Session-1

Background for LLM: Word as Vectors

These slides are adapted from the course materials of **"Speech and Language Processing (3rd Edition, Draft)"** by **Dan Jurafsky and James H. Martin**, available at:
https://web.stanford.edu/~jurafsky/slp3/.

I deeply appreciate the authors for making these materials accessible to the academic community.

Outline

- NLP: Introduction and Applications
- Challenges in Representation of Words
- WordNet
- Word2Vec
- Stochastic Gradient Descent
- The Skip-gram Model with Negative Sampling
- Demonstration

What is NLP

Natural language processing is a field of science and engineering focused on the development and study of **automatic systems** that **understand** and **generate natural (that is, human,) languages**.

Language and Machines

- A key challenge in building language-learning machines is **how do we represent words**?.
- **Deep learning** is a powerful tool for representing natural language.
- We will focus on **the representation of words** in a computer.
- The **goal** is to enable learners to **build applications using modern NLP techniques**.

A few Uses of NLP

Machine Translation:

- NLP is commonly used for translating languages.
- Challenges include handling many languages and maintaining context accuracy.

Question Answering and Information Retrieval:

- NLP is used to answer questions and assist with information retrieval.
- Research is expanding its capabilities to handle more questions and enable interactive dialogue.

Summarization and Text Analysis:

- NLP is used to summarize and analyze text.
- It helps with market research, public opinion analysis, and simplifying complex topics.
- NLP tools support information access and surveillance.

A few Uses of NLP (Cont’d)

Generative AI

- Generative AI autonomously creates new content.
- In text generation, NLP is crucial for understanding language semantics, syntax, and context.
- Notable models like OpenAI's GPT showcase the power of generative AI.

Grammar checkers

- NLP techniques examine sentence structure, syntax, and grammar.
- They fix errors like wrong verbs, punctuation, and incomplete sentences.
- Advanced tools use context and style for better suggestions.
  This improves the clarity and quality of writing.

Challenge: Representation of Words

- See the sentence: **Zuko makes the tea for his uncle.**
- The word **Zuko** is a sign, a **symbol** that represents an entity Zuko in some (real or imagined) world.
- The word **tea** is also a **symbol** that refers to a signified thing–perhaps a specific instance of tea.
- If one were instead to say: **Zuko likes to make tea for his uncle.**
- note that the symbol ***Zuko*** still refers to ***Zuko***, but now **tea** refers to a broader class—tea in general.
- Now Consider the two following sentences:
  - Zuko makes the **coffee** for his uncle.
  - Zuko makes the **drink** for his uncle.
- Which is "more like" the sentence about tea?
- The ***drink*** may be tea (or it may be quite different!) and ***coffee*** definitely isn't tea, but is yet similar, no?
- Is ***Zuko*** similar to ***uncle*** because they both describe people?
- Is ***the*** similar to ***his*** because they both pick out specific instances of a class?

Challenge: Representation of Words

- **Words** capture the **subtleties** and **complexities** of language.
- **Language** balances **rich expression** with **effective information transfer**.
- **Speech** is continuous, but language uses **discrete symbols** for clarity.
- The **challenge** lies in fully **expressing language** while ensuring **efficiency**.
- Representing **words** is a key challenge in **linguistics** and **computation**.

How do we represent the meaning of a word?

Commonest linguistic way of thinking of meaning:

signifier (symbol) ⇔ **signified** (idea or thing)

tree ⇔ {🌳, 🌲, 🌴, …}

WordNet

- How do we have usable meaning in a computer?
- Previously commonest NLP solution: Use, e.g., **WordNet**, a thesaurus containing lists of **synonym** sets and **hypernyms** (“is a” relationships)
- It is a large lexical database that has been widely used in various NLP tasks and computational linguistics research.

WordNet

```
import nltk

nltk.download('wordnet')

from nltk.corpus import wordnet as wn

poses = { 'n':'noun', 'v':'verb', 's':'adj (s)', 'a':'adj', 'r':'adv'}

for synset in wn.synsets("bad"):

    print("{}: {}".format(poses[synset.pos()], ", ".join([l.name() for l in synset.lemmas()])))
```

```
noun: bad, badness
adj: bad
adj (s): bad, big
adj (s): bad, tough
adj (s): bad, spoiled, spoilt
adj: regretful, sorry, bad
adj (s): bad, uncollectible
adj (s): bad
adj (s): bad
adj (s): bad, risky, high-risk,
speculative
adj (s): bad, unfit, unsound
adj (s): bad
adj (s): bad
adj (s): bad, forged
adj (s): bad, defective
adv: badly, bad
adv: badly, bad
```

Problems with resources like WordNet

- A useful resource but **missing nuance**:
  - e.g., “**proficient**” is listed as a **synonym** for “**good**” This is only correct in some contexts
  - Also, WordNet list offensive synonyms in some synonym sets without any coverage of the connotations or appropriateness of words
- Missing new meanings of words:
  - e.g., wicked, nifty, wizard, genius, ninja
  - Impossible to keep up-to-date!
- Subjective
  - Requires human labor to create and adapt
  - Can’t be used to accurately compute word similarity

Representing words as discrete symbols

In traditional NLP, we regard words as discrete symbols:

hotel, conference, motel – a localist representation

Such symbols for words can be represented by one-hot vectors:

motel = [0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]

hotel = [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]

