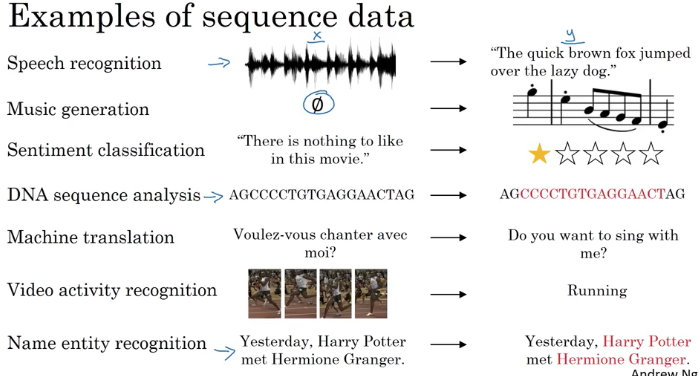
# W1

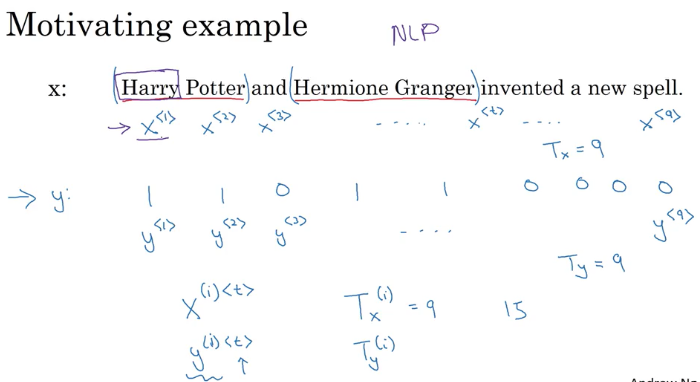
## Why Sequence Models



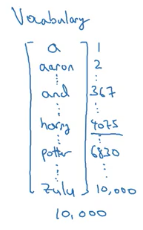
## Notation

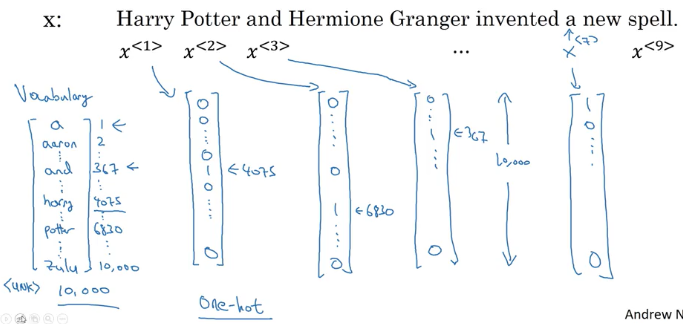
Named\_entity recognition is used by search engines to index last 24 hours news for ex

Y tells you for each of the input words is that part of a person's name. 1 if a name, 0 if not.



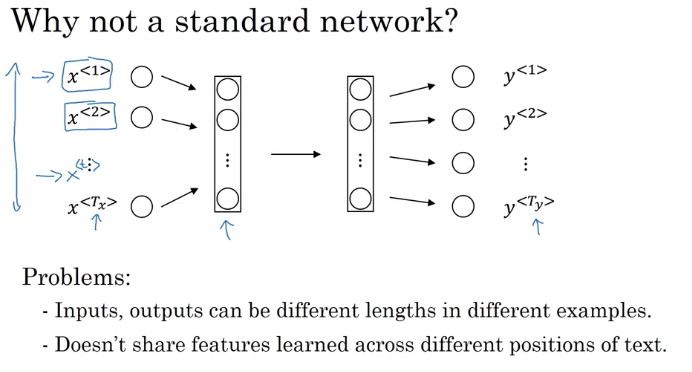
Giving to all words a number 1 to 10K for ex



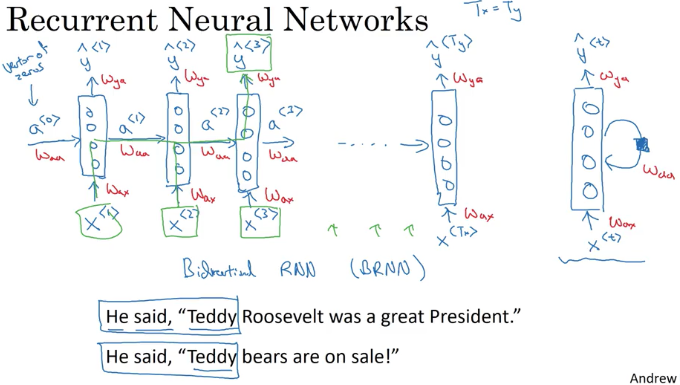


* Each word has a index starting from 1 to 10K in vocabulary vector.
* Only the word 4075 is 1 in x1 others are all zeros
* Each word has 10K vector due to vocabulary.
* <UNK> is used for words not in your vocabulary.

## Recurrent NN

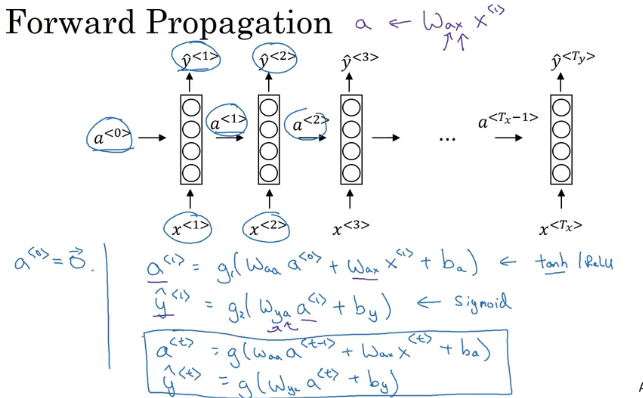


* So, you could imagine trying to take these nine input words, maybe the nine one-hot vectors and feeding them into a standard neural network, maybe a few hidden layers, and then eventually had this output the nine values zero or one that tell you whether each word is part of a person's name.
* each of these is a 10,000 dimensional one-hot vector and so this is just a very large input layer if the total input size was maximum number of words times 10,000. **A weight matrix of this first layer will end up having an enormous number of parameters**. So a recurrent neural network which we'll start to describe in the next slide does not have either of these disadvantages.
* **What a recurrent neural network does is**,
* instead of just predicting y2 using only X2, it also gets to input some information from whether the computer that time step one.
* at each time step, the recurrent neural network that passes on as activation to the next time step for it to use

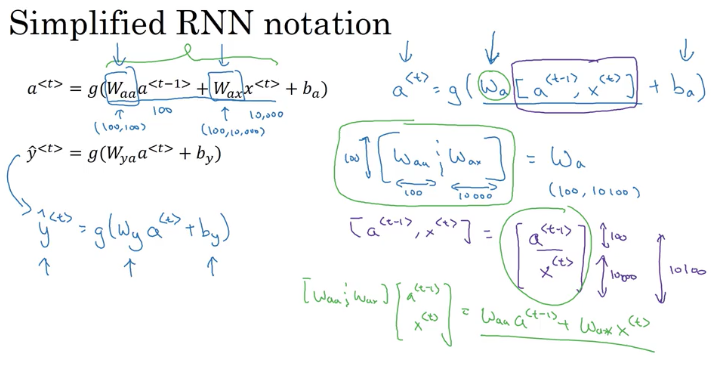


* *So given just the first three words is not possible to know for sure whether the word Teddy is part of a person's name*. In the first example, it is. In the second example, it is not. But you can't tell the difference if you look only at the first three words.
* **Waa = horizontal parameters**
* **Wax= X to hidden layer parameters**
* **Wya = governs the output predictons**
* when making the prediction for y3, it gets the information not only from X3 but also the information from x1 and x2 because the information on x1 can pass through this way to help to prediction with Y3.
* So one limitation of this particular neural network structure is that the prediction at a certain time uses inputs or ***uses information from the inputs earlier in the sequence but not information later in the sequence.***
* This is a problem, because you cannot decide Teddy is a person name or not by looking the first three words in the sentence.

