HW4: Word Segmentation

Virgile Audi
vaudi@g.harvard.edu
Nicolas Drizard
nicolasdrizard@g.harvard.edu

April 1, 2016

1 Introduction

The goal of this assignement is to implement reccurent neural networks for a word segmentation task. The idea is to identify the spaces in sentence based on the previous characters only. This could be particularly helpful for processing languages written without spaces such as Korean or Spanish

2 Problem Description

The problem that needs to be solve in this homework is the following: given a sequence of characters, predict where to insert spaces to make a valid sentence. For instance, consider the following sequence of character:

I A M A STUDENT IN C S 2 8 7

the implemented algorithm should be capable of segmenting this sequence into valid words to give:

I am a student in CS 287

To solve this problem, we will train different language models including count-based models, basic neural networks, and recurrent neural networks, combined with two search algorithms to predict the right position for spaces, i.e. a greedy search algorithm and the Viturbi algorithm.

3 Model and Algorithms

3.1 Count-based Model

The first model is a count-based character n-gram model. The goal is to compute the probability of the newt word being a space:

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

This model is built by computing its MLE which gives:

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

where $c_i = w_{i-n+1}, \dots w_{i-1}$ is the context for the word w_i. We add a smoothing parameter $\alpha = 0.1$ just for the rare corner cases where the context was unseen (which is really rare in comparison to count-based word level models).

3.2 Neural Language Model

As a second baseline, we implemented a neural language model to predict whether the next character is a space or not. The model is similar to the Bengio model coded in HW3 but is adapted to characters. Similarly to what we did for word prediction, we imbed a window of characters in a higher dimension using a look-up table. We first apply a first linear model to the higher dimensional representation of the window of characters, followed by a hyperbolic tangent layer to extract non-linear features. A second linear layer is then applied followed by a softmax to get a probability distribution over the two possible outputs, i.e. a character or a space.
We can summarize the model in the following formula:

$$nnlm_1(x) = \tanh(\mathbf{xW} + \mathbf{b})\mathbf{W}' + \mathbf{b}'$$

where we recall:

- $\boldsymbol{x} \in \Re^{d_{in} \cdot d_{win}}$ is the concatenated character embeddings
- $\boldsymbol{W} \in \Re^{(d_{in} \cdot d_{win}) \times d_{hid}}$, and $\boldsymbol{b} \in \Re^{d_{hid}}$
- $\boldsymbol{W'} \in \Re^{d_{hid} \times 2}$, and $\boldsymbol{b'} \in \Re^2$.

3.3 Algorithm to generate spaces sequences

As mentioned in the problem description, in order to predict the position of a space, we will use two search algorithm. Both of these algorithm use the language models mentioned above to predict the next character or space given the prior context.

3.3.1 Greedy

The greedy algorithm implemented is an algorithm that chooses the locally optimum choice at every step in the sequence. This algorithm does not generally lead to a global maxium but has the advantage of being easilly implementable and efficient both in memory and complexity. The pseudo-code of the algorithm is presented below:

```
procedure GreedySearch
    s=0
    $c \in \mathcal{C}^{n+1}$
    $c_0 = \langle s \rangle$
    for i = 1 to n do
        Predict the distribution $\hat{\mathbf{y}}$ over the two classes given the previous context
        Pick the next class that maximises the distribution $c_i \leftarrow \arg\max_{c'_i} \hat{\mathbf{y}}(c_{i-1})_{c_i}$
        Update the score of the chain: $s + \log \hat{\mathbf{y}}(\mathbf{c_{i-1}})_{c_i}$
        Update the chain/context by adding a space or the following character
    return the chain and the score
```

3.3.2 Viterbi

The second search algorithm that we implemented is the dynamic programming algorithm named after Andrew Viterbi. The main difference with the greedy algorithm is that it evaluates at every step and for every previous state, the best possible next step. This would guarantee a solution closer to the true optimal solution. In our case of predicting character or space, the algorithm keeps track of the best sequences that could lead to a character or a space at step i-1, and then evaluates both path for both class, i.e. space to space, space to character, character to space and character to character, using the language models. It then keeps the path that has the highest score for each of the 2 states. The pseudo-code of the algorithm is given by:

```
procedure VITERBIWITHBP
    $\pi \in \mathbb{R}^{n+1 \times \mathcal{C}}$ initialized to $-\infty$
    $bp \in \mathcal{C}^{n \times \mathcal{C}}$ initialized to $\epsilon$
    $\pi[0, \langle s \rangle] = 0$
    for $i = 1$ to $n$ do
        for $c_{i-1} \in \mathcal{C}$ do
            compute $\hat{\boldsymbol{y}}(c_{i-1})$
            for $c_i \in \mathcal{C}$ do
                $score = \pi[i-1, c_{i-1}] + \log \hat{\boldsymbol{y}}(c_{i-1})_{c_i}$
                if $score > \pi[i, c_i]$ then
                    $\pi[i, c_i] = score$
                    $bp[i, c_i] = c_{i-1}$
    return sequence from $bp$
```

We implemented this algorithm for both bigram, and trigram models.

