Are you adding the emission scores for <START> and <STOP> tokens? I had the same issue and it was because of that.

Did you not add these to forward as well?

Any suggestions? Is there a way we should initialize self.transitionWeights besides making transitions to "START" and from "END" really negative? I've tried normal\_ initialization, and uniform() initialization (with bounds -sqrt(6/numRows+numCols), sqrt(6/numRows+numCols)) but this is still not enough to get me over the edge

had to fix my forward algorithm calculation (specifically, I was calculating it wrong at the first and last step

How did you initialize your alpha and did you skip over something for your last step? Sorry, I think this might be my issue too and I have been trying to figure this out for a while

Set the weights transitioning to start label and from end label to very small values since such transitions are not possible.

Has a tagging example

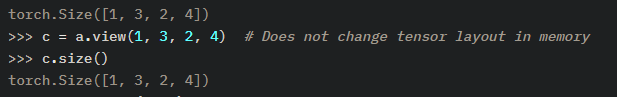
<https://pytorch.org/tutorials/beginner/nlp/sequence_models_tutorial.html>

a = a.view(4, 4)

VIEW IS EQUAL TO RESHAPE BUT WITHOUT A COPY

view(*\*shape*) → Tensor

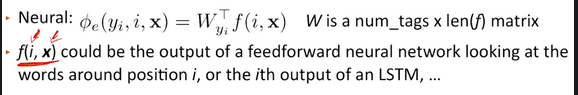
Returns a new tensor with the same data as the self tensor but of a different shape.



So for the project, are we using the features from lstm before it is log-softmaxed?

Yes the output of the Bi LSTM hidden state IS DIRECTLY the f(i,x) here

This is equivalent to a linear layer

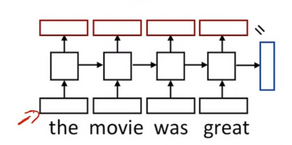


From class

BASIC LSTM

NLLLoss and CrossEntropyLoss are only slightly different NLLLoss Just directly does the negative log likelihood expecting that you are feeding it in log probs and thus are normalized. CrossEntropyLoss does the same thing but will softmax before doing NLLLoss so the inputs are unormalized scores

Obtaining log-probabilities in a neural network is easily achieved by adding a LogSoftmax layer in the last layer of your network. You may use CrossEntropyLoss instead, if you prefer not to add an extra layer.



Note the little equals so the out of the final state and the hidden state are the same

IMPORTANT

The prepare input method pads the inputs so they are of length 50 (batch size) by x where x is the word count of the longest sentence in the batch

So then for this the number of tags is the ten, 44 was the longest sentences number of words

And the 50 was just how many sentences were passed in (batch size)

torch.Size([50, 44])

torch.Size([50, 44])

out.shape torch.Size([50, 44, 500])

tag\_scores.shape torch.Size([50, 44, 10])

tag\_probs.shape torch.Size([50, 44, 10])

FOR LSTM AND WHY YOU HAD TO SET BATCH FIRST

* **batch\_first** – If True, then the input and output tensors are provided as (batch, seq, feature). Default: False
* I initialize my GloVe embeddings using with\_pretrained(Freeze=False)
* My LSTM has batch\_first=True, dropout=0.5, bidirectional=True

I was able to get it to work by formatting by swapping the length and class axes of the LSTM output (ie [5, 10, 32]) and using argmax to make Y 2 dimensional (ie [5, 32]) and get rid of the onehot encoding

50 is the batch size

32 is the max character length eg length of the longest word

10 is the hidden size

10 is the number of tags so a word that still has a score for every word will have 10 as its last dimension

[5, 32, 7]

(seq\_len, batch, num\_directions \* hidden\_size):

HOW TO DO THE MANUAL LOSS

<https://piazza.com/class/kj7vngax6ni7lt?cid=157>

ALAN RECOMMENDED THIS

<https://pytorch.org/docs/master/generated/torch.einsum.html>

<https://blog.floydhub.com/long-short-term-memory-from-zero-to-hero-with-pytorch/>