## Forward Prop



Tanh is a common choice for computing activations

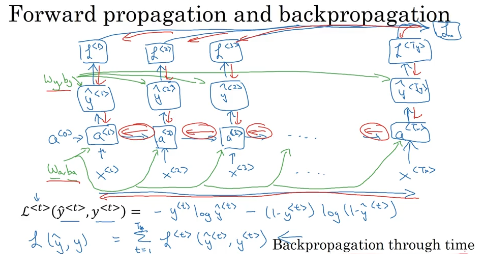


Simplifying Waa and Wax into Wa

## Backpropagation throuh time

Red lines are the backprob flow

in this back propagation procedure, the most significant message or the most significant recursive calculation is the one from atx to a0

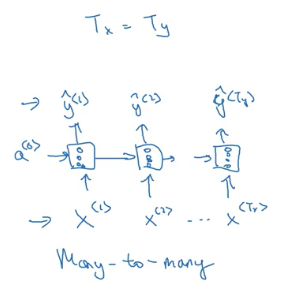


These are RNN, in which the length of the input sequence was equal to the length of the output sequence

## Different types of RNNs

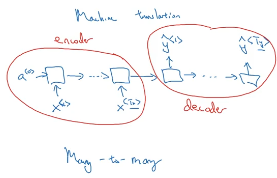
Input and output may be different size like translation from English to french;there may be different number of words as output.

### Many\_to\_Many Architectrure



### Many to many with different size of tx and ty

**translation for ex; And** so, this that collinear network architecture has two distinct parts. There's the encoder which takes as input, say a French sentence, and then, there's is a decoder, which having read in the sentence, outputs the translation into a different language. So this would be an example of a many-to-many architecture.



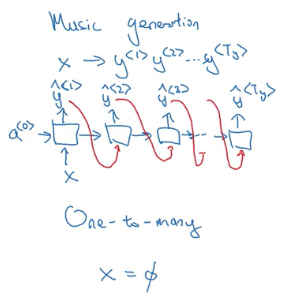
### Many\_to\_One architecture

For example there is a movie classification problem: from the comment text("There is nothing to like in this movie”) , you want to predict as movie is liked or not. IN this case there can only be one Y ; and that will be at the end.

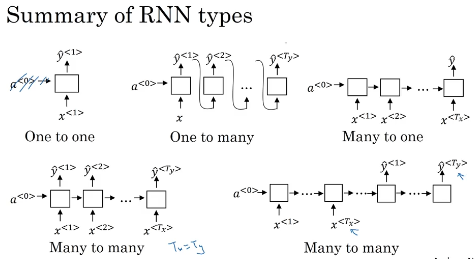


### One\_to\_Many Architecture

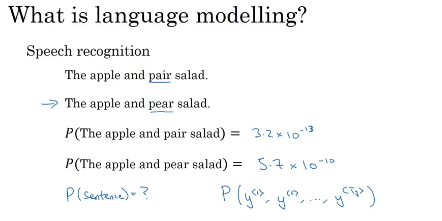
the input x could be maybe just an integer, telling it what genre of music you want or what is the first note of the music you want, and if you don't want to input anything, x could be a null input, could always be the vector zeroes as well.



### Summary



## Language model and sequence generation



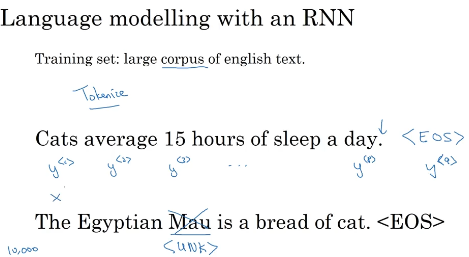
What a language model does is

* given any sentence is job is to tell you what is the probability of a sentence, of that particular sentence;
* if you want to pick up a random newspaper, open a random email or pick a random webpage or listen to the next thing someone says, the friend of you says. What is the chance that the next sentence you use somewhere out there in the world will be a particular sentence like the apple and pear salad?

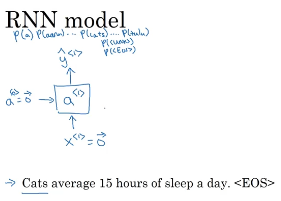
## How to build language model

### Tokenize each word

* That means you would form a vocabulary
* Match every word to a specific indices
* Model when sentences end.EOS- end of sentence token



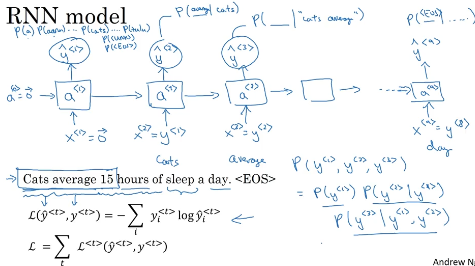
### Replace unknown words wih <UNK>



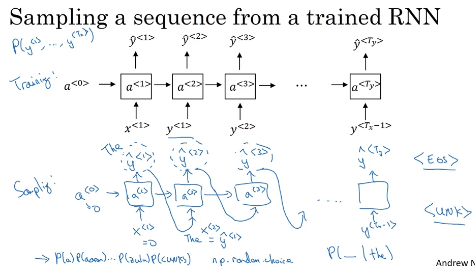
* what A1 does is it will make a soft max prediction to try to figure out what is the probability of the first words y. And so that's going to be y<1>.
* So what this step does is really, it has a soft max it's trying to predict. ***What is the probability of any word in the dictionary***? That the first one is a, what's the chance that the first word is Aaron? And then what's the chance that the first word is cats? All the way to what's the chance the first word is Zulu? Or what's the first chance that the first word is an unknown word? Or what's the first chance that the first word is the in the sentence they'll have, shouldn't have to read?
* Right, so y hat 1 is output to a soft max, it just predicts what's the chance of the first word being, whatever it ends up being. And in our example, it wind up being the word cats, so this would be a 10,000 way soft max output, if you have a 10,000-word vocabulary
* So each step in the RNN will look at some set of preceding words such as,

given the first three words, what is the distribution over the next word?

