{v_{c'}})}$ for a particular (t, c) pair,

$$\frac{\partial L_{t,c}}{\partial \mathbf{u}_t} = -\mathbf{v}_c + \sum_{c' \in V} \mathbb{P}[c' \mid t] \mathbf{v}_{c'}$$ (lecture 5)

$$\frac{\partial L_{t,c}}{\partial \mathbf{v}_k} = \begin{cases} (\mathbb{P}[k \mid t] - 1)\mathbf{u}_t & k = c \\ \mathbb{P}[k \mid t]\mathbf{u}_t & k \neq c \end{cases}$$ (assignment 2)

Negative Sampling

- Naive implementation of skip-gram updates too many parameters
- Instead, modify the problem setup:
  - Given target word t and context word c
  - Randomly sample **K** alternative context words $c_1, c_2, \cdots, c_K$
  - Check if model predicts that c should be in the context of t
  - Check if model predicts that c_i should **not** be in the context of t

$$L_{\mathbf{t},\mathbf{c}} = -\log \sigma(\mathbf{u_t} \cdot \mathbf{v_c}) - \sum_{i=1}^{K} \mathop{\mathbb{E}}_{\mathbf{c_i} \sim V} \log \sigma(-\mathbf{u_t} \cdot \mathbf{v_{c_i}})$$

See precept 3 for deriving gradients

Sequence Models - MEMM

Slides borrowed from Precept 4

Overview - Sequence Models

- We have a sequence of words (outputs) $o_1, o_2, \cdots, o_T$
- Would like to get a sequence of tags (states) $s_1, s_2, \cdots, s_T$
- Need to know $\mathbb{P}[S \mid O]$

<u>HMM (Generative)</u>

Uses Bayes' Rule

$\mathbb{P}[S \mid O] \propto \mathbb{P}[O \mid S]\mathbb{P}[S]$

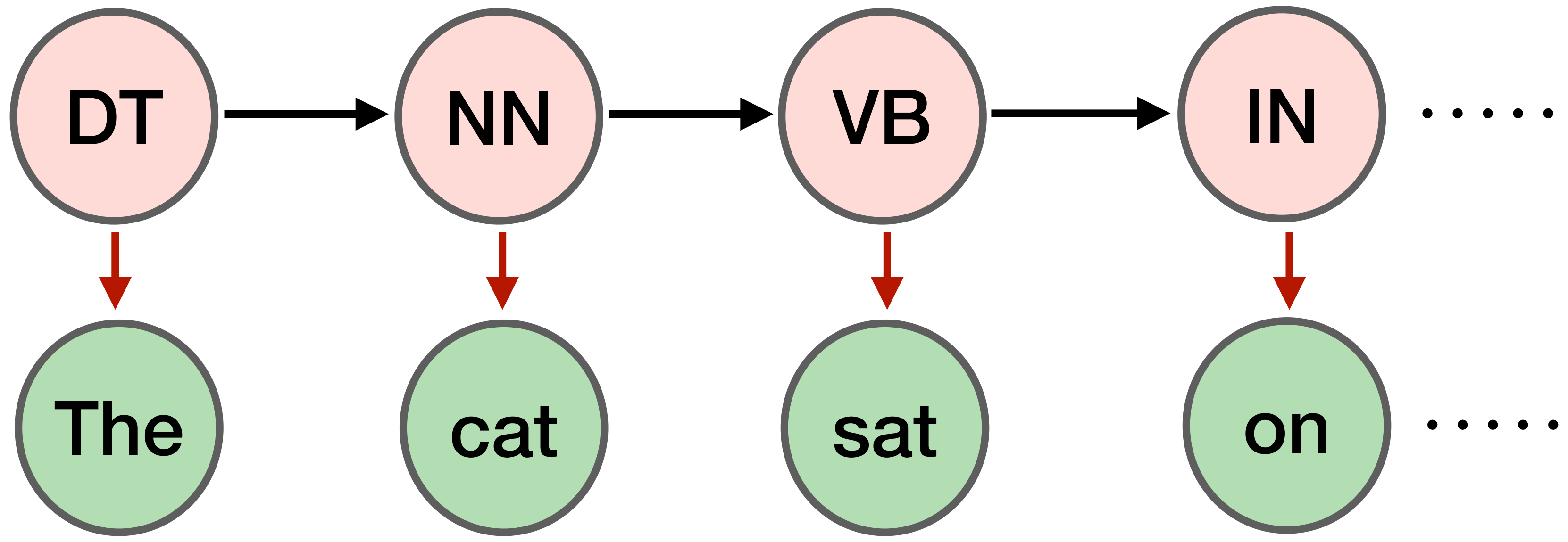
$\mathbb{P}[S]$: initial / transition prob

