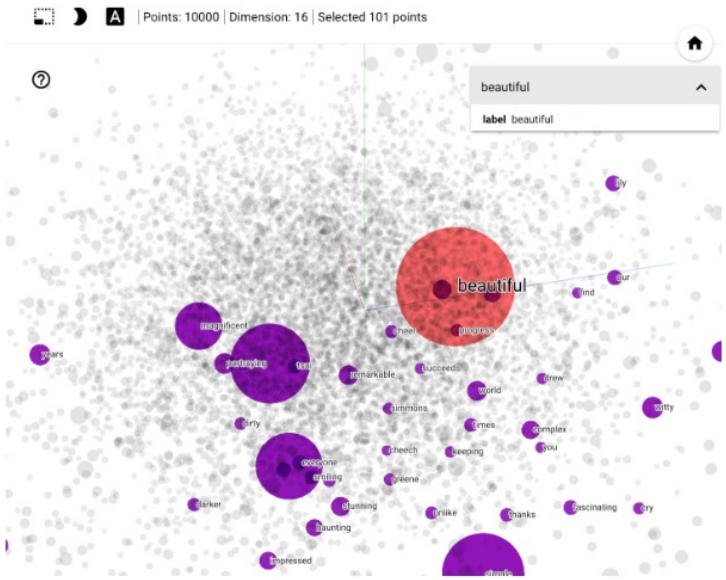
CS 505: Introduction to Natural Language Processing

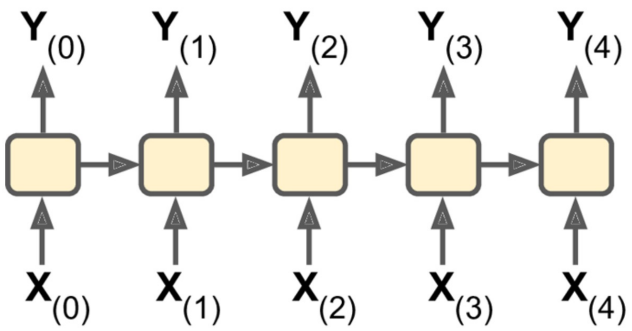
Wayne Snyder
Boston University

Lecture 14 – Generative Models with RNNs, Beam Search; Word and Document Embeddings

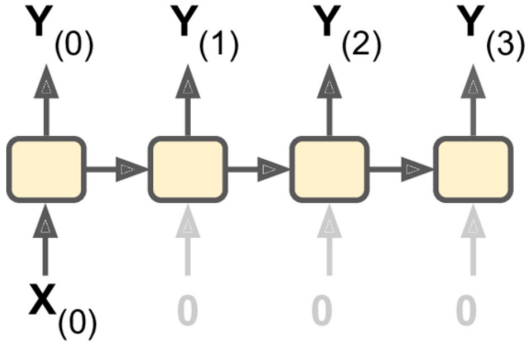


Review: RNN Architectures

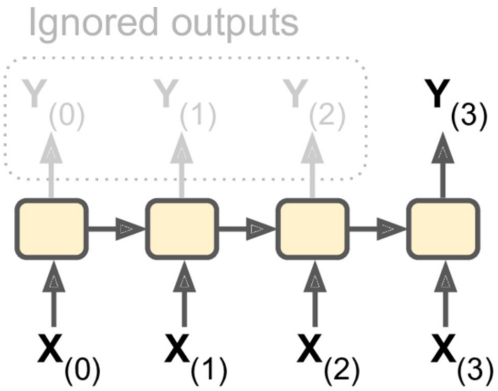
Sequence-to-Sequence



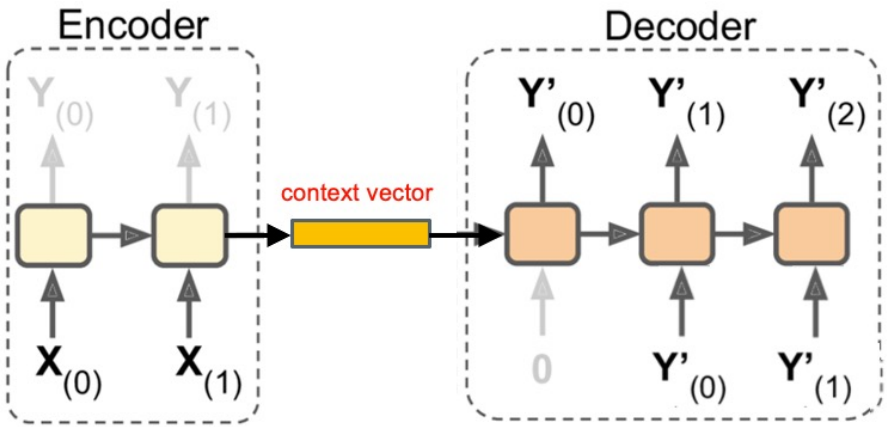
Vector-to-Sequence



Sequence-to-Vector:



