Explanatory Notes for 6.3900

Shaunticlair Ruiz

Fall 2024

Contents

APPENDIX D

Word2vec – Skipgram Approach

D.1 Vector embeddings and tokens

In these notes, we introduce a particular way to choose "good" vector embeddings, based on the word2vec technique.

- These notes were originally spliced from the Transformers chapter, so there are some regions of overlap.

D.1.1 One-hot encoding isn't enough

First, we want to turn words into something computable, like a **vector**.

It's difficult to try to do math on the word "cheddar". It's not numerical.

The simplest approach would be **one-hot encoding**.

- **Example:** Suppose that we want to classify **furniture** as table, bed, couch, or chair.

$$\begin{bmatrix} \text{table} \\ \text{bed} \\ \text{couch} \\ \text{chair} \end{bmatrix} \tag{D.1}$$

- For each class:

$$v_{\texttt{chair}} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \qquad v_{\texttt{table}} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad v_{\texttt{couch}} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \qquad v_{\texttt{bed}} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \tag{D.2}$$

This approach is simple, but often, it's *too* simple.

Concept 1

One-hot encoding loses a lot of information about the objects it's representing.

- It's hard to say which words are "**similar**" to each other, for example.

Example: You probably associate the word "sugar" with "sweet", and "salt" with "savory".

- But, if you use one-hot encoding, all of these words are "equally different".

You could **shuffle** the rows of one-hot vectors, and represent the same information.

So, we can't use the order of 1's and 0's to determine "closeness": the order can be **freely changed**.

$$v_{\texttt{salt}} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \qquad v_{\texttt{savory}} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad v_{\texttt{sugar}} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \qquad v_{\texttt{sweet}} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \tag{D.3}$$

In order to incorporate this information, we'll need a better way to represent words as vectors.

D.1.2 Word Embeddings: Similarity between words

Our new approach will convert each word w into a **vector** v_w of **length** d.

Unlike one-hot encoding, we don't require that d equals the size of our vocabulary.

$$w \longrightarrow v_w \qquad\qquad v_w \in \mathbb{R}^d \tag{D.4}$$

How do we want to convert words into vectors? Above, we mentioned that one-hot doesn't tell us how **similar** two words are.

Clarification 2

There are many ways for words to be **similar**: similar word length, similar choice of letters, etc.

But in our case, we're interested in **semantics**: the **meanings** of the words. We want to know which words have similar meanings.

- **Example:** We don't consider "sugar" and "sweet" to be similar because they both start with "s".

- They're similar because of **meaning**: sugar tastes sweet. Sweet strawberries contain sugar.

Concept 3

We often want our **word embeddings** v_w to tell us which words are **semantically similar** to each other: which words have similar **meanings**.

v_a and v_b are **similar vectors** $\iff$ a and b are **semantically similar words**

Our goal is to make this statement true. But we have a problem: these are *concepts*, rather than computable *numbers*.

- So, we'll have to turn each side into something computable.

D.1.3 Vector Similarity: Dot Products

First, we'll handle the left side: how do we know if vectors are **similar**?

- We've come across this problem multiple times, and we'll solve it the same way as always: using the **dot product**.

Concept 4

Review from the Classification chapter

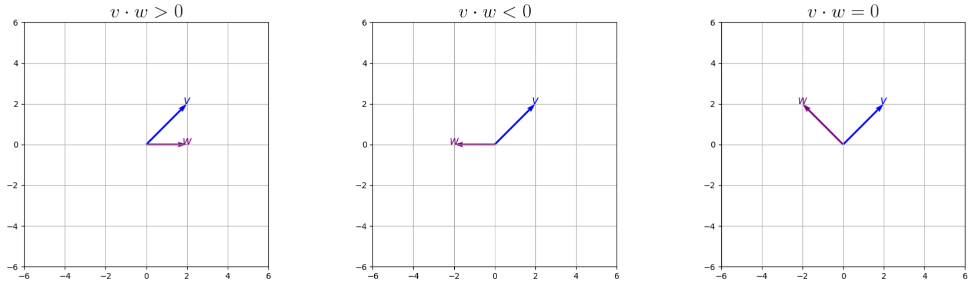
You can use the **dot product** between vectors u and v, **normalized by their magnitudes**, to measure their "**cosine similarity**".

$$S_C(u,v) = \frac{u \cdot v}{|u| \cdot |v|}$$

If two vectors are more **similar**, they have a **larger** normalized dot product.

- This function ranges from -1 (opposite vectors) to +1 (identical vectors). Perpendicular vectors receive a 0.