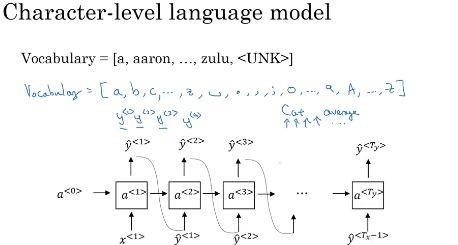
* And so this RNN learns to predict one word at a time going from left to right.
* *Next step is to compute cost*



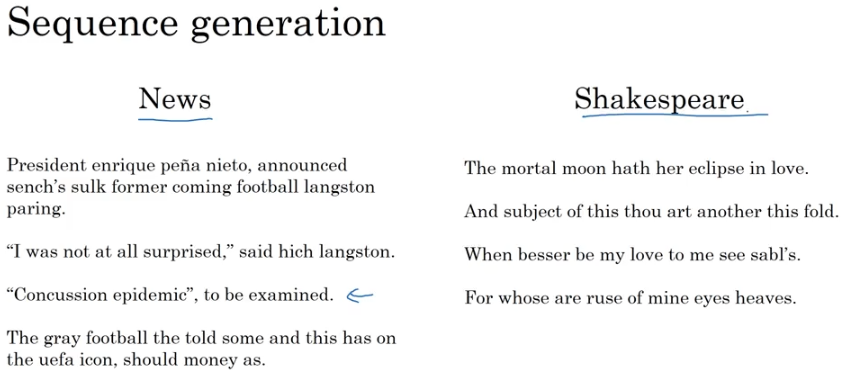
## Sampling novel sequences



* After you train a sequence mode, you can informally get a sense of what is learned is to have a sample novel sequence
* a sequence model models the chance of any particular sequence of words as follows, and so what we like to do is sample from this distribution to generate noble sequences of words.
* So the network was trained using this structure shown at the top of pic above.
* **But to sample, you do something slightly different, so what you want to do is first sample what is the first word you want your model to generate**
* And so for that you input the usual x1 equals 0, a0 equals 0. And now your first time stamp will have some max probability over possible outputs. So what you do is you then randomly sample according to this soft max distribution.
* So what the soft max distribution gives you is it tells you what is the chance that it refers to this a, what is the chance that it refers to this Aaron? What's the chance it refers to Zulu, what is the chance that the first word is the Unknown word token. Maybe it was a chance it was a end of sentence token. And then you take this vector and use, for example, the numpy command np.random.choice to sample according to distribution defined by this vector probabilities, and that lets you sample the first words.
* Next you then go on to the second time step, and now remember that the second time step is **expecting this y1 as input.** But what you do is you then take the y1 hat that you just sampled and pass that in here as the input to the next timestep.
* This was ***word level RNN,*** you could also build ***character level RNN***



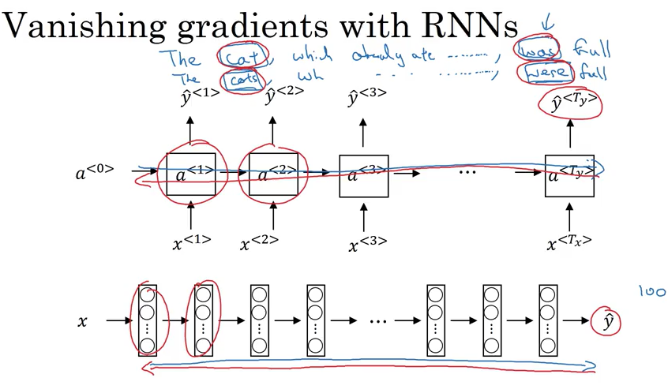
* **the main disadvantage of the character level language model is that you end up with much longer sequences**
* **computationally expensive to train**
* **trend is still using word level,** character level is not widespread; but by the computer’s development, more people starts to look at this character level algorithms



* If the model was trained on news articles, then it generates texts like that shown on the left
* If it was trained on Shakespearean text and then it generates stuff that sounds like Shakespeare could have written it.

## Vanishing gradients with RNNs

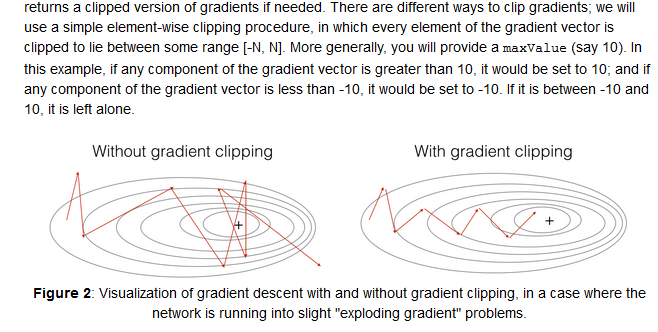
* it turns out the basics RNN we've seen so far it's not very good at capturing very long-term dependencies. Because of vanishing gradient problems.
* So this is a very, very deep neural network say, 100 layers or even much deeper than you would carry out forward prop, from left to right and then back prop. And we said that, if this is a very deep neural network, then the gradient from just output y, would have a **very hard time propagating back to affect the weights of these earlier layers**, to affect the computations in the earlier layers.
* And for an RNN with a similar problem, you have forward prop came from left to right, and then back prop, going from right to left. And it can be quite difficult, because of the same ***vanishing gradients problem, for the outputs of the errors associated with the later time steps to affect the computations that are earlier***.



* And ***so in practice***, what this means is, ***it might be difficult to get a neural network to realize that it needs to memorize the just see a singular noun or a plural noun***, so that later on in the sequence that can generate either was or were, depending on whether it was singular or plural.
* And notice that in English, this stuff in the middle could be arbitrarily long, right? So you might need to memorize the singular/plural for a very long time before you get to use that bit of information.
* So because of this problem, the basic RNN model has many local influences, meaning that the output y^<3> is mainly influenced by values close to y^<3>. And a value here is mainly influenced by **inputs that are somewhere close So this is a weakness of the basic RNN algorithm**

## Gradient clipping

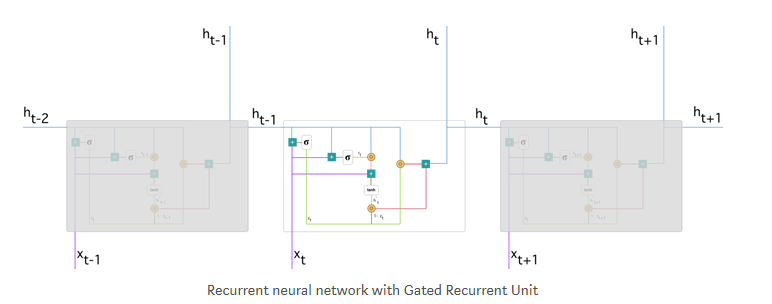
* to avoid exploding gradients(meaning taking on overly large values.)