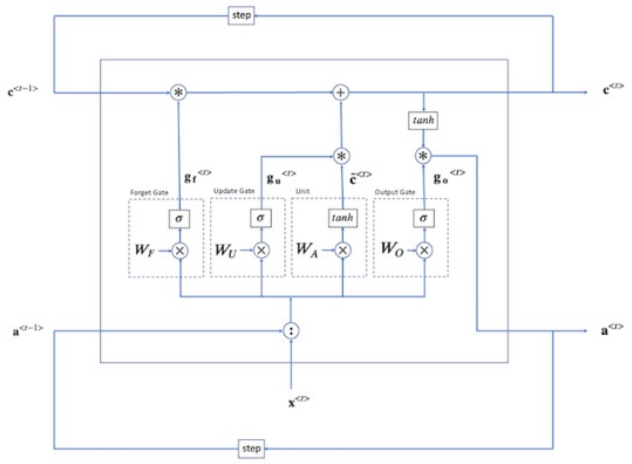
Encoder-Decoder Combination

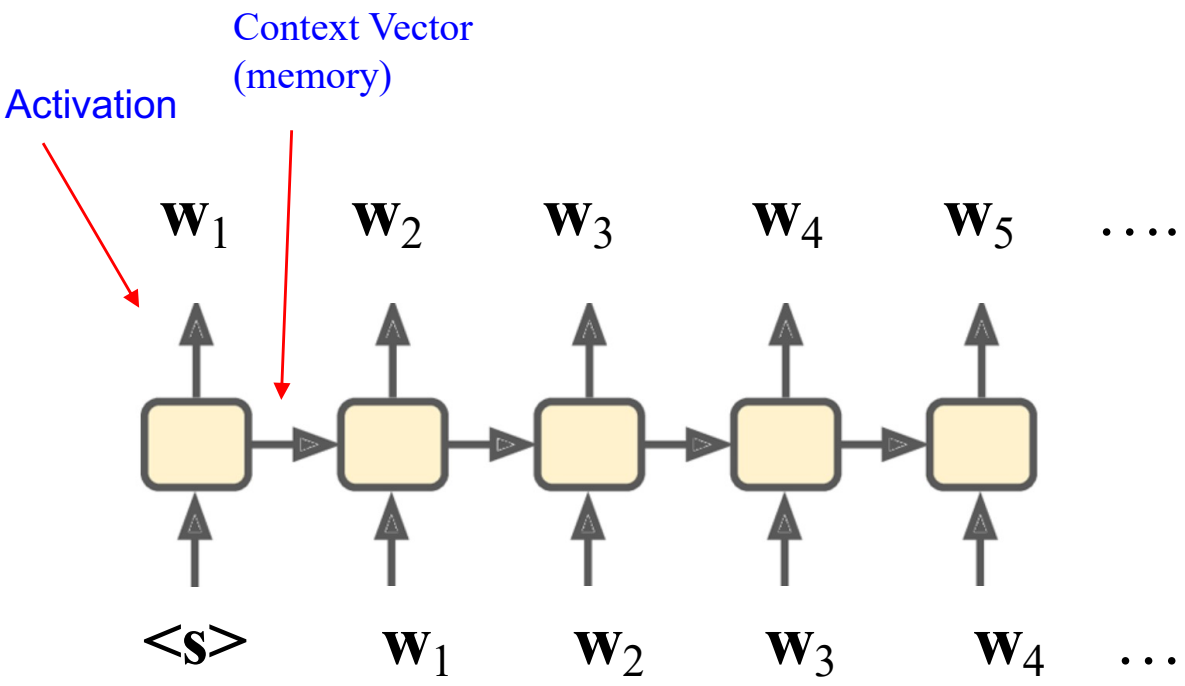


Generative Language Models with RNNs

The basic idea is to do a vector-to-sequence model, starting with the start-of-sentence token:

LSTM cell has a context vector and an activation:





Generative Language Models with RNNs

To train an RNN a language model, we create a dataset of subsequences of sentences:

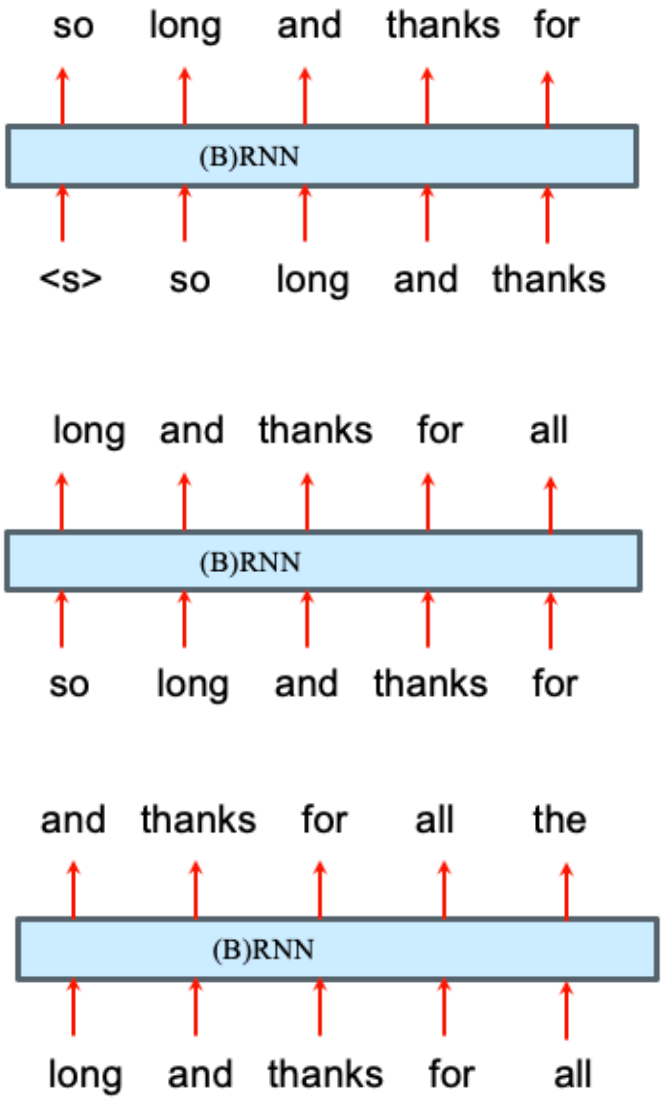
Sentence: <s> so long and thanks for all the fish ! </s>