We call it "cosine similarity", because this is equal to the cosine of the angle α between u and v.



We can see here what we mean by "similar" or "dissimilar".

Clarification 5

You can use $S_C(u, v)$ to measure the **similarity** between two vectors, ignoring magnitude.

But for simplicity, we'll skip the **normalizing** step, and just take the **dot product**:

$$S_D(u, v) = u \cdot v = u^\top v$$

We're getting closer to a computable form:

$$\overbrace{(v_a \cdot v_b)}^{\text{Similar vectors}} \text{ is } \textbf{large} \iff a \text{ and } b \text{ are } \textbf{semantically similar words} \tag{D.5}$$

D.1.4 Semantic Similarity and Word Frequency

The "right side" of our expression is a bit trickier: how do you compute which words have **similar meanings**?

We can't directly turn "meaning" into a number. But instead, we'll focus on a different concept, that might help us predict similarity:

- **Example:** Earlier, we showed that "sweet and "sugar" were related, by referencing the fact that "**sugar tastes sweet**".
- While our machine might not understand the concept, it can see that "sugar" and "sweet" showed up **together** in a sentence.

How much do/can large language models "understand" what they're saying? Lots of very smart people continue to argue exactly how much they know.

Often, words that are related, show up in the same sentences, or paragraphs. So, we'll try to use this to our advantage:

$$a \text{ and } b \text{ are } \textbf{semantically similar words} \overset{\text{maybe?}}{\iff} a \text{ and } b \textbf{ frequently} \text{ show up together}$$

These two aren't *actually* equivalent, but we hope that we can use one to predict the other.

Concept 6

We can predict which words might be **more similar** by observing **how often** they show up **together** in a body ("corpora") of text.

- When two words occur **together** in a context, we call this **co-occurrence**.
- Thus, we're measuring **frequency of co-occurrence**.

If two words show up near each other more frequently, we predict that they might be **more similar**.

This kind of word embedding is often called "**word2vec**", named after a particular set of algorithms that use this approach.

Sometimes, "word2vec" is used to reference any technology that creates word embeddings. But this isn't always technically accurate.

- **Example:** The words "quantum" and "physics" go together often. So do the words "rain" and "weather".

Clarification 7

We **don't actually know** for certain that, if two words often show up together, they have **related meanings**.

But, in practice, we find that "**frequency of co-occurrence**" is a **surprisingly good** measure of similarity.

Because we're talking about frequency, we'll consider the **probability** of seeing both words together.

$$a \text{ and } b \text{ are } \textbf{semantically similar words} \overset{\text{maybe?}}{\iff} \mathrm{P}\Big(\, a \text{ and } b \textbf{ occur together} \Big) \text{ is } \textbf{high}$$

Finally, we have something closer to math:

$$\overbrace{v_a \cdot v_b \text{ is } \textbf{large}}^{\text{Similar vectors}} \iff \overbrace{\mathrm{P}\Big(\, a \text{ and } b \textbf{ occur together} \Big) \text{ is } \textbf{high}}^{\text{Similar words}}$$

Now, we have some "mathematical" concepts: we can start using these to create mathematical **objects**.

D.1.5 Clarifying our probability

In order to proceed, we need to be a little more specific.

$$\overbrace{(v_a \cdot v_b) \text{ is } \textbf{large}}^{\text{Similar vectors}} \iff \overbrace{\text{P}\left(\ a \text{ and } b \textbf{ occur together}\ \right) \text{ is } \textbf{high}}^{\text{Similar words}}$$

The dot product is already an equation, so the left side is fine.

The right side is all we need to clear up: "P (a and b **occur together**)" is a bit **vague**.

One interpretation would be: "if we look at a random phrase, how often do we have words a and b?"

But we only care whether a and b are together/separate: we don't care about sentences containing neither.

- We want to know if a and b tend to show up **together**, rather than **separately**.

Here's a concrete way to say this: "**if** we find one word, **how often** do we find the other nearby?"

Concept 8

To predict how **similar** words a and b are, we want to compute how often they **co-occur**.

- One way to phrase this: "**given** that we find a, what are the **chances** we find b nearby?"

$$\mathbf{P}\left\{ \text{b nearby} \;\middle|\; \text{a found} \right\}$$

$$\overbrace{(v_a \cdot v_b) \text{ is } \textbf{large}}^{\text{Similar vectors}} \iff \overbrace{\mathbf{P}\left\{ \text{b nearby} \;\middle|\; \text{a found} \right\} \text{ is } \textbf{large}}^{\text{Occur together frequently}}$$

We're getting warmer!

