

# HW4: Word Segmentation

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

April 1, 2016

## 1 Introduction

The goal of this assignment is to implement recurrent 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:

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

This model is built by computing its MLE which gives:

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

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:

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

### 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 maximum but has the advantage of being easily implementable and efficient both in memory and complexity. The pseudo-code of the algorithm is presented below:

- 1: **procedure** GREEDYSEARCH
- 2:    $s=0$
- 3:    $c \in \mathcal{C}^{n+1}$
- 4:    $c_0 = \langle s \rangle$
- 5:   **for**  $i = 1$  to  $n$  **do**
- 6:     Predict the distribution  $\hat{\mathbf{y}}$  over the two classes given the previous context

- 7:        Pick the next class that maximises the distribution  $c_i \leftarrow \arg \max_{c'_i} \hat{\mathbf{y}}(c_{i-1})_{c_i}$
  - 8:        Update the score of the chain:  $s + \log \hat{\mathbf{y}}(c_{i-1})_{c_i}$
  - 9:        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  $\hat{\mathbf{y}}(c_{i-1})$ 
      for  $c_i \in \mathcal{C}$  do
         $score = \pi[i-1, c_{i-1}] + \log \hat{\mathbf{y}}(c_{i-1})_{c_i}$ 
        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 transducer.

**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.

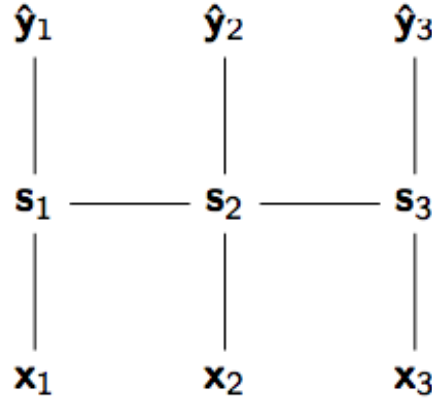


Figure 1: Transducer Architecture

Formally:

$$\begin{aligned}\hat{y}_t &= \text{softmax}(\mathbf{W} + \mathbf{h}_t) \\ \mathbf{h}_t &= \tanh([\mathbf{I}, -\mathbf{1}]\mathbf{W} + \mathbf{h}_{t-1})\end{aligned}$$

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 model introduces the gating operation that allows a vector to mask or gate. This operation is smoothed with a sigmoid:  $t = \sigma(\mathbf{W} + \mathbf{h})$ . 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:

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

Figure 2: GRU equations

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

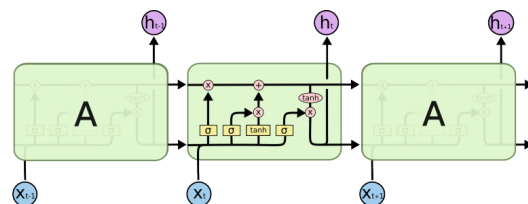


Figure 3: LSTM Architecture

Formally:

$$\begin{aligned}
 R(\mathbf{s}_{i-1}, \mathbf{x}_i) &= [\mathbf{c}_i, \mathbf{h}_i] \\
 \mathbf{c}_i &= \mathbf{j} \odot \mathbf{i} + \mathbf{f} \odot \mathbf{c}_{i-1} \\
 \mathbf{h}_i &= \tanh(\mathbf{c}_i) \odot \mathbf{o} \\
 \mathbf{i} &= \tanh(\mathbf{x}\mathbf{W}^{xi} + \mathbf{h}_{i-1}\mathbf{W}^{hi} + \mathbf{b}^i) \\
 \mathbf{j} &= \sigma(\mathbf{x}\mathbf{W}^{xj} + \mathbf{h}_{i-1}\mathbf{W}^{hj} + \mathbf{b}^j) \\
 \mathbf{f} &= \sigma(\mathbf{x}\mathbf{W}^{xf} + \mathbf{h}_{i-1}\mathbf{W}^{hf} + \mathbf{b}^f) \\
 \mathbf{o} &= \tanh(\mathbf{x}\mathbf{W}^{xo} + \mathbf{h}_{i-1}\mathbf{W}^{ho} + \mathbf{b}^o)
 \end{aligned}$$

Figure 4: Perplexity evolution for the GRU

## 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 given the size of the Ngram, we computed the perplexity of the training and validation data.



