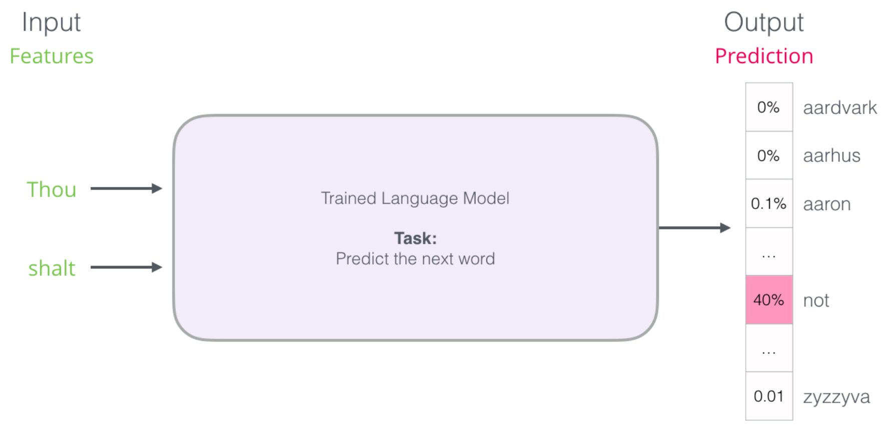
Language Model Training for NLP

Rensheng Wang,
https://sit.instructure.com/courses/43729

September 24, 2020

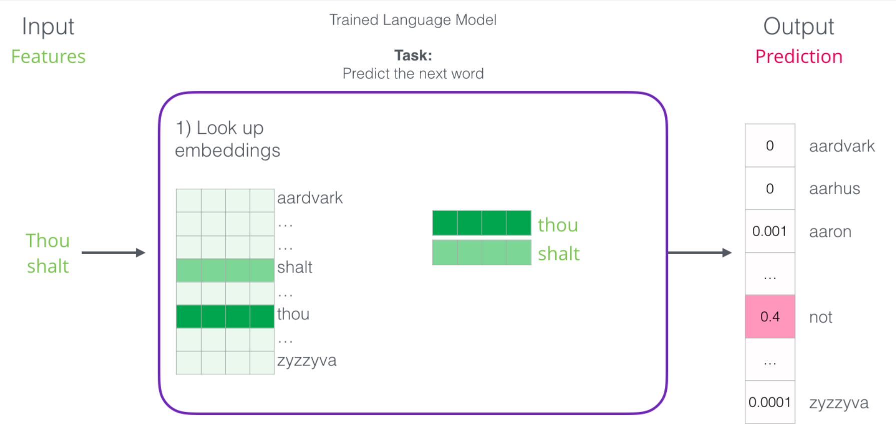
Language Modeling

- In practice, the model does not output only one word. It outputs a probability score for all the words it knows (the model's "vocabulary", 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.



Language Modeling

- 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:



Language Model Training

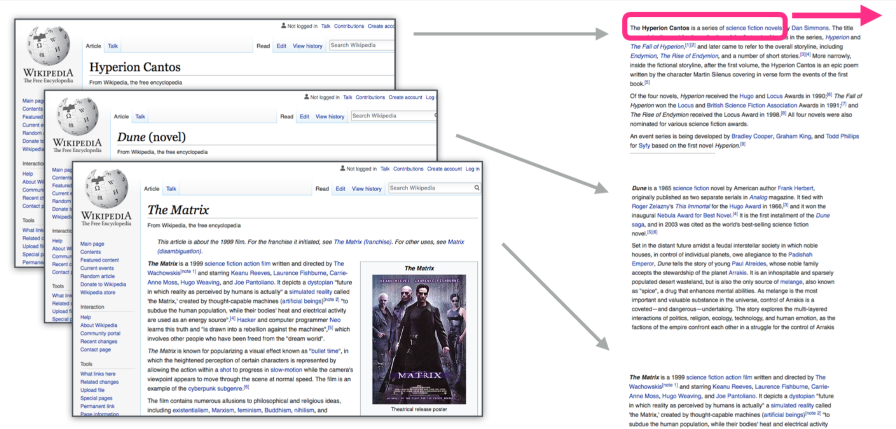
☞ A process cannot be understood by stopping it. Understanding must move with the flow of the process, must join it and flow with it.

