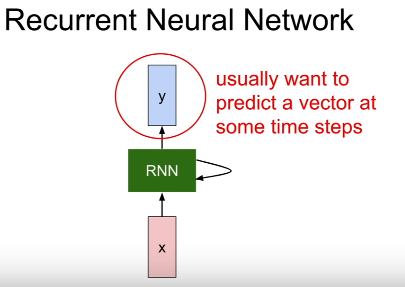
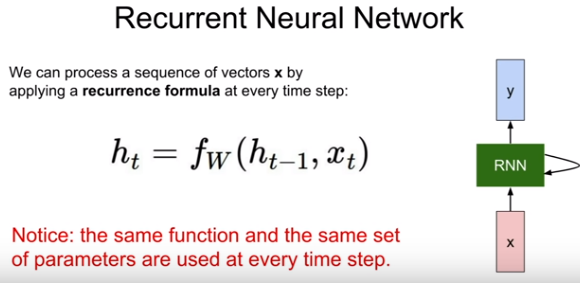
RNN

Batch normalization

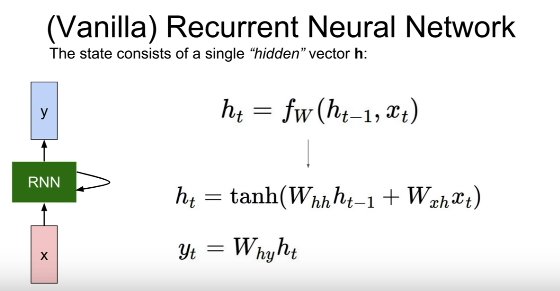
RNN

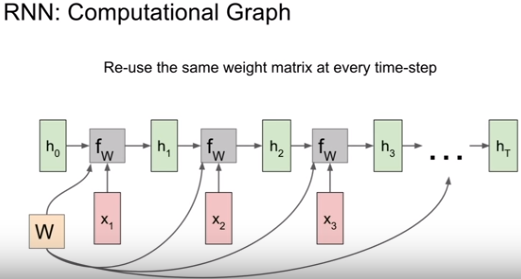


* RNN has some internal hidden state, and that internal hidden state will be updated every time that the RNN reads a new input;
* and that hidden state will be fed back to the model the next time it read an input
* recurrence formula inside the green box:

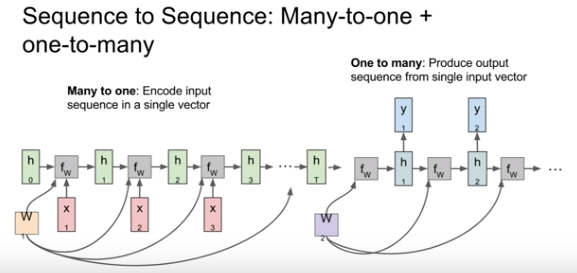


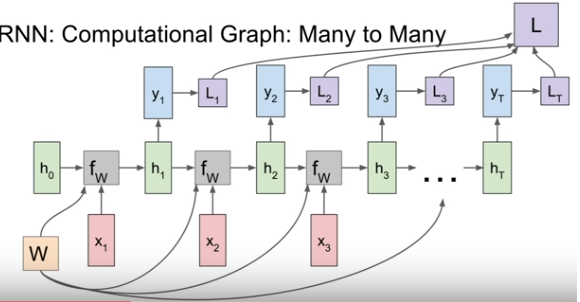
* This function f will depend on weights W, the previous hidden state(ht-1), and current state x as input. And will ouput the next(updated) hidden state
* with the next X(t+1) the new hidden state will be used.
* We might attach some additional FC layers that read this ht at every time step





How gradient flows in backpropagation,





**Sparse/Dense vector:**

* **Dense matrices store’s every entry in the matrix.**
* **Sparse** matrices only store’s the **nonzero** entries.

*the elements of a vector have mostly zero values. Such a vector is said to be sparse(seyrek)*

*It is inefficient to use a one-dimensional array to store a sparse vector.*

*You sometimes know that your vector will have a lot of ui=0 value. Then you may want, to avoid memory wasting, to store values that are not 0, and then, and consider, other values as zero.*

*And for exemple a dense vector (1, 2, 0, 0, 5, 0, 9, 0, 0) will be represented as {(0,1,4,6), (1, 2, 5, 9)}*