Break into equal-length subsequences (here 5, but typically longer)

<s> so long and thanks
so long and thanks for
long and thanks for all
and thanks for all the
thanks for all the fish
for all the fish !
all the fish ! </s>

Applications of RNNs: Generative Language Models

Then we train the network on inputs and outputs:



Etc….

Applications of RNNs: Generative Language Models

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:

Log loss: 0.021 + 0.0034 + 0.0023 + ...

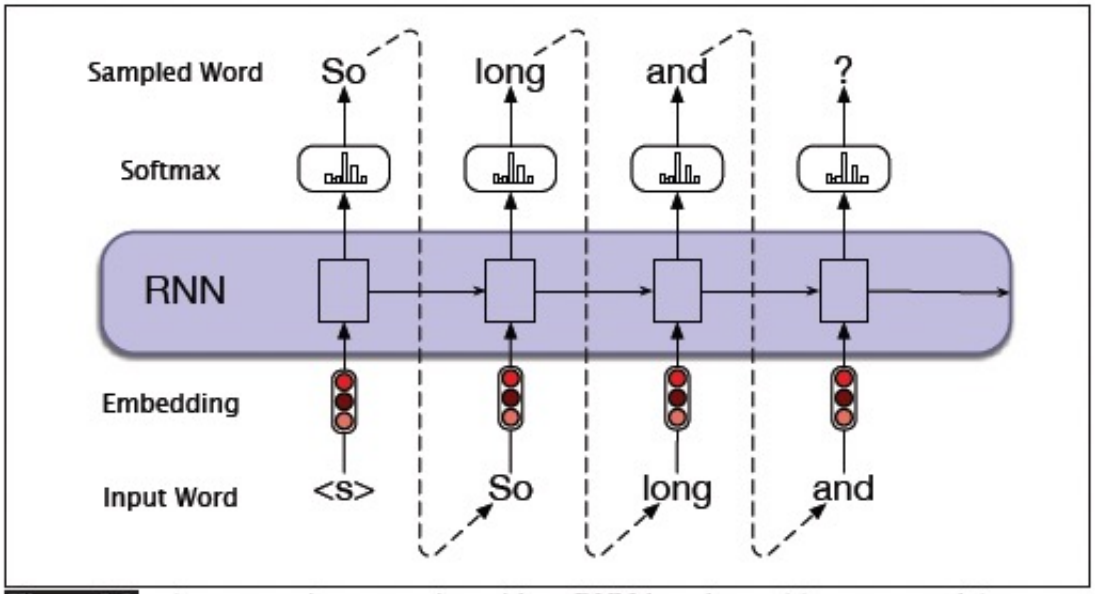


Figure 9.9 Autoregressive generation with an RNN-based neural language model.

Applications of RNNs: Generative Language Models

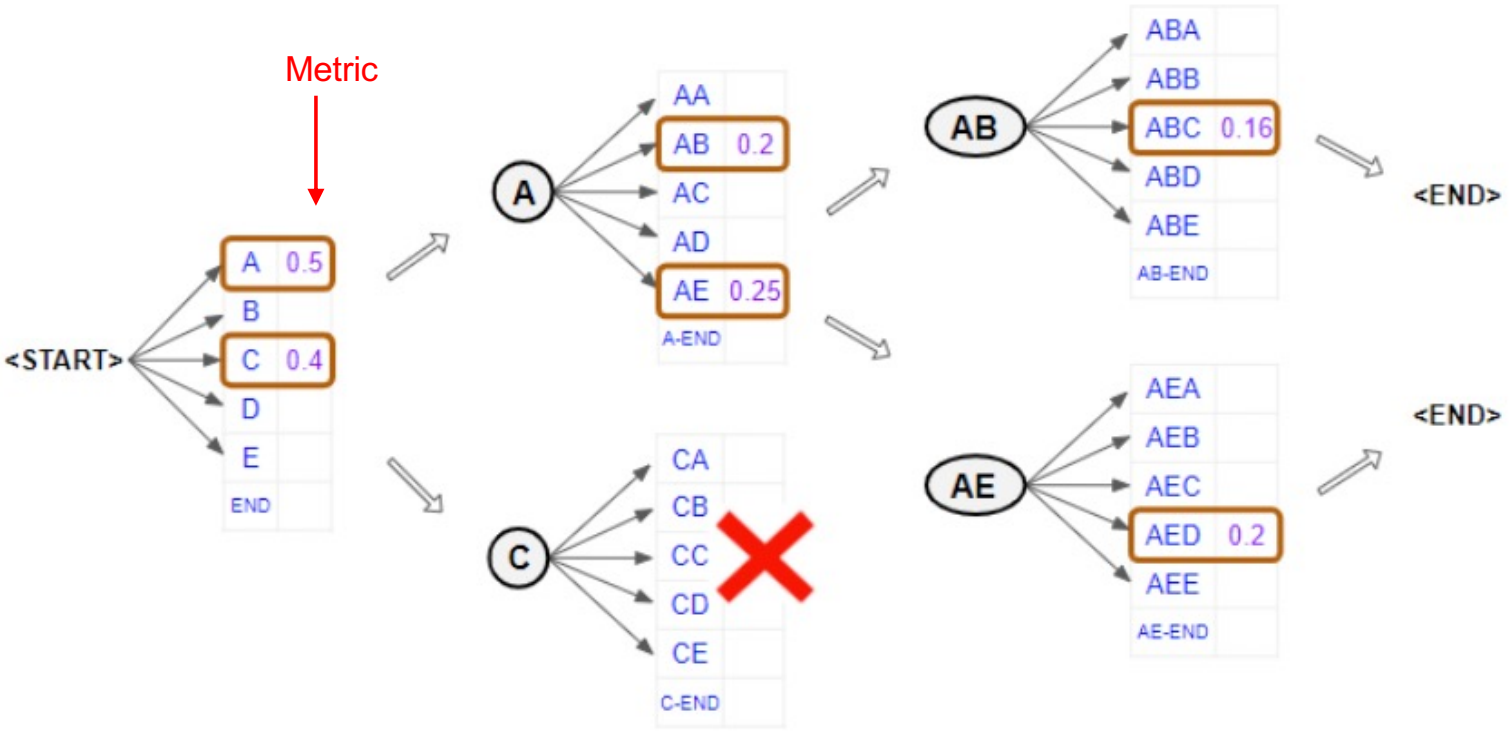
One Problem: The RNN makes local decisions about the most likely next word. However, a series of such local decisions will not necessarily find the globally most likely sentence (cf. gradient descent, which has the same problem).

