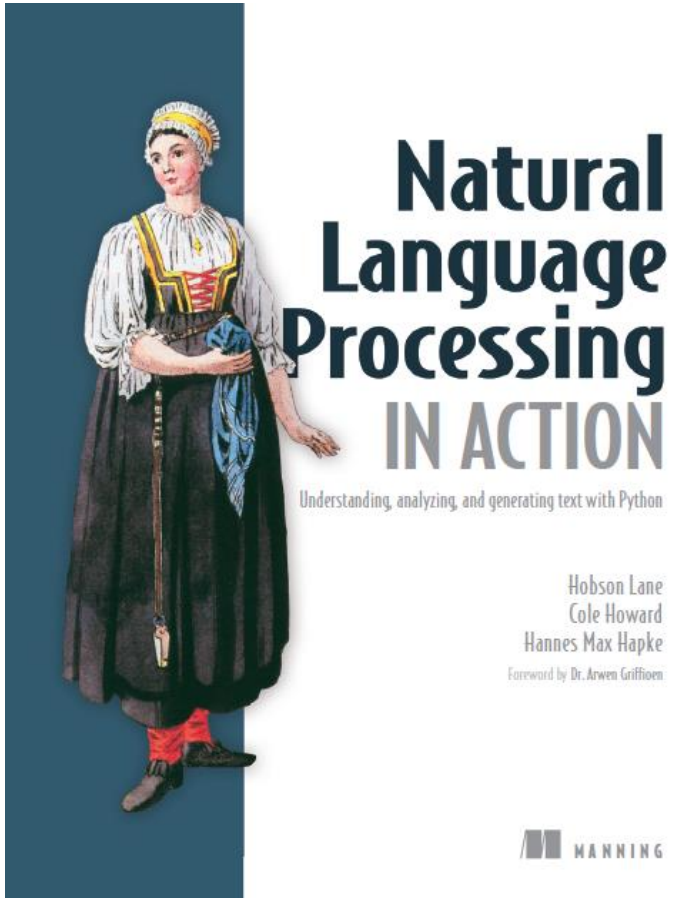
Reasoning with Word Vectors

Prof. Gheith Abandah

Reference 1

- Chapter 6: **Reasoning with Word Vectors (Word2vec)**



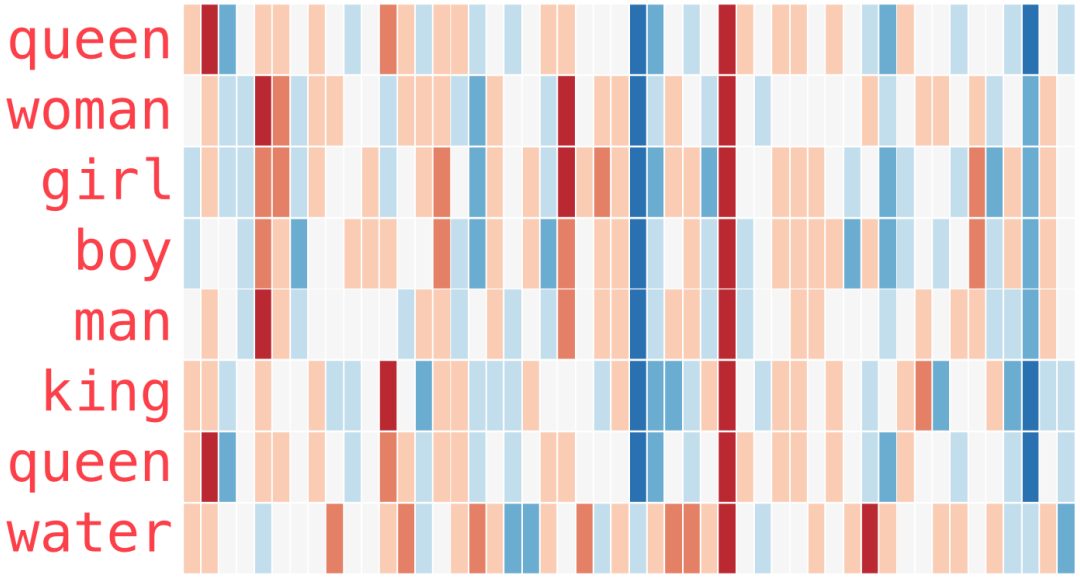
- H. Lane, C. Howard, and H. Hapke, **Natural Language Processing in Action:** Understanding, analyzing, and generating text with Python, Manning, 2019.

