HW4: Word Segmentation

Jeffrey Ling
jling@college.harvard.edu

Rohil Prasad
prasad01@college.harvard.edu

April 2, 2016

1 Introduction

In this assignment, we examine the problem of word segmentation. Given a string of characters without spaces, we want to determine where we can insert spaces to segment the string into words.

We implement and discuss several approaches to word segmentation in this paper, all trained on a portion of the Penn Treebank. Our first class of models are Markov models with emission distributions given by an n-gram count model and an adaptation of Bengio's neural network language model (NNLM). Our second class of models are recurrent neural networks (RNNs), namely the Long Short Term Memory (LSTM) network and the Gated Recurrent Unit (GRU) network. Furthermore, we attempt to optimize the evaluation algorithms used to construct segmentations given a model. For the Markovian models, we use a dynamic programming algorithm which greatly improves predictive accuracy. We also make a small adjustment to the RNN evaluation algorithm to improve the performance of these models as well.

In Section 2, we give a formal description of our problem and establish our notation. In Section 3, we give detailed descriptions of the algorithms listed above. In Section 4, we present our experimental results. In Section 5, we discuss our findings.

2 Problem Description

Assume our training corpus consists of a sequence of characters $c_1, c_2, \dots, c_N$ where c_i is drawn from a vocabulary $\mathcal{V}$ for every i. We use `<sp>` to denote the space character.

Our training data represents this corpus as a pair of sequences $c_1, \dots, c_N$ and $y_1, \dots y_N$. The output variable $y_i \in \{0, 1\}$ indicates if character c_{i+1} is a space. By default, we set $y_N = 1$ since c_N is clearly not followed by a space. We desire to find a function $f : \mathcal{V}^k \to \mathbb{R}$ on sequences of characters that computes the probability of the next character being a space, i.e.

$$f(c_1, \dots, c_k) = p(c_{k+1} = \texttt{<sp>} | c_1, \dots, c_k) = p(y_k = 1 | c_1, \dots, c_k)$$

In the test setup, calculating a word segmentation is equivalent to taking an input sequence of characters $\mathbf{c} \in \mathcal{V}^M$ without spaces and outputting a sequence $\mathbf{c}' \in \mathcal{V}^{M'}$ with spaces inserted, s.t. the corresponding sequence of indicators $\mathbf{y} = y_1 y_2 \dots y_{M'}$ has $y_i = 1$ iff c'_{i+1} is a space.

We determine the *best* word segmentation by finding the sequence $\mathbf{c}'$ with corresponding $\mathbf{y}$ that maximizes the score

$$\prod_{k=1}^{M'} p(y_k | c'_1, \ldots, c'_k) = \prod_{k=1}^{M'} f(c'_1, \ldots, c'_k)$$

For the approaches we are considering, this requires us to train a function f. Then, we apply an evaluation algorithm at test time (either based on greedy or dynamic programming) to generate such a segmented sequence. See Section 3 for details.

2.1 Evaluation

We evaluate our models on a validation set in two different ways. The validation set, a sequence of characters with spaces, can be considered as either a single sequence or a set of sentences.

First, we interpret our model as a language model that takes in a sequence of characters $c_1, \ldots, c_i$ and outputs a probability on the next character being a space, i.e. $p(c_{i+1} = \texttt{<sp>} | c_i \ldots c_1)$. Therefore, we can calculate perplexity over the validation set as a single sequence. This is equal to the exponential of the average negative log likelihood,

$$PPL = \exp\left(-\frac{1}{N} \sum_{i=1}^{N} \log p(c_{i+1} = \texttt{<sp>} | c_i \ldots c_1) \right)$$

Second, we can run our model and evaluation algorithm to get a word segmentation over the entire set. Here, we keep track of the number of spaces inserted into each sentence, then compute the mean squared error (MSE) with respect to the true number of spaces per sentence. This is the Kaggle metric.