<https://www.deeplearningwizard.com/deep_learning/practical_pytorch/pytorch_lstm_neuralnetwork/>

<https://towardsdatascience.com/lstms-in-pytorch-528b0440244>

IMPORTANT HOW TO DO LOSS FOR BASIC LSTM

<https://piazza.com/class/kj7vngax6ni7lt?cid=219>

You can implement the loss by multiplying the negative log probabilities with onehot

*Y*. The CrossEntropyLoss should also give the same result but it takes *Y* as class indices instead of onehot format. As long as your *Y*'s are correctly sent to the CrossEntropyLoss it should give the same/similar loss to manually multiplying the log probabilities with onehot *Y*.

**dim** ([*int*](https://docs.python.org/3/library/functions.html#int)) – A dimension along which Softmax will be computed (so every slice along dim will sum to 1).

CHAR LSTM

Sizing for char representation layers

<https://piazza.com/class/kj7vngax6ni7lt?cid=251>

<https://piazza.com/class/kj7vngax6ni7lt?cid=141>

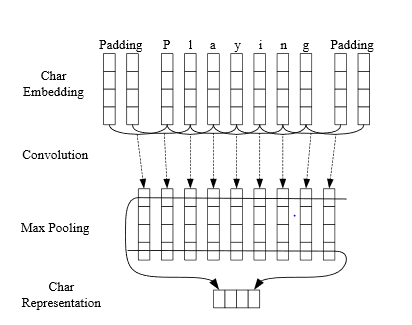
Character lstm output size

<https://piazza.com/class/kj7vngax6ni7lt?cid=141>

<https://piazza.com/class/kj7vngax6ni7lt?cid=234>

<https://piazza.com/class/kj7vngax6ni7lt?cid=220>

<https://piazza.com/class/kj7vngax6ni7lt?cid=219>



It looks like here they are doing the max pool across a row

<https://github.com/pytorch/pytorch/issues/17983#issuecomment-473018802>

I don't know if I'm mistaken, but what I did was first pass X\_char through the character embeddings to get a tensor of size (batch\_size, # words, # chars, char\_embedding\_size). After passing this through a nn.Conv1d (which required me to permute the dimensions and reshape, since nn.Conv1d expects a 3D tensor, and so I transformed that into a (batch\_size \* # words, char\_embedding\_size, # chars) sized tensor, i.e. the first dimension is the total number of words across all sentences and the last two dimensions are swapped) and doing torch.max(), I got a tensor of size (batch\_size, # words, # filters) (so NOT the character embedding size in the last dimension like in the instructor answer; I don't see how we would get that unless you just assume they are the same). I finally concatenated the word embeddings with that to get a (batch\_size, # words, word\_embedding\_size + # filters) sized tensor

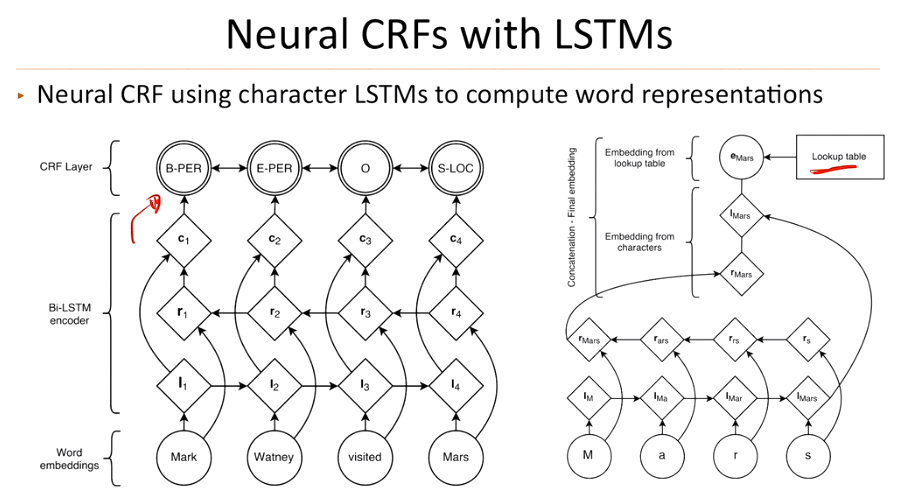
<https://piazza.com/class/kj7vngax6ni7lt?cid=61>

