# CS 287
# Assignment 4: Word Segmentation

**Due:** Friday, April 1st, 11:59 pm

Over the last couple years there has been tremendous interest in recurrent neural networks for natural language tasks, and in particular the long short-term memory (LSTM) network. The last several classes have focused on the theory and structure of recurrent networks. For this assignment you will actually construct utilize an LSTM for a real-world problem.

The problem we will focus on is word segmentation, that is identifying the spaces in sentence-based on the previous characters only. This is a simplified version of word segmentation processes necessary for processing languages written without spaces, such as Korean or Chinese. Unlike past assignments, our work here will not be based on any existing paper. However this will give you a chance to experiment with the LSTM on this task.

Before you start we advise that you get very comfortable with the notes from class, the papers on LSTMs, and especially the torch `rnn` library.

As you complete this assignment, we ask that you submit your results on the test data to the Kaggle competition website at `https://inclass.kaggle.com/c/cs287-hw4` and that you compile your experiences in a write-up based on the template at `https://github.com/cs287/hw_template`.

## 1 Data and Preprocessing

### 1.1 Data

The data for this task is under the `data/` directory. Our data is the same of the previous assignment, except that now we have given you only the characters of the data set separated with explicit space tokens.

```
> head data/valid_chars.txt
c o n s u m e r s <space> m a y <space> w a n t <space> t o <space> m o
    v e <space> t h e i r
<space> t e l e p h o n e s <space> a <space> l i t t l e <space> c l o
    s e r <space> t o <space>
t h e <space> t v <space> s e t </s> < u n k > <space> < u n k > <space
   > w a t c h i n g <space>
a b c <space> ' s <space> m o n d a y <space> n i g h t <space> f o o t
    b a l l <space> c a n
```

```
<space> n o w <space> v o t e <space> d u r i n g <space> < u n k > <
   space> f o r <space> t h e
<space> g r e a t e s t <space> p l a y <space> i n <space> N <space> y
    e a r s <space> f r o m
<space> a m o n g <space> f o u r <space> o r <space> f i v e <space> <
    u n k > <sp
```

Note that for this problem set we no longer print the data file in lines, we simply concatenate together each one of the sentence separated with a $< /s >$ symbol.

The test data is in file `test_chars.txt`. This file consists of one sentence per line and includes no explicit spaces. The aim of the project is to re-insert spaces back into the sentences. We have also provided this file for the validation data.

```
> head data/valid_chars_kaggle.txt
c o n s u m e r s   m a y   w a n t   t o   m o v e   t h e i r   t e l e p h
    o n e s   a   l i t t l
e   c l o s e r   t o   t h e   t v   s e t </s>
< u n k >   < u n k >   w a t c h i n g   a b c   ' s   m o n d a y   n i g h
    t   f o o t b a l l   c
a n   n o w   v o t e   d u r i n g   < u n k >   f o r   t h e   g r e a t e
   s t   p l a y   i n   N   y
 e a r s   f r o m   a m o n g   f o u r   o r   f i v e   < u n k >   < u n k
     > </s>
t w o   w e e k s   a g o   v i e w e r s   o f   s e v e r a l   n b c   < u
   n k >   c o n s u m e r
 s e g m e n t s   s t a r t e d   c a l l i n g   a   N   n u m b e r   f o
    r   a d v i c e   o n   v
a r i o u s   < u n k >   i s s u e s </s>
a n d   t h e   n e w   s y n d i c a t e d   r e a l i t y   s h o w   h a r
    d   c o p y   r e c o r
d s   v i e w e r s   '   o p i n i o n s   f o r   p o s s i b l e   a i r i
    n g   o n   t h e   n e x
 t   d a y   ' s   s h o w </s>
i n t e r a c t i v e   t e l e p h o n e   t e c h n o l o g y   h a s   t
    a k e n   a   n e w   l e
```

The Kaggle answer file should simply provide the number of spaces for each sentence. We have provided you with a sample answer file for reference.

```
> head data/valid_chars_kaggle_answer.txt
ID,Count
1,13
2,26
3,20
4,20
5,18
6,11
7,26
```

## 1.2 Preprocessing

For this assignment you will write your own preprocessing code in `preprocessing.py`. Preprocessing for this assignment means transforming the training corpus into a representation usable for RNN training. As in previous assignments you will need to create a map from characters to indices. However, since there is no longer a fixed window size the layout is a bit different. The following is a description of how to structure the input data for a transducer RNN.

Say our training corpus consists of character $w_1, \dots, w_n$ In a perfect world, we could have the RNN as one long matrix.

$$\begin{bmatrix} w_1 & w_2 & \dots & w_n \end{bmatrix}$$

However, this would mean we would need to wait $n$ steps to backprop. Instead it is more common to select a fixed *sequence length* $l$ for doing backpropagation. We pass the hidden state between sequences ($\mathbf{s}_l$ combines with $w_{l+1}$), but only backprop within a sequence length window.

$$\begin{bmatrix} w_1 & w_2 & \dots & w_l \end{bmatrix} \begin{bmatrix} w_{l+1} & w_{l+2} & \dots & w_{2l} \end{bmatrix} \dots \begin{bmatrix} w_{n-l+1} & w_{n-l+2} & \dots & w_n \end{bmatrix}$$

When you use this strategy, you much call the following rnn function to save states between matrices

```
model:remember('both')
```

While this method has a fixed backprop length of $s$, with a reasonable size $s$ it works well in practice. Unfortunately it is quite slow since we cannot use a batch (sequential processing). To get around this issue we split the sequence into $b$ parts and fold them over each other.

