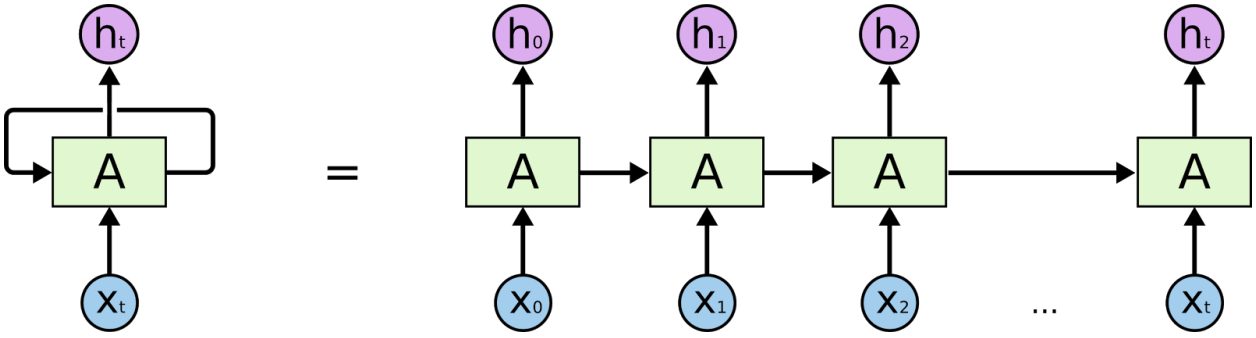
POS Tagging using Long Short Term Memory Networks

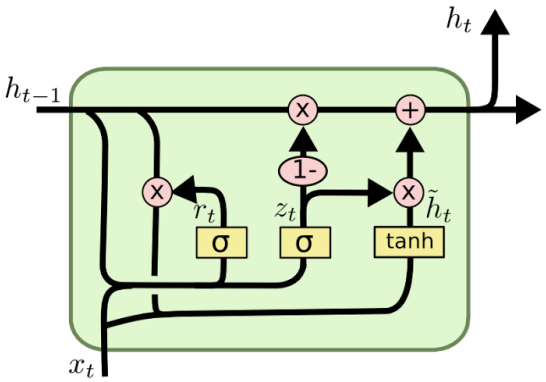
Methodology

Task:
The task is to provide a sequence of tags for a given sequence of words. This can be considered as a sequence modelling task.

LSTMs:
Sequences are modelled well using Recurrent Neural Networks specially Long Short Term Memory Networks (LSTMs).



During each time step in the input data, we provide the input data and the LSTM produces an output while simultaneously updating its internal hidden state. This is similar to the way Hidden Markov Models works except here we use neural networks to do the learning.
Roughly speaking, an LSTM can be represented by

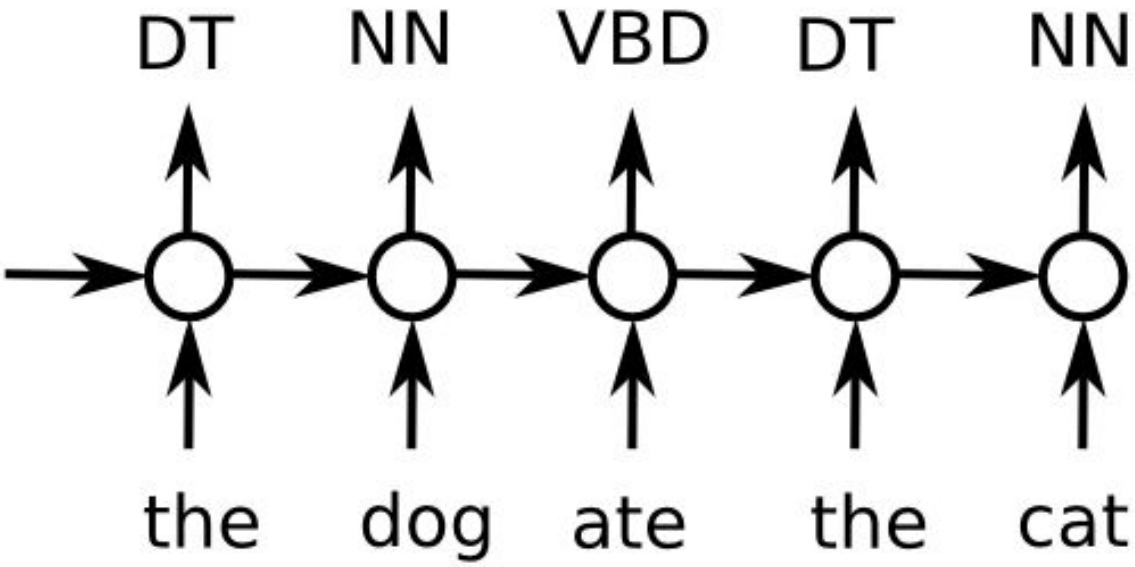


$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

In our case:

We will input a sentence word by word into the LSTM. We expect the LSTM to make a prediction for the tag of the word at each time step for each word.

.



Data Set:

We use the brown corpus [https://archive.org/details/BrownCorpus] to get the word sequences with their respective tags. In its raw form, there are 500 files with around 90 sentences in each file. The total number of tags are 472.

The corpus looks like this:

```
The/at Fulton/np-tl County/nn-tl Grand/jj-tl Jury/nn-tl said/vbd Friday/nr an/at investigation/nn of/
in Atlanta's/np$ recent/jj primary/nn election/nn produced/vbd ``/`` no/at evidence/nn ''/'' that/cs
any/dti irregularities/nns took/vbd place/nn ./.


The/at jury/nn further/rbr said/vbd in/in term-end/nn presentments/nns that/cs the/at City/nn-tl
Executive/jj-tl Committee/nn-tl ,/, which/wdt had/hvd over-all/jj charge/nn of/in the/at election/nn
,/, ``/`` deserves/vbz the/at praise/nn and/cc thanks/nns of/in the/at City/nn-tl of/in-tl Atlanta/
np-tl ''/'' for/in the/at manner/nn in/in which/wdt the/at election/nn was/bedz conducted/vbn ./.


The/at September-October/np term/nn jury/nn had/hvd been/ben charged/vbn by/in Fulton/np-tl Superior/
jj-tl Court/nn-tl Judge/nn-tl Durwood/np Pye/np to/to investigate/vb reports/nns of/in possible/jj
``/`` irregularities/nns ''/'' in/in the/at hard-fought/jj primary/nn which/wdt was/bedz won/vbn by/in
Mayor-nominate/nn-tl Ivan/np Allen/np Jr./np ./.


``/`` Only/rb a/at relative/jj handful/nn of/in such/jj reports/nns was/bedz received/vbn ''/'' ,/, the
/at jury/nn said/vbd ,/, ``/`` considering/in the/at widespread/jj interest/nn in/in the/at election/
nn ,/, the/at number/nn of/in voters/nns and/cc the/at size/nn of/in this/dt city/nn ''/'' ./.


The/at jury/nn said/vbd it/pps did/dod find/vb that/cs many/ap of/in Georgia's/np$ registration/nn and/
cc election/nn laws/nns ``/`` are/ber outmoded/jj or/cc inadequate/jj and/cc often/rb ambiguous/jj
''/'' ./.


It/pps recommended/vbd that/cs Fulton/np legislators/nns act/vb ``/`` to/to have/hv these/dts laws/nns
studied/vbn and/cc revised/vbn to/in the/at end/nn of/in modernizing/vbg and/cc improving/vbg them/ppo
```

The steps involved are:

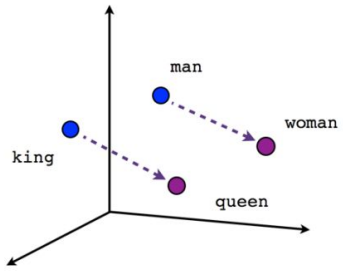
1. Use Glove Vectors as the Knowledge Base.

Each word can be represented as a vector. Words when used in their atomic form gave no syntactic or semantic meaning. Defining a word as a vector gives us more information regarding the word. These vectors carry semantic and syntactic information about the words in a dense and continuous form.