- 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.

Language Model Training

- Words get their embeddings by us looking at which other words they tend to appear next to.
  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

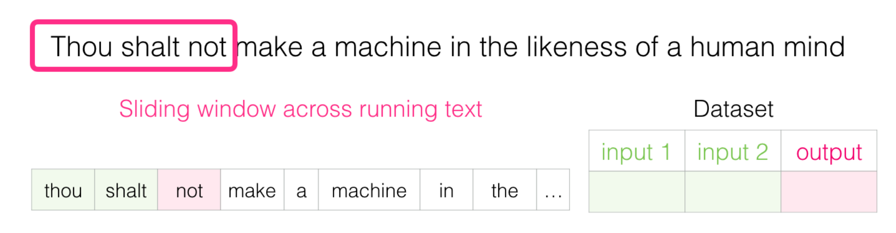


Language Model Training

- As this window slides against the text, we (virtually) generate a dataset that we use to train a model. To look exactly at how thats done, lets see how the sliding window processes this phrase:

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

- 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:

Language Model Training

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

Thou $\boxed{\text{shalt not make}}$ a machine in the likeness of a human mind

Sliding window across running text

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

Dataset

input 1	input 2	output
thou	shalt	not
shalt	not	make

Language Model Training

- 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

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	

Dataset

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

Language Model Training

- 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.

Look Both Ways

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

Jay was hit by a ______

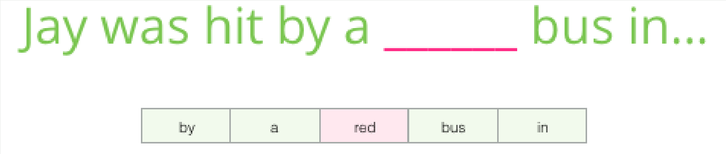
- The context I gave you here is five words before the blank word (and an earlier mention of "bus"). I'm sure most people would guess the word **"bus"** goes into the blank. But what if I gave you one more piece of information a word after the blank, would that change your answer?

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 were guessing) leads to better word embeddings.

Skipgram

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



- If we do this, the dataset were 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 [PDF].

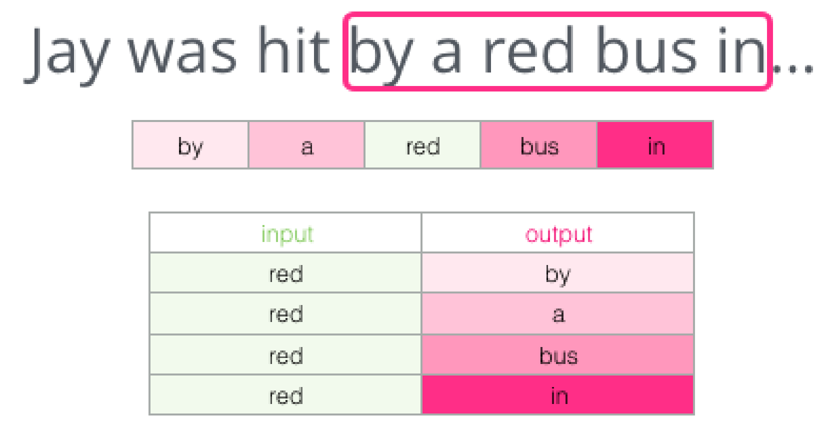
Skipgram

- 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:

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

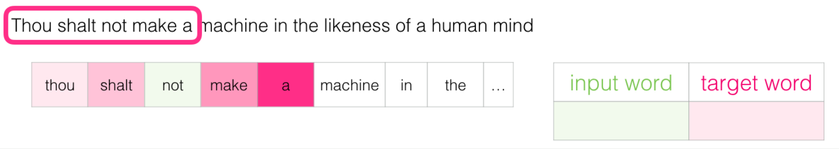
Skipgram

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

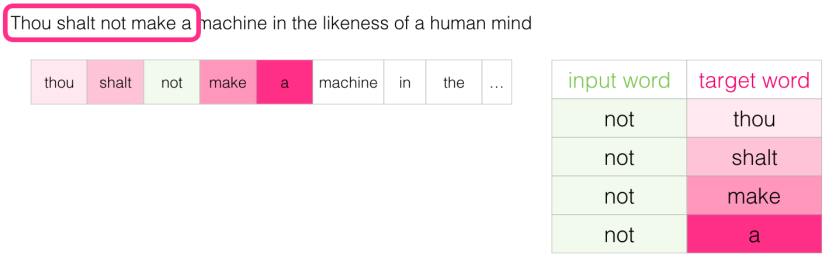


Skipgram

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



- This would add these four samples to our training dataset:



Skipgram

- 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:

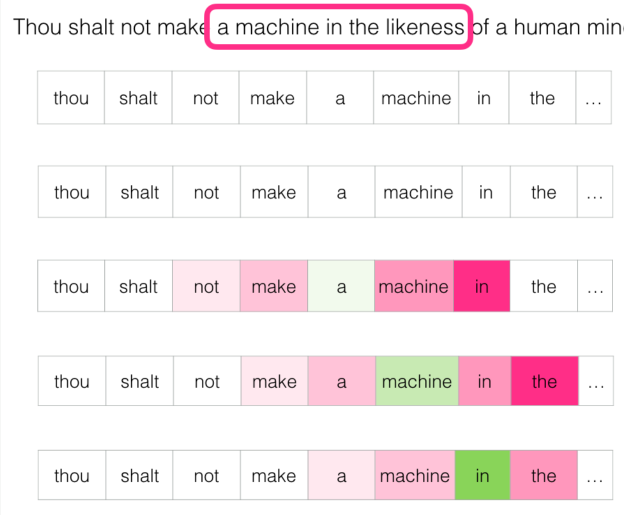
Thou shalt not make a machine in the likeness of a human mind

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

Skipgram

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

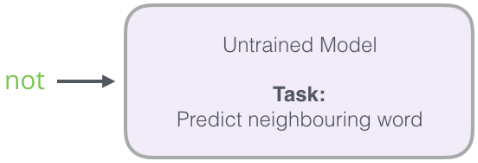


input word	target word
not	thou
not	shalt
not	make
not	a
make	shalt
make	not
make	a
make	machine
a	not
a	make
a	machine
a	in
machine	make
machine	a
machine	in
machine	the
in	a
in	machine
in	the
in	likeness

Revisiting the training process

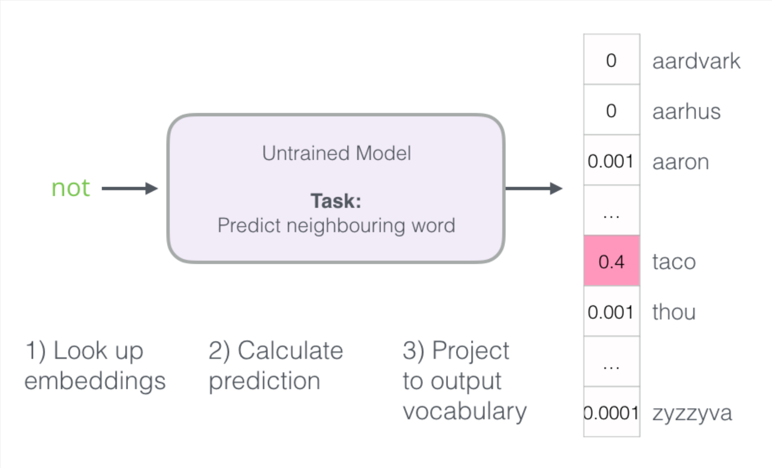
- After we have our skipgram training dataset that we extracted from existing running text, lets see how we use it to train a basic neural language model that predicts the neighboring word.

