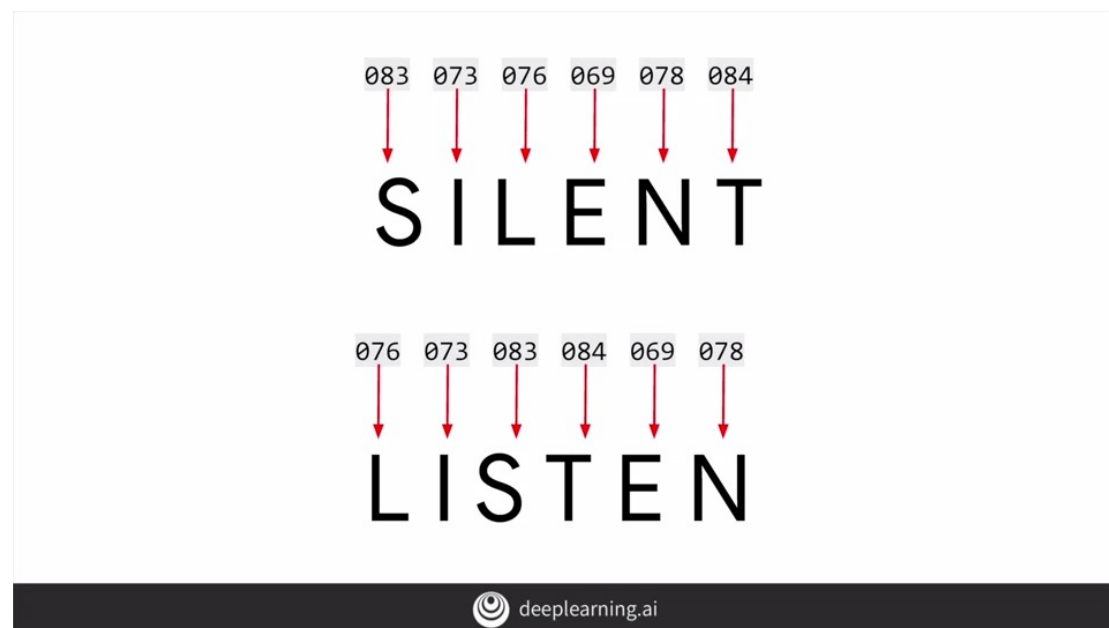
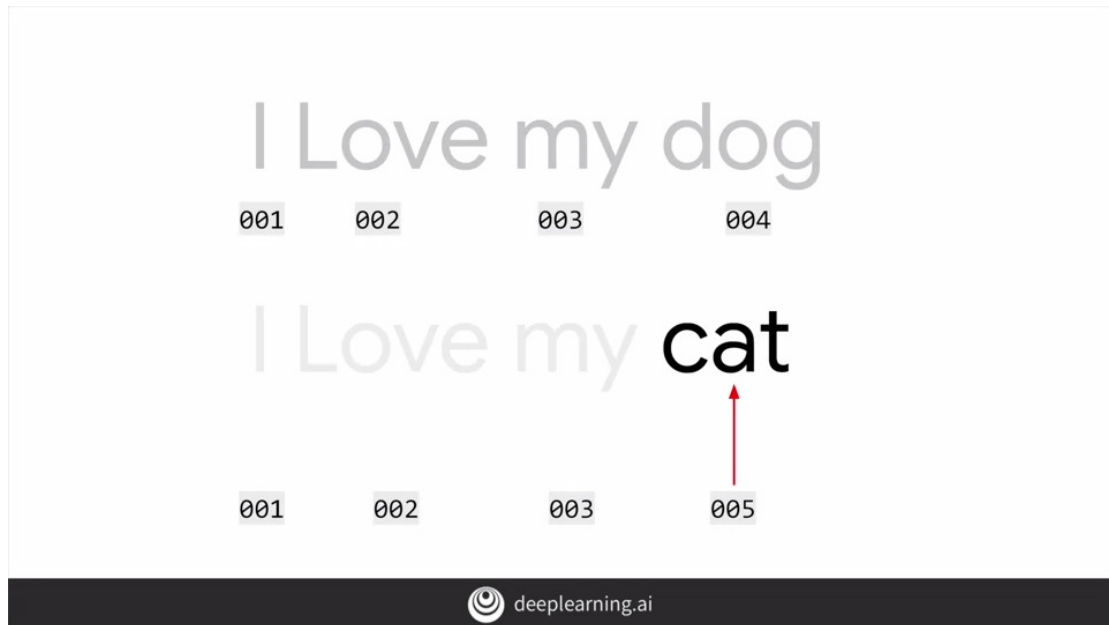


1.1 Word based encodings

We could take character encodings for each character in a set. For example, the ASCII values. But will that help us understand the meaning of a word? So for example, consider the word 'LISTEN' as shown here. A common simple character encoding is ASCII, the American Standard Code for Information Interchange with the values as shown here. So you might think you could have a word like LISTEN encoded using these values. But the problem with this of course, is that the semantics of the word aren't encoded in the letters. This could be demonstrated using the word 'SILENT' which has a very different and almost opposite meaning, but with exactly the same letters. So it seems that training a neural network with just the letters could be a daunting task.



So how about if we consider words? What if we could give words a value and have those values used in training a network? Now we could be getting somewhere. So for example, consider this sentence, I Love my dog. How about giving a value to each word? What that value is doesn't matter. It's just that we have a value per word, and the value is the same for the same word every time. So a simple encoding for the sentence would be for example to give word 'I' the value one. Following on, we could give the words 'Love', 'my' and 'dog' the values 2, 3, and 4 respectively. So then the sentence, I love my dog would be encoded as 1, 2, 3, 4. So now, what if I have the sentence, I love my cat? Well, we've already encoded the words 'I love my' as 1, 2, 3. So we can reuse those, and we can create a new token for cat, which we haven't seen before. So let's make that the number 5.



Now if we just look at the two sets of encodings, we can begin to see some similarity between the sentences. I love my dog is 1, 2, 3, 4 and I love my cat is 1, 2, 3, 5. So this is at least a beginning and how we can start training a neural network based on words. Fortunately, TensorFlow and Care Ask give us some APIs that make it very simple to do this. We'll look at those next.