$$\begin{bmatrix} w_1 & w_2 & \dots & w_l \\ w_{n/b+1} & w_{n/b+1} & \dots & w_{n/b+l} \\ \vdots & & & \\ w_{(b-1)n/b+1} & w_{(b-1)n/b+2} & \dots & w_{(b-1)n/b+l} \end{bmatrix} \begin{bmatrix} w_{l+1} & \dots & w_{2l} \\ w_{n/b+l+1} & \dots & w_{n/b+2l} \\ & \vdots & \\ & \dots & \end{bmatrix} \dots \begin{bmatrix} w_{n/b-l+1} & w_{n/b-l+2} & \dots & w_{n/b} \\ & \dots & & \\ & \vdots & & \\ & \dots & & \end{bmatrix}$$

This allows both batching and backprop with a fixed length. Note though that the hidden states at first timestep ($\mathbf{s}_{n/b+1}$) will be incorrect since they get computed before their correct previous hidden ($\mathbf{s}_{n/b}$). However this is tolerable since it only leads to $b$ incorrect hidden states during training.

Since we will be building a transducer, we also need the training data to be in the same format. Say we are predicting $\hat{y}$ at each step. Then our target output will be of the form,

$$\begin{bmatrix} y_1 & y_2 & \dots & y_l \\ y_{n/b+1} & y_{n/b+1} & \dots & y_{n/b+l} \\ \vdots & & & \\ y_{(b-1)n/b+1} & y_{(b-1)n/b+2} & \dots & y_{(b-1)n/b+l} \end{bmatrix} \begin{bmatrix} y_{l+1} & \dots & y_{2l} \\ y_{n/b+l+1} & \dots & y_{n/b+2l} \\ & \vdots & \\ & \dots & \end{bmatrix} \dots \begin{bmatrix} y_{n/b-l+1} & y_{n/b-l+2} & \dots & y_{n/b} \\ & \dots & & \\ & \vdots & & \\ & \dots & & \end{bmatrix}$$

For language modeling the target output $y_i$ would be the next word $w_{i+1}$. For this assignment, our goal is simply space prediction. Therefore the target output $y_i$ will be 2 if the next word $w_{i+1}$ is a space and 1 otherwise.

## 2 Code Setup

Write your main code in `HW4.lua`. For this assignment part, you can (and should) use the `nn` and `rnn` library in addition to the standard Torch library. You should not need to use any extra libraries.

### 2.1 Hyperparameters

Several of the models described have explicit hyperparameters that you will need to tune. It is your responsibility to cleanly separate these out from the models themselves and expose as command-line options. This makes it much easier to run experiments and to utilize experimental scripts.

### 2.2 Logging and Reporting

As part of the write-up, you will need to report on the training and predictive accuracy of your models. To make this possible, your code should report on various metrics of the model both at training and test time. We will leave it up to you on which metrics to log, but we recommend reporting training speed, training set NLL, training set predictive accuracy, and validation predictive accuracy. It is your responsibility to convince us that the model is correctly training.

## 3 Models

For this assignment you will implement the three models. We will warm up with a count-based model. Then we will try our Bengio NNLM and an LSTM model.

### 3.1 Count-Based Model

To start, modify your code from HW3 to implement a count-based character n-gram model. The model should only give the probability of the next word being a space

$$P(w_i =< \text{space} > | w_{i-n+1}, \dots w_{i-1})$$

Test the model out by determining its perplexity on the validation set and by running on the segmentation task.

For segmentation you should run two inference algorithms.

1. Greedily predict whether the next word is a space, and feedback into the model. (i.e. if it is a space, add the <space> to the context. Otherwise continue normally).
2. Dynamic programming over n-grams as described in class.

### 3.2 Neural Language Model

As a second baseline, you will modify your neural language model from the HW3 to predict whether the next character is a space or not. This should be relatively similar to your previous code, except that you will be prediction 1 or 2 instead of a softmax over words. For this problem you can have a much smaller embedding size of around $\approx 15$

As above you should test your model by running greedy search and dynamic programming (for a small size).

### 3.3 RNN Model

The main portion of the assignment will be implementing the RNN model using the Elements `rnn` library. Much of the main portion of the work is done for you by the library. However getting the details right is crucially important.

For this model you should implement a generic RNN transducer. Given the current state of the RNN you are simply trying to predict whether this is a space in the next word.

**Necessary Hacks:** In any torch model you can call the following function one time to get a vector of all the params and gradients of the params.

```
local params, grad_params = model:getParameters()
```

Once you have these vectors you should do the following:

- To begin you should initialize all the params to be uniform between $-0.05$ and $0.05$.
- Before updating your parameters, you should perform gradient normalization with max norm 5.

Once you have the model, use it in the greedy fashion described above. At each time step predict whether the next word is a space. If it is, feed in a <space> to the model before continuing, otherwise continue onward. Keep track of how many spaces you have seen so far.

### 3.4 Additional Experiments

Once these models are constructed, you should also report on additional experiments on these data sets. We will leave this aspect open-ended, but suggestion include:

- Compare performance between the LSTM, RNN and GRU on this task
- Experiment with stack LSTM and dropout. The rnn library provides details for how to implement this. Also see .
- Try implementing a bidirectional LSTM for prediction. This model is provided by the rnn library. In this model you do not need to do greedy search, however you will have to change your training such that you do not see the spaces in the data.
- Experiment with different optimization techniques. For instance see the `optim` package.

## 4 Report and Submission

For your write-up, follow the report template at `https://github.com/cs287/hw_template`. Be sure to include a link to your code, Kaggle ID, and reports on your results.

In addition to submitting your Kaggle results, we also expect you to report on your experimental process. This should include data tables, graphs and discussion of any issues that you may run into.