Outline

- Introduction
- Creating Word Vectors
  - Skip-gram
  - Continuous bag-of-words
  - GloVe
  - Custom word vectors
- Word Vector Libraries
- Example Code

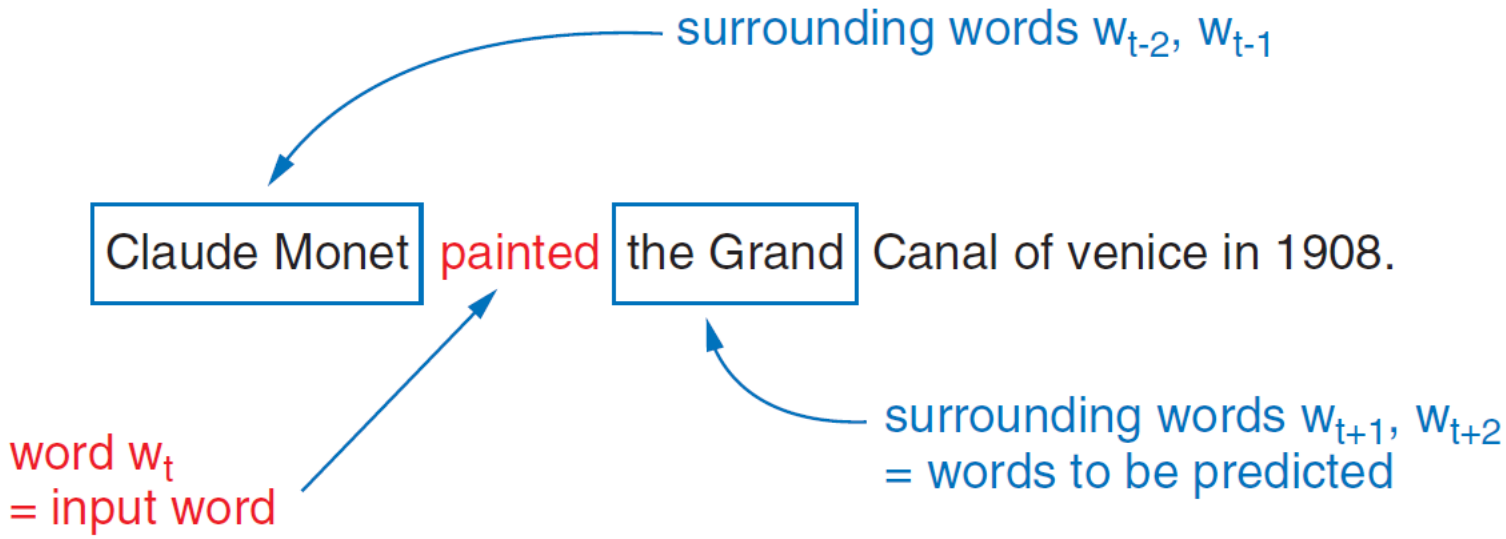
Introduction to Word Vectors (Word2vec)

- Representing words not just as symbols but as **numerical vectors** that **capture their meaning and relationships** with other words.
- **Word2vec** is a powerful technique for learning such word vectors.
- Facilitates capturing the semantic meaning of words, enabling **better performance** in NLP tasks.