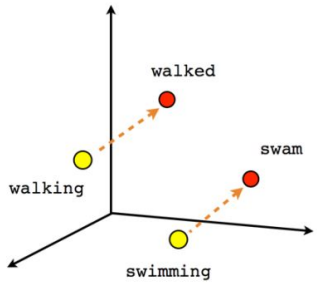
The Glove Vectors are generated by creating a co occurrence matrix of all the words in the corpus followed by applying Singular Value Decomposition to get a dense N dimensional representation of each word. So, each word can be represented by an N dimensional vector.

```
'cat' : [ 26.2  67.3 74.1]
'dog' : [ 96.3 21.7 48.3]
```

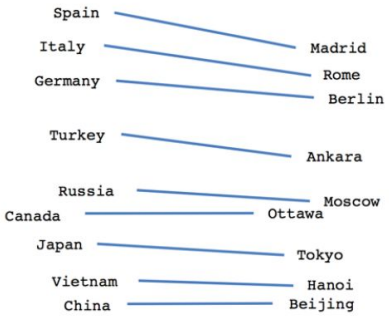
These vectors have the property that they capture linguistic regularities in the language very well.



Male-Female



Verb tense



Country-Capital

This allows us to embed our system with a kind of Knowledge Base about the words. It is a lot better than representing words using unique integer identifiers. Using word embeddings to give each word a vector gives us syntactic and semantic information about the words which in turn can be used for various other tasks. Words which are similar to each other are mapped closer to each other. In the above examples -

Semantic meanings = `vector(king) - vector(queen) + vector(man) = vector(woman)`

Syntactic meanings = `vector(walking) - vector(walked) + vector(swimming) = vector(swam)`

In the **country-capital** example, the countries are mapped to their capitals as they represent some form of relationship between them **=**

`vector(China) - vector(Beijing) + vector(Canada) = vector(Ottawa)`

2. Extract Training Data

This involves mining word-tag relations in a sentence. We chose the **Brown Corpus** as our training data. We mined the brown corpus to get pairs like:

```
I/PRP am/VBP John/NNP.
```

But the data needs to be represented as `input_data` and `output_labels` so that we can train and evaluate our model.

The correct form is:

```
Input_data = [ I, am, John]
```

`Output_labels [ PRP, PP, NNP]`

Most importantly, the sequence needs to be preserved and the sentences need to remain sentences.

3. Training

We need to train an LSTM to do the prediction with Input data

For example :

`INPUT: [[I, am, John], [Phillip,is, young]]`

`OUTPUT: [[PRP, VB, NNP], [NNP, VB, NN]]`

However, instead of `'am'` it will be a 100 dimensional vector from Glove like `[1.2, 34.5 , …, 23]`.

We use a Bidirectional LSTM layer

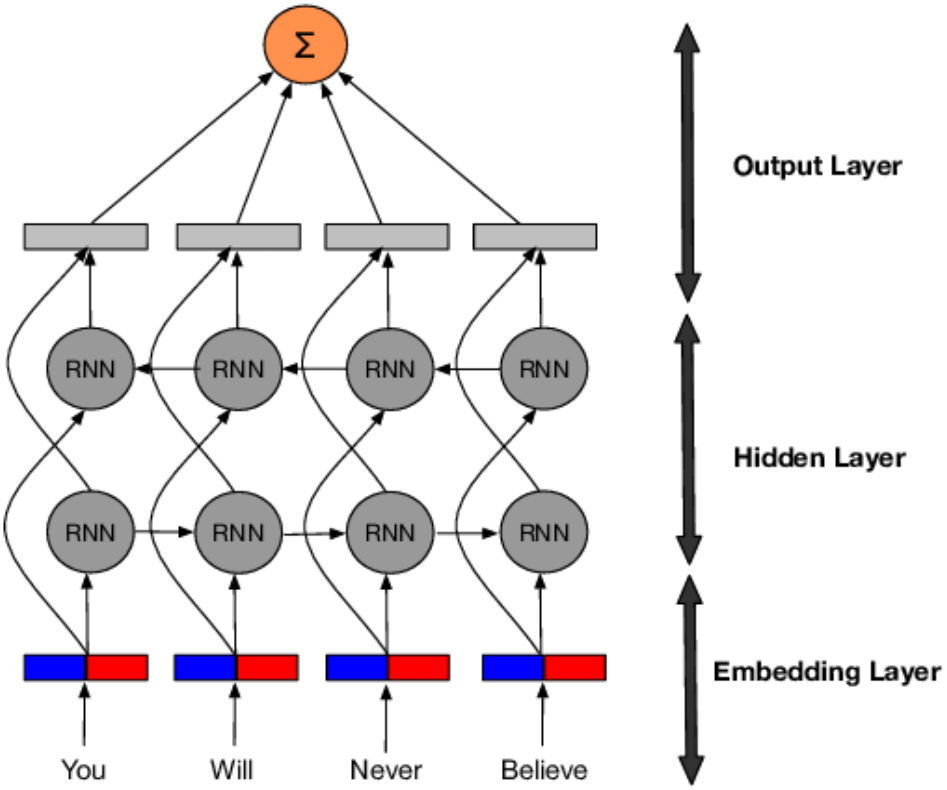
Bidirectional LSTMs (BLSTMs) were introduced to increase the amount of input information available to the network. For example, multilayer perceptrons have limitations on the input data flexibility, as they require their input data to be fixed. Standard recurrent neural network (RNNs) also have restrictions as the future input information cannot be reached from the current state. On the contrary, Bidirectional LSTMs do not require their input data to be fixed. Moreover, their future input information is reachable from the current state.