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



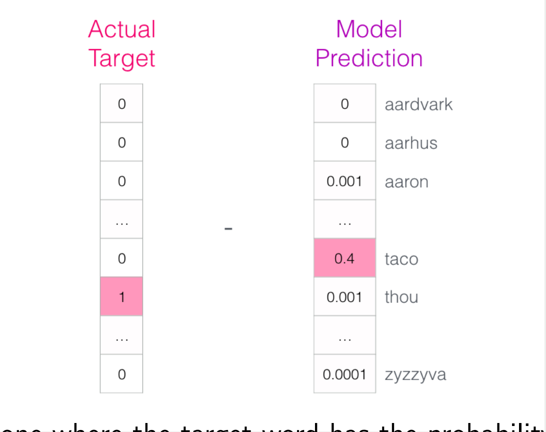
Revisiting the training process

- 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.



Revisiting the training process

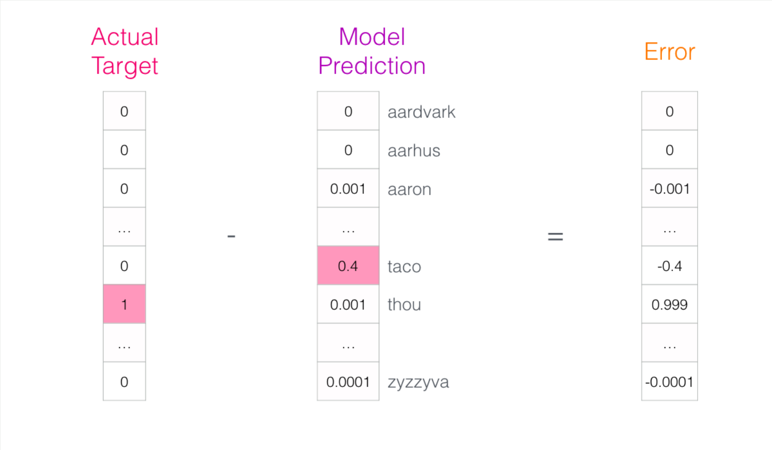
- The model conducts the three steps and outputs a prediction vector (with a probability assigned to each word in its vocabulary). Since the model is untrained, its prediction is sure to be wrong at this stage. But thats okay. We know what word it should have guessed the label/output cell in the row we are 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.

