5-1_Sequence Model

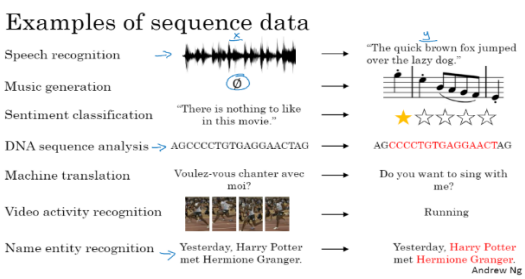
5.1-1. Why choose sequence model

sequence models, one of the most exciting areas in deep learning. Models like recurrent neural networks or RNNs have transformed speech recognition, natural language processing and other areas

supervised learning with label data X as training set:
cases:
1. input and output both are sequnce, length maybe different;
2. only one is sequence, the other not: Input/output

e.g for using sequence:
1. Speech recogniation: input audio,is sequence, output text is also sequence;
Sequence models such as a recurrent neural networks and other variations, you'll learn about in a little bit have been very useful for speech recognition.

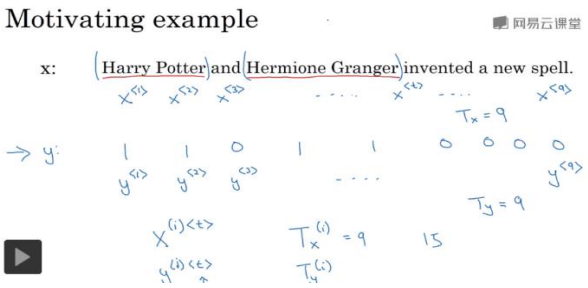
2. Music generation: only output specail sequence. Input can be nothing or a signal trigger.
3. Sentiment classification: input'human feeling feedback'-->star, sequence;
4. DNA sequence analysis: input DNA sequcne, output sequence result
5. Machine tranlation: one sequnce language to other language sequenced
6. Video activity recognition: sequence of pic -differnt gesture-->output activity reuslt : like runing
7. Name entity recognition: Give a sequence with name,-->output sequenct name in the input



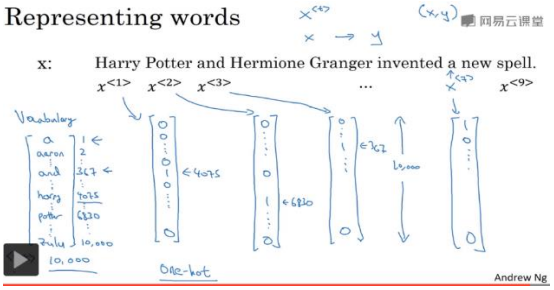
5.1-2 Notions used in sequence model

1. Motivating example:
Name entity recognition: recognize name/country/locaiton in different types of text, ect.
used by seach engine（搜索引擎）.: like in the past 24h peope name mentioned in news artical

E.G: Named-entity recognition
input vector: sentence words
output y vector: for each word whether it is part of a name:1 yes, 0 No.
Note: have more complicaton output y, not only tell if there is a name, but also tell the start and end of the name.



1.1Notes:
Input: X=[X_<1>, X_<2>, ...X_<t>, ...X_<Tx>]; Tx: lengh of input (number of input value);
Output: Y=[y_<1>, y_<2>, ...y_<t>, ...y_<Ty>]; Ty: lenght of output (number of output value)
t: index into the positions in the sequence.
input sample i: X_(i).
Training sample i, t postion: X_(i)<t>
Output of sample i, t postion: y_(i)<t>
Traing samples can have different length, -->length of training sample i: Tx_(i)
Output length of sameple i: Ty_(i)



2. Representative for word:

>1. Come up a vocabulary /dictionary
A list of words that will use in word representations. commercial application may have 30-50,000 size/words dictionary.
Build dictionary: once choosen dictionary size like 10,000, and then could look through training sets, find the top 10,000 occuring words. or look through some of the online dictionaries that tells what are the most common 10,000 words in the English Language.

>2. Use one-hot vector to represent each of the input word: x_<t>
1: in the dictionary position correspodnign to input x<t>; other postions 0;
if no this word/value in dictionary, take a new fake word UNK 'Unknow work' to represent it.

2. Representative for word:

Note:
1. one-hot representation of each word x_(i)<t> , is a vector: dimension is dictionary size.
2. Label y_(i)<t>: is also one-hot vector (output: one 1 and 0 is everywhere else)

3. The goal is given this representation for X to learn a mapping using a sequence model to then target output y.
Take it as a supervised learning problem, **with give labeled one-hot representation of both x and y.**

NLP is supervise learning, with training data label (x, y), to train a neural network mapping x to y.

5.1-3 Recurrent Neural Networks

1. why not a standard network?

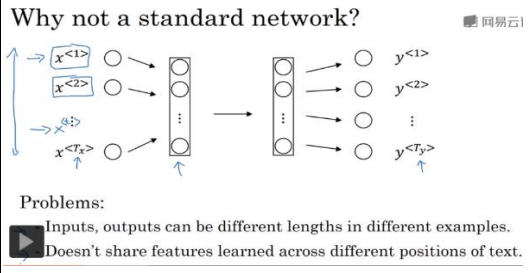
1. 1. Standard network: X[X_<1>, ...X_<Tx>]T-->network -->output Y[y<1>, ..y<Tx>]
Note: each input word x<t> is one-hot vector

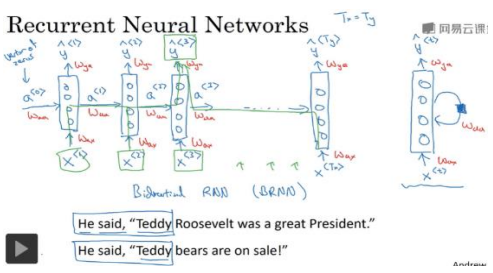
e.g:
input 9 words - each word is one-hot vector, feeding them into a standard neural network, maybe a few hidden layers, and then eventually had this output the nine values zero or one that tell whether each word is part of a person's name.
But this turns out not to work well .
(note: a classfire for one word, output 1/0 -is a name or not. then go through each word of input.
not work as could not consider words dependency in input)

Standard network Problem:
1. Inputs and outputs length could be differnt in differnet input example (x(i))
it's not as if every single example had the same input length Tx or the same output length. and even using padding /zero padingto make every input to that maxi length, this still does not seems a good representation.

2. Does not share features learned across different positions of text:
Features shared in diffrent postion means: if a value learned in input postion x<t> is predicted as part of name, then automatically predict it as a part of name when the same value occure in other postions in input x<t'>.-->kind of like convnet: things learned in one part of image also generize to other parts of the image (filter paramter same for all image postions).and we like a similar effects for sequence data as well.

3. high compution: W (features, Input : sum of all input x(i): each postion dimenson * postions in input x(i)), enormous number of parameters.
e.g: input examples's each feature x(i)_t, is dictornary size dimension vector (like dictionary size 1000, then x(i)_t dimension is (1000,1)), input dimenson : vocab size x word number





2. Recurrent Neural Network (RNN)

RNN method without above standard networ problem: length difference, feature share postion-<---weight same in each position, --->less paramete, compution cost less

2.1 RNN idea:
eg.: feed example i to neural network:
>1. Feed first word of example x<1>, get output y^<1>
>2 . Feed send word of example x<2> + a<1> , get output y<2>
a<1>: some info.of first word computed, activation value from time step 1.

...
>3. Feed j_th word of example + previous word activation a<(j-1>, get output y<j>

So at each time step, the recurrent neural network passes on previous time /word activation to the next time step for it to use.

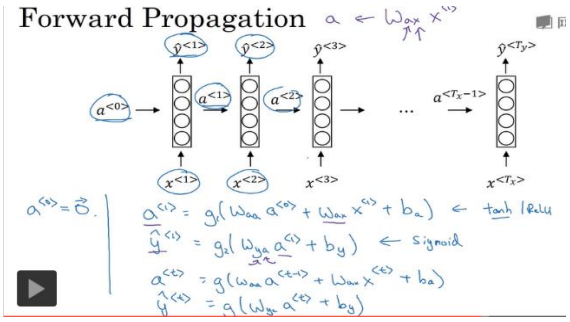
2.3 Advantage

>1. Parameters are shared
>>>1. Parameters used in each step are shared. Parameters:
Weight on input x<t> : Wax , same for every time step/ all postion x<t'>

>>>2. Horizental activation function are governed by parameter Waa: which is also same for every time step/ all postion x<t'>

>>>3. Output predictons, govenred by Wya: same for every time step/ all postion x<t'>

>2.each time step output, will consider both current step input x<t>, and info. in ealier time step: x<1>, ...x<t-1>.
e.g: in time step3, while making the prediction for y<3>, algorithm gets the information not only from X<3> but also the information from x<1> and x<2>: because the information on x<1>, x<2>, can pass through the horizental way/activation to help to prediction with y<3>.



2.5 RNN_Forward Propageation

>0. started off with the input a<0> =0 vectors.

Forward propagation:
>1.a<1>=g_a (Waa*a(0)+Wax * x(1) +ba)
>2. y<1>=g_y (**Wya * a<1>** +by)
...
time step t:
a<t>=g_a (Waa*a(t-1)+Wax * x(1) +ba)
y<t>=g_y (Wya * a<t> +by)

Activation funciton choice:
1. Activation function g_a : often is tanh , sometimes are Relu.
Have other ways of preventing the vanishing gradient problem.
(note: when to use Relu, tanh?)

2. Activation function g_y: depending on what output y is.
If it is a binary classification problem, then a sigmoid activation function, or it could be a softmax that have a k-way classification problem.
The choice of activation function here will depend on what type of output y you have.
e.g: name entity recognition task, output y^<t> is either 0 or 1, g_y could be a sigmoid activation function.

2. Recurrent Neural Network (RNN)

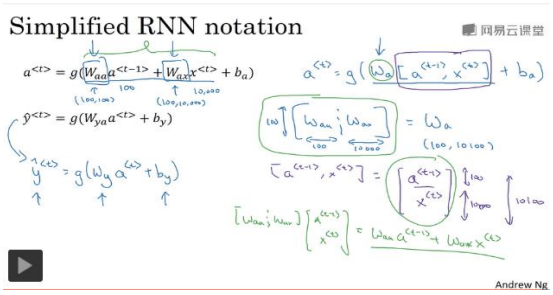
2.2 Note:
1. in this example length of input and output word are same Tx= Ty. Algorithm architure need to modify a little bit if Tx/=Ty.

2. to kick off the whole RNN, need made-up activation at time zero a<0>.
a<0> usually is 0 vector, Some researchers will initialized a <0> randomly. Could use other ways to initialize a<0>, but really having a 0 vector as the fake times 0's activation is the most common choice.

3. There are other type of neural network drawn-right pic, in which there is line to denote the recurrent connection, like a loop / shaded box to denote a time delay of one step.
Left pic is the unrolled diagram, more easier to understand.

2.4: Disadvantage:
only use info. earlier in the sequence to make a perdiction, **do not consider info. in the later sequence.**
While it is really helpful to know info. in both ealier and later sequence to make prediciton for current positon. will sove in the BRNN(bidirection recurrent neural network)
BRNN-Just have to make a quick modifications to these ideas later to enable, say, the prediction of y^<3> to use both information earlier in the sequence as well as information later in the sequence.

e.g: input "He said Teddy Roosevelt was a great president."
In order to decide whether or not the word Teddy is part of a person's name, it would be really useful to know not just information from the first two words but to know information from the later words in the sentence as well. because the sentence could also have been, "He said teddy bears are on sale." 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.



2.6 Simplifed RNN notation

for time step t:
a<t>=g_a (Waa*a(t-1)+Wax * x(t) +ba)
y<t>=g_y (Wy * a<t> +by)

>1. combine Waa, Wax to matrix Wa=[Waa| Wax], --> **a<t>=g_a (Wa[a(t-1), x(t)]+ba)**
Advantage: rather than carrying around two parameter matrices, Waa and Wax, we can compress them into just one parameter matrix Wa, and just to simplify our notation for when we develop more complex models.

Note:
1. Dimension: a<t> in different time step, have same dimension, = Waa. Wax row number.
e.g: a[t] is (n x1), input x<t> (n_d x1) n_d- dictionary size
-->Waa: n x n, always square matrix; Wax : n x n_d
-->Wa =[Waa| Wax]: n x (n+n_d)

> 2. [a(t-1), x(t)] = [a(t-1); x(t)]T: (n+n_d, 1)

>3. Wy = Wya

5.1-4 Backpropagation through time

when implement this in one of the programming frameworks, often, the programming framework will automatically take care of backpropagation. But it's still useful to have a rough sense of how backprop works in RNNs.

1. Forwardprop and Backpropagation

1. 1 Forwardprop

time step t:
a<t>=g_a (Waa*a(t-1)+Wax * x(1) +ba)
y<t>=g_y (Wya * a<t> +by)

1.2: Backprop:

>1. Loss function:

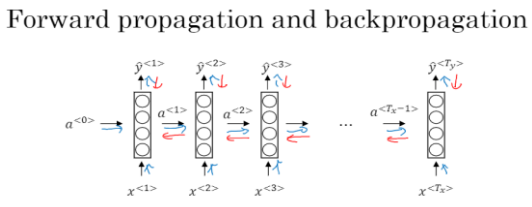
>>>1. **Loss function with a signel prediciton at a singel postion/time step** l<t>

e.g: define an element-wise loss , which is supposed for a certain word in the sequence.
if It is a person's name, so labeled y<t> is one, neural network outputs some probability y^t <t>, of the particular word being a person's name. So define lost function as the standard logistic regression loss,

So this is the loss associated with a single prediction at a single position or at a single time set, t, for a single word.

>>>2. **Total loss functio**n l: sum over all the individual losses per timestep . (Ty number)
Loss for the entire sequence L will be defined as the sum over all t: 1, 2, to, _x or T_y. T_x is equals to T_y in this example.
In a computation graph, to compute the loss: given y^<1>, y^<2>, ...y^<t>, ...y^<Ty> --> compute the loss for the them independently : l<1>, ...l<t>, ...l<Ty). and then sum them all up to compute the final L.

Forward + Backprop is the computation graph.



2. Backpropagation through time:

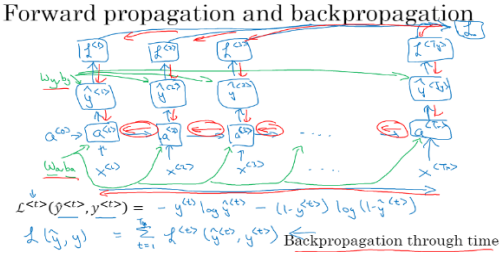
Backprop then just requires doing computations in the opposite directions:
da<t> =dy<t>*Wya.* g_y'(Wya * a<t> +by) +da<t+1>~~{ = dy<t+1>*Wya.* g_y'(Wya * a<t+1> +by)}~~ +Waa.*g'_a
>1. Vertical direction: da<t> =dy<t>*Wya.* g_y'(Wya * a<t> +by)
Total loss function -->each single loss function-->output y<t>--> activation a<t>

>2. Horizental direction: activation a<t>---->a[t-1]

Note:
1. In this back propagation procedure, the most significant message or the most significant recursive calculation is Horizental direction: activation a<t>---->a[t-1]

2. Horizental back prop goes from right to left, that's reason this algorithm called backpropagation through time. The motivation for this name is that for forward prop, you are scanning from left to right, increasing indices of the time t, whereas, the backpropagation, from right to left, kind of going backwards in time.

Summary:
So far, only seen this main motivating example in RNN in which the length of the input sequence was equal to the length of the output sequence.



5.1-5 RNN architecure type

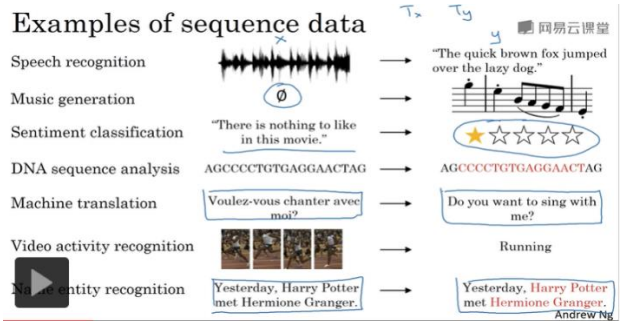
Have seen an RNN architecture where the number of inputs, Tx-, is equal to the number of outputs, Ty. It turns out that for other applications, Tx and Ty may not always be the same. it turns out that we could modify the basic RNN architecture to address all of these problems.
Here will see a much richer family of RNN architectures.

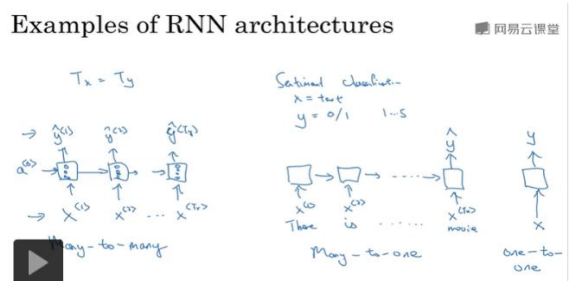
RNN architecture range

1: many to many: Input length= output lenght: Tx=Ty

2. Many to 1: only output AT the last time step when have the entire input sequence.
e.g: address sentiments classification
input x might be a piece of text, such as it might be a movie review that says, "There is nothing to like in this movie." So x is going to be sequenced, and y might be a number from 1 to 5, or maybe 0 or 1.

In this case, we can simplify the neural network architecture as follows:
>1. input the words one at a time.
If the input text was, "There is nothing to like in this movie." So "There is nothing to like in this movie," would be the input.
>2. Then rather than having to use an output at every single time-step, just have the RNN read into entire sentence and have it output y at the last time-step.



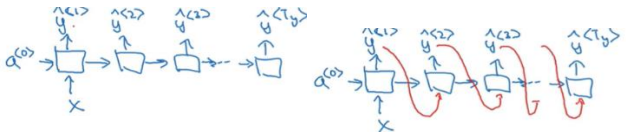


RNN architecture range

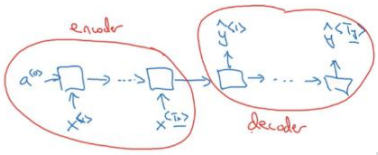
3. One to many:
e.g music generation:
output is a number of notes corresponding to a piece of music, input x could be an integer telling what kind of music want, or if don't want to input anything, x could be a null input, could always be the vector zeroes as well.

Neural network architecture:
>1. input x, a<0>have your RNN first output y<1>
>2. with no further inputs, only activation a<1> in first step, output y<2>
...
>Tx: with no input X, only activation a<Ty-1>, output y<Ty>

Note: One technical used when you're actually generating sequences, often take the previous time step output y^<t-1> as activation a<t-1> and feed it to the next layer.
(then oupt put y<t> is like one-one architecture, input is only y<t-1>, output only depending on previous step output).



CRNN architecture:



4. Many to many: yet Tx = Ty (like machine translation). encoding (having read input sequence), then decode (output translation)

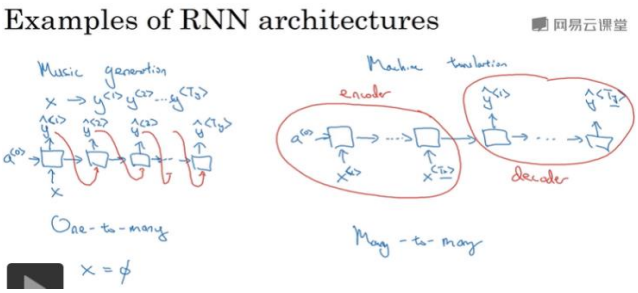
In the many-to-many problem, there are cases input and output have different length.
e.g: machine translation, the number of words in the input sentence, and the number of words in the output sentence, those sentences could be different lengths.

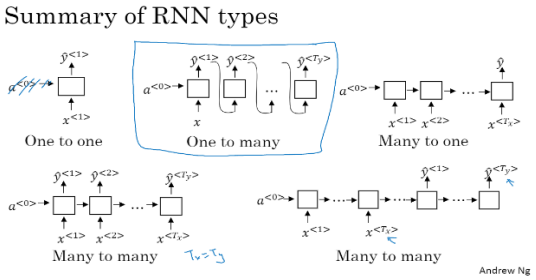
Architecture:
>1. Encoding: reading the entire sentence.
first reading the entire input. and having done that, you then go to decoding.

>2. Decoding: have the neural network output the translation.

with this architecture, Tx and Ty can be different lengths. And again, could draw on the a(0)

5 . One to one:
Remove a<0>, this is just a standard generic neural network, don't need an RNN for this.





Summarize: the wide range of RNN architectures

there is one-to-many: like a music generation or sequenced generation;
there's many-to-one: like sentiment classification. Where you might want to read as input all the text with a movie review. And then, try to figure out that they liked the movie or not.
There is many-to-many: like the name entity recognition, where Tx is equal to Ty.
there's other version of many-to-many: like machine translation, Tx and Ty no longer have to be the same.

Now learned most of the building blocks, the building are pretty much all of these neural networks except that there are some subtleties with sequence generation, which is what we'll discuss in the next video.

5.1-6 Language Model and sequence generation

Language modeling is one of the most basic and important tasks in natural language processing. There's also one that RNNs do very well.

1. What is language modelling

1.1 Example: Input speech-->choose the best output text

Output choice 1: the apple and pair salad.
Output choice 2: the apple and pear salad.
You probably think the second sentence is much more likely, and in fact, that's what a good speech recognition system would help with even though these two sentences sound exactly the same.

The way a speech recognition system picks the second sentence is by using a language model which tells it what the probability is of either of these two sentences:
e.g: P(choice 1) = 3.2 x 10 ^-13, p (choice 2) = 5.7 x10 ^-10.
With these probabilities, the second sentence is much more likely by over a factor of 10 ^3 compared to the first sentence.
And that's why speech recognition system will pick the second choice.

1. What is language modelling

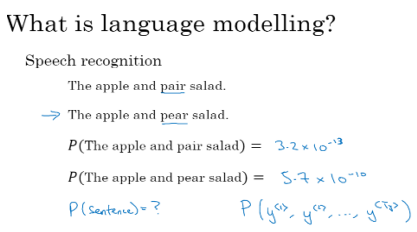
1.2: Language model

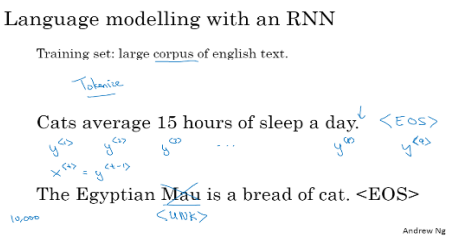
>1. **Language model Function:** given any sentence, language model is to tell what is the probability of a sentence, of that particular sentence.

>2. Probability of sentence: means, 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. 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?

Note:
Language model is a fundamental component for both speech recognition systems, as well as for machine translation systems where translation systems wants output only sentences that are likely.

>3. Language model basic job:
Input a sentence, y<1>, y<2> up to y<Ty>. language model estimates the probability of that particular sequence of words:
p (y<1>, y<2> , ... y<Ty>)





2. Building language model with RNN

2.1 Build Id language model:

>1. first need Traing set: comprising a large corpus of english/franch , or text from whatever language you want to build a language model.
Word corpus:NLP terminology means a large body /a very large set of english text, of english sentence.

E.g of one training senence: 'Cat average 15 hours of sleep a day'
>2. **Tokenize training sentence:**
>>>1. bulid vocabulary/dictionary:
>>>2. Map the each of the workd in training sentence to one-hot vector or the indices to vocabulary

>3.Commonly also add another extra token 'EOS', to the end of every sentence in the training set.
EOS: stands for sentence end and appended to the end of every sentence in the training set , so could help you figure out when a sentence ends.
in this way training sentence length add one more: y<Ty> = EOS.

2. Building language model with RNN

2.1 Build Id language model:

Note:
1. If want to treat the period （标点） or other punctuation as explicit token, then could add them to vocabulary as well.

2. If some of the words in training set, are not in vocabulary, could replace this word with a unique token UNK - stands for unknown words and just model the chance of the unknown word instead of the specific word.
e.g: Mau in sentence not found in dictionalry, could replace with UNK in sentence, and then model the probability of this 'modified' sentence.

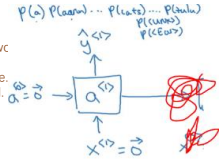
3. RNN_Mode: l chance of difference sequences

e.g: one of training sentence is : cat average 15 hours of sleep a day. <EOS.

RNN architecture:
>1. At time 0 : make a soft max prediction to try to figure out what is the probability of the first words y^<1>" P(first word=?)
computing activation a<1> , as a function of :
some inputs x<1>---just be set it to 0 vector: x<1>= 0 vector + and the previous a<0>, by convention, also set that to vector zeroes: a<0> = 0 vector.

a<1> function: make a soft max prediction to try to figure out what is the probability of the first words y^<1>.

Step time 0: get y<1>: using softmax to predict the first one word:
e.g: what's the chance that the first word is a, ' cats', ' Zulu', unknown word,.
y^<1> is output acc. to a softmax, it just predicts what's the chance of the first word being (in our example, it will end up being the word cats)



Note::
1. Softmax function is trying to predict what is the probability of any wc in the dictionary.
2. y<1> is a vocabulariy size softmax output: vector of vocabulary size.
Predict the probability of each word in vocabulary to be the first word.
e.g: vocabulary have 10,000 words, then softmax have 10,000 output.

3. RNN_Mode: l chance of difference sequences

e.g: one of training sentence is : cat average 15 hours of sleep a day. <EOS.

RNN architecture:

>2. Time1: has activation a<1>: try to figure out what is the second word: p(2nd word=?| cats)
Input: correct first word y<1> , = x2,
RNN here use softmax functions to predict the chance of a being whatever the second word it is, given what had come previously (correctly previous word).

Note:
1. softmax function also used;
2. y<2> is vecotor of vocabulary size;

>3. Continue next step till the end of sentence, which is the EOS token.
Function: predivce the the chance of each vocabulary word, given all the correct word in ealier sequence**: p (? | cat average 15 hours of sleep a day.)**
Hopefully it will predict that there's a high chance of it, EOS

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 (chance of each vocabulary word being the i_th word in sentence).
And so this RNN learns to predict one word at a time going from left to right.

4. Train neural nework:_cost funciton defination

>1. Cost functionat a certain time step t, l (y<t>, y' $\mathcal{L}(\hat{y}^{<t>}, y^{<t>}) = -\sum_i y_i^{<t>} \log \hat{y}_i^{<t>}$

True word was y<t>: one-hot vector, vocabulary size;
, prediction y^<t> : vector of vocabulary size.

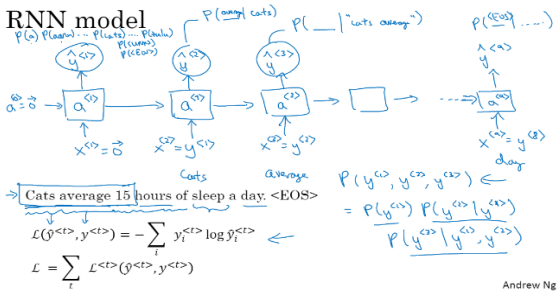
-->loss at this time step (soft max loss function) : y_i <t> * log (y_i^<t>), sum over all vector size.

>2. Overall loss : sum over all time steps of the loss associated with the individual predictions.
Summarize:
And if you train this RNN on the last training set, what you'll be able to do is given any initial set of words, such as cats average 15 hours of, it can predict what is the chance of the next word.

e.g: new sentence: y= y<1>, y<2>, y<3>, with just a three words. to get this three words sentence probability.
The way to figure out what is the chance of this entire sentence would be:
The first softmax tells he chance of y<1>. That would be this first output.
And then the second one can tell given the correct first work y<1>, what's the chance of p (y<2> |y<1>)
And then the third one tells what's the chance of y(3) given correct y<1)> & y<2: p (y<3> |y<1>, y<2>) .

---> multiplying out these three probabilities, end up with the probability of the three-word sentence p(y<1>, y<2> , y<3>).
Sentence probability:
P(y<1>, y<2>, ...y<Ty>)=P (y<1>) * P(y<2>|y<1>)* P(y<3>|y<2>, y<1>)....P(y<t>|y<t-1>, y<t-2>, ...y<1>)...*P(y<Ty>|y<Ty-1>, .y<1>)

So that's the basic structure of how you can train a language model using an RNN.
(note: language model: is many-many RNN, Tx=Ty (Tx +x1:0 = Ty + (EOS)); only output y^(t) in each step is the chance of each vocab word being the output)



5.1-7 Sampling novel sequence_Recurrent neural network

Afte train a sequence model, one of the ways can informally get a sense of what is learned is to have a sample novel sequences.

1. Sampling a sequence from a trained RNN

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.

To sample, need to do something slightly different:
>1. Time 0: sample the first word want model to generate.
>>>1. Inpute x<1>=0, a<0>=0, -->output softmax probability , vector of vocabularty size.
>>>2. Then randomly sample according to this soft max distribution.
e.g: 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 take this vector y^<1> and use,like the numpy command np.random.choice to sample according to distribution defined by this vector probabilities, and that lets you sample the first words.

