## Jay Alammar (/)

Visualizing machine learning one concept at a time.
@JayAlammar (https://twitter.com/JayAlammar) on Twitter. YouTube Channel
(https://www.youtube.com/channel/UCmOwsoHty5PrmE-3QhUBfPQ)

Blog (/)    About (/about)

# The Illustrated Word2vec

Discussions: Hacker News (347 points, 37 comments) (https://news.ycombinator.com/item?id=19498356), Reddit r/MachineLearning (151 points, 19 comments) (https://www.reddit.com/r/MachineLearning/comments/b60jtg/p_the_illustrated_word2vec/)
Translations: Chinese (Simplified) (https://mp.weixin.qq.com/s?
__biz=MjM5MTQzNzU2NA==&mid=2651669277&idx=2&sn=bc8f0590f9e340c1f1359982726c5a30&chksm=bd4c648e8a3bed9817f30c5a512e79fe0cc6fbc58544f97c857c3
French (https://lbourdois.github.io/blog/nlp/word_embedding/), Korean (https://databreak.netlify.com/2019-04-25-illustrated_word2vec/), Portuguese
(https://pessoalex.wordpress.com/2019/03/29/o-word2vec-ilustrado/), Russian (https://habr.com/ru/post/446530/)
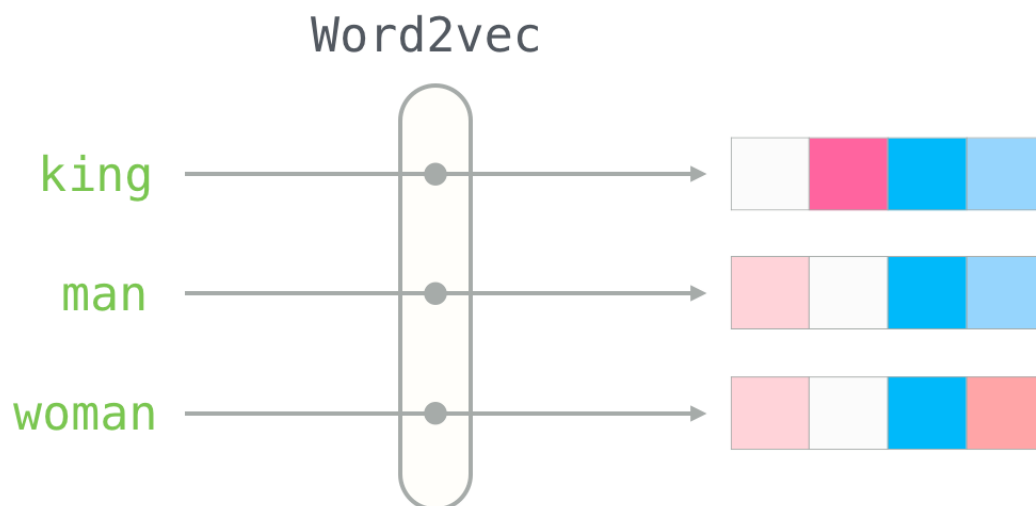


> *"**There is in all things a pattern that is part of our universe. It has symmetry, elegance, and grace** - those qualities you find always in that which the true artist captures. You can find it in the turning of the seasons, in the way sand trails along a ridge, in the branch clusters of the creosote bush or the pattern of its leaves.*
>
> *We try to copy these patterns in our lives and our society, seeking the rhythms, the dances, the forms that comfort. Yet, it is possible to see peril in the finding of ultimate perfection. It is clear that the ultimate pattern contains it own fixity. In such perfection, all things move toward death." ~ Dune (1965)*

I find the concept of embeddings to be one of the most fascinating ideas in machine learning. If you've ever used Siri, Google Assistant, Alexa, Google Translate, or even smartphone keyboard with next-word prediction, then chances are you've benefitted from this idea that has become central to Natural Language Processing models. There has been quite a development over the last couple of decades in using embeddings for neural models (Recent developments include contextualized word embeddings leading to cutting-edge models like BERT (https://jalammar.github.io/illustrated-bert/) and GPT2).

Word2vec is a method to efficiently create word embeddings and has been around since 2013. But in addition to its utility as a word-embedding method, some of its concepts have been shown to be effective in creating recommendation engines and making sense of sequential data even in commercial, non-language tasks. Companies like Airbnb (https://www.kdd.org/kdd2018/accepted-papers/view/real-time-personalization-using-embeddings-for-

search-ranking-at-airbnb), Alibaba (https://www.kdd.org/kdd2018/accepted-papers/view/billion-scale-commodity-embedding-for-e-commerce-recommendation-in-alibaba), Spotify (https://www.slideshare.net/AndySloane/machine-learning-spotify-madison-big-data-meetup), and Anghami (https://towardsdatascience.com/using-word2vec-for-music-recommendations-bb9649ac2484) have all benefitted from carving out this brilliant piece of machinery from the world of NLP and using it in production to empower a new breed of recommendation engines.

In this post, we'll go over the concept of embedding, and the mechanics of generating embeddings with word2vec. But let's start with an example to get familiar with using vectors to represent things. Did you know that a list of five numbers (a vector) can represent so much about your personality?

# Personality Embeddings: What are you like?

> "I give you the desert chameleon, whose ability to blend itself into the background tells you all you need to know about the roots of ecology and the foundations of a personal identity" ~Children of Dune

On a scale of 0 to 100, how introverted/extraverted are you (where 0 is the most introverted, and 100 is the most extraverted)? Have you ever taken a personality test like MBTI – or even better, the Big Five Personality Traits (https://en.wikipedia.org/wiki/Big_Five_personality_traits) test? If you haven't, these are tests that ask you a list of questions, then score you on a number of axes, introversion/extraversion being one of them.



Example of the result of a Big Five Personality Trait test. It can really tell you a lot about yourself and is shown to have predictive ability in academic (http://psychology.okstate.edu/faculty/jgrice/psyc4333/FiveFactor_GPAPaper.pdf), personal (https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1744-6570.1999.tb00174.x), and professional success (https://www.massgeneral.org/psychiatry/assets/published_papers/soldz-1999.pdf). This (https://projects.fivethirtyeight.com/personality-quiz/) is one place to find your results.
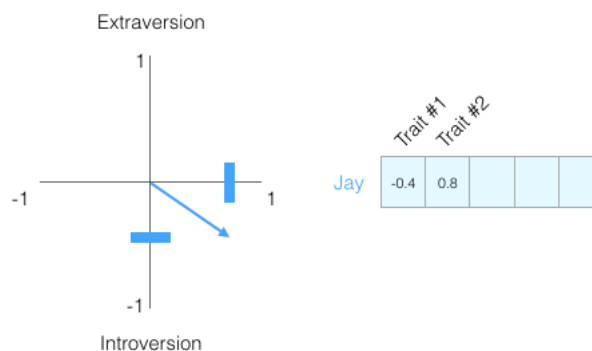
Imagine I've scored 38/100 as my introversion/extraversion score. we can plot that in this way:
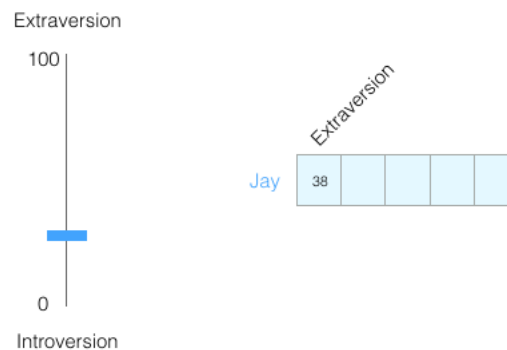


Let's switch the range to be from -1 to 1:
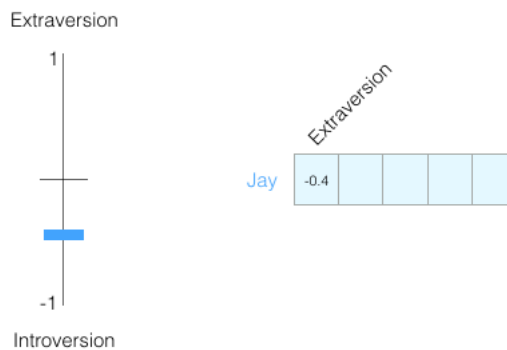
How well do you feel you know a person knowing only this one piece of information about them? Not much. People are complex. So let's add another dimension – the score of one other trait from the test.