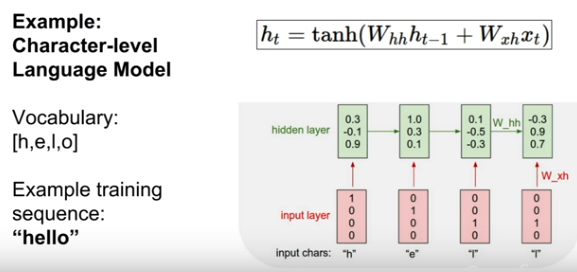
*Yellow ones for positions*

**Example**

**Language modelling**

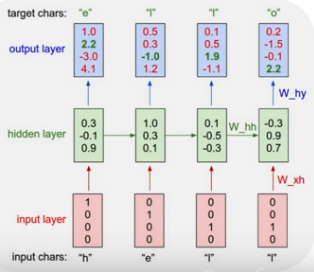
1 character level

2 word level

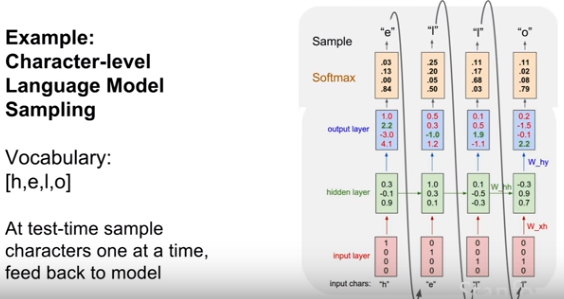


During forward pass,

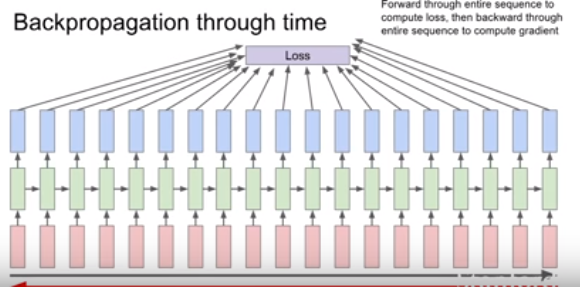
* At first time step: it will receive the input letter h; will go into 1st RNN cell, and then will produce the output yt(which letter is most likely , in this example model predict letter O ) we’ll use to softmax loss to quantify our unhappiness with these predictions
* In the second step, it has high loss because l has the lowest prob.
* After training with different sequences, the model will learn to predict true.



Test time



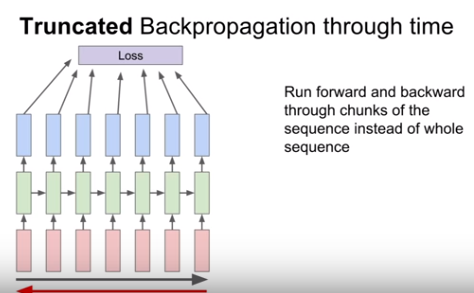
* At test time sample **characters one at a time** feed back to model; taking 1st output to second input and so on.
* we’ll use a softmax function convert those scores into a probabilty distribution
* and then we’ll **sample** from that prob distr. to actually synthesize the second letter in the seq
* **question**: why sampling rather than taking the largest scored char?
  + The prob distr we had(1st prob is 0.83 for char O which is wrong prediction) ,it was impossible to get the right char
  + In practice you’ll see both, sometimes you ll take argmax prob which will more stable, but one **advantage of sampling** it lets you get **diversity** of outputs from your models
* **Question:** at test timeCould we feed the whole softmax vector rather than 1hot vector?
  + Two problems with that,
    - 1: that’s very different from the data that it saw at training time; if you ask a model at test time that it didn’t trained for, it will blow
    - 2: If you train a model for word sequencing then you ll have tens of thousands of words ; This first operation that’s taking this one hot vector, is often performed using sparse vector operations rather than dense factors;
    - It would be computationally really bad if you want to have this load of 10K softmax vector

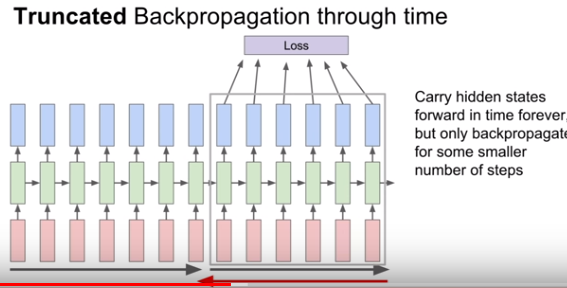


This could be problematic if the seq is very very long; for ex training a model for the entire text of wiki

Fwd pass thru the entire text of all wiki,then backwrd pass thru all, and finally making a single gradient update will be extremely slow, taking lots of memory and time and will not converge

In practice people use truncated backprob thru time





Question

Is this kind of making the Markov assumptions?