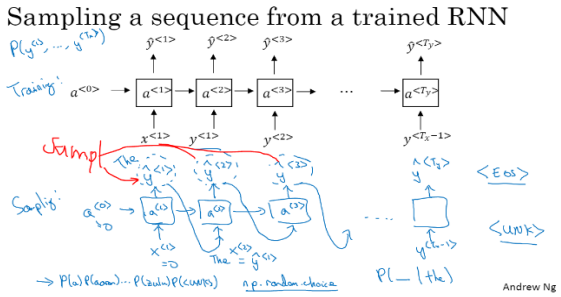
(note: could use a trained language model RNN to generate novel sequence)

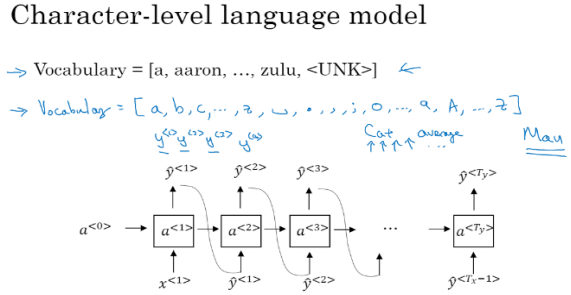
>2. Time1: sample y^<2>, given sampled y^<1>.
Input: instead using y<1> as input, take the y^<1> that just sampled, then this soft max will make a prediction for what is y^< 2>
e.g: sample the first word to be 'the, then you pass to time 1 as x<2>= y^<1>.
And now you're trying to figure out what is the chance of what the second word is, given that the first word is 'the'.
Then again use this type of sampling function to sample y^<2>.

>3. Go to next step: take whatever choice had sampled and pass that to next time step.
keep going until you get to the last time step.

Note:
1. how to know when the sequence ends.
>1. If the end of sentence token is part of vocabulary, then could keep sampling until generate an EOS token.
that tells you have hit the end of a sentence and can stop.
>2. Or alternatively, if do not include end token in vocabulary , then you can also just decide to sample 20 words or 100 words or something, and then keep going until you've reached that number of time steps

2. This particular procedure will sometimes generate an unknown word token.
>1. If want to make sure that algorithm never generates this token, could just reject any sample that came out as unknown word token and just keep resampling from the rest of the vocabulary until you get a word that's not an unknown word.
>2. Or can just leave it in the output as well if you don't mind having an unknown word output.





2. Character-level language model

Above is building a words level RNN, by which means the vocabulary are words from English.

2.1 Character-level language Defination
Depending on application, one thing you can do is also build a character level RNN: in which vocabulary will just be the alphabets.

Character-level language vocabulary: could be a up to z, and as well as maybe space, punctuation , the digits 0 to 9. And if want to distinguish the uppercase and lowercase, can include the uppercase alphabets as well.

2.2 Build character vocabulary: can just look at training set and look at the characters that appears there and use that to define the vocabulary.

2.3: Language model output sequence:
Output sequence y<1>, y<2>, y<3>, would be the individual characters in training data, rather than the individual words in training data.

e.g: traning sentence is: cats average 15 hours of sleep a day.
In this example, c would be y<1>, a would be y<2>, t will be y<3>, the space will be y<4> and so on.

2.4: a character level language model pros and cons.

>Pros: Don't ever have to worry about unknown word tokens
In particular, a character level language model is able to assign a sequence like MAU, a non-zero probability. Whereas if MAU was not in vocabulary for the word level language model, you just have to assign it the unknown word token.

> Cons:
>>1. main disadvantage of the character level language model is that end up with much longer sequences.

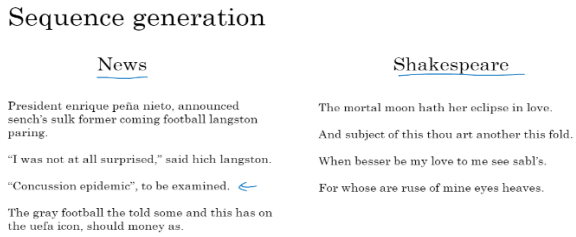
So many english sentences will have 10 to 20 words but may have many, many dozens of characters.
so character language models are not as good as word level language models at capturing long range dependencies between how the the earlier parts of the sentence also affect the later part of the sentence.

>>2. character level models are also just more computationally expensive to train.

Character level model tends to be much harder, much more computationally expensive to train, so they are not in widespread use today.Except for maybe specialized applications where you might need to deal with unknown words or other vocabulary words a lot. Or they are also used in more specialized applications where you have a more specialized vocabulary.

The trend in natural language processing is that for the most part, word level language model are still used. but as computers gets faster there are more and more applications where people are, at least in some special cases, starting to look at more character level models.

So under these methods, can build an RNN to look at the purpose of English text, build a word level, build a character language model, sample from the language model that have trained.



3. Sequence generation exampel

E.g:1_News
Tex generated from a character-level language model. the texs looks vaguely like news text, not quite grammatical, but maybe sounds a little bit like things that could be appearing news.

E.g2_Shakepeare
Language model was trained on Shakespearean text, and then it generates stuff that sounds like Shakespeare could have written it. The mortal.

Summary:

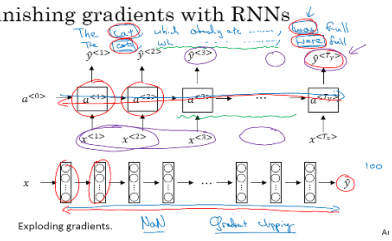
So this is it for the basic RNN, and how you can build a language model using it, as well as sample from the language model that you've trained.
(basic RNN-->Language model -->sampling from Lamguage modle: genrating noval sentence
Language mdoel: modified on basic RNN, input x<t>= correct y<t-1>
>>>softmax predict the next word given preivous words: p(y(t+1> | y<1>, ..y<t>)(note: loss is: y<t>-one vector * prediction y^<t>)
>>>for generating y^<t>: input correct y<t-1>=x<t>, and a<t-1>; while y^<1> excpet, input x<1>=0, a<0>=0

Samling/generating noval sentecne: use a trained language model, while setting x<t>= sampled y^<t-1>)

In the next few videos, I want to discuss further some of the challenges of training RNNs, as well as how to adjust some of these challenges, specifically vanishing gradients by building even more powerful models of the RNN. So in the next video let's talk about the problem of vanishing the gradient and we will go on to talk about the GRU, Gate Recurring Unit as well as the LSTM models.

5.1-8 Vanishing gradients with RNNs

You've learned about how RNNs work and how they can be applied to problems like name entity recognition, as well as to language modeling, and you saw how backpropagation can be used to train in RNN. It turns out that one of the problems with a basic RNN algorithm is that it runs into vanishing gradient problems.



1. Vanishing gradients with RNNs

Basic RNN run into vanishing gradient problems.

1.1 Language long-term dependecy:
Language can have very long-term dependencies, where it worked at this much earlier can affect what needs to come much later in the sentence.

E.g: below two sentence, cat or cats can affect to use was or were.
The cat, which alreay ate....., was full;
The 'cats', which already ate...., 'were' full.

1. Vanishing gradients with RNNs
Basic RNN run into vanishing gradient problems.

1.2: Basics RNN is not very good at capturing very long-term dependencies.

>1. Vanishing gradient probelm in deep neural network:
In a very, very deep neural network like 100 layers or even much deeper, then you would carry out forward prop, from left to right and then back prop. And in a deep neural network, the gradient fromt 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.

>2. RNN Vanishing gradient problem:
RNN has a similar problem: 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, it means: 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 in English, this stuff in the middle could be arbitrarily long, then might need to memorize the singular/plural for a very long time before you get to use that bit of information.

1.2: Basics RNN is not very good at capturing very long-term dependencies

Summarize:
---> due to vanishing gradient, basic RNN model has many local influences: , meaning output like y^<3> is mainly influenced by values close to y^<3>, and it's difficult to be strongly influenced by an input that was very early in the sequence.
And this is because whatever the output is, whether this got it right or wrong, it's just very difficult for the area to backpropagate all the way to the beginning of the sequence, and therefore to modify how the neural network is doing computations earlier in the sequence.
So this is a weakness of the basic RNN algorithm.

2 Language model exbloring gradient problem

Neural network exploring gradients:

>1. Also have exploring gradient problem
when doing back prop, the gradients should not just decrease exponentially, they may also increase exponentially with the number of layers you go through.

>2. Exploring gradient problme easy to find:

It turns out that vanishing gradients tends to be the bigger problem with training RNNs, although when exploding gradients happens, it can be catastrophic because the exponentially large gradients can cause your parameters to become so large that your neural network parameters get really messed up, so easier to spot because the parameters just blow up and you might often see NaNs, or not a numbers, meaning results of a numerical overflow in your neural network computation.

>3. Exploring gradient solution: apply gradient clipping.
Gradient clipping: if gradient vector is bigger than some threshold, re-scale some of your gradient vector so that is not too big.
Clipping method: could clips according to some maximum value.
(note: reduce gradient step by clipping, direction no change)

Summarize:
So there is exploding gradients,the derivatives do explode or you see NaNs, then just apply gradient clipping, and that's a relatively robust solution that will take care of exploding gradients.
But vanishing gradients is much harder to solve and it will be the subject of the next few videos.

3. Summary:

While training of very deep neural network, you can run into a vanishing gradient or exploding gradient problems with the derivative, either decreases exponentially or grows exponentially as a function of the **number of layers.** And in RNN processing data over like a thousand times sets, over 10,000 times sets, that's basically a 1,000 layer or 10,000 layer neural network, and so, it too runs into these types of problems. Exploding gradients could sort of address by just using gradient clipping, but vanishing gradients will take more work to address.

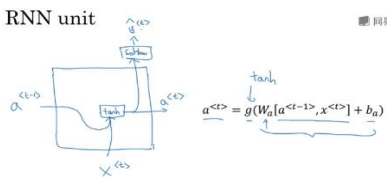
So what we do in the next video is talk about GRU, the gate recurrent units, which is a very effective solution for addressing the vanishing gradient problem and will allow your neural network to capture much longer range dependencies.

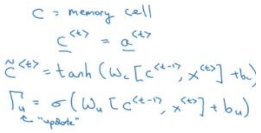
5.1-9 Gate Recurrent Unit (GRU)

Gated Recurrent Unit is a modification to the RNN hidden layer that makes it much better capturing long range connections and helps a lot with the vanishing gradient problems.

1. RNN unit

Visulization of the RNN unite of the hidden layer: right pic.
a<t> -->softmax -->output y^<t>.





2. GRU (Simplified) (modifed on RNN):
Purpose: captuirng long term dependency,prevent gradient vanishir

2. 1. New cell: memory cell: c<t>

Like previous sentence example, need to remember the cat was singular, to make sure you understand why that was rather than were. So the cat-was r or the cats -werer.

Memery cell function: c
So as we read in this sentence from left to right, the GRU unit is going to have a new variable called c,standsfor memory cell, to provide a bit of memory to remember, like , whether cat was singular or plural, so that when it gets much further into the sentence it can still work under consideration whether the subject of the sentence was singular or plural.

Note:
1. in GRU c<t>=a<t> (in LSTM c<t> and a<t> is different value)

2.2. : Candidate for replacing c<t>: C~<t>= tanh (Wc [c<t-1>, x<t>] + bc);
At every time-step, we're going to consider overwriting the memory cell c<t> with a value c ~ <t>

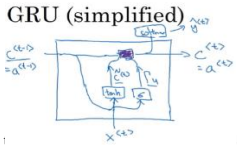
2.3. Gate : Gu: (between 0-1): sigmoid (Wu [c<t-1>, x<t>] + bu):

Gu will be a value between(0,1). to develop intuition about how GRUs work, think it as being always 0 or 1.
In practice, compute Gu with a sigmoid function, and it's value is always between (0,1). And for most of the possible ranges of the input (too large or small), the sigmoid function is either very, very close to zero or very, very close to one. So for intuition, think of gamma Gu as being either 0 or 1 most of the time.

2.5 GRU unite picture

Inpute: x<t>, c<t-1> = a<t-1>
Output:
C~<t> = tanh (Wc [c<t-1>, x<t>] +bc)
Gu = sigmoid ((Wu [c<t-1>, x<t>] +bu)

C<t> = Gu * C~<t> + (1-Gu) * C<t-1>
y<t> = softmax (Wy * C<t> + by)

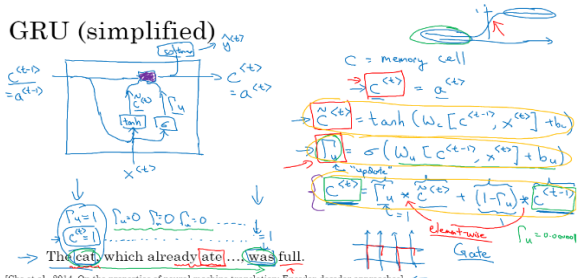


GRU unite takes x<t>, c<t-1> as input and together ate c~<t> and then the new value for the memory cell c<t> = a<t>. which could also use soft max or something to make some prediction for y<t>.

Above is he GRU unit or at least a slightly simplified version of it.
And what is remarkably good at is through the gates deciding that when you're scanning the sentence from left to right say, that's a good time to update one particular memory cell and then to not change, not change it until you get to the point where you really need it to use this memory cell that is set even earlier in the sentence.

2.6. Help vanishing gradient:
Because gamma Gu can be so close to 0, then it doesn't suffer from much of a vanishing gradient problem:
The gate Gu is quite easy to set to 0, as long as sigmoid content is large negative. in this case, c<t> = c<t-1> : this is very good at maintaining the value for the cell. c<t> is maintained pretty much exactly even across many many many many time-steps.
So this can help significantly with the vanishing gradient problem and therefore allow a neural network to go on even very long range dependencies, such as a cat and was related even if they're separated by a lot of words in the middle.

(note: backward prop: dc<t> = f(dc<t+1>): nearly: dc<t> ~==dc<t+1> due to signmoid and tanh function derivative: s' = s(1-s),
s' = (1-s^2) ... =1 when activation close to 0)



2.4 Key part of GRU: memeory cell c will be update by candidate c~, and gate Gu will decide whether or not acutally update c with c~: c<t>=Gu * c~<t> +(1-Gu) * c<t-1>

We have come up with a candidate C~<t>, where we're thinking of updating c<t>. and the gate Gu will decide whether or not we actually update it.

e.g: The cat, which already ate.... was full:
And so the way to think about it is:
>1. In 'cat' time step, memory cell C<t> is set to either 0 or 1 depending on whether the word you are considering, really the subject of the sentence is singular or plural (e.g: if sentence subject cat is sigular, set c<t>=1, otherwise set c<t>=0).
>2. then the GRU unit would memorize the value of the c<t> all the way until 'was' time step,
where c<t> still =1, tells sentence object is singulas so use the choice 'was'.

>1. The job of the gate Gu, is to decide when to update c<t>:
e.g: when reach 'cat' time step, realize this is sentence ojbect and updatet c<t>=1, then go to next time step , no need to udpdate c<t>, and thus set Gate Gu =0. Until reach 'was' time step, use c<t> to choice 'was'.
then don't need to memorize anymore, I can just forget that.

>2.Gate formulation: c<t>=Gu * c~<t> +(1-Gu) * c<t-1>
if Gu =1, new value of c<t> = c~<t>
e.g Gu =1 in 'was' time step, then for all of these values in the middle, Gu=0: do not update c<t>, just hang onto the old value till 'was' time step: c<t> it still memorizes, the cat was singular.
if Gu=0, new value of c<t> = c<t-1> old value.

>2.7 : matrix dimension:

While implementing GRU:
>1. C<t> can be a vector:
e.g: if have 100 dimensional or hidden activation value then c<t> can be a 100 dimensional.
>2. C~<t>, Gu <t> : are the same dimension with C<t>

>3. Formual: C<t> = Gu * C~<t> + (1-Gu) * C<t-1>
actually element wise multiplication between Gu, C~<t> and Gu , C<t-1>.

e.g: if gate Gu is 100 dimensional vector, what it is really a 100 dimensional vector of bits, the value is mostly zero and one. That tells of this 100 dimensional memory cell which are the bits you want to update.

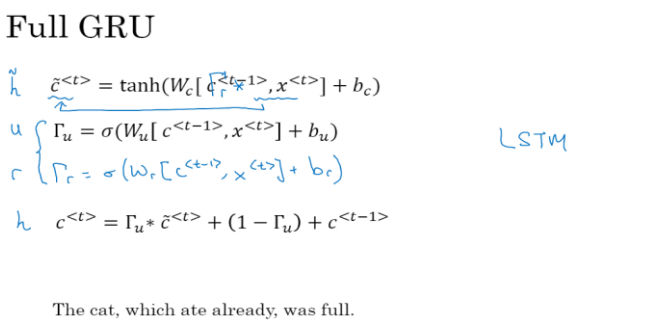
Note:
1. in practice gate Gu won't be exactly zero or one. Sometimes it takes values in the middle as well but it is convenient for intuition to think of it as mostly taking on values that are exactly zero, or pretty much exactly one.

2. these element wise multiplications just tells GRU which are the dimensions of memory cell vector to update at every time-step.
So you can choose to keep some bits constant while updating other bits:
e.g; maybe you use one bit to remember the singular or plural cat and maybe use some other bits to realize that you're talking about food.
You can use different bits and change only a subset of the bits every point in time.

Here is actually a slightly simplified GRU unit.

3. Full GRU unit
only one change, that is for c~<t>: C~<t>= tanh (Wc [**Gr*** c<t-1>, x<t>] + bc);

One more gate Gr added: Gr = sigmoid (Wr * [c<t-1>, x<t>] + bc)
Gr tells how relevant is c<t-1> to computing the next candidate for c<t>.



Summary:
There are multiple ways to design these types of neural networks. And why do we have gate Gr, Why not use a simpler GRU.. So it turns out that over many years researchers have experimented with many, different possible versions of how to design these units, to try to have longer range connections, to try to have more the longer range effects and also address vanishing gradient problems.

And the GRU is one of the most commonly used versions that researchers have converged to and found as robust and useful for many different problems.

And the other common version is called an LSTM which stands for Long Short Term Memory. GRUs and LSTMs are two specific instantiations of this set of ideas that are most commonly used.

Note: there could be diffrent notation used at academic literature, like instead c<t>, use h<t> etc.

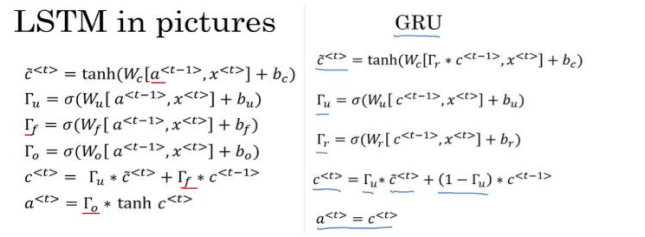
So that's it for the GRU, for the Gate Recurrent Unit. This is one of the ideas in RNN that has enabled them to become much better at **capturing very long range dependencies** has made RNN much more effective. the other most commonly used variation of this class of idea is something called the LSTM unit, Long Short Term Memory unit.

5.1-10 LSTM (Long short term memory) Unit

The gated recurrent units, and how that can allow you to learn very long range connections in a sequence. The other type of unit that allows you to do this very well is the LSTM or the long short term memory units, and this is even more powerful than the GRU

1. GRU and LSTM

The LSTM is an even slightly more powerful and more general version of the GRU, and is due to Sepp Hochreiter and Jurgen Schmidhuber. And this was a really seminal paper, a huge impact on sequence modelling and Deep Learning community.



1.1 LSTM Compare vs GRU
>1. c<t> not equal to a<t>,
>2. candidate for replacing c<t>: C~<t>= tanh (Wc [a<t-1>, x<t>] + bc), no relavent gate Gr
>3. **Update Gate Gu** is same;
>4. Have one more gate: **forget gate Gf** to control c<t>;Gf= sigmoid (Wf [a<t-1>, x<t>] + bf);

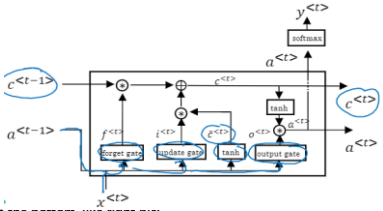
>5. Have another one more gate: **outputt gate Go** = sigmoid (Wo [a<t-1>, x<t>] + bo);

>6. c<t>= Gu * c~<t> + Gf * c<t-1>
This is a vector-vector element-wise multiplication, have a separate forget gate Gf : gives the memory cell the option of keeping the old value c<t-1> and then just adding to c~<t>. So, use a separate update and forget gates.

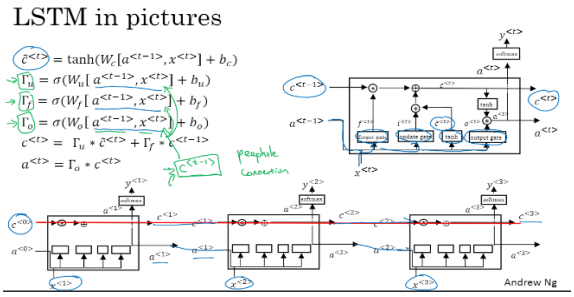
>7. a<t> = Go *tanh c<t> (why make a<t> !=c<t>?)
>8. y<t> = softmax (a<t>)

these equations govern LSTM behavior

2. LSTM unit and network picture



simplified to diagrams a little bit in the bottom, like right pic:



there are two horizental lines from left to right:
>1. C<0>------------------>c<t>---------------------->c<Ty>
>2. a<0>------------------>a<t>---------------------->a<Ty>

the cool thing is: for the top horizental line c<0>----c<t>: , so long as set the forget Gf and the update gate Gu appropriately, it is relatively easy for the LSTM to have some value c <0> and have that be passed all the way to the right to have maybe, c<3> = c<0>.
And this is why the LSTM, as well as the GRU, is very good at memorizing certain values even for a long time, for certain real values stored in the memory cell even for many, many timesteps.

(note: all the gate decided by activation and input: a<t-1>, x<t>, while a<t-1> = f(c<t-1>
---> gate actually decided by: c<t>, x<t>)

3. Othe version LSTM

The most common one is that instead of just having the gate values be dependent only on a<t-1>, & x<t>, sometimes, people also sneak in the values c<t-1> as well. **This is called a peephole connection.**

Peephole connection: means the gate values may depend not just on a<t-1>, & x<t>, but also on the previous memory cell value c<t-1> .
And the peephole connection can go into all three of these gates' computations.
Gu = sigmoid (Wu [a<t-1>, x<t>, **c<t-1>** + bu)
Gf = sigmoid (Wf [a<t-1>, x<t>, **c<t-1>** + bf)
Go = sigmoid (Wo [a<t-1>, x<t>, **c<t-1>** + bo)

Note:
 One technical detail is that these are, like 100-dimensional vectors. So if you have a 100-dimensional hidden memory cell unit, and the fifth element of c<t-1> affects only the fifth element of the corresponding gates, so that relationship (c<t-1>, and gates) is one-to-one, where not every element of the 100-dimensional c<t-1> can affect all elements of the case. But instead, the first element of c<t-1> affects the first element of the case, second element affects the second element, and so on.
If you ever read the paper and see someone talk about the peephole connection, that's when they mean that c<t-1> is used to affect the gate value as well.

Summary:
>1. When to choose GRU or LSTM:
 There isn't widespread rule in this. LSTMs actually came much earlier, and then GRUs were relatively recent invention that were maybe derived as Pavia's simplification of the more complicated LSTM model.
 Researchers have tried both of these models on many different problems, and on different problems, different algorithms will win out. So, there isn't a universally-superior algorithm which is why I want to show you both of them
.
Advantage of the GRU: 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.
LSTM: is more powerful and more effective since it has three gates instead of two.

If need to choice one, LSTM has been the historically more proven choice, most people today will still use the LSTM as the default first thing to try.
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.

5.1-11: Bidirectional RNN (BRNN)

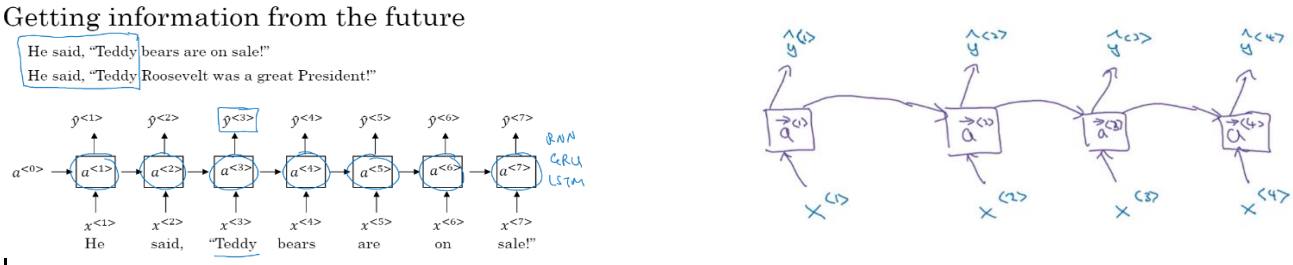
By now, you've seen most of the cheap building blocks of RNNs.
But, there are just two more ideas that let you build much more powerful models:
One is bidirectional RNNs, which lets you at a point in time to take information from both earlier and later in the sequence, so we'll talk about that in this video.
 And second, is deep RNNs.

1. Getting information from the future

1.1 Unidirectional or forward directional only RNN :
e.g:
He said "Teddy bears are on sale!"
He said " Teddy Roosevelt was a great President!"
-->To recognize name in sentence.
One of the problems of this 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, which doesn't tell you if they'll talking about Teddy bears or talk about the former US president, Teddy Roosevelt.

So this is a unidirectional or forward directional only RNN., whether these cells are standard RNN blocks or GRU units or LSTM blocks. all of these blocks are in a forward only direction.

A bidirectional RNN or BRNN, could fix this issue.



2. Bidirectional RNN (BRNN)

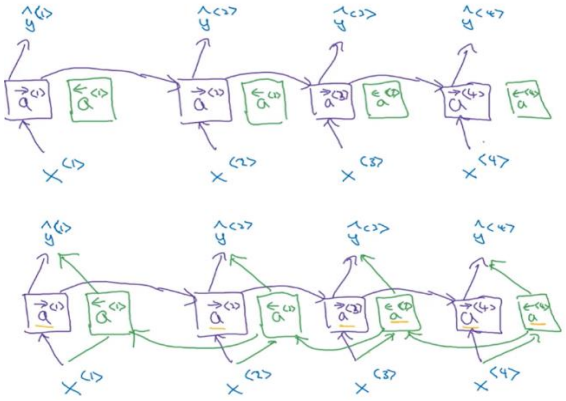
e.g :
Inpute: x<1>, x<2>, x<3>, x<4>
activation in forward: a<1>, a<2>, a<3>, a<4>
activation in backward: a<1>_b, a<2>_b, a<3>_b, a<4>_b
output: y<1>, y<2>, y<3>, y<4>

2.1: Forward prop: same as basic RNN:
Horizentally: **a<t-1>, x<t> ----->a<t>**
vertically: **a<t>** (f(x<t> ,a<t-1>)) **--->y<t>**

2.2: Backwrod prop:
Horizentally: **a<t>_b, x<t-1> ----->a<t-1>_b**
vertically: a<t-1>_b --->y<-1t>

And this a <t> activation backward connections will be connected to each other going backward in time.

the forward and backward network defines a a cyclic graph （无环图):
Given an input sequence x<1> ---x<4>, forward process first compute a<1> forward , then to a<2> forward, then to a<3>forward, then a<4> forward .
Whereas, the backward sequence would start by computing a<4>_b backward , and then to a<3_b backward, ...to a<1>_b backward.
Note: as computing network activation in backward, this is not backward actually is forward prop. But the forward prop has part of the computation going from left to right and backward prop has part of computation going from right to left in this diagram.
(note: parameter wa,ba, should be different in forward and backward?)



2.3: Make predictions y<t>:
after have computed all the activations in forward and backward, can make predictions, based on both forward a<t> and backward activation a<t>_b:
y<t>= g (Wy [a<t>, a<t>_b] + by)

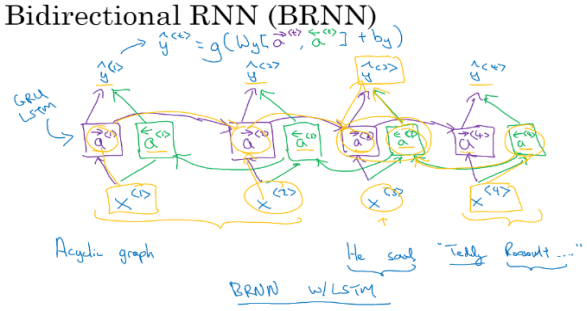
e.g: for time step 3:
Input from current
>1. x<3> ---> a<3>
>2. x<3> ---> a<3>_b
current info. goes into both forward and backward.

Input from forward:
>1. x<1>: ---> a<1> ---> a<2> -----> a<3>
>2. x<2>: ---> a<2> -----> a<3>

Input from backward:
>1. x<4> ---->a<4>_b ----->a<3>_b

output: y<3> = y<t>= g (Wy [a<3>, a<3>_b] + by)

Forward and backward allows the prediction at time three to take as input both information from: 1. the past, 2. as well as information from the present which goes into both the forward and the backward things at this step,3. as well as information from the future.



Summary:

1. BRNN block:
The bidirectional recurrent neural network blocks here (represented by activation functioncan a<t>, a<t>_b), can be standard RNN bloc or GRU blocks or LSTM blocks.
In fact, for a lots of NLP problems, a bidirectional RNN with a LSTM appears to be commonly used.

So, 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.

2. BRNN Advantage:
BRNN is able to make predictions anywhere even in the middle of a sequence by taking into account information potentially from the entire sequence.

3. BRNN disadvantage : do need the entire sequence of data before can make predictions anywhere.
e.g: while building a speech recognition system, BRNN will let you take into account the entire speech utterance. using BRR diretly , need to wait for the person to stop talking to get the entire utterance before you can actually process it and make a speech recognition prediction.
So for a real type speech recognition applications, they're somewhat more complex modules as well rather than just using the standard bidirectional RNN showed here.