3.4 Recurrent Neural Networks

We implemented three different recurrent neural networks and benchmark their performance in our experiments. The main point is that we want to compute one output for each timestep and not only for the last one, that's why the generic structure of our networks is a tranducer.

Generic RNN Transducer The motivation is to maintain history in the model by the introduction of hidden states at each time steps (here each character of the input sequence). The model contains two main transformation: the transition function that define the hidden state given the current input x_i and the previous hidden state $_{-1}$ and the output layer producing the output at each timestep. We used Elman tanh layer for the output.

$$\begin{tikzcd}
\hat{\mathbf{y}}_1 & \hat{\mathbf{y}}_2 & \hat{\mathbf{y}}_3 \\
\mathbf{s}_1 \arrow[u, dash] \arrow[r, dash] & \mathbf{s}_2 \arrow[u, dash] \arrow[r, dash] & \mathbf{s}_3 \arrow[u, dash] \\
\mathbf{x}_1 \arrow[u, dash] & \mathbf{x}_2 \arrow[u, dash] & \mathbf{x}_3 \arrow[u, dash]
\end{tikzcd}$$

Figure 1: Transducer Architecture

Formally:

$$\hat{} = softmax(\mathbb{W} +)$$
$$= tanh([, _{-1}]\mathbb{W} +)$$

We used a batch version to learn the model and split the batched sequences in small chunks of characters of a given length to do the backpropagation to make it run faster. We explored different values for the two parameters length and batch size.

GRU This models introduces the gating operation that allows a vector to mask or gate . This operation is smoothed with a sigmoid: $t = \sigma(\mathbb{W} +)$. This operation is used to stop connection by applying the reset gate . This operation may be useful to avoid issue with the long sequence of gradients we need to compute in the backpropagation phase.

Formally:

$$\begin{aligned}
R(\mathbf{s}_{i-1}, \mathbf{x}_i) &= (1-\mathbf{t}) \odot \tilde{\mathbf{h}} + \mathbf{t} \odot \mathbf{s}_{i-1} \\
\tilde{\mathbf{h}} &= \tanh(\mathbf{x}\mathbf{W}^{x} + (\mathbf{r} \odot \mathbf{s}_{i-1})\mathbf{W}^{s} + \mathbf{b}) \\
\mathbf{r} &= \sigma(\mathbf{x}\mathbf{W}^{xr} + \mathbf{s}_{i-1}\mathbf{W}^{sr} + \mathbf{b}^{r}) \\
\mathbf{t} &= \sigma(\mathbf{x}\mathbf{W}^{xt} + \mathbf{s}_{i-1}\mathbf{W}^{st} + \mathbf{b}^{t}) \\
\mathbf{W}^{xt}, \mathbf{W}^{xr}, \mathbf{W}^{x} &\in \mathbb{R}^{d_{\text{in}} \times d_{\text{hid}}} \\
\mathbf{W}^{st}, \mathbf{W}^{sr}, \mathbf{W}^{s} &\in \mathbb{R}^{d_{\text{hid}} \times d_{\text{hid}}} \\
\mathbf{b}^{t}, \mathbf{b} &\in \mathbb{R}^{1 \times d_{\text{hid}}}
\end{aligned}$$

Figure 2: GRU equations

LSTM The long short term memory network uses also the gate idea with three gates: input, output and forget.

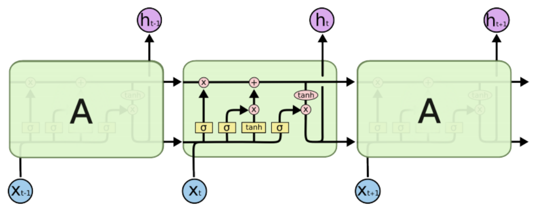

Figure 3: LSTM Architecture

4 Experiments

4.1 Count-based Model

This first approach relies on a window approach where we predict the next character given a fixed size of previous character. This size is the only parameter of the model. Then, we can apply the two algorithms described to predict a sequence given our trained model.

To evaluate the performance of the model gienve the size of the Ngram, we computed the perplexity of the training and validation data.