The usual optimization is Beam Search:

1. Pick the "width of the beam" N (at each iteration, we will store the N most likely sequences of words);
2. Pick the expansion factor M (each sequence is extended with M new next words);
3. Choose some "goodness" metric (which sequences are better, e.g., perplexity);
4. Start with <s>;
5. 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.
6. Order the list in descending order of "goodness"; delete all but best N sequences;
7. Repeat until some maximum length is reached or some other criterion is satisfied.

Applications of RNNs: Generative Language Models

Example of Beam Search with N = 2 and M = 5
using letters instead of words:

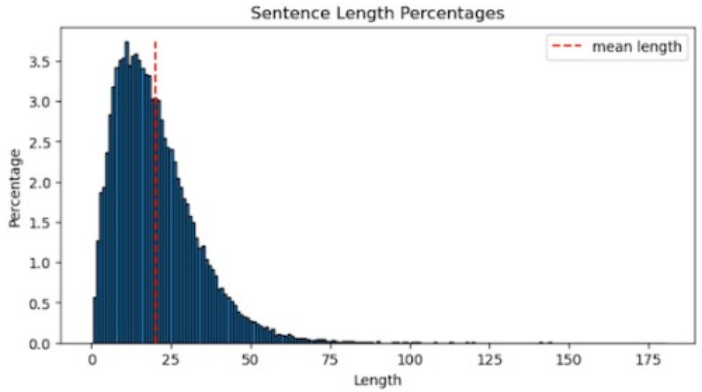


Result:
AED

Applications of RNNs: Generative Language Models

The ”goodness” metric in beam search can be almost anything, such as a weighted mean of

- Perplexity: How likely is the grammar of this sentence, ignoring length?
- Length: How likely is the length of this sentence?
- Meaning: Is this sentence expressing what I want it to express?
- Etc.



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.

Word Embeddings

Sparse versus dense vectors

- TF or TF-IDF vectors are
  - long (length |V|= 20,000 to 50,000) and sparse (most elements are 0)
- Alternative: learn vectors which are
  - short (length 50-1000) and dense (most elements are non-zero)
- **Why dense vectors?**
  - Short vectors may be easier to use as features in machine learning (fewer weights to tune)
  - Dense vectors may generalize better than explicit counts
  - Dense vectors may do better at capturing synonymy:
    - car and automobile are synonyms; but are distinct dimensions
      - "hood" and "headlight" should be similar, since both occur near both "car" and "automobile" but they aren't in sparse vectors!
  - In practice, they work better!

Word Embeddings

There are two general classes of word embeddings:

- Static Embeddings inspired by NN language models
  - Word2vec (skip-gram, continuous bag of words), GloVe, fastText
  - Each word has a unique embedding computed from co-occurrence statistics
  - Problems: Can not deal with polysemy (different meanings for same token, e.g., “lead the way” vs “lead bullets”)
- Contextual Embeddings
  - ELMo, BERT
  - Compute dynamic embeddings based on a word occurrence in its sentence
  - Created by large language models using transformers
  - Can deal with polysemy!

For now we’ll only discuss static embeddings in detail; contextual embeddings in about two weeks….

Word Embeddings with Word2Vec

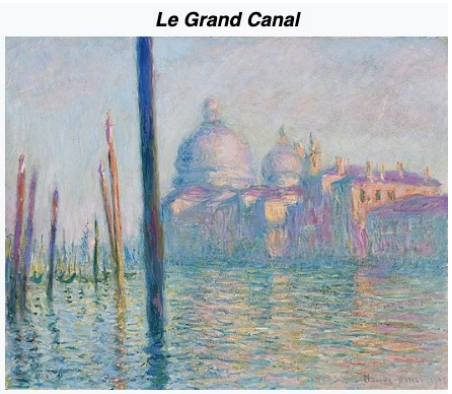
Word2Vec uses a NN prediction model to generate embeddings for a target word.