We can represent the two dimensions as a point on the graph, or better yet, as a vector from the origin to that point. We have incredible tools to deal with vectors that will come in handy very shortly.

I've hidden which traits we're plotting just so you get used to not knowing what each dimension represents – but still getting a lot of value from the vector representation of a person's personality.

We can now say that this vector partially represents my personality. The usefulness of such representation comes when you want to compare two other people to me. Say I get hit by a `bus` and I need to be replaced by someone with a similar personality. In the following figure, which of the two people is more similar to me?
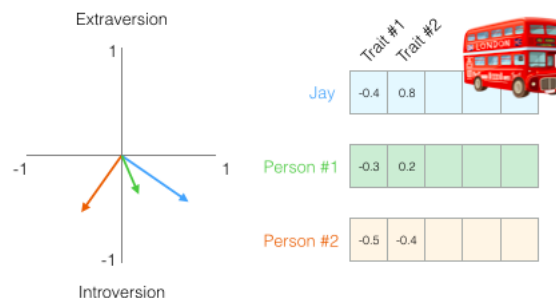


When dealing with vectors, a common way to calculate a similarity score is cosine_similarity (https://en.wikipedia.org/wiki/Cosine_similarity):



Person #1 is more similar to me in personality. Vectors pointing at the same direction (length plays a role as well) have a higher cosine similarity score.

Yet again, two dimensions aren't enough to capture enough information about how different people are. Decades of psychology research have led to five major traits (and plenty of sub-traits). So let's use all five dimensions in our comparison:



The problem with five dimensions is that we lose the ability to draw neat little arrows in two dimensions. This is a common challenge in machine learning where we often have to think in higher-dimensional space. The good thing is, though, that cosine_similarity still works. It works with any number of dimensions:



cosine_similarity works for any number of dimensions. These are much better scores because they're calculated based on a higher resolution representation of the things being compared.

At the end of this section, I want us to come out with two central ideas:

1. We can represent people (and things) as vectors of numbers (which is great for machines!).
2. We can easily calculate how similar vectors are to each other.



# Word Embeddings

> *"The gift of words is the gift of deception and illusion"* ~Children of Dune

With this understanding, we can proceed to look at trained word-vector examples (also called word embeddings) and start looking at some of their interesting properties.

This is a word embedding for the word "king" (GloVe vector trained on Wikipedia):

```
[ 0.50451 , 0.68607 , −0.59517 , −0.022801, 0.60046 , −0.13498 , −0.08813 , 0.47377 , −0.61798 , −0.31012 ,
−0.076666, 1.493 , −0.034189, −0.98173 , 0.68229 , 0.81722 , −0.51874 , −0.31503 , −0.55809 , 0.66421 , 0.1961
, −0.13495 , −0.11476 , −0.30344 , 0.41177 , −2.223 , −1.0756 , −1.0783 , −0.34354 , 0.33505 , 1.9927 ,
−0.04234 , −0.64319 , 0.71125 , 0.49159 , 0.16754 , 0.34344 , −0.25663 , −0.8523 , 0.1661 , 0.40102 , 1.1685 ,
−1.0137 , −0.21585 , −0.15155 , 0.78321 , −0.91241 , −1.6106 , −0.64426 , −0.51042 ]
```

It's a list of 50 numbers. We can't tell much by looking at the values. But let's visualize it a bit so we can compare it other word vectors. Let's put all these numbers in one row:



Let's color code the cells based on their values (red if they're close to 2, white if they're close to 0, blue if they're close to -2):
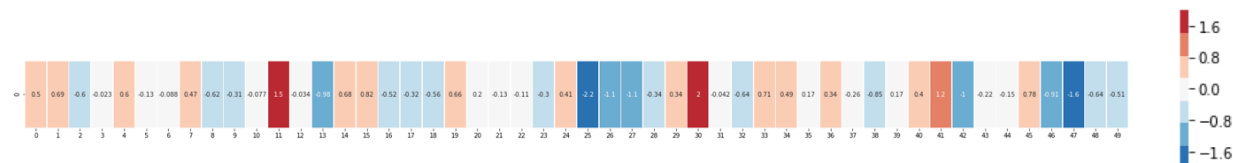


We'll proceed by ignoring the numbers and only looking at the colors to indicate the values of the cells. Let's now contrast "King" against other words:



See how "Man" and "Woman" are much more similar to each other than either of them is to "king"? This tells you something. These vector representations capture quite a bit of the information/meaning/associations of these words.

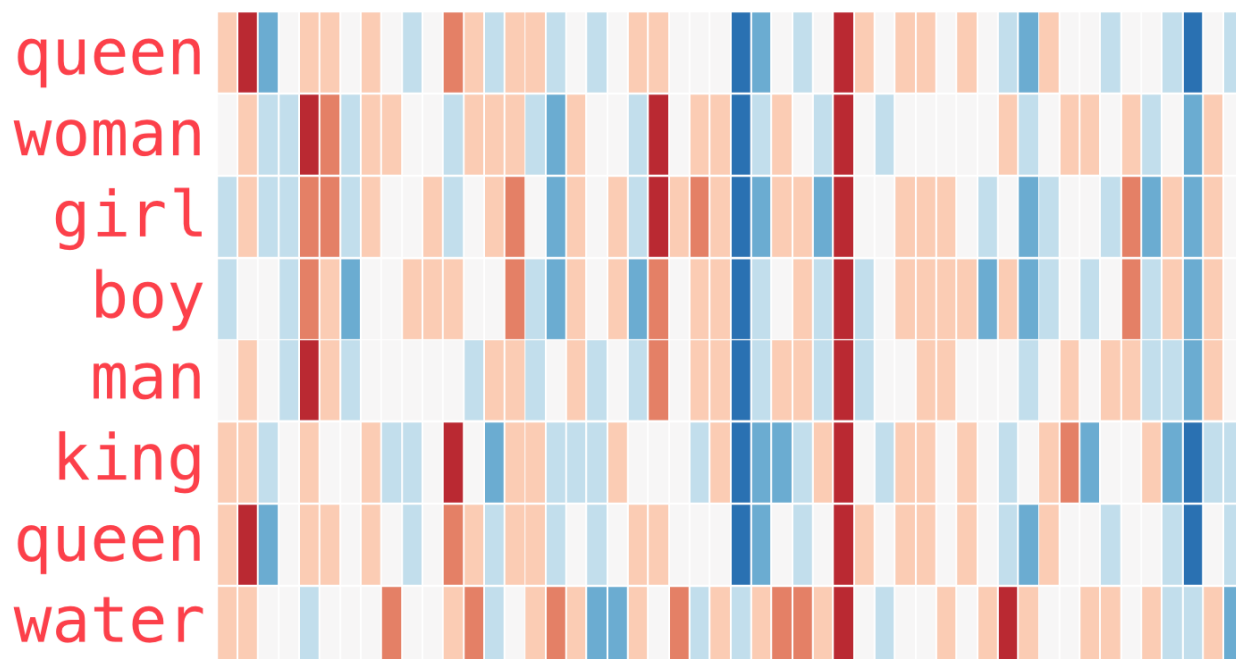Here's another list of examples (compare by vertically scanning the columns looking for columns with similar colors):



A few things to point out:

1. There's a straight red column through all of these different words. They're similar along that dimension (and we don't know what each dimensions codes for)
2. You can see how "woman" and "girl" are similar to each other in a lot of places. The same with "man" and "boy"
3. "boy" and "girl" also have places where they are similar to each other, but different from "woman" or "man". Could these be coding for a vague conception of youth? possible.
4. All but the last word are words representing people. I added an object (water) to show the differences between categories. You can, for example, see that blue column going all the way down and stopping before the embedding for "water".
5. There are clear places where "king" and "queen" are similar to each other and distinct from all the others. Could these be coding for a vague concept of royalty?