Exploding gradients, you could sort of address by just using gradient clipping, but vanishing gradients will take more work to address.(gated recurrent units )

## Gated Recurrent Unit (GRU)

* Gated Recurrent Unit is a modification to the RNN hidden layer that makes it much ***better capturing long range connections***
* ***helps a lot with the vanishing gradient problems.***
* *They have fewer parameters than LSTM, as they lack an output gate*
* *To solve the vanishing gradient problem of a standard RNN, GRU uses, so called, update gate and reset gate. Basically, these are two vectors which decide what information should be passed to the output. The special thing about them is that they can be trained to keep information from long ago, without washing it through time or remove information which is irrelevant to the prediction.(* [*https://towardsdatascience.com/understanding-gru-networks-2ef37df6c9be*](https://towardsdatascience.com/understanding-gru-networks-2ef37df6c9be)*)*

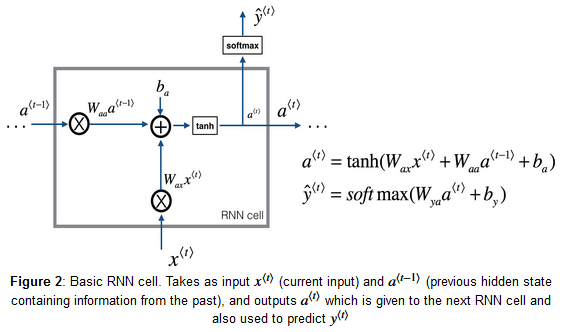


***The update gate*** *helps the model to determine how much of the past information (from previous time steps) needs to be passed along to the future.*



**Reset gate is used from the model to decide how much of the past information to forget. To** calculate it, we use:

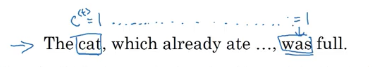




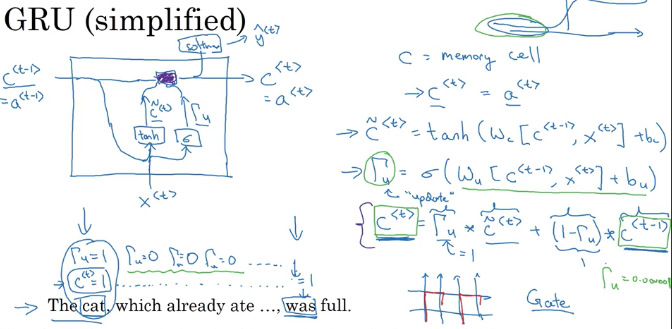
***Memory cell:*** memorizes the word cat in this example, whether it is plural or singular

symbol c : to denote the memory cell value ;

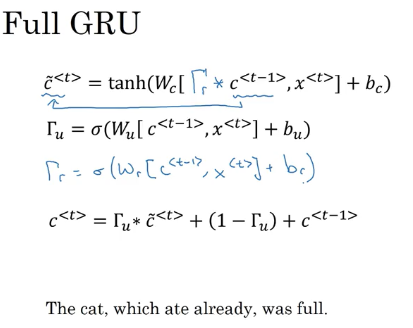
a for the output activation value



* memory cell c is going to be set to either zero or one depending on whether the is singular or plural. So because it's singular, let's say that we set this to one. And if it was plural, maybe we would set this to zero, and then the GRU unit would memorize the value of the c<t> all the way until here, where this is still equal to one and so that tells it, oh, it's singular so use the choice was. ***And the job of the gate, of gamma u, is to decide when do you update these values.***

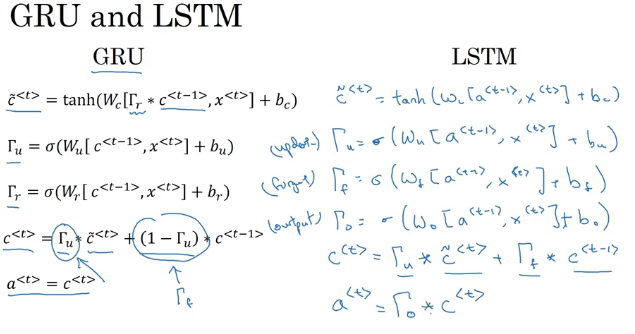


* Hopefully you've just been setting c<t> equals c<t> minus one all along. And it still memorizes the cat was singular.

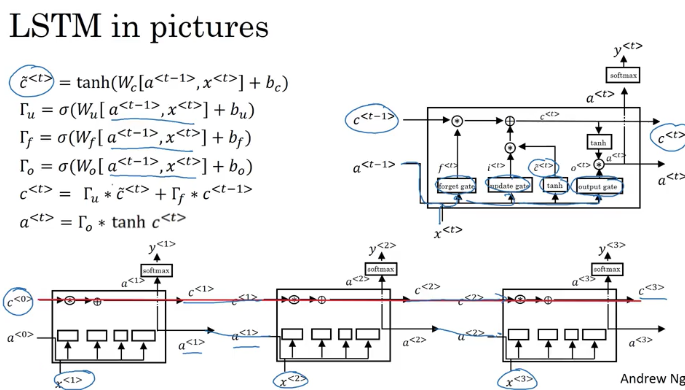


## LSTM Long Short Term Memory

A common LSTM unit is composed of a cell, **an input gate, an output gate and a forget gate**. The cell is responsible for "remembering" values over arbitrary time intervals; hence the word "memory" in LSTM. Each of the three gates can be thought of as a "conventional" artificial neuron, as in a multi-layer (or feedforward) neural network: that is, they compute an activation (using an activation function) of a weighted sum. Intuitively, they can be thought as regulators of the flow of values that goes through the connections of the LSTM; hence the denotation "gate". There are connections between these gates and the cell.



The expression long short-term refers to the fact that LSTM is a model for the short-term memory which can last for a long period of time. **An LSTM is well-suited to classify, process and predict time series given time lags of unknown size** and duration between important events. LSTMs were developed to deal with the exploding and vanishing gradient problem when training traditional RNNs. Relative insensitivity to gap length gives an advantage to LSTM over alternative RNNs, hidden Markov models and other sequence learning methods in numerous applications

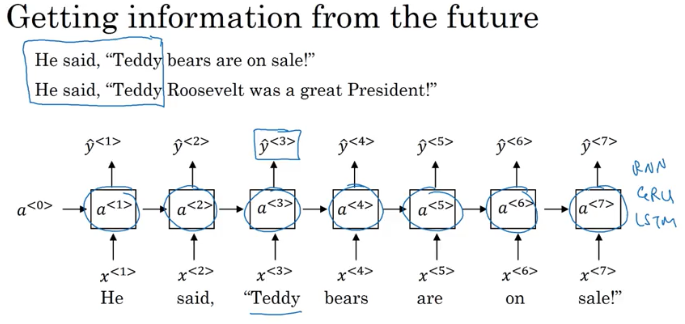


* The advantage of the **GRU** is that it's a simpler model and so it is actually easier to build a much bigger network, **it only has two gates**, so computationally, it runs a bit faster.
* So, it scales the building somewhat bigger models but the ***LSTM is more powerful and more effective since it has three gates instead of two.***

If you want to pick one to use, I think LSTM has been the historically more proven choice. So, if you had to pick one, I think most people today will still use the LSTM as the default first thing to try.

Although, I think in the last few years, GRUs had been gaining a lot of momentum and I feel like more and more teams are also using GRUs because they're a bit simpler but often work just as well. It might be easier to scale them to even bigger problems.

## Bidirectional RNN

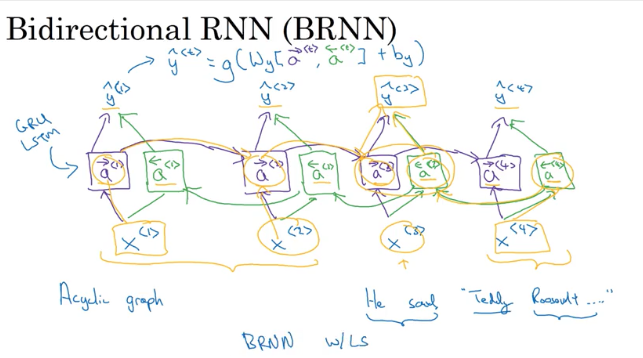


**one of the problems of unidirectional network is** that, to figure out whether the third word Teddy is a part of the person's name, **it's not enough to just look at the first part of the sentence.** So to tell, if Y three should be zero or one, you need more information than just the first three words because the first three words doesn't tell you if they'll talking about Teddy bears or talk about the former US president, Teddy Roosevelt.