3 Model and Algorithms

3.1 Markovian Models

The key to training tractable models is the Markov assumption. This allows us to get a simple expression for the joint probability that we can calculate using either a count-based model or a windowed feedforward neural network model. Recall that we desire to model

$$p(c_i = \texttt{<sp>} | c_1, \ldots, c_{i-1}) = p(y_i = 1 | c_1, \ldots, c_{i-1})$$

Under the Markov assumption, we only need to consider the last $n-1$ characters, so that we only need to model

$$p(c_i = \texttt{<sp>} | c_{i-1}, \ldots, c_{i-n+1}) = p(y_i = 1 | c_{i-1}, \ldots, c_{i-n+1}) \tag{1}$$

We call models with such an assumption an n-gram model.

3.1.1 Count Model

To model equation 1 as an n-gram model, we compute co-occurrence counts $C(c_i = \texttt{<sp>}, c_{i-1}, \dots, c_{i-n+1})$ and $C(c_i \neq \texttt{<sp>}, c_{i-1}, \dots, c_{i-n+1})$ in the training sequence. Then with a smoothing parameter α, we let

$$p(c_i = \texttt{<sp>} | c_{i-1}, \dots, c_{i-n+1}) = \frac{\alpha + C(c_i = \texttt{<sp>}, c_{i-1}, \dots, c_{i-n+1})}{2\alpha + C(c_i = \texttt{<sp>}, c_{i-1}, \dots, c_{i-n+1}) + C(c_i \neq \texttt{<sp>}, c_{i-1}, \dots, c_{i-n+1})}$$

For our models we set $\alpha = 0.1$.

3.1.2 Neural Network Language Model (NNLM)

We use the language model from **?** like last assignment to model equation 1. We have parameters

- $W_0 \in \mathbb{R}^{|\mathcal{V}| \times d_{in}}$, a lookup table of character embeddings
- $W_1 \in \mathbb{R}^{nd_{in} \times d_{hid}}$, $b_1 \in \mathbb{R}^{d_{hid}}$,
- $W_2 \in \mathbb{R}^{d_{hid} \times 2}$, $b_2 \in \mathbb{R}^2$

First, the character context is transformed into character embeddings by multiplying with W_0, and then concatenated to get a vector $\mathbf{x}_0$ of size $n \cdot d_{in}$. Then, we get a size 2 vector of scores

$$\mathbf{z} = \tanh(\mathbf{x}_0 W_1 + b_1) W_2 + b_2 \in \mathbb{R}^2$$

Finally, we can force a probability distribution by taking the softmax $\hat{y} = \text{softmax}(\mathbf{z})$. Then, we have the probability that the next word is a space is $\hat{y}_2$.

As hyperparameters, we set $d_{in} = 15, d_{hid} = 100$, and we used a gram size of 5.

3.2 Recurrent Neural Networks

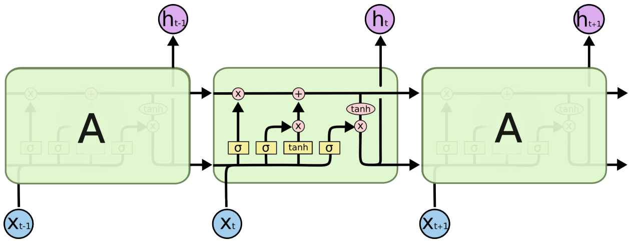
Our recurrent neural networks have the following parameters:

- $W_0 \in \mathbb{R}^{|\mathcal{V}| \times d_{in}}$,
- $\mathbf{s}_0 \in \mathbb{R}^{d_{hid}}$, the initial hidden state,
- $R : \mathbb{R}^{d_{hid}} \times \mathbb{R}^{d_{in}} \to \mathbb{R}^{d_{hid}}$, a parameterized state transition function,
- $W_1 : \mathbb{R}^{d_{hid}} \to \mathbb{R}^2$, $b_1 \in \mathbb{R}^2$

The exact definition of R depends on the type of RNN we are using. All of our RNNS are **transducer** RNNs, meaning that they consume an input sequence $c_1, \dots, c_M$ of characters and output a sequence $\hat{y}_1, \dots, \hat{y}_M$, where $\hat{y}_i = p(y_i | c_i, c_{i-1}, \dots, c_1)$.