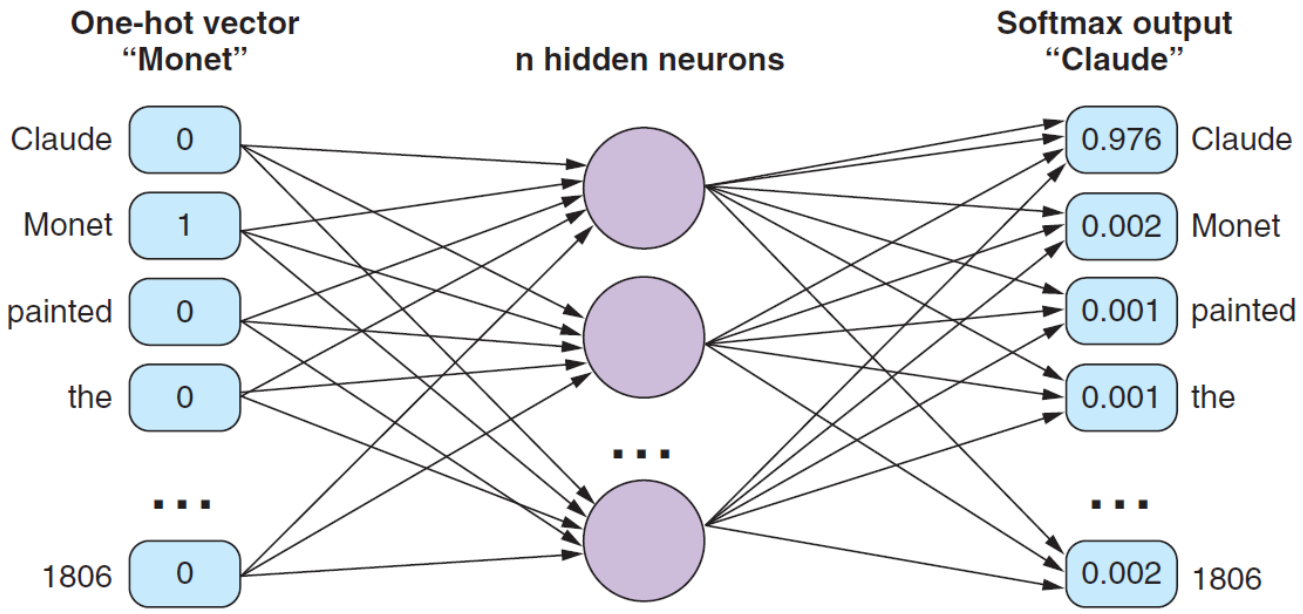
Word2vec Using Skip-gram

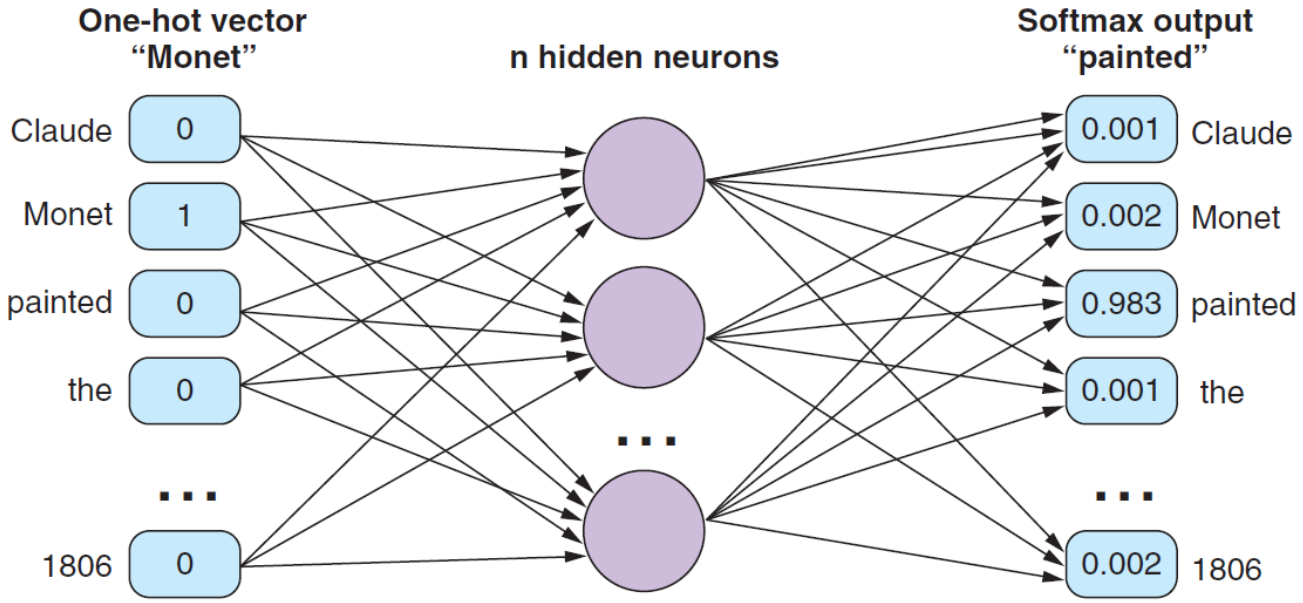
- **Predicts surrounding words** based on a center word.
- It aims to maximize the co-occurrence probability of surrounding words given a center word.
- Improves at capturing semantic relationships between words.