Not really, because we’re carrying this hidden state forward in time forever,

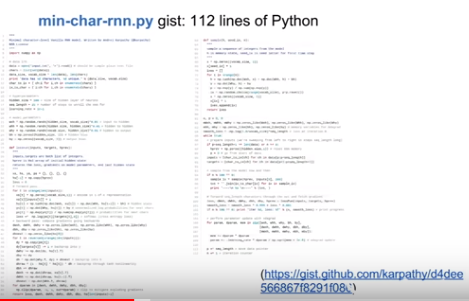
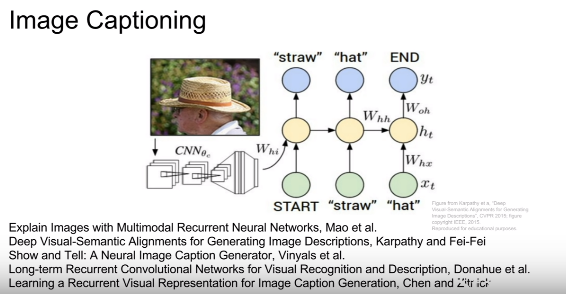
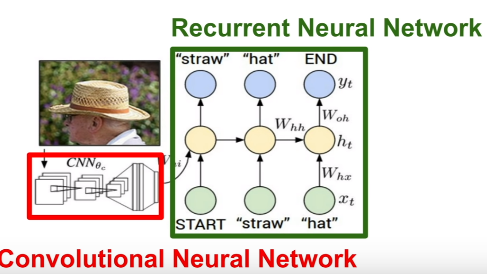
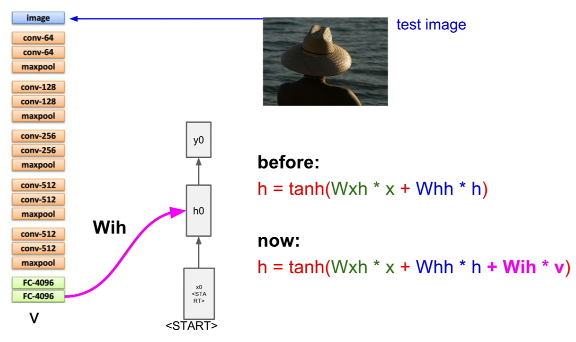


Image Captioning





* CNN will produce a summary vector of the image ; which will then feed into the first time step of RNN language models which will then produce words of the caption one at a time
* At test time

 -> 

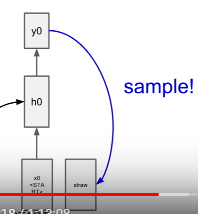
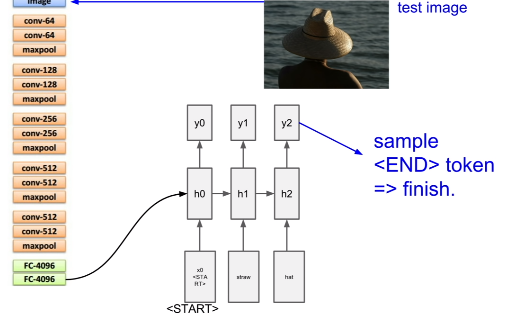
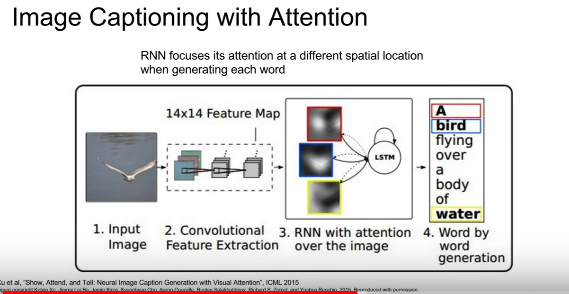
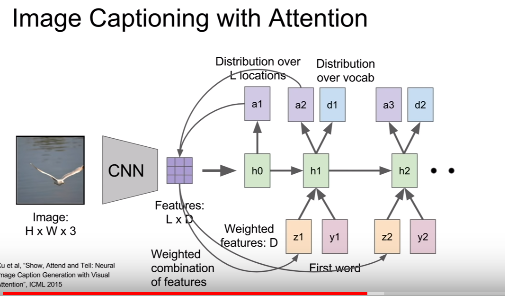
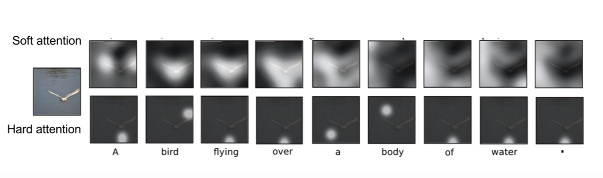
* Instead of softmax layer, we’ll use the FC4096 to summarize the whole content of the image.
* There is start token which enables the model to generate a word.
*  
* When end token is seen the model stops sampling.
* Supervised learning dataset -> Microsoft coco