3.2.1 Long Short Term Memory Network

We use the LSTM model of **?**, illustrated below:



Formally, this requires the introduction of some new parameters

- $W^{xi} \in \mathbb{R}^{d_{in} \times d_{hid}}$, $W^{si} \in \mathbb{R}^{d_{hid} \times d_{hid}}$, $b^i \in \mathbb{R}^{d_{hid}}$,
- $W^{xi} \in \mathbb{R}^{d_{in} \times d_{hid}}$, $W^{sj} \in \mathbb{R}^{d_{hid} \times d_{hid}}$, $b^j \in \mathbb{R}^{d_{hid}}$,
- $W^{xi} \in \mathbb{R}^{d_{in} \times d_{hid}}$, $W^{sf} \in \mathbb{R}^{d_{hid} \times d_{hid}}$, $b^f \in \mathbb{R}^{d_{hid}}$,
- $W^{xi} \in \mathbb{R}^{d_{in} \times d_{hid}}$, $W^{so} \in \mathbb{R}^{d_{hid} \times d_{hid}}$, $b^o \in \mathbb{R}^{d_{hid}}$,
- $\mathbf{C}_0 \in \mathbb{R}^{d_{hid}}$, the initial cell state

Let c_i be our input character, and let $\mathbf{C}_{i-1}$ and $\mathbf{s}_{i-1}$ be the previously calculated cell and hidden states, respectively. First, we transform c_i into a vector $\mathbf{x}_i$ of size d_{in} by multiplying it by W_0. Then, we calculate the four following vectors

$$\begin{aligned}
\mathbf{i} &= \tanh(\mathbf{x}_i W^{xi} + \mathbf{s}_{i-1} W^{si} + b^i) \\
\mathbf{j} &= \sigma(\mathbf{x}_i W^{xj} + \mathbf{s}_{i-1} W^{sj} + b^j) \\
\mathbf{f} &= \sigma(\mathbf{x}_i W^{xf} + \mathbf{s}_{i-1} W^{sf} + b^f) \\
\mathbf{o} &= \sigma(\mathbf{x}_i W^{xo} + \mathbf{s}_{i-1} W^{so} + b^o)
\end{aligned}$$

and combine them to produce the new cell state and hidden state

$$\begin{aligned}
\mathbf{C}_i &= \mathbf{j} \odot \mathbf{i} + \mathbf{f} \odot \mathbf{C}_{i-1} \\
\mathbf{s}_i &= R(\mathbf{s}_{i-1}, \mathbf{x}_i) = \tanh(\mathbf{C}_i) \odot \mathbf{o}
\end{aligned}$$

Then, we multiply $\mathbf{s}_i$ by W_1 to produce an output vector $\mathbf{z}_i$ of size 2. Finally, we force a probability distribution by taking the softmax to get the desired output $\hat{y}_i = \text{softmax}(\mathbf{z}_i)$.

As hyperparameters, we use $d_{in} = 15, d_{hid} = 100$ as in the NNLM.

3.2.2 Gated Recurrent Unit Network

We use the GRU model of **?**. This model requires the following parameters in addition to the generic RNN parameters

- $W^{xt} \in \mathbb{R}^{d_{in} \times d_{hid}}$, $W^{st} \in \mathbb{R}^{d_{hid} \times d_{hid}}$, $b^t \in \mathbb{R}^{d_{hid}}$,

- $W^{xr} \in \mathbb{R}^{d_{in} \times d_{hid}}$, $W^{sr} \in \mathbb{R}^{d_{hid} \times d_{hid}}$, $b^r \in \mathbb{R}^{d_{hid}}$,
- $W^x \in \mathbb{R}^{d_{in} \times d_{hid}}$, $W^s \in \mathbb{R}^{d_{hid} \times d_{hid}}$, $b \in \mathbb{R}^{d_{hid}}$

Let c_i be our input character, and let $\mathbf{s}_{i-1}$ be the previously calculated hidden state. First, we transform c_i into a vector $\mathbf{x}_i$ of size d_{in} by multiplying it by W_0. Then, we calculate the following three vectors

$$\begin{aligned}\widetilde{\mathbf{h}} &= \tanh(\mathbf{x}_i W^x + (\mathbf{r} \odot \mathbf{s}_{i-1}) W^s + b) \\ \mathbf{r} &= \sigma(\mathbf{x}_i W^{xr} + \mathbf{s}_{i-1} W^{sr} + b^r) \\ \mathbf{t} &= \sigma(\mathbf{x}_i W^{xt} + \mathbf{s}_{i-1} W^{st} + b^t)\end{aligned}$$

and combine them to produce the new hidden state

$$\mathbf{s}_i = R(\mathbf{s}_{i-1}, \mathbf{x}_i) = (1 - \mathbf{t}) \odot \widetilde{\mathbf{h}} + \mathbf{t} \odot \mathbf{s}_{i-1}$$