So what a bidirectional RNN does or BRNN, is fix this issue.

For a lot of text with natural language processing problems, a bidirectional RNN with a LSTM appears to be commonly used.

So, if we have NLP problem and you have the complete sentence, you try to label things in the sentence, **a bidirectional RNN with LSTM blocks both forward and backward** would be a pretty views of first thing to try.



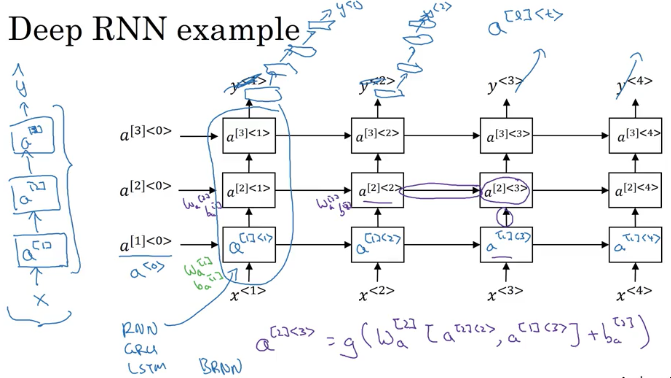
And so, given an input sequence, X1 through X4, the fourth sequence will first compute A forward1 , then use that to compute A forward 2, then A forward three, then A forward four.

Whereas, the backward sequence would start by computing A backward 4, and then go back and compute A backward 3, and then as you are computing network activation, this is not backward this is forward prop.

So this allows the prediction at time three to take as input both information from the past, as well as information from the present which goes into both the forward and the backward things at this step, as well as information from the future. So, in particular, given a phrase like, "He said, Teddy Roosevelt..." To predict whether Teddy is a part of the person's name, ***you take into account information from the past and from the future.***

The disadvantage of the bidirectional RNN is that you do need the entire sequence of data before you can make predictions anywhere

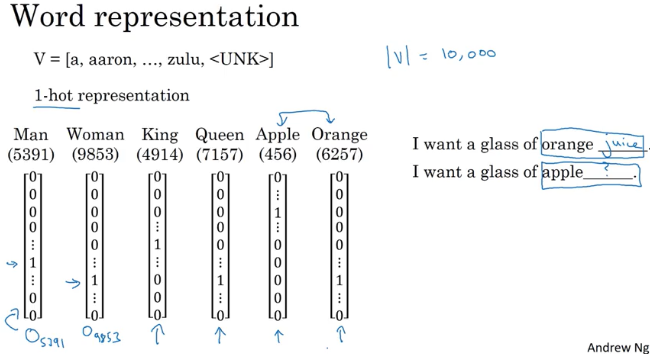
## Deep RNNs



# W2

## Word Representation

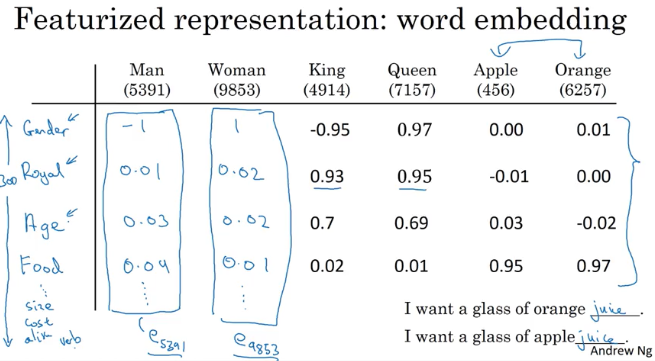
**Onehot represantation**



* Even if the learning algorithm has learned that I want a glass of orange juice is a likely sentence, if it sees I want a glass of apple blank. As far as it knows the relationship between apple and orange is not any closer as the relationship between any of the other words man, woman, king, queen, and orange.
* And so, it's not easy for the learning algorithm to generalize from knowing that orange juice is a popular thing, to recognizing that apple juice might also be a popular thing or a popular phrase. **And this is because the any product between any two different one-hot vector is zero.**
* If you take any two vectors say, queen and king and any product of them, the end product is zero. If you take apple and orange and any product of them, the end product is zero. And you couldn't distance between any pair of these vectors is also the same. So it just doesn't know that somehow apple and orange are much more similar than king and orange or queen and orange

**Featurized representation**

300 dimensional(features)



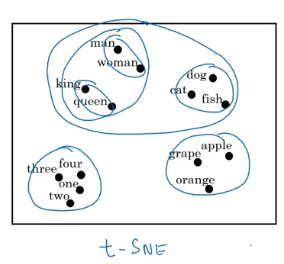
A lot of the features of apple and orange are actually the same, or take on very similar values. And so, this increases the odds of the learning algorithm that has figured out that orange juice is a thing, to also quickly figure out that apple juice is a thing. So this allows it to generalize better across different words.

We just need you to learn high dimensional feature vectors like these, *that gives a better representation than one-hot vectors for representing different words*. And the features we'll end up learning, won't have a easy to interpret interpretation like that component one is gender, component two is royal, component three is age and so on.

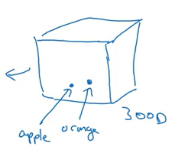
Exactly what they're representing will be a bit harder to figure out. But nonetheless, the featurized representations we will learn, ***will allow an algorithm to quickly figure out that apple and orange are more similar than say, king and orange or queen and orange.***

**t\_SNE**

* If we're able to learn a **300 dimensional feature vector or 300 dimensional embedding** for each words, one of the popular things to do is also to take this 300 dimensional data and embed it say, in a two dimensional space so that you can visualize them.

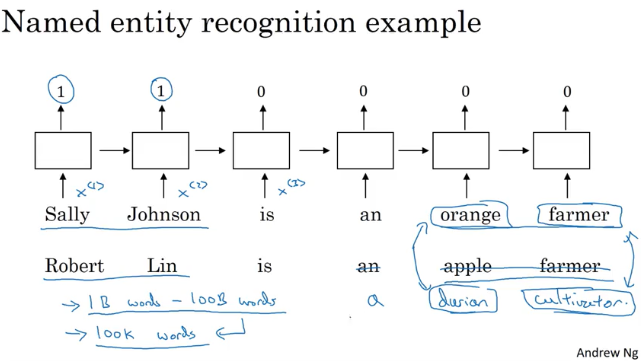
* And so, one common algorithm for doing this is the t-SNE algorithm due to Laurens van der Maaten and Geoff Hinton.
* And if you look at one of these embeddings, one of these representations, you find that words like man and woman tend to get grouped together, king and queen tend to get grouped together,



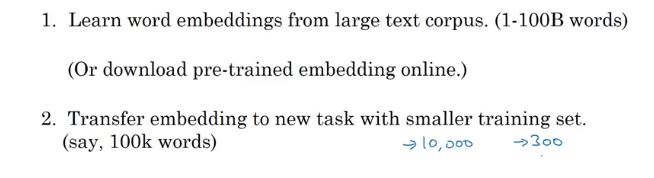
* what you do is you take every words like orange, and have a three dimensional feature vector so that word orange gets embedded to a point in this 300 dimensional space.
* And the word apple, gets embedded to a different point in this 300 dimensional space. And of course to visualize it, ***algorithms like t-SNE, map this to a much lower dimensional space,*** you can actually plot the 2D data and look at it. But that's what the term embedding comes from.
* Word embeddings has been one of the most important ideas in NLP, in Natural Language Processing.