1.2 Using APIs

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.preprocessing.text import Tokenizer

sentences = [
    'I love my dog',
    'I love my cat'
]

tokenizer = Tokenizer(num_words = 100)
tokenizer.fit_on_texts(sentences)
word_index = tokenizer.word_index
print(word_index)
```

Here's the code to encode the two sentences that we just spoke about. Let's unpack it line by line. Tensorflow and keras give us a number of ways to encode words, but the one I'm going to focus on is the tokenizer. This will handle the heavy lifting for us, generating the dictionary of word encodings and creating vectors out of the sentences. I'll put the sentences into an array. Note that I've already capitalized 'I' as it is at the beginning of the sentence. I then create an instance of the tokenizer. A passive parameter num words to it. In this case, I'm using 100 which is way too big, as there are only five distinct words in this data. If you're creating a training set based on lots of text, you usually don't know how many unique distinct words there are in that text. So by setting this hyperparameter, what the tokenizer will do is take the top 100 words by volume and just encode those. It's a handy shortcut when dealing with lots

of data, and worth experimenting with when you train with real data later in this course. Sometimes the impact of less words can be minimal and training accuracy, but huge in training time, but do use it carefully. The fit on texts method of the tokenizer then takes in the data and encodes it. The tokenizer provides a word index property which returns a dictionary containing key value pairs, where the key is the word, and the value is the token for that word, which you can inspect by simply printing it out.

```
{ 'i': 1, 'my': 3, 'dog': 4, 'cat': 5, 'love': 2 }
```

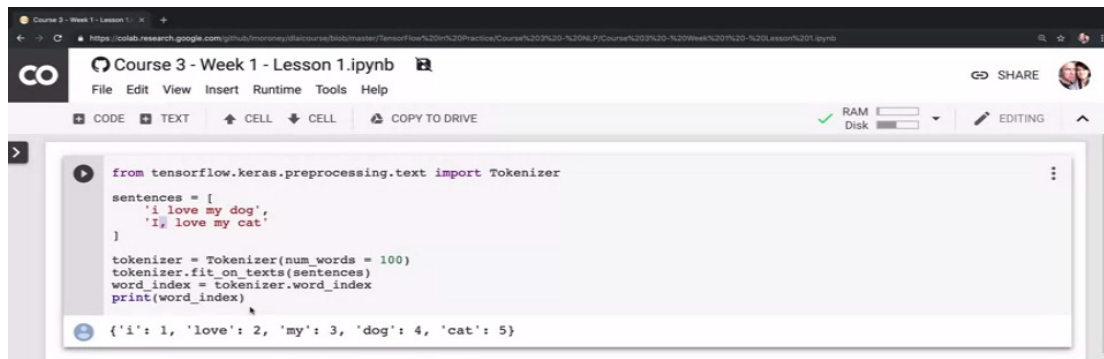
You can see the results here. Remember when we said that the word I was capitalized, note that it's lower-cased here. That's another thing that the tokenizer does for you. It strips punctuation out. This is really useful if you consider this case. Here, I've added another sentence, 'You love my dog!' but there's something very different about it. I've added an exclamation after the word 'dog!' Now, should this be treated as a different word than just dog? Well, of course not. So the results of the code that we saw earlier with this new corpus of data, will look like this. Notice that we still only have 'dog' as a key. That the exclamation didn't impact this, and of course, we have a new key for the word 'you' that was detected. So you've seen the beginnings of handling texts by creating word-based encodings of that text, with some very simple code intensive flow and keras. In the next video, we'll take a look at the code and see how it works.

1.3 Notebook for lesson 1

Here you can see the tokenizer from the keras' reprocessing library. The **tokenizer** is your friend when it comes to doing natural language processing. It does all the heavy lifting of managing tokens, turning your text into streams of tokens etc.

Now, the reason why you would need this is that when it comes to training neural networks, you're going to be doing a lot of math and math deals with numbers, and instead of having the words being trained in a neural network, you can actually have the number representing that word and it just makes your life a lot easier.

So here you can see I have a body of texts where my sentences "I love my dog and I love my cat," and I'm going to tokenize those using the tokenizer. In order to know, the tokenizer often creates the tokenizer using the **NumWords** property or the NumWords parameter. In this case, what it's going to do is, in your body of texts that it's tokenizing, **it will take the 100 most common words or whatever value that you actually put in here.** I have a lot less than a 100 unique words here, so it's not really going to have any effect.



The screenshot shows a Jupyter Notebook titled "Course 3 - Week 1 - Lesson 1.ipynb". The code in the cell is as follows:

```
from tensorflow.keras.preprocessing.text import Tokenizer

sentences = [
    'i love my dog',
    'I love my cat'
]

tokenizer = Tokenizer(num_words = 100)
tokenizer.fit_on_texts(sentences)
word_index = tokenizer.word_index
print(word_index)
```

The output of the code is a dictionary: `{'i': 1, 'love': 2, 'my': 3, 'dog': 4, 'cat': 5}`.

What fit on texts will then do is it will go through the entire body of text and it will create a **dictionary with the key being the word and the value being the token for that word**.

So those are the unique words that are actually in this corpus of text. A few things to take note of:

1. punctuation like **spaces and the comma, have actually been removed**. So it cleans up my text for me in that way too just to actually pull out the words.
2. you may have noticed that I have a lowercase i here and an uppercase I here. As you can see to make a case insensitive, it's just using I and it's giving the same token for both of these. Now if I were to change this a little bit by adding some new words to it, for example here you love my dog, notice that U is capitalized and dog has an exclamation after it, but it's not going to confuse that with the previous dog. So if I run it, we'll see now that I have a whole new set of tokens. I have one new one, I have six downside of five and that's because the word you is the only unique new word in this corpus, because love my and dog or their previously, but you'll see the exclamation from dog was removed. So that's a basic introduction to how the tokenizer actually works, and you'll be using that a lot in this course.

1.4 Text to sequence

In the previous video, you saw how to tokenize the words and sentences, building up a dictionary of all the words to make a corpus.

The next step will be to **turn your sentences into lists of values based on these tokens**. Once you have them, you'll likely also need to **manipulate these lists**, not least to make every sentence **the same length**, otherwise, it may be hard to train a neural network with them.

Remember when we were doing images, we defined an input layer with the size of the image that we're feeding into the neural network. In the cases where images were differently sized, we would resize them to fit. Well, you're going to face the same thing with text. Fortunately, TensorFlow includes APIs to handle these issues.