Then, we multiply $\mathbf{s}_i$ by W_1 to produce an output vector $\mathbf{z}_i$ of size 2. Finally, we force a probability distribution by taking the softmax to get the desired output $\hat{y}_i = \text{softmax}(\mathbf{z}_i)$.

3.3 Evaluation Algorithms

We have three different algorithms that we use to take a model and use it to predict a sequence $y_1 \dots y_M$ for an input sequence $c'_1 \dots c'_M$. The first two are a greedy algorithm and a dynamic programming algorithm for our Markovian models. The last one is a greedy algorithm for our RNN models. In all of them, we will denote our model by a function f from the input space to the space/non-space probability distribution.

3.3.1 Markovian Greedy

Given a sequence without spaces $c_1, \dots, c_M$, we proceed greedily as follows. Given a context $c_{i-n+1}, \dots, c_{i-1}$, we compute $p(c_i = \texttt{<sp>} | c_{i-1} \dots c_{i-n+1})$. If this is greater than 0.5, we insert a space and repeat with context $c'_{i-n+1}, \dots, c'_{i-1}$ with $c'_{i-1} = \texttt{<sp>}$ and $c'_j = c_{j+1}$; otherwise we set c'_{i-1} as the next character in the sequence.

In this manner we can process the entire test sequence to output spaces.

3.3.2 Markovian Dynamic Programming

Dynamic programming seeks to optimize the joint probability of a sequence with spaces:

$$\prod_{i=1}^{N} p(c_i = \texttt{<sp>} | c_{i-1}, \dots, c_{i-n+1})$$

Note that for a bigram model, if we assume that spaces cannot follow spaces, this probability can be optimized by the greedy algorithm (since the only terms in this probability are of the form $p(c_i = \texttt{<sp>} | c_{i-1})$ for c_{i-1} a non-space).

In a general n-gram model, we use a version of the Viterbi algorithm. We keep a table $\pi(i, B)$ where $B = y_{i-n+1} \dots y_i$ is a bitmask of length n. The bits of B, y_i, indicate which characters in the original sequence $c_{i-n+1}, \ldots, c_{i-1}, c_i$ have a space following. Then $\pi(i, B)$ is equal to the joint log probability of the sequence $c_1, \ldots, c_i$ with final indicators of spaces $B = y_{i-n+1} \cdots y_i$.

The Viterbi transition is then given by the recursion

$$\pi(i, B) = \max_{B'} \pi(i-1, B') + g(\mathbf{c}_{(i-n+1):i}, B, B')$$

Here, B' is a bitmask which has the last $n-1$ bits equalling the first $n-1$ bits of B, and $g(\mathbf{c}_{(i-n+1):i}, B, B')$ gives

$$\log p(y_i | c_i, \ldots, c_{i-n+1}, y_{i-1}, \ldots, y_{i-n+1})$$

Note that this probability only depends on the previous $n-1$ characters as determined by the c_j and spaces y_j.

We can compute the optimal insertion of spaces by computing the table for π and saving backpointers.

3.3.3 RNN Greedy

The algorithm is given formally below. In plain language, we simply feed in each character one at a time to our RNN. We set a cutoff probability $\mu \in (0, 1)$, so that we insert a space if the RNN outputs probability $> \mu$ of the next character being space. If the RNN predicts a space, we record that in our output dictionary, feed the space back into the RNN, and then continue to the next character.

```
procedure RNNGreedy($c'_1, \ldots, c'_M, f, \mu$)
    $i \leftarrow 1$
    $sn \leftarrow 1$                                   ▷ Set sentence number to 1
    $S = \{\}$                                          ▷ $S$ maps sentences to number of spaces predicted
    $S[sn] = 0$
    while $i \leq M$ do
        if $c'_i =< /s >$ then
            $sn \leftarrow sn + 1$
            $S[sn] \leftarrow 0$                        ▷ Increment sentence number at end of sentence
        $y \leftarrow f(c'_i)_2$                        ▷ Feed $c'_i$ into the model
        if $y \geq \mu$ then
            $y \leftarrow f(< sp >)$
            $S[sn] = S[sn] + 1$                         ▷ Feed a space into the model
        $i \leftarrow i + 1$
    return $S$
```

4 Experiments

We evaluate train and valid perplexity for all models. Table 1 summarizes results.