## Using Word Embeddings

* Given a sentence like Sally Johnson is an orange farmer, hopefully, you'll figure out that Sally Johnson is a person's name, hence, the outputs 1 like that. And one way to be sure that Sally Johnson has to be a person, rather than say the name of the corporation is that you know orange farmer is a person.



* Knowing that orange and apple are very similar will make it easier for your learning algorithm to generalize to figure out that Robert Lin is also a human, is also a person's name
* A durian is a rare type of fruit, popular in Singapore and a few other countries. But if you have a small label training set for the named entity recognition task, you might not even have seen the word durian or seen the word cultivator in your training set.
* But if you have learned a word embedding that tells you that durian is a fruit, so it's like an orange, and a cultivator, someone that cultivates is like a farmer, then you might still be generalize from having seen an orange farmer in your training set to knowing that a durian cultivator is also probably a person
* So one of the reasons that word embeddings will be able to do this is the algorithms to learning word embeddings can examine very large text corpuses, maybe found off the Internet.
* So you can examine very large data sets, maybe a billion words, maybe even up to 100 billion words would be quite reasonable. So very large training sets of just unlabeled text.And by examining tons of unlabeled text, which you can download more or less for free, you can figure out that orange and durian are similar.
* Now having discovered that orange and durian are both fruits by reading massive amounts of Internet text, what you can do is then take this word embedding and apply it to your named entity recognition task, for which you might have a much smaller training set, maybe just 100,000 words in your training set, or even much smaller.
* And so this allows you to carry out transfer learning, where you take information you've learned from huge amounts of unlabeled text that you can suck down essentially for free off the Internet to figure out that orange, apple, and durian are fruits.



* So rather than using a 10,000 dimensional one-hot vector, you can now instead use maybe a 300 dimensional dense vector. Although the one-hot vector is fast and the 300 dimensional vector that you might learn for your embedding will be a dense vector.

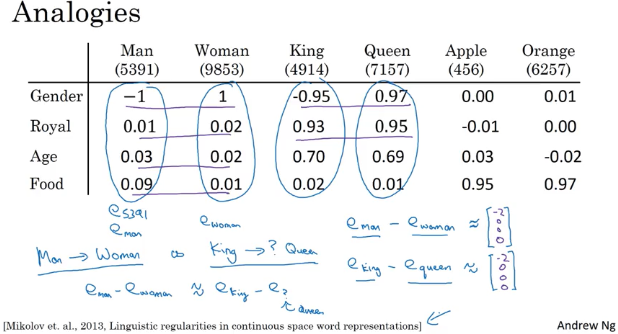


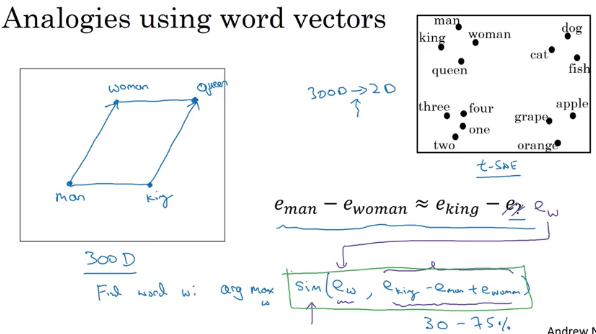
* finally, as you train your model on your new task, on your named entity recognition task with a smaller label data set, one thing you can optionally do is to continue to fine tune,

continue to adjust the word embeddings with the new data. (In practice; you would do this only if you have task2 a big dataset)

## Properties of word embeddings

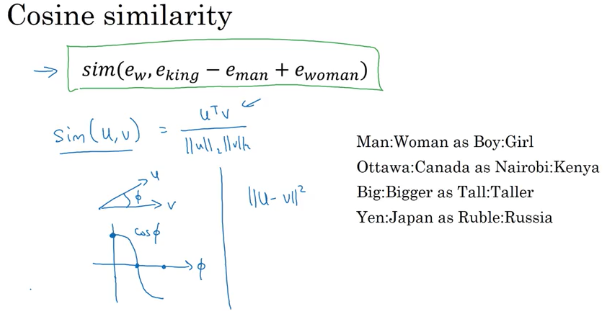
Used for analogy reasoning





### Cosine similarity

* U and v are two vectors, looking for similarity we use cosine sim.

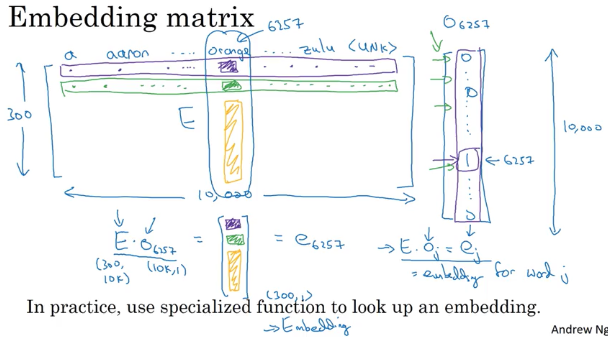


* if the angle between them is 0, then the cosine similarity is equal to 1. And if their angle is 90 degrees, the cosine similarity is 0. And then if they're 180 degrees, or pointing in completely opposite directions, it ends up being -1.

## Embedding Matrix

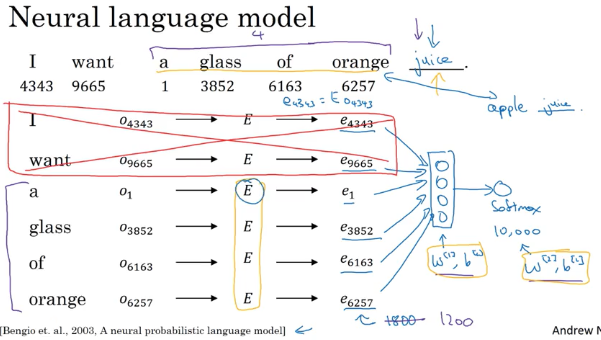
Matrix E : 10K vocabulary with 300 features each.

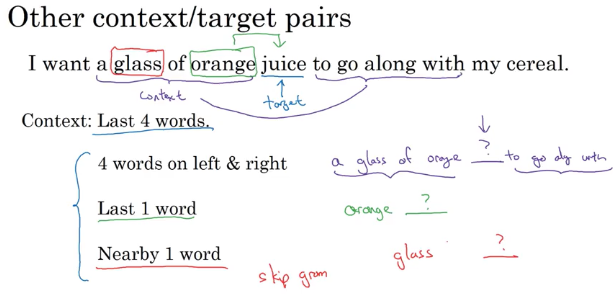
Vector e: for ex orange’s all features inside the matrix E.



So, in this video you saw the notations were used to describe algorithms to learning these embeddings and the key terminology is this matrix capital E which contain all the embeddings for the words of the vocabulary.