Figure 5: 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 stuck to this value for the model.

We implemented the greedy algorithm and the Viterbi one up to the trigram (so with a bi-gram 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:

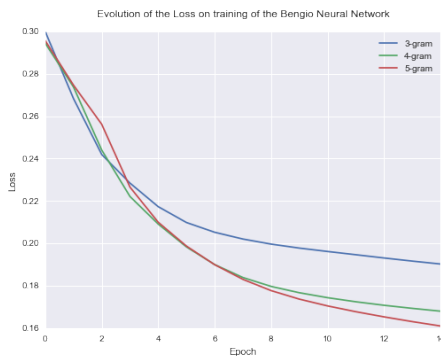


Figure 6: Training Loss

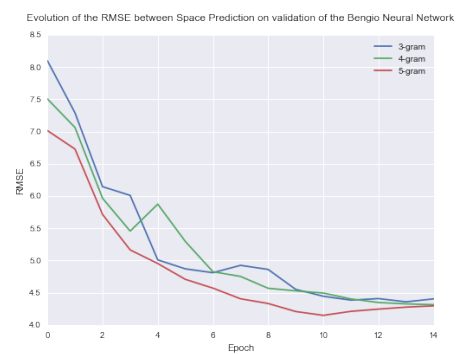


Figure 7: RMSE on Validation set

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

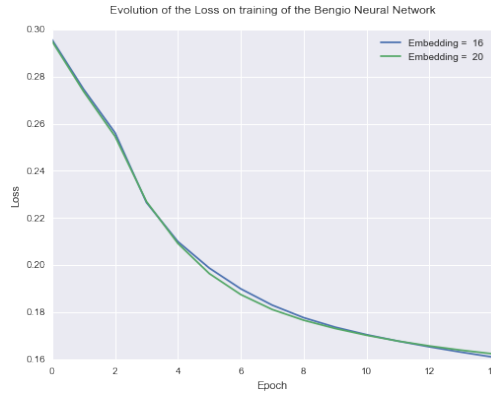


Figure 8: Training Loss

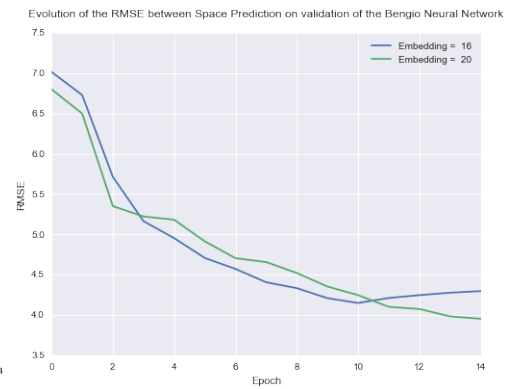


Figure 9: RMSE on Validation set

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 threshold instead of using argmax prediction, which is equivalent to a cutoff of 0.5. But since, the ratio of spaces to character is relatively small, we might get better results by setting a cutoff smaller than 0.5, i.e. if the probability of generating a space is above that cutoff, which is itself smaller than 0.5, we assign a space as the next element. We present results for cutoffs ranging from 0.2 to 0.5:

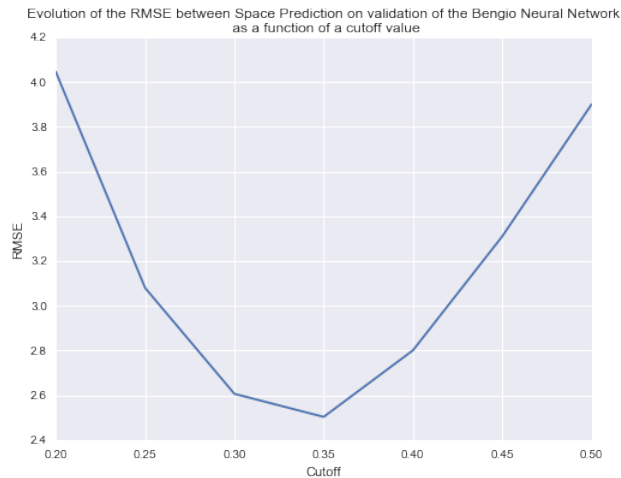


Figure 10: RMSE as a function of the cutoff

By changing the cutoff, we improve drastically the results with an RMSE value that drops from 3.95 when using argmax prediction to 2.55 with a cutoff of 0.35 (35% improvement).

### 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  $[30, \dots, 50]$  without significant peak so we kept values in this zone. We set the embedding dimension to 20 for the experiments with some prior explorations also.

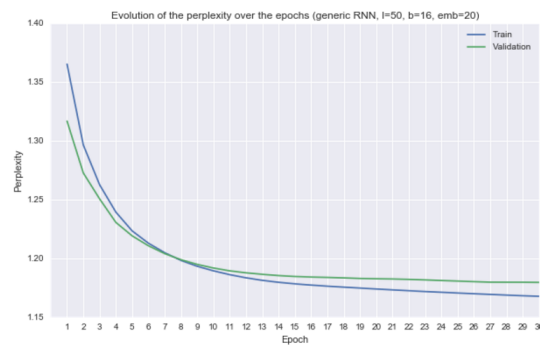


Figure 11: Perplexity evolution for the RNN

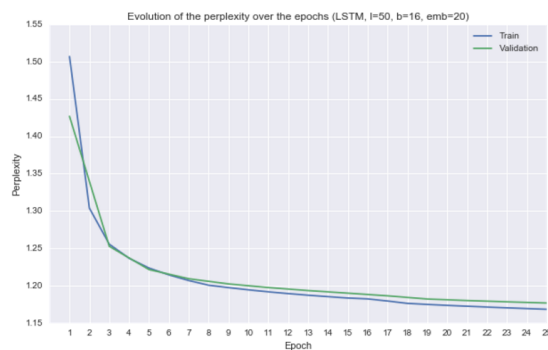


Figure 12: Perplexity evolution for the LSTM



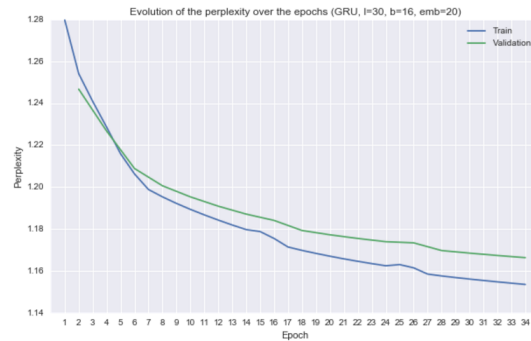


Figure 13: Perplexity evolution for the GRU

The best results on the Kaggle were provided with the GRU after a large number of epochs (around 100).

#### 4.4 Model performance summary

TABLE TO SUMMARIZE RESULT (perptrain, perpvalid, pred on kaggle)

## 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

# Appendices

## Preprocessing:

```

1 import numpy as np
2 import h5py
3 import argparse
4 import sys
5 import re
6 import codecs
7
8 from collections import Counter
9
10
11 FILE_PATHS = ("data/train_chars.txt",