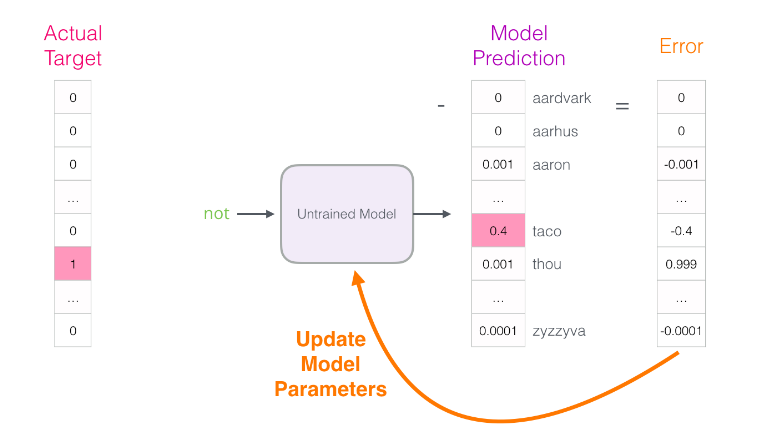
Revisiting the training process

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



Revisiting the training process

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

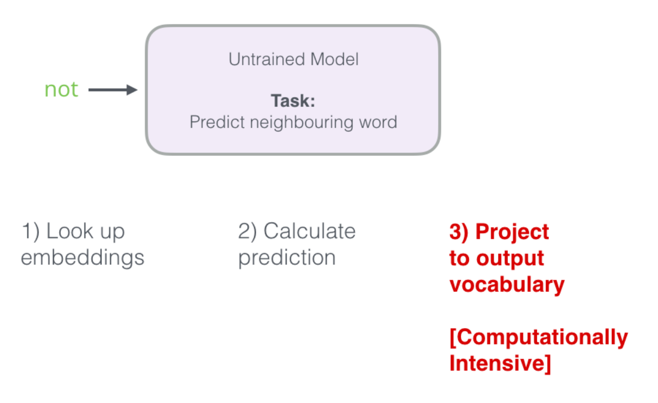


Revisiting the training process

- It 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 weve 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, its still not how word2vec is actually trained. We're missing a couple of key ideas.

Negative Sampling

- Recall the three steps of how this neural language model calculates its prediction:



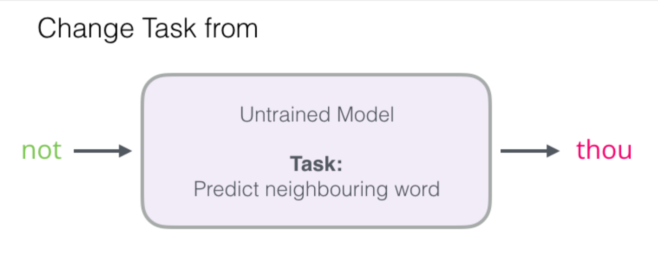
- The third step is very expensive from a computational point of view especially knowing that we will do it once for every training sample in our dataset (easily tens of millions of times). We need to do something to improve performance.

Negative Sampling

- One way is to split our target into two steps:

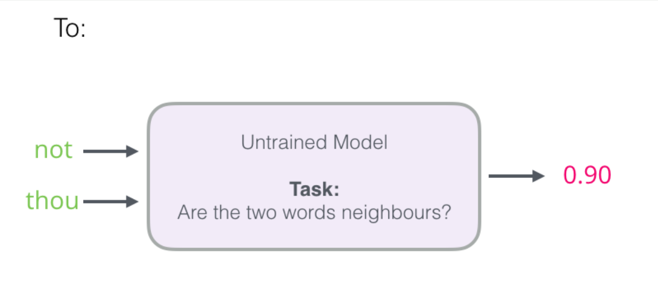
  1. Generate high-quality word embeddings (Don't worry about next-word prediction).
  2. Use these high-quality embeddings to train a language model (to do next-word prediction).

Well 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:



Negative Sampling

- 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.

Negative Sampling

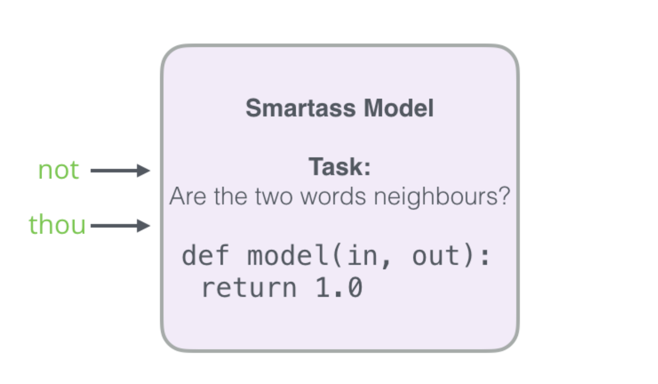
- This switch requires we switch the structure of our dataset the label is now a new column with values 0 or 1. They will be all 1 since all the words we added are neighbors.

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

Negative Sampling

- 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.