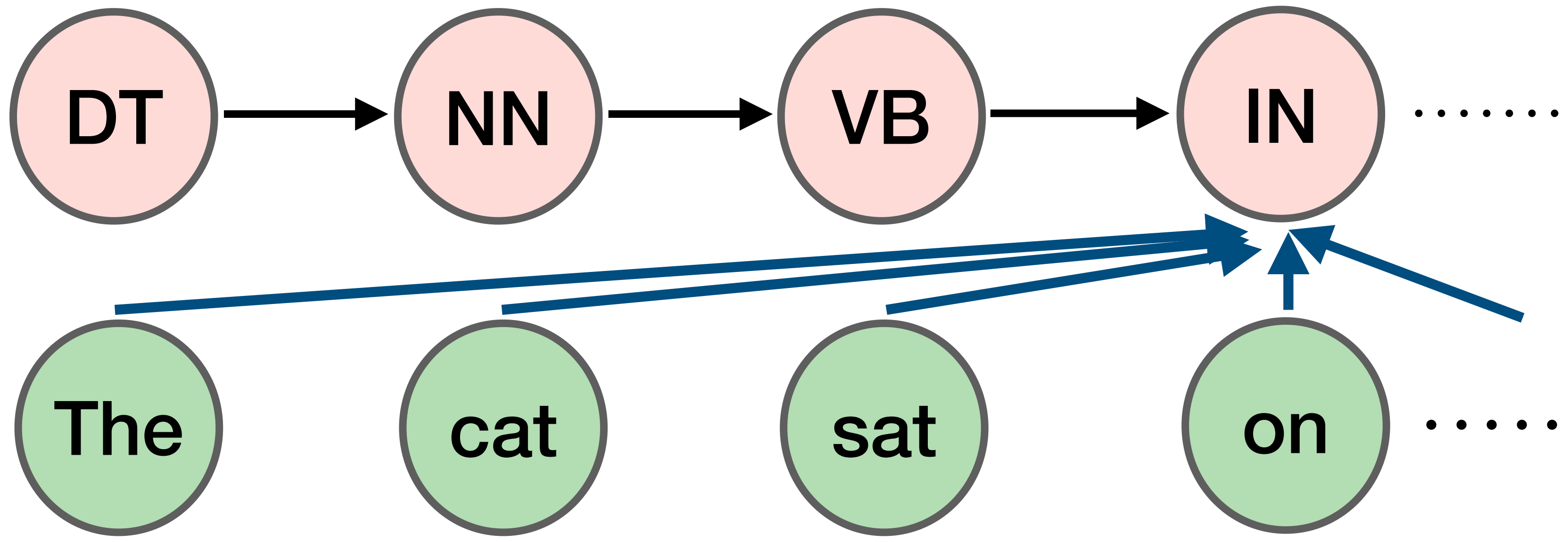
$\mathbb{P}[O \mid S]$: emission prob

<u>MEMM (Discriminative)</u>

Directly computes $\mathbb{P}[S \mid O]$

Overview - Sequence Models





<u>HMM (Generative)</u>

With bigram assumption

$$\mathbb{P}[S \mid O] = \prod_{i=1}^{n} \mathbb{P}[o_i \mid s_i]\mathbb{P}[s_i \mid s_{i-1}]$$

Each word depends on its tag

Each tag depends on previous tag

<u>MEMM (Discriminative)</u>

With bigram assumption

$$\mathbb{P}[S \mid O] = \prod_{i=1}^{n} \mathbb{P}[s_i \mid s_{i-1}, O]$$

Each tag depends on all words + the previous tag

Overview - ML pipeline

- Initialize
- Loop over data:
  - Compute logits for all possible options
  - Normalize with softmax
  - Compute loss (negative log likelihood)
  - Compute gradient of loss w.r.t. each parameter
  - Update parameter via GD

Overview - ML pipeline

- Initialize
- Loop over data:
  - Compute logits for all possible options
  - Normalize with softmax
  - Compute loss (negative log likelihood)
  - Compute gradient of loss w.r.t. each parameter
  - Update parameter via GD

MEMM - Featurization

- Choose features $f_1, f_2, \cdots, f_m$ (functions) whose values depend on
  - Current tag s_i
  - Previous tag s_{i-1}
  - All words O
  - Position index i
- For example,
  - $f_1 = 1(s_i = \text{VB} \ \wedge \ s_{i-1} = \text{NNP})$ will be 1 if current tag is “Verb, base form” and the previous tag is “Proper noun, singular” else 0