## Learning Word Embeddings



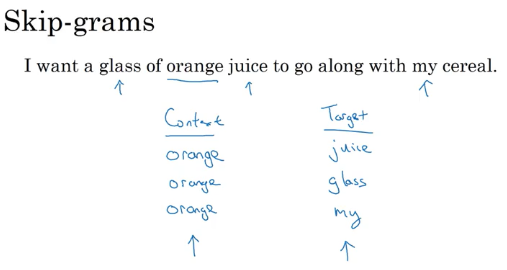


* In this video you saw how the language modeling problem which causes the pose of machines learning problem where you input the context like the last four words and predicts some target words, how posing that problem allows you to learn input word embedding

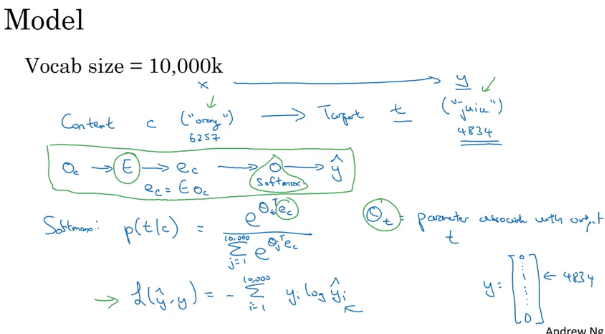
## Word2Vec

There are two versions of this Word2Vec model, the skip gram was one. and the other one is called the CBow.

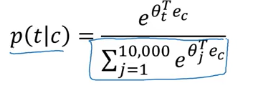
### Skip-grams



* Rather than having the context be always the last four words or the last end words immediately before the target word, what I'm going to do is, say, randomly pick a word to be the context word. And let's say we chose the word orange.
* And what we're going to do is randomly pick another word within some window. Say plus minus five words or plus minus ten words of the context word and we choose that to be target word.

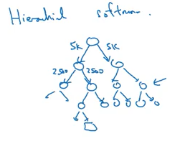


* So we use y to represent the target word. And we use a one-hot representation for y hat and y here. Then the lost would be The negative log liklihood, so sum from i equals 1 to 10,000 of yi log yi hat. So that's a usual loss for softmax where we're representing the target y as a one hot vector. So this would be a one hot vector with just 1 1 and the rest zeros. And if the target word is juice, then it'd be element 4834 from up here. That is equal to 1 and the rest will be equal to 0.
* Y hat will be a 10,000 dimensional vector output by the softmax unit with probabilities for all 10,000 possible targets words.
* ***This is called the skip-gram model because is taking as input one word like orange and then trying to predict some words skipping a few words from the left or the right side***. To predict what comes little bit before little bit after the context words.

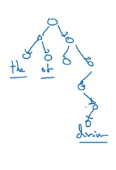


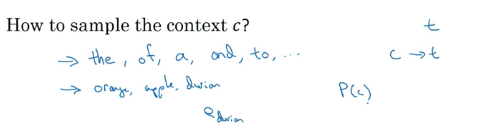
* Now, it turns out there are a couple problems with using this algorithm. And the primary problem is ***computational speed***. In particular, for the softmax model, every time you want to evaluate this probability, **you need to carry out a sum over all 10,000 words in your vocabulary**
* in fact, 10,000 is actually already that will be quite slow, but it makes even harder to scale to larger vocabularies. So there are a few solutions to this, one which you see in the literature is to use a hierarchical softmax classifer

### hierarchical softmax classifier



* Imagine if you have one classifier, it tells you is the target word in the first 5,000 words in the vocabulary? Or is in the second 5,000 words in the vocabulary? And lets say this binary cost that it tells you this is in the first 5,000 words, think of second class to tell you that this in the first 2,500 words of vocab or in the second 2,500 words vocab and so on.
* Until eventually you get down to classify exactly what word it is, so that the leaf of this tree, and so having a tree of classifiers like this, means that each of the retriever nodes of the tree can be just a binding classifier
* And so you don't need to sum over all 10,000 words or else it will capsize in order to make a single classification. In fact, the computational classifying tree like this scales like log of the vocab size rather than linear in vocab size
* In practice the below tree is constructed .





**How to sample the context c?**

* Some words are used frequently (the,of, a etc.)
* you don't want your training site to be dominated by these extremely frequently or current words, because then you spend almost all the effort updating ec, for those frequently occurring words. But you want to make sure that you spend some time updating the embedding, even for these less common words like e durian.
* In practice the distribution of words pc isn't taken just entirely uniformly at random for the training set purpose, but instead **there are different heuristics that you could use in order to balance out something from the common words together with the less common words**

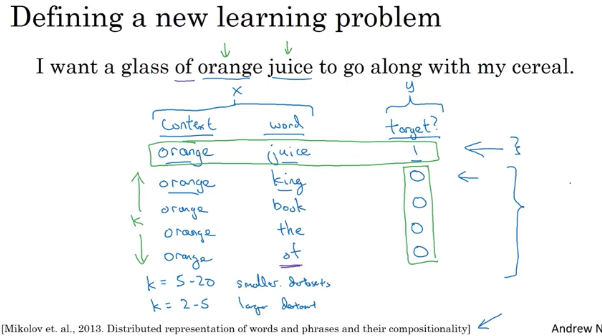
### Cbow

CBow, the continuous backwards model, which takes the surrounding contexts from middle word, and uses the surrounding words to try to predict the middle word, and that algorithm also works, it has some advantages and disadvantages

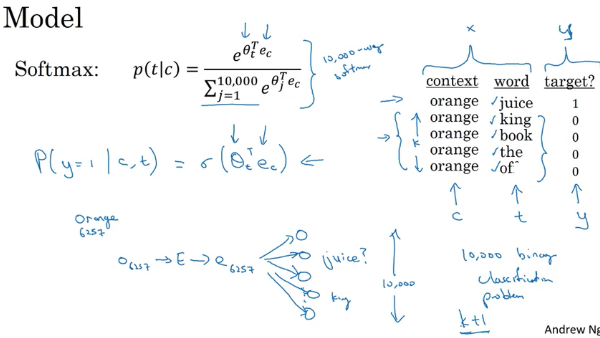
But the key problem with this algorithm with the skip-gram model as presented so far is that the softmax step is very expensive to calculate because needing to sum over your entire vocabulary size into the denominator of the soft packs.

## Negative Sampling

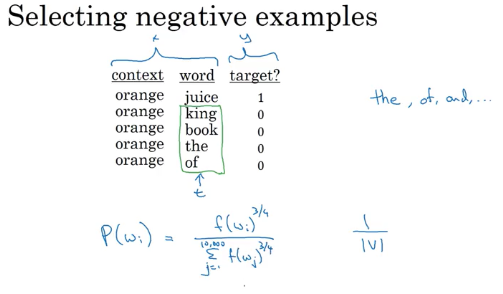
The problem is, given a pair of words like orange and juice, we're going to predict, is this a context-target pair? So in this example, orange juice was a positive example. And how about orange and king? Well, that's a negative example,



* So the problem is really given a pair of words like orange and juice, do you think they appear together?
* Do you think I got these two words by sampling two words close to each other?
* Or do you think I got them as one word from the text and one word chosen at random from the dictionary? It's really to try to distinguish between these two types of distributions from which you might sample a pair of words.So this is how you generate the training set.