Vector dimension = number of words in vocabulary (e.g., 500,000+)

Problem with Words as Discrete Symbols

- Example: in web search, if a user searches for “motel”, we would like to match documents containing “hotel”
- But:

motel = [0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]

hotel = [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]

- These two vectors are orthogonal
- There is no natural notion of similarity for one-hot vectors!
- Solution:
  - Could try to rely on WordNet’s list of synonyms to get similarity?
  - But it is well-known to fail badly: incompleteness, etc.
  - Instead: learn to encode similarity in the vectors themselves

Representing words by their context

- **Distributional semantics**: **A word's meaning is given by the words that frequently appear close-by**
    - "***You shall know a word by the company it keeps***" (**J. R. Firth 1957: 11**)
    - One of the most successful ideas of modern statistical NLP!
- When a word w appears in a text, its context is the set of words that appear nearby (within a fixed-size window).
- We use the many contexts of w to build up a representation of w

…government debt problems turning into banking crises as happened in 2009…
…saying that Europe needs unified banking regulation to replace the hodgepodge…
…India has just given its banking system a shot in the arm…

These context words will represent banking

Word Vectors

- We will build a dense vector for each word, chosen so that it is similar to vectors of words that appear in similar contexts, measuring similarity as the vector dot (scalar) product

$$\text{banking} = \begin{pmatrix} 0.286 \\ 0.792 \\ -0.177 \\ -0.107 \\ 0.109 \\ -0.542 \\ 0.349 \\ 0.271 \end{pmatrix} \qquad \text{monetary} = \begin{pmatrix} 0.413 \\ 0.582 \\ -0.007 \\ 0.247 \\ 0.216 \\ -0.718 \\ 0.147 \\ 0.051 \end{pmatrix}$$

word vectors are also called **(word) embeddings** or **(neural) word representations** They are a distributed representation

Word Meaning as a Neural Word Vector – Visualization

$$\text{expect} = \begin{pmatrix} 0.286 \\ 0.792 \\ -0.177 \\ -0.107 \\ 0.109 \\ -0.542 \\ 0.349 \\ 0.271 \\ 0.487 \end{pmatrix}$$

Word2Vec

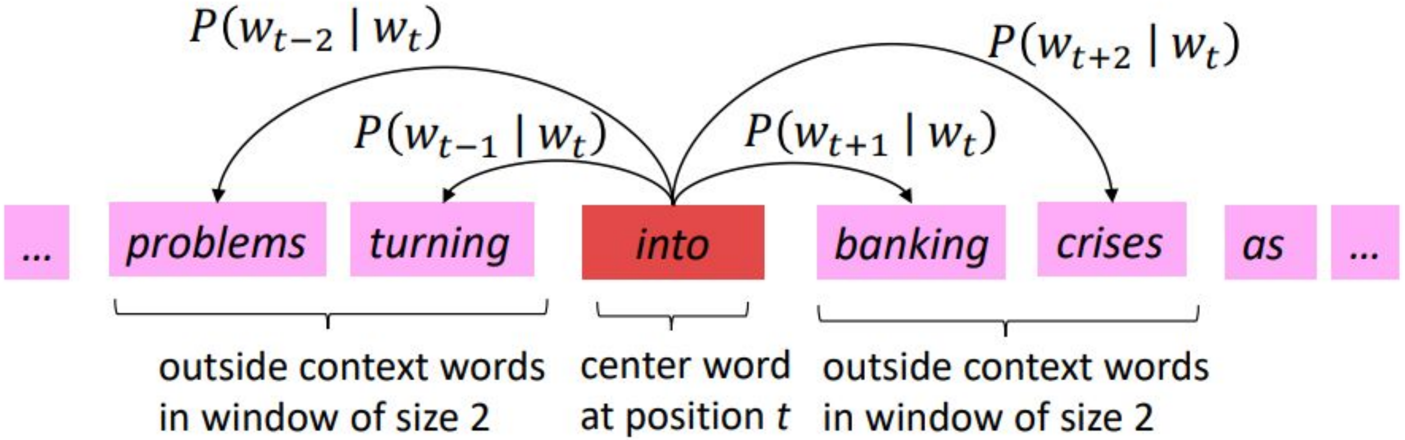
Word2vec (Mikolov et al. 2013) is a framework for learning word vectors

Idea:

- We have a large corpus ("body") of text: a long list of words
- Every word in a fixed vocabulary is represented by a vector
- Go through each position t in the text, which has a **center word** **c** and **context ("outside") words** **o**
- Use the similarity of the word vectors for **c** and **o** to calculate the probability of **o** given **c** (or vice versa)
- Keep adjusting the word vectors to maximize this probability