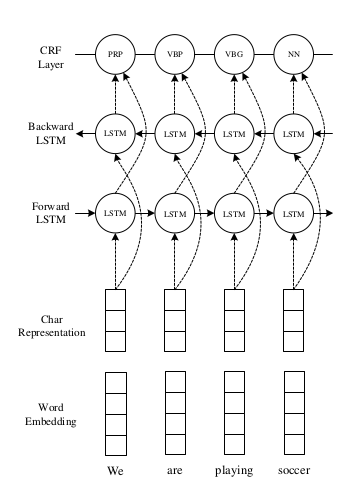
IMPORTANT

<https://piazza.com/class/kj7vngax6ni7lt?cid=141>

Your character CNN should give you an output of size (batch, max\_sentence\_length, CHAR\_EMB), while the word embedding layer would give you an output of size (batch, max\_sentence\_length, WORD\_EMB)

Then you should simply concatenate these two outputs across the last dimension such that the final combined output should look like (batch, max\_sentence\_length, CHAR\_EMB + WORD\_EMB)





CRF LSTM

WATCH FROM TIM 36:00 onward

<https://bluejeans.com/playback/s/DOFz5CBKwgTgPVhL97RqTtqzuPjn8mrOkFtNpq9ZBxdQWp0KhpMmi7UrDtEYRKTy>

CRF TRAINING LECTURE

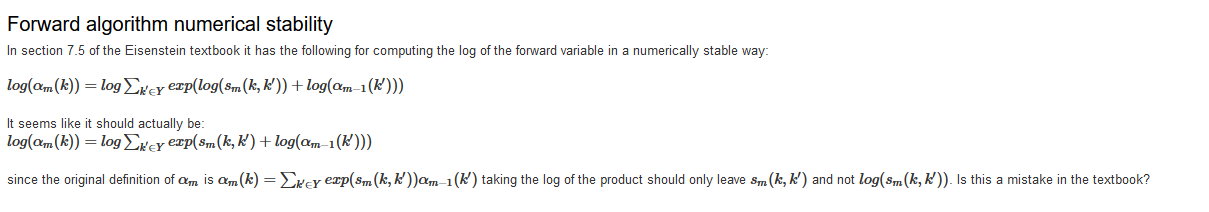
<https://bluejeans.com/playback/s/h2YqB5Vjmdo4UcIQC8Vnwt5OLvLAGX3zepKd5VxfDUdboFvbcBAyz7vQm2F8tnxd>

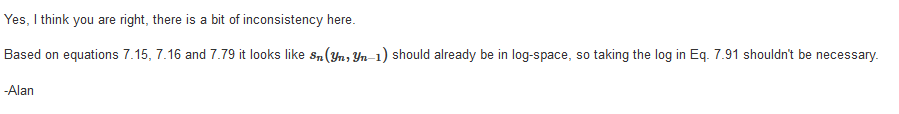
CRF FORWARD ALGO LECTURE

<https://bluejeans.com/playback/s/DOFz5CBKwgTgPVhL97RqTtqzuPjn8mrOkFtNpq9ZBxdQWp0KhpMmi7UrDtEYRKTy>