```

```

12         "data/valid_chars.txt",
13         "data/test_chars.txt")
14
15
16 def get_input(filename, n, char_to_ind=None):
17     # Contain the list of characters indices in the data
18     # initialized with a padding
19     if n > 2:
20         input_data = [2]*(n-2)
21     else:
22         input_data = []
23     if char_to_ind is None:
24         # Map each character to an index with
25         # Index of <space> set to 1
26         char_to_ind = {'<space>': 1, '</s>': 2}
27         count = 3
28     with open(filename, 'r') as f:
29         # Loop to index the char and store them inside the input
30         for line in f:
31             for c in line[:-1].split(' '):
32                 # Input data
33                 if c in char_to_ind:
34                     input_data.append(char_to_ind[c])
35                 else:
36                     char_to_ind[c] = count
37                     count += 1
38                     input_data.append(char_to_ind[c])
39     return input_data, char_to_ind
40
41
42 def build_train_data(input_data, n):
43     # Build the input matrix: (num_records, n-1)
44     # and the output vector (num_records,1)
45     # which stores the output for the given (n-1)gram
46     input_matrix = np.zeros((len(input_data)-n, n-1))
47     output_matrix = np.zeros(len(input_data)-n)
48     for i in xrange(len(input_data)-n):
49         # Countext is a (n-1)gram
50         w = input_data[i:i+(n-1)]
51         input_matrix[i, :] = w
52         output_matrix[i] = (1 if input_data[i+(n-1)] == 1 else 2)
53     return input_matrix, output_matrix
54
55
56 def build_count_matrix(input_matrix, output_matrix, n):

```

```

57     count_matrix_raw = np.concatenate((input_matrix ,
58                                         output_matrix.reshape(
                                                output_matrix.shape[0], 1)),
                                                axis=1)

59
60     num_rows = len(set([tuple(s) for s in input_matrix]))
61     count = Counter([tuple(s) for s in count_matrix_raw])
62
63     # count matrix: (num_(n-1grams, 2))
64     F = np.zeros((num_rows, n + 1))
65     gram_to_ind = {}
66     i = 0
67     for k, v in count.iteritems():
68         gram = k[:n-1]
69         if gram not in gram_to_ind:
70             gram_to_ind[gram] = i
71             i += 1
72         F[gram_to_ind[gram], n-1 + int(k[-1]) - 1] = v
73         F[gram_to_ind[gram], :n-1] = list(gram)
74
75     return F
76
77
78 def main(arguments):
79     global args
80     parser = argparse.ArgumentParser(
81         description=__doc__ ,
82         formatter_class=argparse.RawDescriptionHelpFormatter)
83
84     parser.add_argument('--N', default=2, type=int, help='Ngram size ')
85     args = parser.parse_args(arguments)
86     N = args.N
87
88     train, valid, test = FILE_PATHS
89
90     # Train
91     input_data_train, char_to_ind = get_input(train, N)
92     input_matrix_train, output_matrix_train = build_train_data(
93         input_data_train, N)
94     F_train = build_count_matrix(input_matrix_train,
95                                 output_matrix_train, N)
96
97     # Valid
98     input_data_valid, char_to_ind = get_input(valid, N, char_to_ind)

```

```

98     input_data_valid_nospace = filter(lambda a: a != 1,
    input_data_valid)
99
100    # Test
101    input_data_test, char_to_ind = get_input(test, N, char_to_ind)
102
103    filename = 'data_preprocessed/' + str(N) + '-grams.hdf5'
104    with h5py.File(filename, "w") as f:
105        # Stores a matrix (num_records, N-1) with at each row
106        # the (N-1) grams appearing in the input data
107        f['input_matrix_train'] = input_matrix_train
108        f['F_train'] = F_train
109        # Vector (num_records) storing the class of the next word
110        # after the (N-1) gram stored at the same index in input_matrix
111        # 1 is space; 2 is character
112        f['output_matrix_train'] = output_matrix_train
113        # Stores the list of consecutives character (or space) as their
114        # index from the mapping char_to_ind
115        f['input_data_train'] = np.array(input_data_train)
116        f['input_data_valid'] = np.array(input_data_valid)
117        f['input_data_valid_nospace'] = np.array(
            input_data_valid_nospace)
118        f['input_data_test'] = np.array(input_data_test)
119
120
121    if __name__ == '__main__':
122        sys.exit(main(sys.argv[1:]))