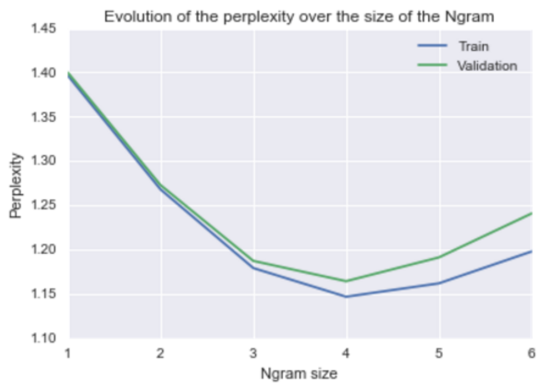

Figure 4: Perplexity evolution for the RNN

We observed an optimum of perplexity for the Ngram in both the validation and the train set. Then the steeper slope of the validation is due to overfitting. As a result, we sticked to this value for the model.

We implemented the greedy algorithm and the Viterbi one up to the trigram (so with a bigram as a context). Coding the Viterbi for larger Ngram size requires to cover more and more possibilities in our class C (given the position of spaces in the sequence).

4.2 Neural Language Model

Based on the results of the dynamic search on count-based models using bigram, we concluded that it was best to show results of the greedy algorithm with greater n-grams for the neural language model. In order to compare the results, we fixed the embedding size of the characters to 15, as well as the hidden dimension to 80 and the batch size to 20. We then train models for 3,4 and 5-grams, evaluate the loss on training, and use for validation the RMSE of the number of spaces predicted on each sentence of the validation set.

We present the results:

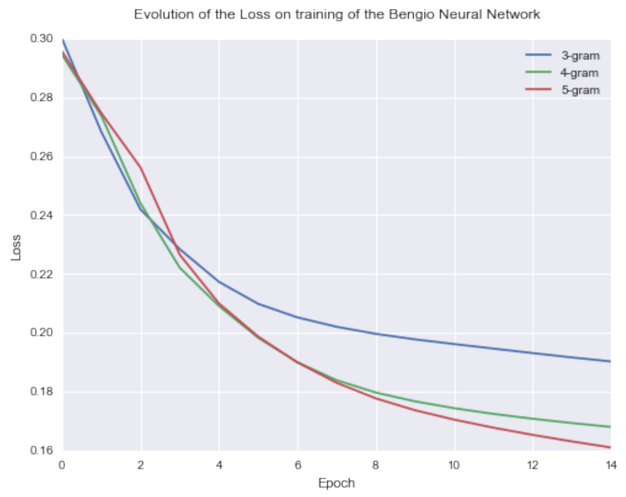

Figure 5: Training Loss

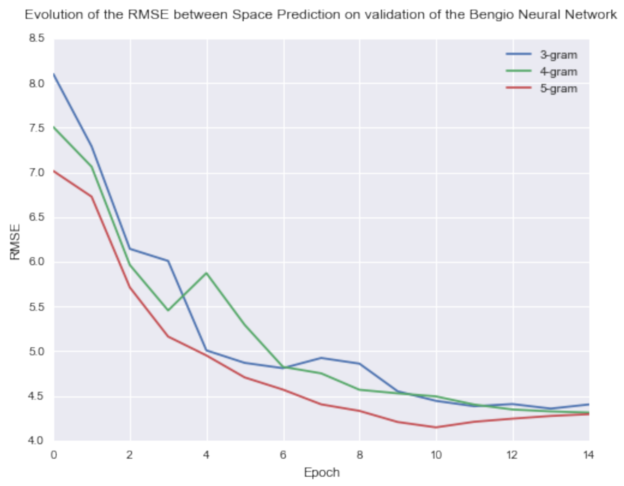

Figure 6: RMSE on Validation set

As expected, performance increases with the size of the n-grams. We then tested the impact of the embedding size.