5-grams for “Claude Monet painted the Grand Canal of Venice in 1806.”

Input word w_t	Expected output w_{t-2}	Expected output w_{t-1}	Expected output w_{t+1}	Expected output w_{t+2}
Claude			Monet	painted
Monet		Claude	painted	the
painted	Claude	Monet	the	Grand
the	Monet	painted	Grand	Canal
Grand	painted	the	Canal	of
Canal	the	Grand	of	Venice
of	Grand	Canal	Venice	in
Venice	Canal	of	in	1908
in	of	Venice	1908	
1908	Venice	in		

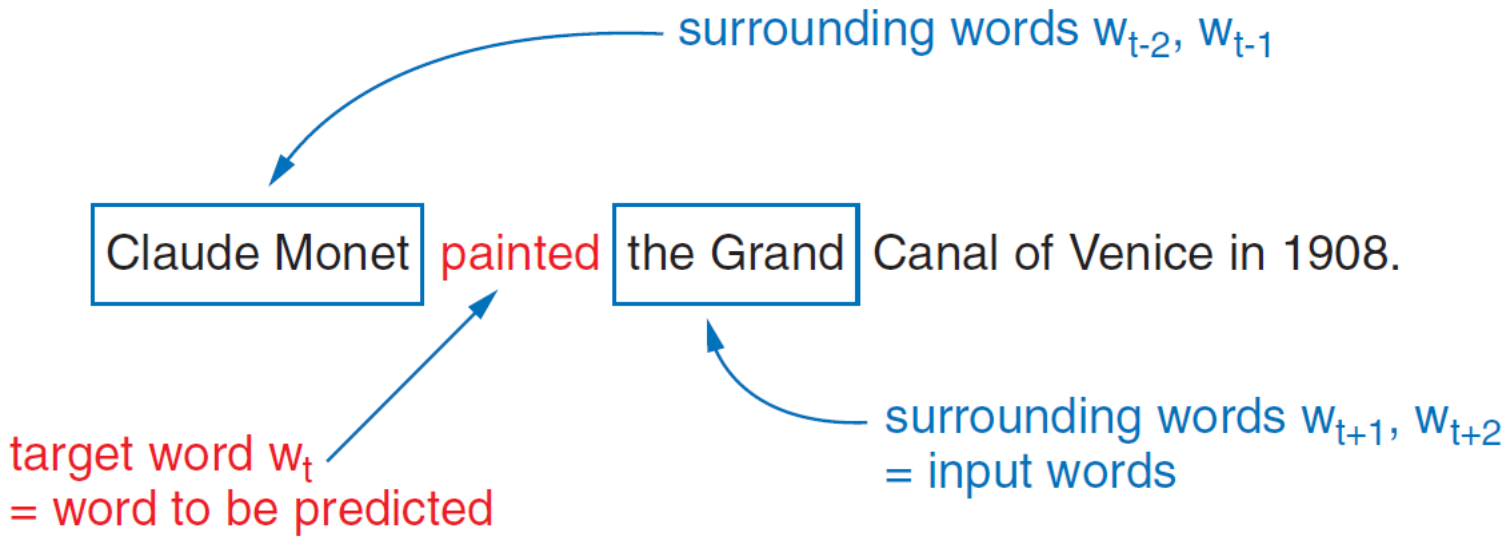




- Only the weights of the inputs to the hidden layer are used for word embedding.
- The **input weight matrix**:
  - **Rows** = number of words
  - **Columns** = number of hidden neurons