After training, we also record the MSE for each model on valid with both greedy and dynamic programming test methods. Table 2 summarizes results.

Model	Train PPL	Valid PPL
2-GRAM COUNT	1.397	1.400
3-GRAM COUNT	1.267	1.273
4-GRAM COUNT	1.173	1.183
5-GRAM COUNT	1.122	1.149
6-GRAM COUNT	1.102	1.168
NNLM	1.153	1.162
LSTM	1.110	1.142
GRU	1.097	1.138

Table 1: Perplexity on space insertion.

Model	MSE
5-gram Count+Greedy	23.67
5-gram Count+DP	3.20
NNLM+Greedy	15.14
NNLM+DP	11.26
LSTM	5.42
GRU	5.26

Table 2: MSE on space insertion for valid. The RNNs used $\mu = 0.3$ as cutoff probability.

4.1 Count Model

For the count model, we experiment with the gram size up to 5. We use smoothing $\alpha = 0.1$. We find that 5-gram gets the best valid perplexity, so we use this for the NNLM as well.

4.2 NNLM

For the NNLM, we trained with hidden size 100, word embedding size 15, learning rate 0.01, batch size 32, and trained for 10 epochs (upon which validation perplexity stopped decreasing). We used a context size (i.e. gram size) of 5. The NNLM takes about 30 seconds to train per epoch.

4.3 RNN

For both the LSTM and GRU, we trained with hidden size 100, word embedding size 15, learning rate 0.1, batch size 32, and trained for 20 epochs. We track validation perplexity per epoch and halve learning rate if it stops decreasing (though this wasn't necessary in 20 epochs). The LSTM and GRU both take about 30 seconds per training epoch.

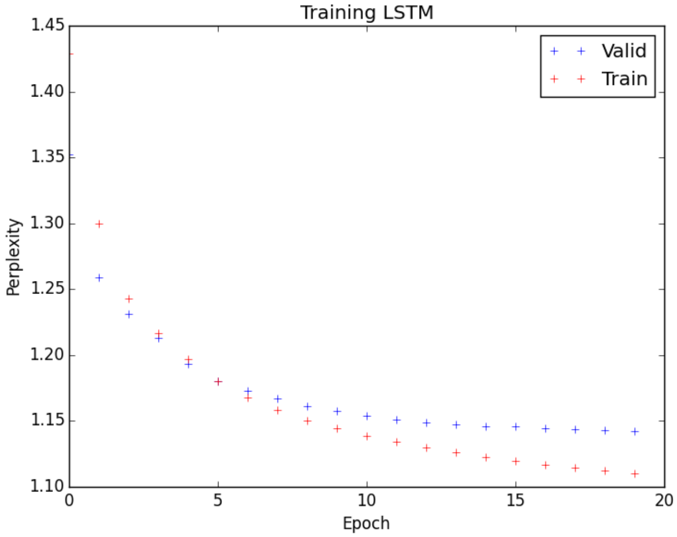


Figure 1: Training curves for LSTM.