- We "**find**" a at index t:

w_t is the t^{th} word in our passage.

$$w_t = a \tag{D.6}$$

- Now, let's define what it means for b to be "**nearby**".

Definition 9

In a text, we may want to find the "context" for **center word** w_t: we want all of the words **nearby**.

- We'll use the c nearest words on either side: these are our **context words**. c is our **maximum skip distance**.

This collection of $2c + 1$ words is called our **context window**.

$$\cdots \quad w_{t-4} \quad \overbrace{w_{t-3} \quad w_{t-2} \quad w_{t-1} \quad w_t \quad w_{t+1} \quad w_{t+2} \quad w_{t+3}}^{\text{Context window for } c\,=\,3} \quad w_{t+4} \quad \cdots$$

- Notice the similarity to the filter size from Convolution: we still have an idea of "locality".

We call c our "maximum skip distance", because it's the largest number of words we can "skip" over, starting from w_t.

We're allow to move over by c words, in either direction.

So, we want to look for b in our context window. There are two ways we can turn this into a probability:

- You check all of the context words **at the same time**:

$$\cdots \quad w_{t-3} \quad \overbrace{w_{t-2} \quad w_{t-1} \quad w_t \quad w_{t+1} \quad w_{t+2}}^{\text{All words within } c\,=\,2 \text{ context window}} \quad w_{t+3} \quad \cdots \tag{D.7}$$

- You check **one word at a time**: j units to the right/left.

$$\cdots \quad w_{t-3} \quad \overbrace{w_{t-2} \quad w_{t-1} \quad w_t}^{\text{Only } w_{t-2}\text{. Thus, } j\,=\,-2} \quad w_{t+1} \quad w_{t+2} \quad w_{t+3} \quad \cdots \tag{D.8}$$

For now, it's easier to use the latter approach: **each index** has a **separate probability**.

Concept 10

We measure the **co-occurrence** of a and b by asking:

- "**Given** that we find a at index t...

$$w_t = a$$

- what are the **chances** that we find b at index $t + j$?"

$$w_{t+j} = b$$

With this, we find our result:

$$\mathbf{P}\left\{w_{t+j} = b \;\middle|\; w_t = a\right\}$$

We did it! This is a clear, explicit probability.

$$\overbrace{(v_a \cdot v_b) \text{ is large}}^{\text{Similar vectors}} \iff \overbrace{\mathbf{P}\left\{w_{t+j} = b \;\middle|\; w_t = a\right\} \quad \text{is large}}^{\text{Occur together frequently}}$$

Notation 11

We can make this notation a little denser:

$$\mathbf{P}\left\{w_{t+j} = b \;\middle|\; w_t = a\right\} \quad = \quad \mathbf{P}\left\{b \;\middle|\; a\right\}_j$$

This assumes that t **doesn't** affect our probability: it doesn't matter **where** we found a, just **how far away** b is (and on which side).

- This is a reasonable assumption for our purposes.

D.1.6 Computing predicted probabilities

How do we turn a **real number** $v_a \cdot v_b$ into a **probability** $P(b \mid a)_j$?

- $P(b \mid a)_j$ is the chance of finding b at index $t + j$, if a is at index t.

$$\cdots \quad w_{t-3} \quad \overbrace{w_{t-2}\quad w_{t-1}\quad w_t}^{\text{What word is at } w_{t-2}\text{? Is it } b\text{?}} \quad w_{t+1} \quad w_{t+2} \quad w_{t+3} \quad \cdots \tag{D.9}$$

- So, we need to compare b to every other word that we could find at $t + j$: this is a

multi-class problem, using the **softmax function**.

We have one class for each possible word we could find at $t+j$.

$$\text{Softmax}(z_k) = \frac{e^{z_k}}{\sum_i e^{z_i}} \tag{D.10}$$

Let's review the concept behind "softmax":

Concept 12

Suppose that we have n **possible words** (n "classes"), and we want to figure out which one is **correct**.

The k^{th} **class** has a score, z_k, used to compute probability.

- The bigger z_k is, the **more likely** k is to be the **correct class**.

To keep it **positive**, z_k is converted to e^{z_k}: each e^{z_i} competes to see which class is more likely.

- To create a probability, we **compare** the score of class k to all of our other classes, using **softmax**.

$$\overbrace{e^{z_k}}^{\text{Class k}} \quad \text{vs} \quad \overbrace{\sum_i e^{z_i}}^{\text{All classes}} \qquad \Longrightarrow \qquad \text{Softmax}(z_k) = \frac{e^{z_k}}{\sum_i e^{z_i}}$$

- We repeat this process for every possible word i, to get all of our predictions.