## Analogies

> *"Words can carry any burden we wish. All that's required is agreement and a tradition upon which to build." ~God Emperor of Dune*

The famous examples that show an incredible property of embeddings is the concept of analogies. We can add and subtract word embeddings and arrive at interesting results. The most famous example is the formula: "king" - "man" + "woman":



Using the Gensim (https://radimrehurek.com/gensim/) library in python, we can add and subtract word vectors, and it would find the most similar words to the resulting vector. The image shows a list of the most similar words, each with its cosine similarity.

We can visualize this analogy as we did previously:
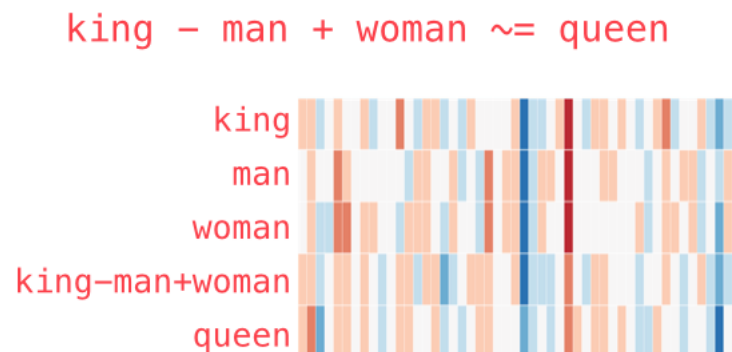


The resulting vector from "king-man+woman" doesn't exactly equal "queen", but "queen" is the closest word to it from the 400,000 word embeddings we have in this collection.

Now that we've looked at trained word embeddings, let's learn more about the training process. But before we get to word2vec, we need to look at a conceptual parent of word embeddings: the neural language model.

## Language Modeling

> *"The prophet is not diverted by illusions of past, present and future. **The fixity of language determines such linear distinctions.** Prophets hold a key to the lock in a language.*
>
> *This is not a mechanical universe. The linear progression of events is imposed by the observer. Cause and effect? That's not it at all. **The prophet utters fateful words.** You glimpse a thing "destined to occur." But the prophetic instant releases something of infinite portent and power. The universe undergoes a ghostly shift." ~God Emperor of Dune*

If one wanted to give an example of an NLP application, one of the best examples would be the next-word prediction feature of a smartphone keyboard. It's a feature that billions of people use hundreds of times every day.



Next-word prediction is a task that can be addressed by a *language model*. A language model can take a list of words (let's say two words), and attempt to predict the word that follows them.

In the screenshot above, we can think of the model as one that took in these two green words (`thou shalt`) and returned a list of suggestions ("not" being the one with the highest probability):

input/feature #1                    input/feature #2                    output/label

# Thou    shalt    _____

We can think of the model as looking like this black box:

Input                                                                                    Output
Features                                                                              Prediction



Thou  ────▶   ┌─────────────────────────────────┐
              │      Trained Language Model       │  ────▶   not
              │             **Task:**             │
shalt ────▶   │      Predict the next word        │
              └─────────────────────────────────┘
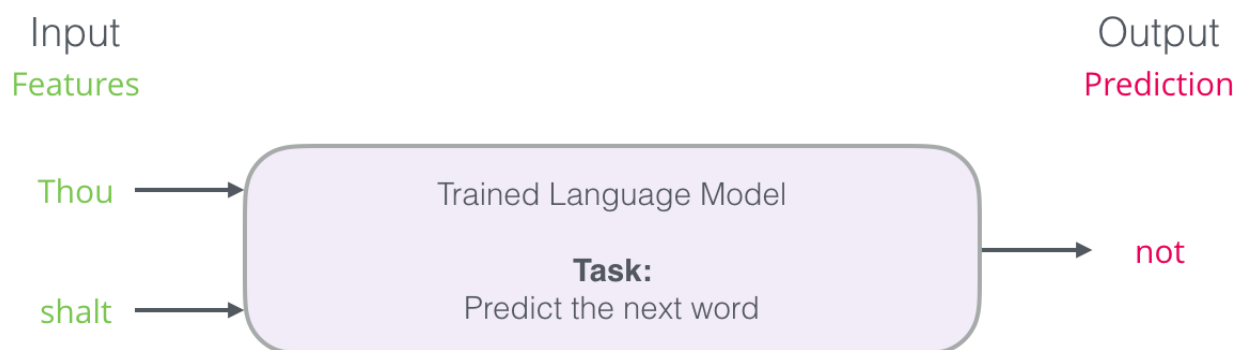
But in practice, the model doesn't output only one word. It actually outputs a probability score for all the words it knows (the model's "vocabulary", which can range from a few thousand to over a million words). The keyboard application then has to find the words with the highest scores, and present those to the user.

Input                                                                                   Output
Features                                                                                Prediction
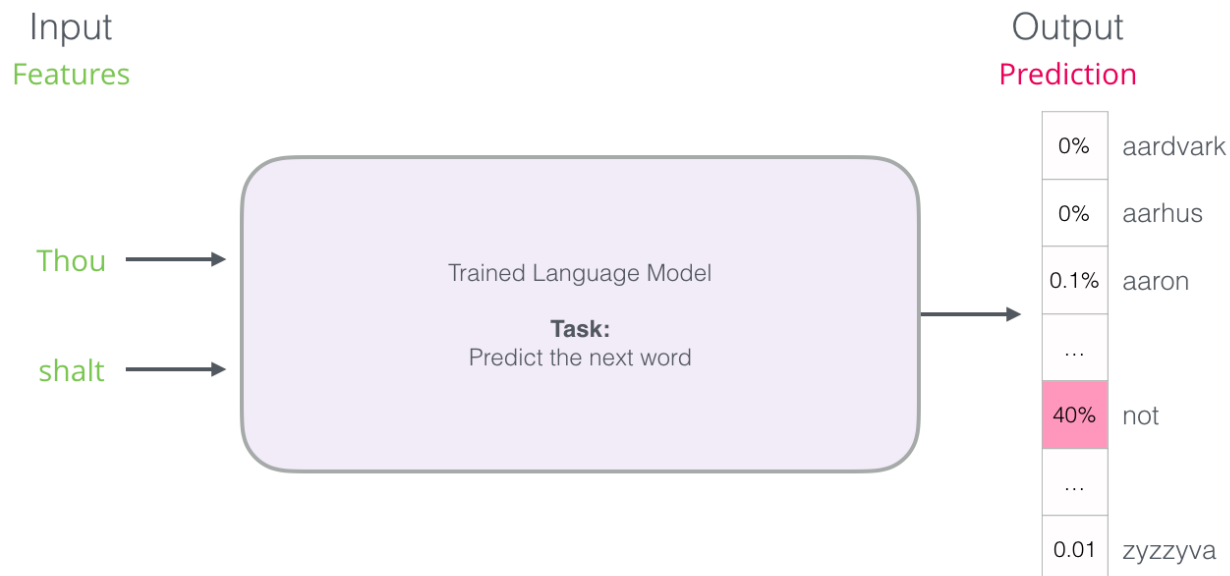


The output of the neural language model is a probability score for all the words the model knows. We're referring to the probability as a percentage here, but 40% would actually be represented as 0.4 in the output vector.