The basic idea of BLSTMs is to connect two hidden layers of opposite directions to the same output.

By this structure, the output layer can get information from past and future states.

In our implementation of the model, we use a bidirectional LSTM instead of a unidirectional one. LSTM in its core, preserves information from inputs that has already passed through it using the hidden state. Unidirectional LSTM only preserves information of the past because the only inputs it has seen are from the past.

Using bidirectional will run your inputs in two ways, one from past to future and one from future to past and what differs this approach from unidirectional is that in the LSTM that runs backwards you preserve information from the future and using the two hidden states combined you are able in any point in time to preserve information from both past and future. Because of their ability to approach a unit from both the directions, they tend to understand context better.



4. Testing

Finally, once the model has been trained, we will test it on the testing set. We have divided our data set into two parts - 80% of it is used for training, and 20% is used for testing.

Implementation Details

We implemented the whole system on Keras which is a wrapper for the TensorFlow library by Google. The programming language used was Python.

Using Glove Vectors

In order to get the Glove vectors, we can:

1. Train the system on our own data to get the vectors
2. Use pretrained vectors

The problem with training is that:

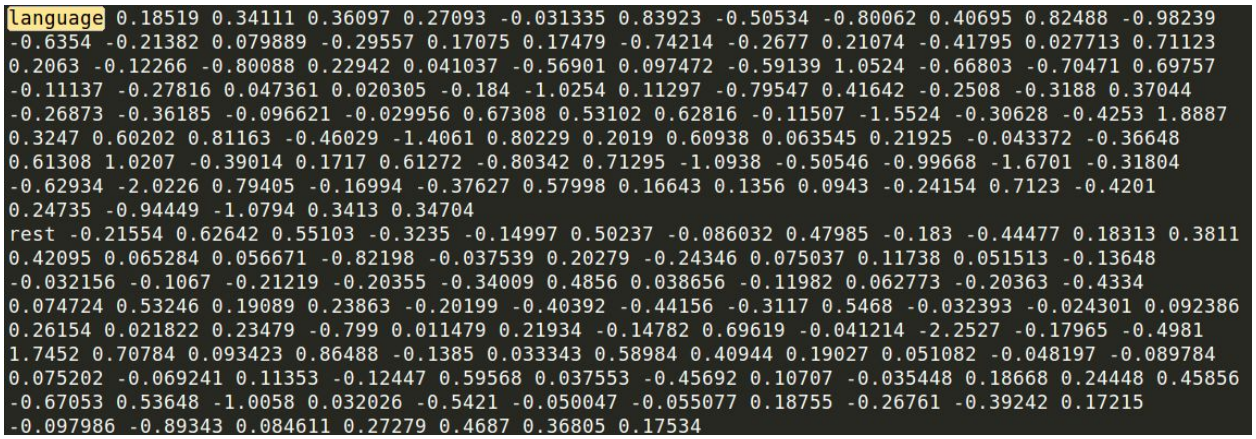
1. It takes a lot of time
2. It requires high amount of compute resources
3. It requires a lot of training data. Although the training data doesn't need to be hand labelled, the sheer amount is very large. Google trained their model on 6 Billion tokens for 3 days and only managed to train for 3 iterations.

Considering all this, we decided to go with a pretrained model taken from https://nlp.stanford.edu/projects/glove/

They provide word vectors in varying dimensions from 50, 100 till about 200. Adding more dimensions means that the vectors can better express the words and their dimensional meanings.

We had to consider a trade off between the **high expressibility** and **computation/memory resources** for high dimensional (more than 100 dimensions)
After reading a few papers and experimenting with different sizes, we finally settled on 100 dimensions, a standard very common in academia as well as industry.