MEMM - Computing Logits / Probability

- Given the set of all possible K tags, m features $\mathbf{f} = (f_1, f_2, \cdots, f_m)$
- Train an ML model with m parameters: $\mathbf{w} \in \mathbb{R}^m$
- Given words O and previous tag s_{i-1}, probability that s_i is a particular tag s (as opposed to an alternative s') is determined by $\mathbf{w} \cdot \mathbf{f}(s_i = s, s_{i-1}, O, i)$
- After softmax normalization,

$$\mathbb{P}[s_i = s \mid s_{i-1}, O] = \frac{\exp(\mathbf{w} \cdot \mathbf{f}(s_i = s, s_{i-1}, O, i))}{\sum_{\mathbf{s}'} \exp(\mathbf{w} \cdot \mathbf{f}(s_i = s', s_{i-1}, O, i))}$$

Overview - ML pipeline

- Initialize
- Loop over data:
  - Compute logits for all possible options
  - Normalize with softmax
  - Compute loss (negative log likelihood)
  - Compute gradient of loss w.r.t. each parameter
  - Update parameter via GD

MEMM - Computing Loss

- Given a sequence of words $O = (o_1, o_2, \cdots, o_n)$ and a sequence of tags $S = (s_1, s_2, \cdots, s_n)$
- For $i = 1,2,\cdots, n$,
  - Compute probability of the observed tag $\mathbb{P}[s_i \mid s_{i-1}, O]$
  - Compute loss (negative log likelihood), assuming all tags happen independently

$$L = -\log \prod_{i=1}^{n} \mathbb{P}[s_i \mid s_{i-1}, O] = -\sum_{i=1}^{n} \log \mathbb{P}[s_i \mid s_{i-1}, O]$$

Overview - ML pipeline

- Initialize
- Loop over data:
  - Compute logits for all possible options
  - Normalize with softmax
  - Compute loss (negative log likelihood)
  - Compute gradient of loss w.r.t. each parameter
  - Update parameter via GD