Now, the big question: what is z_k?

$$\overbrace{(v_a \cdot v_b) \text{ is large}}^{\text{Similar vectors}} \quad \Longleftrightarrow \quad \overbrace{\mathbf{P}\Big\{w_{t+j} = b \;\Big|\; w_t = a\Big\} \quad \text{is large}}^{\text{Occur together frequently}}$$

z_k and $(v_a \cdot v_b)$ serve the **same purpose**:

- Large **dot product** predicts high probability.
- Large z_k predicts high probability.

So, we can use our dot product as a "score" z_k:

$$z_b = v_a \cdot v_b \tag{D.11}$$

Now, we can plug this into our probability equation!

Key Equation 13

The **more similar** (bigger dot product) a and b are, the **more likely** we predict to find them together.

- We use a **softmax** to compute this probability for each possible word b.

$$\mathbf{P}\left\{w_{t+j} = b \;\middle|\; w_t = a\right\} \quad = \quad \frac{e^{v_a \cdot v_b}}{\sum_i e^{v_a \cdot v_i}}$$

Or, in alternate notation:

$$\mathbf{P}\left\{b \;\middle|\; a\right\} \quad = \quad \frac{\exp\left(v_a \cdot v_b\right)}{\sum_i \exp\left(v_a \cdot v_i\right)}$$

Ta-da! We've combined two separate concepts into a single equation.

Note that, in both top and bottom, we keep v_a: we're considering every possible word for w_{t+j}, while we *know* $w_t = a$.

D.1.7 Skip-gram approach: Training our word2vec model

One remaining issue: this equation doesn't tell us what the "true" probabilities are: they tell us the probability that our model **predicts**.

- Now, we have to choose a **good model** (word embedding).

Clarification 14

Our equation is $\mathbf{P}\left\{w_{t+j} = b \;\middle|\; w_t = a\right\}$ is our **estimation** for the probability.

- The real probabilities could be **different**: we'll design our word embedding to give us the most **accurate probabilities**.

First: what does our model look like? How do we even **generate** word embeddings?

- Often, we rely on a neural network.

Definition 15

We have two common **models for word embedding** (θ):

- Separately assigning a **vector** to each word.
- Using a shared **neural network** to embed every word as a vector.

Our neural network uses **parameters** θ. We'll use θ to represent our embedding, that we want to **train**.

$$w \xrightarrow{\theta} v_w$$

How do we pick a good model?

- We'll **train** our embedding θ, so that our **probabilities** are as accurate as possible.

As we established, our problem is multi-class classification:

Concept 16

Review from Classification chapter

For **multi-class classification**, we use the **negative log-likelihood multiclass** (NLLM) equation to compute **loss**:

$$\mathcal{L}_{NLLM}(g, y) = -\sum_{i=1}^{n} y_i \log(g_i)$$

y is a one-hot vector, so all terms of the sum except the "correct" term $i = k$ cancel out to 0:

$$-y_k \log(g_k) \qquad \xrightarrow{y_k=1} \qquad -\log(g_k)$$

g_k is the probability we assigned to the correct answer.

Next, we need training data: a body of **text**.

- For an example, let's visit index t in the text: this is the center of our **context window**.
- a is replaced by whatever word we find at that index: w_t. We still want to predict w_{t+j}.

$$\cdots \quad w_{t-3} \quad w_{t-2} \quad w_{t-1} \quad w_t \quad \cdots \quad w_{t+j} \quad w_{t+j+1} \quad \cdots \tag{D.12}$$

How good is our word embedding? According to NLLM: "how likely were we to correctly predict w_{t+j}?"

$$\mathcal{L}_{\text{NLLM}}(g, y) = -\log\left(\mathbf{P}\left\{\text{Correct word for index } t+j \;\middle|\; w_t\right\}\right)$$

The correct word for index $t+j$ would be... w_{t+j}. We can read the outcome from the text, and use our model to check how **likely** we thought that outcome was.

$$\mathcal{L}_{\text{NLLM}}(g, y) = -\log\left(\overbrace{\mathbf{P}\left\{w_{t+j} \;\middle|\; w_t\right\}}^{\text{How likely we thought } w_{t+j} \text{ was, based on model}} \right)$$

Note that, the higher this probability is (the more sure we are of the correct answer), the closer the loss gets to 0.