The Glove vector file we downloaded, however, was a .txt file and needed further extraction.



In this image, we can see the vectors for the word 'language' and 'rest'

For this we wrote a script, `make_glove_pickle.py` which creates a dictionary with the word as the key and the vector as the value. Finally, the dictionary was saved as binary file using the pickle library.

For data extraction

We wrote a script (`extract_data.py`) to extract the sentences with the tags for each word.

The output from running extract_data.py:

```
TOTAL NO. OF FILES  500

RUNNING ON  20  FILES

CORPUS SIZE 3575

OMITTED sentences:  12

TOTAL NO OF SAMPLES:  2258

sample X_train:  ['rhode', "island's", 'rate', 'of', '$.07', 'per', 'mile',
'is', 'considerably', 'lower', 'than', 'reimbursable', 'rates', 'in', 'the',
'federal', 'government', 'and', 'in', 'industry', 'nationally', 'which',
'approximate', 'a', '$.09', 'per', 'mile', 'average', '.']

sample Y_train:  ['np-tl', 'nn$-tl', 'nn', 'in', 'nns', 'in', 'nn', 'bez',
'ql', 'jjr', 'cs', 'jj', 'nns', 'in', 'at', 'jj', 'nn', 'cc', 'in', 'nn', 'rb',
'wdt', 'vb', 'at', 'nns', 'in', 'nn', 'nn', '.']

VOCAB SIZE:  7862
TOTAL TAGS:  194

sample X_train_numberised:  [448, 4093, 4880, 783, 7000, 2919, 2684, 306, 4997,
112, 6107, 3817, 1313, 4128, 6589, 4242, 5824, 6869, 4128, 919, 4240, 7290,
3627, 6545, 7213, 2919, 2684, 962, 900]

sample Y_train_numberised:  [131, 78, 101, 133, 52, 133, 101, 102, 189, 186,
53, 161, 52, 133, 35, 161, 101, 59, 133, 101, 100, 163, 26, 35, 52, 133, 101,
101, 183]

Saved as pickle file
```

As can be seen from the output, we :

1. Extracted word-tag pairs in a sentence wise manner
2. Converted all the words and tags into unique numbers using a dictionary
3. Saved the formatted training data word2int and tag2int dictionaries in a binary format called data.pkl

We found that:

While the Brown Corpus has around 87 tags, a revised version of the corpus has 472 different tags.
The full list can be found here:
https://github.com/slavpetrov/universal-pos-tags/blob/master/en-brown.map

```
WDT-HL  DET
WDT-NC  DET
WP$     DET
WPO     PRON
WPO-NC  PRON
WPO-TL  PRON
WPS     PRON
WPS+BEZ PRT
WPS+BEZ-NC      PRT
WPS+BEZ-TL      PRT
WPS+HVD PRT
WPS+HVZ PRT
WPS+MD  PRT
WPS-HL  PRON
WPS-NC  PRON
WPS-TL  PRON
WQL     ADV
WQL-TL  ADV
WRB     ADV
WRB+BER PRT
WRB+BEZ PRT
WRB+BEZ-TL      PRT
WRB+DO  PRT
WRB+DOD PRT
WRB+DOD*        PRT
WRB+DOZ PRT
WRB+IN  PRT
WRB+MD  PRT
WRB-HL  ADV
WRB-NC  ADV
WRB-TL  ADV
``      .
```

A sample of the 472 tags

Training the model

This part was the most important and time consuming part in the project. We used the keras library with TensorFlow backend to run the training.
TensorFlow is a library which makes

- Numerical computations like matrix multiplication highly efficient
- Parallelizes the code as much as possible without programmer interference
- Treats the data like tensors flowing between operating points

This was specially useful for us because training on such data takes a lot of time. We had at our disposal an NVidia GTX 960M.

For this, we wrote a script: `make_model.py`
The script imports all the data we had pickled initially (the Glove vector dictionary and the training data)

While the data was in a good format, we needed to make it compatible with the format which could be accepted by a neural network.