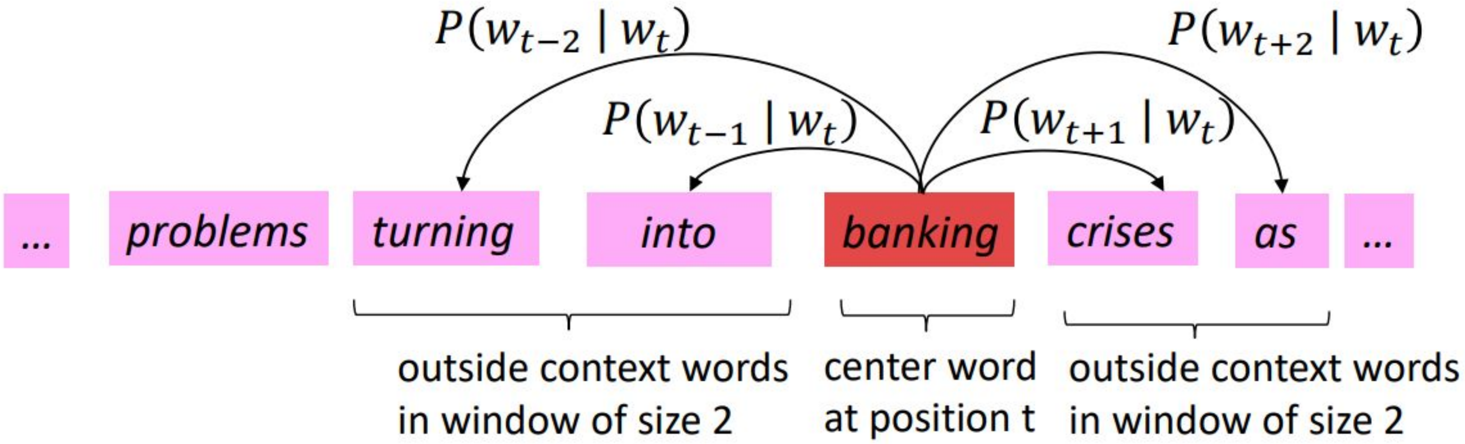
Word2Vec

Example windows and process for computing $P(w_{t+j} | w_t)$



Word2Vec

Example windows and process for computing $P\ (w_{t+j} |\ w_t)$



Word2Vec: objective function

- The objective is to predict context words given a target word.
- The model tries to maximize the likelihood of observing the context words given the target word.
- The likelihood can be expressed as a product of conditional probabilities over all observed word pairs in the training data.

Word2Vec: objective function

For each position $t = 1, \ldots, T$, predict context words within a window of fixed size *m*, given center word w_t. Data likelihood:

$$\text{Likelihood} = L(\theta) = \prod_{t=1}^{T} \prod_{\substack{-m \leq j \leq m \\ j \neq 0}} P(w_{t+j} \mid w_t; \theta)$$

θ is all variables to be optimized

sometimes called a *cost* or *loss* function

The objective function $J(\theta)$ is the (average) negative log likelihood:

$$J(\theta) = -\frac{1}{T} \log L(\theta) = -\frac{1}{T} \sum_{t=1}^{T} \sum_{\substack{-m \leq j \leq m \\ j \neq 0}} \log P(w_{t+j} \mid w_t; \theta)$$

Minimizing objective function $\Leftrightarrow$ Maximizing predictive accuracy

Word2Vec: objective function

- We want to minimize the objective function:

$$J(\theta) = -\frac{1}{T}\sum_{t=1}^{T}\sum_{\substack{-m\le j\le m \\ j\neq 0}} \log P(w_{t+j} \mid w_t; \theta)$$

- Question: How to calculate $P\,(w_{t+j} \mid w_t; \theta)$?
- Answer: We will use two vectors per word w:
  - v_w when w is a center word
  - u_w when w is a context word
- Then for a center word c and a context word o:

$$P(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w\in V}\exp(u_w^T v_c)}$$

Word2vec: prediction function

① Dot product compares similarity of o and c.
$u^T v = u.v = \sum_{i=1}^{n} u_i v_i$
Larger dot product = larger probability

② Exponentiation makes anything positive

$$P(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w \in V} \exp(u_w^T v_c)}$$

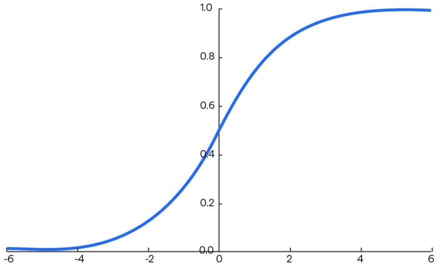
③ Normalize over entire vocabulary
to give probability distribution

Open
region

This is an example of the softmax function $\mathbb{R}_n \rightarrow (0,1)^n$

$$\text{softmax}(x_i) = \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)} = p_i$$

Softmax Function



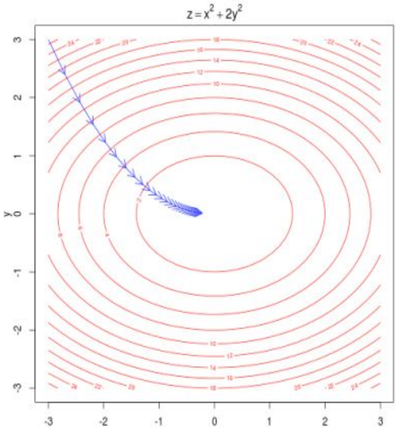
- The softmax function maps arbitrary values x_i to a probability distribution p_i
  - "max" because amplifies probability of largest x_i
  - "soft" because still assigns some probability to smaller x_i
  - Frequently used in Deep Learning

To train the model: Optimize value of parameters to minimize loss

To train a model, we gradually adjust parameters to minimize a loss

- Recall: θ represents all the model parameters, in one long vector
- In our case, with d-dimensional vectors and V-many words, we have $\rightarrow$
- Remember: every word has two vectors

$$\theta = \begin{bmatrix} v_{aardvark} \\ v_a \\ \vdots \\ v_{zebra} \\ u_{aardvark} \\ u_a \\ \vdots \\ u_{zebra} \end{bmatrix} \in \mathbb{R}^{2dV}$$



- We optimize these parameters by walking down the gradient (see right figure)
- We compute all vector gradients!