```

from tensorflow.keras.preprocessing.text import Tokenizer

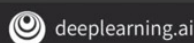
sentences = [
    'I love my dog',
    'I love my cat',
    'You love my dog!',
    'Do you think my dog is amazing?'
]

tokenizer = Tokenizer(num_words = 100)
tokenizer.fit_on_texts(sentences)
word_index = tokenizer.word_index

sequences = tokenizer.texts_to_sequences(sentences)

print(word_index)
print(sequences)

```



Let's start with creating a list of sequences, the sentences encoded with the tokens that we generated and I've updated the code that we've been working on to this. First of all, I've added another sentence to the end of the sentences list.

Note that all of the previous sentences had four words in them. So this one's a bit longer. We'll use that to demonstrate padding in a moment. The next piece of code is this one, where I simply **call on the tokenizer to get texts to sequences**, and it will turn them into a set of sequences for me. So if I run this code, this will be the output.

```

{'amazing': 10, 'dog': 3, 'you': 5, 'cat': 6,
 'think': 8, 'i': 4, 'is': 9, 'my': 1, 'do': 7,
 'love': 2}

[[4, 2, 1, 3], [4, 2, 1, 6], [5, 2, 1, 3], [7, 5,
8, 1, 3, 9, 10]]

```

At the top is the new dictionary. With new tokens for my new words like amazing, think, is, and do. At the bottom is my list of sentences that have been encoded into integer lists, with the tokens replacing the words. So for example, I love my dog becomes 4, 2, 1, 3.

One really handy thing about this that you'll use later is the fact that the text to sequences called can take any set of sentences, so it can encode them based on the word set that it learned from the one that was passed into fit on texts. This is very significant if you think ahead a little bit. If you train a neural network on a corpus of texts, and the text has a word index generated from it, then when you want to do inference with the train model, you'll have to encode the text that you want to infer on with the same word index, otherwise it would be meaningless.

```

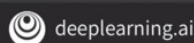
test_data = [
    'i really love my dog',
    'my dog loves my manatee'
]

test_seq = tokenizer.texts_to_sequences(test_data)
print(test_seq)

[[4, 2, 1, 3], [1, 3, 1]]

{'think': 8, 'amazing': 10, 'my': 1, 'love': 2, 'dog': 3, 'is': 9, 'you': 5, 'do': 7,
 'cat': 6, 'i': 4}

```



So if you consider this code, what do you expect the outcome to be? There are some familiar words here, like love, my, and dog but also some previously unseen ones. If I run this code, this is what I would get. I've added the dictionary underneath for convenience. So I really love my dog would still be encoded as 4, 2, 1, 3, which is 'I love my dog' with 'really' being lost as the word is not in the Word Index, and 'my dog loves my manatee' would get encoded to 1, 3, 1, which is just 'my dog my'.

1.5 Looking more at the Tokenizer

So what do we learn from this? First of all, we really need a lot of training data to get a broad vocabulary or we could end up with sentences like, my dog my, like we just did. Secondly, in many cases, it's a good idea to instead of just ignoring unseen words, to **put a special value in when an unseen word is encountered**. You can do this with a property on the tokenizer.

```

from tensorflow.keras.preprocessing.text import Tokenizer

sentences = [
    'I love my dog',
    'I love my cat',
    'You love my dog!',
    'Do you think my dog is amazing?'
]

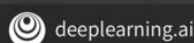
tokenizer = Tokenizer(num_words = 100, oov_token="<OOV>")
tokenizer.fit_on_texts(sentences)
word_index = tokenizer.word_index

sequences = tokenizer.texts_to_sequences(sentences)

test_data = [
    'i really love my dog',
    'my dog loves my manatee'
]

test_seq = tokenizer.texts_to_sequences(test_data)
print(test_seq)

```



Here's the complete code showing both the original sentences and the test data. What I've changed is to **add a property oov token** to the tokenizer constructor. You can see now that I've specified that I want the token oov for outer vocabulary to be used for words that aren't in the word index. You can use whatever you like here, but remember that it should be something unique and distinct that isn't confused with a real word.

```
[[5, 1, 3, 2, 4], [2, 4, 1, 2, 1]]

{'think': 9, 'amazing': 11, 'dog': 4, 'do': 8, 'i': 5, 'cat': 7,
 'you': 6, 'love': 3, '<OOV>': 1, 'my': 2, 'is': 10}
```

So now, if I run this code, I'll get my test sequences looking like this. I pasted the word index underneath so you can look it up. The first sentence will be, i out of vocab, love my dog. The second will be, my dog oov, my oov Still not syntactically great, but it is doing better. As the corpus grows and more words are in the index, hopefully previously unseen sentences will have better coverage.

Next up is **padding**. As we mentioned earlier when we were building neural networks to handle pictures. When we fed them into the network for training, we needed them to be uniform in size. Often, we use the generators to resize the image to fit for example. With texts you'll face a similar requirement before you can train with texts, we needed to have some level of uniformity of size, so padding is your friend there.

1.6 Padding

So I've made a few changes to the code to handle padding.

```
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences

sentences = [
    'I love my dog',
    'I love my cat',
    'You love my dog!',
    'Do you think my dog is amazing?'
]

tokenizer = Tokenizer(num_words = 100, oov_token="<OOV>")
tokenizer.fit_on_texts(sentences)
word_index = tokenizer.word_index

sequences = tokenizer.texts_to_sequences(sentences)

padded = pad_sequences(sequences)
print(word_index)
print(sequences)
print(padded)
```

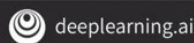
Here's the complete listing and we'll break it down piece by piece. First, in order to use the

padding functions you'll have to **import pad sequences from tensorflow.keras.preprocessing.sequence**. Then once the tokenizer has created the sequences, these sequences can be passed to pad sequences in order to have them padded like this.