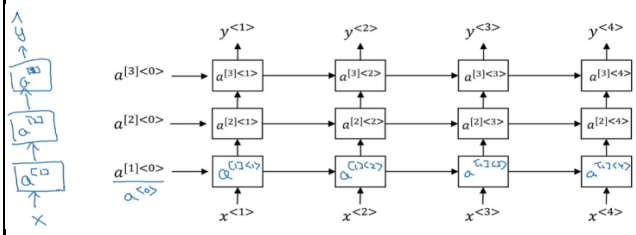
But for a lot of natural language processing applications where you can get the entire sentence all the same time, the standard BRNN algorithm is actually very effective.

5.1-12 Deep RNNs

The different versions of RNNs you've seen so far will already work quite well by themselves. But for learning very complex functions sometimes is useful to stack multiple layers of RNNs together to build even deeper versions of these models.

Deep RNN example:

1. Standard neural network:
>1. Have an input X
>2. then that's stacked to some hidden layer :
Might have activations a<1> for the first hidden layer, and then that's stacked to the next layer with activations a<2>, then maybe another layer, activations a<3>
>3. and then you make a prediction ŷ.



2. Deep RNN :
Deep RNN is a bit like standard RNN, by taking this network and unrolling that in time.

2.1: Unrolling standard RNN in time:

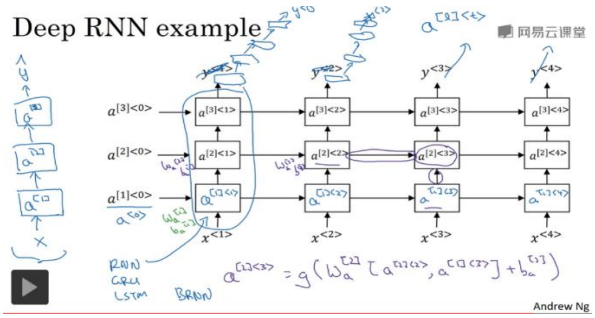
notation: a[l]<t>: a[l] to denote that it's an activation associated with layer l and then <t> to denote that that's associated over time t.

in layer 1:
>time 1: x[1]<1> + a[1]<0> ---->a[1]<1>
> Time 2: x<2> + a<1> ---->a<2>
.....
> Time t: x[1]<t> + a[1]<t-1> ---->a[1]<t>

2.2: Deep RNN:
stack unrolled RNN vertically and so this will be a new network with three hidden layers.

>1. Computation detail:
e.g: a[2]<3> : has two inputs: from the bottom, and from the left. : a[2]<3> = g (Wa[2] . [a[2]<2>, a[1]<3>] + b[2])

Wa[2] and ba[2] are used for every activation computationsin layer 2.
Whereas, first layer has its own parameters Wa[1] and ba[1].



Note:
1. neural networks may have very, very deep, maybe over 100 layers. but for RNNs, having three layers is already quite a lot. Because of the temporal dimension （时间维度- time connection ） , these networks can already get quite big even if you have just a small handful of layers.

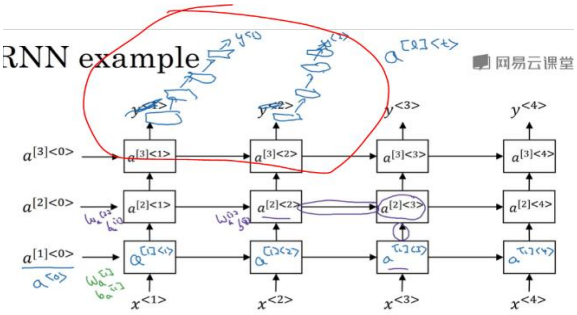
2. There are cases, output prediction y<t> in some layer do not have horizental connection.- red circle
Have recurrent layers stacked on top of each other, but might have a bunch of deep layers that are not connected horizontally.
e.g: right finally predicts in time step 1 y<1>. and haves same deep network that predicts y<2>.

This type of network architecture we see a little bit more:
Have three recurrent units (3 layer) that connected in time, followed by a network after that, as we seen for right- y<1>, y<2>, y<3> and y<4>, there's a deep network, but that does not have the horizontal connections-.red circle

So that's one type of architecture we seem to be seeing more of.

3. And quite often, these blocks (a[l]<t>) don't just have to be standard RNN, the simple RNN model. They can also be GRU blocks LSTM blocks.

4. Can also build deep versions of the bidirectional RNN.
Because deep RNNs are quite computationally expensive to train, there's often a large temporal extent already, even with not as many deep recurrent layers. right example: three deep recurrent layers that are connected in time.



Summary:
Have learned ranging from the basic RNN, the basic recurrent unit, to the GRU, to the LSTM, to the bidirectional RNN, to the deep versions of this , now have a very rich toolbox for constructing very powerful models for learning sequence models.

summary:

1. sequence model: basic RNN (many to many Tx= Ty)
take sequence order in cosider-->Wa, Wy same in each time step.

2. Language model: P(y<1>, ...y<Ty> | X): predict possiblity of next word given previous words.
Architecture: based on RNN, modified only: x<t> = correct y<t>; excpet time 1: x<1>=0, a<0>=0;
Inpute: x<t> = y<t-1> (labeled word in previous time step)
output for each word: softmax function used.

3. Generating noval sentence: used trained language model, sampling
x<t> = sampled y^<t-1>

3. Sequence architecture: many to many (Tx= Ty), many to one, one to many, many -many (Tx \=Ty)

4. Activation block: GRU, LSTM: for gradient descent vanishing problem-sequence long term dependency
c<t> = Gu . *c~<t> + Gf .*C <t-1>

5. BRNN: for caputuring further info. in each direction: ealier and later

6. Deep RNN: Stack unrolled nueral network -RNN together.-more powerful :
each layer parameter Wa[l], b[l], is same.

5-2_NLP and Word Embedding

5.2-1 Word representation _NLP and word embedding

Last week, we learned about RNNs, GRUs, and LSTMs. In this week, you see how many of these ideas can be applied to NLP, to Natural Language Processing, which is one of the features of AI because it's really being revolutionized by deep learning.

One of the key ideas is word embeddings: which is a way of representing words. That let your algorithms automatically understand analogies like that, man is to woman, as king is to queen, and many other examples.

And through these ideas of word embeddings, you'll be able to build NPL applications, even with models the size of relatively small label training sets.

Finally towards the end of the week, you'll see how to debias word embeddings: that's to reduce undesirable gender or ethnicity or other types of bias that learning algorithms can sometimes pick up.

1. Word reprensation:

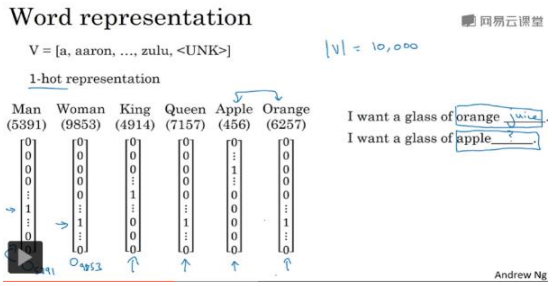
1.1 1-hot representation

We've been representing words using a vocabulary of words, and we've been representing words using a one-hot vector.
> 1-hot vector:
e.g: , if man is word number 5391 in this dictionary, then represent it with a vector with 1 in position 5391, use O _5391 to represent this vector, O here stands for one-hot. all the word will be similarly represented with one-hot vector.

>2 one weaknesses of 1-hot representation:
It treats each word as a thing onto itself, and it doesn't allow an algorithm to easily generalize the cross words:

e.g: have a language model that has learned that when you see sentence' I want a glass of orange__'. the blank word will be very likely ' juice'.
But 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 _", 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,
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.
e.g: If take any two vectors like queen and king and any product of them, the end product is zero. If take apple and orange and any product of them, the end product is also zero.
-->Couldn't get the distance between any pair of these vectors , as all are same 0. So it just doesn't know that somehow apple and orange are much more similar than king and orange or queen and orange.

So, it will be nice if instead of a one-hot presentation we can instead learn a featurized representation with each of these words.



1.2 featurized representation

>1. Feature Intuition:
Featurized representation with each of these words: A man, woman, king, queen, apple, orange or really for every word in the dictionary, we could learn a set of features and values for each of them.

e.g: for each of right words, we want to know:
>>1. Feature-Gender: what is the gender associated with each of these words.
If gender goes from -1 for male to +1 for female, then the gender associated with man might be -1, for woman might be +1. And then eventually, learning these things maybe for king you get -0.95, for queen+ 0.97, and for apple and orange sort of genderless.
>>2. Feature -Royal: how royal are these word.
Words man and woman are not really royal, so they might have feature values close to 0. Whereas king and queen are highly royal. And apple and orange are not really loyal.
>>3. Feature-age:
words man and woman doesn't connotes much about age. Maybe men and woman implies that they're adults, but maybe neither necessarily young nor old. So maybe values close to 0 . Whereas kings and queens are always almost always adults. And apple and orange might be more neutral with respect to age.
>>4. Feature-food:
word man is not a food, woman is not a food, neither are kings and queens, but apples and oranges are foods.

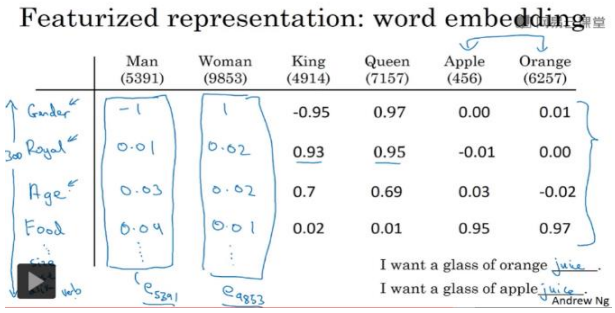
These words can be many other features as well ranging from size, cost, Is this something that is a live, Is this an action, or is this a noun, or is this a verb, or is it something else? And so on. So you can imagine coming up with many features.

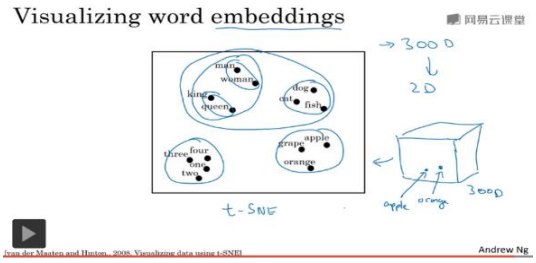
1.2 featurized representation

>2. Featue note:
If take 300 different features, and it allows you to take this list of 300 numbers, that then becomes a 300 dimensional vector for representing the a word .
e.g : take word man. use the notation e_5391 to denote a representation of it's feature vector. while 5391 corresponding to word postion in dictionary.
And similarly, for the other words.

>3. Summarize:
if use this representation-feature vector to represent the words orange and apple, then the representations for orange and apple are now quite similar: Some of the features will differ because of the color of an orange, the color an apple, the taste, but by a large, 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 feature vector representing allows algorithm to generalize better across different words.





3. Feature visualization

Over the next few videos, we'll find a way to learn words embeddings: We just need to learn high dimensional feature vectors that gives a better representation than one-hot vectors for representing different words.

The features we'll end up learning, won't have a easy to interpret interpretation:
will not be 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 like, king and orange or queen and orange.

3.1 Visulization-feature vector:
Once learn a feature vector (e.g: 300 dimensional), one of the popular things to do is also to take this 300 dimensional data and embed it in a 2D dimensional space to visualize them.

3.2 Visulizaiton method: one common algorithm for doing this is the t-SNE algorithm: (due to Laurens van der Maaten and Geoff Hinton).

3.3 Visualization example: In right 2D visulaization of words feature vectors:
find that words like man and woman tend to get grouped together, king and queen tend to get grouped together, and these are the people which tends to get grouped together;
Those are animals who can get grouped together.
Fruits will tend to be close to each other. Numbers like one, two, three, four, will be close to each other.
And then, maybe the animate objects as whole will also tend to be grouped together.

This feature vector visulization gives a sense/intuition that, word embeddings algorithms like this can learn similar features for concepts that feel like they should be more related, as visualized by that concept that seem to you and me like they should be more similar, end up getting mapped to a **more similar feature vectors.**

3.4: word embedding naming:

Embedding: these representations will use these sort of featurized representations in maybe a 300 dimensional space, these are called embeddings.
The reason of calling them embeddings is, you can think of a 300 dimensional space, take every words like orange, and has a 300 dimensional feature vector so that word orange gets embedded to a point in this 300 dimensional space.

Visulization method:
Algorithms like t-SNE, map feature vector to a much lower dimensional space: can actually plot the 2D data and look at it.
that's what the term embedding comes from.

Word embeddings has been one of the most important ideas in NLP, in Natural Language Processing.
In this video, you saw why you might want to learn to use word embeddings. In the next video, let's take a deeper look at how you'll be able to use these algorithms, to build NLP algorithims.

5.2-2 Using word embedding_NLP and word embedding

In the last video, you saw what it might mean to learn a featurized representations of different words. In this video, you see how we can take these representations and plug them into NLP applications.

1. Named entity recognition example:

1.1: Examples

Object-:Trying to detect people's names

> Sentence 1-training set: Given a test sentence like "Sally Johnson is an orange farmer"

Hopefully, trained algorithm figure out that Sally Johnson is a person's name, hence, the outputs is 1.

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. (BRNN needed)

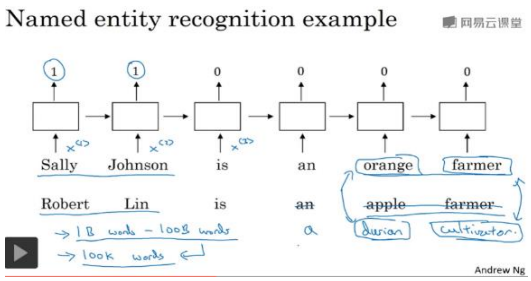
>Sentence 2-test set: New test sentence: "Robert Lin is an apple farmer"

if now use the featurized representations- the embedding vectors, then after having trained a model that uses word embeddings as the inputs, if has above new input, Knowing that orange and apple are very similar will make it easier for learning algorithm to generalize to figure out that Robert Lin is also a human, is also a person's name. (Word embedding needed as input)

>Sentence3-test set: has much less common words in sentence: " Robert Lin is a durian cultivator"

A durian is a rare type of fruit, 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.



1.1 Word embedding able to do transfer learning is :

Word embedding enable algorithm to generalizing words learned in training set to test set, due to word embedding algorithm has a large training set of **unlabled text**, examin very large text corpuses-get large words' feature.

(note: word embedding matrix is large engough, and trained by large unlabeld text-->get large words feature vector -which reflected words' similarity;

-->words/feature vectors used in new algorithm training set, could cover new algorithm test set (feature vector, as relation with training feature vector has been learned in workd embedding matrix/feature vector per se)

if their feature vector is similar, then RNN algorithm could generalizing from knowing one words role to another word of similar feature.

So one of the reasons that word embeddings will be able to do this (generalizing words learned in training set to test set) 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. And farmer and cultivator are similar, and therefore, learn embeddings, that groups them together.

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.

---->

1.1 Word embedding able to do transfer learning is :

--->

And so this allows 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.

And then transfer that knowledge to a task, such as named entity recognition, for which you may have a relatively small labeled training set.

Note: for simplicity, in this example only draw a unidirectional RNN. but for amed entity recognition task, should use BRNN.

(Note:

Word embedding enable trained algorithm (with small training set) to transfer to test words: due to word embedding :

trained word embedding algorithm get word embedding matrix: feature number x vocab size-->each word in vocab get it's learned feature;

for algorithm trained on small training set, once learned training set -words' embedding vector as input, then could genaralize to similar embedding vectors-->correspoinding to test set (should have same distribution as training, dev set-->feature vectors in training set could almost cover feature vector in dev, test set.).

(Note: use smaller training set/feature vectors to represent whole data set (feature vector), as one training example/feature vector could represents ten/more examples(feature vector) in dev/test set. kind of like: pre-processing database , get their relationship by data feature vecotr-->one training set is more representative for the whole database, smaller training set need.)

(large training set needed as to cover almost all features of the whole database, as now database itself already learned relationshipe internally, no need so many training set).

2. Transfer learning and word embedding

Process of carrying out transfer learning using word embeddings:

1. first is to learn word embeddings from a large text corpus (1-100B words): E
   (or download pre-trained word embeddings online)
   There are several word embeddings that you can find online under very permissive licenses.

2. Transfer embedding to new task, where you have a much smaller labeled training sets (like 100k words).
   And use this,like 300 dimensional embedding, to represent your words. One nice thing also about this is you can now use relatively lower dimensional feature vectors. 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

3. Optional: continue to fine tune the word embedding with new data
   as you train your model on your new task with a smaller label data set, one thing you can optionally do is to continue to fine tune / adjust the word embeddings with the new data.
   In practice, do this only if new task has a pretty big data set. If your label data set for new task is quite small, then usually, would not bother to continue to fine tune the word embeddings.

2. Transfer learning and word embedding

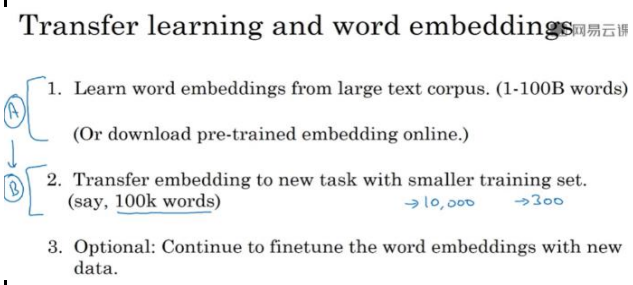
Summarize

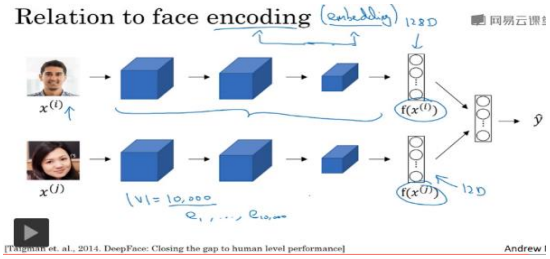
Word embeddings for transfer learning:

1. Tend to make the biggest difference when the task you're trying to carry out has a relatively smaller training set. So it has been useful for many NLP tasks like named entity recognition, text summarization, co-reference resolution,r parsing.
2. Less useful for language modeling, machine translation, especially if you're accessing a language modeling or machine translation task for which you have a lot of data just dedicated to that task.

If trying to transfer from some task A to some task B, the process of transfer learning is just most useful when you happen to have a tons of data for A and a relatively smaller data set for B.

And so this rule is true for a lot of NLP tasks,but just less true for some language modeling and machine translation settings.





3. Word embedding relation to face encoding

3.1: Face recognition:

Train a Siamese network architecture that would learn, say, a 128 dimensional representation for different faces. And then you can compare these encodings in order to figure out if these two pictures are of the same face.

The words encoding and embedding mean fairly similar things.

In the face recognition literature, people also use the term encoding to refer to these vectors, f(x(i)) and f(x(j)).

3.2: Difference of word embeding and face encoding:

One difference between the face recognition literature and what we do in word embeddings is :

> for face recognition, we want to train a neural network that can take as input any face picture, even a picture you've never seen before, and have a neural network compute an encoding for that new picture.

> Whereas for learning word embedding: we'll have a fixed vocabulary of, say, 10,000 words. And we'll learn a vector e1 through, say, e10,000 that just learns a fixed encoding or learns a fixed embedding for each of the words in our vocabulary

So that's one difference between the set of ideas you saw for face recognition versus what the algorithms we'll discuss in the next few videos.

3. Word embedding relation to face encoding

Note:;

the terms encoding and embedding are used somewhat interchangeably. So the difference of face encoding and word embedding is not represented by the difference in terminologies.

It's just a difference in how we need to use these algorithms in face recognition, where there's unlimited sea of pictures you could see in the future.

Versus natural language processing, where there might be just a **fixed vocabulary**, and everything else like that we'll just declare as an unknown word.

Summary:

This item show how using word embeddings allows to implement this type of transfer learning. And how, by replacing the one-hot vectors we're using previously with the embedding vectors, you can allow your algorithms to generalize much better, or you can learn from much less label data.

Next, I want to show you just a few more properties of these word embeddings. And then after that, we will talk about algorithms for actually learning these word embeddings. Let's go on to the next video, where you'll see how word embeddings can help with reasoning about analogies.

5.2.3 Properties of word embedding

One of the most fascinating properties of word embeddings is that they can also help with analogy reasoning. And while reasonable analogies may not be by itself the most important NLP application, they might also help convey a sense of what these word embeddings are doing, what these word embeddings can do.

1. Analogies

e.g: man-->woman, king-->?
Hope algorithm could learn analigies automatically.
First calcualte vector: e_man-e_woman, then find a word could get similar result vector by calcuating with e_king: e_man - e_woman ~~ e_king - e_?

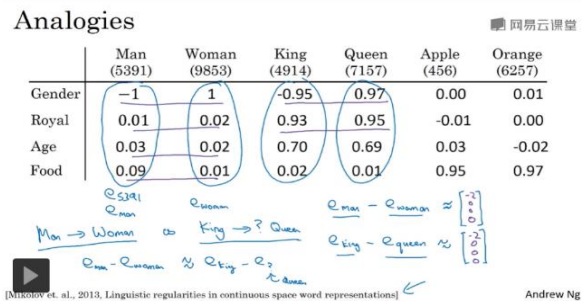
1.1 E.g: question, man is to woman as king is to_ what?
--> to have an algorithm figure this 'queen' out automatically?

vocabulary feature: only four feature: Gender, Royal, Age, Food.

Process to find word 'queen':
>1. using this four dimensional vector-feature vector to represent man, denote as e5391, and also woman, king, queen.

>2. One interesting property of these vectors is :
e_man - e_woman ~= [-2; 0; 0; 0]T
e_king - e_queen ~= [-2; 0; 0; 0]T

This is capturing is that the main difference between man and woman is the gender. And the main difference between king and queen, as represented by these vectors, is also the gender. Which is why the difference e man- e woman, and the difference e king- e queen, are about the same.



2 Analogies using word vectors

in Feature space, vector: e_man-e_wan is the vector differance, very similar to the vector differance; e_king-e_queen.-->vector difference actually is a gender difference

Analogy method： word vector:
finding a new word w, to maximize the similarity : **max sim (e_w, e_king - e_man + e_woman)**
have some propriate similarity function for measuring how similiar is some word w to this quantity of the right.

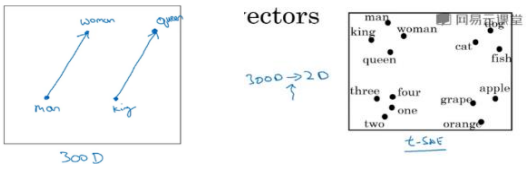
Note:
1. this method -using word vectors works , by maximizing the similarity, can actually get the exact right answer.
2. Accuracy in papers like 30%-75% , count an anology attempt as correct only if it guessed the exact word right.

2.3 feature vector similarity in visulized space

Algorithms like t-SAE could visualize words from high dimesion feature space to lower dimension space.
SAE : it takes original feature vector (like 300-D data), and maps it in a very **non-linear way to a 2D space.**

The mapping that t-SAE learns is a very complicated and very non-linear mapping. So after the t-SAE mapping, should not expect these types of parallelogram -words' feature vector differeance relationships, like the one we saw on the left, to hold true.

And it's really in this original 300 dimensional space that you can more reliably count on these types of parallelogram relationships in analogy pairs to hold true. And it may hold true after a mapping through t-SAE, but in most cases, because of t-SAE's non-linear mapping, you should not count on that. And many of the parallelogram analogy relationships (平行四边形类似关系)will be broken by t-SAE.



4. similarity function that is most commonly used

The most commonly used similarity function is called cosine similarity:.
In cosine similarity, define the similarity between two vector as basically the inner product between u and v.

cosine similarity: cosine of the angle between two vectors: Sim (u, v)=uT•v/(||u||_2 x ||v||_2)
if u, v are very similar, inner product uT•v tend to be large: angle =0-->consie=1, angle=90, consie=0, angle =180, cosine=-1

This is called cosine similarity because this is actually the cosine of the angle between the two vectors, u and v.

Note:
1. Cosine similarity works quite well for these analogy reasoning tasks.

2. Can also use square distance or Euclidian distance, -|| u-v|| ^2.
Technically, this would be a measure of dissimilarity rather than a measure of similarity. So we need to take the negative of this, and this will work okay as well. Yet cosine similarity being used a bit more often.

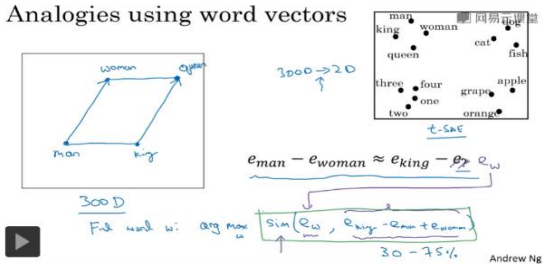
The main difference between these similarity functions is how it normalizes the lengths of the vectors u and v.

1. Analogies

1.2: One way to carry out this analogy reasoning

If the algorithm is asked, man is to woman as king is to what?
Algorithm can compute vector difference of e man- e woman, and try to find a feature vector-a word , so it's difference with king feature vector is close to e_man - e_woman.
And it turns out that when queen is the word plugged in here, then the left hand side is close to the the right hand side.

So these ideas were first pointed out by Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. And it's been one of the most remarkable and surprisingly influential results about word embeddings.



Formalize above procedure in example , and turn it into an algorithm. In pictures.
2.1 Analogy method: word vector

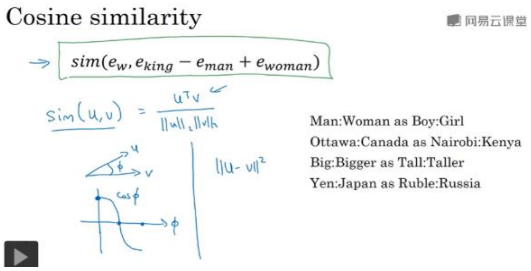
for above example: analogy of man to woman, find word to map king.
The word embeddings live in maybe a 300 dimensional space-feature space. And so the word man, woman, king, queen are represented as different point in the space.
the vector difference between man and woman is very similar to the vector difference between king and queen.
Note: the arrow drew in right pic. is really the vector that represents a difference in feature 'gender'.

2.2: Analogy using word vectors

So in order to carry out this kind of analogical reasoning to figure out, man is to woman is king is to what, need to find the word w,so this equation holds true: e_man - e_woman ~= e_king - e_w
>1. Algorightm object is:
find the word w that maximizes the similarity between,e_w compared to e_king and e_man -e_woman:
max sim (e_w, eking-e_man+ewoman)

Note:
1. have some appropriate similarity function for measuring embedding vector similarity.
finding the word that maximizes the similarity should hopefully let you pick out the word queen.

2. The remarkable thing is, this actually works.
>1. If you learn a set of word embeddings and find a word w that maximizes this type of similarity, you can actually get the exact right answer.
>2. At research papers, it's not uncommon for research papers to report 30% -75% accuracy on analogy using tasks like these. Where count an anology attempt as correct only if it guesses the exact word right. So only if, in this case, it picks out the word queen.



Summary:
One of the remarkable results about word embeddings is the generality of analogy relationships they can learn: e.g , it can learn that man is to woman as boy is to girl, because the vector difference between man and woman, similar to king and queen and boy and girl, is primarily just the gender.
And all of these things can be learned just by running a word embedding learning algorithm on the large text corpus. It can spot all of these patterns by itself, just by running from very large bodies of text.

So in this video, you saw how word embeddings can be used for analogy reasoning. And while you might not be trying to build an analogy reasoning system yourself as an application, this I hope conveys some intuition about the types of feature-like representations that these representations can learn. And you also saw how cosine similarity can be a way to measure the similarity between two different word embeddings. Now, we talked a lot about properties of these embeddings and how you can use them. Next, let's talk about how you'd actually learn these word embeddings, let's go on to the next video
(note: word embedding matrix: -->transfer learning (: new algorithm trianed with less training data;) + analogy reasoning (similarity function of words embedding vecotrs)

5.2-4 embedding matrix

Formalize the problem of learning a good word embedding. When you implement an algorithm to learn a word embedding, what you end up learning is an embedding matrix

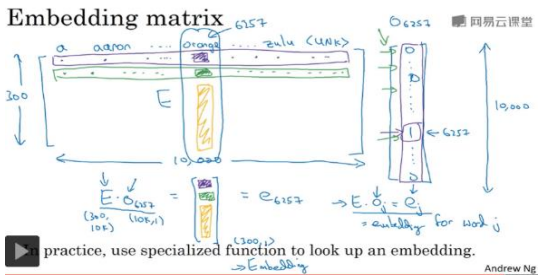
1. Embedding matrix

e.g: using 10,000-word vocabulary, each word have 300 feature.
Object: s learn embedding matrix E, 300 dimensional x 10,000 dimensional matrix,

1.1 Embedding matrix :E (feature number x vocabulary size)
Columns: different embeddings for the each word in vocabulary.
Rows: feature for words

1.2 One-hot vector: O:
vector element is 0/1, 1 correspding to the position in vocabulary, 0 for other positions.
e.g : Orange was word number 6257 in ocabulary of 10,000 words. So use notation O_6257 as its'one-hot vector, with zeros everywhere and a one in position 6257. And so, this will be a 10,000-dimensional vector with a one in just one position

1.3 for word j embedding vector : e_j , have equation: e_j = E . O_j
So, the embedding matrix E times one-hot vector here winds up selecting out this 300-dimensional column corresponding to the word j.