This involves 2 steps:

1. **Padding the data**
Since all the sentences aren't of the same length, it creates a lot of variability. This creates a staggered matrix/tensor which isn't good for fast training in the TensorFlow paradigm. Therefore, we had to pad all the words with zeros in the beginning.
For example:

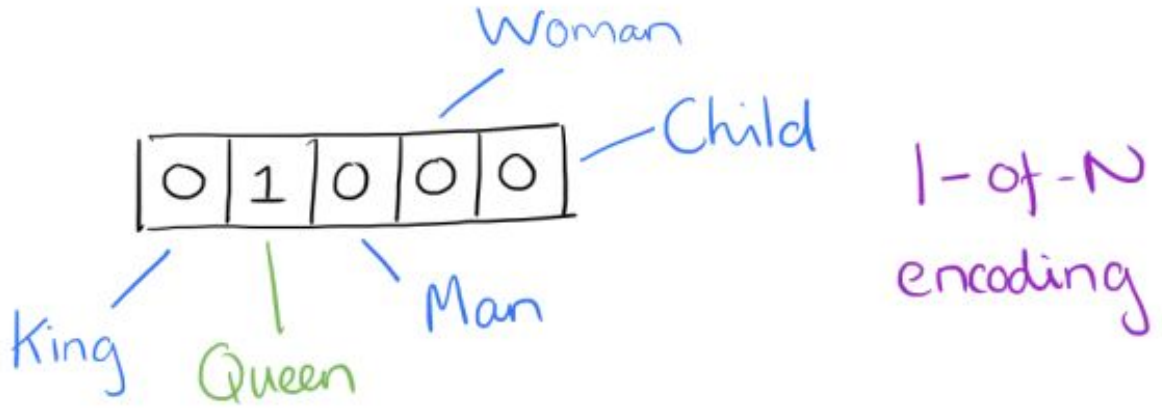
```
X_train : [ John, is, running ]
X_train_numberised: [ 23, 75, 234]
X_train_padded: [0, 0, 0, 0, 0, …, 0, 23, 75, 234]
```

This way, all the sentences have the same dimension.
Empirically, we chose 100. However, this might be too big and could be replaced by a smaller number like 50 or 25. (But 100 is good for scalability.)

2. Converting tags to categorical form
The neural network that we designed provides an output probability for each tag.
For example, given the sequence: ['John', 'is', 'running']
It will predict the probabilities for all the possible tags for each word. So, for 'John', it will make a prediction: [.2 .2 0.1 0.1 0.4] of the probabilities.
This can be read as
20% probability of NNP
20% probability of VBX
.
.
.

Here NNP will always be index 0, VBX index 1, etc

But the tags we have are in the form of integers.
We need to represents them as M dimensional vectors where M is the total number of tags. The vector should also be a one hot vector.