```
{'do': 8, 'you': 6, 'love': 3, 'i': 5, 'amazing': 11, 'my': 2, 'is': 10, 'think': 9, 'dog': 4, '<OOV>': 1, 'cat': 7}

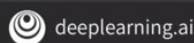
[[5, 3, 2, 4], [5, 3, 2, 7], [6, 3, 2, 4], [8, 6, 9, 2, 4, 10, 11]]

[[ 0  0  0  5  3  2  4]
 [ 0  0  0  5  3  2  7]
 [ 0  0  0  6  3  2  4]
 [ 8  6  9  2  4 10 11]]
```



The result is pretty straight forward. You can now see that the list of sentences has been padded out into a matrix and that each row in the matrix has the same length. It achieved this by putting the appropriate number of zeros before the sentence.

```
padded = pad_sequences(sequences, padding='post', maxlen=5)
```



If you, like me, are more comfortable with that, you can change the code to this, adding the parameter padding equals post. You may have noticed that the matrix width was the same as the longest sentence. But you can override that with the maxlen parameter. So for example if you only want your sentences to have a maximum of five words. You can say maxlen equals five like this.

This of course will lead to the question. If I have sentences longer than the maxlength, then I'll lose information but from where. Like with the padding the default is pre, which means that

you will lose from the beginning of the sentence. If you want to override this so that you lose from the end instead, you can do so with the truncating parameter like this. So you've now seen how to encode your sentences, how to pad them and how to use Word indexing to encode previously unseen sentences using out of vocab characters. But you've done it with very simple hard-coded data. Let's take a look at the coded action in a screencast and then we'll come back and look at how to use much more complex data.

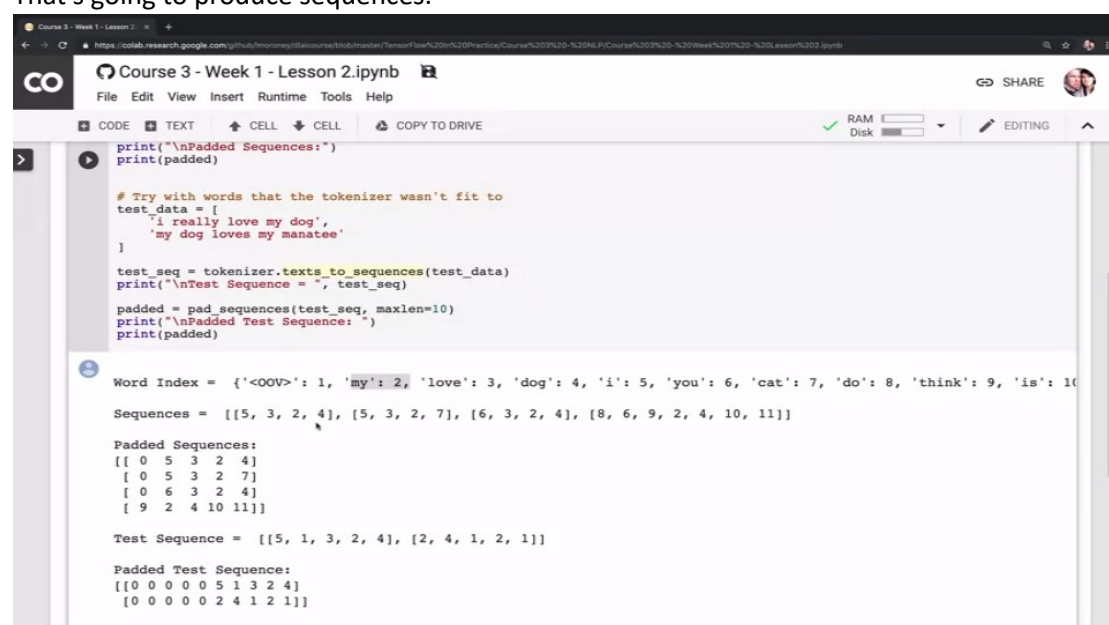
1.7 Notebook for lesson 2

So I'm going to take the tokenizer as we had before, but I'm also going to introduce this **Pad Sequences tool**. The idea behind the pad sequences tool is that it allows you to use sentences of different lengths, and use padding or truncation to make all of the sentences the same length.

So in this case, I have the same sentences as before; 'I love my dog,' 'I love my cat,' 'You love my dog,' but I've added this new sentence; 'Do you think my dog is amazing?' Which is a different length from these other sentences. These all had four words, this one has more. So my tokenizer, I'm going to create as before.

But I'm also going to use this parameter called an **OOV token**. The idea here is that I'm going to create a new token, a special token that I'm going to use for words that aren't recognized, that aren't in the word index itself. So I am going to just create this. I'm going to create something unique here that I wouldn't expect to see in the corpus. Something like bracket OOV, and I'm going to specify my OOV token is that.

So then I'm going to call tokenizer fit and text sentences, and I'm going to take a look at the word index for that. Let's actually run this. We'll see now, that in my word index, OOV is now value one, my is value two, love is three, etc. We have a total of 11 words, 11 unique words in this corpus. It's actually 10 words plus the OOV token. So on the tokenizer, I can then convert the words in those sentences to sequences of tokens by calling the text to sequences method. That's going to produce sequences.



```
print("\nPadded Sequences:")
print(padded)

# Try with words that the tokenizer wasn't fit to
test_data = [
    'i really love my dog',
    'my dog loves my manatee'
]

test_seq = tokenizer.texts_to_sequences(test_data)
print("\nTest Sequence = ", test_seq)

padded = pad_sequences(test_seq, maxlen=10)
print("\nPadded Test Sequence: ")
print(padded)

Word Index = {'<OOV>': 1, 'my': 2, 'love': 3, 'dog': 4, 'i': 5, 'you': 6, 'cat': 7, 'do': 8, 'think': 9, 'is': 10}

Sequences = [[5, 3, 2, 4], [5, 3, 2, 7], [6, 3, 2, 4], [8, 6, 9, 2, 4, 10, 11]]