2. Learn an embedding matrix E
Idea: initialize E randomly and use gradient descent to learn all the parameters of this feature number x vocabulary size (300 x 10,000)dimensional matrix .
and E x this one-hot vector gives the embedding vector.

Note: equation e_j = E . O_j
Writing the equation, as it is convenient to write this type of notation.
But when implementing this, it is not efficient to actually implement this as a mass matrix vector multiplication, because the one-hot vectors, now this is a relatively high dimensional vector and most of these elements are zero.
In practice use a specialized function to just look up a column of the Matrix E rather than do this with the matrix multiplication.

Summary:
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. In the next video, we'll start to talk about specific algorithms for learning this matrix E.

5.2-5 Learning word embedding

In this item start to learn some concrete algorithms for learning word embeddings.
In the history of deep learning as applied to learning word embeddings, people actually started off with relatively complex algorithms. And then over time, researchers discovered they can use simpler and simpler and simpler algorithms and still get very good results especially for a large dataset. But what happened is, some of the algorithms that are most popular today, they are so simple that if I present them first, it might seem almost a little bit magical, how can something this simple work? So, here to start off with some of the slightly more complex algorithmsas it's actually easier to develop intuition about why they should work, and then we'll move on to simplify these algorithms and show you some of the simple algorithms that also give very good results.

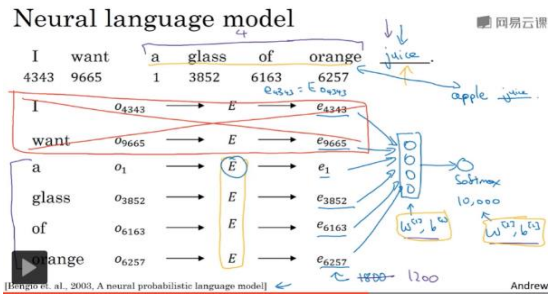
1. Neural language model

e.g using neural netowrk to build a language model, given previous few words, to predict the next word.
Input/training set: "I want a glass of orange_"
expected output: : during training, want neural network to predict the next word in the sequence.

Note: each word of the input already have the index in the vocabulary .

Building a **neural language model** is a reasonable way to learn a set of embeddings
(note: why not sequence language model? input word short, neural network more fast?
neural language model is just one time step of sequence language mode:
in time t, input x<t> =correct y<t>, a<t-1>-previous words activation:
a<t> = g_a(Wa*[a<t-1>, x<t>] + b_a)
y^<t> = softmax (Wy* a<t> + b_y)
-->loss function: max output probability of right labeld y<t>

while transfer this step to neural network:
a<t>= g_a (W1*[e1, e2...en] +b1)
y^<t> = softmax (w2* a<1> + b2)
)

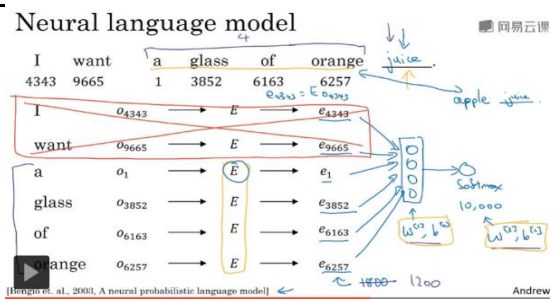


1.1 Procedure : build a neural network to predict the next word in the sequence:
>1. Construct one-hot vector corresponding to each input word I. O_j
one-hot vector is vocabulary size dimensional vectorr.

>2. Get embedding vector for each input word: e_j: O_j--------->E ---------->e_j
e_j = E. O_j

>3. Feed all of these embedding vectors into a neural network-have its own parameter: W1, b1 :a<t> = W1 . e<t> + b1
each word embedding vectors will be stacked together and feed to softmax,
Input to neural network, dimension is : word embedding-feature number x input word number
e.g: 6 words, and feature number is 300, then neural network input dimension is 300 x 6-->unrolled (1800,1)

>4. feeds to a softmax, which has it's own parameters as well: W2, b2: y<t> = g (W2 . a<t> + b2)
soft max classfies 1000 possible outputs in the vocabulary for the final word trying to predict.
Output vocabulary size vector: possiblity of each word in vocabulary being the next word in input sentence.
-->so if in the training slide we saw the word juice then, the target for the softmax in training would be that it should predict the same word juice .



(note: could use RNN: many to one architecture??:
e<1> --> a<1>: g1(wa(e<1>, a<0>) + ba<2>) --> + e<2> --->a<2> .....--->a<t> ---> y^ = g2(wy* a<t> + by)
parameter: only wa, ba, wy, by, less than above;-no: wa: dimension is a_activation dim x (a+n_feature dim) ,
while above w1; f_word number x n_feature dim, less parameters
but this still have input dimension change problem?-->**W1 dimension need to fix input size**)

1.2 Note:
> 1. Historical window size
word embeddings feed to neural network: More commonly have a fixed historical window-fixed window size, is hyperameter of algorithm.
e.g: might decide always want to predict the next word given the previous four words (note: four here is a hyperparameter of the algorithm.)
So this is how you adjust to either very long or very short sentences where decide to always just look at the previous four words, so neural network will input a 1,200 dimensional feature vector, go into this hidden neural layer, then have a softmax and try to predict the output.

Using a fixed history, means can deal with even arbitrarily long sentences because the input sizes are always fixed.

>2. parameters in this language model:
>>>1. Embedding matrix E: use same matrix E for all the words in embedding vector computation: e_j = E. O_j
don't have different matrices for different positions in the proceedings four words, is the same matrix E.

>>>2. Weight W1, b1; W2, b2

These parameters of the algorithm can use backprop to perform gradient descent to maximize the likelihood of your training set : e.g repeatedly predict given four words in a sequence, what is the next word in vocabulary.

1.3 Summarize:
This algorithm will learn pretty decent word embeddings.
Reason: if algorithm remember orange juice, apple juice example, it is in the algorithm's incentive to learn pretty similar word embeddings for orange and apple, because doing so allows it to fit the training set better:
as it's going to see orange juice sometimes, or see apple juice sometimes, and so the algorithm will find that it fits the training set fast if apples, oranges, and grapes, and pears, and so on and maybe also durians , with similar feature vectors.

So, this is one of the earlier and pretty successful algorithms for learning word embeddings- this matrix E.

(note: this is one time step of language model: is stack this model here in time step -->get the language model:
language model: predict sentence probability: p(y<1>, ..y<Ty>) = p(y<1>) * p(y<2>|y<1>)*p(y<3>|y<1>, y<2>).....
except p(y<1>): generated by x<1> =0, a<1>=0, others time step input is y<t-1>, a<t-1>;
here in word embedding training, a<t-1> is all previous word e1, ..e<t-1>, not the activation of previous words)

2. Other context/target pairs

Ggeneralize above algorithm and derive even simpler algorithms.

e.g: input sentence in training set : " I want a glass of orange juice to go along with my cereal".
Object of the algorithm was to predict some target word 'juice'. , given some context -like the last four words.

2.1 Contex to use for training algorithm

Researchers have experimented with many different types of context and get this conclusion:
>1. If the goal is to build a language model:
Then it is natural for the context to be a few words right before the target word.

$$P(w_i) = \frac{f(w_i)^{3/4}}{\sum_{j=1}^{10,000} f(w_j)^{3/4}}$$

>2. If the goal is not to learn the language model per se,
Tthen can choose other contexts:
>>>1. 4 words on left and right
Means we're posing a learning problem where the algorithm is given four words on the left, and four words on the right, to go along with, and this has to predict the word in the middle.
And posing a learning problem like this where you have the embeddings of the left four words and the right four words feed into a neural network, similar to try to predict the word in the middle, try to put it target word in the middle.

This can also be used to learn word embeddings.

3. Summary:
in this video show 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.
In the next video, you'll see how using even simpler context and even simpler learning algorithms to map from context to target word, can also allow you to learn a good word embedding.

2. Other context/target pairs

2.1 Contex to use for training algorithm

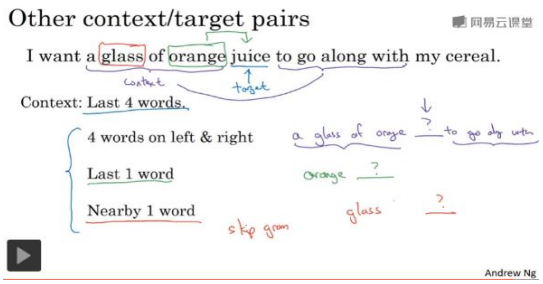
>>>2. Last 1 word:
This will be different learning problem: given one word, and predict the next word.
Can construct a neural network that just feed the embedding of the one previous word to a neural network as you try to predict the next word.

>>>3. Nearby one word-works suprisingly well
e.g: take the word 'glass', and then there's another words somewhere close to glass, what do you think that word is?

This is the idea of a Skip-Gram model, a simpler algorithm where the context is just one word rather than four words, but this works remarkably well.

Summary:
If you really want to build a language model, it's natural to use the last few words as a context.
But if your main goal is really to learn a word embedding, then can use all of these other contexts and they will result in very meaningful work embeddings as well.

(note: word embedding transfer learning is less usefull for language model , which has engough large datat to train algorithm, -->learn word embedding by simpler method would be better??)



5.2-6 Word2Vec_NLP and word embeddings

In the last video, showed how by learning a neural language model to get good word embeddings.
In this item, will see the Word2Vec algorithm which is simple and comfortably more efficient way to learn this types of embeddings.

1. Skip-grams

E.g:
Given one sentence of training set : "I want a glass of orange juice to go algong with my cereal"
Object: In the skip-gram model, come up with a few context to target pairs to create our supervised learning problem.

1.1: Idea:
>Most of the ideas here are due to Tomas Mikolov, Kai Chen, Greg Corrado, and Jeff Dean.

1.2: Set a supervised learnig probelm

Process:
>1. Context word: Rather than having the context be always the last four words or the last end words immediately before the target word, instead randomly pick a word to be the context word.
e.g chose the word orange as context word

>2. Target word: Randomly pick another word within some window of choosen context word as target word.
e.g: within window +/- 5 words or +/-10 words of the context word , randomly choose a word to be target word.
So maybe just by chance you might pick juice to be a target word, Or might choose glass so have another pair , or maybe choose the word 'my' as the target.

And so we'll set up a supervised learning problem where given the context word, you're asked to predict what is a randomly chosen word within like, +/- 10 or +/- 5 word window, of that input context word.

Note:
This is not a very easy learning problem, because within +/- 10 words of the word orange, it could be a lot of different words.
But a goal that's setting up this supervised learning problem, isn't to do well on the supervised learning problem per se, it is that we want to use this learning problem to learn good word embeddings.

2. Model

e.g: vocab has 10,000 words. S
In vocabulary, orange is word O_6257, and the word juice is the word O_4834

Object: the basic supervised learning problem we're going to solve is that we want to learn the mapping from some Context c, such as the word orange to some target, which might be the word juice or the word glass or the word my.
(note: labeled y: only label one word in the specify window of context word as right(1), other word in vocab as 0.
but in training set, there are many other words in the specify window of context word, if algorithm learned to output possibility for those word close to 0, should be a problem??_

input x: context word
labeldd output y: target word t

Softmax: $p(t|c) = \frac{e^{\theta_t^T e_c}}{\sum_{j=1}^{10,000} e^{\theta_j^T e_c}}$

Object: learn to map x to that open y.

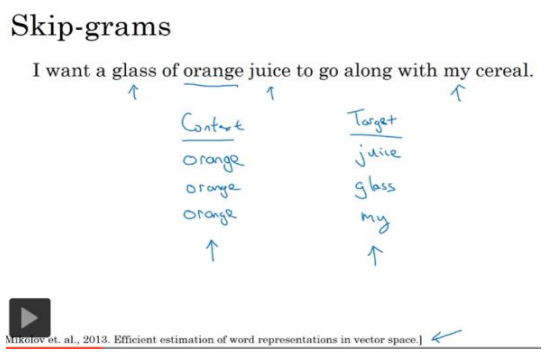
2.1 Model process

>1. Construct inpute context word: get embedding vector for the input context word
input word-->one-hot vector O_c --->embedding matrix E ------->embedding vector e_c = E. O_c

>2. Feed contxt word pembedding vector to neural network.
(note: similar to use neural language model as above, only directly gointo softmax,not middle activation)

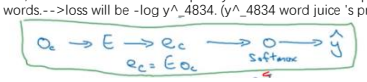
2.2 Mode-detail. softmax model:

Probability of different target words given the input context word :
p (t|c) = exp [(θ_t)^T * e_c]/ sum (exp [(θ_j)T * e_c]) over all word in vocab.



2.3 . loss function for softmax $\mathcal{L}(\hat{y}, y) = -\sum_{i=1}^{10,000} y_i \log \hat{y}_i$

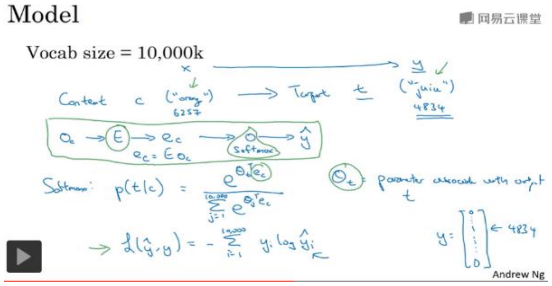
target word:
>labeled y: represneted by one-hot vector
>predicted y^, vector of vocab size, probability of being the target for each word in vocab.
l (y^, y) = - sum y_i log (y^_i) over all word in vocab.
e.g:
if the target word is juice O_4834, then its one-hot vector 4834-th element equal to 1 and the rest is 0.
And similarly y^ t will be a 10,000 dimensional vector output by the softmax unit with probabilities for all 10,000 possible targets words.-->loss will be -log y^_4834. (y^_4834 word juice 's probability for being the target word)

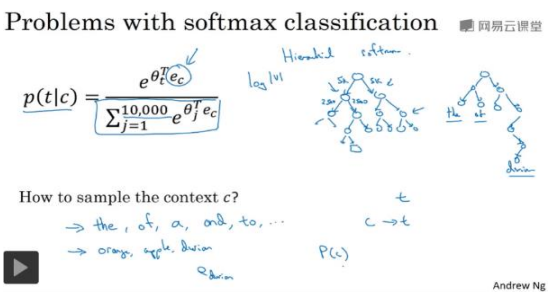


2.4 Summarize:
Above is the overall little model,- neural network with basically looking up the embedding and then just a soft max unit.
The matrix E will have a lot of parameters corresponding to all of these embedding vectors,e_c.
and the softmax unit also has parameters θ_i (i=1.....10000-vocab. size)

by optimizing this loss function with respect to the all of these parameters,will actually get a pretty good set of embedding vectors.

This is called the skip-gram model: because it 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.





3. Problems with softmax classification $p(t|c) = \frac{e^{\theta_t^T e_c}}{\sum_{j=1}^{10,000} e^{\theta_j^T e_c}}$

There are a couple problems with using this algorithm, and the primary problem is computational speed

3.1 Softmax classification computaion cost:
Every time/iteration to evaluate this probability, need to carry out a sum over all vocab words : sum ((θ_j)T * e_c) over all word j in vocab. (θ_j different in every iteration)

when vocab size is very big like 100,000 or a 1,000,000, it gets really slow to sum up over this denominator every single time/iteration/step. In fact 10,000 is actually already quite slow, and it makes even harder to scale to larger vocabularies.

3.2 hierarchical softmax classifier
One solutions to this is to use a hierarchical softmax classifier.

hierarchical softmax classifier: instead of trying to categorize something into all vocabulary words like 10,000 categories on one go but:
>1.Have first binary classifier, tells is the target word in the first half vocab. or in the second half of vocab.
If this binary cost tells target word in the first half vocab like first 5,000 words, then use a second binary classfier.
>2. use a second binary classfier, to tell if the target word in the first half or second half of 'first half vocab

continue do this: using binary classfier to localize target word in a smaller new 'vocab size', until eventually you get down to classify exactly what word it is.

3. Probles with softmax classification

Note: Hierarchicalsoft max:
1. Having a tree of classfier like right , means that each of the interior nodes of the tree can be just a binding classifier-like logistic classfier.
So don't need to sum over all vocab size in order to make a single classification.

2. Computational classifying tree like this, scales like log of the vocab size log |v|rather than linear in vocab size.
That is why it is called a hierarchical softmax classifier.

3.In practice, the hierarchical softmax classifier doesn't use a perfectly balanced tree or this perfectly symmetric tree, with equal numbers of words on the left and right sides of each branch.
In practice, the hierarchical softmax classifier can be developed : common words tend to be on top, whereas the less common words like durian can be buried much deeper in the tree:
Because we see the more common words more often, and so in this way need only a few traversals to get to common words like 'the' and 'of'. Whereas less seen frequent words like durian is okay to be buried deep in the tree as see it less often-do not go that deep frequently.

-->So there are various heuristics for building the tree how you used to build the hierarchical softmax classfier..

In the next video, will talk about a different method- nectar sampling, which I think is even simpler. And also works really well for speeding up the softmax classifier and the problem of needing the sum over the entire vocab size in the denominator

4. Sampllling the context C

Once sampled the context C, the target t can be sampled within,some word window of the context C.

Choosing context C: method:
>1. Sample uniformly, at random, from training corpus
Not recommended , as training site will be dominated by these extremely frequently words, because then you spend almost all the effort updating e_c, for those frequently occurring words.
e.g: there are some words like 'the, of, a, and, to' and so on that appear extremely frequently, if sampling uniformly, at random, then context to target mapping pairs just get these these types of words extremely frequently. whereas there are other words like orange, apple, and also durian that don't appear that often.

But we need to make sure to spend some time updating the embedding, even for these less common words like e_durian.

>2. heuristics to balance common and less common word
In practice the distribution of words p(c) isn't taken just entirely uniformly at random for the training set corpus, 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.

HOW TO IMPLEMENT IN CODING?
How to generate training set?

5. Summary:

Above is for the Word2Vec skip-gram model. The original paper actually had two versions of this Word2Vec model, the skip gram was one, and the other one is called the 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 entire vocabulary size into the denominator of the soft max.

In the next video I show you an algorithm that modifies the training objective that makes it run much more efficiently therefore lets you apply this in a much bigger fitting set as well and therefore learn much better word embeddings.

5.2-7 Negative sampling_NLP and word embedding

Last video showed how the Skip-Gram model allows to construct a supervised learning task: mapping from context to target and this allows you to learn a useful word embedding.
But the downside of that was the Softmax objective was slow to compute. (should be a common problem for algorithm using softmax with large output dimensioin)

In this video, will show a modified learning problem - negative sampling: do something similar to the Skip-Gram model y, but with a much more efficient learning algorithm.

1. Defining a new learning problem

Most of the ideas presented in this video are due to Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeff Dean.

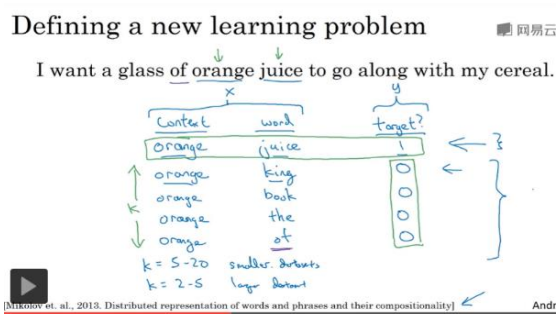
1.1 negative sampling algorithm object: create a new supervised learning problem: given a pair of words like orange and juice, to predict if this is a context-target pair.

1.2: example:
e.g: training sentence: ' I want a glass of orange juice to go along with my cereal"
Input: orange- juice was a positive example, target (output) y =1
Input: orange - king is a negative example, target (output) y =0

1.3:Generating training set:

1.3.1 Process:
>1. generated a positive input/ pair: sample a context and a target word.
e.g: orange - juice is associate that with a label of 1
Positive example generation method: generated exactly as skip model: in a training sentence, Sample a context word, look around a window of say, plus-minus ten words and pick a target word.

>2. generate k negative examples:
Take the same context word **and then just pick a word at random from the dictionary**.
Under the assumption that if we pick a random word from vocab, it probably won't be associated with the word from training sentence: so
e.g: chose the word king at random and label pair orange-king as 0. and then take orange and pick another random word from the dictionary, and label this pair as 0.



1.3.2 Note:
for all pairs, whose target word sampling from vocab instead of training sentence that contex word in, labled as 0, even if the target word appears next to contex word in training sentence as well.
e.g: sampled targe word 'of' from vocab, label orange-of as 0 aslo even if 'of' next to orange in training sentence.

1.3.3 Summarize:
for generating traing data-postive and negative pairs:
Pick a context word and then pick a target word in training sentence -the first row of this table in right pic., this gives us a positive example. So context- target, and then give that a label of 1.

For some number of times say, k times: take the same context word and then pick random words from the dictionary, king, book, the, of, whatever comes out at random from the dictionary and label all those 0, and those will be our negative examples.
and it's okay if just by chance, one of those words we picked at random from the dictionary happens to appear in the window, in a plus-minus ten word window say, next to the context word, orange. (also label as 0)

(note: could use logistic model??, architectur too simple to learn word embeddding?)
trianin set (c, t), labeled 0/1?-->input is a pair (c,t)-->sigmoid output 0/1

2. Mode:
Supervised learning model for learning a mapping from x to y.

$$p(t|c) = \frac{e^{\theta_t^T e_c}}{\sum_{j=1}^{10,000} e^{\theta_j^T e_c}}$$

Inpute/Training set: one example: postive and negative pair, + target labeled 1 or 0
inpute context word embedding e_c, labeled output y= [0,0,..1...0,0..]-vocab size/K+1 size.
output: predict for each pair, target is 0 or 1. (vocab size vector: for each word is it postive target 1 or negatvie target 0)
Denotion:
C: context word,
t: possible target word, t
target: denote 0, 1, this is for a context target pair.

$$P(y=1|c,t) = \sigma(\theta_t^T e_c)$$

2.1 Mode:
Define a logistic regression model: chance of y = 1, given the input c, t pair, : p (y=1| c, t) = sigmoid (θtT . e_c)
y^ = sigmoid(w* e_c +b) :
Input size: e_c : feature f x 1
output size: y^: vocab size v x 1
weight: -->w: dim: v x f=[w1; w2; ...w_v]
w_t : parameter of each vocal word->θt

2.2 Model parameter: θt , e_c
Vector θt for each possible target word t.
Embedding vector e_c for each possible context word.

2.3 Model illustration:
For every positive examples, have k negative examples with which to train this logistic regression-like model.
>1. Generate context word embedding vector
context word C as input--->one-hot vector O_c--------->E---------->contex word embedding vector: e_c

2. Feed context word embedding vector to softmax: y= sigmoid (θtT . e_c)

Output y^: vecotor (vocab. size x 1): predicting for each word t in vocab, the chance of p (y=1 |c, t)
-->predict for vocab size pairts (c, t): t take each word from vocab, get a pair (c, t)-c is same, and predict their target 0/1.
Now it is vocab size like 10,000 logistic regression classification problems: each classfier (θt)corresponding to predict if the target word is this specific word t like juce in vocab.
So think of this as having vocab size (like 10,000) binary logistic regression classifiers.

2.4 . parameter updating:

But instead of training all 10,000 of them on every iteration, we're only going to train k+1 (1 postive and k negative=4 here) of them.
Train the one responding to the actual target word and then train k number randomly chosen negative examples.

Model: input context word --->one-hot O_c --->E--->embedding vector e_c----->sigmoid on vocab : vocab size classfier: -->get vocab size output vector .

for one word t, in vocab:
>1. forward prop:
Z_t= θt T . e_c
y^_t= a_t =sigmoid (θt T . e_c)

word t prediction loss: L_t = L(y^_t, y_t) = y_t * log (y^_t) + (1-y_t) * log (1-y^_t)

>2. backprop:
dZ_t= - y_t* (1-y^_t) + y^_t * (1- y_t)= y^_t - y_t
dθt = d Z_t . e_c
d e_c = d Z_t . θt

>3. update parameter:
θt+= - a *dθt
e_c+= - a *e_c

as only k+1 labeled y_t- if it is not the word of negative and positive pairs, no label y,-->no loss L could compute--> then could only do backprop for these labeld k+1 words.

3.Selecting the negative examples
So after having chosen the context word orange, how do you sample these words to generate the negative examples

>1. Sample 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 will end up with a very high representation of words like ' the', 'of', and, and so on.
>2. The other extreme: use 1/(vocab size), sample the negative examples uniformly a
but this is also very non-representative of the distribution of English words

$$P(w_i) = \frac{f(w_i)^{3/4}}{\sum_{j=1}^{10,000} f(w_j)^{3/4}}$$

>3. Authors, Mikolov et al, reported that empirically, this heuristic value work best:
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.

Sampled proportional to their frequency of a word to the power of3/4:
e.g: f(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 3/4, 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.

And so I'm not sure this is very theoretically justified, but multiple researchers are now using this heuristic, and it seems to work decently well.

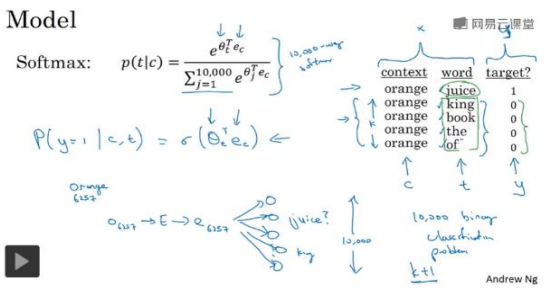
1.4: Create a supervised learning problem
>1. learning problem:
>>> training set:
Inputs x: these pair of words, positive and negative pairs: input context word embedding e_c
labeled output-vocab size vector: only labeled 1 for positive target position, 0 for negative target word position. other position do not care.: [0,0,...1, 0, 0,]: positon in vocab.
or just make output k+1 dimension vecotr: 1 for positive target, 0 for k negative target word .

>>> algorithm Object: predict the target label: pedict the output y.: predict 1 for postive target word, 0 for negative target word.

>2. Idea & object:
So the problem is really, given a pair of words like orange and juice:
do you think they appear together/ Do you think these two words is gotten by sampling close to each other , Or do you think one word is gotten from the training text and one word chosen at random from the dictionary?

algorithm object is really to try to distinguish, for a given pair of words, from which of the two distributions they come from: distribution one-label 1 distribution: cloese to each other; distribution two-label 0 distribution: not close at all-one from training sentence, one from vocab.

So this is how you generate the training set.
Note: choose k:-number of negative pairs for the same context word: **Large values of k for smaller data sets, why?-->make training set larger: as Context word C number is small, increase K-->increase training pairs.**
Mikolov et al, recommend :
for smaller data set: maybe k is 5 to 20,
If have a very large data set: then chose k to be smaller. k is like 2 - 5 for larger data sets.
In this example, just use k = 4.



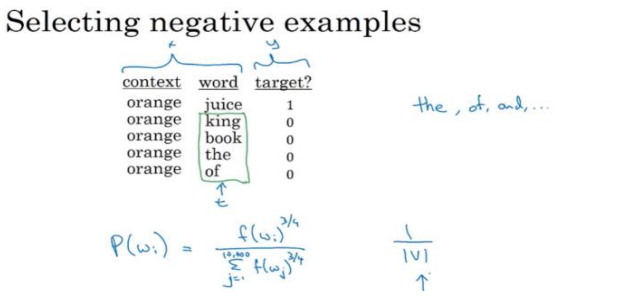
2.5 Summarize:

So instead of having one giant 10,000 way Softmax, which is very expensive to compute, we've instead turned it into vocab size 10,000 binary classification problems, each of which is quite cheap to compute.
And 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.

(word2v-skim: -->one iteration will update all vocab word parameter θ_i & word embedding e_c)

And this is why the computation cost of this algorithm is much lower because you're updating k + 1 units, k + 1 binary classification problems. Which is relatively cheap to do on every iteration rather than updating a 10,000 way Softmax classifier.

This technique is called negative sampling:
Because 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,
And on every iteration, choose four different random negative words with which to train your algorithm on.
(one training example every iteration??)



4. Summarize

Learn word vectors in a Softmax classier is very computationally expensive, by changing that to a bunch of binary classification problems, you can very efficiently learn a pretty good words vectors.

And as is the case in other areas of deep learning as well, there are open source implementations. And there are also pre-trained word vectors that others have trained and released online under permissive licenses. And so if you want to get going quickly on a NLP problem, it'd be reasonable to download someone else's word vectors and use that as a starting point.

So that's it for the Skip-Gram model (negative sampling and word 2vect). In the next video, I want to share with you yet another version of a word embedding learning algorithm that is maybe even simpler than what you've seen so far.

(note:
word2 vect: contex word -->target word:softmax (vocab size)
negative sammpling: context word-->target positive and negetaive word-sigmoid (vocabsize, yet only train k+1 embeddings)-why need combine together 1+k negative? could not use context->target: sigmoid?-update only one word embedding every iteration-too slow so choose k negative? -->or could choose k' positve + k negative, for faster traiing??)

5.2-8 Glove word vectors_NLP and word embedding

You learn about several algorithms for computing words embeddings. Another algorithm that has some momentum in the NLP community is the GloVe algorithm.
This is not used as much as the Word2Vec or the skip-gram models, but it has some enthusiasts, may because in part of its simplicity

1. GloVe: (global vectors for word representation)

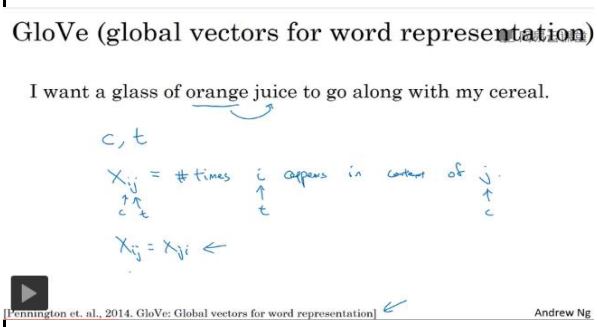
The GloVe algorithm was created by Jeffrey Pennington, Richard Socher, and Chris Manning.