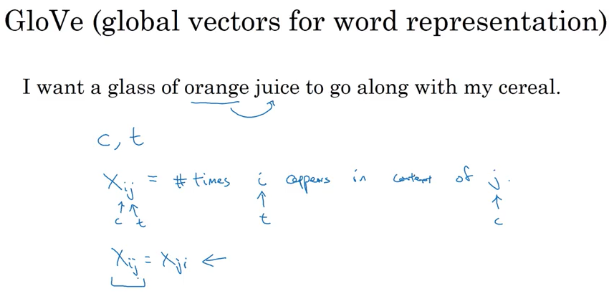


* instead of having one giant 10,000 way Softmax, which is very expensive to compute, we've instead turned it into 10,000 binary classification problems, each of which is quite cheap to compute.
* ***on every iteration, we're only going to train five of them or more generally,*** k + 1 of them, of k negative examples and one positive examples**. And this is why the computation cost of this algorithm is much lower because you're updating k + 1,** let's just say units, k + 1 binary classification problems.Which is relatively cheap to do on every iteration rather than updating a 10,000 way Softmax classifier.
* what you're doing is, ***you have a positive example, the orange and then juice. And then you will go and deliberately generate a bunch of negative examples,*** negative samplings, hence, the name negative sampling, with which to train four more of these binary classifiers.
* ***And on every iteration, you choose four different random negative words with which to train your algorithm on***



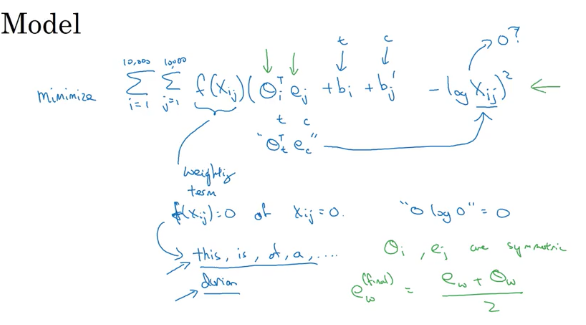
* So after having chosen the context word orange, how do you sample these words to generate the negative examples?
  + So one thing you could do is sample the words in the middle, the candidate target words. One thing you could do is sample it according to the empirical frequency of words in your corpus. So just sample it according to how often different words appears. But the problem with that is that you end up with a very high representation of words like the, of, and, and so on.
  + One other extreme would be to say, you use 1 over the vocab size, sample the negative examples uniformly at random, but that's also very non-representative of the distribution of English words.
  + So the authors, Mikolov et al, reported that empirically, what they found to work best was to take this heuristic value, which is a little bit in between the two extremes of sampling from the empirical frequencies, meaning from whatever's the observed distribution in English text to the uniform distribution. *And what they did was they sampled proportional to their frequency of a word to the power of three-fourths. So if f of wi is the observed frequency of a particular word in the English language or in your training set corpus*, then by taking it to the power of three-fourths, this is somewhere in-between the extreme of taking uniform distribution. And the other extreme of just taking whatever was the observed distribution in your training set.

## GloVe Word Vectors

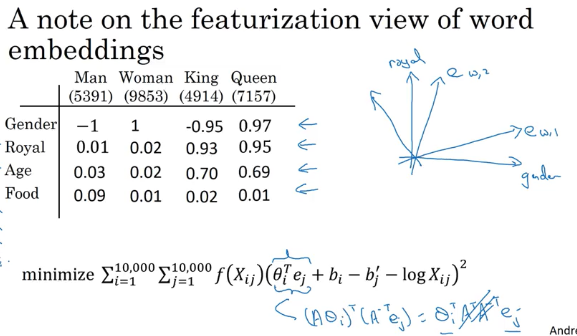


So, X\_ij is a count that captures how often do **words i and j appear with each other**, or close to each other.

So what the GloVe model does is, it optimizes the following.



So you just want to learn vectors, so that their end product is a good predictor for how often the two words occur together.

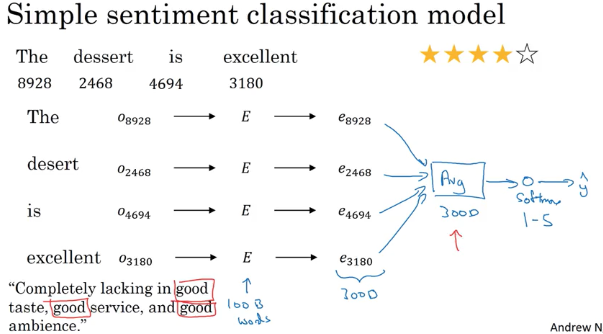


## Sentiment Classification

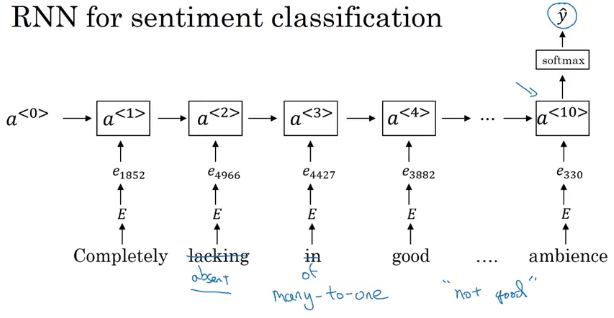
* Sentiment classification is the task of looking at a piece of text and telling if someone likes or dislikes the thing they're talking about.
* One of the challenges of sentiment classification is you might not have a huge label training set for it. But with word embeddings, you're able to build good sentiment classifiers even with only modest-size label training sets.



* there's O8928 which is a one-hot vector multiplied by the embedding matrix E, which can learn from a much larger text corpus. It can learn in embedding from, say, a billion words or a hundred billion words, and use that to extract out the embedding vector(e) for the word "the", and then do the same for "dessert", do the same for "is" and do the same for "excellent".



* And if this was trained on a very large data set, like a hundred billion words, then this allows you to take a lot of knowledge even from infrequent words and apply them to your problem, even words that weren't in your labeled training set.
* Output softmax as 1 star to 5 star selection. 5 possible outcomes.
* So notice that by using the average operation here, this particular algorithm works for reviews that are short or long because even if a review that is 100 words long, you can just sum or average all the feature vectors for all hundred words and so that gives you a representation, a 300-dimensional feature representation, that you can then pass into your sentiment classifier.
* So this average will work decently well. And what it does is it really averages the meanings of all the words or sums the meaning of all the words in your example.
* So one of the problems with this algorithm is it ignores word order in particular, this is a very negative review, "Completely lacking in good taste, good service, and good ambiance". But the word good appears a lot. This is a lot. Good, good, good. **So if you use an algorithm like this that ignores word order and just sums or averages all of the embeddings for the different words,** then you end up having a lot of the representation of good in your final feature vecto**r** and ***your classifier will probably think this is a good review even though this is actually very hars****h*. This is a one-star review.
* So here's a more sophisticated model which is that, instead of just summing all of your word embeddings, you can instead use a RNN for sentiment classification.



* The job of the RNN is to then compute the representation at the last time step that allows you to predict Y-hat. So this is an example of a many-to-one RNN architecture
* it will be much better at taking word sequence into account and realize that "things are
* lacking in good taste" is a negative review and "not good" a negative review unlike the previous algorithm, which just sums everything together into a big-word vector mush and doesn't realize that "not good" has a very different meaning than the words "good" or "lacking in good taste" and so on.