There are two flavors, depending on what is being predicted:

- **Skip-Grams:** predict the context (co-occurring words) of a target word;

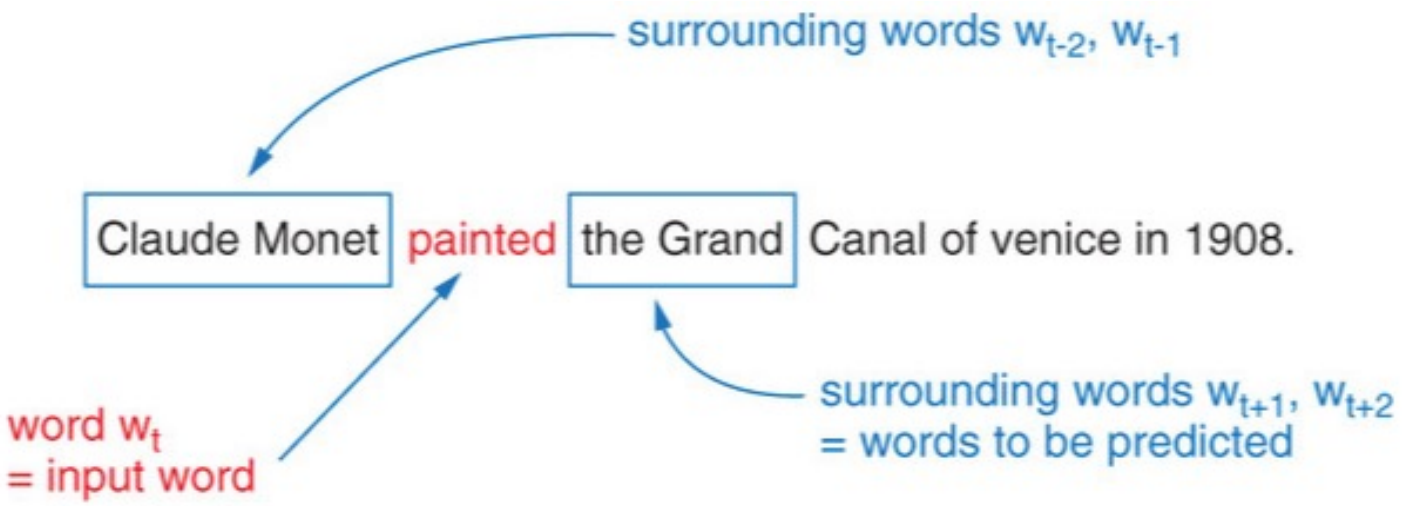
- **Continuous BOWs:** Predict the target word from the context

Word Embeddings with Word2Vec: Skip-Grams



Recall: A skip-gram is like an N-gram, except it has context on both sides of the target word.

Here the window size is 5 (target word plus 2 words before and after):



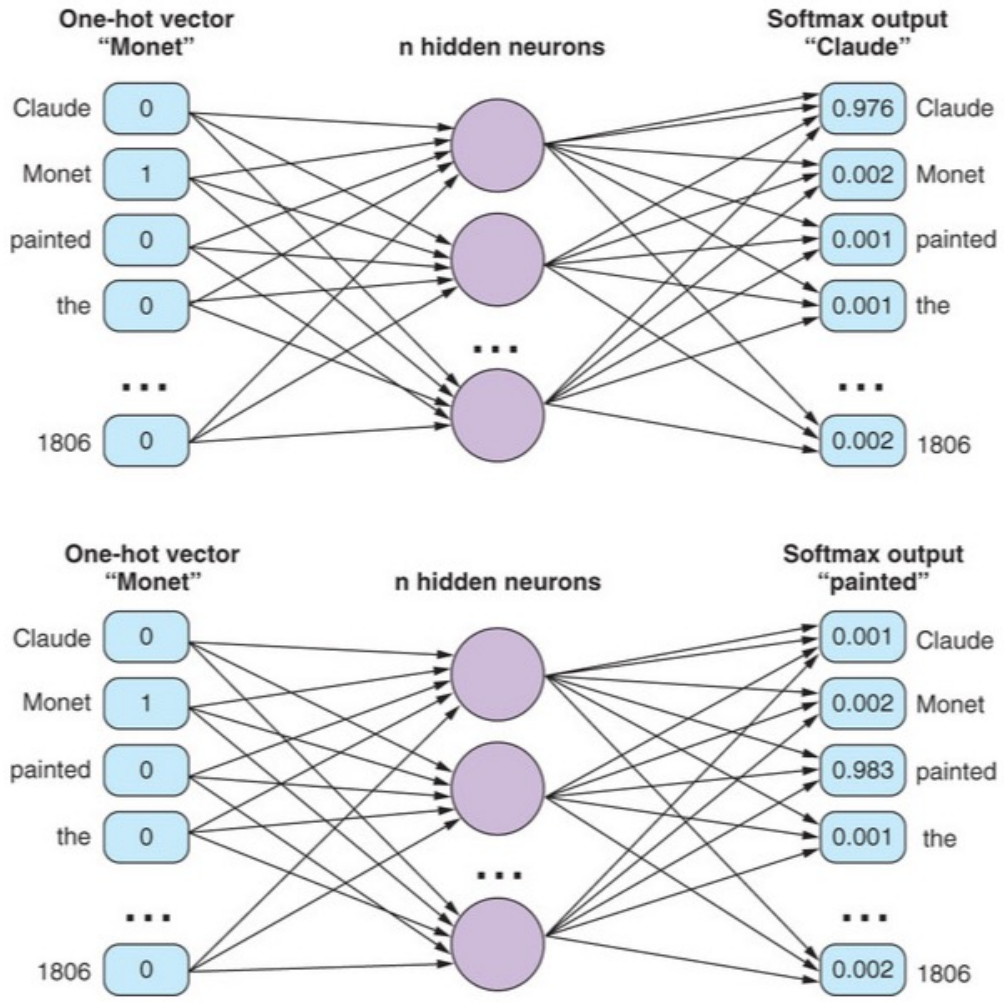
Claude Monet painted the Grand Canal of Venice in 1908.