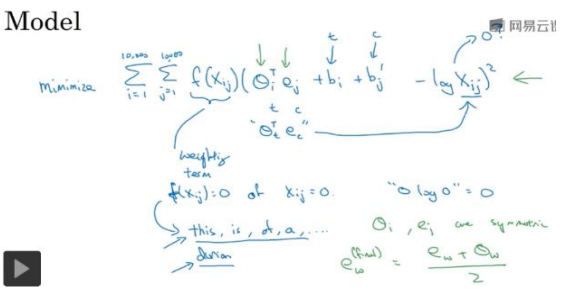
1.1 Glove Context and target words:
Previously, we were sampling pairs of words, context and target words, by picking two words that appear in close proximity to each other in text corpus.

GloVe algorithm: starts off just by making that explicit:
X_{ij} : is the number of times that a word i appears in the context of j.
word i play the role of word t, and j play the role of word c. so you can think of X_{ij} as being X_{tc}.
for each word i, j in vocab size ,Go through your training corpus and just count up how many times does a word i appear in the context of a different word j.
(note: training set size: vocab. size x vocab size
if i,j symmetric-->training set size = vocab size x vocab size /2

1. GloVe: (global vectors for word representation)

1.2 Symmetric property:
Depending on the definition of context and target words, you might have that $X_{ij} = X_{ji}$ (symmetric).
e.g, if define context and target as: whether or not they appear within +/-10 words of each other, then it would be a symmetric relationship. yet if choice of context is always the word immediately before the target word, then X_{ij} and X_{ji} may not be symmetric .
But for the purposes of the GloVe algorithm, we can define context and target as whether or not the two words appear in close proximity, say within+/- 10 words of each other.
Idea: So, X_{ij} is a count that captures how often do words i and j appear with each other, or close to each other.





2. Model:

2.1 Labeled training set:
Inpute: pair word j-i; (note: training set size: vocab. size x vocab size; if i,j symmetric-->training set size = vocab size x vocab size /2)
labeled output: x_ij: How related are words i and j as measured by how often they occur with each other in training data.

2.2 modle output:
>1. (θ_ i)T. e_j : tell how related are those two words.
Think of i and j as playing the role of t and c.---> (θ_ t)T. e_c , (θ_ i)T. e_j , is to tell you how related are those two words.

2.3 GloVe model object/ loss function:
Minimize the difference between θ_ iT. e_j and log of X_{ij}

$$\text{Minimize} \sum_{i=1}^{10,000}\sum_{j=1}^{10,000} f(X_{ij})(\theta_i^T e_j + b_i + b_j' - \log X_{ij})^2$$

cost for a singal example: word j: l (x(j)): sum (θ_ i)T. e_j - log x_ij) over alll i in vocab;
cost for all training example : word j go through vocab size: sum [l (x(j))] over all word j in vocab

2.4 **Model parameter:**
>1. θ_i, e_j (i,j=1, ...vocab size)

Using gradient descent to minimize the sum difference of ((θ_ i)T. e_j - log x_ij) over i from 1 to vacob size, sum over j from 1 to vacob size,
to learn vectors, so that their end product (θ_ i)T. e_j is a good predictor for how often the two words occur together, which represented here by x_ij.
(note: paramter θ_i, e_j play same role if X_{ij} is symettric)

>2. weight parameter f(x_ij):

if X_{ij} =0, then log (0) 0 is negative infinity, we won't sum over the difference of ((θ_ i)T. e_j - log x_ij) when X_{ij} =0, so add an extra weighting term f(x_ij).
(why not penalt prediction for no related words?)

wight f(x_ij) function:

>>>1. make minimize objec, sum only over the pairs of words that have co-occurred at least once in that context-target relationship.
f(x_ij) = 0, if x_ij = 0. use a convention that 0 * log (0) = 0, so this means if x_ij =0, just don't bother to sum over that x_ij pair.
So then this log (0) term is not relevant, means sum only over the pairs of words that have co-occurred at least once in that context-target relationship.

>>>2. make minimize objec : balance common word and less common word:
there are some words they just appear very often in the English language like, this, is, of, a, and so on. Sometimes used to call them stop words; but there is really a continuum between frequent and infrequent words (介于频繁词和不常用单词之间) ; And then there are also some infrequent words like durion, which actually still want to take into account, but not as frequently as the more common words.

So weighting factor f(x_ij) can be a function: that gives a meaningful amount of computation, even to the less frequent words like durion, and gives more weight but not an unduly large amount of weight to words like, this, is, of, a, which just appear a lot in language.
There are various heuristics for choosing weighting function f(x_ij), that neigher gives these words (like of, the) too much weight nor gives the infrequent words too little weight.

You can take a look at the GloVe paper, referenced in the previous slide, if you want the details of how F can be chosen to be a heuristic to accomplish this.

(should be a function of frequency??)

2.5 Note:
parameter θ_i, e_j (i,j=1, ...vocab size), their role are completely symmetric.

θ_i, e_j are symmetric in that, at the math, they play pretty much the same role and you could reverse them or sort them around, and they actually end up with the same optimization objective.
One way to train the algorithm is: initialize θ_i,and e_ j both uniformly at random, run gradient descent to minimize its objective, and then when you're done for every word, to then take the average.
e.g: For a given words w, can have :
e_w_final = e_w +θ_w / 2
e_w : word w embedding that was trained through this gradient descent procedure $e_w^{(final)} = \frac{e_w + \theta_w}{2}$
θ_w : θ_w trained through this gradient descent procedure
because θ and e in this particular formulation play symmetric roles unlike the earlier models we saw in the previous videos, where θ and e actually play different roles and couldn't just be averaged like that.
(note: skip-gram: negative sample * word2v: de_c, dθ_w not same-->not play symettric role)

That's it for the GloVe algorithm. one confusing part of this algorithm is, it seems almost too simple. How could it be that just minimizing a square cost function like this allows you to learn meaningful word embeddings? But it turns out that this works. And the way that the inventors end up with this algorithm was, they were building on the history of much more complicated algorithms like the newer language model, and then later, there came the Word2Vec skip-gram model, and then this came later. And we really hope to simplify all of the earlier algorithms.

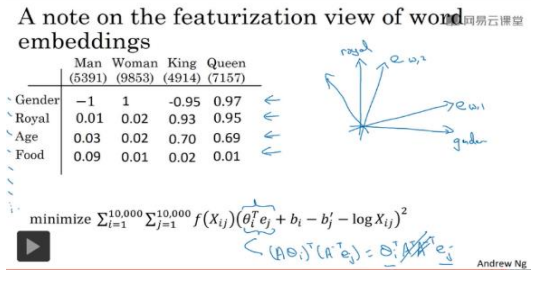
3. A note on the featurization view of word embeddings

We started off with right talbe featurization view as the motivation for learning word vectors:
"maybe the first component of the embedding vector to represent gender, the second component to represent how royal it is, then the age and then whether it's a food, and so on."

for the learned word embedding using one of the algorithms that we've seen like the GloVe algorithm, what happens is, you cannot guarantee that the individual components of the embeddings are interpretable.

3.1 Reason:
e.g right: there is some feature space with the first axis is gender and the second axis is royal. What you can do is guarantee that the first axis of the embedding vector is aligned with this axis of meaning, of gender, royal, age and food.
And in particular, the learning algorithm might choose right e_w1 to be axis of the first dimension.
So given maybe a context of words, the first dimension might be right e_w1 and the second dimension might be e_w2. (it might not even be orthogonal-正交), maybe it'll be a second non-orthogonal axis, could be the second component of the word embeddings you actually learn.

$$(\theta_i^T e_j + b_i - b_j' - \log X_{ij})^2$$



3.2 Proof: Glove minimize object: term: (θ_i)T. e_j:

given invertible matrix A, then θ_i, e_j could be easily be replaced : $(A.\theta\_i)T.(A^{-1}.\ e\_j) = (\theta\_i)T.\ e\_j$:

This is a brief proof that shows, with an algorithm like this, you can't guarantee that the axis used to represent the features will be well-aligned with what might be easily humanly interpretable axis. In particular, the first feature might be a combination of gender, and royal, and age, and food, and cost, and size, and all the other features. It's very difficult to look at individual components, individual rows of the embedding matrix and assign the human interpretation to that.

3.3 Note:

But despite this type of linear transformation (A.θ_i, A-T. e_j) , the parallelogram map that we worked out when describing analogies, that still works.

So, despite this potentially arbitrary linear transformation of the features, you end up learning the parallelogram map for figure analogies still works.

5.2-9 Sentiment classfication_NLP and Word Embeddings

1. Sentiment classificaiton problem:

1.1 Sentiment classificaiton defination:

Sentiment classification is the task: looking at a piece of text and telling if someone likes or dislikes the thing they're talking about. It is one of the most important building blocks in NLP and is used in many applications.

1.2. Chanllenge:

One of the challenges of sentiment classification is might not have a huge label training set for it.

But with word embeddings, y able to build good sentiment classifiers even with only modest-size label training sets.

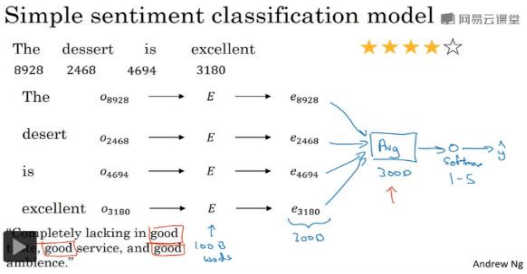
(inpute to neural network for rating is inpute sentence words embedding-->with trained embedding matrix, get to generalize neural network for rating, from mapping small training sentence words embedding to rating, to mapping huge, other dev/test sentence word embedding to rating. as words embedding relation between small training sentence and huge dev/test sentence have learned well by word embedding matrix E.

1.3: Example:

Input X: is a piece of text

Output Y: predict what is the sentiment, such as the star rating of, like a restaurant review.

e.g: someone says, "The dessert is excellent" and they give it a four-star review, "Service was quite slow" two-star review, etc.



1.4 Summarize

If train a algorithm to map from X to Y based on a label data set like this, then could use this algorithm to monitor comments that people are saying about maybe a restaurant, it can look just a piece of text and figure out how positive or negative is the sentiment toward restaurant.

So one of the challenges of sentiment classification is you might not have a huge label data set. So for sentimental classification task, training sets with maybe anywhere from 10,000 to maybe 100,000 words would not be uncommon. Sometimes even smaller than 10,000 words. And word embeddings can help you to much better understand especially when you have a small training set.

2. Simple model_sentiment classification

2.1 Example

Input sentence: . "dessert is excellent"

output: labeled : 4 star

object: build a classifier to map it to the output Y

Look up those training sentence words in dictionary, and build a classifier to map it to the output Y that this was four stars.

2.2: Process

>1. Generate embedding vectors for all words of input sentence

>>>1. Take input sentence words and look up the one-hot vector.

>>>2. a one-hot vector multiplied by the embedding matrix E, , get embedding vectors for each words

Embedding matrix E can learn from a much larger text corpus, like a billion words or a hundred billion words, and use that to extract out the embedding vector for the word "the "dessert" is" excellent",

If embedding matrix E was trained on a very large data set, 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.

>2. Build a classfier_simple one: sum/average embedding vectors and feed into softmax classfier

one simple way to build a classifier: take input sentence embedding vectors, then just sum or average them.

e.g: given a 300-dimensional feature vector , and pass it to a soft-max classifier which then outputs Y-hat: output the probabilities of the five possible outcomes from one-star up to five-star.

2.2: Process

>2. Build a classfier_simple one: sum/average embedding vectors

Note: simple mode-sum/average embedding vector

1.advantage: by using the average operation, this particular algorithm works for reviews/sentence that are short or long .

Because even if a review/sentence that is 100 words long, can just sum or average all the feature vectors for all hundred words and gives a 300-dimensional feature representation, that can then pass into sentiment classifier.

So this average will work decently well. averages or sum the meanings of all the words in your example sentence.

2. Disadvantage: ignores word order

So one of the problems with this algorithm is it ignores word order.

e.g: sentence "Completely lacking in good taste, good service, and good ambiance".

As the word good appears a lot, the algorithm like this will 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 vector and then classifier will probably think this is a good review while it is a one-star review.

3. RNN for sentiment classificaiton_many to one architecture

Inpute sentence: "Completely lacking in good taste, good service, and good ambiance"

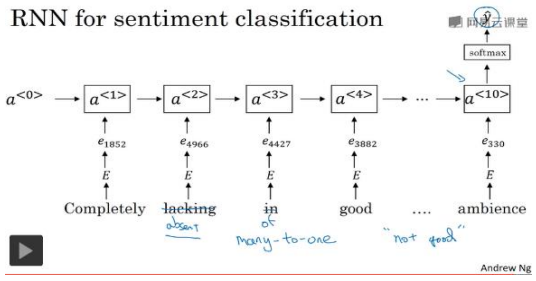
3.1 Process:

>1. Generate one-hot vector for each word of input sentence O_i

>2. Generate embedding vector for each word of input sentence: e_i = E. O_i

>3. Feed all embedding vectors of input sentence words, to RNN.

RNN job is to compute the representation of input sentence at the last time step (e.g: right a<10>) that allows you to predict y^.



3.2 RNN modle Property/advantage:

>1. This RNN is a many-to-one RNN architectur.

>2. And with an algorithm like this, it will be much better at taking word sequence into account :

e.g: realize "things are lacking in good taste" is a negative review unlike the above sum/aver algorithm, which just sums everything together into a big-word vector mush and doesn't realize that "lack good" has a very different meaning than the words "good" and so on.

So train this RNN algorithm, will end up with a pretty decent sentiment classification algorithm.

3.3 . Note: word embeding matrix E

As word embeddings can be trained from a much larger data set, this will do a better job generalizing to maybe even new words now that do not see in training set,.

e.g: sentence "Completely absent of good taste, good service, and good ambiance" , then even if the word "absent" is not in label training set, if it was in your 1 billion or 100 billion word corpus used to train the word embeddings, it might still get this right and generalize much better even to words that: were in the training set used to train the word embeddings but not in the label training set for specifically the sentiment classification problem.

So that's it for sentiment classification, and I hope this gives you a sense of how once you've learned or downloaded from online a word embedding, this allows you to quite quickly build pretty effective NLP systems.

5.2-10: Debiasing word embedding _NLP and word embedding

Machine learning and AI algorithms are increasingly trusted to help make, extremely important decisions. And so we like to make sure that as much as possible that they're free of undesirable forms of bias, such as gender bias, ethnicity bias and so on.

This video will show some of the ideas for diminishing or eliminating these forms of bias in word embeddings.

1. The problem of bias in word embedding

1.1 Bias defination here: gender, ethnicity, sexual orientation bias.
That's a different sense of bias then is typically used in the technical discussion on machine learning.

1.2 Bias problem in word embedding:

gender stereotype problem:

E.G1: Word embeddings can learn analogies like man is to woman as king is to queen. But if ask it, man is to computer programmer as woman is to what: a horrifying result where a learned word embedding might output: Man to Computer Programmer as Woman to Homemaker.
It'd be much more preferable to have algorithm output man is to computer programmer as a woman is to computer programmer.

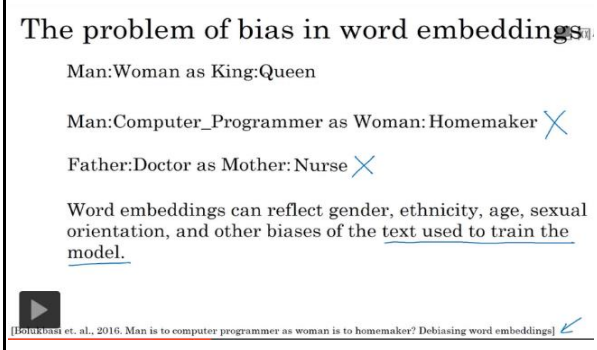
e.g 2: Father to Doctor as Mother is to what? And the really unfortunate result is that some learned word embeddings would output Mother:Nurse

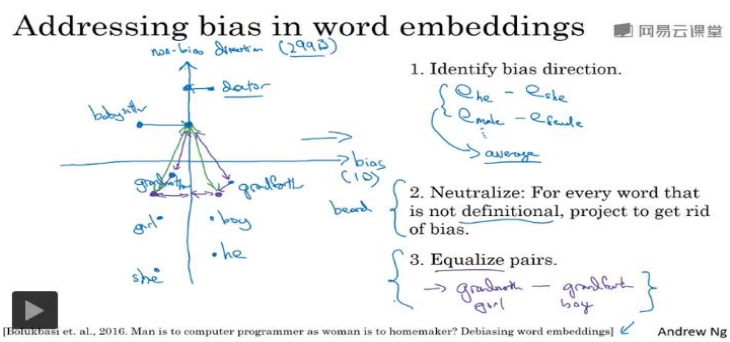
This enforces a very unhealthy gender stereotype.
Word embeddings can reflect the gender, ethnicity, age, sexual orientation, and other biases of the text used to train the model/word embedding matrix.

1.3: Bias influence on real life
As machine learning algorithms are being used to make very important decisions. They're influencing everything ranging from college admissions, to the way people find jobs, to loan applications, whether your application for a loan gets approved, to in the criminal justice system, even sentencing guidelines. Learning algorithms are making very important decisions and so it's important that we try to change learning algorithms to diminish as much as is possible, or, ideally, eliminate these types of undesirable biases

In the case of word embeddings, it can pick up the biases of the text which used to train the model and so the biases they pick up will tend to reflect the biases in text as is written by people.

In this video willl share one example of a set of ideas due to the paper referenced at the bottom by Bolukbasi and others on reducing the bias in word embeddings.





2. Addressing bias in word embedding

2.1:Process

>0: Assuming already learned a word embedding: like right pic. : draw position in feature space of word: babysitter, doctor, , grandmother , grandfather, gir, boy , she, he.

>1. first : identify bias direction:
Identify the direction corresponding to a particular bias we want to reduce or eliminate.
e.g: focus on gender bias (note: these ideas are applicable to all of the other types of bias that mentioned above).

>>>1. Identifiy bias direction method:
e.g: for the case of gender, take the embedding vector for defintional words and substract several pairs of them:
e_he - e_she
e male-e female,
and take a few of these differences and basically average them. And this will allow you to figure out in this case the gender direction/ the bias direction- e.g right pic, horizental direction.

Non-bias directon:
is the direction is unrelated to the particular bias we're trying to address.
e.g: in this case, think of the gender bias direction as a 1D subspace whereas a non-bias direction, this will be 299-dimensional subspace.

>>>2. Note:
here has simplified the description a little bit in the original paper. The bias direction can be higher than 1-dimensional, and rather than take an average, it's actually found using a more complicated algorithm called a SVU, a singular value decomposition (奇异值分解)- principle component analysis, it uses ideas similar to the pc (主成分分析(pc)) or the principle component analysis algorithm.

>3. Equalize pairs

Equalization for pairs of words that is definitional.
e.g: such as grandmother and grandfather, or girl and boy, where you want the only difference in their embedding to be the gender.
>>>1. Motivation for equalizing definitional words:
eg. right pic.: the distance / similarity, between babysitter and grandmother is actually smaller than the distance between babysitter and grandfather. And so this maybe reinforces an unhealthy, or maybe undesirable, bias that grandmothers end up babysitting more than grandfathers.

So in equalization step, will make sure that words of defination , like grandmother and grandfather, are both exactly the same similarity/ the same distance, from words that should be gender neutral-projected on non-bias dirction already, such as babysitter or such as doctor.

>>>2. method:
There are a few linear algebra steps for that. But what it will basically do is move grandmother and grandfather to a pair of points (new positions) that are equidistant-(等轴距) from non-bias axis (like right vertical axile). And so the effect of that is that now the distance between babysitter, compared to these two words, will be exactly the same.

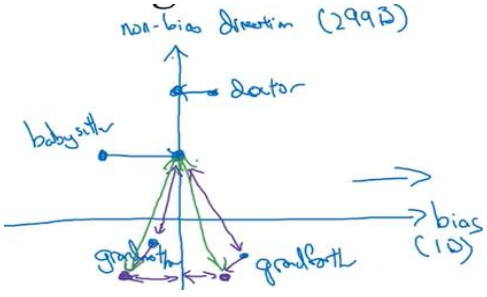
>2. second: neutralization(中和) : for every word that's not definitional (（定义不明确是否是例如女性，男性）), project it to get rid of bias.

>>>1. word of definitional: there are some words that intrinsically capture gender ((定义本身就说明了性别):), eg: grandmother, grandfather, girl, boy, she, he, a gender is intrinsic in the definition.

>>>2. word not definitional: e.g: word like doctor and babysitter, etc. that we want to be gender neutral.
in the more general case, u might want words like doctor or babysitter to be ethnicity neutral or sexual orientation neutral, and so on, but we'll just use gender as the illustrating example here.

But so for every word that is not definitional, this basically means not words like grandmother and grandfather, which really have a very legitimate gender component, because, by definition, grandmothers are female, and grandfathers are male.

>>>3. Projection:
So for words like doctor and babysitter, just project them onto non-bias directon (right pic. vertical axile) to reduce/ eliminate their component in the bias direction- e.g reduce their component in this horizontal direction.
怎么处理： non definitional word embedding, project on non-bias vector?



>>>3. Decide what word to neutralize:

In general, there are many pairs of definitional words like this grandmother-grandfather, boy-girl, sorority-fraternity, girlhood-boyhood, sister-brother, niece-nephew, daughter-son, that you might want to carry out through this equalization step.
e.g: the word doctor seems like a word you should neutralize to make it non-gender-specific or non-ethnicity-specific. Whereas the words grandmother and grandmother should not be made non-gender-specific.

The paper authors did is train a classifier to try to figure out what words are definitional: should be gender-specific and what words should not be. and it turns out that most words in the English language are not definitional-means gender is not part of the definition. and it's such a relatively small subset of words like this, grandmother-grandfather, girl-boy, sorority-fraternity, and so on that should not be neutralized.

So a linear classifier can tell what words to pass through the neutralization step to project out this bias direction, to project it on to this essentially 299-dimensional subspace.
And the number of pairs want to equalize is actually also relatively small (at least for the gender example), it is quite feasible to hand-pick most of the pairs you want to equalize.

The full algorithm is a bit more complicated than present it here.

3. Summarize:
Reducing or eliminating bias of our learning algorithms is a very important problem because these algorithms are being asked to help with or to make more and more important decisions in society. In this video I shared just one set of ideas for how to go about trying to address this problem, but this is still a very much an ongoing area of active research by many researchers.

(note: after eliminatin bias: neutralize every non-defination words and equalize every defination words, the parallel relationship still be kept (linear transformation or not (non-linear tranformation)??)

Word embedding learning algorithms

Word embedding learning algorithm:

1. Neural language model: predict the next word given previous word

$$p(t|c) = \frac{e^{\theta_t^T e_c}}{\sum_{j=1}^{10,000} e^{\theta_j^T e_c}}$$

> Training set:
input: previous specify window of input word embeddings
labeled output: one-hot vector of next word;

>Architecture: neural language model: input word embeddings feed to hidden layer 1-->softmax: (vocab size vector)
hidden layer 1: Z1= W1* [e1, e2, e3...] +b1; a1 =g (z1)
softmax: y^= a2 = softmax (w2* a1 +b2)

>Parameter:
word embeddign matrix E, weight w1, w2, bias b1,b2.

$$\mathcal{L}(\hat{y}, y) = -\sum_{i=1}^{10,000} y_i \log \hat{y}_i$$

note: architecture actually is one time step of sequence language model.

Sequence language model : many to one RNN architecture:
Training set: sentence example + labeled y (O_t-one hot vector)=[y<1>, ...y<t>, ..y<Ty>]
Inpute: a<t-1>, x<t> = y<t-1>
output labeled: target word [y<1>, ...y<t>, ..y<Ty>], y<t> is one-hot vector
model prediciton y^: softmax function for each word in vocab: probability of each word in vocab being the target word: p(t| y<1>, y<2>..y<t-1>).
loss: for one time step: -y<t>*log(y^<t>):-->maxmize y^<t>_i: i: the ith postion in vocab, labeled as 1

3. Skip-gram mdoel_ negative sampling: nerual network: just 'softmax'-sigmoid function for each word in vocab

>Training set: one example: K+1 pair: 1 positve context-target (from training sentence), k negative context-target (from vocab)
inpute: context word embedding e_c
output labeld: vecotor like one-hot: but only labeled k+1 element: 1 for positve target word, k 0 for negative target word.
model prediction output: sigmoid function for each word in vocab-->vocab size binary classfier

> Architecute: vocab size bianry classfier: for each word in vocab: y^_t = sigmoid (θT_t * e_c +b_t).
> Cost: only compute k+1 element loss of predicted output

note: while in backprop, only use K+1 word (labeled 1/0 in output y) in each iteration, not go through all vocab word. --> in the iteration : only update k+1 words θT_t & word embedding e_c

save computation/itration.

Word embedding learning algorithm:

2. skim gram model_ V2V:- neural softmax function
one to one RNN archituecture: neural network-simplily just softmax function

>>> Training set: pair of word: context + target
inpute: context word embedding: e_c
output labeled: target word one-hot vector
output-softmax- vocab size category: probability of each word in vocab being the target word:

>>> Architecture: just softmax function: y^_t = exp (θT_t * e_c +b_t)/ sum exp (θT_t * e_c +b_t) over t-all words in vocab.

>>>Parameter:
word embeddings E, each vocab word t weight θT_t
Loss function:-softmax loss: for one training sentence: sum y(k) * log (y^(k)) over k-all vocab words.

4. Glove: neural network :linear fucntion: θT*e_c
> training set: pair word i, j+ relation x_ij (i occure times in some window of j ,in all training sentecne), i,j word in vocab .
inpute: context word embedding e_c
output labeld: x_ij
model output prediction: θT*e_c

>Architecture: line regression: θT*e_c

> cost function: minimize [θT_i*e_j - log (x_ij)] sum over i (one trainig example cost, word j), sum over j (all training example)

-->One iteration: sum over vocab size twice-->more computation cost than softmax function??

(note: paramter θ_i , e_i play same role if X_ij is symettric)

Word embeddign motivation: transfer learing: for new taks have small labele training set.

As word embeddings can be trained from a much larger data set, this will do a better job generalizing to maybe even new words now that do not see in training set,.
as in the huge word corpus used to train the word embeddings, it might generalize much better even to words that: were in the training set used to train the word embeddings but not in the label training set for specifically like 'the sentiment classification' problem.

5-3_Sequence to Sequence Models

5.3-1 Basic models_Sequence to sequence models

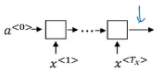
In this week, you'll hear about sequence to sequence models, which are useful for everything from machine translation to speech recognition. Let's start with the basic models, and then later this week, you'll hear about beam search, the attention model, and we will wrap up the discussion of models for audio data like speech.

1. Sequence to sequence model:

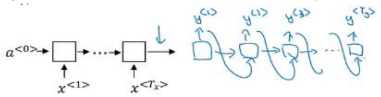
1.1 Example:
Training set:
Inpute: "a French sentence like Jane visite l'Afrique Septembre":
use x<1>, ...x<5> as representations of input sequence

labeled output: translate it to the English sentence, "Jane is visiting Africa in September".
use y<1>, ...y<6> as representations of output sequence

1.2 Training a neural network
Note: ideas here are mainly from these two papers in right pic.



Process:
>1. first: have encoder network:-outputs a vector that represents the input sentence.
>>>1. First have a encoder network be built as a RNN (could be GRU / LSTM);
>>>2. feeding the input French words one word at a time.
>>>3. After ingesting the input sequence the RNN then **outputs a vector C that represents the input sentence.**
(when input sentecne is very long, hard for this feature vector C to remember all input features,-->hard to provdie engough infor. for decoder to translater outp



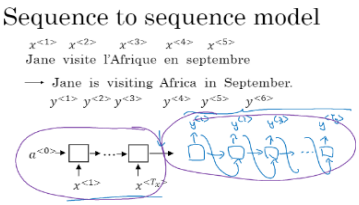
>2. Build a decoded network:
Takes the encoding output as input , and then can be trained to output the translation one word at a time. y<1>, ...y<6>.
Eventually, it output the end of sequence taken, upon which the decoder stops.

Note: as usual, could take the generated output y<t> and feed them to the next step in sequence like what we were doing before when synthesizing text using sampling language model.

1.3 Summarize:
One of the most remarkable recent results in deep learning is that this model works:
Given enough pairs of French and English sentences, if you train a model to input a French sentence and output the corresponding English translation, this will actually work decently well.

This model simply uses an encoding network whose job it is to find an encoding of the input French sentence, and then use a decoding network to then generate the corresponding English translation.

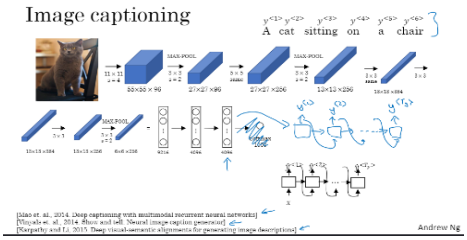
(note: input x<t> is one-hot vecotor? as machine translation have engough labeled training set, no need word embedding?)