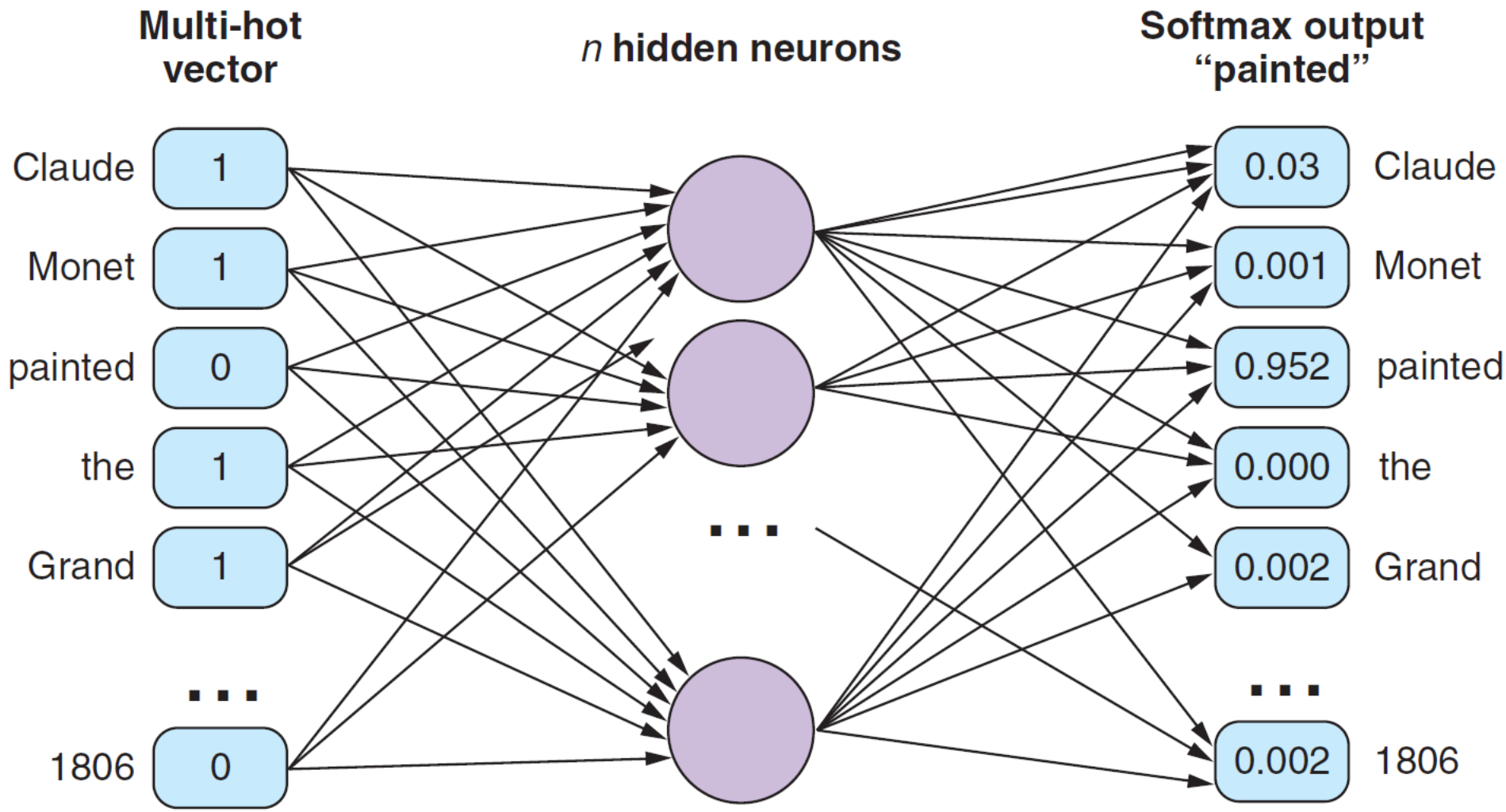
Word2vec using Continuous Bag-of-words

- CBOW model **predicts a center word based on its surrounding** words.
- It aims to maximize the probability of the center word given surrounding words.
- Often effective at capturing syntactic relationships between words.