Objective Function

Maximize $J'(\theta) = \prod_{t=1}^{T} \prod_{\substack{-m \le j \le m \\ j \ne 0}} p(w_{t+j}' \mid w_t ; \theta)$

Or minimize ave. neg. log likelihood

$$J(\theta) = -\frac{1}{T} \sum_{t=1}^{T} \sum_{\substack{-m \le j \le m \\ j \ne 0}} \log p(w_{t+j}' \mid w_t)$$

[negate to minimize; log is monotone]

Text length

window size

where

$$p(o \mid c) = \frac{\exp(u_o^T v_c)}{\sum_{w=1}^{V} \exp(u_w^T v_c)}$$

word IDs

Each word type (vocab entry) has two word representations: as center word and context word

We now take derivatives to work out minimum

$$\frac{\partial}{\partial v_c} \log \frac{\exp(u_o^T v_c)}{\sum_{w=1}^{V} \exp(u_w v_c)}$$

$$= \underbrace{\frac{\partial}{\partial v_c} \log \exp(u_o^T v_c)}_{1} - \underbrace{\frac{\partial}{\partial v_c} \log \sum_{w=1}^{V} \exp(u_w^T v_c)}_{2}$$

1 $\frac{\partial}{\partial v_c} \underbrace{\log \exp}_{\text{inverses}} (u_o^T v_c) = \frac{\partial}{\partial v_c} u_o^T v_c = u_o$

Vector! Not high school single variable calculus

You can do things one variable at a time, and this may be helpful when things get gnarly.

$$\forall j \quad \frac{\partial}{\partial (v_c)_j} u_o^T v_c = \frac{\partial}{\partial (v_c)_j} \sum_{i=1}^{d} (u_o)_i (v_c)_i$$

$$= (u_o)_j$$

Each term is zero except when $i=j$

$$② \; \frac{\partial}{\partial v_c} \underbrace{\log}_{f} \underbrace{\sum_{w=1}^{V} \exp(u_w^T v_c)}_{z = g(v_c)}$$

$$= \underbrace{\frac{1}{\sum_{w=1}^{V} \exp(u_w^T v_c)}}_{} \cdot \underbrace{\frac{\partial}{\partial v_c} \sum_{x=1}^{V} \exp(u_x^T v_c)}_{}$$

Important to change index

$$\frac{\partial}{\partial v_c} f(\overbrace{g(v_c)}^{z}) = \frac{\partial f}{\partial z} \cdot \frac{\partial z}{\partial v_c}$$

Use chain rule

$$= \frac{1}{\sum_{w=1}^{V} \exp(u_w^T v_c)} \cdot \left(\sum_{x=1}^{V} \frac{\partial}{\partial v_c} \underbrace{\exp}_{f} (\underbrace{u_x^T v_c}_{z = g(v_c)}) \right)$$

Move deriv inside sum

$$\left(\sum_{x=1}^{V} \exp(u_x^T v_c) \frac{\partial}{\partial v_c} u_x^T v_c \right)$$

Chain rule

$$\left(\sum_{x=1}^{V} \exp(u_x^T v_c) \, u_x \right)$$

$$\frac{\partial}{\partial v_c} \log(p(o|c)) = u_o - \frac{1}{\sum_{w=1}^{V} \exp(u_w^T v_c)} \cdot \left(\sum_{x=1}^{V} \exp(u_x^T v_c)\, u_x \right)$$

$$= u_o - \sum_{x=1}^{V} \frac{\exp(u_x^T v_c)}{\sum_{w=1}^{V} \exp(u_w^T v_c)}\, u_x$$

Distribute term across sum

$$= u_o - \underbrace{\sum_{x=1}^{V} p(x|c)\, u_x}$$

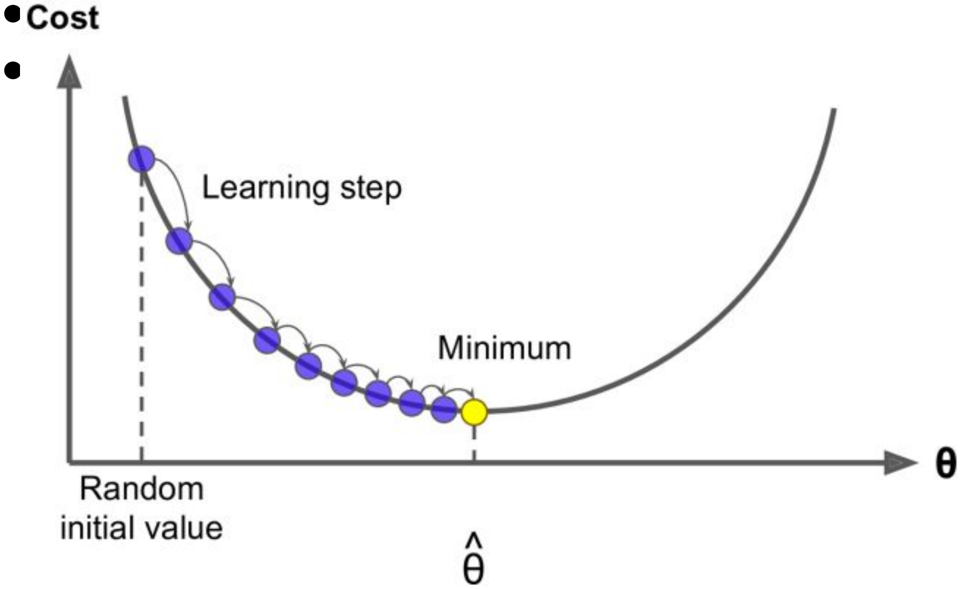
This an expectation: average over all context vectors weighted by their probability