Claude Monet painted the Grand Canal of Venice in 1908.

Context may be shortened at ends of sentence!

Word Embeddings with Word2Vec: Skip-Grams

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

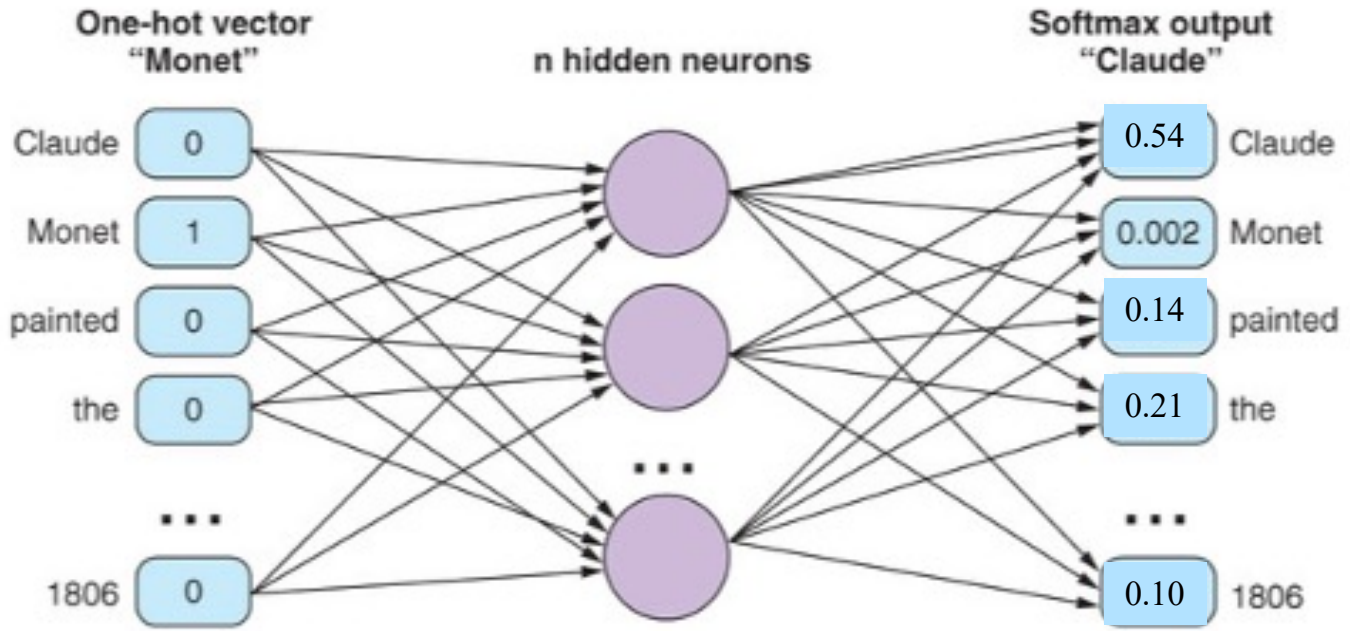


The predictor is training on pairs of words
(target-word, context-word):

Input	Output
Monet	Claude
Monet	painted

Word Embeddings with Word2Vec: Skip-Grams

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:

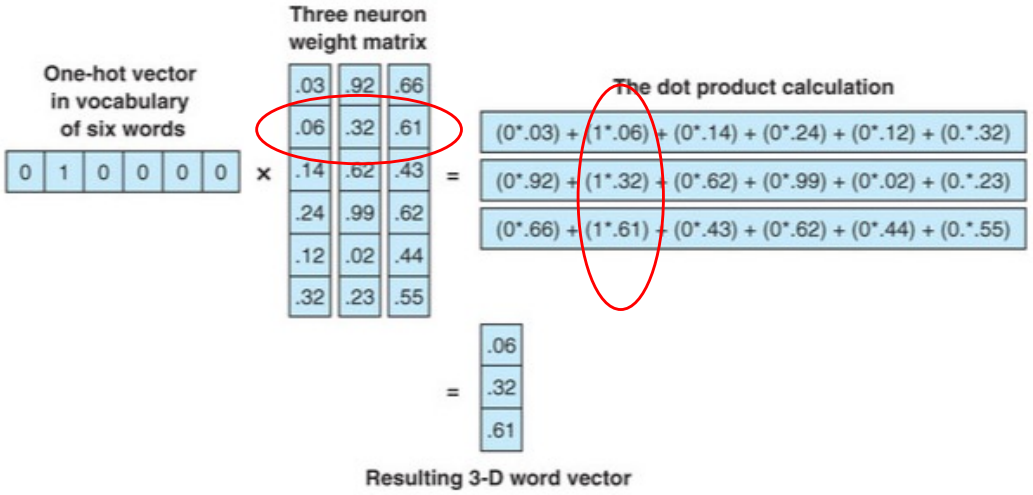


0.00001 computer

Word Embeddings with Word2Vec: Skip-Grams

But here's the most important part of the algorithm:

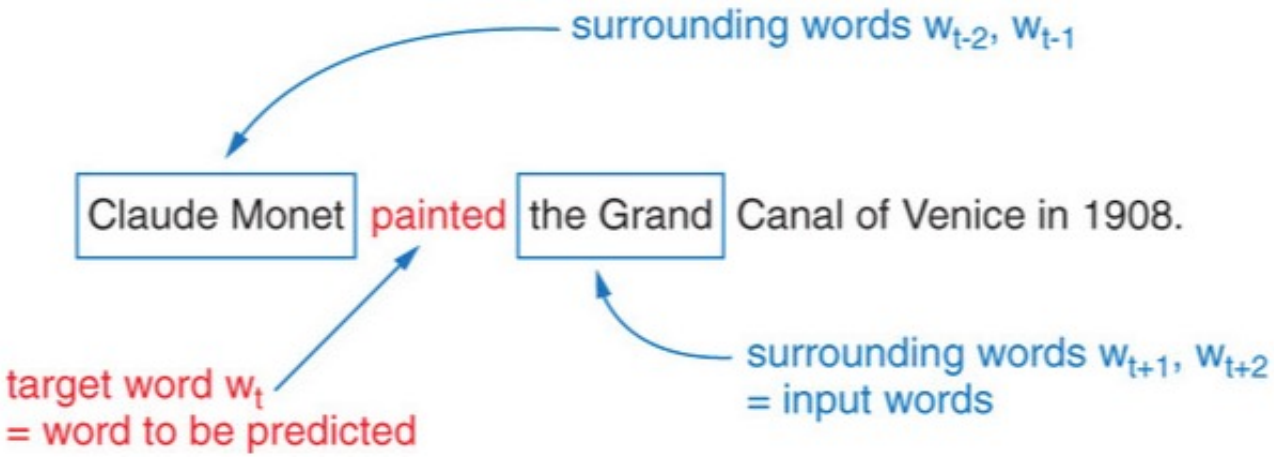
The embedding – the representation of the target word – is the weight vector inside the hidden layer corresponding to that input in a one-hot vector:

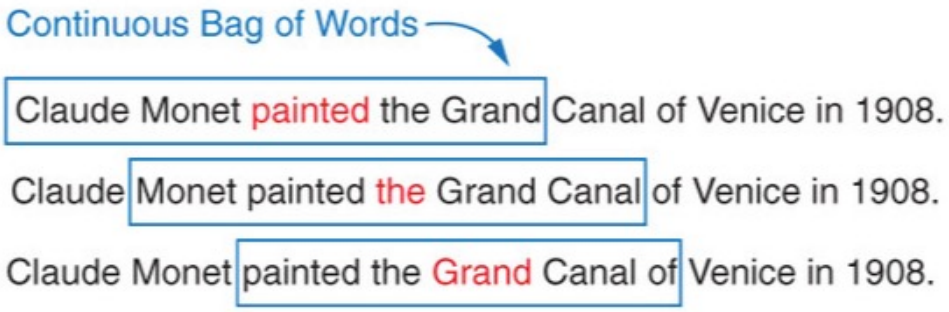


You can choose any size embedding you want, and that will determine the number of neurons in the hidden layer.

Word Embeddings with Word2Vec: CBOW

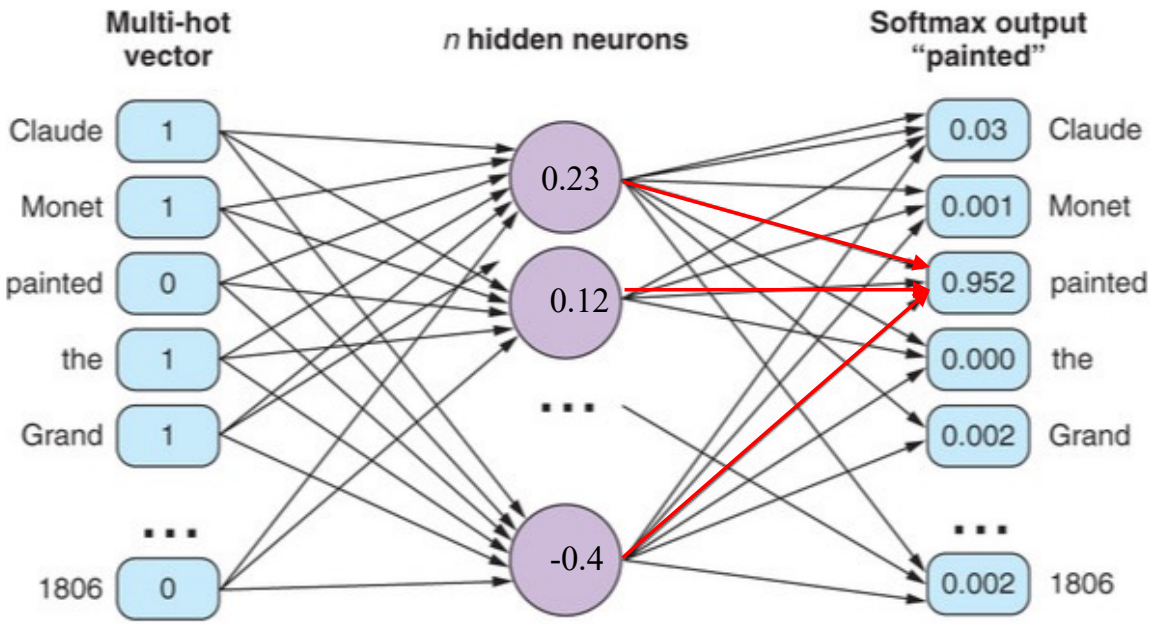
In the Continuous Bag of Words approach, you are training the network to predict the target word using the context words,