Negative Sampling

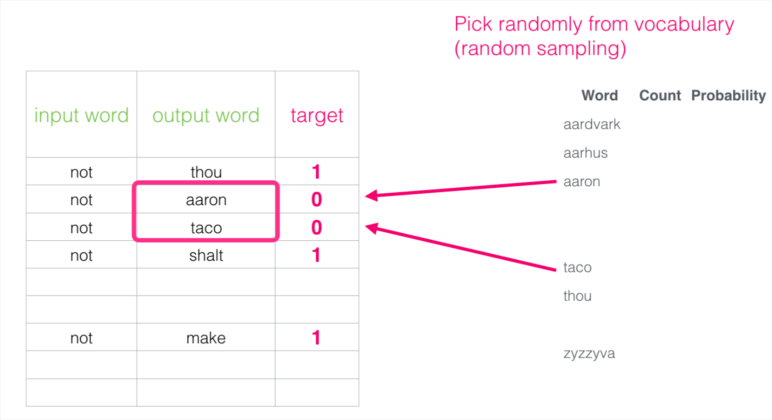
- To address this, we need to introduce negative samples to our dataset samples of words that are not neighbors. Our model needs to return 0 for those samples. Now thats a challenge that the model has to work hard to solve but still at blazing fast speed.

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

Negative examples

Negative Sampling

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



- This idea is inspired by Noise-contrastive estimation [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, theyre called skipgram with negative sampling.

Skipgram

shalt	not	make	a	machine

input	output
make	shalt
make	not
make	a
make	machine

Negative Sampling

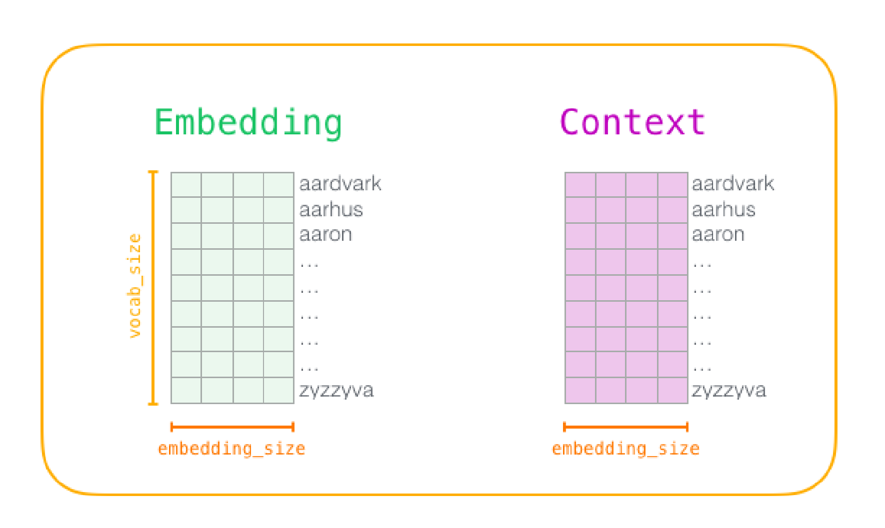
input word	output word	target
make	shalt	1
make	aaron	0
make	taco	0

Word2vec Training Process

- Now that weve 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 were 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 weve looked at an example of 50 earlier in this post).

Word2vec Training Process

- At the start of the training process, we initialize these matrices with random values.