= observed − expected

This is just the derivatives for the center vector parameters
Also need derivatives for output vector parameters
(they're similar)
Then we have derivative w.r.t. all parameters and can minimize

Optimization: Gradient Descent

- We have a cost function $J(\theta)$ we want to minimize
- Gradient Descent is an algorithm to minimize $J(\theta)$
- all step in direction
-



Note: Our objectives may not be convex like this ☹

But life turns out to be okay ☺

Gradient Descent

- Update equation (in matrix notation):

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

α = *step size* or *learning rate*

- Update equation (for single parameter):

$$\theta_j^{new} = \theta_j^{old} - \alpha \frac{\partial}{\partial \theta_j^{old}} J(\theta)$$

- Algorithm:

```
while True:
    theta_grad = evaluate_gradient(J,corpus,theta)
    theta = theta - alpha * theta_grad
```

Stochastic Gradient Descent

- Problem: $J(\theta)$ is a function of all windows in the corpus (potentially billions!)
  - So $\triangle_{\theta} J(\theta)$ is very expensive to compute
- You would wait a very long time before making a single update!
- Very bad idea for pretty much all neural nets!
- Solution: **Stochastic gradient descent** (SGD)
  - Repeatedly sample windows, and update after each one
- Algorithm:

```
while True:
    window = sample_window(corpus)
    theta_grad = evaluate_gradient(J,window,theta)
    theta = theta - alpha * theta_grad
```

Word2vec Algorithm Family (Mikolov et al. 2013)

1. Two model variants:
   a. **Skip-grams (SG)** Predict context ("outside") words (position independent) given center word
   b. **Continuous Bag of Words (CBOW)** Predict center word from (bag of) context words
2. Loss functions for training:
   a. Naïve softmax (simple but expensive loss function, when many output classes)
   b. Negative sampling

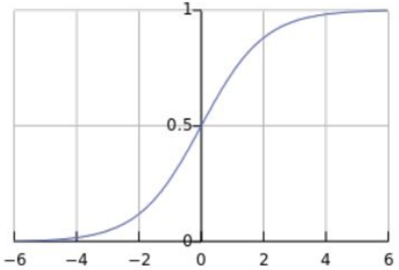
The Skip-gram Model with Negative Sampling

- Introduced in: "Distributed Representations of Words and Phrases and their Compositionality" (Mikolov et al. 2013)
- Overall objective function (they maximize):

$$J(\theta) = \frac{1}{T}\sum_{t=1}^{T} J_t(\theta)$$

$$J_t(\theta) = \log \sigma\left(u_o^T v_c\right) + \sum_{i=1}^{k} \mathbb{E}_{j \sim P(w)}\left[\log \sigma\left(-u_j^T v_c\right)\right]$$

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



- The logistic/sigmoid function:
- We maximize the probability of two words co-occurring in first log and minimize probability of noise words in second part

The Skip-gram Model with Negative Sampling (Con't)