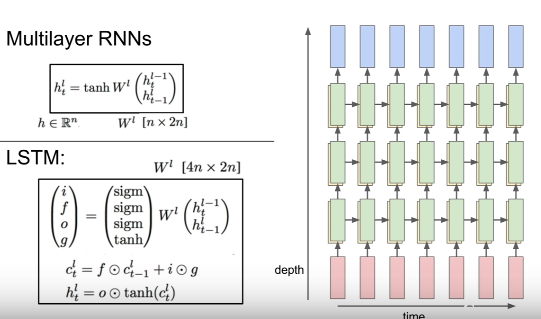
Image captioning with Attention

* 
* An advanced model

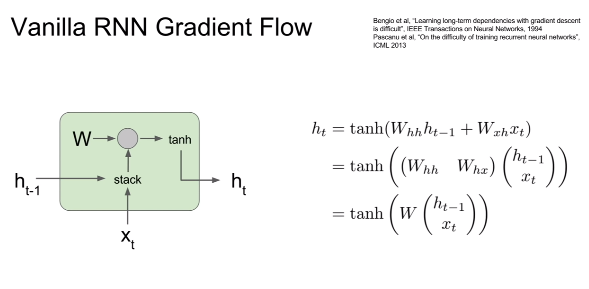


* CNN rather than producing a single vector summarizing the entire image, now it produces some grid of vectors that summarizes one vector for each spatial location
* In addition to sampling the vocabulary at every time step, it also produces a distribution over the locations in the image where it wants to look
* And now this distribution over image locations can be seen as a kind of a tension where the model should look during training; so that in the first hidden state computes this distr over image locs, which then goes back to the set of vectors to give a single summary vector that maybe focuses the attention on one part of the image
* The new hidden state will produce to outputs :
  + One is our distr. over vocabulary words
  + The other is a distr over image locs
* D t
* After training it will shift its attention around the image for every word that it generates in the caption
* Here you can see, it produced the caption a bird is flying over,

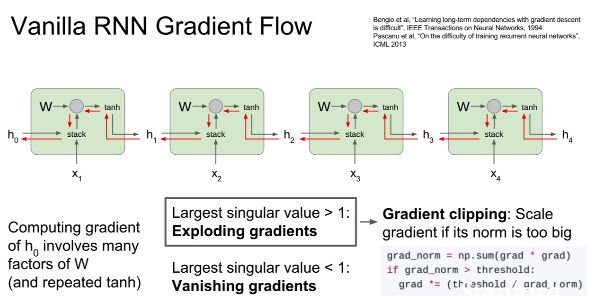
Multilayer RNNs



**You typically don’t see super deep models in RNNs. 2,3,4 is layer rnn is common,**

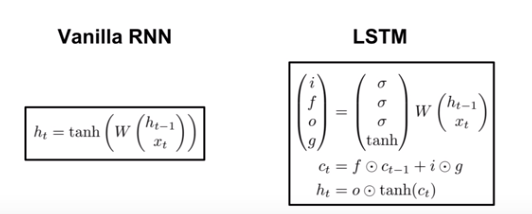


What happens in this architecture during the backward pass when we try to compute gradients?



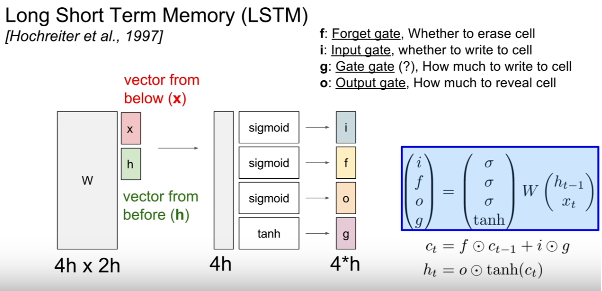


Changing rnn architecture means using lstm ,gru etc.