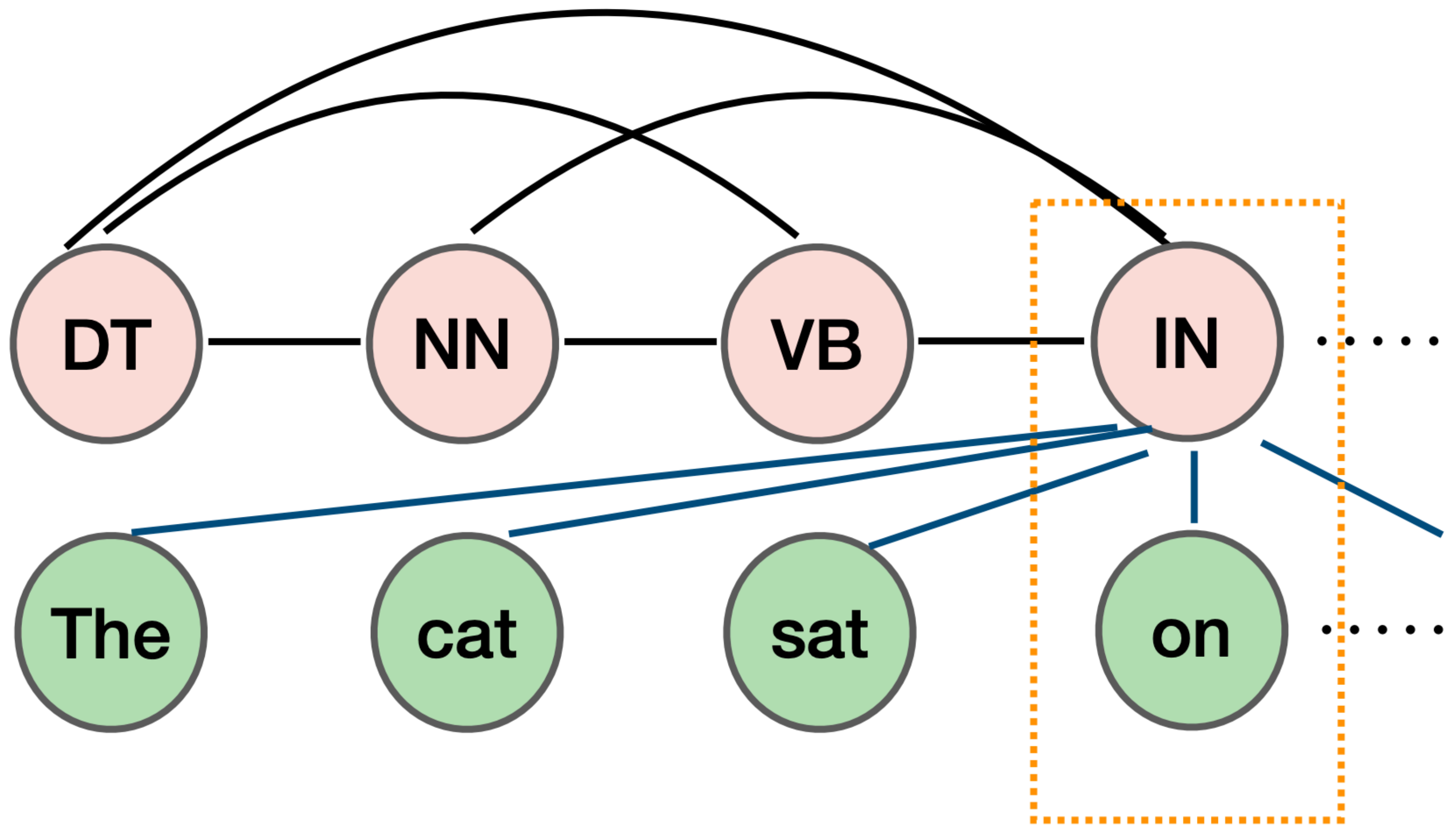
MEMM - Updating Parameters

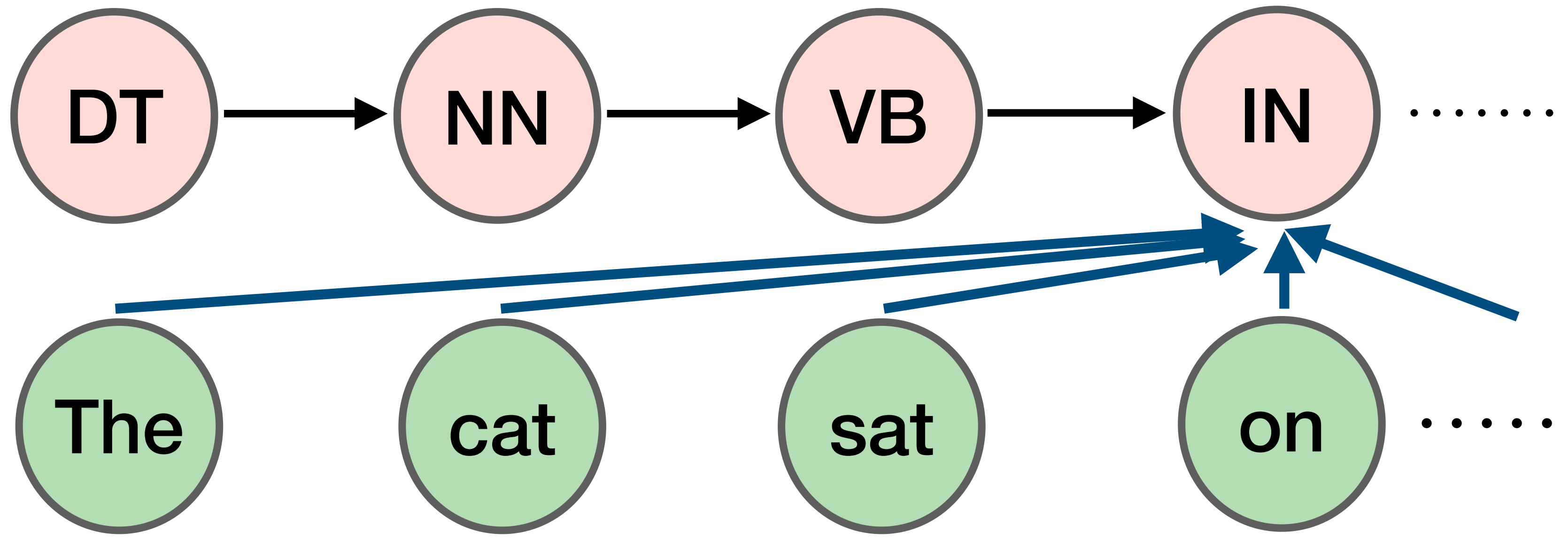
- Gradient update via

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \frac{\partial L}{\partial \mathbf{w}}$$