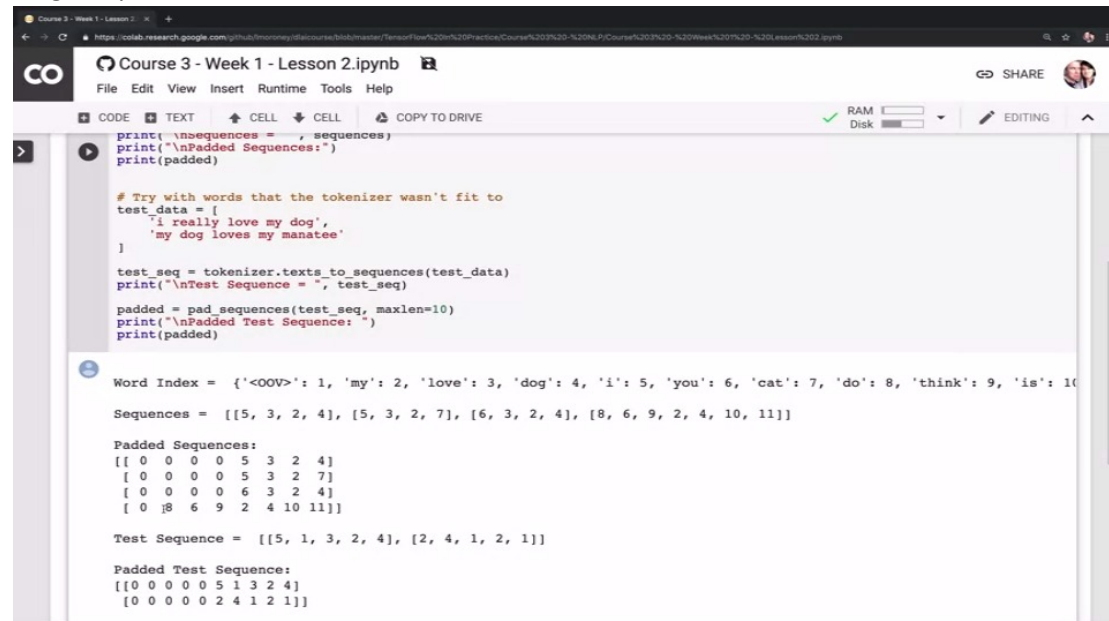
Padded Sequences:
[[0 5 3 2 4]
 [0 5 3 2 7]
 [0 6 3 2 4]
 [9 2 4 10 11]]

Test Sequence = [[5, 1, 3, 2, 4], [2, 4, 1, 2, 1]]

Padded Test Sequence:
[[0 0 0 0 5 1 3 2 4]
 [0 0 0 0 2 4 1 2 1]]
```

That's what I'm printing out here. So my sequences are five, three, two, four, for the first sentence, which is, 'I love my dog,' five, three, two, four, etc. So these are the sequence is 5324, 5327, 6324, 8692, 41011. Now, we can see these are all different length, but we want to make them the same length. So that's where Pad Sequences comes into it. So I'm going to say here **my pad is set is pad sequences with the sequences.**

Let's make it a maximum length of five words. So this maximum length of five words means that these four-letter, or these four-words sentences end up being pre-padded with a zero. This six-word sentence **ends up having the first word cutoff** because we did say maximum length equals five.



```
print("\nsequences = ", sequences)
print("\nPadded Sequences:")
print(padded)

# Try with words that the tokenizer wasn't fit to
test_data = [
    'i really love my dog',
    'my dog loves my manatee'
]

test_seq = tokenizer.texts_to_sequences(test_data)
print("\nTest Sequence = ", test_seq)

padded = pad_sequences(test_seq, maxlen=10)
print("\nPadded Test Sequence: ")
print(padded)

Word Index = {'<OOV>': 1, 'my': 2, 'love': 3, 'dog': 4, 'i': 5, 'you': 6, 'cat': 7, 'do': 8, 'think': 9, 'is': 10}

Sequences = [[5, 3, 2, 4], [5, 3, 2, 7], [6, 3, 2, 4], [8, 6, 9, 2, 4, 10, 11]]

Padded Sequences:
[[ 0 0 0 0 5 3 2 4]
 [ 0 0 0 0 5 3 2 7]
 [ 0 0 0 0 6 3 2 4]
 [ 0 8 6 9 2 4 10 11]]

Test Sequence = [[5, 1, 3, 2, 4], [2, 4, 1, 2, 1]]