[Sutskever et al., 2014. Sequence to sequence learning with neural networks]
[Cho et al., 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation]

2. Image captioning
An architecture very similar to this-machine translation also works for image captioning.

2.1: Example:-right pic.
Inpute: Given an image
output: output captions automatically like " a cat sitting on a chair"



2.2 Training algorithm
Algorithm object: train network to input an image and output caption(like above phrase)

Process:
>1. Encode network: modified on AlexNet, output feature vector representing input image
>>>1. Use AlexNet neura network:
Earlier course on the convnet have shown how you can input an image into a convolutional network, maybe a pre-trained AlexNet, and have that learn an encoding or learn a set of features of the input image. This is actually the AlexNet architecture.

>>>2. Get rid of AlexNet final softmax unit, the pre-trainined AlexNet can give a multi (like a 4,096) dimensional feature vector of which to represent this picture of a cat.

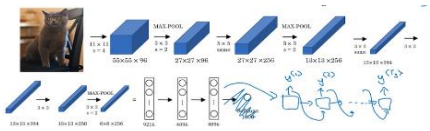
This pre-trained network can be the encoded network for the input image and now have a multi (like 4,096)-dimensional vector that represents the image.

(note: or the Siamese network: used for face verification??)

2.2 Training algorithm
Algorithm object: train network to input an image and output caption(like above phrase)

Process:
>2. Decoder network: take feature vector of input image as input, and then output caption
>>>1. Feed feature vector of input image to an RNN:
>>>2. RNN will generate the caption one word at a time.
Similar to what we saw with machine translation, translating from French the English, you can now input a feature vector describing the inputs and then have it generate an output set of words, one word at a time.

This actually works pretty well for image captioning, especially if the caption you want to generate is not too long.

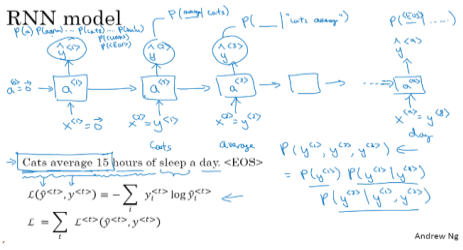


3. Summarize
Now have seen how a basic sequence to sequence model works. How basic image to sequence- image capturing , model works. But there are some differences between how you'll run a model like this, generating a sequence compared to how you were synthesizing novel text using a language model. One of the key differences is you don't want a randomly choosen translation: may be want the most likely translation or don't want to randomly choose in caption, might want the best caption and most likely caption. Let's see in the next video how you go about generating that.

5.3-2: Picking the most likely sentence_ seq 2 seq models.

There are some similarities between the sequence to sequence machine translation model and the language models that you have worked within the first week of this course, but there are some significant differences as well.

1. Machine translation as building a conditional language model
Can think of machine translation as building a conditional language model.



1.1: Language modeling

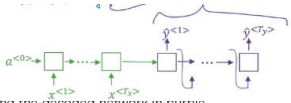
Training set: one sentence y = y<1>, ...y<Ty>
Inpute:
activation a<0> = 0 vector
inpute in each time step: x<t> = labeled y<t-1>
(sampled language model: x<t>=sampled y^<t-1> ,

output in each time step-labeled: y<t> :one-hot vector
output in each time step -predicted:soft max function:
y^< t | y<1>, ...y<t-1> >: given previous labeled/ sampled words, the probabiltiy of the next word is t (in vocab).

Language modeling function:,(right pic. was the network we had built in the first week), :
>1. Allows to estimate the probability of a sentence P(y<1>, ...y<Ty>). = p(y<1>) * p(y<2> |y<1>) ....p(Ty|y<1>, ...y<Ty-1)
>2. Can also use to generate novel /new sentences:
in this example let input x<2> and x<3>, = labeled y<1>, y<2>, etc. just to clean up for the slide,

1. Machine translation as building a conditional language model
Can think of machine translation as building a conditional language model.

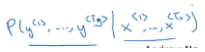
1.2: Machine translation



Model structrue: looks as right.
>1. have encode network
Right pic. encoded network in green and the decoded network in purple.

>2. Decoded network (in purple)
The decoded network looks pretty much identical to the (sampled) **language model.**

Function:
Modeling the probability of, like the output English translation, conditions on some input French sentence: P(y<1>, ...y<Ty> | x<1>, x<2>, ...x<Tx>).



1.3 Compare machine translation vs language model:

> 1. Differance: starting activation

Machine translation model is very similar to the language model, except for **starting activation:**
Instead of always starting along with the vector of all zeros a<0>, it has an encoded network as input.

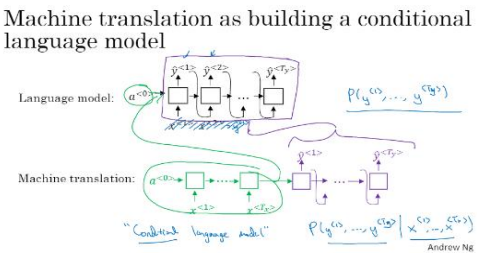
Machine translation encoded network figures out some representation for the input sentence, and it takes that input sentence and starts off the decoded network with representation of the input sentence rather than with the representation of all zeros.

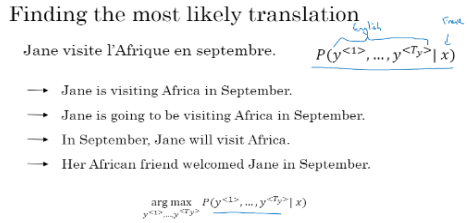
-->that is the reason call machine translation a conditional language model: instead of modeling the probability of any sentence, it is now modeling the probability of, like the output English translation, conditions on some input French sentence.
e.g: trying to estimate the probability of an English translation: what's the chance that the translation is "Jane is visiting Africa in September," but conditions on the input French censors like, "Jane visite l'Afrique en septembre."

>2. Differance: 'decoder' inpute

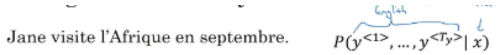
Language model: x<t> = labeled y<t-1>: inpute is previous word
yet sampling language model inpute is the sampled previous generated word y^<t-1>.

machine tranlation: decoder inpute in the word generated in previous step: y^<t-1> -same as sampled language modle





2. Find the most likely translation



2.1: Example
Apply machine translation model to actually translate a sentence from French into English.

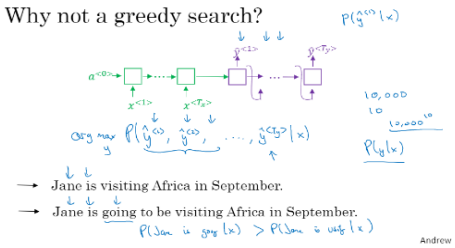
Inpute: right; French sentence: "Jane visite l'Afrique en septembre."
Model Output: the probability of difference English translations for input sentence

2.2: Finding best output sentence
>1. Attention: Do not want to sample outputs at random.
If sample words from this distribution, p (y^<t> |x):p (y^ |x) = p(y^<1>|x) * p(y^<2>|x, y^<1>) * p(y^<3> | x, y^<1>, y^<2>)...
e.g: maybe one time get a pretty good translation, "Jane is visiting Africa in September." But, maybe another time get a different translation, "Jane is going to be visiting Africa in September. " Which sounds a little awkward but is not a terrible translation, just not the best one. And sometimes, just by chance,get others: "In September, Jane will visit Africa."

>2. Object: of while using model p(y^|x) for machine translation: find output sentence y^ that maximize probability p(y^|x):
So when using this model p(y^|x) for machine translation, do not sample at random from this distribution, but instead try to find the English sentence y^ that maximizes that conditional probability p(y^|x).

>3. Method of find best y^: come up an algorithm to finding best output sentence y^
So in developing a machine translation system, one of the things you need to do is come up with an algorithm that can actually find the value of y^ that maximizes probabilty: p(y^<1>, ...y^<Ty> | X)

The most common algorithm for doing this is called beam search, and it's something you'll see in the next video.



3. Why not a greedy search

3.1 Defination:
Greedy search algorithm : generate the first word by just picking whatever is the most likely first word according to the conditional language model. Feed this choosen first word to machine translation model and then pick whatever is the second word that seems most likely, then pick the third word that seems most likely.

3.2: Greedy search problem
Machine translation model object: is to pick the entire sequence of words, y<1>, y<2>, up to y<Ty>, that maximizes the joint probability of that whole thing: P(y<1>, y<2>, ... y<Ty>,|x)

It turns out that the greedy approach, could not get maximizes the join probability.
Greedy search just pick the best first word, and then, after having picked the best first word, try to pick the best second word, and then, after that, try to pick the best third word, doesn't really work.

Example: greedy search problem
have two output translation:
1: " Jane is **visiting** Africa in Septemeber."
2: "Jane is **going** to be visiting Africa in September"

The first one is a better translation, so expect machine translation model will have probability p(y |x) is higher for the first sentence But if use greedy search algorithm, after picked "Jane is" as the first two words as "going" is a more common English word, probably the chance of "Jane is going," given the French input, this might actually be higher than the chance of "Jane is visiting," given the French sentence: p(going | Jane is, X) > p (visitiong | Jane is, X) -->P (Jane is going|x) > P(Jane is visiting|x)
-->

3.2: Greedy search problem

Example: greedy search problem
-->
If just pick the third word based on whatever maximizes the probability of just the first three words, might end up choosing the second translation, and ultimately ends up resulting in a less optimal sentence as measured by this model for p (y|x).

Output sequence word choose
Above is an example of a broader phenomenon: when try to find the sequence of words, y<1>, y<2>, all the way up to the final word that together maximize the probability, it's not always optimal to just pick one word at a time.

And total number of combinations of words in the English sentence is exponentially larger: Picking words from the vocabulary size, is just a huge space of possible sentences, and it's impossible to rate them all, which is why the most common thing to do is use an approximate search out of them.
e.g have 10,000 words in a dictionary and while contemplating translations that are up to ten words long, then there are 10000^10 possible sentences that are ten words long.

Approximate search algorithm: object: it try, it won't always succeed, to pick the sentence y that maximizes that conditional probability p(y|x).
Even though it's not guaranteed to find the value of y that maximizes this, it usually does a good enough job.

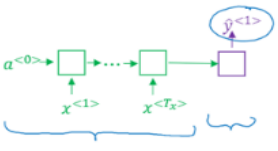
4. Summarize
This video show how machine translation can be posed as a conditional language modeling problem. But one major difference between this and the earlier language modeling problems is rather than wanting to generate a sentence at random, you may want to try to find the most likely English sentence, most likely English translation. But the set of all English sentences of a certain length is too large to exhaustively enumerate. So, we have to resort to a search algorithm. So, with that, let's go onto the next video where you'll learn about beam search algorithm.

5.3-3 Beam search_Seq 2 Seq models

In the last video, you remember how for machine translation given an input French sentence, you don't want to output a random English translation, you want to output the best and the most likely English translation. The same is also true for speech recognition where given an input audio clip, you don't want to output a random text transcript of that audio, you want to output the best, maybe the most likely, text transcript. Beam search is the most widely used algorithm to do this.

1. Beam Search algorithm

1.1: Example_translation:
Inpute: French sentence, "Jane, visite l'Afrique en Septembre"
output: expected being translated into, "Jane, visits Africa in Septemb



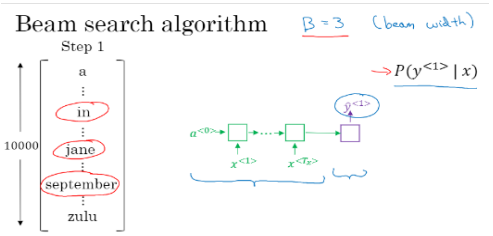
1.2 Process_beam search

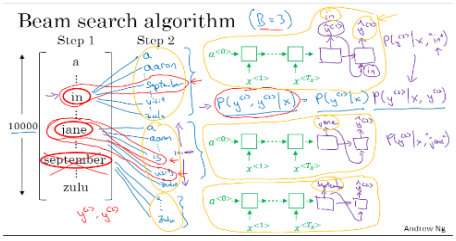
> 0: Pre-process: generate p(y<1>|x)
input French sentence through this encoder network and then this first step will then decode the network y<1>, this is a softmax output overall vocab size like 10,000possibilities.

>1. First words: choose B number of p(y<1>| x)
Take those vocab size like 10,000 possible outputs and keep in memory which were the top B for p(y<1>| x).

Idea: Whereas greedy search will pick only the one most likely words and move on, Beam Search instead can consider multiple B alternatives.
Parameter: B: beam width, the number of possibility Beam search will pick at one time.

e.g: if B=3, while evaluating this probability over different choices the first words, finds that the choices 'in, Jane and September ' are the most likely three possibilities for the first words in the English outputs. Then Beam search will storaway in computer memory that it wants to try all of three of these words, and if the beam width parameter were said differently, the beam width parameter was 10, then we keep track of not just three but of the ten, most likely possible choices for the first word.





> 2. Second word: choose B numbers of P(y^<1>, y^<2> |X) = P(y^<1> |X) * P(y^<2> |X, y^<1>)

Second step object: is to find the pair of the first and second words that is most likely P(y<1>, y<2> |X) , not just a second where is most likely p(y<2> | x, y<1>);

>>> 1. For each of these B choices of first word:
>>>>>1. Consider what should be the second word: p(y<2> | x, y<1>): have vocab size vector.
network frame used: encoder for input + decoder portion for y<1>, y<2>;

>>>>>2. Calculate likely of P(y<1>, y<2> |X) = P(y<2> | X, y<1>) X P(y<1> | X), get vocab size result.
for each of the B words have chosen for the first word y<1>, like "in", "Jane," and "September" , save away this probability P(y<1> |x) ., Then can multiply them by this second probabilities P(y<2> |X, y<1>) to get the probability of the first and second words.

>>>2. for all other B choices of first word ,do the same thing
In the second step of beam search, end up considering B x vocab size possibilities.
As: for each chosed first word, there are vocab size probabilities P(y<2> | X, y<1>).

>>>3. Pick the top B of P(y<1>, y<2> | X)
Evaluate all of these B x vocab size like 30000 options according to the probably the first and second words and then pick the top B.
So with a cut down, these 30,000 possibilities down to the beam width-3 rounded again. that's what Beam's search would memorize away and take on to the next step beam search.

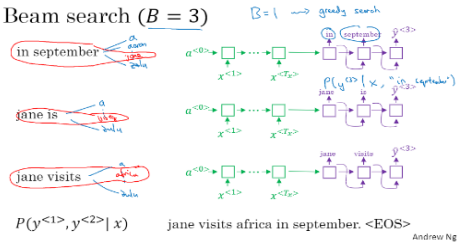
> 2. Second word: choose B numbers of P(y<1>, y<2> |X)

Note:
1. The second step choice may narrow down the first word choice.
e.g: if beam search decides that the most likely choices are the first and second words are :'in September', or 'J'ane is', or 'Jane visits'. Then that means it is now rejecting September as a candidate for the first words of the output English translation.
so we're now down to two possibilities for the first words but we still have a beam width of three keeping track of three choices for pairs of Y1, Y2 before going onto the third step of beam search.
2. In every step, use B copies of the network with different choisces of P(y<1>, ...y<t-1> |X) to evaluate these partial sentence fragments and the output: P(y<t> | X, y<1>, ...y<t-1>)
These B copies of the network can be very efficiently used to evaluate all vocab size X B like 30,000 options for the current step word. So just don't instantiate 30,000 copies of the network, only need B/ three copies of the network to very quickly evaluate all 10,000 possible outputs at that softmax output for y^<t>.

>3rd word: choose B numbers of P(y<1>, y<2>, y<3> |X)
Given B pairs of first two words y<1>, y<2>, find third word, that P(y<1>, y<2>, y<3> |X) is top B.

E.G: that the most likely choices for first two words were " in September, Jane is, and Jane visits"
>>> 1. For each of these B choices of first two words:
Saved away in computer memory the probability of P(y<1>, y<2> |X)
>>>>>>1. Consider what should be the 3rd word: p(y<3> | x, y<1>, y<2>): have vocab size vector.
network frame used: encoder for input + decoder portion for y<1>, y<2>, y<3>;
>>>>>>2. Calculate likely of P(y<1>, y<2>, y<3> |X) = P(y<3> | X, y<1> ,y<2>) * P(y<1>, y<2> | X), get vocab size result.
for each of the B pairs have chosen for the first two word y<1>, y<2> , save away this probability P(y<1>, y<2> |x) ., Then can multiply them by this 3rd probabilities P(y<3> |X, y<1>, y<2>) to get the probability of the three words.
(note: could just pick the top B 3rd word, and multiply probability of previous choosed first two words-->get top B probabilities of three word sentence: p(y^<1>, y^<2>, y^<3>|x))

>>>2. for all other B choices of first two words ,do the same thing
In the 3rd step of beam search, end up considering B x vocab size possibilities.
>>>3. Pick the top B of P(y<1>, y<2>, y<3> | X)
Evaluate all of these B x vocab size like 30000 options according to the probably the first and second words and then pick the top B.
(or evaluate all B X B size options-->pick top B)



2. Summary:
With a beam of B being searched considers B possibilities at a time. If the beam width =1, then this essentially becomes the greedy search algorithm.
By considering multiple possibilities say three or ten or some other number at the same time beam search will usually find a much better output sentence than greedy search. You've now seen how Beam Search works but it turns out there's some additional tips and tricks from our partners that help you to make beam search work even better. Let's go onto the next video to take a look.

5.3-4 Refinments to beam search_Seq 2 seq models.

Length normalization is a small change to the beam search algorithm that can help get much better results

1. Length normalizaiton

Beam search object: maximizing probability: P (y<1>, y<2>, ...y<t> | X) = P(y<1> |X) . P(y<2> |X, y<1>).....P(y <Ty> | X, y<1>, ...y<Ty-1>)

1.1. Original object formula problem:
>1.1.1 Numerical underflow （数值下溢出)--->log.

$$\arg\max_{y} \prod_{t=1}^{T_y} P(y^{<t>} \mid x, y^{<1>}, \dots, y^{<t-1>})$$

>>>1. Problem:
Probabilities are all numbers often much less than 1, multiplying a lot of numbers less than 1 will result in a tiny number, which can result in numerical underflow: too small for the floating part representation in computer to store accurately.
So in practice, instead of maximizing this product, we will take logs.

>1.1.2: Tends to prefer very short translations

>>> 1. Problem
original objective: maximize: P (y<1>, y<2>, ...y<t> | X) = P(y<1> |X) . P(y<2> |X, y<1>).....P(y <Ty> | X, y<1>, ...y<Ty-1>)

If have a very long sentence, the probability of that sentence is going to be low, because multiplying as many terms here that are less than 1 to estimate the probability of that sentence.
If multiply all the numbers that are less than 1 together, you just tend to end up with a smaller probability.

And so this objective function has an undesirable effect, that maybe it unnaturally tends to prefer very short translations. It tends to prefer very short outputs.
As the probability of a short sentence is determined just by multiplying fewer of these numbers are less than 1. And so the product would just be not quite as small.

Note:
The same thing is true for the log (P), P<=1, lop(P) <=0, So the more terms p_i you have together, the more negative log (p) becomes.

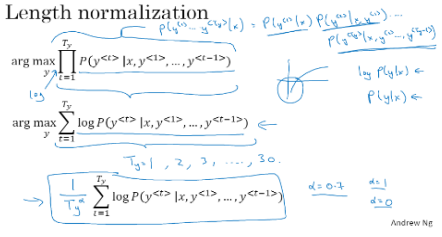
>1.1.1 Numerical underflow （数值下溢出)--->log.

> >>2. Solution: Log functon: log (P) = log (p1) + log (p2) +..log (p_Ty)
>>>1. Multiplicaiton of p_i, turn in to sum of log (p_i).

>>>2. Maximizing this sum of log probabilities should give the same results in terms of selecting the most likely sentence y.
And as log function is strictly monotonically increasing function, then maximizing log P(y|x)should give the same result as maximizing P(y|x).

>>>3. By taking logs, you end up with a more numerically stable algorithm that is less prone to numerical rounding errors （四舍五入）, or to really numerical underflow.

So in most implementations, keep track of the sum of logs of the probabilities rather than the protocol of probabilities.



>1.1.2: Tends to prefer very short translations

>>>2. Solution Normalize length
Normalize length: normalize orginal object by the number of words- Ty translation, so it takes the average of the log (P_i) (<0) of each word, significantly reduces the penalty for outputting longer translations.
a^x: (0<a<1)-->a^x decrease with x increase-->
(p)^(1/Ty) : increase with Ty increase

Heuristic method: 1/ Ty^a
instead of dividing by Ty, by the number of words in the output sentence, sometimes use a softer approach: Ty ^a.

Parameter: a, is hyperparameter that can tune to try to get the best results.
a: may = 0.7.
if a=1, then completely normalizing by length.
If a=0, then not normalizing at all.
a= 0.7: is somewhat in between full normalization, and no normalization

Note: using a this way is a heuristic, no great theoretical justification for it. but people have found this works well.

2. Implement: length normalization in Beach search

Normalized log likelihood objective: log [P(y<1>, y<2>, ...y<t>|X)] / Ty^a

Process:
>1. Run beam search to certain steps: sentence up to certain length
Run beam search like 30 steps and consider output sentences up to length 30. if B=3, then algorithm will be keeping track of the top three possibilities for each of these possible sentence lengths, 1, 2, 3, 4 and so on, up to 30, all sentence output options = B * length

>2. look at all of the output sentences and score them using normalized log likelihood objective log (p) / Ty^a
>>>Take top B sentences P(y<1>, y<2>, ...y<t>|X) in each step/sentence length t=1, 2...30.
>>> compute modified objective function log [P(y<1>, y<2>, ...y<t>|X)] / Ty^a onto them -sentences that have seen through the beam search process.

>3. after all of these sentences validate this way,pick the one that achieves the highest value on this normalized log probability objective log [P(y<1>, y<2>, ...y<t>|X)] / Ty^a
And then that would be the final translation, your outputs.

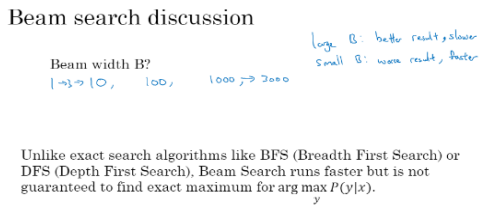
(Note: process:
>1. trained encoder , decoder
>2. usde decoder with Beam search-->get B * length options
>3. use normalized log likelihood evaluate all optoions-->final output)

3. Beam search discussion:

The Larger B is, the more possibilities is considering, and the better the sentence you probably find.
But the the more computationally expensive your algorithm is, as also keeping a lot more possibilities around.

3.1 Concens to choose B:
> 1. If the beam width is very large: then consider a lot of possibilities, and tend to get a better result because you are consuming a lot of different options.
But it will be slower, the memory requirements will also grow, will also be compositionally slower.

>2. If use a very small beam width: then may get a worse result as is keeping less possibilities in mind as the algorithm is running.
But get a result faster and the memory requirements will also be lower.



3. Beam search discussion:

3.2 Choosen B:
> In practice, In production systems, it's not uncommon to see a beam width maybe around 10. and beam width of 100 would be considered very large for a production system, depending on the application.

> For research systems: where people want to squeeze out every last drop of performance in order to publish the paper with the best possible result. It's not uncommon to see people use beam widths of 1,000 or 3,000,

3.3 Note:
when B gets very large, there is often diminishing returns:
for many applications, there is a huge gain as increase beach width from like 1, which is very greedy search, to 3, to maybe 10. But the gains as you go from 1,000 to 3,000 in beam width might not be as big.

4. Summarize

There are exact search algorithms: computer science search algorithms like BFS, Breadth First Search, or DFS, Depth First Search.
The way to think about beam search is that, unlike those other algorithms, Beam search runs much faster but does not guarantee to find the exact maximum for this arg max that you would like to find. This is how beam search relates to those algorithms.

Beam search is a widely used algorithm in many production systems, or in many commercial systems. Now, in the circles in the sequence of courses of deep learning, we talked a lot about error analysis. It turns out, one of the most useful tools I've found is to be able to do error analysis on beam search. So you sometimes wonder, should I increase my beam width? Is my beam width working well enough? And there's some simple things you can compute to give you guidance on whether you need to work on improving your search algorithm. Let's talk about that in the next video.

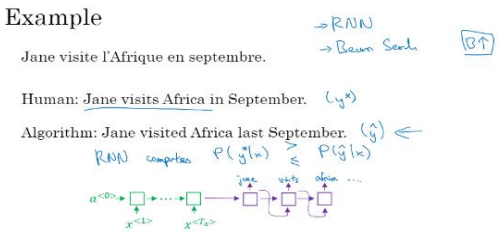
5.3-5 Error analysis on beam search_ Seq 2 Seq models

In the third course of this sequence of five courses, you saw how error analysis can help you focus your time on doing the most useful work for your project.
Now, beam search is an approximate search algorithm, also called a heuristic search algorithm. And so it doesn't always output the most likely sentence. It's only keeping track of B equals 3 or 10 or 100 top possibilities. So what if beam search makes a mistake? In this video, you'll learn how error analysis interacts with beam search and how you can figure out whether it is the beam search algorithm that's causing problems and worth spending time on. Or whether it might be your RNN model that is causing problems and worth spending time on.

1. Example_translation output mistake

1.1: Example:
Dev set: input sentence: Jane visite l'Afrique en septembre
Dev set: labeled output (human tranlation) y*: Jane visits Africa in Septembe
Run beam search on learned RNN model (conditional language model): ends up translation y^: Jane visited Africa last September

Model output is a much worse translation of the French sentence. It actually changes the meaning, so it's not a good translation.



1.2: Machine translation:
Machine translation model has two main components:
>1. RNN: Neural network model/ the sequence to sequence model = encoder + decoder
>2. Beam search algorithm: with beam width B.

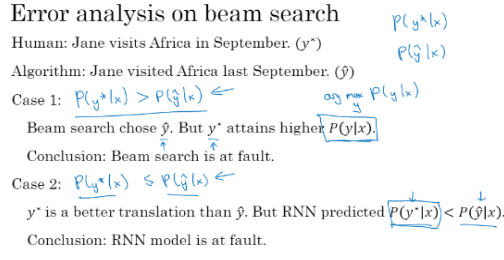
Try to find which components contribute to translation error, RNN or beam search.

Note:
as t it's always tempting to collect more training data that never hurts. So in similar way, it's always tempting to increase the beam width that never hurts or pretty much never hurts.
But just as getting more training data by itself might not get you to the level of performance you want. In the same way, increasing the beam width by itself might not get you to where you want to go.

2. Error analysis
To decide whether or not improving the search algorithm is a good use of your time, need to break the problem down and figure out what's actually a good use of your time.

2.1 Principle:
>1. RNN function (conditional language model): given input ,computes each output sequence word probability: P(y^<1> |X), P(y^<2> |X, y<1>),...P(y^<Ty> |X, y<1>, y<2>, ..y<Ty-1>))
(while traing conditional langue model, input x<t> should = correct labeled y<t-1>?, yes: decoder actually is trained language model)
>2. Beam search function: try to find a value of y that gives that arg max P(y^<1>, y^<2>, ..y^<Ty>|x) = P(y^<1> |X). P(y^<2> |X, y^<1>).....P(y^<Ty> |X, y^<1>, y^<2>, ...y^<Ty-1>)).

e.g: for a sentence, Jane visits Africa in September, RNN compute P(Jane|X), P(visits|X, Jane)....P(September|X, Jane, ..in)



2.2 Process: break down problem
>1. using this model RNN to compute P(y*|x) and P(y| x) using your RNN model.
P(y*|x) = P(y*<1>|x) . P(y*<2>|x, y*<1>) . P(y*<3>|x, y*<1>, y*<2>).....P(y*<Ty>|x, y*<1>, y*<2>,...y*<Ty-1>)

>2. Compare P(y*|x) and P(y| x):
And then to see which of these two is bigger.

>>>1. P(y*|x) as output by the RNN model > P(y| x)---->Beam search fault
now sentence y* actually attains a higher value for P(y* | x) than the y^ sentence. So come to conclusion beam search is failing to actually give the value of y that maximizes P(y|x) .
as one job that beam search had was to find the value of y that makes this really big. But it chose y^, the y* actually gets a much bigger value. So in this case, you could conclude that beam search is at fault.

>>>2. P(y* |x) <= P(y^hat | x)------>RNN fault

In example, y* is a better translation than y^, but according to the RNN, P(y*|x) is less than P(y^|x): means y* is a less likely output than y^. So in this case, it seems that the RNN model is at fault and it might be worth spending more time working on the RNN.

Note: if using some sort of length normalization evaluating output sentence probabilities, then should be evaluating the optimization objective that takes into account length normalization while comparing : may 1 / (Ty*)^a* log(y*|x) and 1/ (Ty^)^a* p(y^|x).

should use seperatedly?? compare P(y* | X) + P(y^ | X), and log P(y*|X) / Ty^a + log P(y|X) / Ty^a
RNN fault: P(y* | X) <= P(y^ | X)
Beam search fault: log P(y*|X) / Ty*^a > log P(y|X) / Ty^a

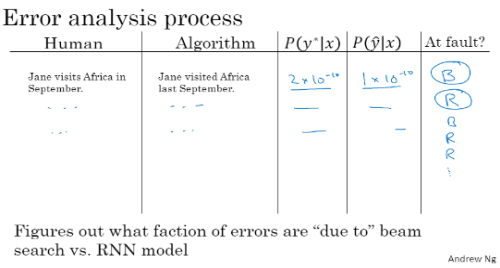
What if : log P(y*|X) / Ty*^a <= log P(y|X) / Ty^a & P(y* | X) > P(y^ | X)
log P(y*|X) / Ty*^a <= log P(y|X) / Ty^a --->beam search no fault-
P(y* | X) > P(y^ | X)---->RNN no fault. ??