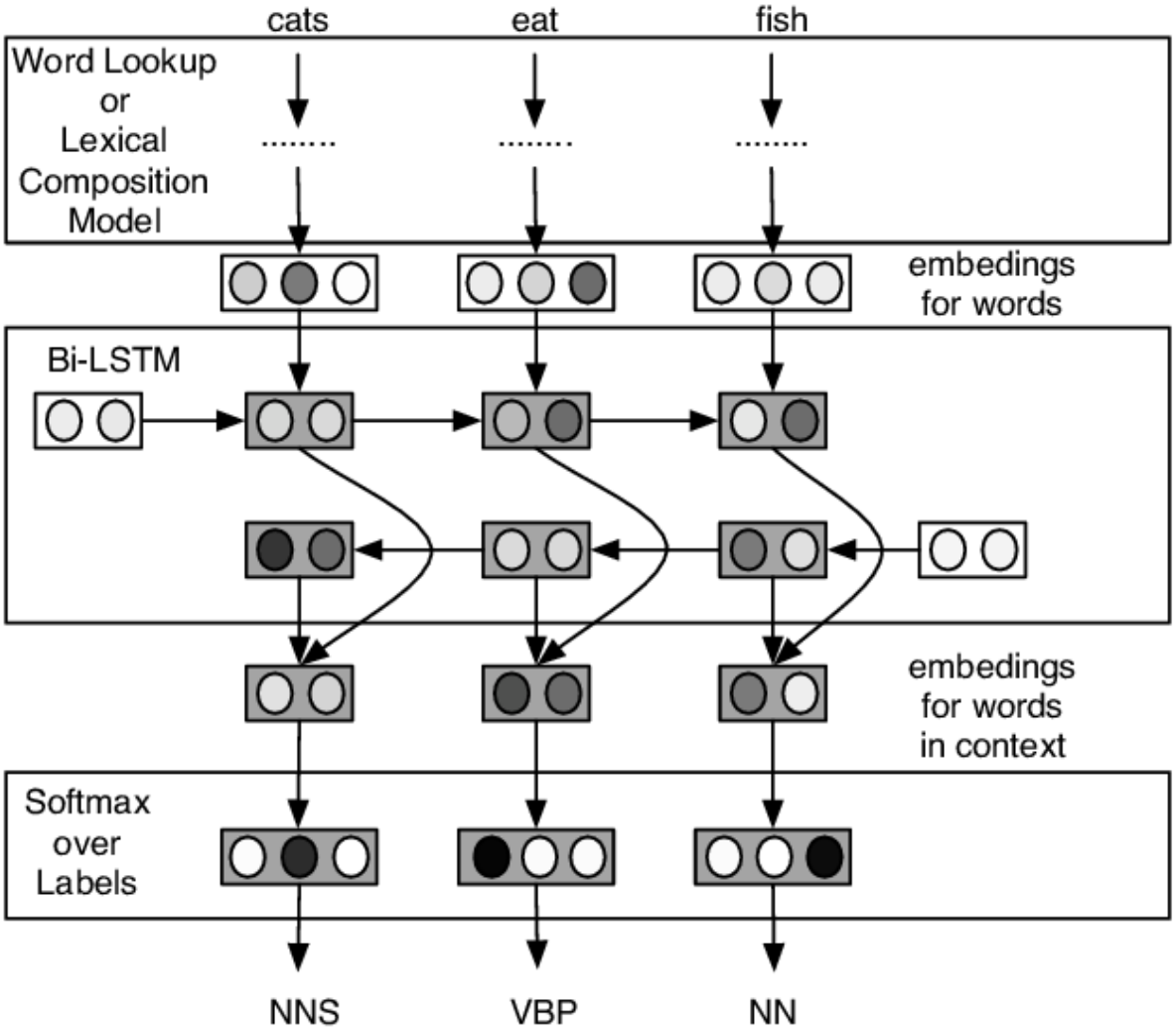


Here index 0 represents King and so on

So,
`NNP -> 4 -> [0,0,0,0,1,0,0,0]`
Where the total number of unique tags are 8

At this point, the data was ready for the Neural Network.

Network Architecture



The basic outline of the architecture is:

1. An Embedding Layer

This layer converts the input words from their unique number identifiers to their Glove vectors using the vectors we downloaded.

2. A Bidirectional LSTM layer

This layer studies the pattern in the sequences presented by the data and creates feature patterns which make it easier for the fully connected layer to make a prediction on the data.

3. A Fully Connected Layer:
This layer takes its input from the BLSTM and transforms into a lower dimensional form. This lower dimensional vector is of the size of the number of tags. The output is passed through the softmax activation function which converts the vector to probability values.

$$\sigma(x_j) = \frac{e^{x_j}}{\sum_i e^{x_i}}$$

The tag with the highest probability is considered as the output tag.

Consider the summary of the network given below

```
model fitting - Bidirectional LSTM
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 100)               0
_________________________________________________________________
embedding_1 (Embedding)      (None, 100, 100)          786300
_________________________________________________________________
bidirectional_1 (Bidirection (None, 100, 138)          93840
_________________________________________________________________
time_distributed_1 (TimeDist (None, 100, 195)          27105
=================================================================
Total params: 907,245
Trainable params: 907,245
Non-trainable params: 0
_________________________________________________________________
Train on 1807 samples, validate on 451 samples
```

Optimization Technique:
The Neural Network is made to reduce the classification error using
Categorical Cross Entropy as the Loss function and
RMSProp as the optimization algorithm to reduce the defined loss

```
model.compile(loss='categorical_crossentropy',
              optimizer='rmsprop',
              metrics=['acc'])
```

Training Parameters:
We trained the model for 20 epochs on 250 samples of the Brown corpus.
It was able to attain 98% validation set accuracy.
All the other parameters were kept to default values as suggested by academia.