Padded Test Sequence:
[[ 0 0 0 0 5 1 3 2 4]
 [ 0 0 0 0 2 4 1 2 1]]
```

I said maximum length equals eight, for example, and then ran this. We can see now that they're all pre-padded with zeros, including this long sentence, it's being prepared with a single zero. There are methods on pad sequences that we saw, and the lessons that will allow us to do a post if we want to do so, and then the zeros would appear afterwards. So now, if I want to take a look at words that the tokenizer wasn't fit to. So for example, my test data is I really love my dog and my dog loves my manatee, if I now tokenized them and create sequences out of that, we'll see 51324 for the first sentence. Five is I, one is out of vocabulary, because really wasn't actually there, and three to four, 'I still love my dog.' So this is how the outer vocabulary token comes into it. When it sees a word that wasn't in the word index, it will replace it, it will just use the out of vocabulary token one for that. Similarly, for 'my dog loves my manatee,' I get 24121, the word 'loves' is not in it, even though the word love is, and of course, manatee isn't in it either. So I end up with just 242, other words that really have meaning in this, and that's 'my dog,' 'my,' which is, 'my dog my loves manatee,' out of vocabulary tokens. Of course here, you can see I'm also padding them. So my 51324 gets padded, and my 24121 also gets padded. Because I'd said a max length of 10, if I set that for example to two, we'll see they end up getting truncated. I am getting the last two words here, and getting the last two words here. So that's a basic introduction to how tokenizer works, and how padding actually works, to give you padding, to be able to get your sentences all the same length. I hope this was useful for you.

1.8 Sarcasm, really?

So far this week, you've been looking at texts, and how to tokenize the text, and then turn sentences into sequences using the tools available in TensorFlow. You did that using some very simple hard-coded sentences. But of course, when it comes to doing real-world problems, you'll be using a lot more data than just these simple sentences. So in this lesson, we'll take a look at some public data-sets and how you can process them to get them ready to train a neural network.

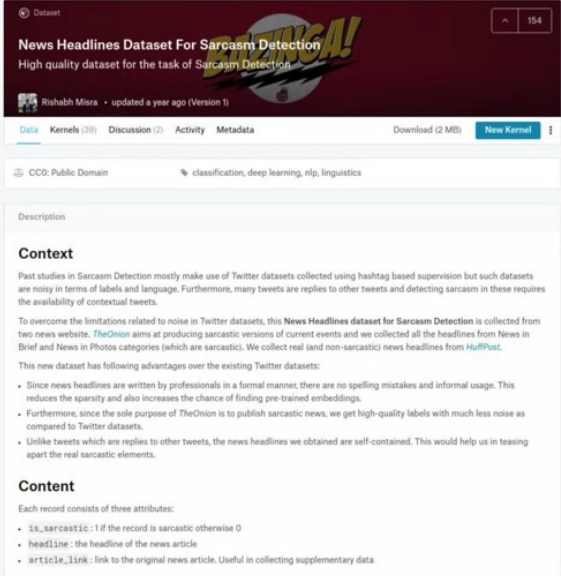


Sarcasm in News Headlines Dataset by Rishabh Misra

<https://rishabhmisra.github.io/publications/>

deeplearning.ai

We'll start with this one published by Rishabh Misra with details on Kaggle at this link. It's a really fun CC0 public domain data-set at all around sarcasm detection. Really? Yeah, really. This data-set is very straightforward and simple, not to mention very easy to work with. It has three elements in it.



News Headlines Dataset For Sarcasm Detection
High quality dataset for the task of Sarcasm Detection

Rishabh Misra · updated a year ago (Version 1)

CC0: Public Domain · classification, deep learning, nlp, linguistics

Description

Context

Past studies in Sarcasm Detection mostly make use of Twitter datasets collected using hashtag based supervision but such datasets are noisy in terms of labels and language. Furthermore, many tweets are replies to other tweets and detecting sarcasm in these requires the availability of contextual tweets.

To overcome the limitations related to noise in Twitter datasets, this **News Headlines dataset for Sarcasm Detection** is collected from two news website. *TheOnion* aims at producing sarcastic versions of current events and we collected all the headlines from News in Brief and News in Photos categories (which are sarcastic). We collect real (and non-sarcastic) news headlines from *HuRPoast*.

This new dataset has following advantages over the existing Twitter datasets:

- Since news headlines are written by professionals in a formal manner, there are no spelling mistakes and informal usage. This reduces the sparsity and also increases the chance of finding pre-trained embeddings.
- Furthermore, since the sole purpose of *TheOnion* is to publish sarcastic news, we get high-quality labels with much less noise as compared to Twitter datasets.
- Unlike tweets which are replies to other tweets, the news headlines we obtained are self-contained. This would help us in teasing apart the real sarcastic elements.

Content

Each record consists of three attributes:

- `is_sarcastic`: 1 if the record is sarcastic otherwise 0
- `headline`: the headline of the news article
- `article_link`: link to the original news article. Useful in collecting supplementary data

`is_sarcastic`: 1 if the record is sarcastic otherwise 0

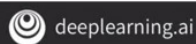
`headline`: the headline of the news article

`article_link`: link to the original news article. Useful in collecting supplementary data

The first is `is_sarcastic`, is our label. It's a one if the record is considered sarcastic otherwise it's zero. The second is a `headline`, which is just plain text and the third is the `article_link` to the article

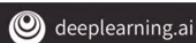
that the headline describes. Parsing the contents of HTML, stripping out scripts, and styles, etc, is a little bit beyond the scope of this course. So we're just going to focus on the headlines.

```
{ "article_link":  
  "https://politics.theonion.com/boehner-just-wants-wife-to-listen-not-come-up-with-alt-1819574302",  
  "headline": "boehner just wants wife to listen, not come up with alternative debt-reduction ideas",  
  "is_sarcastic": 1}  
  
{ "article_link":  
  "https://www.huffingtonpost.com/entry/roseanne-revival-review_us_5ab3a497e4b054d118e04365",  
  "headline": "the 'roseanne' revival catches up to our thorny political mood, for better and worse",  
  "is_sarcastic": 0}  
  
{ "article_link":  
  "https://local.theonion.com/mom-starting-to-fear-son-s-web-series-closest-thing-she-1819576697",  
  "headline": "mom starting to fear son's web series closest thing she will have to grandchild",  
  "is_sarcastic": 1}
```



If you download the data from that Kaggle site, you'll see something like this. As you can see, it is a set of list entries with name-value pairs where the name is article link, headline and is_sarcastic and the values are as shown.

```
[  
{ "article_link":  
  "https://politics.theonion.com/boehner-just-wants-wife-to-listen-not-come-up-with-alt-1819574302",  
  "headline": "boehner just wants wife to listen, not come up with alternative debt-reduction ideas",  
  "is_sarcastic": 1},  
  
{ "article_link":  
  "https://www.huffingtonpost.com/entry/roseanne-revival-review_us_5ab3a497e4b054d118e04365",  
  "headline": "the 'roseanne' revival catches up to our thorny political mood, for better and worse",  
  "is_sarcastic": 0},  
  
{ "article_link":  
  "https://local.theonion.com/mom-starting-to-fear-son-s-web-series-closest-thing-she-1819576697",  
  "headline": "mom starting to fear son's web series closest thing she will have to grandchild",  
  "is_sarcastic": 1}
```



To make it much easier to load this data into Python, I made a little tweak to the data to look like this, which you can feel free to do or you can download my amended data-set from the link in the co-lab for this part of the course.