Sequence Models - CRF

Overview - Sequence Models





<u>CRF (Discriminative)</u>

NO Markov assumption

$$\mathbb{P}[S \mid O] = \mathbb{P}[S \mid O]$$

Each tag depends on all words + **all tags**

<u>MEMM (Discriminative)</u>

With bigram assumption

$$\mathbb{P}[S \mid O] = \prod_{i=1}^{n} \mathbb{P}[s_i \mid s_{i-1}, O]$$

Each tag depends on all words + the **previous** tag

Overview - ML pipeline

- Initialize
- Loop over data:
  - Compute logits for all possible options
  - Normalize with softmax
  - Compute loss (negative log likelihood)
  - Compute gradient of loss w.r.t. each parameter
  - Update parameter via GD

CRF - Featurization

- Choose features $f_1, f_2, \cdots, f_m$ (functions) whose values depend on
  - Current tag s_i
  - Previous tag s_{i-1}
  - All words O
  - Position index i
- Then define **global** features $F_1, F_2, \cdots, F_m$ as the sum of the **same feature** applied across the input sequence; i.e., $F_k = \sum_{i=1}^{n} f_k(s_i, s_{i-1}, O, i)$

CRF - Featurization

- **Local** features $f_1, f_2, \cdots, f_m$ (functions) depend on
  - Current tag s_i
  - Previous tag s_{i-1}
  - All words O
  - Position index i
- **Global** features $F_1, F_2, \cdots, F_m$ (functions) depend on
  - All tags S
  - All words O

CRF - Computing Logits / Probability

- Given the set of all possible K tags, m global features $\mathbf{F} = (F_1, F_2, \cdots, F_m)$
- Train an ML model with m parameters: $\mathbf{w} \in \mathbb{R}^m$
- Given words O, probability that we see a particular sequence of tags S (as opposed to an alternative S') is determined by $\mathbf{w} \cdot \mathbf{F}(S, O)$
- After softmax normalization, $\mathbb{P}[S \mid O] = \dfrac{\exp(\mathbf{w} \cdot \mathbf{F}(S, O))}{\sum_{\mathbf{S}'} \exp(\mathbf{w} \cdot \mathbf{F}(S', O))}$

Overview - ML pipeline

- Initialize
- Loop over data:
  - Compute logits for all possible options
  - Normalize with softmax
  - Compute loss (negative log likelihood)
  - Compute gradient of loss w.r.t. each parameter
  - Update parameter via GD

CRF - Computing Loss / Updating Parameters

- Given a sequence of words $O = (o_1, o_2, \cdots, o_n)$ and a sequence of tags $S = (s_1, s_2, \cdots, s_n)$
- Compute probability of the observed tags $\mathbb{P}[S \mid O]$
- Compute loss (negative log likelihood) $L = -\log \mathbb{P}[S \mid O]$
- Gradient update via $\mathbf{w} \leftarrow \mathbf{w} - \eta \frac{\partial L}{\partial \mathbf{w}}$
  - Can be efficiently done via “Forward-backward algorithm” (dynamic programming)

Decoding Strategies for Sequence Models

Viterbi Decoding (Core Idea)

- Compute the joint probability of the sequence $(s_0, \ldots, s_{i-1}, s_i = s)$ that gives us the best score / highest probability

- Recover the sequence via backtracking

Viterbi Decoding (Implementation)

- Recall: in class, we iteratively define $\text{score}_1(s) = P(o_1 \mid s) \cdot P(s)$

$$\ldots\ldots$$

$$\text{score}_i(s) = \max_{s_{i-1}} P(o_i \mid s)P(s \mid s_{i-1}) \cdot \text{score}_{i-1}(s_{i-1})$$

Viterbi Decoding (Implementation)

- Recall: in class, we iteratively define $\text{score}_1(s) = P(o_1 \mid s) \cdot P(s)$

$$\ldots\ldots$$

$$\text{score}_i(s) = \max_{s_{i-1}} P(o_i \mid s)P(s \mid s_{i-1}) \cdot \text{score}_{i-1}(s_{i-1})$$

Need to compute for all possible s!

For each s, the best s_{i-1} may be different!