After being trained, early neural language models (Bengio 2003 (http://www.jmlr.org/papers/volume3/bengio03a/bengio03a.pdf)) would calculate a prediction in three steps:
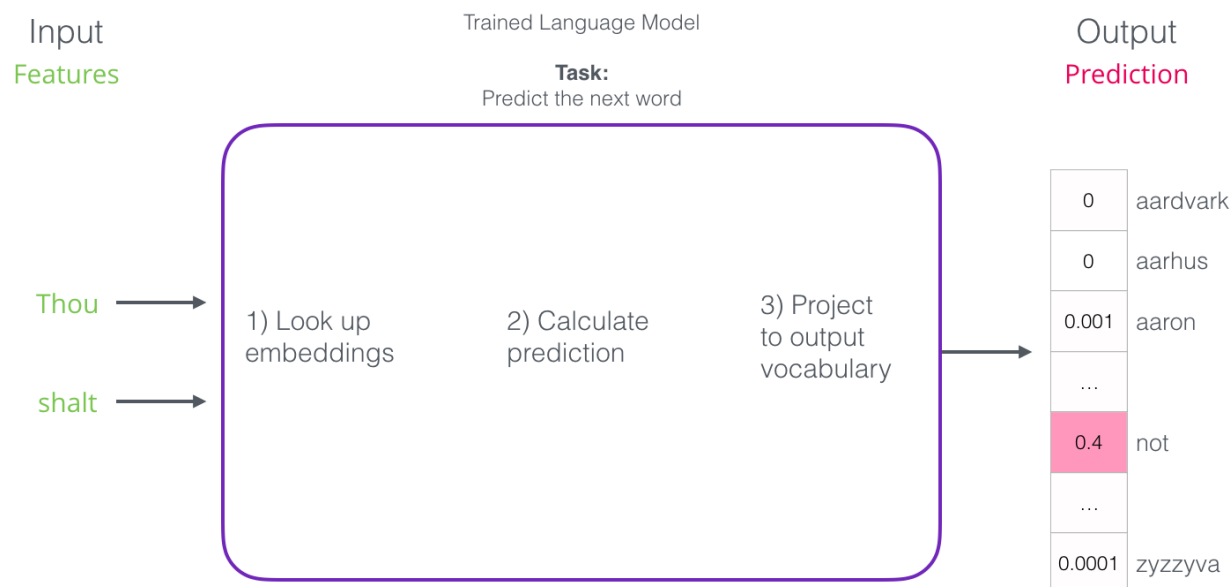


The first step is the most relevant for us as we discuss embeddings. One of the results of the training process was this matrix that contains an embedding for each word in our vocabulary. During prediction time, we just look up the embeddings of the input word, and use them to calculate the prediction:
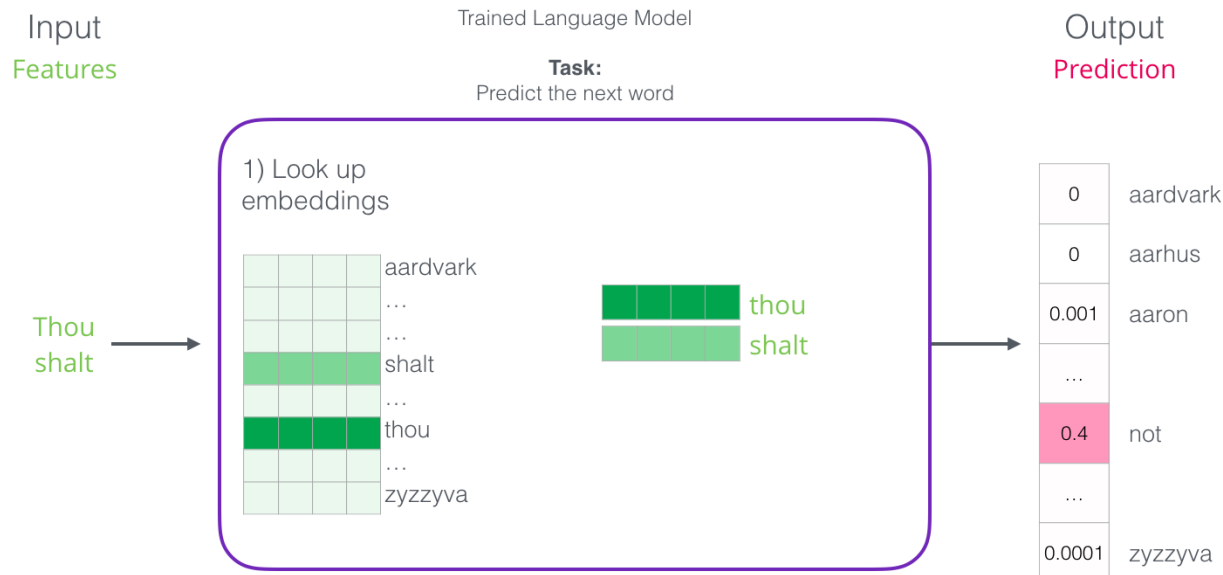
Let's now turn to the training process to learn more about how this embedding matrix was developed.

# Language Model Training

> *"A process cannot be understood by stopping it. Understanding must move with the flow of the process, must join it and flow with it." ~Dune*

Language models have a huge advantage over most other machine learning models. That advantage is that we are able to train them on running text – which we have an abundance of. Think of all the books, articles, Wikipedia content, and other forms of text data we have lying around. Contrast this with a lot of other machine learning models which need hand-crafted features and specially-collected data.

> *"You shall know a word by the company it keeps" J.R. Firth*

Words get their embeddings by us looking at which other words they tend to appear next to. The mechanics of that is that

1. We get a lot of text data (say, all Wikipedia articles, for example). then
2. We have a window (say, of three words) that we slide against all of that text.
3. The sliding window generates training samples for our model
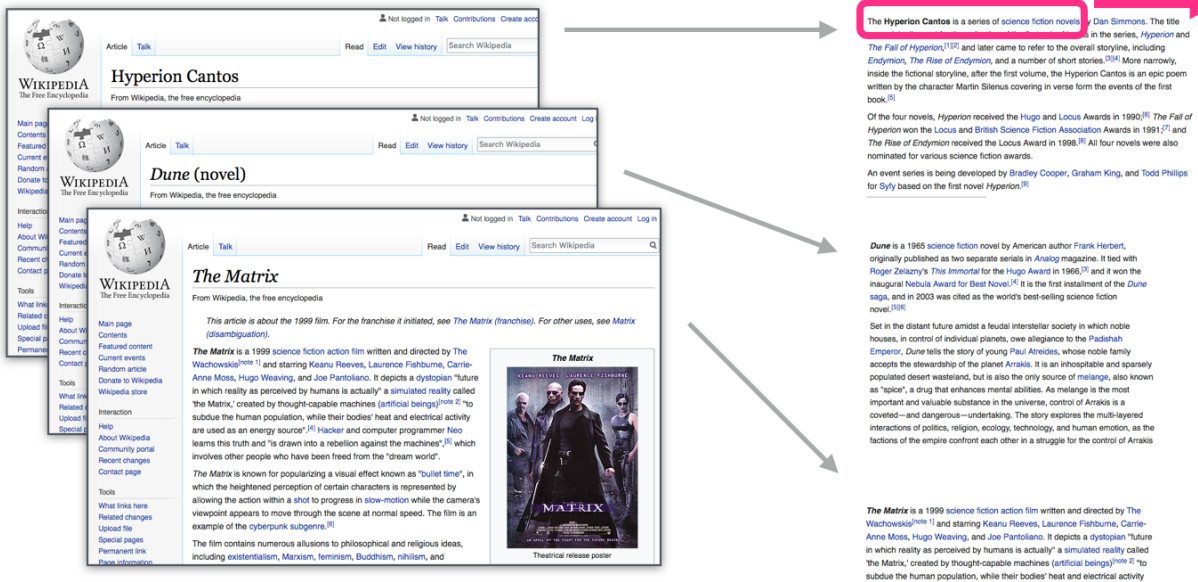
As this window slides against the text, we (virtually) generate a dataset that we use to train a model. To look exactly at how that's done, let's see how the sliding window processes this phrase:

> *"Thou shalt not make a machine in the likeness of a human mind"* ~Dune

When we start, the window is on the first three words of the sentence:
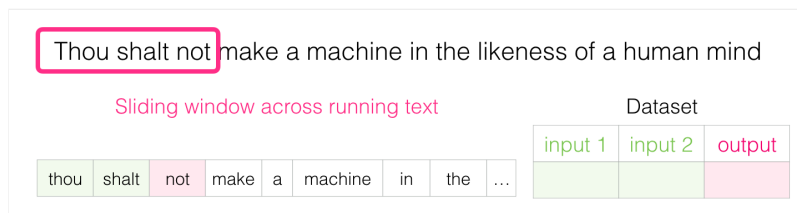


We take the first two words to be features, and the third word to be a label:
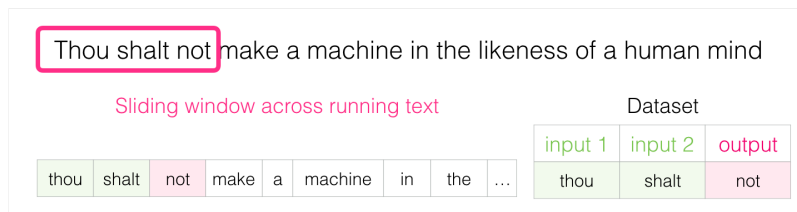


We now have generated the first sample in the dataset we can later use to train a language model.

We then slide our window to the next position and create a second sample:

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

Sliding window across running text

Dataset

| thou | shalt | not | make | a | machine | in | the | … |
|------|-------|-----|------|---|---------|-----|-----|---|
| thou | shalt | not | make | a | machine | in | the | |

| input 1 | input 2 | output |
|---------|---------|--------|
| thou | shalt | not |
| shalt | not | make |

An the second example is now generated.

And pretty soon we have a larger dataset of which words tend to appear after different pairs of words:

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

Sliding window across running text

Dataset

| thou | shalt | not | make | a | machine | in | the | … |
|------|-------|-----|------|---|---------|-----|-----|---|
| thou | shalt | not | make | a | machine | in | the | |
| thou | shalt | not | make | a | machine | in | the | |
| thou | shalt | not | make | a | machine | in | the | |
| thou | shalt | not | make | a | machine | in | the | |

| input 1 | input 2 | output |
|---------|---------|--------|
| thou | shalt | not |
| shalt | not | make |
| not | make | a |
| make | a | machine |
| a | machine | in |

In practice, models tend to be trained while we're sliding the window. But I find it clearer to logically separate the "dataset generation" phase from the training phase. Aside from neural-network-based approaches to language modeling, a technique called N-grams was commonly used to train language models (see: Chapter 3 of Speech and Language Processing (http://web.stanford.edu/~jurafsky/slp3/)). To see how this switch from N-grams to neural models reflects on real-world products, here's a 2015 blog post from Swiftkey (https://blog.swiftkey.com/neural-networks-a-meaningful-leap-for-mobile-typing/), my favorite Android keyboard, introducing their neural language model and comparing it with their previous N-gram model. I like this example because it shows you how the algorithmic properties of embeddings can be described in marketing speech.

## Look both ways

> "Paradox is a pointer telling you to look beyond it. If paradoxes bother you, that betrays your deep desire for absolutes. The relativist treats a paradox merely as interesting, perhaps amusing or even, dreadful thought, educational." ~God Emperor of Dune

Knowing what you know from earlier in the post, fill in the blank:

Jay was hit by a _____

The context I gave you here is five words before the blank word (and an earlier mention of "bus"). I'm sure most people would guess the word bus goes into the blank. But what if I gave you one more piece of information – a word after the blank, would that change your answer?

Jay was hit by a _____ bus

This completely changes what should go in the blank. the word red is now the most likely to go into the blank. What we learn from this is the words both before and after a specific word carry informational value. It turns out that accounting for both directions (words to the left and to the right of the word we're guessing) leads to better word embeddings. Let's see how we can adjust the way we're training the model to account for this.

## Skipgram

> *"Intelligence takes chance with limited data in an arena where mistakes are not only possible but also necessary." ~Chapterhouse: Dune*

Instead of only looking two words before the target word, we can also look at two words after it.

Jay was hit by a _____ bus in...

| by | a | red | bus | in |
|----|---|-----|-----|----|

If we do this, the dataset we're virtually building and training the model against would look like this:

| input 1 | input 2 | input 3 | input 4 | output |
|---------|---------|---------|---------|--------|
| by | a | bus | in | red |

This is called a **Continuous Bag of Words** architecture and is described in one of the word2vec papers (https://arxiv.org/pdf/1301.3781.pdf) [pdf]. Another architecture that also tended to show great results does things a little differently.

Instead of guessing a word based on its context (the words before and after it), this other architecture tries to guess neighboring words using the current word. We can think of the window it slides against the training text as looking like this:

Jay was hit by a red bus in...

The word in the green slot would be the input word, each pink box would be a possible output.

The pink boxes are in different shades because this sliding window actually creates four separate samples in our training dataset:

Jay was hit by a red bus in...

| by | a | red | bus | in |
|----|---|-----|-----|----|

| input | output |
|-------|--------|
| red | by |
| red | a |
| red | bus |
| red | in |

This method is called the **skipgram** architecture. We can visualize the sliding window as doing the following:

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

| thou | shalt | not | make | a | machine | in | the | ... |
|------|-------|-----|------|---|---------|----|----|-----|

| input word | target word |
|------------|-------------|
|            |             |
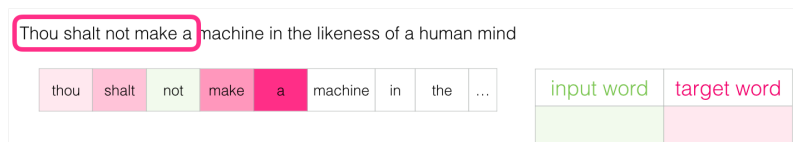
This would add these four samples to our training dataset:

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

| thou | shalt | not | make | a | machine | in | the | ... |

| input word | target word |
| --- | --- |
| not | thou |
| not | shalt |
| not | make |
| not | a |

We then slide our window to the next position:

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

| thou | shalt | not | make | a | machine | in | the | ... |

| thou | shalt | not | make | a | machine | in | the | ... |

| input word | target word |
| --- | --- |
| not | thou |
| not | shalt |
| not | make |
| not | a |

Which generates our next four examples:

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

| thou | shalt | not | make | a | machine | in | the | ... |

| thou | shalt | not | make | a | machine | in | the | ... |

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

A couple of positions later, we have a lot more examples:

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

| thou | shalt | not | make | a | machine | in | the | … |
|------|-------|-----|------|---|---------|----|----|----|

| thou | shalt | not | make | a | machine | in | the | … |
|------|-------|-----|------|---|---------|----|----|----|

| thou | shalt | not | make | a | machine | in | the | … |
|------|-------|-----|------|---|---------|----|----|----|

| thou | shalt | not | make | a | machine | in | the | … |
|------|-------|-----|------|---|---------|----|----|----|

| thou | shalt | not | make | a | machine | in | the | … |
|------|-------|-----|------|---|---------|----|----|----|

| 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 |

# Revisiting the training process

> "Muad'Dib learned rapidly because his first training was in how to learn. And the first lesson of all was the basic trust that he could learn. It's shocking to find how many people do not believe they can learn, and how many more believe learning to be difficult." ~Dune

Now that we have our skipgram training dataset that we extracted from existing running text, let's glance at how we use it to train a basic neural language model that predicts the neighboring word.