Affinity of word c for context o

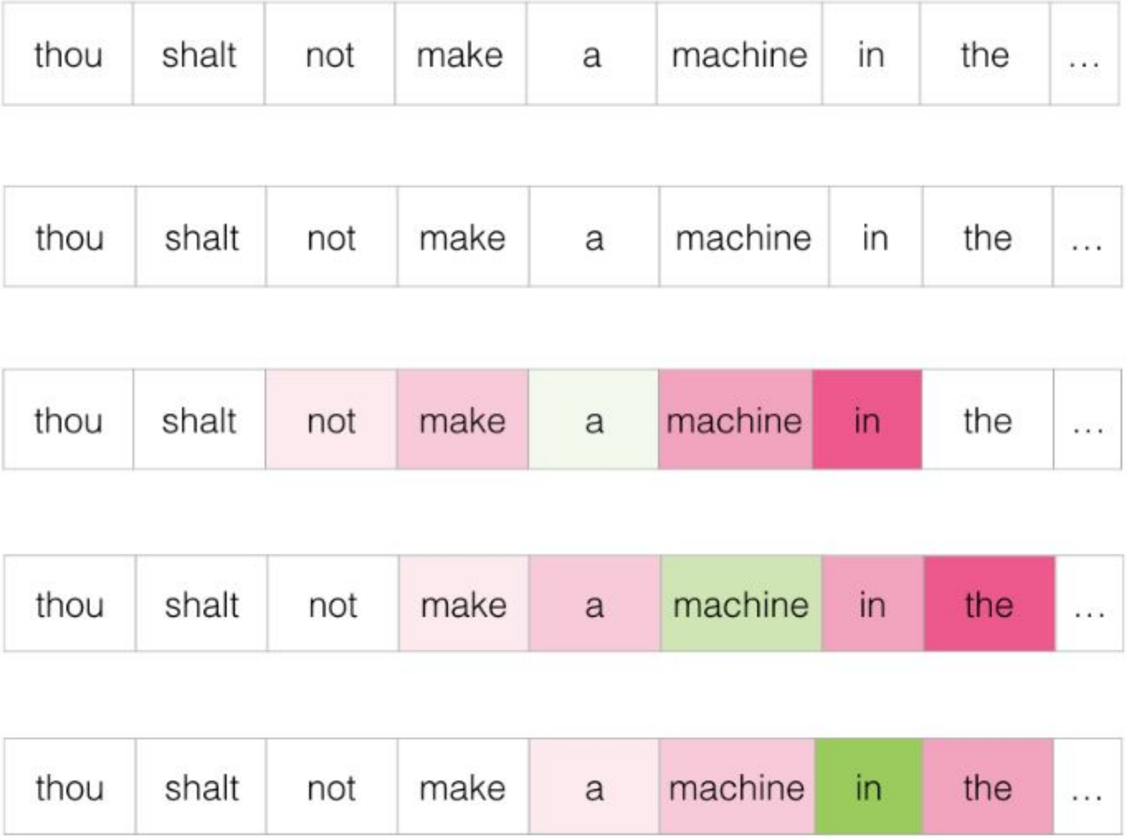
$$\boxed{\log \sigma(u_o^\top v_c)} + \boxed{\sum_{\ell=1}^{k} \left[\log \sigma(-u_\ell^\top v_c)\right]} \tag{15}$$

Push down everything else in expectation (replacement for partition fn)