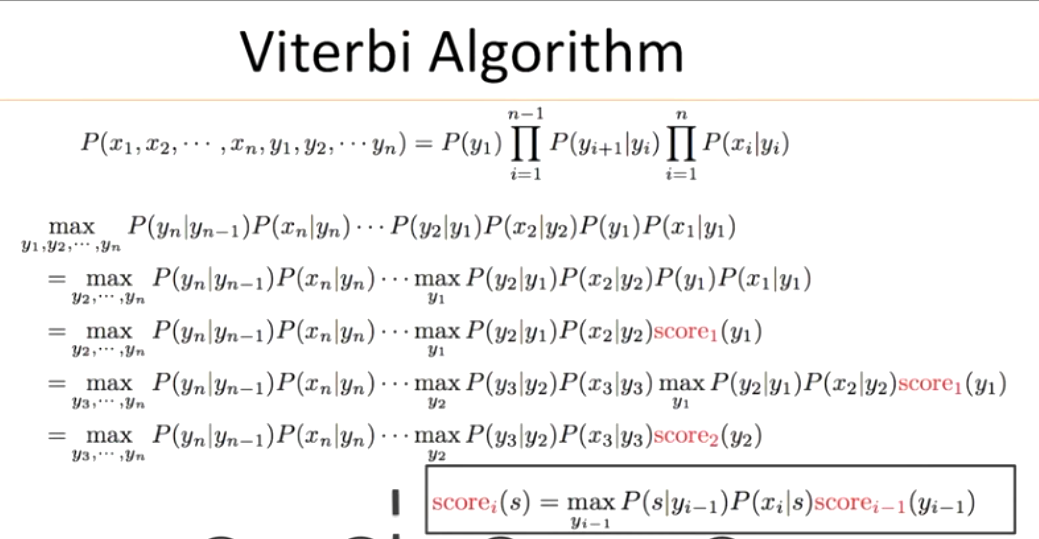
VITERBI ALGO LECTURE

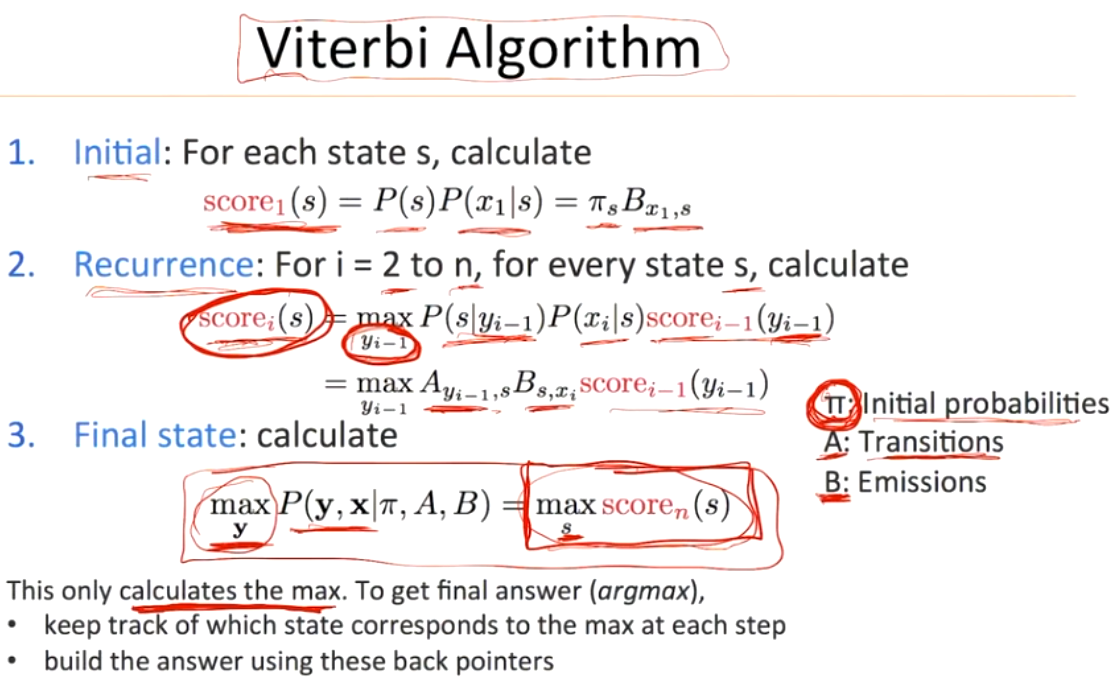
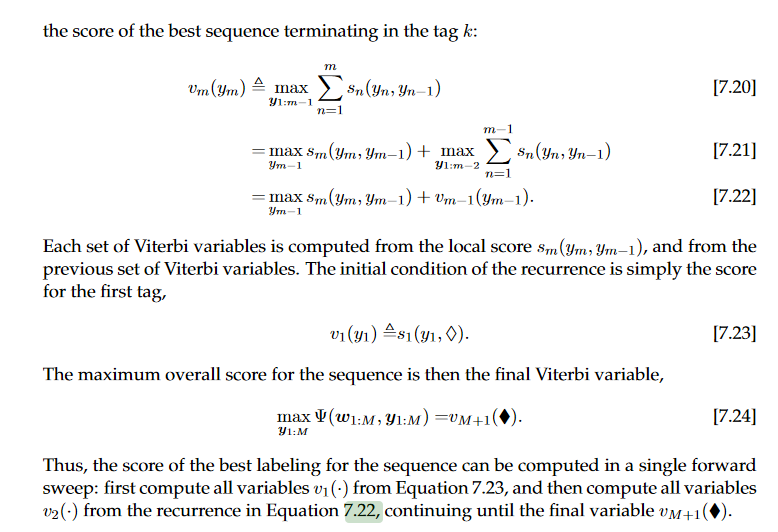
<https://bluejeans.com/playback/s/79GUvmqUDkFpvzeRSMQRkAOM0d0HRLpxyB3UpRB4HXpuMLxnFOcT4zij13bjQRxG>

