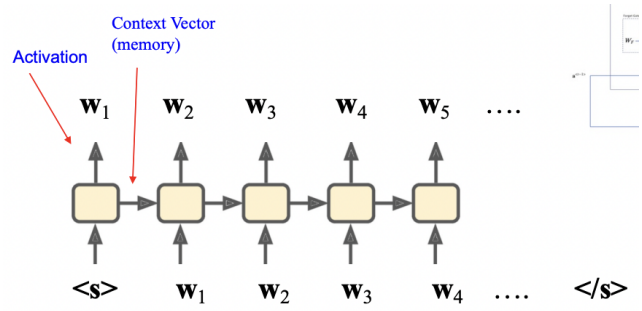
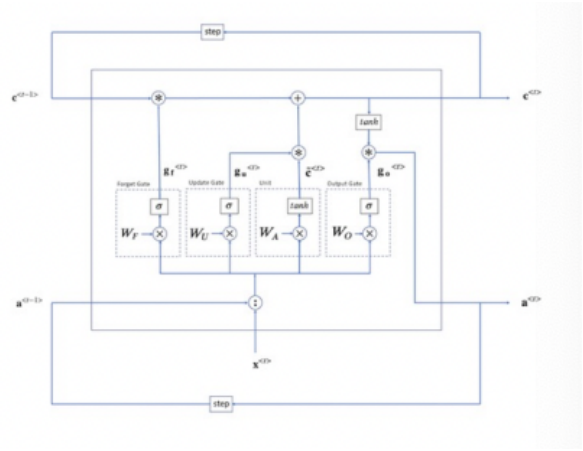
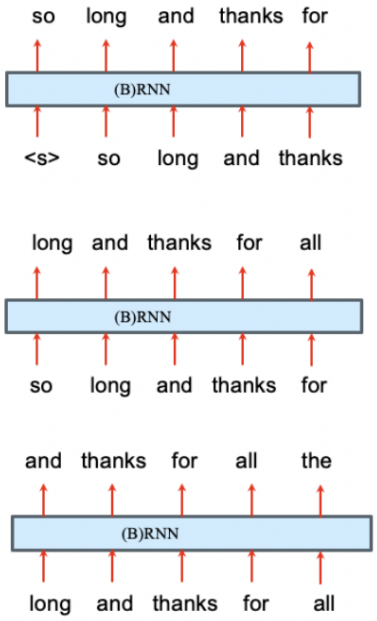
CAS CS 505
Lec 13

Generative Models with RNNs, Beam Search; Word and Document Embeddings

1. Generative Language Models with RNNs
    a. The basic idea is to do a vector-to-sequence model, starting with the start-of-sentence token:
    b. 

    c. LSTM cell has a context vector and an activation:


    d. One loop is memory (called context)
    e. Subneurons that erase from memory, writes to memory, calculate the output
    f. There is a path for activation
    g. To train an RNN a language model, we create a dataset of subsequences of sentences:
        i. Sentence: <s> so long and thanks for all the fish ! </s>
        ii. Break into equal-length subsequences (here 5, but typically longer) → looks like n-gram in some sense
        iii. <s> so long and thanks
                so long and thanks for
                …
                for all the fish !
                all the fish ! </s>

2. Applications of RNNs: Generative Language Models
    a. Then we train the network on inputs and outputs (difficult to go bi-direction → one dimensional process)
    b. 