Viterbi Decoding (Implementation)

- Recall: in class, we iteratively define $\text{score}_1(s) = P(o_1 \mid s) \cdot P(s)$

$$\cdots\cdots$$

$$\text{score}_i(s) = \max_{s_{i-1}} P(o_i \mid s)P(s \mid s_{i-1}) \cdot \text{score}_{i-1}(s_{i-1})$$

Greedy: $\text{score}_i(s) = P(o_i \mid s)P(s \mid s_{i-1}) \cdot \max_{s_{i-1}} \text{score}_{i-1}(s_{i-1})$

Viterbi Decoding (Implementation)

- Dynamic programming
  - score$[i, s]$: best probability of a sequence ending with j at the i-th token

	Tag 0	Tag 1	Tag 2
Token 0	score[0,0]	score[0,1]	score[0,2]
Token 1			
Token 2			

Viterbi Decoding (Implementation)

- Dynamic programming
    - score$[i, s]$: best probability of a sequence ending with j at the i-th token

	Tag 0	Tag 1	Tag 2
Token 0	score[0,0]	score[0,1]	score[0,2]
Token 1			
Token 2			

↓ $\times P(o_1 \mid 0) \cdot P(0 \mid 0)$

Viterbi Decoding (Implementation)

- Dynamic programming
  - score$[i, s]$: best probability of a sequence ending with j at the i-th token

	Tag 0	Tag 1	Tag 2
Token 0	score[0,0]	score[0,1]	score[0,2]
Token 1			
Token 2			

$\times P(o_1 \mid 0) \cdot P(0 \mid 1)$

Viterbi Decoding (Implementation)

- Dynamic programming
  - score$[i, s]$: best probability of a sequence ending with j at the i-th token

	Tag 0	Tag 1	Tag 2
Token 0	score[0,0]	score[0,1]	score[0,2]
Token 1			
Token 2			

$\times P(o_1 \mid 0) \cdot P(0 \mid 2)$

Viterbi Decoding (Implementation)

- Dynamic programming
  - score$[i, s]$: best probability of a sequence ending with j at the i-th token

	Tag 0	Tag 1	Tag 2
Token 0	score[0,0]	score[0,1]	score[0,2]
Token 1	score[1,0]		
Token 2			

max!!

Refer to Precept 4 slides for a concrete example!

Viterbi Decoding (Analysis)

- Why does it work?

$$\text{score}_i(s) = \max_{s_0,\ldots,s_{i-1}} P(s_0, \ldots, s_{i-1}, s_i = s, o_0, \ldots, o_i)$$

$$\text{score}_i(s) = \max_{s_{i-1}} P(o_i \mid s) P(s \mid s_{i-1}) \cdot \text{score}_{i-1}(s_{i-1})$$

Viterbi Decoding (Analysis)

- Why does it work?

$$\text{score}_i(s) = \max_{s_0,\ldots,s_{i-1}} P(s_0, \ldots, s_{i-1}, s_i = s, o_0, \ldots, o_i)$$

$$\text{score}_i(s) = \max_{s_0,\ldots,s_{i-1}} P(o_i, s_i = s \mid s_0, \ldots, s_{i-1}, o_0, \ldots, o_{i-1})$$

Markov assumption!

$$P(s_0, \ldots, s_{i-1}, o_0, \ldots, o_{i-1})$$

$$\text{score}_i(s) = \max_{s_{i-1}} P(o_i \mid s) P(s \mid s_{i-1}) \cdot \text{score}_{i-1}(s_{i-1})$$