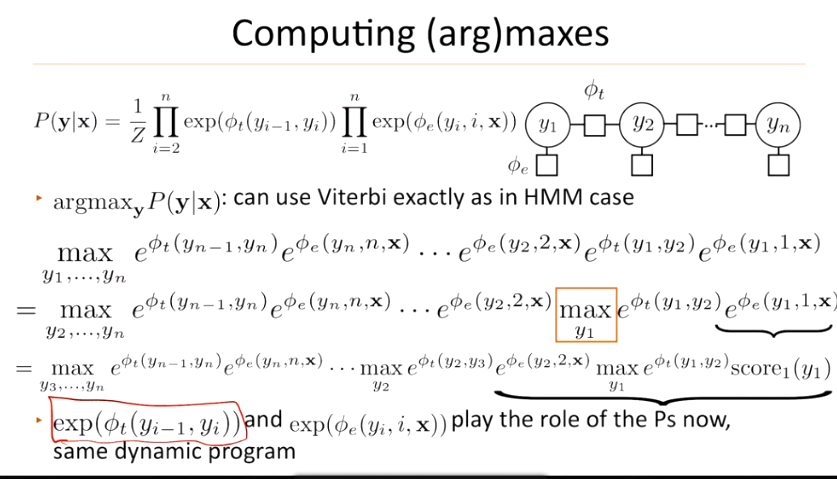
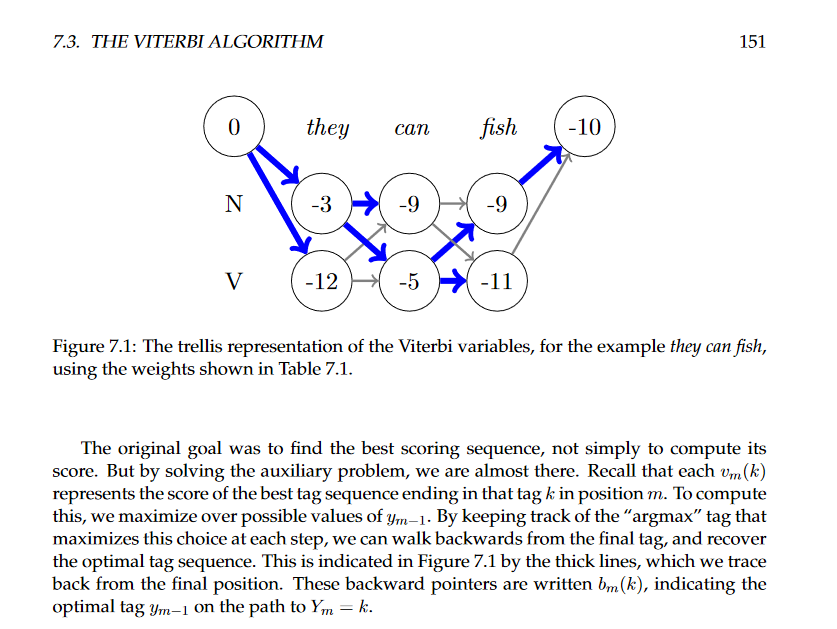