| 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 |
| | |

not →

Untrained Model

**Task:**
Predict neighbouring word
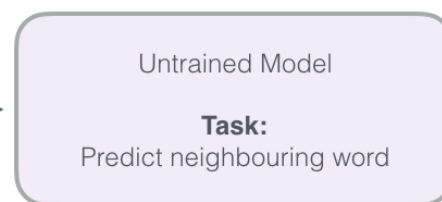
We start with the first sample in our dataset. We grab the feature and feed to the untrained model asking it to predict an appropriate neighboring word.
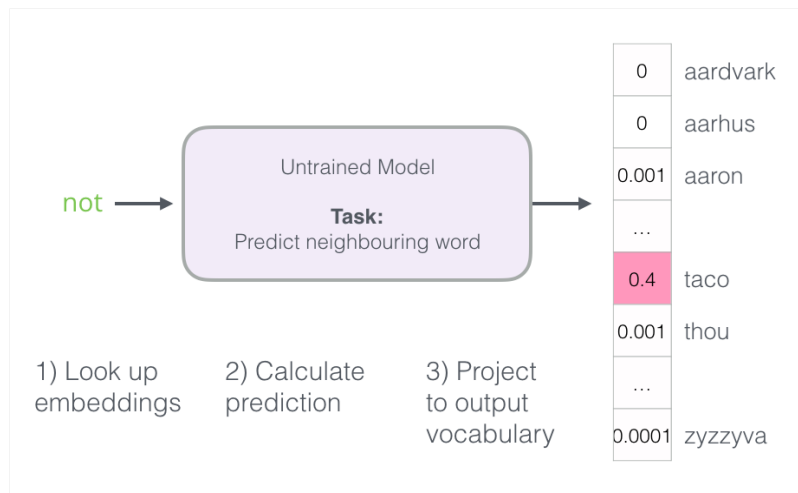
The model conducts the three steps and outputs a prediction vector (with a probability assigned to each word in its vocabulary). Since the model is untrained, it's prediction is sure to be wrong at this stage. But that's okay. We know what word it should have guessed – the label/output cell in the row we're currently using to train the model:
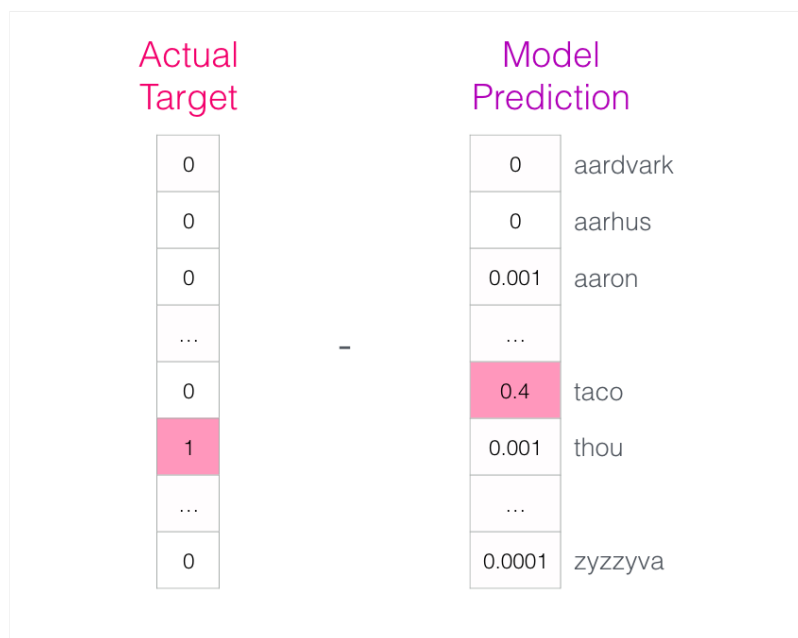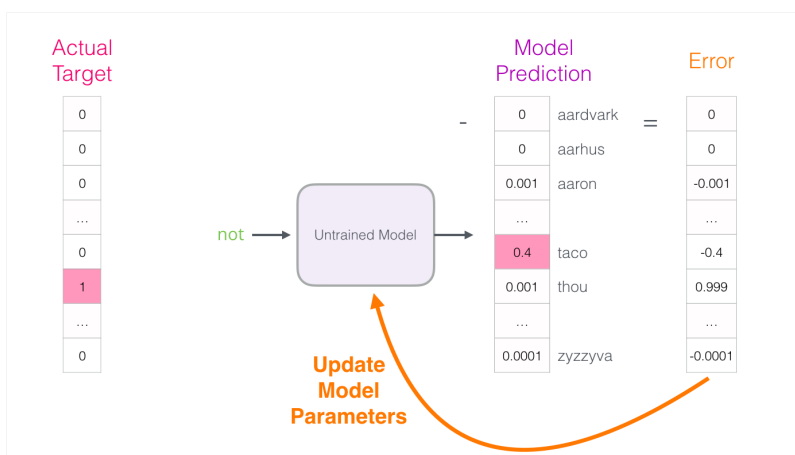


The 'target vector' is one where the target word has the probability 1, and all other words have the probability 0.

How far off was the model? We subtract the two vectors resulting in an error vector:

This error vector can now be used to update the model so the next time, it's a little more likely to guess `thou` when it gets `not` as input.



And that concludes the first step of the training. We proceed to do the same process with the next sample in our dataset, and then the next, until we've covered all the samples in the dataset. That concludes one *epoch* of training. We do it over again for a number of epochs, and then we'd have our trained model and we can extract the embedding matrix from it and use it for any other application.

While this extends our understanding of the process, it's still not how word2vec is actually trained. We're missing a couple of key ideas.

# Negative Sampling

> *"To attempt an understanding of Muad'Dib without understanding his mortal enemies, the Harkonnens, is to attempt seeing Truth without knowing Falsehood. It is the attempt to see the Light without knowing Darkness. It cannot be." ~Dune*

Recall the three steps of how this neural language model calculates its prediction:
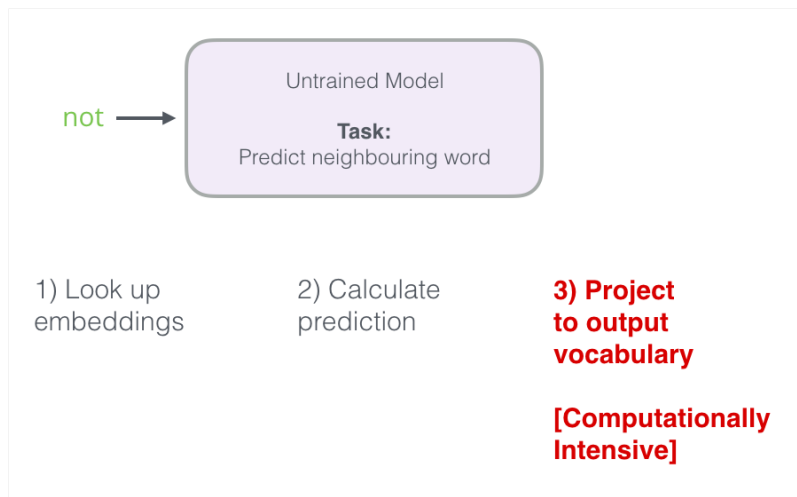
The third step is very expensive from a computational point of view – especially knowing that we will do it once for every training sample in our dataset (easily tens of millions of times). We need to do something to improve performance.

One way is to split our target into two steps:

1. Generate high-quality word embeddings (Don't worry about next-word prediction).
2. Use these high-quality embeddings to train a language model (to do next-word prediction).