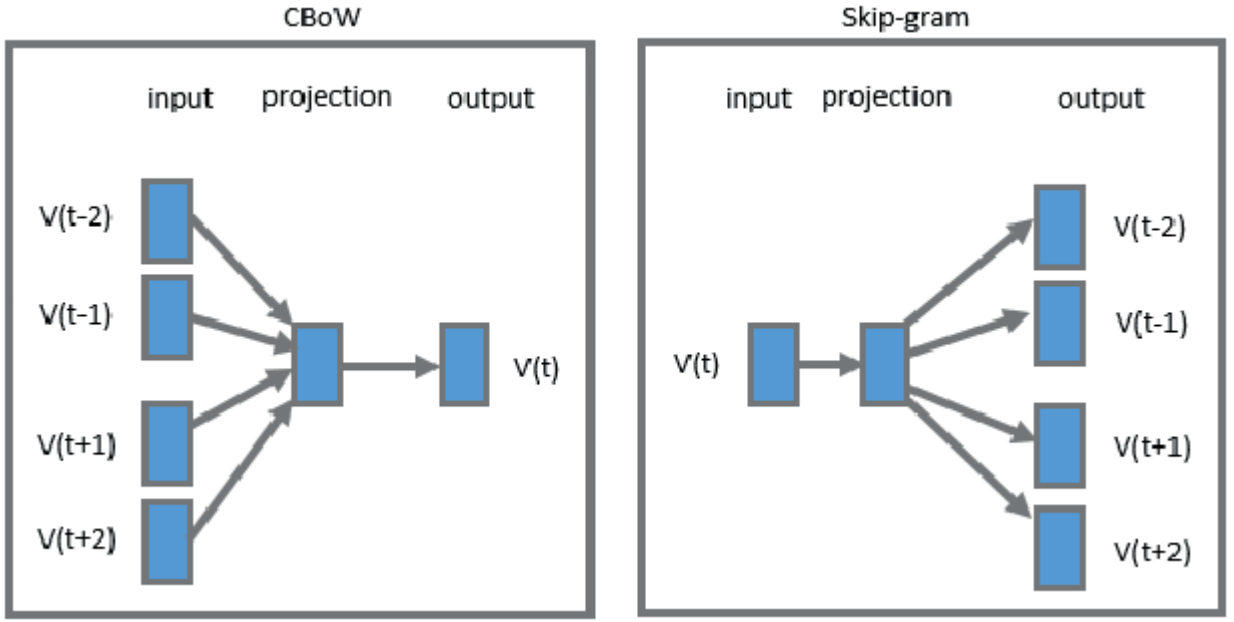
CBOW 5-grams for "Claude Monet painted the Grand Canal of Venice in 1806."

Input word w_{t-2}	Input word w_{t-1}	Input word w_{t+1}	Input word w_{t+2}	Expected output w_t
		Monet	painted	Claude
	Claude	painted	the	Monet
Claude	Monet	the	Grand	painted
Monet	painted	Grand	Canal	the
painted	the	Canal	of	Grand
the	Grand	of	Venice	Canal
Grand	Canal	Venice	in	of
Canal	of	in	1908	Venice
of	Venice	1908		in
Venice	in			1908


Multi-hot vector
n hidden neurons
Softmax output "painted"
Claude
Monet
painted
the
Grand
1806
1
1
0
1
1
0
0.03
0.001
0.952
0.000
0.002
0.002
Claude
Monet
painted
the
Grand
1806

Skip-Gram *vs*. CBOW

- **Skip-gram** approach works well with small corpora and rare terms.
- **CBOW** shows higher accuracies for frequent words and is much faster to train.



Computational Tricks of Word2vec

- **Frequent bigrams**: Treat frequent word pairs (bigrams) as single entities.
- **Subsampling frequent tokens**: Reducing the representation of high-frequency words to balance the training data.
- **Negative sampling**: Instead of updating all word weights, efficiently train against a small set of selected word pair samples.