3.Do error analysis:

3.1 Process
>1. Go through the development set and find the mistakes that the algorithm made in the development set.
>2. Bread dowk probelm, get conclusiton for each mistakes, conributor is RNN or beam search;
e.g: for the first mistakes, compute P(y* | x) = 2 x 10 ^ -10, P(y^ | x) =1 x 10^-10. -->beam search is at fault.
then go through a second mistake, look at these probabilities, and think the model is at fault. And then go through more examples, sometimes the beam search is at fault, sometimes the model is at fault, and so on.

>3. Carry out error analysis to figure out what fraction of errors are due to beam search versus the RNN model.

3.2 Summarize
With an error analysis process like this, for every example in dev sets, where the algorithm gives a much worse output than the human translation, you can try to ascribe the error to either the search algorithm or to the objective function, or to the RNN model that generates the objective function that beam search is supposed to be maximizing.
And through this, you can try to figure out which of these two components is responsible for more errors. And only if you find that beam search is responsible for a lot of errors, then maybe is we're working hard to increase the beam width. Whereas in contrast, if you find that the RNN model is at fault, then you could do a deeper layer of analysis to try to figure out if you want to add regularization, or get more training data, or try a different network architecture, or something else. And so a lot of the techniques that in the third course in the sequence will be applicable there.



4. Summarize:
This particular error analysis process very useful whenever you have an approximate optimization algorithm, such as beam search that is working to optimize some sort of objective, some sort of cost function that is output by a learning algorithm, such as a sequence-to-sequence model or a sequence-to-sequence RNN that we've been discussing in these lectures. So with that, I hope that you'll be more efficient at making these types of models work well for your applications.

5.3-6 Bleu score_seq 2 seq models

One of the challenges of machine translation is that, given a French sentence, there could be multiple English translations that are equally good translations of that French sentence. So how do you evaluate a machine translation system if there are multiple equally good answers, unlike, image recognition where there's one right answer , just measure accuracy?
If there are multiple great answers,to measure accuracy, the way this is done conventionally is through something called the BLEU score. So, in this optional video, I want to share with you, I want to give you a sense of how the BLEU score works.

1. Evaluation machine translation

1.1 Example
Inpute: French sentence: "Le chat est sur le tapis"
Labeled output: human generated translation: there are multiple, pretty good translations of this. So a different human might translate it differently.
1: the cat is on the mat.
2. there is a cat on the mat.

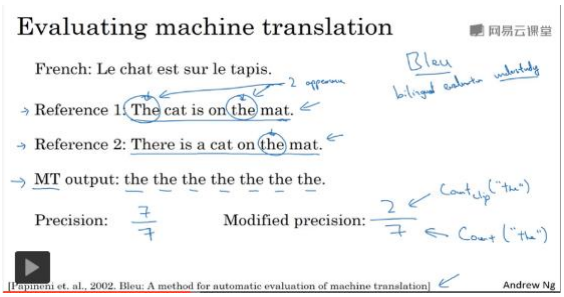
1.2 BLEU Score:
>1. BLEU score function: given a machine generated translation, allows to automatically compute a score that measures how good is that machine translation.

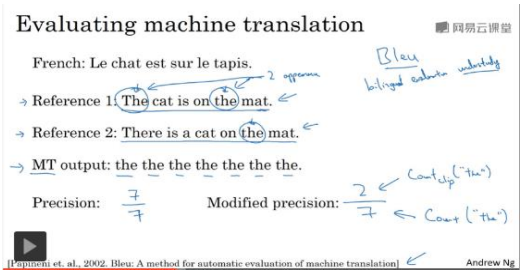
>2. BLEU score intuition/Principle: so long as the machine generated translation is pretty close to any of the references provided by humans, then it will get a high BLEU score.
The intuition behind the BLEU score is we're going to look at the machine generated output and see if the types of words it generates appear in at least one of the human generated references. And so these human generated references would be provided as part of the dev set or as part of the test set.

>3. BLEU defination: stands for bilingual evaluation Understudy(双语评估替补)
In the theater world, an understudy is someone that learns the role of a more senior actor so they can take over the role of the more senior actor, if necessary.

And motivation for BLEU is that, whereas you could ask human evaluators to evaluate the machine translation system, the BLEU score is an understudy, could be a substitute for having humans evaluate every output of a machine translation system.

BLEU score was due to Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. This paper has been incredibly influential, and is, actually, quite a readable paper-marked in right red line.





2. Method to measure how good machine translattion output is

Example: MT output: " the the the the the the the."

2.1: Basic Precision for isolated word: of the machine translation output:

>1. MT output basic precision defniation: Look at each the words in the output and see if it appears in the references; measure what fraction of the words in the MT output also appear in the references.
n= MT output sentence word number;
n_i= i-th word of output sentence, appear in referance 1or not 0;
Precision = sum n_i over all words /n

>2. Basic precison problem:
This is not a particularly useful measure, because it seems to imply that this MT output has very high precision.

e.g: in above example, there are seven words in the machine translation output. And every one of these 7 words appears in either Reference 1 or Reference 2, So the word 'the' appears in both references. So each of these words looks like a pretty good word to include.---> So this MT output have a precision of 7 / 7.
It looks like it was a great precision.

So this is how the basic precision measure of what fraction of the words in the MT output also appear in the references. This is not a particularly useful measure, because it seems to imply that this MT output has very high precision.

2.2: Modified Precision for isolated word: of the machine translation output

>1. Modified precision defination:Look at each the words in the output and see if it appears in the references; In measure give each word credit only up to the maximum number of times it appears in each reference sentences.
n= MT output sentence word number;
n'_i= i-th word of output sentence, appear times in referance (credit only to the maximum number of times it appears in each referance sentence;
n''_i= i-th word of output sentence, appear times in per se
Precision = sum n'_i over all words / sum n''_i over all output words.

e.g:
word 'the' occure in output 7 times, n''_i=7 (i=1);
word 'the' in Reference1 appears twice, in Reference 2 appears just once. n'_i =2
So, with a modified precision, it gets a score : 2/7

Modified Precision the denominator : is the sum of the count of the number of times each output word, appears per se.
Modified precision numerator: is the sum of the max count of the number of times each output word appears in each referance.

>2. Problem:
Have been looking at words in isolation.

3. Bleu score on bigrams:

In the BLEU score, you don't want to just look at isolated words. You maybe want to look at pairs of words as well.

>3.1 Defination:
Bigrams: means pairs of words appearing next to each other.

Blue score on bigrams: a portion of the final BLEU score
final BLEU score: take unigrams-single words, as well as bigrams-pairs of words into account as well as maybe even longer sequences of words, such as trigrams,-three words pairing together.

3.2 Example: MT ouput2: "The cat the cat on the mat". Still not a great translation, but maybe better than the last one.

3.3 Blue score computing :
>1. List all the bigrams in MT output sentence
e.g: "the cat", cat the, the cat - as already had that, so skip that, cat on, on the, the mat.
So these are the bigrams in the machine translation output.

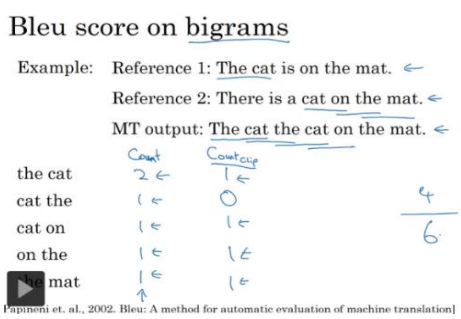
>2. Count up, How many times each of these bigrams appear in MT output per se
e.g: " The cat" appears twice, cat the appears 1, and the others all appear just once.

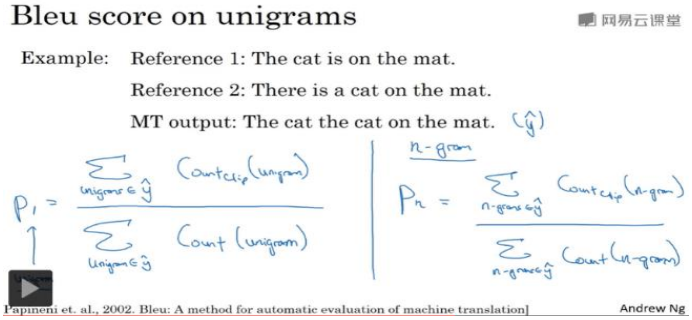
>3. Define the clipped count, count, and then subscript clip in referance output
Take all the bigrams, and give our algorithm credit only up to the maximum number of times that that bigram appears in either Reference 1 or Reference 2.

e.g: the cat appears a maximum of once in either of the references. So clip that count to 1. Cat the, doesn't appear in Reference 1 or Reference 2, so clip that to 0. Cat on, appears once, give it credit for once. On the appears once, give that credit for once, and the mat appears once. So these are the clipped counts.

We're taking all the counts and clipping them, really reducing them to be no more than the number of times that bigram appears in at least one of the references.

>4. modified bigram precision computation:
modified bigram precison = sum of the count clipped/ sum of the count per se(MT output)
e.g: sum of the count clipped is 1+0+1+1+1 = 4 / sum of the count per se(MT output)=2+1+1+1+1= 6 =4/6





4. Bleu score on unigrams

4.1 Modified precision on unigrams: P1
P stands for precision and the subscript 1 here means that we're referring to unigrams, meaning we're looking at single words in isolation.

p1 denominator （分母）: sum of unigrams in the machine translation output of count, number of counts of that unigram,
P1 numerator（分子）: sum of unigrams in the machine translation output of count, number of count clip of that unigram,

4.2 Modified precision on n-gram: Pn, instead of unigram, for n-gram.

p1 denominator （分母）: sum of n-gramss in the machine translation output of count, number of counts of that unigram,
P1 numerator（分子）: sum of n-grams in the machine translation output of count, number of count clip of that unigram,

And so these modified precision scores, measured on unigrams or on bigrams, or on trigrams, or even higher values of n for other n-grams. This allows you to measure the degree to which the machine translation output is similar or maybe overlaps with the references.

4. Bleu score on unigrams

4.3 Note: Pn=1
If the MT output is exactly the same as either Reference 1 or Reference 2, then all of these values P1, and P2 and so on, they'll all be equal to 1.0.
So to get a modified precision of 1.0, have to be exactly equal to one of the references. sometimes it's possible to achieve this even if you aren't exactly the same as any of the references, but you kind of combine them in a way that hopefully still results in a good translation.

5. Bleu details:

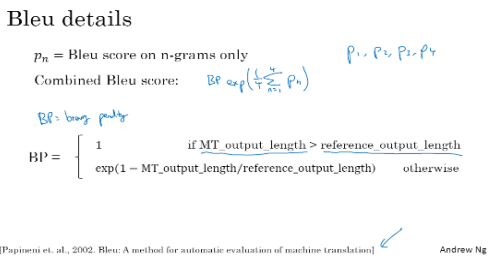
Finally, let's put this together to form the final BLEU score.

Pn: the BLEU score computed on n-grams only. Also the modified precision computed on n-grams only.

5.1 Final BLEU Score: combined Bleu score: BP * exp (average (P1, P2, ...Pn))
Combining the computed P1 ,P2, ..Pn together by above formula.

>1. average (P1, P2, ...Pn):
e.g: by convention to compute one number, you compute P1, P2, P3 and P4, take the average of them

>2. exp (average):
By convention the BLEU score is defined asexponentiations (求幂)of average score, exponentiation is strictly monotonically increasing operation.



5. Bleu details:

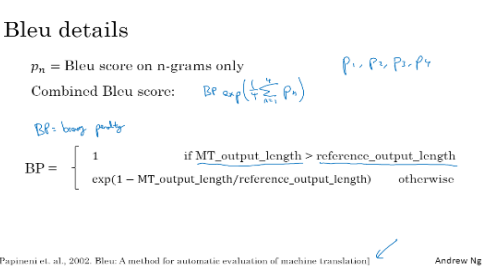
>3. BP:BP penalty:
Added to adjust convention BLEU score.

>>>1. Motivaiton: it turns out that if MT output very short translations, it's easier to get high precision Pi. Because probably most of the words you output appear in the references.
But we don't want translations that are very short.

>>>2. Funtion: BP is an adjustment factor that penalizes translation systems that output translations that are too short.

>>>3. Formula:
If length of MT output > referance length, BP =1 , (>min(referances??)
Otherwise: BP = e^ (1- length of MT/ length of refereance) : 1<BP<2.7, should <1,something wrong?

Note: the details can find in this paper-right.



6. Summarize

1. Earlier in this set of courses, have shown the importance of having a single real number evaluation metric: it allows you to try out two ideas, see which one achieves a higher score, and then try to stick with the one that achieved the higher score.

2. So the reason the BLEU score was revolutionary for machine translation was: because this gave a pretty good, by no means perfect, but pretty good single real number evaluation metric. And so that accelerated the progress of the entire field of machine translation.

3. In practice, few people would implement a BLEU score from scratch. There are open source implementations that you can download and just use to evaluate your own system.
(seems very simple algorithm, need to C/O?)

4. Today, BLEU score is used to evaluate many systems that generate text.
Such as machine translation systems, as well as the example I showed briefly earlier of image captioning systems where you would have a system, have a neural network generated image caption. And then use the BLEU score to see how much that overlaps with maybe a reference caption or multiple reference captions that were generated by people.

So the BLEU score is a useful single real number evaluation metric to use whenever you want your algorithm to generate a piece of text. And you want to see whether it has similar meaning as a reference piece of text generated by humans. This is not used for speech recognition, because in speech recognition, there's usually one ground truth. And you just use other measures to see if you got the speech transcription on pretty much, exactly word for word correct. But for things like image captioning, and multiple captions for a picture, it could be about equally good, or for machine translation there are multiple translations, but equally good. The BLEU score gives you a way to evaluate that automatically and therefore speed up your development.

(Note: BLEU: billingual evaluation understudy: method:
1. basic precision: for isloaion word in MT: credit 1/0 if occure/not occure in ref.-->sum credit of all word in MT output sentence/ MT output sentence length
2. modified precison: for unigram/bigram/trigram word in MT: count clip/count (in MT per se)

-->final BLEU used: BP EXP (average of sum p1, ..pn) n: nigram of modified precison.)

5.3-7 Attention model intution_seq 2 seq models

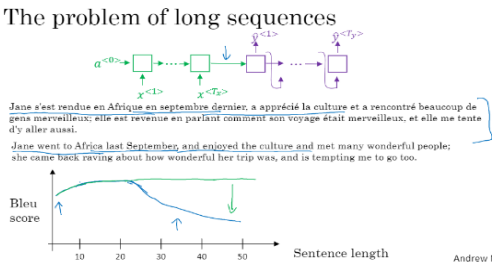
For most of this week, you've been using a Encoder-Decoder architecture for machine translation. Where one RNN reads in a sentence and then a different one RNN outputs a sentence.
Attention Model make a modification to this, and it makes all this work much better. The attention algorithm, the attention idea has been one of the most influential ideas in deep learning.

1. The problem of long sequence

Example: input long sentence as right.

1.1 MT output principle: : encoder + decoder:
Encoder read in the whole sentence and then memorize the whole sentences and store it in the activations conveyed here. Then the decoder network will generate the English translation.

1.2 Human translator principle:
human translator Kind of , work part by part through the sentence. Because it's just really difficult to memorize the whole long sentence like that.
Not to first read the whole French sentence and then memorize the whole thing and then regurgitate an English sentence from scratch. Instead, :
>1. Read the first part of input sentence, maybe generate part of the translation.
>2. Look at the second part, generate a few more words
>3. Look at a few more words, generate a few more words and so on.

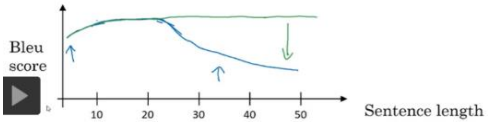


1.3 MT output problem:

The Encoder-Decoder architecture above, works quite well for short sentences, so we might achieve a relatively high Bleu score, but for very long sentences, maybe longer than 30 or 40 words, the performance comes down.

1.3.1 MT output Bleu score vs sentence length-right blue curve:
>1. Sentence is very short: Bleu score small
short sentences are just hard to translate, hard to get all the words

>2. Sentence reasonable long, Bleu score is big
>3. Sentence very long , Blue score is small
Very long sentences, it doesn't do well because it's just difficult to get in your network to memorize a super long sentence.
(encoder result C , reprezentation of input sentence, could not remember all input sentence info. when it is too long.)

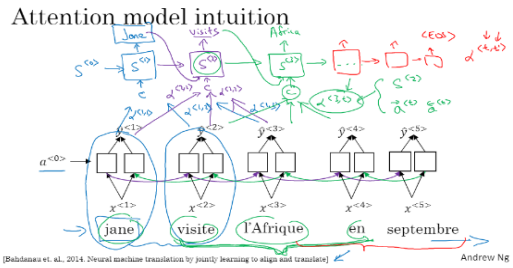


1.3.2 With Modified MT output, attention Model:
which translates maybe a bit more like humans might, looking at part of the sentence at a time, machine translation systems performance can look like this-right green curve: Bleu score is big even input sentence is very long:
as by working one part of the sentence at a time, you don't see this huge dip (gap between green curve and blue curve while sentence is very long), which is really measuring the ability of a neural network to memorize a long sentence, which maybe isn't what we most badly need a neural network to do.
(note: as long as encoder result C could remember enough info. of input sentence, than output sentence BELUE is high-->green curve.
but may not need this remembering long sentence ability to achieve this high BLEU, could modify architeture to reach this high BLEU also)

2. Attention model intuition

The Attention Model was due to Dimitri, Bahdanau, Camcrun Cho, Yoshe Bengio and even though it was obviously developed for machine translation, it spread to many other application areas as well.
This is really a very influential, seminal paper in the deep learning literature.

Example:
illustrate this with a short sentence: Jane visite l'Afrique en Septembre
even though these ideas were maybe developed more for long sentences, but it'll be easier to illustrate these ideas with a simpler example.



Architecture:
2.1. 'Encoder'-BRNN:
Generate features of each word/maybe surrongding words in input sentence word position
In this example use a modified BRNN, in order to compute some set of features for each of the input words.

BRNN: Standard BRNN outputs y<1>, y<2>, ...y<Ty>.
Modified BRNN here: standard BRNN modified here as we're not doing a word for word translation: get rid of the y on top, instead using a BRNN for each of the wrod/time sequence in input sentence, **compute a very rich set of features about the words in the sentence and maybe surrounding words in every position.**

2.2. 'Decoder'- RNN: Generate translation output words
Use another RNN to generate the English translations.

Activation Denote:
in order to avoid confusion with the activations here, use a different notation ' S' to denote the hidden state in this RNN.

Horizental forwad: S<0>------<S<1>--------->S<2>------>.....------------>S<Ty>,
Vertical forward: y<1>, y<2>, ...y<Ty>

Process:
>1. First word y<1>
when trying to generate this first output word y<1>, what part of the input French sentence should you be looking at.
e.g: In this example seems like should be looking primarily at this first input word 'jane', maybe a few other words close by, but don't need to be looking way at the end of the sentence.

>>> 1. Attention Model functon: compute a set of attention weights and we're going to use: a<1, t>, t=1,...Tx
a<1,1>: denot when genrating the first words, how much should you be paying attention to this first piece of information here: feature of the first word of input setence (which is y<1> in standard BRNN)
a<1, 2>: then we'll also come up with a second weight, which tells us what trying to compute the first work of Jane, how much attention we're paying to this second feature vector from the second word in inputs and so on and a<1,3>and so on, to a<1, Ty)

>2. Denote C of all these activation together: tell what is exactly the context from denoter C that we should be paying attention to.
>3. Input C<1> to RNN unit S<1>, then try to generate the first word y<1>

That's one step of the RNN

2.2. 'Decoder'- RNN: Generate translation output words

Process:
>2. For the second step of this RNN
>1. have a new hidden state S<2> and have a new set of the attention weights: a<2,1>, a<2,2>,...a<2, Tx>

>2. first word generated y^<1> also feed as input

>3. C<2>: of together all a<2,1>, ...a<2, Tx>, some context that we're paying attention to a

>4, feed S<1>,y<1>, C<2> to S<2>, generated the second output word y<2>

> Third step,
>1. have a new hidden state S<3> and new context C<3> that depends on the various a<3,1>, ...a<3, Tx>.
C<3>: tells how much should we be paying attention to the different words from the input French sentence and so on.

3. Note:
1. context C<t>: the goal of the context C is: for the t-th output word, is really should capture , maybe we should be looking around surronding (like 2) words of of t-th input word.

2. a<3,t>:
when trying to generate the third word, the amounts that this RNN step should be paying attention to the input (French) word that time t, that depends on:
>1. a<t>, The activations of the bidirectional RNN at time t, which depends on the forward prop activations and backward activations at time t
>2. s<2> and it will depend on the state from the previous steps,S<2>
These things together will influence, how much you pay attention to a specific word in the input French sentence.

4. The key intuition :'Decoder' RNN marches forward generating one word at a time, until eventually it generates maybe the EOS and at every step, there are these attention weighs a<t,t'>, that tells it, when you're trying to generate the t-th output(English) word, how much should you be paying attention to the t'-th input(French) words.
And this allows it on every time step to look only maybe within a local window of the input (French) sentence to pay attention to, when generating a specific output (English) word.
(note: how works actually in live:
1. need to input long sentence , then make prediction?-->could not make predcition before injesting whole sentence?
2. need to compute weight on all input sentence words' feature, input to decode-contex c<t, t'> also need to remember enough /comput all input words weight/info.-do not save from membering long sentence?)

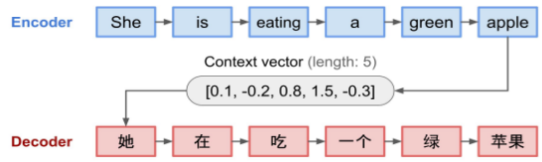
原始的 Seq2Seq 模型有什么问题

Seq2Seq 是为了解决 language modeling 问题而被提出的 (Sutskever, et al. 2014)。 通俗的说，它的目标是将输入的一个句子(source) 转化成一个新的目标句子(target)，同时输入和输出的句子长度可以是任意的。Seq2Seq 的应用领域比较广泛，有机器翻译(文字或者语音)、自动问答和自动摘要等等。。。

传统的 Seq2Seq 都拥有一个 encoder-decoder 的架构：

- Encoder 负责处理输入，并将输入的信息进行 encode，转化成一个固定长度的 context 向量，这个 context 向量包含了整个输入的信息。
- Decoder 负责接收 context 向量并将其转化成输出，早期的 Seq2Seq 模型只是利用了encoder 最后一个 state 作为 decoder 的初始 state。

Encoder 和 Decoder 通常使用 LSTM 或者 GRU。



The encoder-decoder model, translating the sentence "she is eating a green apple" to Chinese. The visualization of both encoder and decoder is unrolled in time.

将 context 设定为一个固定长度的向量有一个明显的缺点，就是当句子长度增加之后，它并不能很好的记住所有信息，一个典型的栗子就是，通常当他处理完所有的输入之后，输入的前一部分很多信息就已经遗忘啦。。。很显然，想用一个 context 向量来承当编码输入句子的重担，这任务有点重。。

所有，为了解决这个问题，Attention Mechanism (Bahdanau et al., 2015) 应运而生 。

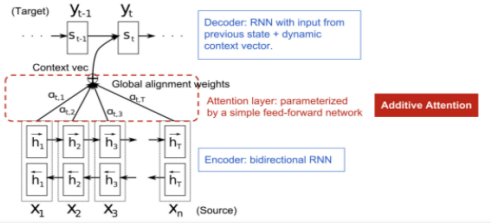
为 Translation 而生

Attention Mechanism 起初为了在机器翻译任务中，帮助更好的记忆长输入的情形。和原始的只是通过 encoder 最后一个 state 构建出 context 向量不同的是，Attention 在 context 向量和 input 之间构建了很多个 shortcuts (可以理解成 Attention 矩阵)，这些 shortcut 的权重相对于不同的 output，都是不同的。

现在，在有了 Attention Mechanism 之后，相当于我们有了新的 context 向量，此时，输入和输出之间的对齐关系 (alignment relation) 可以被 context 向量学习到。

本质上来说，context 向量由以下三个部分组成。

- encoder hidden states;
- decoder hidden states;
- alignment between source and target.



引入Attention

上面的模型使用的还依然是RNN，所以在处理长文本的时候比较困难，效果也不太理想，这个时候就引入了**注意力机制**，可以让模型只注意输入相关的部分。

加入注意力机制后，和以前的Seq2seq有两点不同。

第一点：以前的Seq2seq是只传递最后一个的隐藏层，而加入Attention之后会传递所有隐藏层

encoder

5.2-8 Attention model_seq 2 seq model

Attention model allows a neural network to pay attention to only part of an input sentence while it's generating a translation, much like a human translator might. Let's now formalize that intuition into the exact details of how you would implement an attention model.

1. Attention model

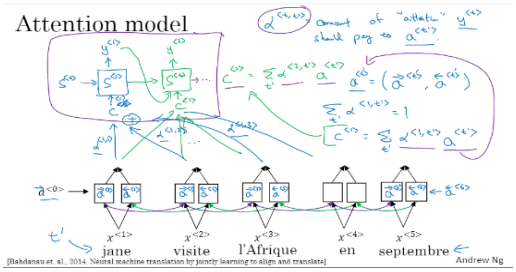
Example: inpute sentence-as right.

1.1: Encoder:-BRNN--->compute features on every word

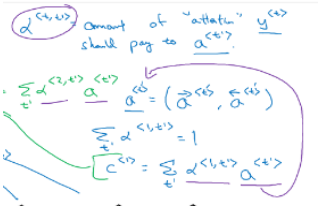
>1. Encoder neural network:
Here use BRNN, or bidirectional GRU, or bidirectional LSTM to compute features on every word instead of standard BRNN output-translation.
In practice, GRUs and LSTMs are often used for this, with maybe LSTMs be more common.

>2. Denotion
α<t'> = (α<t'>_f, α<t'>_b): feature vector'-th word in input sentence
α<t'>_f: activation in forward prop of t'-th word in input sentence
α<t'>_b: activation in backward prop of t'-th word in input sentence

Forward: α<0>----->α<1>_f---------->α<5>_f
backward: α<6> ------>α<5>_b----->α<1>_b
note: a<0>, a<6> could set to 0 vector.



1.2: Decoder:



y<t>=g1(Wy* S<t> + by);
S<t>= g2 (Ws * [y<t-1>, c<t>] +bs)
c<t>= sum a<t,t'> * α<t'> over t' (input time step)

α<t'>: [α<t'>_f, α<t'>_b]: generated by encoder using modified BRNM
a<t,t'> = exp(e<t, t'>) / sum (exp(e<t, t'>) over input words- t'=1--Tx)
e<t, t'> = g3(We* [a<t'>, s<t-1>] + be)

1.2.1 Neural network:
As in decoder only have our forward prop, so it's a single direction RNN with state S<t> to generate the translation.

1.2.2 Process:

>1. Fisrt word: S<0> + C<1> ---->y<1>
>>>1. Inpute: S<0>, and some context C<1>

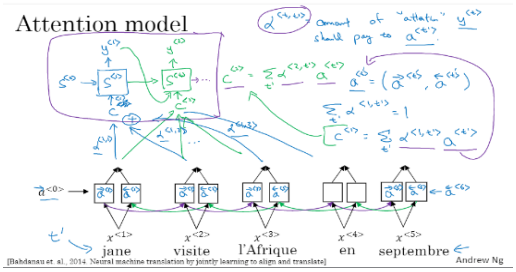
>>>2. Contex C<1>: = sum ϑ<1, t'> . α<t'> over t' all input words.
depend on the attention parameters ϑ<1, 1> , ϑ<1, 2> , and so on.

>>>3. Attention paramter: ϑ<1, t'> :
tells how much the context C<1> would depend on the features α<t'> gotten from the different time steps of input sentence.
So the way we define the context C is actually be a way to sum the features from the different time steps weighted by these attention weights.

Attention weight ϑ<t, t'> satisfy:
>>>>>>1. All be non-negative, >=0
>>>>>>2 sum over all time steps = 1: sum ϑ<t, t'> over all t' words in input sentence =1

So ϑ<t, t'>: is the amount of attention that's y<t> should pay to a a<t'>; in other words, when generating the t-th of the output words, how much you should be paying attention to the t'-th of input word.

So that's one step of generating the output and then at the next time step.



>2. Second word: y<1> + C<2> (<-- S<1> **) ---->y<2>**

Performe similarly as step 1, only now have a new set of attention weights ϑ<2, t'> and they find a new way to sum, generates a new context C<2>.

Note: now y<1> is also input as well as S<1>, C<2> to generate the second word.

Summarize:
Using these context vectors C1 , C2, and so on as the input of decoder.
The encoder network looks like a pretty standard RNN sequence with the context vectors C <t> as output and we can just generate the translation y<t> one word at a time.

2. Computing attention ϑ<t, t'>

2.1 Defination: ϑ<t, t'>
ϑ<t, t'> : is the amount of attention should paid to α<t'> when trying to generate the t-th words in the output translation.