```

## Helper:

```

1  function get_kaggle_format(predictions_test, N)
2      — Counting sentences
3      local num_sentence = 0
4      for i=N-1,predictions_test:size(1) do
5          if predictions_test[i] == 2 then
6              num_sentence = num_sentence + 1
7          end
8      end
9
10     — Counting space per sentence
11     local num_spaces = torch.DoubleTensor(num_sentence,2)
12     local row = 1
13     local count_space = 0
14     for i=N-1,predictions_test:size(1) do
15         if predictions_test[i] == 2 then
16             num_spaces[{row, 1}] = row

```

```

17         num_spaces[{row, 2}] = count_space
18         count_space = 0
19         row = row + 1
20     elseif predictions_test[i] == 1 then
21         count_space = count_space + 1
22     end
23 end
24 return num_spaces
25 end
26
27 function compute_rmse(true_kaggle, pred_kaggle)
28     local rmse = 0
29     for i=1,true_kaggle:size(1) do
30         rmse = rmse + math.pow(true_kaggle[{i,2}] - pred_kaggle[{i,2}],
31                                 2)
32     end
33     return(math.sqrt(rmse/ true_kaggle:size(1)))
34 end

```

## Count-Based Models:

```

1  — Documentation:
2  — ——— How to call it from the command line?
3  — For example:
4  — $ th count_based.lua -N 5
5  — Other argument possible (see below)
6  —
7  — ——— Is there an Output?
8  — By default, the predictions on the test set are saved in hdf5 format
   as classifier .. opt.f
9
10 — Only requirements allowed
11 require("hdf5")
12 require 'helper.lua';
13
14 cmd = torch.CmdLine()
15
16 — Cmd Args
17 cmd:option('-N', 2, 'Ngram size for the input')
18 cmd:option('-algo', 'greedy', 'Algorithm to use: either greedy or
   viterbi')
19 cmd:option('-f', 'pred_test.f5', 'File name for the predictions on the
   test')
20
21 — Build the mapping from (N-1)gram to row index
22 — and the count matrix F_count: (num_context, 2)

```

```

23 function get_F_count(F, N)
24     local ngram_to_ind = {}
25     local key
26     for i=1,F:size(1) do
27         key = tostring(F[{i,1}])
28         — Building key
29         for k = 2,N-1 do
30             key = key .. '-' .. tostring(F[{i,k}])
31         end
32         ngram_to_ind[key] = i
33     end
34     return F:narrow(2,N,2) , ngram_to_ind
35 end
36
37 — Compute proba distribution over (space, char) for the context
38 — F is here the count matrix (num_context, 2)
39 function compute_count_based_probability(context, F_count, ngram_to_ind
    , alpha)
40     local probability = torch.zeros(2)
41     — Building key, ie (N-1)gram (from i to i+(N-2))
42     local key = tostring(context[1])
43     for k = 2,context:size(1) do
44         key = key .. '-' .. tostring(context[k])
45     end
46     — If (N-1)gram never seen, prior distribution
47     if (ngram_to_ind[key] ~= nil) then
48         — index of the current (n-1)gram in the F matrix
49         local index = ngram_to_ind[key]
50         probability:copy(F_count:narrow(1,index,1))
51         — Adding smoothing
52         probability:add(alpha)
53     — Case unseen context
54     else
55         — Prior
56         probability:copy(torch.DoubleTensor({ F_count:narrow(2,1,1):sum
            (), F_count:narrow(2,2,1):sum() }))
57     end
58     return probability:div(probability:sum())
59 end
60
61 — Compute perplexity on entry with space
62 function compute_perplexity(gram_input, F_count, ngram_to_ind, N)
63     local perp = 0
64     local context = torch.zeros(N-1)
65     local probability = torch.zeros(2)