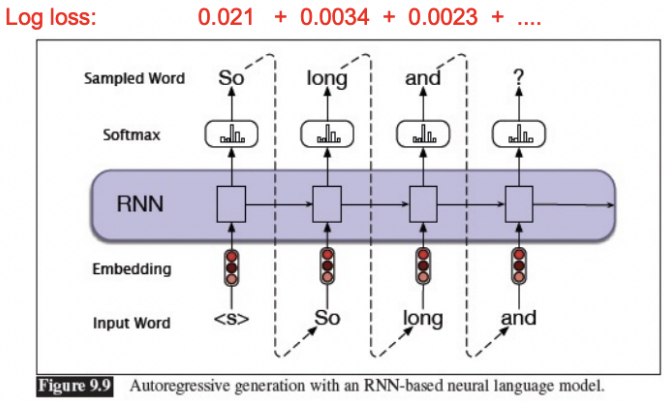
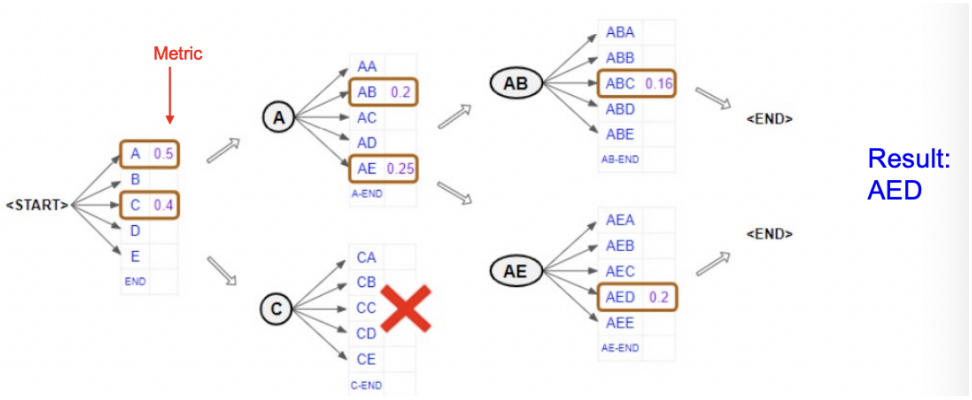
    c. Recall: A Language Model assigns a probability to each sequence of words. To teach an RNN a language model, we can add the log loss of each word generated compared with an N-Gram model:
    d. 

Figure 9.9 Autoregressive generation with an RNN-based neural language model.
    e. The back propagation is difficult and takes a lot of time since if the first word is wrong, it will also affect the word in the next sequence
    f. The context vector stores something about the previous word
    g. One Problem: The RNN makes local decisions about the most likely next word. However, a series of such local decisions will not necessarily find the globally most likely sentence (cf. gradient descent, which has the same problem).
    h. Similar to N-gram, we do not always want to find the most likely next word since it always generates only one sentence if we do that

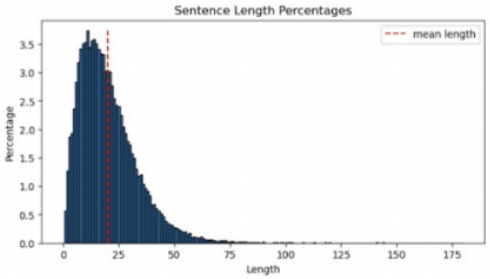
i. The usual optimization is Beam Search:
    i. Pick the “width of the beam” N (at each iteration, we will store the N most likely sequences of words); → choose N next words and some of the words will stay or die
    ii. Pick the expansion factor M (each sequence is extended with M new next words);
    iii. Choose some “goodness” metric (which sequences are better, e.g., perplexity);
    iv. Start with <s>
    v. At each iteration, extend each sequence M times using the generative model; if a sequence ending in </s> is generated, remove it and store among finished sentences.
    vi. Order the list in descending order of “goodness”; delete all but best N sequences; (it is competition and we delete sentences)
    vii. Repeat until some maximum length is reached or some other criterion is satisfied.

j. Example of Bean Search with N = 2 and M = 5 using letters instead of words:



    i. Each round has 2 survivors and each round has 5 choices
    ii. Choose or keep only the 2 highest probability words/letters and extend them (drop the rest)
    iii. Therefore, you only expand the N words

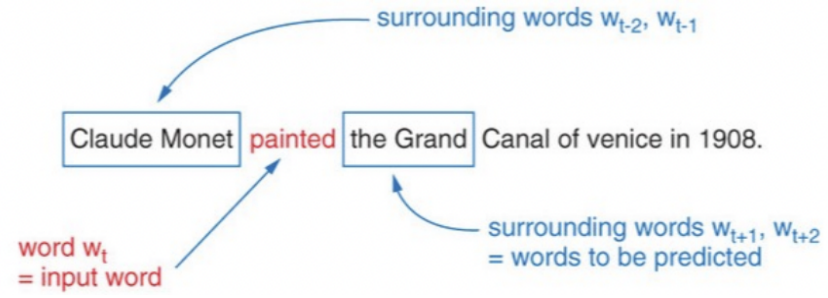
k. The ”goodness” metric in beam search can be almost anything, such as a weighted mean of
    i. Perplexity: How likely is the grammar of this sentence, ignoring length?
    ii. Length: How likely is the length of this sentence?
    iii. Meaning: Is this sentence expressing what I want it to express?
    iv. Etc.