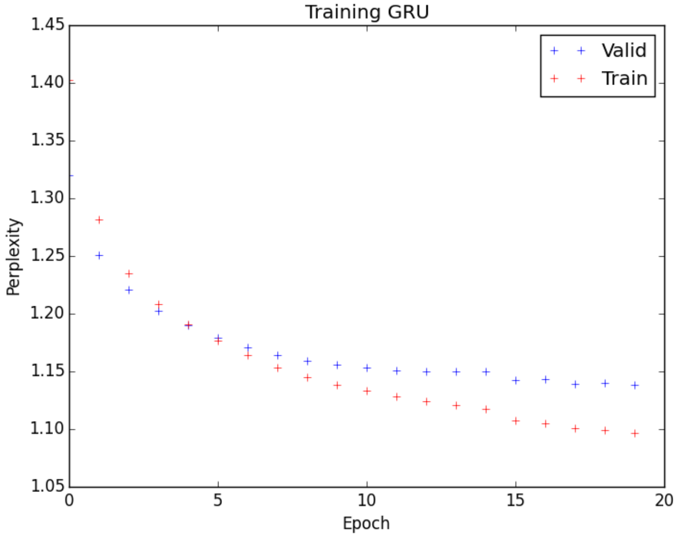


Figure 2: Training curves for GRU.

We experimented with the cutoff probability μ for the LSTM. We find that MSE on valid is optimized at $\mu = 0.3$, which was optimized by grid search on the LSTM trained for 20 epochs. The results are given in the figure below.

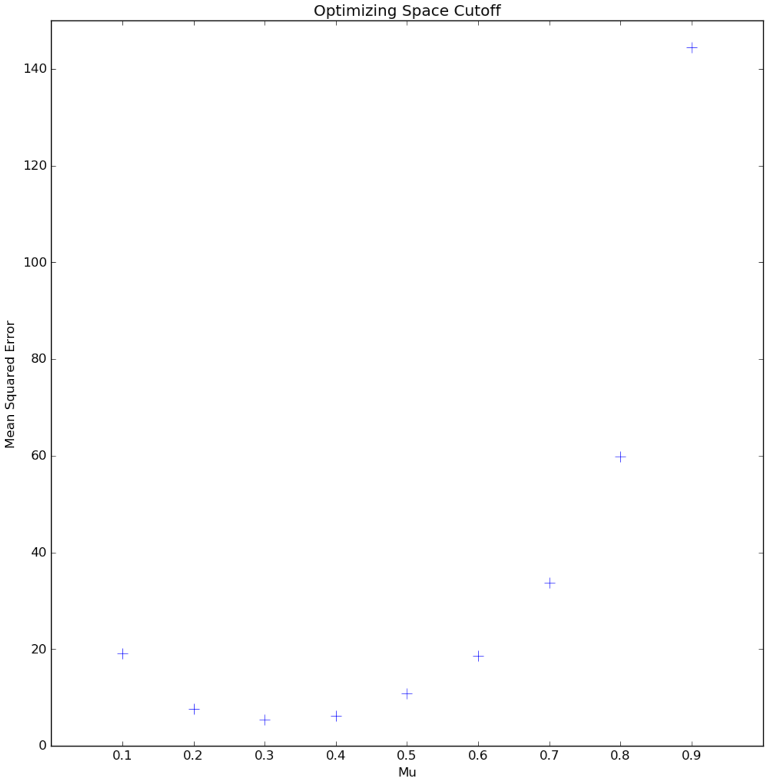


Figure 3: Grid search for μ

5 Conclusion

Our results for perplexity on the validation set with spaces were generally as expected. Since our RNNs could dispense with the Markovian assumption and capture the *entire* context of a character, it seems like they might perform best. However, the 5-gram count model came quite close in perplexity.

For our MSE results on the validation set without spaces, it seems that optimizing our test-time sequence prediction algorithms contributed the most to lowering the MSE. For our RNNs, adjusting the space cutoff μ had a very pronounced effect on the MSE, lowering from 10.9 at the standard $\mu = 0.5$ to 5.4 at $\mu = 0.3$. Dynamic programming also performed dramatically better than greedy.

Our best-performing model on this validation set (with MSE score) was a simple 5-gram count model with the Viterbi algorithm used for test time sequence prediction. This is a clear demonstration of the fact that the greedy evaluation used at test time for the Markovian models and the RNNs is far from optimal.

There was no clear difference between LSTM and GRU for this task.

From our work here, we have a couple more ideas for improving our word segmentation models. The first thing to note is that the count model is quite strong with the Viterbi algorithm, but we don’t perform any smoothing besides Laplace smoothing. Therefore, using Witten-Bell or Kneser-Ney smoothing may improve our model even further. It would also be interesting to try bidirectional LSTMs, or stacked LSTMs for experimentation.