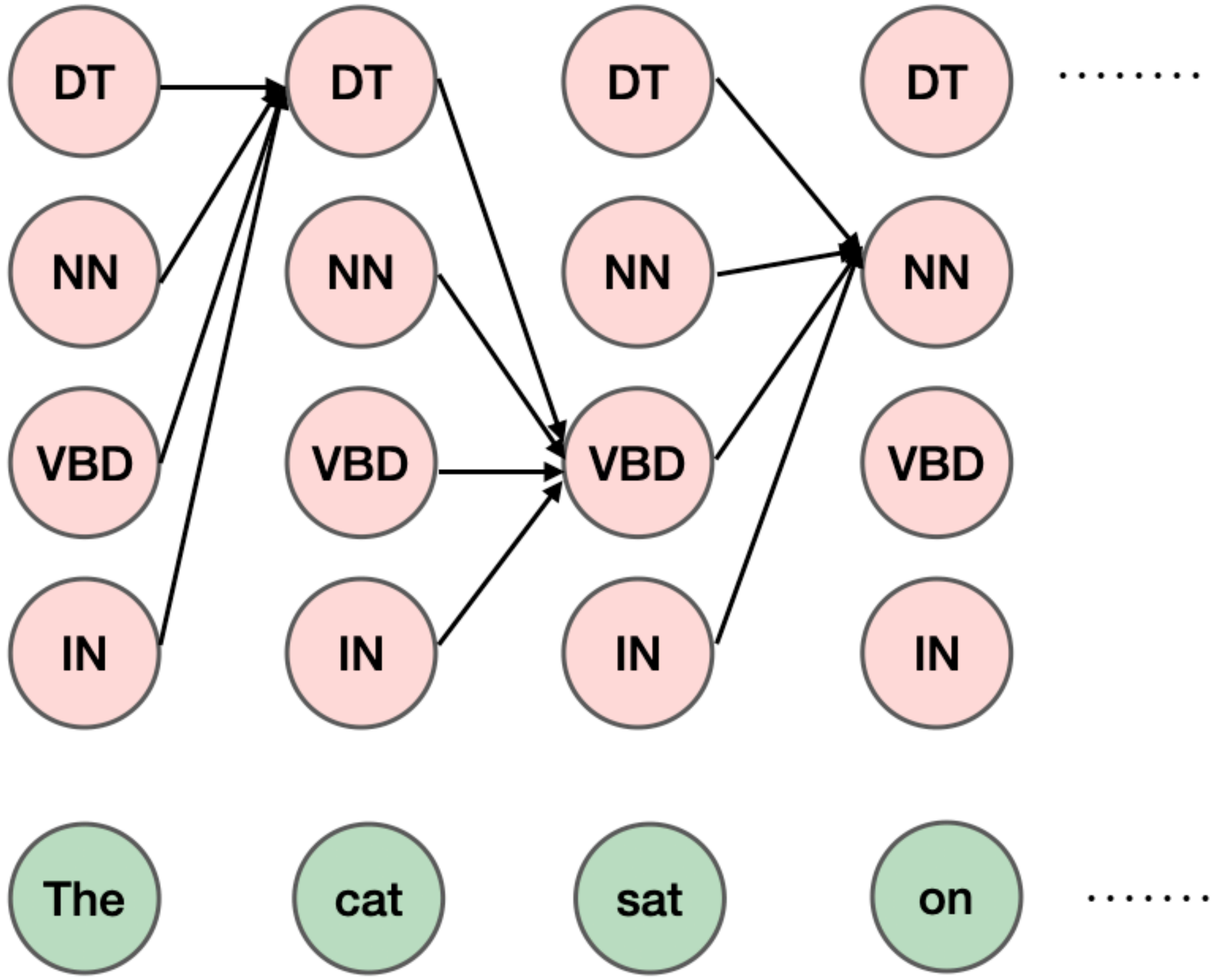
Viterbi Decoding (Analysis)

- Complexity: $O(nK^2)$
  - Very expensive if K is large

- Beam search: tradeoff between accuracy and efficiency
  - Set $K = \beta$ fixed (beam width): only keep track a few best sequences so far instead of exploring the entire space
  - Complexity: $O(nK\beta)$

Viterbi Decoding (MEMMs)

$$M[i,j] = \max_k M[i-1,k]\, P(s_i = j \mid s_{i-1} = k, O) \qquad 1 \le k \le K \quad 1 \le i \le n$$



$M[i,j]$ stores joint probability of most probable sequence of states ending with state j at time i

Precept 4

Neural Networks for NLP

FeedForward Neural Language Model (Core Idea)

- Approximate the probability based on the previous m words (context)
  - $$P(x_0, \ldots, x_n) \approx \prod_{i=0}^{n} P(x_i \mid x_{i-m+1}, \ldots, x_{i-1})$$
  - m is a hyperparameter

FeedForward Neural Language Model (Modeling)

- Input layer / **E**mbedding layer:
  - $\mathbf{x} = [Ex_0, \ldots, Ex_{m-1}]$
  - E: embedding matrix that transforms tokens to pre-trained embedding
- Hidden layer
  - $\mathbf{h} = \tanh(\mathbf{W}\mathbf{x} + \mathbf{b})$
  - $\mathbf{W}, \mathbf{b}, \tanh$: hidden weights, bias and activation
- Output layer / **U**nembedding layer:
  - $\mathbf{z} = \mathbf{U}\mathbf{h}$
  - Probability = $\text{softmax}_i(\mathbf{z}) = \dfrac{e^{z_i}}{\sum_k e^{z_k}}$

FeedForward Neural Language Model (Limitations)