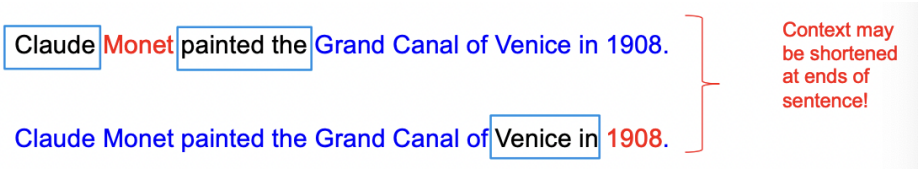
v. 

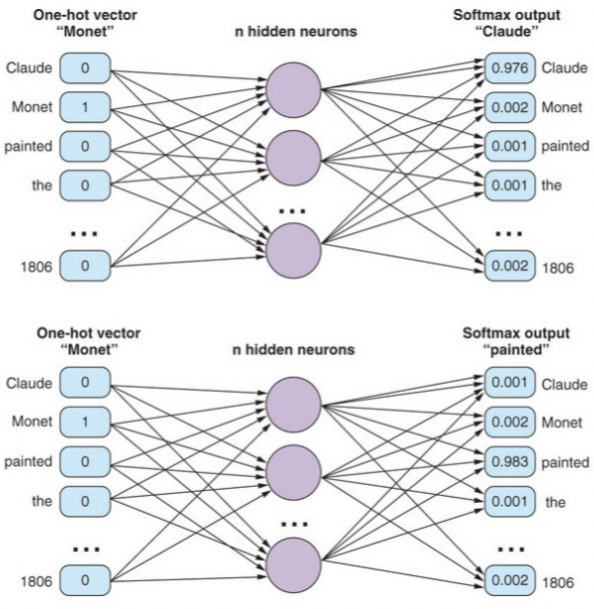
vi. Punchline: Beam search is not guaranteed to find the optimal sequence, but as a heuristic it works very well. There is an obvious efficiency/performance tradeoff. Appropriate Goodness Metrics are crucial.

vii. It is a broader search (adds breadth search into depth search)

3. Word Embeddings
    a. Sparse versus dense vectors
    b. TF or TF-IDF vectors are
        i. long (length |V|= 20,000 to 50,000) and sparse (most elements are 0)
    c. Alternative: learn vectors which are
        i. short (length 50-1000) and dense (most elements are non-zero)
    d. Why dense vectors?
        i. Short vectors may be easier to use as features in machine learning (fewer weights to tune)
        ii. Dense vectors may generalize better than explicit counts
        iii. Dense vectors may do better at capturing synonymy:
            1. car and automobile are synonyms; but are distinct dimensions
                a. “hood” and “headlight” should be similar, since both occur near both “car” and “automobile” but they aren’t in sparse vectors!
        iv. In practice, they work better!
    e. There are two general classes of word embeddings:
        i. Static Embeddings inspired by NN language models
            1. Word2vec (skip-gram, continuous bag of words), GloVe, fastText
            2. Each word has a unique embedding computed from co-occurrence statistics
            3. Problems: Can not deal with polysemy (different meanings for same token, e.g., “lead the way” vs “lead bullets”)
        ii. Contextual Embeddings
            1. ELMo, BERT
            2. Compute dynamic embeddings based on a word occurrence in its sentence
            3. Created by large language models using transformers
            4. Can deal with polysemy!

4. Word Embeddings with Word2Vec
    a. Word2Vec uses a NN prediction model to generate embeddings for a target word.
    b. There are two flavors, depending on what is being predicted:
        i. Skip-Grams: predict the context (co-occurring words) of a target word;
        ii. Continuous BOWs: Predict the target word from the context
5. Word Embeddings with Word2Vec: Skip-Grams
    a. Recall: A skip-gram is like an N-gram, except it has context on both sides of the target word.
    b. Here the window size is 5 (target word plus 2 words before and after):
    c. 