```
import json

with open("sarcasm.json", 'r') as f:
    datastore = json.load(f)

sentences = []
labels = []
urls = []
for item in datastore:
    sentences.append(item['headline'])
    labels.append(item['is_sarcastic'])
    urls.append(item['article_link'])
```



Once you have the data like this, it's then really easy to load it into Python. Let's take a look at the code. So first you need to import JSON. This allows you to load data in JSON format and automatically create a Python data structure from it. To do that you simply open the file, and pass it to `json.load` and you'll get a list containing lists of the three types of data: headlines, URLs, and `is_sarcastic` labels. Because I want the sentences as a list of their own to pass to the tokenizer, I can then create a list of sentences and later, if I want the labels for creating a neural network, I can create a list of them too. While I'm at it, I may as well do URLs even though I'm not going to use them here but you might want to. Now I can iterate through the list that was created with a `for item in data store` loop. For each item, I can then copy the headline to my sentences, the `is_sarcastic` to my labels and the `article_link` to my URLs. Now I have something I can work with in the tokenizer, so let's look at that next.

1.9 Working with the Tokenizer

```
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences

tokenizer = Tokenizer(oov_token="<OOV>")
tokenizer.fit_on_texts(sentences)
word_index = tokenizer.word_index

sequences = tokenizer.texts_to_sequences(sentences)
padded = pad_sequences(sequences, padding='post')
print(padded[0])
print(padded.shape)
```

This code is very similar to what you saw in the earlier videos, but let's look at it line by line. We've just created sentences less from the headlines, in the sarcasm data set. So by calling **tokenizer.fit** on texts, will generate the word index and we'll initialize the tokenizer. We can see the word index as before by calling the word index property. Note that this returns all words that the tokenizer saw when tokenizing the sentences. If you specify num words to get the top 1000 or whatever, you may be confused by seeing something greater than that here.

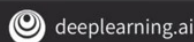
It's an easy mistake to make, but the key thing to remember, is that when it takes the top 1000 or whatever you specified, it does that in the text to sequence this process. Our word index is much larger than with the previous example.

```
{'underwood': 24127, 'skillingsballe': 23055, 'grabs': 12293, 'mobility': 8909,
 "'assassin's": 12648, 'visualize': 23973, 'hurting': 4992, 'orphaned': 9173,
 "'agreed'": 24365, 'narration': 28470}
```

So we'll see a greater variety of words in it. Here's a few.

```
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
tokenizer = Tokenizer(oov_token="<OOV>")
tokenizer.fit_on_texts(sentences)
word_index = tokenizer.word_index

sequences = tokenizer.texts_to_sequences(sentences)
padded = pad_sequences(sequences, padding='post')
print(padded[0])
print(padded.shape)
```



Now we'll create the sequences from the text, as well as padding them. Here's the code to do that. It's very similar to what you did earlier, and here's the output. First, I took the first headline in the data set and showed its output.

```
[ 308 15115 679 3337 2298 48 382 2576 15116 6 2577 8434
  0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0]

(26709, 40)
```

We can see that it has been encoded with the values for the keys that are the corresponding word in the sentence. This is the size of the padded matrix. We had 26,709 sentences, and they were encoded with padding, to get them up to 40 words long which was the length of the longest word. You could truncate this if you like, but I'll keep it at 40. That's it for processing the Sarcasm data set. Let's take a look at that in action in a screen cast.

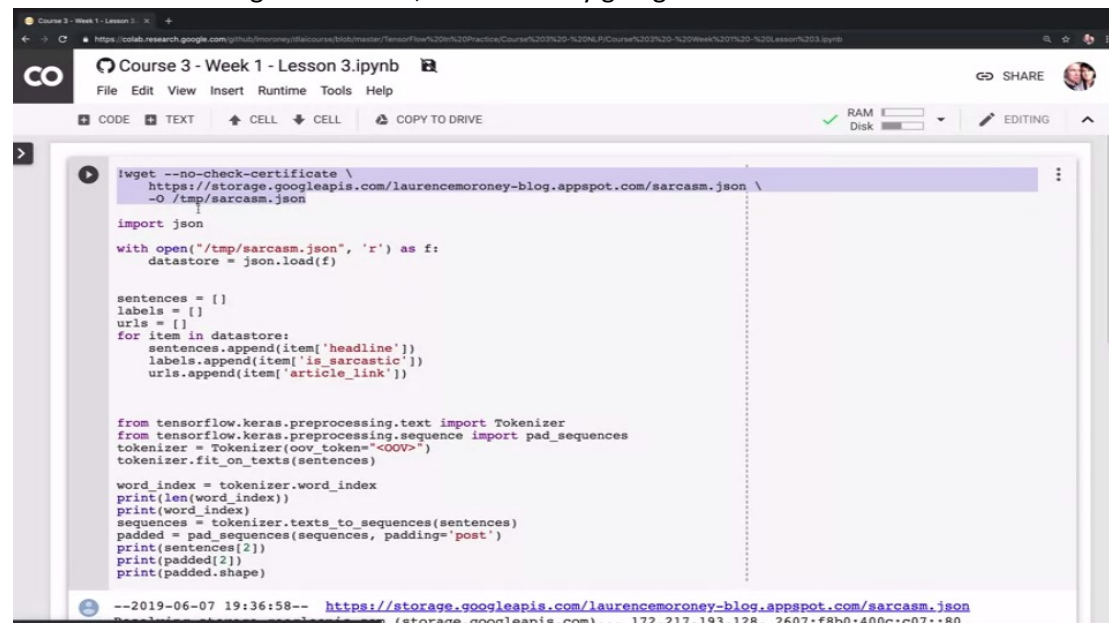
The following is the public domain dataset based on sarcasm, as depicted in the previous video. The link is provided here for your convenience:

[Sarcasm in News Headlines Dataset by Rishabh Misra](#)

1.10 Notebook for lesson 3

So let's take a look at tokenizing and padding a lot more data. So instead of just a few sentences

that we were looking at earlier on, we're actually going to use a full data set.