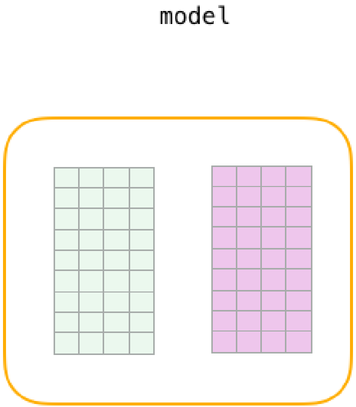


Word2vec Training Process

- Then we start the training process. In each training step, we take one positive example and its associated negative examples. Lets take our first group:

dataset

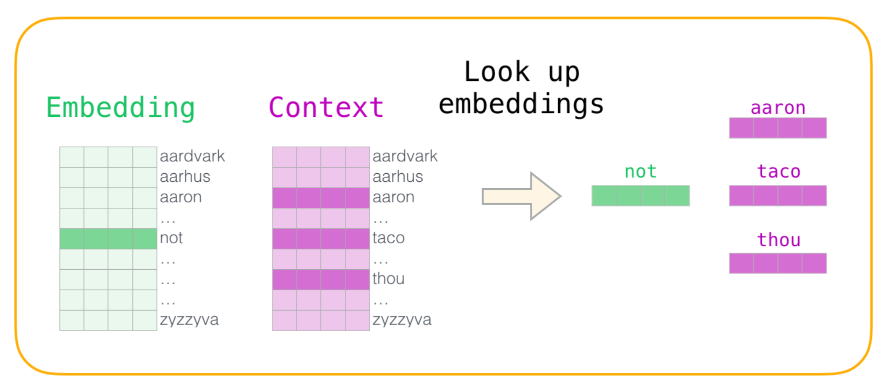
input word	output word	target
not	thou	1
not	aaron	0
not	taco	0
not	shalt	1
not	mango	0
not	finglonger	0
not	make	1
not	plumbus	0
...	...	...



Word2vec Training Process

- 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).



Word2vec Training Process

- We take the dot product of the input embedding with each of the context embeddings, that results in a number. The 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 - be positive and have values between 0 and 1, i.e., 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

Word2vec Training Process

- We can now treat the output of the sigmoid operations as the models 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, lets calculate how much error is in the models prediction. To do that, we just subtract the sigmoid scores from the target labels.

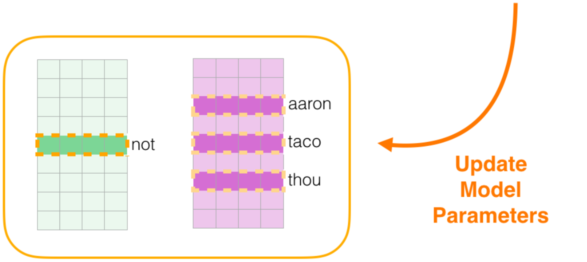
input word	output word	target	input • output	sigmoid()	Error
not	thou	1	0.2	0.55	0.45
not	aaron	0	-1.11	0.25	-0.25
not	taco	0	0.74	0.68	-0.68

- Error = target - sigmoid_scores

Word2vec Training Process

- 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

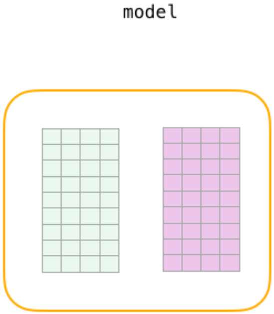


Word2vec Training Process

- To 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 proceed to our next step (the next positive sample and its associated negative samples) and do the same process again.

dataset

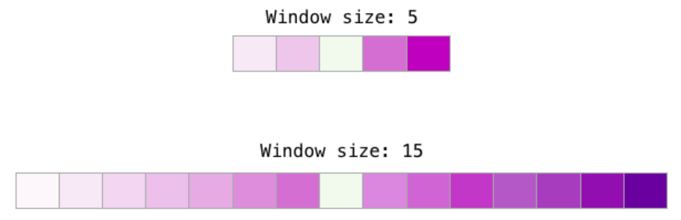
input word	output word	target
not	thou	1
not	aaron	0
not	taco	0
not	shalt	1
not	mango	0
not	finglonger	0
not	make	1
not	plumbus	0
...	...	...



- 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 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 are 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.

Window Size and Number of Negative Samples

- In practice, youll often have to provide annotations 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).
- 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.

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