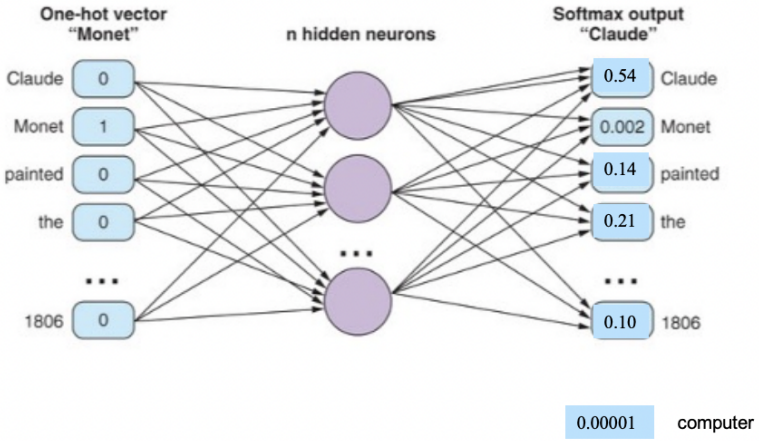
    d. 

    e. The task for the skip-gram approach is to predict the context given the target word, here “Monet”:
    f. 

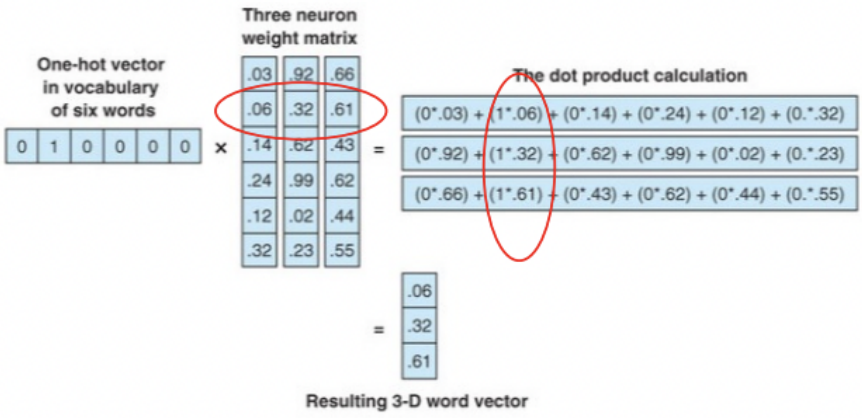
    g. The predictor is training on pairs of words (target-word, context-word)

Input	Output
Monet	Claude

h. After training, when you input the target word, you will get higher probabilities in the softmax output for all context words that the target word appears with in your corpus:

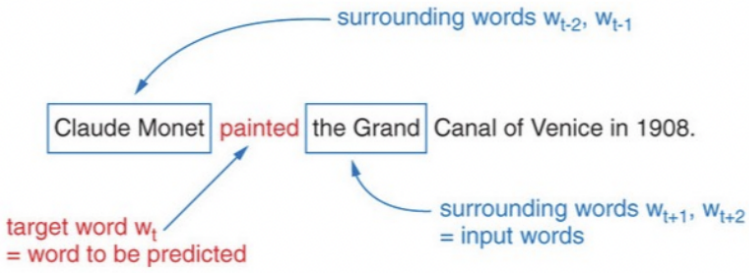


i. But here's the most important part of the algorithm: The embedding – the representation of the target word – is the weight vector inside the hidden layer corresponding to that input in a one-hot vector:

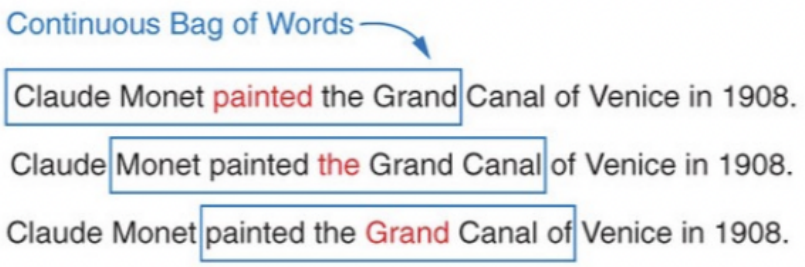


j. You can choose any size embedding you want, and that will determine the number of neurons in the hidden layer