One-hot encoded vectors

The number of unique words in a corpus will be very less when compared to the total number of words. If a word is represented in its atomic format, it becomes very difficult to solve a variety of problems. One-hot encoded vectors do away with this problem by constructing an array of size which is equal to the number of unique words in the corpus. Each word in the corpus is given a unique ID. A word is represented according to its ID wherein the word is marked as 1 and rest of the words are marked as 0.

$$[0 \quad 0 \quad 0 \quad 1 \quad 0] \times \begin{bmatrix} 17 & 24 & 1 \\ 23 & 5 & 7 \\ 4 & 6 & 13 \\ 10 & 12 & 19 \\ 11 & 18 & 25 \end{bmatrix} = [10 \quad 12 \quad 19]$$

When we multiply the one hot vectors with W1, we get access to the row of W1 which is in fact the embedded representation of the word represented by the input one hot vector. So W1 essentially acts as a lookup table.

<u>Case study</u>

The POS-tagger can be used to disambiguate homonyms. Homonyms are words that sound similar or have the same spellings. These words can easily be confused with each other. POS-tagging can be used to disambiguate these words. It embeds information which is specific to a word which makes it easier to differentiate between the words. From the computer's point of view, the words become distinct. They can be

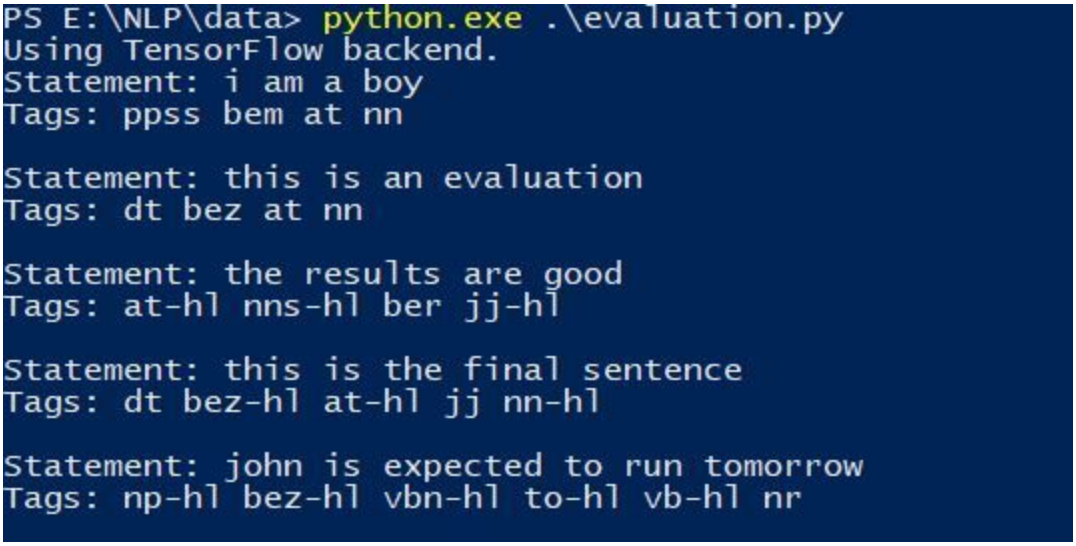
processed more efficiently. Assigning POS tags to the words give us more information more about the word. POS-tagging can be used to characterise the context in which a word is being used in spoken or written text. If a word can be used in multiple ways, then POS-tagging helps in getting the context in which the word is being used.

POS-tagging is useful for determining authorship. Word frequency distributions can be used to determine if an article is written by the same author or not.

It can be used for speech synthesis and recognition. Each word can be pronounced in a different way according to the way it is used in a sentence. Assigning POS tags to the words can help in speech synthesis and recognition as the pronunciation will be different for each word according to the context.

Results

We were successfully able to achieve a high tagging accuracy *(96.3%)* for the given corpus. Following is an example of various random sentences tested against the trained model.



We can see that in most cases, the words are accurately tagged. Since we used a smaller subset of the actual corpus for training due to resource limitations, we have some ambiguities (*This* in the 4th sentence is tagged as a determiner *dt*, even though it is an *existential this*).

Tools

1. Keras - Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow.
2. TensorFlow - TensorFlow is an open-source software for machine intelligence.

References

1. Part-of-speech tagging with bidirectional long-short term memory recurrent neural network (https://arxiv.org/abs/1510.06168)
2. Sequence Tagging with TensorFlow (https://guillaumegenthial.github.io/sequence-tagging-with-tensorflow.html)
3. Sequence models and long-short term memory networks (http://pytorch.org/tutorials/beginner/nlp/sequence_models_tutorial.html)
4. Understanding LSTMS (http://colah.github.io/posts/2015-08-Understanding-LSTMs/)