```
!wget --no-check-certificate \
  https://storage.googleapis.com/laurencemoroney-blog.appspot.com/sarcasm.json \
  -O /tmp/sarcasm.json

import json

with open("/tmp/sarcasm.json", 'r') as f:
    datastore = json.load(f)

sentences = []
labels = []
urls = []
for item in datastore:
    sentences.append(item['headline'])
    labels.append(item['is_sarcastic'])
    urls.append(item['article_link'])

from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
tokenizer = Tokenizer(oov_token="<OOV>")
tokenizer.fit_on_texts(sentences)

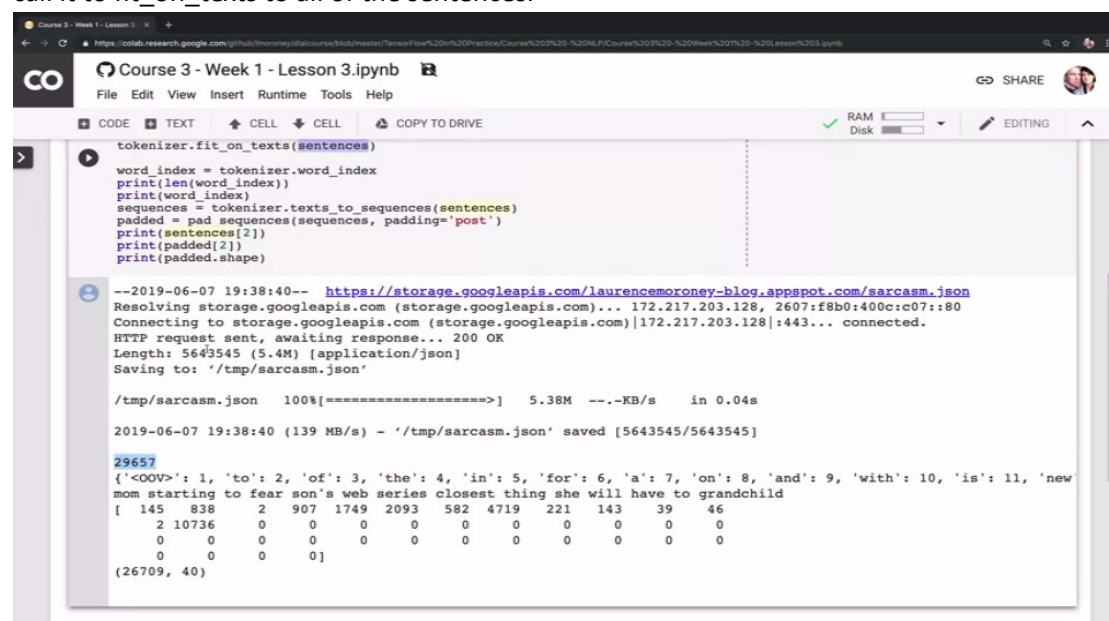
word_index = tokenizer.word_index
print(len(word_index))
print(word_index)
sequences = tokenizer.texts_to_sequences(sentences)
padded = pad_sequences(sequences, padding='post')
print(sentences[2])
print(padded[2])
print(padded.shape)

--2019-06-07 19:36:58-- https://storage.googleapis.com/laurencemoroney-blog.appspot.com/sarcasm.json
Resolving storage.googleapis.com (storage.googleapis.com)... 172.217.193.128, 2607:f8b0:400c:c07:80
```

And this is the sarcasm data set that we were talking about in the lesson. So here is, I'm just going to **wget that sarcasm data set and download it to /tmp/sarcasm.json**. So once I've downloaded it, I'm now going to just create three lists, one for the sentences, one for the labels, and one for the URLs. I'm not actually going to use the URLs here, but just if you want to use them yourself here's how you do it.

And then for each item in the data store, because in JSON it's really easy for me to iterate through it. For each item in the data store I can then just add the headline to sentences, I can add the item is_sarcastic to label, and I can add the item article_link to URLs.

Okay now that my data is ready I'm going to input my tokenizer and my pad sequences as before, I'm going to use the tokenizer to specify my out of vocabulary token. And I'm going to call it to fit_on_texts to all of the sentences.



```
tokenizer.fit_on_texts(sentences)

word_index = tokenizer.word_index
print(len(word_index))
print(word_index)
sequences = tokenizer.texts_to_sequences(sentences)
padded = pad_sequences(sequences, padding='post')
print(sentences[2])
print(padded[2])
print(padded.shape)

--2019-06-07 19:38:40-- https://storage.googleapis.com/laurencemoroney-blog.appspot.com/sarcasm.json
Resolving storage.googleapis.com (storage.googleapis.com)... 172.217.203.128, 2607:f8b0:400c:c07:80
Connecting to storage.googleapis.com (storage.googleapis.com)|172.217.203.128|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 5643545 (5.4M) [application/json]
Saving to: '/tmp/sarcasm.json'

/tmp/sarcasm.json 100%[=====] 5.38M --.-KB/s in 0.04s

2019-06-07 19:38:40 (139 MB/s) - '/tmp/sarcasm.json' saved [5643545/5643545]

29657
{'<OOV>': 1, 'to': 2, 'of': 3, 'the': 4, 'in': 5, 'for': 6, 'a': 7, 'on': 8, 'and': 9, 'with': 10, 'is': 11, 'new
mom starting to fear son's web series closest thing she will have to grandchild
[ 145 838 2 907 1749 2093 582 4719 221 143 39 46
0 0 0 0 0 0 0 0 0 0 0 0
2 10736 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
(26709, 40)]
```

Now it's a case of we've a much larger vocabulary, we have 29,657 words in the index. And we can start seeing things like out of vocabulary is 1, to is number 2, of is number 3, the is number

4. Remember these are going to be sorted into their order of commonality. So you should see basic words like the and for pretty high up in the list. So once I have my word index, and you can see, I'm just printing out the length of that which is what gives me the 29,657. And I printed the word index.

So now on my tokenizer I can call `texts_to_sequences` as before and get my sentences into that to turn them into sequences. And the last screencast we used `pad_sequences`, but the padding was used in the default, which was `pre`, and everything was prefixed with zeros. This time to show `padding=post`, will just allow us to put zeros after the sentence so that the sentence will be post padded. And if I for example look at sentence number two in the corpus and what padded to in the corpus looks like we'll see sentence number two is mom starting to fear sons web series is the closest thing she'll have to a grandchild. And we can see the tokens for those actual words in here. So, for example here was number two moms starting to and we can see that the word to is number 2. And maybe, are there any others that are pretty high up in the list? We could say for example number three 39 if I go through my list here and see what 39 is, We'll see that the word will is number 39. So if I come back to here she will have to a grandchild is number 39. And then finally the shape of the padded one is that each of the sentences in the data set is being padded up to 40 characters long, or 40 words long. And so we have 26,709 sentences in the data set, so my shape of my padded array will be that. And this is what could be used to then train a neural network with embeddings, that you'll be seeing next week.