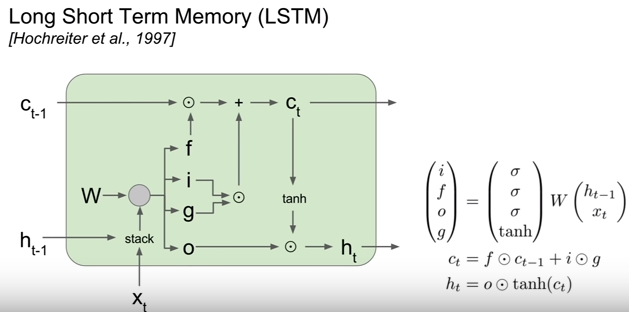


In vanilla there is only 1 hidden state, whereas lstm has 2 hidden state(ct-cell state and ht)

Ct is inside LSTM, doesnot exposed to outside world

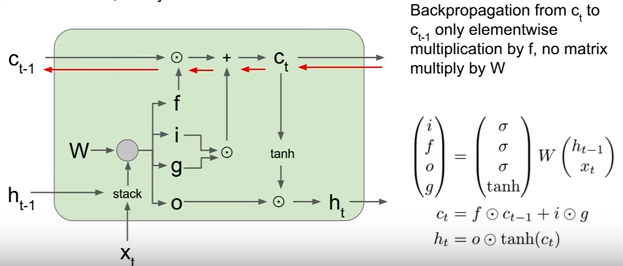


* In the vanilla rnn, we took two input vectors(x and h), concatenate them, then we did a matrix multiply to directly compute the next hidden state in the RNN
* LSTm does something a little bit different; we take the prev hidden state and x as input, stack them, Multiply by a very big weight matrix W, to compute 4 different gates which has the same size as the hidden state
* Only gate-gate uses tanh so the values in it will be between -1 & 1
* The other gates uses sigmoid that means 0 to 1
* After computing the gates, if you look at this next equation, you can see that our previous cell state is being multiplied elementwise by the forget gate
* You can think of this forget gate as being a vector of zeros and ones that telling us for each element in the cell state, do we want to forget that element of the cell in the case if the forget gate is zero?
* Or do we want to remember that element in the case of forget gate is 1



We saw in vanilla rnn during backward prop vanishing and exploiding gradient problems occur.

**What is happening to LSTM’s during backward pass?**



When we have our upstream gradient from the cell coming in, then once we backprop backwards through this addition operation, remember that this addition copies that upstream gradient into the two branches. So our upstream gradient gets copied directly and passed directly to backprop thru this elementwise multiply.

So then our upstream gradient ends up getting multiplied elementwise by the forget gate. As we backprop backwrds thru this cell state, the only thing that happens to our upstream cell state gradient is that it ends up getting multiplied element wise by the forget gate; so this is really a lot nicer than vanilla rnn for two reasons:

1. One is that this forget gate is now an **element wise multiplication rather than a full matrix** multiplication. So element wise multiplication is going to be a little bit nicer than full matrix multiplication
2. Second is it elementwise multiplication will potentially be multiplying by a different forget gate at every time step.

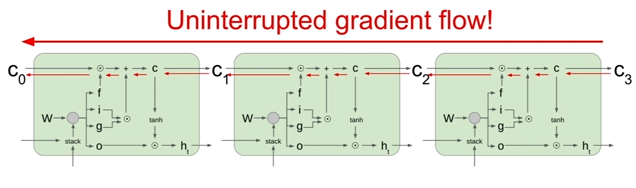
* So in vanilla rnn we were continually multiplying by that same weight matrix over and over again

which led to these exploding or vanishing gradients.

* But in LSTM; this forget gate can vary from each time step , now it is much easier for the model to avoid these problems (exploid or vanish)
* Finally, this forget gate is coming out from a **sigmoid** , this element wise multiply is **guaranteed to be between zero and one.**
* In vanilla during backprop our gradients were flowing thru tanh at every time step; but in lstm

Our hidden state is used to compute those outputs yt;

* If you imagine backprob from the final hidden state back to the first cell state. Then thru that backward path we only backprop thru a single tanh nonlinearity , rather thant thru separate tanh at every time step



**Question: what about gradient in respect to w?**

At every time step, we will take our current cell state, that will give us our local gradient on w, so because our cell state we’ll end up adding those first time step w gradients to compute our final gradient on w

Imagine we have a very long sequence

We’ll get a local gradient on w for each time step and that local gradient on w will be coming thru these gradients on c and h. So because we’re maintaining the gradients on c much more nicely in lstm, those local gradients on w at each time step will also be carried fwd and bckwrd

much more cleanly.