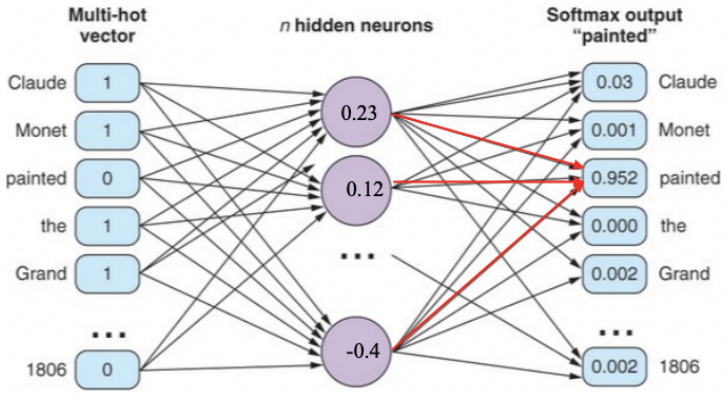
6. Word Embeddings with Word2Vec: CBOW
   a. In the Continuous Bag of Words approach, you are training the network to predict the target word using the context words,



b.



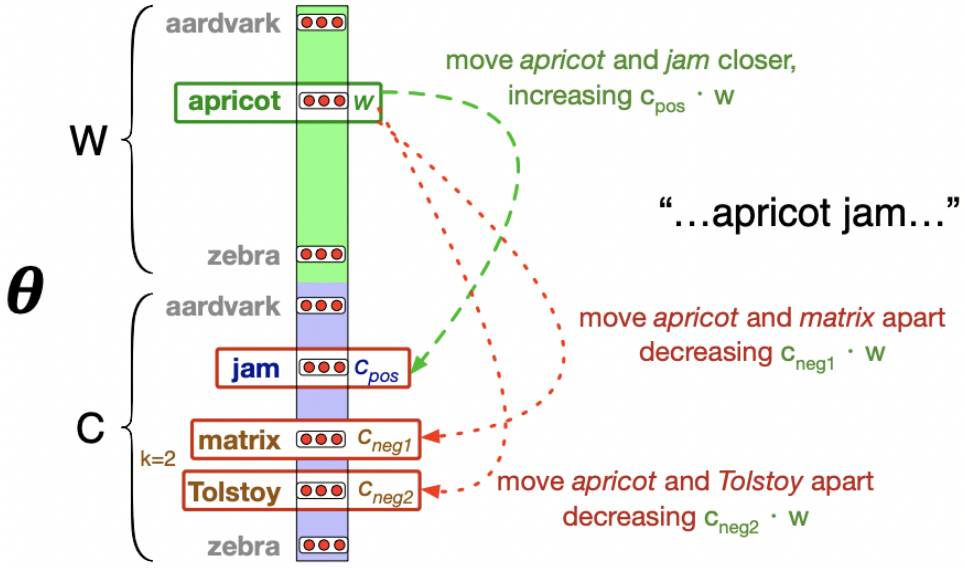
c. To train the network, you give the entire context as input in a multi-hot vector, and use softmax on output to predict the target word:



d. Now the embedding is the vector of weights which produced the target word:

[0.23, 0.12, …, -0.4]

e. In both approaches, we use gradient descent to move similar words closer together in the vector space, and dissimilar words farther apart:



f. Which is better?
    i. The originators of the approach showed that the skip-gram approach works well with small corpora and rare terms. You will have more training examples.
    ii. But CBOS shows higher accuracies for frequent words and is faster to train (because fewer examples).
    iii. Refinements:
        1. Fold frequent N-grams into unigrams: San Francisco -> San_Francisco
        2. Adjust sampling probability to the probability of words in the corpus (e.g., don't sample "the", "a", etc. as much as "computer" and "CBOW"
        3. Negative sampling: Use negative examples to train as well as positive