Word Vectors using GloVe

- GloVe (**Global Vectors**) is another popular approach for word embeddings.
- Unlike word2vec, which focuses on local context, GloVe leverages statistical information from the entire corpus.
- It builds word vector representations based on **word co-occurrence statistics**.
- Computes the **singular value decomposition** (SVD) of the co-occurrence matrix, splitting it into the same two weight matrices that Word2vec produces.
- Pre-trained GloVe models are available for download.

$$W_{\text{mxn}} \Rightarrow U_{\text{mxp}}\ S_{\text{pxp}}\ V_{\text{pxn}}{}^{\text{T}}$$

Generating Your Own Word2vec

- You can train your own word2vec model on your **custom text corpus**.
- This allows you to tailor the word vectors to your **specific domain or application**.
- **Steps**
  - Preprocess your text data (split into sentences and words).
  - Choose appropriate hyperparameters for the word2vec model (vector size, window size *etc*.).
  - Train the model for enough time to ensure good quality embeddings.

Word2vec Libraries

1. **Gensim**: Known for its ease of use and efficiency, Gensim is a robust library that supports multiple word embedding models like Word2Vec, FastText, and Doc2Vec. It is optimized for memory efficiency and speed, making it suitable for handling large text corpora.
2. **FastText**: Developed by Facebook Research, FastText is specifically designed for efficient learning of word representations and sentence classification. It extends Word2Vec to consider subword information, making it more effective for languages with complex morphology. FastText comes with pre-trained models in 157 languages and is optimized for performance.
3. **spaCy**: Includes support for word embeddings. It offers pre-trained models for multiple languages and integrates seamlessly with other NLP tasks like tokenization, part-of-speech tagging, and named entity recognition. spaCy's models are optimized for speed and efficiency.
4. **Transformers by Hugging Face**: This library, although not exclusively focused on word embeddings, offers state-of-the-art transformer models like BERT, GPT, and their derivatives, which can be used for generating word embeddings. These models support multiple languages and are at the forefront of NLP research. The library is designed for ease of use and flexibility.

Gensim.word2vec Module

- Gensim is a popular Python library for NLP tasks.
- It provides a user-friendly implementation of word2vec through the gensim.word2vec module.
- You can **load pre-trained models**, **train on your own corpus**, and **perform various word similarity queries**.

```
from gensim.models import KeyedVectors

# Load a pre-trained word vector model
model_path = 'GoogleNews-vectors-negative300.bin.gz'
model = KeyedVectors.load_word2vec_format(model_path, binary=True,
                                          limit=3000000)
```

```
# Get the word embedding of a word
word_embedding = model['computer']

print(f"Embedding for 'computer':\n{word_embedding}")
```

```
Embedding for 'computer':
[ 1.07421875e-01 -2.01171875e-01  1.23046875e-01  2.11914062e-01
 -9.13085938e-02  2.16796875e-01 -1.31835938e-01  8.30078125e-02
 …
```

KING : MAN :: ? : WOMAN

```
# Find the most similar words
similar_words =
    model.most_similar(positive=['king', 'woman'],
                       negative=['man'], topn=5)

print("Words most similar to 'king' + 'woman' - 'man':")

for word, similarity in similar_words:
    print(f"{word}: {similarity}")
```

```
Words most similar to 'king' + 'woman' - 'man':

queen: 0.7118191123008728

monarch: 0.6189674735069275

princess: 0.5902430415153503

crown_prince: 0.5499458909034729

prince: 0.5377322435379028
```

```
# Find unrelated terms
unrelated_terms = model.doesnt_match(
    "breakfast cereal dinner lunch".split())
print(f"Term not related to the others:
{unrelated_terms}")
```

```
Term not related to the others: cereal
```

```
# Compute word similarity
similarity_score = model.similarity('woman', 'man’)
print(f"Similarity between 'woman' and 'man':
{similarity_score}")
```

```
Similarity between 'woman' and 'man': 0.7664012908935547
```

Summary

- You've learned how word vectors and vector-oriented reasoning can solve some surprisingly subtle problems like analogy questions.
- You can train Word2vec and other word vector embeddings on the words you use in your applications.
- Use gensim to explore, visualize, and even build your own word vector vocabularies.
- If you respect sentence boundaries with your n-grams and are efficient at setting up word pairs for training, you can greatly improve the accuracy of your latent semantic analysis word embeddings.