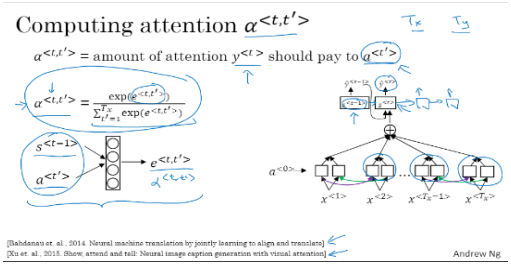
2.2 Formula: ϑ<t, t'> = exp(e<t, t'>) / sum (exp(e<t, t'>) over input words- t'=1--Tx)
Note: use essentially a softmax to make sure that these weights ϑ<t, t'> sum to one if sum over t'.
So for every fix value of t, these weights sum to one if summing o

2.3 Factor e<t,t'> computation: use a small neural network
One way to compute factor e<t, t'> is to use a small neural networ

$$\alpha^{<t,t'>} = \frac{\exp(e^{<t,t'>})}{\sum_{t'=1}^{T_x} \exp(e^{<t,t'>})}$$

2.3.1 Input:
>1. S<t-1>: neural network state from the previous time step
While trying to generate y<t>, the hidden state from the previous step S<t-1> just fell into S<t>.
S<t> is one hidden layer in neural network because you need to compute these S<t> a lot.

>2. α<t'>: feature vecotor of t'-th input word.



2.3 Factor e<t,t'> computation: use a small neural network

2.3.2 Neural network intuition of e<t,t'> :
If you want to decide how much attention to pay to the activation α<t'> , seems like it should depend the most on: what is your own hidden state activation from the previous time step.
You don't have the current state activation s<t> yet because of context C<t> feeds into s<T> , so you haven't computed S<t>.

But look at whatever the hidden stages of this RNN generated in the upper translation S<t-1> and then look at the features of each input words α<t'> . So it seems pretty natural that weight parameter ϑ<t, t'> and e(t, t') should depend on these two quantities α <t'> , S<t-1> .

But we don't know what the function is to generate e<t, t'> based on α<t'> , S<t-1> .
So one thing could do is just train a very small neural network to learn whatever this function should be. And trust backpropgation, trust gradientd descent to learn the right function.

It turns out that if you implemented this whole model and train it with gradient descent, the whole thing actually works.
This little neural network does a pretty decent job telling you how much attention y<T> should pay to α<t'> .

Note:
what is this neural architecure: layer number, hidden untis number, etc.
what's cost function of a<t, t'>?? no need, combine with whole RNN optimizing?

2.4 Summarize:

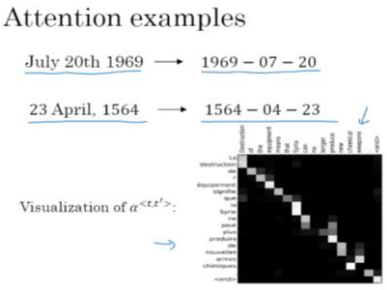
1. As chug along generating one word at a time, this neural network actually pays attention to the right parts of the input sentence that learns all this automatically using gradient descent.

2. Disadvantage: of this algorithm: it take quadratic time / quadratic cost (三次方) to run this algorithm:
e.g: input sentence and output sentence lenght Tx Ty, then the total number of these attention parameters is Tx . Ty. And so this algorithm runs in quadratic cost.
In machine translation applications neither input nor output sentences is usually that long, maybe quadratic cost is actually acceptable, and there is some research work on trying to reduce costs as well.

3. This idea has been applied to many other problems as well, like image capturing.
e.g: in the image capturing problem the task is to look at the picture and write a caption for that picture. So Look at the picture and pay attention only to parts of the picture at a time while you're writing a caption for a picture.
In paper set to the bottom by Kevin Chu, Jimmy Barr, Ryan Kiros, Kelvin Shaw, Aaron Korver, Russell Zarkutnov, Virta Zemo, and Andrew Benjo they also showed that you could have a very similar architecture.

3. Attention examples

3.1 Date normalization problem:
Machine translation is a very complicated problem, in the prior exercise get to implement and play of the attention while for the date normalization problem: inputting a date like "July 20th 1969" and normalizing it into standard formats like "1969-07-20".
you can train a neural network to input dates in any of these formats and have it use an attention model to generate a normalized format for these dates.



3.2 visualize attention weight ϑ<t, t'>
right pic. example: a machine translation example and plotted in different colors, the magnitude of the different attention weights.
could find that for the corresponding input and output words, the attention weights will tend to be high. Thus, suggesting that when it's generating a specific word in output, usually paying attention to the correct words in the input.
All this including learning where to pay attention, when, was all learned using backpropagation with an attention model.

So that's it for the attention model really one of the most powerful ideas in deep learning

5.3-9_Speech recognition_Audio data

One of the most exciting developments were sequence-to-sequence models has been the rise of very accurate speech recognition. here to give you a sense of how these sequence-to-sequence models are applied to audio data, such as the speech.

1. Speech recognition problem

1.1 Defination:
Speech recognition problem: given an audio clip x, to automatically find a text transcript y.

1.2 Training set
>1. Inpute: audio clip:
audio clip: saying "the quick brown fox"
plot it looks right pic (grey pic.): the horizontal axis is time, vertical is air pressure;

Note: sound principle
Microphone measures minuscule changes in air pressure, and the way of hearing my voice is : your ear detects little changes in air pressure, generated either by speakers or a headset.
And some audio clips like this plots with the air pressure against time.

>2. Output: transcript.
Hopefully, a speech recognition algorithm can input that audio clip and output that transcript "the quick brown fox".

2. Speech recognition model history:
2.1 . Build phonems （音节）cell:
speech recognition systems used to be built using phonemes- hand-engineered basic units of cells:
e.g: "the quick brown fox " represented as phonemes: simplify for illestration: Quick, has a "ku" and "wu", "ik", "k" sound.

Linguist used to write off these basic units of sound, and try to break language down to these basic units of sound. and linguists used to hypothesize that writing down audio in terms of these basic units of sound called phonemes would be the best way to do speech recognition.

2. 2. End-to-end deep learning <------large data sets
with end-to-end deep learning, phonemes representations are no longer necessary.
Instead, can built systems that input an audio clip and directly output a transcript without needing to use hand-engineered representations.
this is contribute to the much larger data sets.

Data size:
Academic data sets on speech recognition might be as a 300/ 3000 hours which is considered reasonable size.
for commercial systems, are now trains on over 10,000 hours and sometimes over a 100,000 hours of audio.
And it's really still moving to a much larger audio data sets-transcribe audio data sets both x and y, together with deep learning algorithm, has driven a lot of progress in speech recognition.

3. Speech recognition model

3.1 Attention model

encoder: like BRNN
Input: on the horizontal axis, take in different time frames of the audio input,

decoder: attention model
Have an attention model try to output the transcript like, "the quick brown fox"

3. Speech recognition model
3.2: CTC cost model for speech recognition

CTC: Connectionist Temporal Classification
due to Alex Graves, Santiago Fernandes, Faustino Gomez, and Jürgen Schmidhuber.

>1. Idea: use a neural network structed with an equal number of input and output sentence
example:
Input: "the quick brown fox",

>2. Neural network: structured with an equal number of input x and output y.
Note: in this example drawn a simple of uni-directional RNN, but in practice, this will usually be a bidirectional LSTM and bidirectional GRU and usually, a deeper model.

>3. Speech recognation property:
>>>1. Number of input time steps here is very large ;
>>>2. in speech recognition, usually the number of input time steps is much bigger than the number of output time steps.

e.g: have 10 seconds of audio and your features come at a 100 hertz so 100 samples per second, then a 10 second audio clip would end up with a 1000 inputs. But output might not have a thousand characters.

>4. CTC cost function: allows the RNN to generate an output like this repeated charater: ttt, and special character "- blank character" , and space.
e.g: output 'ttt---h---eee---space----qqq------- ", And, this is considered a correct output for the first parts of "the space q"

3. Speech recognition model
3.2: CTC cost model for speech recognitio

>5. Basic rule for the CTC cost function: to collapse (折叠) repeated characters not separated by "blank ".
e.g: model output 'ttt---h---eee---space----qqq------- " by collapsing repeated characters, not separated by blank, it actually collapse the sequence into "the s'pace q",

and this allows network to have a thousand outputs by repeating characters a lot of times.
So, inserting a bunch of blank characters and still ends up with a much shorter output text transcript.
e.g: input sentence "the quick brown fox" including spaces actually has 19 characters, and if somehow, the newer network is forced to output a thousand characters by allowing the network to insert blanks and repeated characters , the output could still represent this 19 character with this 1000 outputs y

>6. Right bottom paper by Alex Grace, as well as by those deep speech recognition system, which I was involved in, used this idea to build effective Speech recognition systems.

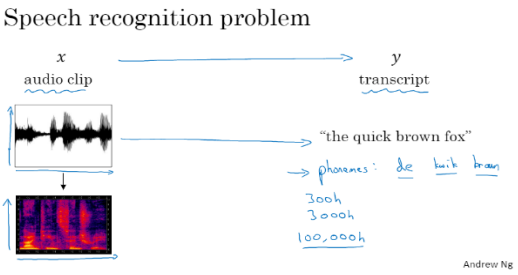
(note: what is cost function for attention model /CTC, predict output is words' one-hot vector, labeled output is word one-hot vector?-use logistic regression for each element of one-hot vector? l = sum y_i log (y^_i) - (1-y_i) * log (1-y^_i) over i -word in vocab; or just care the labeld 1 position: sum y_i log (y^_i) over i -word in vocab ??)-->should be softmax function output vocab size vector.

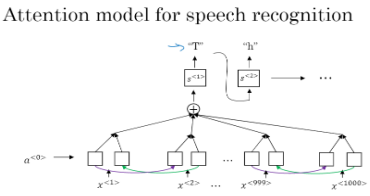
1.2 Training set

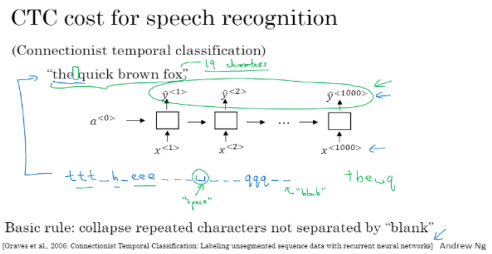
>3. Pre-process: audio clip:
Principle:
Even the human ear doesn't process raw wave forms, but the human ear has physical structures that measures the amounts of intensity of different frequencies, there is, a common pre-processing step for audio data .

Pre-processing step for audio data :-false back outputs (伪空白输出)
Run raw audio clip and generate a spectrogram(声谱图) -right colored pic.:
the horizontal axis is time, and the vertical axis is frequencies, and intensity of different colors shows the amount of energy:
-->reflect: how loud is the sound at different frequencies and at different times

thsi process, also called false back outputs, is often commonly applied pre-processing step before audio is pass into in the running algorithm. And the human ear does a computation pretty similar to this pre-processing step.







4. Summary:

This video shows a rough sense of how speech recognition models work. Attention like models work and CTC models work and present two different options of how to go about building these systems.

Now, today, building effective, production skills speech recognition system is a pretty significant effort and requires a very large data set. But, what I like to do in the next video is share you, how you can build a trigger word detection system, where keyword detection system which is actually much easier and can be done with even a smaller or more reasonable amount of data.

5.3-10: Trigger word detection_Audio data

You've now learned so much about deep learning and sequence models that we can actually describe a trigger word system quite simply just on one slide, as you see in this video.

1. Trigger word detection:

Definiation:
trigger word detection systems: with the rise of speech recognition, t there have been more and more devices you can **wake up with your voice** and those are sometimes called trigger word detection systems

Examples: of trigger word systems:
Amazon Echo: s woken up with the word "Alexa"
Baidu DuerOS : woken up with the phrase"xiaodunihao"
Apple Siri : working up with "hey, Siri"
Google Home: woken up with "okay, Google"

Apllicaiton:
e.g1: an Amazon Echo in your living room, you can walk in your living room and just say, Alexa, what time is it.
e.g 2: if you can build a trigger word detection system, maybe you can make your computer do something by telling it, computer, activate.
e.g3: One of my friends also worked on turning on and off a particular lamp using a trigger word kind of as a fun project.

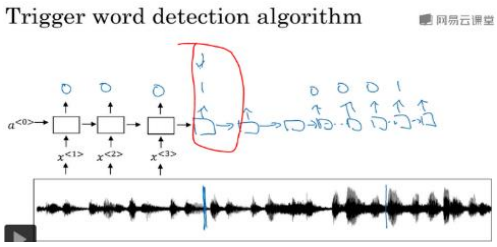
2. Trigger word detection algorithm

The literature on trigger detection algorithm is still evolving, so there isn't wide consensus yet on what's the best algorithm for trigger word detection.

Examples of an algorithm can use:
2.1: Algorithm : setting target label to 1 only for trigger word (many to many architecture-Tx=Ty, standard RNN)

>1. Process:
>>>1. Inpute sentece pre-processing: computed spectrogram features of input audio clip: x<1>, ...x<Tx>
>>>2. Pass audio spectrogram feature x<1>, ...x<Tx> thgough an RNN. (attention model or CTC,)

>>>3. Define the target labels: 1 for triger word, 0 for rest.
Idea: if some point in the input audio clip is when someone just finished saying the trigger word, then in the training sets:
>1. set the target labels to be 0 for everything before that point,
>2. right after that to set the target label of 1.
>3. And then if a little bit later on, the trigger word was said again at other time/ point, again set the target label to be 1 right after that.

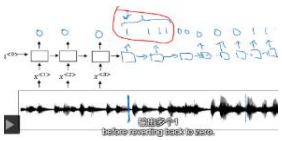


>2. One slight disadvantage:
Model creates a very imbalanced training set to have a lot more 0 than 1. **(consequences?)--> cost function: if output just all is 0, still get high precision.**

(note: labled y for one exmaple: skewed: many 0, small1: could not use accuray to measure algorithm performance in dev, instead use F1, or others.
training example: one example:
input x: x<1>, ..x<Tx> (there are negative words and positive words- very less positive words)
labeled output: y=[0,...0, 1, 1,1..., 0, 0, ..., 0_Ty] (for one positive word, output certain number like 50 of 1)
predict output y^(sigmoid): 0/1 for each y^<t>---y^<Ty>.

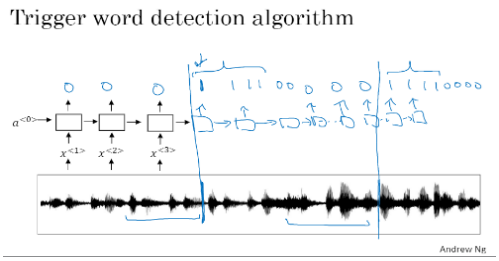
for one example, the prediciton output is Ty: why only has like 50 x positive word number of 1, result is 0-->accuracy on this example could not represent algorighm performance.
e.g: just set prediction y^=0-->if only have one positive word in example, then accuracy is =(Ty-50)/Ty * 100, still high
(multi examples on dev set, -->average perfromance over all dev set examples)
)

Examples of an algorithm can use:
2.2: Algorithm : setting target label to multi 1 for trigger word



Idea:
Make above model a little bit easier to train: instead of setting only a single time step output 1, make it output a few 1 for several times or for a fixed period of time before reverting back to 0.

So that slightly evens out the ratio of 1 to 0, but this is a little bit of a hack.



3. Summarize:
In this course on sequence models, learned about our RNNs, including both GRUs and LSTMs, and then in the second week, learned a lot about word embeddings and how to learn representations of words. And then in this week, you learned about the attention model as well as how to use it to process audio data.

Summary

1. S2S model: many-many (Tx /= Ty) RNN architecture: encoder + decoder
application on machine translation , image captioning, etc.

2. Machine translation model: vs language model
>Language model: predict what's the next word given previsous words: p(t|y<1>, ...y<t-1>)
>>> standard language model: inpute x<t> = labeled y<t-1> (except x<1>=0, a<0>=0 vector)
>>>sampling standard lange model: inpute x<t> = sampled, from prediction softmax distribution y^<t-1>.
>>> starting off activaton a<0> =0

>Machine translation model: -->pick the maxim. translation maxP(y|x)
>>>architecture: many-many , encoder +decoder
>>> encoder: -->inpute sentence-->output representation of input sentence as starting off for decoder: context c
>>> decoder: similar with language model, only a<0> /=0, but the representation of input, generated by encoder.-->also called conditional langauge model.;
decoder input: = sampled previous step output y<t-1> + activaiton of previous step: a<t-1>

3. Manchine translation: choose best translation: max p(y|x)
Method:
>1. greedy-->only pick max p(y<1>, ...y<t> |x) at each time step-->could not gurantee get output y^ is the max p(y<1>, ...y<t>,...y<Ty |x)
>2. Beam search: could not gurantte find the best translation, but much better
Fine tunning beam search:
- Log: for numerical underfloor;
- reduce penalty for long output: by divide Ty^a for log p.

4. Error analysis for machine translation: RNN model (encoder + decoder) or beam search
Method:
- Take all the errors in dev/test set, where labeled output y* is provided by human.
- Get p(y*|x) vs p(y|x) through 'decoder'
- Compare: p(y*|x) vs p(y|x)
-Summarize error fractor , caused by RNN and beam search-->get direction to work on

5. Belue score: evaluation translation
Principle: check similarity /overlap with human translation
>basic precsion for isolation word: for each outptuword, credit 1/0 if occur in ref.-->sum credit/ Ty
>modified precison for isolation word: p1:cout clip(ref)/cout (per se output)
> modified precison for n-gram word: pn
> Final belue scoare: = BP * EXP (average of p1-pn)
BP: penalty for short output: =1 if output length > reference ,
otherwise <1: = exp (1- referance length / output length)

6. Attention model: for very long sentence translation

S2S model, input all sentence (encoder) , then do the translation one word at a time.
when input sentence is very long, hard to memorize engough info. of input sentence -->Belue score reduce

Attention model: when output one word at a time, only look at certain input words
architecture: (more context C to memorize input sentence info.-hidden layer info. of encoder)
a littel like deep model:
> first layer: BRNN: in each step, output feature a<t'>of input word, instead of y<t'>; as input x<t'> to second laryer-context layer
> second layer-context layer: input: come from inpute word feature + output/decoder previous step activation s<t-1>:
--> assign weight to different inpute feature , get context C =sum , a<t,t'> * a<t'>over all word t' in input sentence
>>>> weight a<t,t'> , use softmax to make sure sum over all input word t is 1 = exp e<t,t'>/ sum (exp e<t,t'> all over inpute word)
>>> e<t,t'> : function of output previous step status s<t-1>, and feature of t'-th input word a<t'>-use small neural network, train together in whole neural network.

>third layer-un-bidirectional RNN for output (only forward)
inpute : previous step output word y^<t-1> + c<t>
output: activagtion s<t>, output word y^<t>.

(note: context layer c<1>, ...c<t> help to remember input sentence info., yet still need to read all input sentence, then go to next layer/step/before making prediction output)

7. Speech recognation:
output: is different from machine translation:
machine translation: output choose best during multi translations : max P(y|x)-BLEU used to estimate output
speech recognition: recognize the correct character/word: output is right or not.

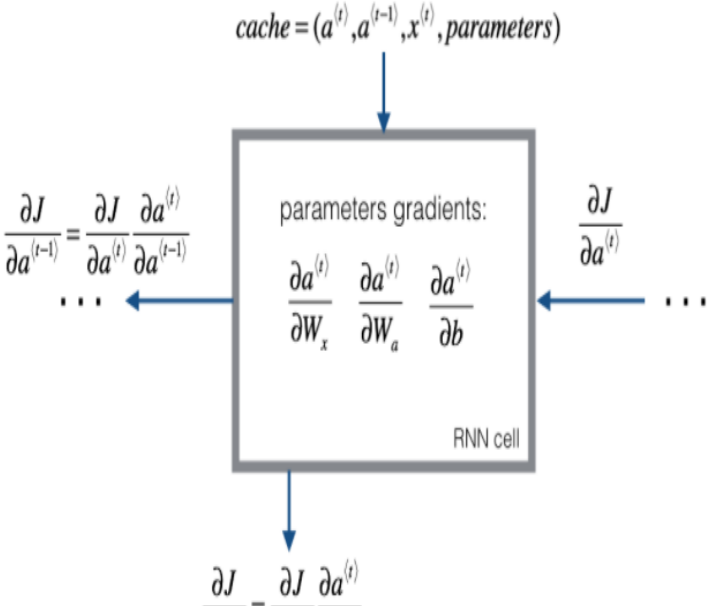
Property of speech recognition problem: Tx >>>Ty

mode: 1. attention model
2. CTC model: output is character, input time step is very long
Tx=Ty, then collapse Ty

8. Trigger word: many-many sequence architecture:
model: output single 1 for triger word/ output multi 1 for triger word (even output 1 and 0)

3.1 - Basic RNN backward pass

We will start by computing the backward pass for the basic RNN-cell.



$$a^{\langle t \rangle} = \tanh(W_{ax} x^{\langle t \rangle} + W_{aa} a^{\langle t-1 \rangle} + b)$$

$$\frac{\partial \tanh(x)}{\partial x} = 1 - \tanh(x)^2$$

$$\frac{\partial a^{\langle t \rangle}}{\partial W_{ax}} = (1 - \tanh(W_{ax} x^{\langle t \rangle} + W_{aa} a^{\langle t-1 \rangle} + b)^2) x^{\langle t \rangle T}$$

$$\frac{\partial a^{\langle t \rangle}}{\partial W_{aa}} = (1 - \tanh(W_{ax} x^{\langle t \rangle} + W_{aa} a^{\langle t-1 \rangle} + b)^2) a^{\langle t-1 \rangle T}$$

$$\frac{\partial a^{\langle t \rangle}}{\partial b} = \sum_{batch} (1 - \tanh(W_{ax} x^{\langle t \rangle} + W_{aa} a^{\langle t-1 \rangle} + b)^2)$$

$$\frac{\partial a^{\langle t \rangle}}{\partial x^{\langle t \rangle}} = W_{ax}{}^T . (1 - \tanh(W_{ax} x^{\langle t \rangle} + W_{aa} a^{\langle t-1 \rangle} + b)^2)$$

$$\frac{\partial x^{\langle t \rangle}}{\partial x^{\langle t \rangle}} = \frac{\partial a^{\langle t \rangle}}{\partial a^{\langle t \rangle}} \frac{\partial x^{\langle t \rangle}}{\partial x^{\langle t \rangle}}$$

$$\frac{\partial a^{\langle t \rangle}}{\partial a^{\langle t-1 \rangle}} = W_{aa}{}^{T} .(1 - \tanh(W_{ax} x^{\langle t-1 \rangle} + W_{aa} a^{\langle t-1 \rangle} + b)^2)$$

Figure 5: RNN-cell's backward pass. Just like in a fully-connected neural network, the derivative of the cost function J backpropagates through the RNN by following the chain-rule from calculas. The chain-rule is also used to calculate $(\frac{\partial J}{\partial W_{ax}}, \frac{\partial J}{\partial W_{aa}}, \frac{\partial J}{\partial b})$ to update the parameters (W_{ax}, W_{aa}, b_a).

Deriving the one step backward functions:

To compute the `rnn_cell_backward` you need to compute the following equations. It is a good exercise to derive them by hand.

The derivative of $\tanh$ is $1 - \tanh(x)^2$. You can find the complete proof here. Note that: $\text{sech}(x)^2 = 1 - \tanh(x)^2$

Similarly for $\frac{\partial a^{\langle t \rangle}}{\partial W_{ax}}, \frac{\partial a^{\langle t \rangle}}{\partial W_{aa}}, \frac{\partial a^{\langle t \rangle}}{\partial b}$, the derivative of $\tanh(u)$ is $(1 - \tanh(u)^2)du$.

The final two equations also follow same rule and are derived using the $\tanh$ derivative. Note that the arrangement is done in a way to get the same dimensions to match.

Backward pass through the RNN

Computing the gradients of the cost with respect to $a^{\langle t \rangle}$ at every time-step t is useful because it is what helps the gradient backpropagate to the previous RNN-cell. To do so, you need to iterate through all the time steps starting at the end, and at each step, you increment the overall db_a, dW_{aa}, dW_{ax} and you store dx.

Instructions:

Implement the `rnn_backward` function. Initialize the return variables with zeros first and then loop through all the time steps while calling the `rnn_cell_backward` at each time timestep, update the other variables accordingly.

```
# Increment global derivatives w.r.t parameters by adding their derivative at time-step t (≈4 lines)
dx[:, :, t] = dxt
dWax += dWaxt
dWaa += dWaat
dba += dbat
```

3.2 - LSTM backward pass

3.2.1 One Step backward

The LSTM backward pass is slighltly more complicated than the forward one. We have provided you with all the equations for the LSTM backward pass below. (If you enjoy calculus exercises feel free to try deriving these from scratch yourself.)

3.2.2 gate derivatives

$$d\Gamma_o^{\langle t \rangle} = da_{next} * \tanh(c_{next}) * \Gamma_o^{\langle t \rangle} * (1 - \Gamma_o^{\langle t \rangle}) \tag{7}$$

$$d\tilde{c}^{\langle t \rangle} = dc_{next} * \Gamma_u^{\langle t \rangle} + \Gamma_o^{\langle t \rangle}(1 - \tanh(c_{next})^2) * i_t * da_{next} * \tilde{c}^{\langle t \rangle} * (1 - \tanh(\tilde{c})^2) \tag{8}$$

$$d\Gamma_u^{\langle t \rangle} = dc_{next} * \tilde{c}^{\langle t \rangle} + \Gamma_o^{\langle t \rangle}(1 - \tanh(c_{next})^2) * \tilde{c}^{\langle t \rangle} * da_{next} * \Gamma_u^{\langle t \rangle} * (1 - \Gamma_u^{\langle t \rangle}) \tag{9}$$

$$d\Gamma_f^{\langle t \rangle} = dc_{next} * \tilde{c}_{prev} + \Gamma_o^{\langle t \rangle}\left(1 - \tanh(c_{next})^2\right) * c_{prev} * da_{next} * \Gamma_f^{\langle t \rangle} * \left(1 - \Gamma_f^{\langle t \rangle}\right) \tag{10}$$

3.2.3 parameter derivatives

$$dW_f = d\Gamma_f^{\langle t \rangle} * \begin{pmatrix} a_{prev} \\ x_t \end{pmatrix}^T \tag{11}$$

$$dW_u = d\Gamma_u^{\langle t \rangle} * \begin{pmatrix} a_{prev} \\ x_t \end{pmatrix}^T \tag{12}$$

$$dW_c = d\tilde{c}^{\langle t \rangle} * \begin{pmatrix} a_{prev} \\ x_t \end{pmatrix}^{\cdot} \tag{13}$$

$$dW_o = d\Gamma_o^{\langle t \rangle} * \begin{pmatrix} a_{prev} \\ x_t \end{pmatrix}^{T} \tag{14}$$

To calculate db_f, db_u, db_c, db_o you just need to sum across the horizontal (axis= 1) axis on $d\Gamma_f^{\langle t \rangle}, d\Gamma_u^{\langle t \rangle}, d\tilde{c}^{\langle t \rangle}, d\Gamma_o^{\langle t \rangle}$ respectively. Note that you should have the `keep_dims = True` option.

Finally, you will compute the derivative with respect to the previous hidden state, previous memory state, and input.

$$da_{prev} = W_f^T * d\Gamma_f^{\langle t \rangle} + W_u^T * d\Gamma_u^{\langle t \rangle} + W_c^T * d\tilde{c}^{\langle t \rangle} + W_o^T * d\Gamma_o^{\langle t \rangle} \tag{15}$$

Here, the weights for equations 13 are the first n_a, (i.e. $W_f = W_f[:n_a, :]$ etc...)

$$dc_{prev} = dc_{next}\Gamma_f^{\langle t \rangle} + \Gamma_o^{\langle t \rangle} * \left(1 - \tanh(c_{next})^2\right) * \Gamma_f^{\langle t \rangle} * da_{next} \tag{16}$$

$$dx^{\langle t \rangle} = W_f^T * d\Gamma_f^{\langle t \rangle} + W_u^T * d\Gamma_u^{\langle t \rangle} + W_c^T * d\tilde{c}_t + W_o^T * d\Gamma_o^{\langle t \rangle} \tag{17}$$

where the weights for equation 15 are from n_a to the end, (i.e. $W_f = W_f[n_a :, :]$ etc...)

Exercise: Implement `lstm_cell_backward` by implementing equations $7 - 17$ below. Good luck! :)

3.3 Backward pass through the LSTM RNN

This part is very similar to the `rnn_backward` function you implemented above. You will first create variables of the same dimension as your return variables. You will then iterate over all the time steps starting from the end and call the one step function you implemented for LSTM at each iteration. You will then update the parameters by summing them individually. Finally return a dictionary with the new

gradients.

Instructions: Implement the `lstm_backward` function. Create a for loop starting from T_x and going backward. For each step call `lstm_cell_backward` and update the your old gradients by adding the new gradients to them. Note that `dxt` is not updated but is stored.

```
    # loop back over the whole sequence
    for t in reversed(range(T_x)):
        # Compute all gradients using lstm_cell_backward
        gradients = lstm_cell_backward(da[:, :, t] + da_prevt, dc_prevt, caches[t])
        # Store or add the gradient to the parameters' previous step's gradient
        dx[:,:,t] = gradients['dxt']
        dWf = dWf + gradients['dWf']
        dWi = dWi + gradients['dWi']
        dWc = dWc + gradients['dWc']
        dWo = dWo + gradients['dWo']
        dbf = dbf + gradients['dbf']
        dbi = dbi + gradients['dbi']
        dbc = dbc + gradients['dbc']
        dbo = dbo + gradients['dbo']
    # Set the first activation's gradient to the backpropagated gradient da_prev.
```

```
    # Set the first activation's gradient to the backpropagated gradient da_prev.
    da0 = gradients['da_prev']

    ### END CODE HERE ###
```