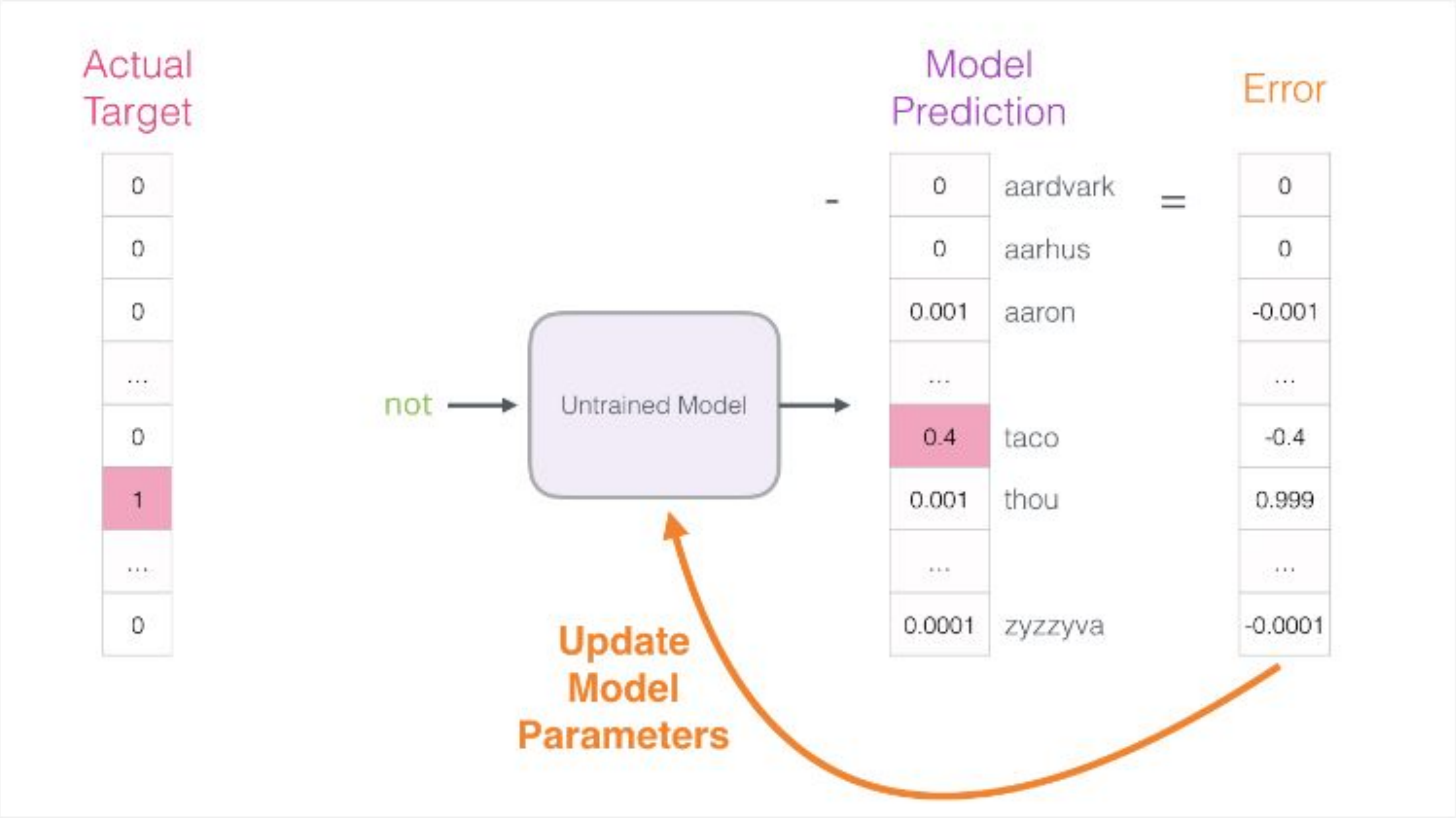
Understanding the Skip Gram Model

- https://jalammar.github.io/illustrated-word2vec/

Thou shalt not make a machine in the likeness of a human mind



input word	target word
not	thou
not	shalt
not	make
not	a
make	shalt
make	not
make	a
make	machine
a	not
a	make
a	machine
a	in
machine	make
machine	a
machine	in
machine	the
in	a
in	machine
in	the
in	likeness



Actual Target
0
0
0
...
0
1
...
0
not
Untrained Model
Model Prediction
0 aardvark
0 aarhus
0.001 aaron
...
0.4 taco
0.001 thou
...
0.0001 zyzzyva
-
=
Error
0
0
-0.001
...
-0.4
0.999
...
-0.0001
Update Model Parameters

Negative Sampling

input word	target word
not	thou
not	shalt
not	make
not	a
make	shalt
make	not
make	a
make	machine

input word	output word	target
not	thou	1
not	shalt	1
not	make	1
not	a	1
make	shalt	1
make	not	1
make	a	1
make	machine	1

- If all training examples are **positive** (target: 1), the model may always predict 1, achieving **100% accuracy** without learning meaningful embeddings.
- Such a model would **overfit** and produce **useless embeddings**.
- To address this, **negative samples** (non-neighbor words) are introduced into the dataset.
- The model is trained to predict **0 for negative samples**, forcing it to distinguish between true neighbors (positive) and non-neighbors (negative).

Negative Sampling

- But what do we fill in as output words? We randomly sample words from our vocabulary

input word	output word	target
not	thou	1
not		0
not		0
not	shalt	1
not	make	1

Negative examples

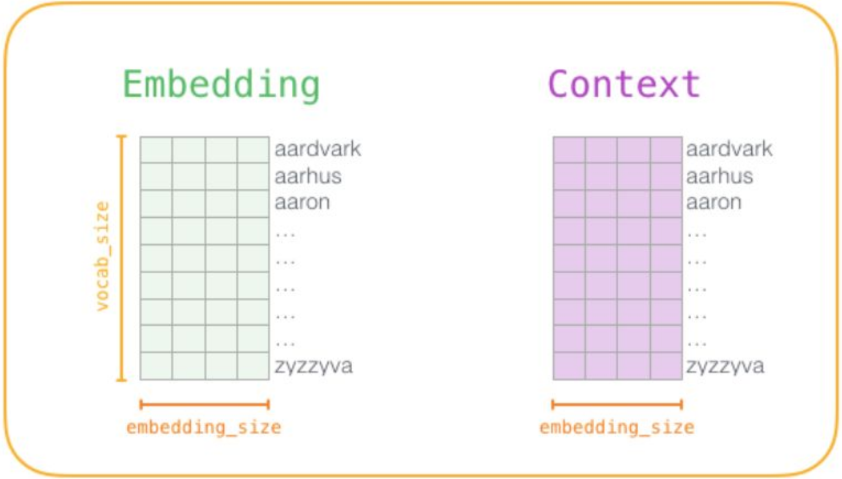
Pick randomly from vocabulary (random sampling)

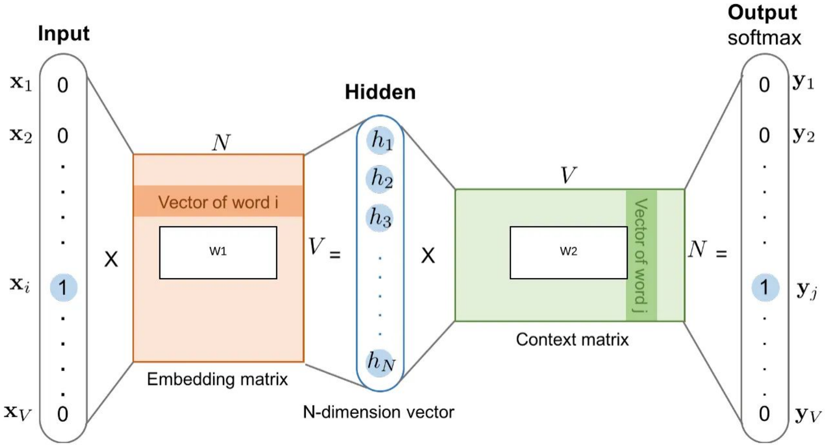
input word	output word	target
not	thou	1
not	aaron	0
not	taco	0
not	shalt	1
not	make	1

Word	Count	Probability
aardvark		
aarhus		
aaron		
taco		
thou		
zyzzyva		

Word2vec Training Process

- Before training, the text data is **pre-processed** to determine the **vocabulary size** (`vocab_size`), e.g., 10,000 words.
- Words outside this vocabulary are excluded.
- At the start of training, two matrices are created:
  - **Embedding matrix**
  - **Context matrix**
- Both matrices have dimensions:
  - vocab_size(number of words in the vocabulary).
  - embedding_size(length of each word's embedding, e.g., 50 or 300).