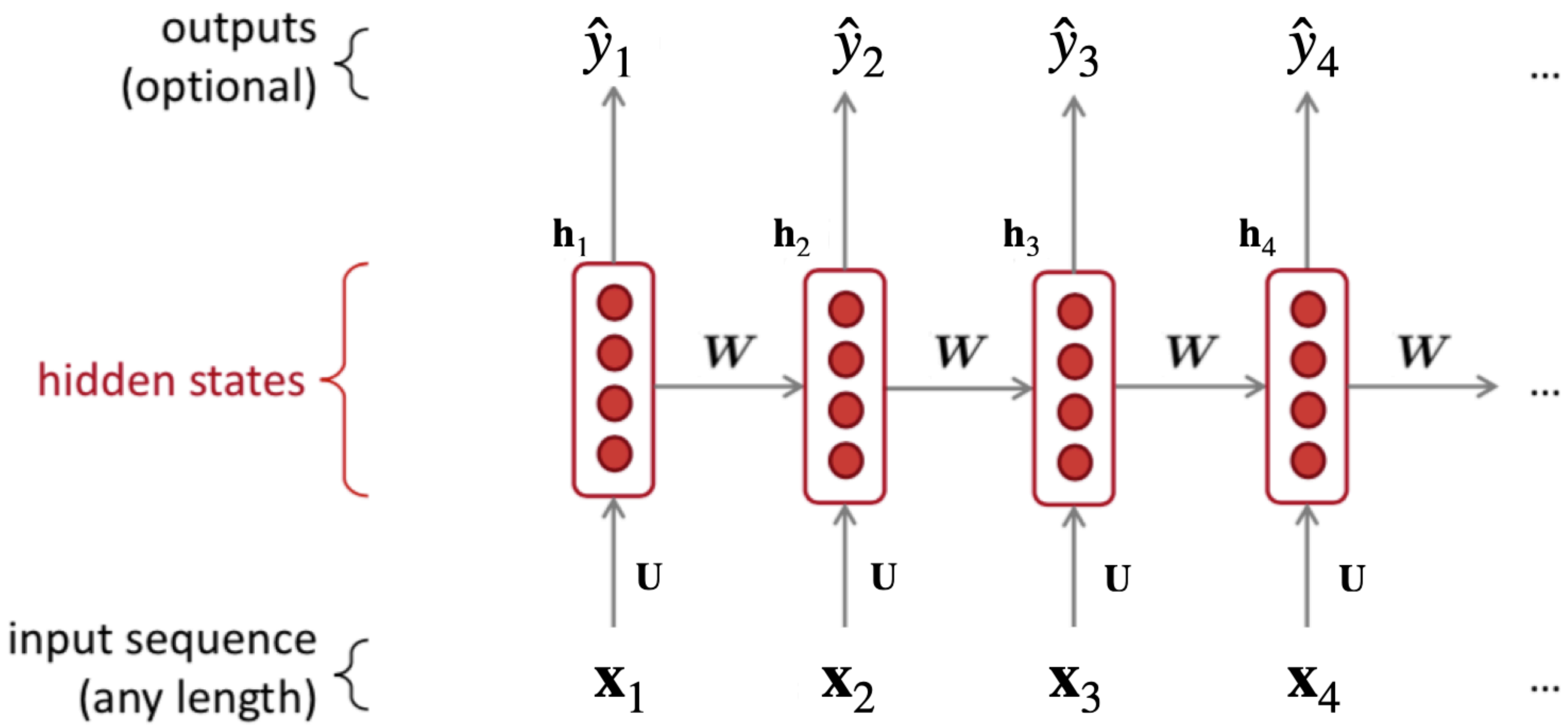
- $\mathbf{W}$ linearly scales with the context size m
- Model learns separate patterns for different positions

Recurrent Neural Network (Core Idea)

- Apply the same weights repeatedly at different positions
- Highly effective approach for various language modeling tasks

Recurrent Neural Network (Modeling)

- $\mathbf{h}_t = g(\mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t + \mathbf{b})$
  - g: activation
  - $\mathbf{W}, \mathbf{U}, \mathbf{b}$: learnable parameters



Lecture 9

Recurrent Neural Network

- No Markov assumption!

$$P(x_0, \ldots, x_n) = p(x_0) \cdot p(x_1 \mid x_0) \cdot \ldots \cdot p(x_n \mid x_0, \ldots, x_{n-1})$$

$$\approx P(x_1 \mid \mathbf{h}_0) \cdot \ldots \cdot P(x_n \mid \mathbf{h}_{n-1})$$

BackPropagation Through Time (BPTT)

- Generally,

$$\frac{\partial \ell}{\partial \mathbf{W}} = -\frac{1}{n}\sum_{t=1}^{n}\sum_{k=1}^{t}\frac{\partial \ell}{\partial \mathbf{h}_t}\left(\prod_{j=k+1}^{t}\frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}}\right)\frac{\partial \mathbf{h}_j}{\partial \mathbf{W}}$$

- Gradient exploding / vanishing problem if k, t are far away
  - Gradient exploding harms convergence -> solution: gradient clipping

- Become expensive to compute for long sequence
  - Truncated BPTT: only apply backprop for a smaller number of steps

Q&A

Good Luck!