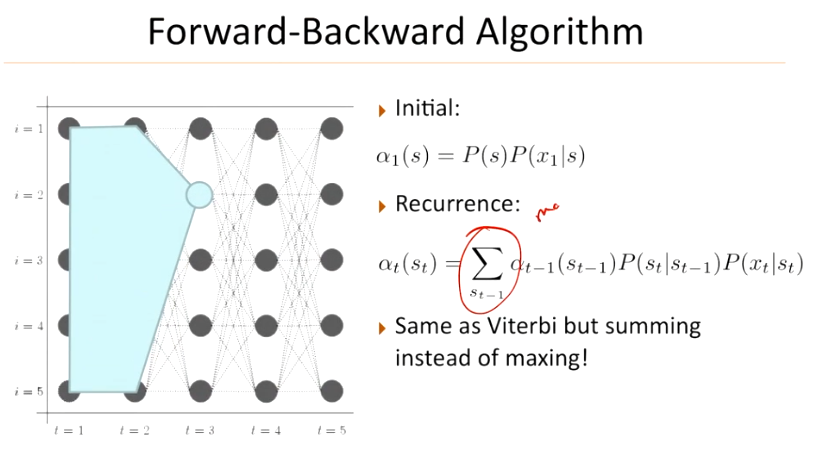
Viterbi is for inference and takes the max but otherwise is identical to the forward algo

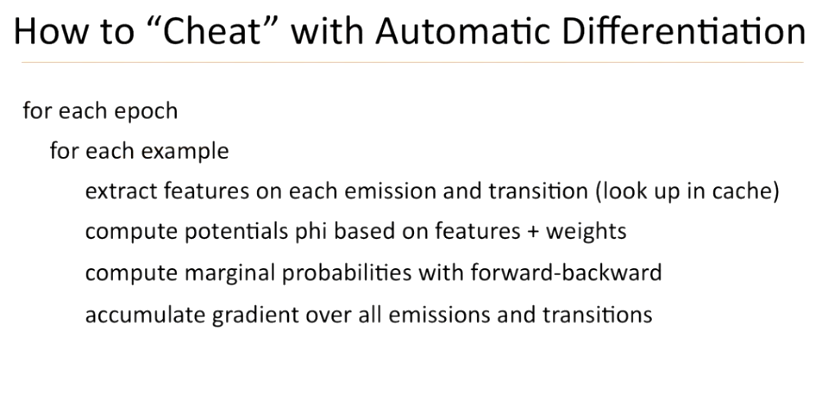






This is the forward algo same as the viterbi but sums instead of maxing (for crf these )