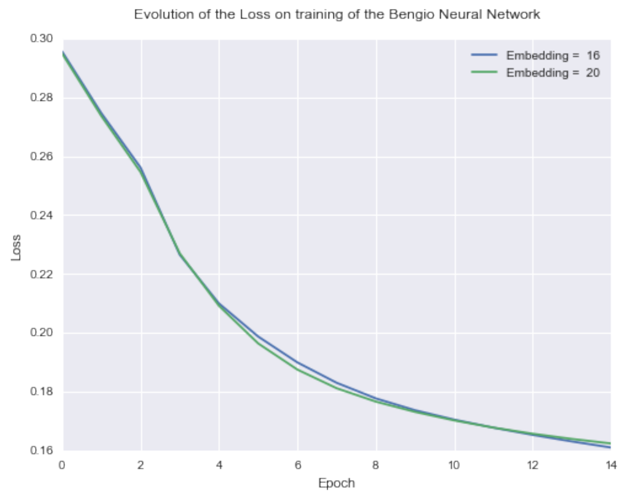

Figure 7: Training Loss

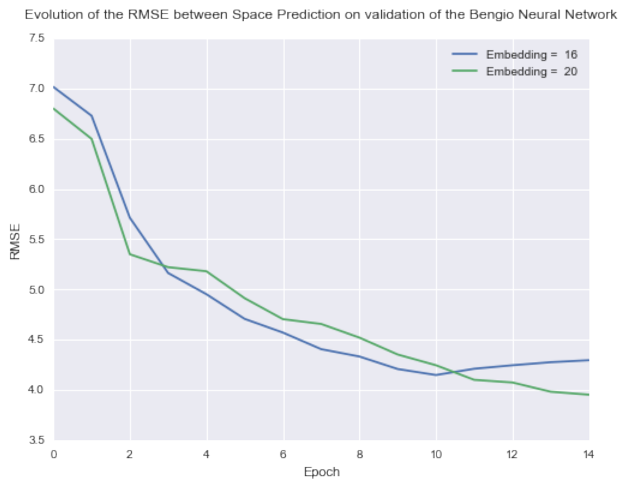

Figure 8: RMSE on Validation set

If the losses on training are very similar, we observed that greater embedding dimension yield better results on the validation. We therefore submitted to Kaggle, results using the latter model trained on 20 epochs and obtained:

$$RMSE_{nn} = \sqrt{13.37} = 3.65$$

We then experimented with this model by assignment a space as the next prediction by using a threshhold instead of using argmax prediction.

4.3 Recurrent Neural Networks

For the three recurrent networks implemented, we have different parameters to take into account:

- batch size l
- length of sequences b
- embedding dimension emb
- number of epochs nEpochs

Choosing the right batch-size seems to be a tradeoff between performance and running time, a smaller one provides smaller perplexity but takes more time to run. The length of the sequence seems to provide good result when in the interval $[30, .., 50]$ without significant peak so we kept values in this zone. We set the embedding dimension to 20 for the experiments with some prio explorations also.

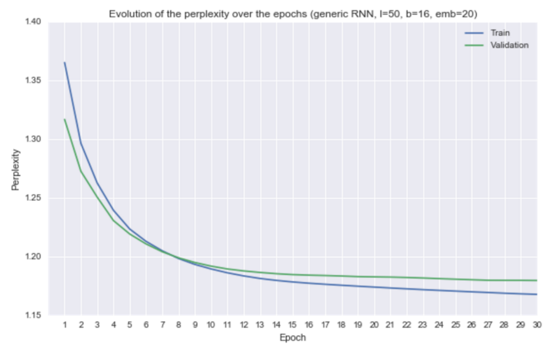

Figure 9: Perplexity evolution for the RNN

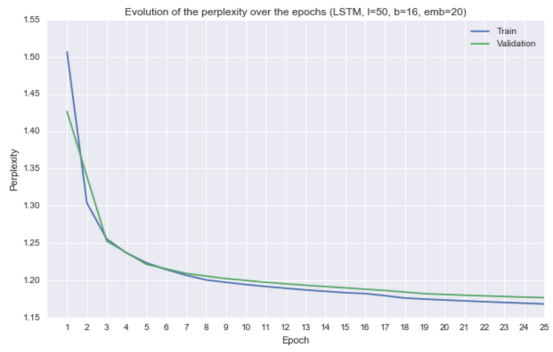

Figure 10: Perplexity evolution for the LSTM

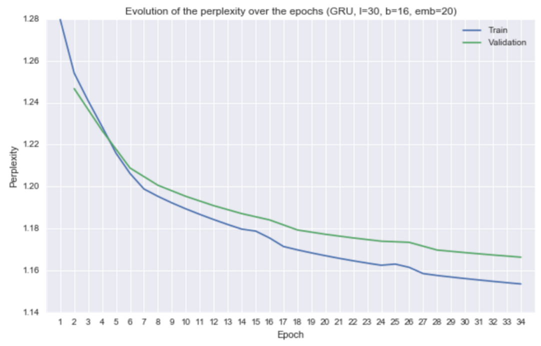

Figure 11: Perplexity evolution for the GRU

4.4 Model performance summary

5 Conclusion

End the write-up with a very short recap of the main experiments and the main results. Describe any challenges you may have faced, and what could have been improved in the model.

References