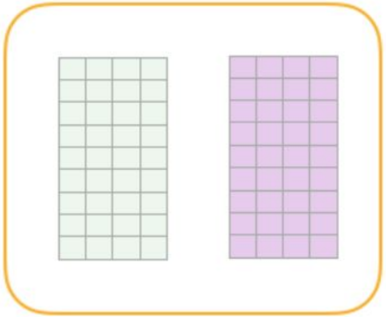
- At the start of training, the `Embedding` and `Context` **matrices** are initialized with **random values**.
- The training process begins by iterating through examples.
- In each training step:
  - A **positive example** is selected.
  - Its associated **negative examples** are included for contrast.
- The model learns by updating embeddings based on these examples.
- Now we have four words: the **input word not** and **output/context words**: **thou** (the actual neighbor), **aaron**, and **taco** (the negative examples).

- We proceed to look up their embeddings – for the input word, we look in the `Embedding` matrix. For the context words, we look in the `Context` matrix (even though both matrices have an embedding for every word in our vocabulary).

dataset

input word	output word	target
not	thou	1
not	aaron	0
not	taco	0
not	shalt	1
not	mango	0
not	finglonger	0
not	make	1
not	plumbus	0
...	...	...

model



- Then, we take the dot product of the input embedding with each of the context embeddings.
- In each case, that would result in a number, that number indicates the similarity of the input and context embeddings

input word	output word	target	input • output
not	thou	1	0.2
not	aaron	0	-1.11
not	taco	0	0.74

- Now we need a way to turn these scores into something that looks like probabilities – we need them to all be positive and have values between zero and one.
- This is a great task for **sigmoid**, the **logistic operation**.

input word	output word	target	input • output	sigmoid()
not	thou	1	0.2	0.55
not	aaron	0	-1.11	0.25
not	taco	0	0.74	0.68

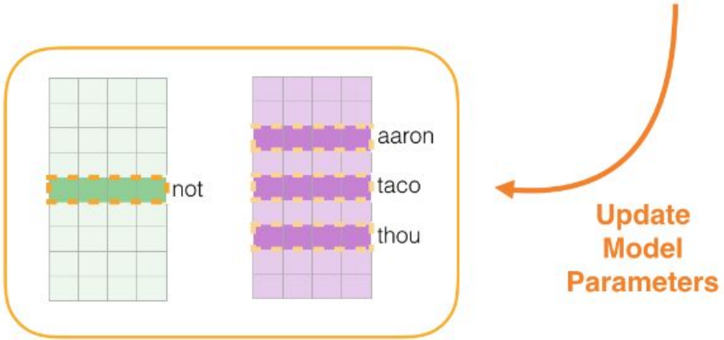
- The error in the model's prediction is calculated by comparing predictions to target labels.
- Specifically, the error is obtained by subtracting the sigmoid scores (model predictions) from the target labels.

input word	output word	target	input • output	sigmoid()	Error
not	thou	1	0.2	0.55	0.45
not	aaron	0	-1.11	0.25	-0.25
not	taco	0	0.74	0.68	-0.68

```
error = target - sigmoid_scores
```

Here comes the “learning” part of “machine learning”. We can now use this error score to adjust the embeddings of `not`, `thou`, `aaron`, and `taco` so that the next time we make this calculation, the result would be closer to the target scores.

input word	output word	target	input • output	sigmoid()	Error
not	thou	1	0.2	0.55	0.45
not	aaron	0	-1.11	0.25	-0.25
not	taco	0	0.74	0.68	-0.68



- The process is repeated for the **next positive sample** and its associated **negative samples**.
- This continues as the model cycles through the **entire dataset** multiple times (epochs).
- During this process, the **embeddings are improved** iteratively.
- After training:
  - The `Context` **matrix** is discarded.
  - The `Embedding` **matrix** is retained as **pre-trained embeddings** for future tasks.

History of Word Represented as Vectors

- **One-hot Encoding**: Proposed by Alan Turing (Year: 1943)
- **Co-Occurrence Matrix**: Proposed by Firth, J. R. (Year: 1957)
- **CBOW** (Continuous Bag of Words): Proposed by Mikolov, T., Chen, K., Corrado, G., & Dean, J. (Year: 2013)
- ***Skip-gram**: Proposed by Mikolov, T., Chen, K., Corrado, G., & Dean, J. (Year: 2013)*
- **GloVe** (Global Vectors for Word Representation): Proposed by Pennington, J., Socher, R., & Manning, C. D. (Year: 2014)
- **FastText**: Proposed by Bojanowski, P., Grave, E., Joulin, A., Mikolov, T., & Mikolov, J. (Year: 2017) Poincaré Embedding: Proposed by Nickel, M., Kiela, D. (Year: 2017)
- **ELMo** (Embeddings from Language Models): Proposed by Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., & Zettlemoyer, L. (Year: 2018)
- **BERT** (Bidirectional Encoder Representations from Transformers): Proposed by Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (Year: 2018)

Thanks