We'll focus on step 1. in this post as we're focusing on embeddings. To generate high-quality embeddings using a high-performance model, we can switch the model's task from predicting a neighboring word:
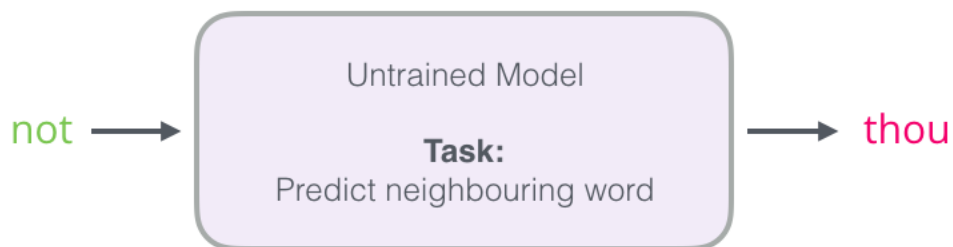


And switch it to a model that takes the input and output word, and outputs a score indicating if they're neighbors or not (0 for "not neighbors", 1 for "neighbors").

This simple switch changes the model we need from a neural network, to a logistic regression model – thus it becomes much simpler and much faster to calculate.

This switch requires we switch the structure of our dataset – the label is now a new column with values 0 or 1. They will be all 1 since all the words we added are neighbors.
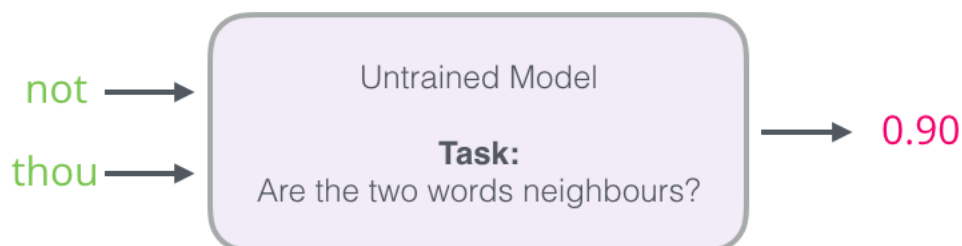
| input word | target word |
| --- | --- |
| not | thou |
| not | shalt |
| not | make |
| not | a |
| make | shalt |
| make | not |
| make | a |
| make | machine |
|  |  |

| 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** |
|  |  |  |

This can now be computed at blazing speed – processing millions of examples in minutes. But there's one loophole we need to close. If all of our examples are positive (target: 1), we open ourself to the possibility of a smartass model that always returns 1 – achieving 100% accuracy, but learning nothing and generating garbage embeddings.
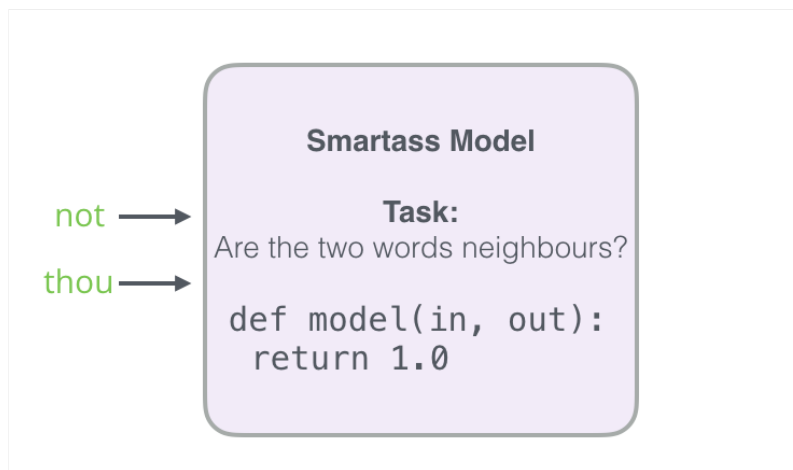


To address this, we need to introduce *negative samples* to our dataset – samples of words that are not neighbors. Our model needs to return 0 for those samples. Now that's a challenge that the model has to work hard to solve – but still at blazing fast speed.

| input word | output word | target |
|------------|-------------|--------|
| not        | thou        | **1**  |
| not        |             | **0**  |
| not        |             | **0**  |
| not        | shalt       | **1**  |
|            |             |        |
|            |             |        |
| not        | make        | **1**  |
|            |             |        |
|            |             |        |

Negative examples

For each sample in our dataset, we add **negative examples**. Those have the same input word, and a 0 label.

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

Pick randomly from vocabulary
(random sampling)

| input word | output word | target | | Word | Count | Probability |
|------------|-------------|--------|--|------|-------|-------------|
| not        | thou        | **1**  | | aardvark | | |
| not        | aaron       | **0**  | | aarhus | | |
| not        | taco        | **0**  | | aaron | | |
| not        | shalt       | **1**  | | | | |
|            |             |        | | taco | | |
|            |             |        | | thou | | |
| not        | make        | **1**  | | | | |
|            |             |        | | zyzzyva | | |
|            |             |        | | | | |

This idea is inspired by Noise-contrastive estimation (http://proceedings.mlr.press/v9/gutmann10a/gutmann10a.pdf) [pdf]. We are contrasting the actual signal (positive examples of neighboring words) with noise (randomly selected words that are not neighbors). This leads to a great tradeoff of computational and statistical efficiency.

# Skipgram with Negative Sampling (SGNS)

We have now covered two of the central ideas in word2vec: as a pair, they're called skipgram with negative sampling.

## Skipgram

| shalt | not | make | a | machine |
|-------|-----|------|---|---------|

| input | output |
|-------|--------|
| make  | shalt  |
| make  | not    |
| make  | a      |
| make  | machine|

## Negative Sampling

| input word | output word | target |
|------------|-------------|--------|
| make       | shalt       | **1**  |
| make       | aaron       | **0**  |
| make       | taco        | **0**  |

# Word2vec Training Process

> *"The machine cannot anticipate every problem of importance to humans. It is the difference between serial bits and an unbroken continuum.*
> *We have the one; machines are confined to the other." ~God Emperor of Dune*

Now that we've established the two central ideas of skipgram and negative sampling, we can proceed to look closer at the actual word2vec training process.

Before the training process starts, we pre-process the text we're training the model against. In this step, we determine the size of our vocabulary (we'll call this `vocab_size`, think of it as, say, 10,000) and which words belong to it.

At the start of the training phase, we create two matrices – an `Embedding` matrix and a `Context` matrix. These two matrices have an embedding for each word in our vocabulary (So `vocab_size` is one of their dimensions). The second dimension is how long we want each embedding to be (`embedding_size` – 300 is a common value, but we've looked at an example of 50 earlier in this post).
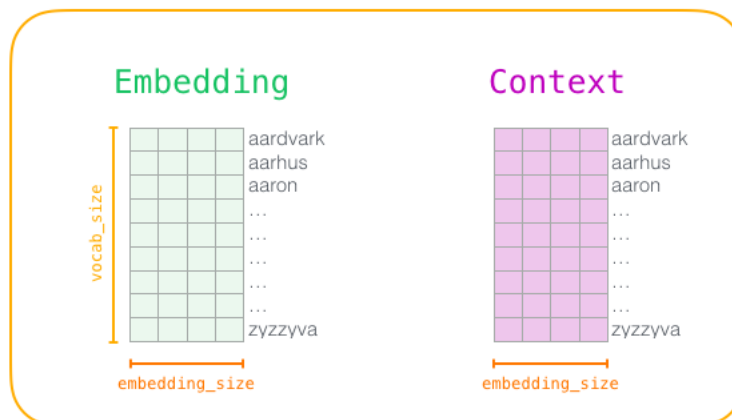


At the start of the training process, we initialize these matrices with random values. Then we start the training process. In each training step, we take one positive example and its associated negative examples. Let's take our first group:



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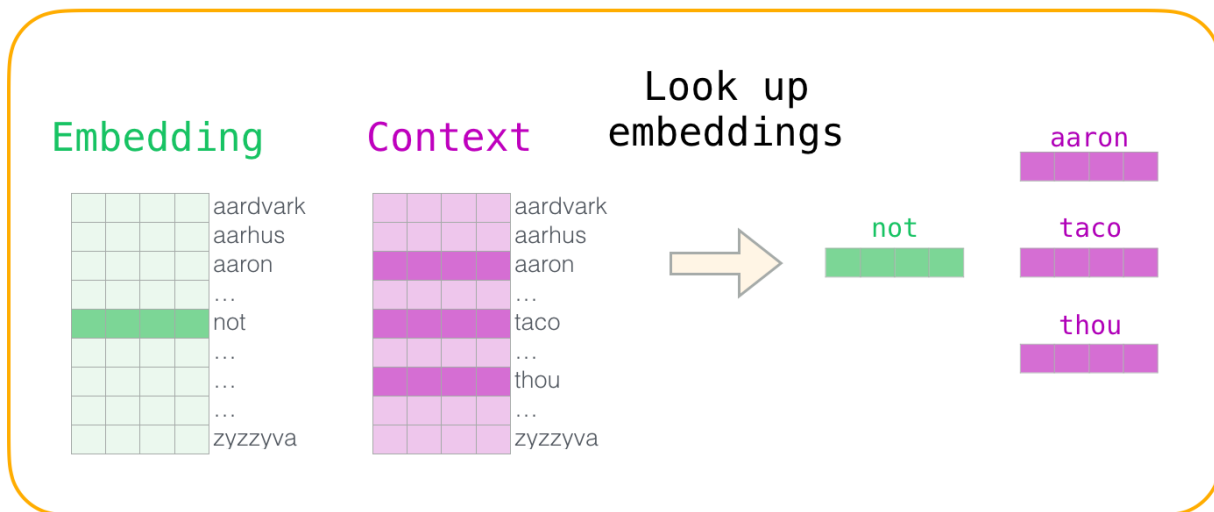
Then, we take the dot product of the input embedding with each of the context embeddings. In each case, that would result in a number, that number indicates the similarity of the input and context embeddings

| 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 (https://jalammar.github.io/feedforward-neural-networks-visual-interactive/#sigmoid-visualization), the logistic operation (https://en.wikipedia.org/wiki/Logistic_function).

| 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 |

And we can now treat the output of the sigmoid operations as the model's output for these examples. You can see that `taco` has the highest score and `aaron` still has the lowest score both before and after the sigmoid operations.

Now that the untrained model has made a prediction, and seeing as though we have an actual target label to compare against, let's calculate how much error is in the model's prediction. To do that, we just subtract the sigmoid scores 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 |



This concludes the training step. We emerge from it with slightly better embeddings for the words involved in this step (`not`, `thou`, `aaron`, and `taco`). We now proceed to our next step (the next positive sample and its associated negative samples) and do the same process again.



The embeddings continue to be improved while we cycle through our entire dataset for a number of times. We can then stop the training process, discard the `Context` matrix, and use the `Embeddings` matrix as our pre-trained embeddings for the next task.

# Window Size and Number of Negative Samples

Two key hyperparameters in the word2vec training process are the window size and the number of negative samples.
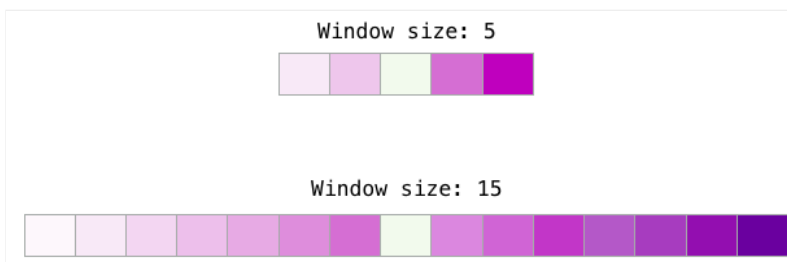


Different tasks are served better by different window sizes. One heuristic (https://youtu.be/tAxrlAVw-Tk?t=648) is that smaller window sizes (2-15) lead to embeddings where high similarity scores between two embeddings indicates that the words are *interchangeable* (notice that antonyms are often interchangable if we're only looking at their surrounding words – e.g. *good* and *bad* often appear in similar contexts). Larger window sizes (15-50, or even more) lead to embeddings where similarity is more indicative of *relatedness* of the words. In practice, you'll often have to provide annotations (https://youtu.be/ao52o9l6KGw?t=287) that guide the embedding process leading to a useful similarity sense for your task. The Gensim default window size is 5 (two words before and two words after the input word, in addition to the input word itself).

| Negative samples: 2 | | |
| --- | --- | --- |
| input word | output word | target |
| make | shalt | **1** |
| make | aaron | **0** |
| make | taco | **0** |

| Negative samples: 5 | | |
| --- | --- | --- |
| input word | output word | target |
| make | shalt | **1** |
| make | aaron | **0** |
| make | taco | **0** |
| make | finglonger | **0** |
| make | plumbus | **0** |
| make | mango | **0** |

The number of negative samples is another factor of the training process. The original paper prescribes 5-20 as being a good number of negative samples. It also states that 2-5 seems to be enough when you have a large enough dataset. The Gensim default is 5 negative samples.

# Conclusion

> *"If it falls outside your yardsticks, then you are engaged with intelligence, not with automation" ~God Emperor of Dune*

I hope that you now have a sense for word embeddings and the word2vec algorithm. I also hope that now when you read a paper mentioning "skip gram with negative sampling" (SGNS) (like the recommendation system papers at the top), that you have a better sense for these concepts. As always, all feedback is appreciated @JayAlammar (https://twitter.com/JayAlammar).

# References & Further Readings

- Distributed Representations of Words and Phrases and their Compositionality (https://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf) [pdf]
- Efficient Estimation of Word Representations in Vector Space (https://arxiv.org/pdf/1301.3781.pdf) [pdf]
- A Neural Probabilistic Language Model (http://www.jmlr.org/papers/volume3/bengio03a/bengio03a.pdf) [pdf]
- Speech and Language Processing (https://web.stanford.edu/~jurafsky/slp3/) by Dan Jurafsky and James H. Martin is a leading resource for NLP. Word2vec is tackled in Chapter 6.
- Neural Network Methods in Natural Language Processing (https://www.amazon.com/Language-Processing-Synthesis-Lectures-Technologies/dp/1627052984) by Yoav Goldberg (https://twitter.com/yoavgo) is a great read for neural NLP topics.
- Chris McCormick (http://mccormickml.com/) has written some great blog posts about Word2vec. He also just released The Inner Workings of word2vec (https://www.preview.nearist.ai/paid-ebook-and-tutorial), an E-book focused on the internals of word2vec.
- Want to read the code? Here are two options:
  - Gensim's python implementation (https://github.com/RaRe-Technologies/gensim/blob/develop/gensim/models/word2vec.py) of word2vec
  - Mikolov's original implementation in C (https://github.com/tmikolov/word2vec/blob/master/word2vec.c) – better yet, this version with detailed comments (https://github.com/chrisjmccormick/word2vec_commented/blob/master/word2vec.c) from Chris McCormick.
- Evaluating distributional models of compositional semantics (http://sro.sussex.ac.uk/id/eprint/61062/1/Batchkarov,%20Miroslav%20Manov.pdf)
- On word embeddings (http://ruder.io/word-embeddings-1/index.html), part 2 (http://ruder.io/word-embeddings-softmax/)
- Dune (https://www.amazon.com/Dune-Frank-Herbert/dp/0441172717/)

*Written on March 27, 2019*

**Subscribe to get notified about upcoming posts by email**

Email Address

Subscribe

Attribution example:
*Alammar, J (2018). The Illustrated Transformer [Blog post]. Retrieved from https://jalammar.github.io/illustrated-transformer/ (https://jalammar.github.io/illustrated-transformer/)*

Note: If you translate any of the posts, let me know so I can link your translation to the original post. My email is in the about page (/about).

(https://github.com/jalammar)          (https://www.linkedin.com/in/jalammar)
(https://www.twitter.com/jayalammar)