Word Embeddings with Word2Vec: CBOW

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:

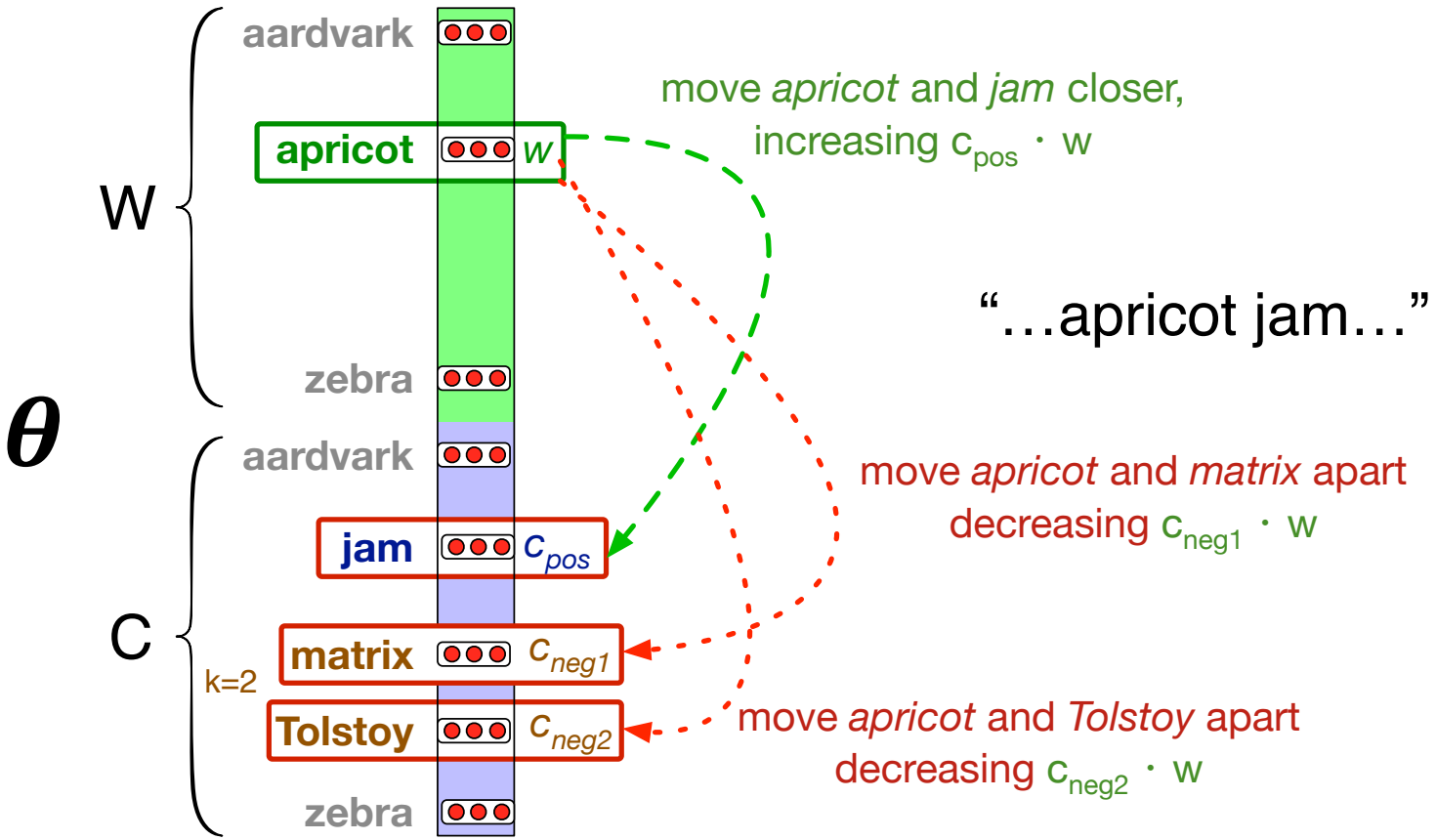


Now the embedding is the vector of weights which produced the target word:

[0.23, 0.12, …, -0.4]

Word Embeddings with Word2Vec: CBOW

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



Word Embeddings with Word2Vec: CBOW

Which is better?

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.

But CBOS shows higher accuracies for frequent words and is faster to train (because fewer examples).

Refinements:

- Fold frequent N-grams into unigrams: San Francisco -> San_Francisco
- 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"
- Negative sampling: Use negative examples to train as well as positive.

Word Embeddings with Word2Vec

Which is better?

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.

But CBOS shows higher accuracies for frequent words and is faster to train (because fewer examples).

Refinements:

- Fold frequent N-grams into unigrams: San Francisco -> San_Francisco
- 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"
- Negative sampling: Use negative examples to train as well as positive.