Key Equation 17

We train our **word embedding** θ by **maximizing** the probability $\mathbf{P}(w_{t+j} \mid w_t)$ of predicting the **correct word** in each spot.

In our case, we want to **minimize**

$$\mathcal{L}_{\text{NLLM}}(\theta, j) = -\log\left(\mathbf{P}\left\{w_{t+j} \;\middle|\; w_t\right\}\right)$$

where

$$\mathbf{P}\left\{b \;\middle|\; a\right\} = \frac{\exp\left(v_a \cdot v_b\right)}{\sum_i \exp\left(v_a \cdot v_i\right)}$$

Now, we know how to compute these odds for a **single index**, $t+j$. We want to repeat this process for the rest of our context window:

$$\cdots \quad w_{t-3} \quad \overbrace{w_{t-2} \quad w_{t-1} \quad w_t \quad w_{t+1} \quad w_{t+2}}^{\text{All words within } c = 2 \text{ context window}} \quad w_{t+3} \quad \cdots \tag{D.13}$$

Key Equation 18

We can find the **total loss** of our embedding θ, over our entire **context window**, by adding up the loss from each **context word**.

This includes all of the indices $(t+j)$, going from $j=-c$ to $j=+c$. Meaning, we want

- $|j| \leqslant c$ (within window)
- $j \neq 0$ (don't want to compare w_t with itself)

$$\mathcal{L}_t(\theta) = -\sum_{|j|\leqslant c}^{j\neq 0} \log\left(\mathbf{P}\left\{w_{t+j} \;\middle|\; w_t\right\}\right)$$

One more modification: the loss function above only computes loss for a **single context window**.

But, for a passage of text, there are many possible context windows: all we have to do is shift our target word, w_t.

- **Example:** Below, with $c = 2$, we'll show all of our possible context windows:

Target word is red, context words are blue.

This is a sample sentence
This is a sample sentence
This is a sample sentence (D.14)
This is a sample sentence
This is a sample sentence

We'll average the loss over **all context windows**.

We'll ignore negative indices, so we don't cause problems when $t = 0$ or $t = T$.

Key Equation 19

Take a body of text with T words, and a **context window** with a **max skip distance** of c. We use word embedding θ.

Our **objective function** $J(\theta)$ for the **skip-gram word2vec** algorithm, over the entire passage, is:

$$J(\theta) = \frac{1}{T}\sum_{t=1}^{T} \mathcal{L}_t(\theta)$$

or,

$$J(\theta) = -\frac{1}{T}\sum_{t=1}^{T} \left(\sum_{|j|\leqslant c}^{j\neq 0} \log\left(\mathbf{P}\left\{w_{t+j} \;\middle|\; w_t\right\}\right) \right)$$

This is the completed loss function that we can use to train our embedding.

Concept 20

We can train our **embedding** θ using our **loss function** J, via **gradient descent**.

$$\theta' = \theta - \eta \nabla_{\theta} J$$

- If we're using a **neural network** to create embeddings, we'll need **back-propagation** to train.

D.1.8 Issues with skip-gram

There are a few problems worth addressing. First, look again at our equation:

$$\mathbf{P}\left\{w_{t+j} = b \;\middle|\; w_t = a\right\} \quad = \quad \frac{e^{v_a \cdot v_b}}{\sum_i e^{v_a \cdot v_i}}$$

Something might strike you: our probability is **totally independent** of which index $t + j$ we want to predict.

- That means, our model would make the exact same prediction for every nearby word.
- This is more easily resolved in our transformer model, so we won't worry about it for now.

Concept 21

Our **probability** calculation in skip-gram is **independent** of the **skip distance** j between words w_t and w_{t+j}.

- All words within the **context window** have the **same probability distribution**.

Another problem: if "more **similar**" means "more likely to **co-occur**", doesn't that suggest that we would expect a word to appear with itself, really often?

- This would be true for every word: nothing can be more similar to a vector than itself, after all.
- Our solution is to just **exclude** w_t from predictions about nearby words.

Concept 22

The most similar word to w_t, is **itself**!

- So, we often **exclude** w_t from predictions.

Another problem: our objective function $J(\theta)$ includes a logarithm. To optimize θ, we'd need to compute its **derivative**.

- This becomes really expensive, especially when our vocabulary can have millions of words.
- Our solution is to "**prune**"/remove a lot of words from our probability calculation.

We can predict in advance, that some words don't need to be included.

Concept 23

Skip-gram can become expensive to train, when the **vocabulary** becomes too large.

- So, we **prune** some unlikely words in our vocabulary, to speed up our **predictions**.