THIS IS WHAT WE WANT TO DO THE PHI E IS THE EMISSION PROB

PHI T IS THE GOLD SCORE OUTPUT

Explains what the gold score is

<https://piazza.com/class/kj7vngax6ni7lt?cid=171>

<https://piazza.com/class/kj7vngax6ni7lt?cid=152>

<https://piazza.com/class/kj7vngax6ni7lt?cid=254>

Section 7.3 has the viterbi algo

After calculating the lstm scores, you will need to make separate calls to the gold score and the forward algo function to calculate the conditional log likelihood. In the training loop, you just need to calculate the loss by negating the conditional log likelihood. You don't have to concern about decoding since it is already been taken cared of by the line lstm.write\_predictions(sentences\_dev, 'dev\_pred'

In CRFs, normalization is done globally over the entire sequence using the partition function, Z(X), unlike the basic LSTM where outputs are normalized probability distributions over a single label at each timestep (using softmax).

<https://pytorch.org/docs/stable/generated/torch.logsumexp.html>

The tutorial Sarai has been referencing

<https://pytorch.org/tutorials/beginner/nlp/advanced_tutorial.html>

It should be equal to the size of the vocab + any special tokens (see definition of word2i at the top of the file).

Yes, you need to create an embedding layer for characters. After getting the random embedding for each characters, you feed it into the dropout layer, CNN and other layers as in the figure in your question shows.

DIM\_HID is the dimension of the hidden layer

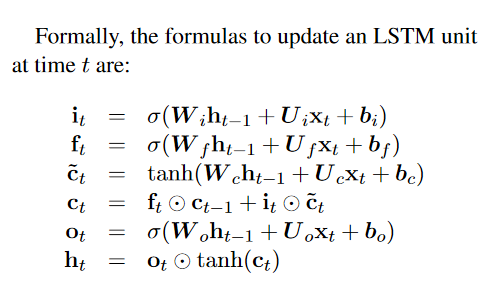
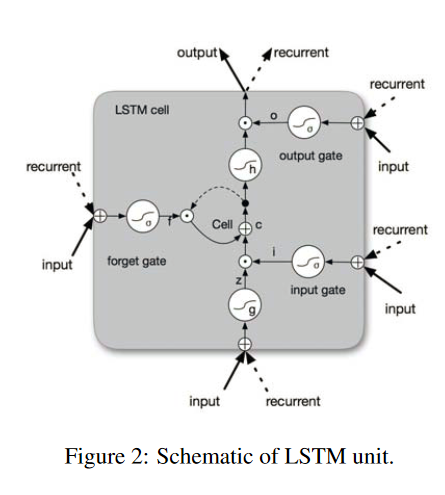
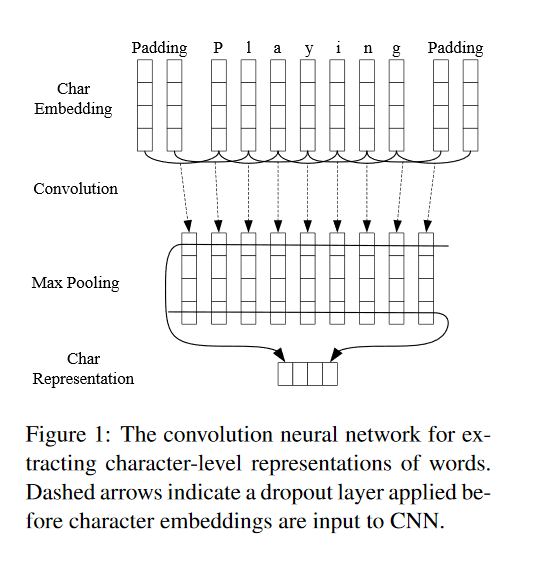
They use this in the given inference code to get the result

pred = self.forward(X).argmax(dim=2)

USE FUNCTIONS IN PADDING SECTION FOR INPUT PREP

Except that we use only character embeddings as the inputs to CNN, without character type features. A dropout layer (Srivastava etal., 2014) is applied before character embeddings are input to CNN

This description seems to contradict the diagram



C~\_t here is the z input which is not described well