```

```

66 — Do not predict for the last char
67 —for i=1,gram_input:size(1)-N do
68 local size=gram_input:size(1) - (N-1)
69 for i=1,size do
70     context:copy(gram_input:narrow(1,i,N-1))
71     — Line where the model appears
72     probability:copy(compute_count_based_probability(context,
73         F_count, ngram_to_ind, 1))
74     if gram_input[i+(N-1)] == 1 then
75         right_proba = probability[1]
76         —print('space')
77         —print(right_proba)
78     else
79         right_proba = probability[2]
80     end
81     perp = perp + math.log(right_proba)
82 end
83 perp = math.exp(-perp/size)
84 —perp = math.exp(-perp/(gram_input:size(1)-N))
85 return perp
86 end
87 — Greedy algorithm to predict a sequence from gram_input with a count
88 — based probability model
89 function predict_count_based_greedy(gram_input, F_count, ngram_to_ind,
90     N)
91     — Next Position to fill in predictions
92     local position = N
93     — We allocate the maximum of memory that could be needed
94     — Default value is -1 (to know where predictions end afterwards)
95     local predictions = torch.ones(2*(gram_input:size(1) - N)):mul(-1)
96     — Copy the first (N-1) gram
97     predictions:narrow(1,1,N-1):copy(gram_input:narrow(1,1,N-1))
98     local probability = torch.zeros(2)
99     local context = torch.zeros(N-1)
100
101     — Build mapping
102     for i=1,gram_input:size(1)-N do
103         — Compute proba for next char
104         context:copy(predictions:narrow(1,position-(N-1),N-1))
105         — Line where the model appears
106         probability:copy(compute_count_based_probability(context,
107             F_count, ngram_to_ind, 1))
108         m,a = probability:max(1)

```

```

108     — Case space predicted
109     if (a[1] == 1) then
110         predictions[position] = 1
111         position = position + 1
112     end
113
114     — Copying next character
115     predictions[position] = gram_input[i+N-1]
116     position = position + 1
117 end
118 — Adding last character (</s>)
119 predictions[position] = gram_input[gram_input:size(1)]
120 — Cutting the output
121 return predictions:narrow(1,1,position)
122 end
123
124 — Viterbi algorithm to predict a sequence from gram_input with a count
125 — based probability model
126 — pi matrix format (col1: space; col2: char)
127 function predict_count_based_viterbi(gram_input, F_count, ngram_to_ind,
    N)
128     — Backpointer
129     local score
130     local bp = torch.zeros(gram_input:size(1) + 1, 2)
131     local context = torch.DoubleTensor(1)
132     local y_hat = torch.DoubleTensor(2)
133     local pi = torch.ones(gram_input:size(1) + 1, 2):mul(-9999)
134     — Initialization
135     pi[{1,1}] = 0
136     — i is shifted
137     for i=2,gram_input:size(1)+1 do
138         for c_prev=1,2 do
139             — Precompute y_hat(c_prev)
140             if c_prev == 1 then
141                 context[1] = c_prev
142             else
143                 context[1] = gram_input[i-1]
144             end
145             — Line where the model appears
146             y_hat:copy(compute_probability(context, F_count,
                ngram_to_ind, 1))
147
148             for c_current=1,2 do
149                 score = pi[{i-1, c_prev}] + math.log(y_hat[c_current])
150                 if score > pi[{i, c_current}] then

```



```

151             pi[{i, c_current}] = score
152             bp[{i, c_current}] = c_prev
153         end
154     end
155 end
156 end
157 return pi, bp
158 end
159
160 — Building the sequences from the backpointer
161 function build_sequences_from_bp(bp, gram_input)
162     local predictions = torch.DoubleTensor(2*gram_input:size(1))
163     — Next position to fill in predictions (have to do it backward)
164     local position = 2*gram_input:size(1)
165     local col = 2
166     — Loop until the 3rd position (because 2nd is the first one, could
        be set by hand)
167     for i=bp:size(1),3,-1 do
168         — coming from a space
169         if bp[i][col] == 1 then
170             predictions[position] = 1
171             position = position - 1
172             col = 1
173         else
174             col = 2
175         end
176         — index i is shifted of 1 wrt local index in gram_input
177         predictions[position] = gram_input[i-1]
178         position = position - 1
179     end
180     — Beginning of gram_input set
181     predictions[position] = gram_input[1]
182     position = position - 1
183
184     return predictions:narrow(1,position+1,predictions:size(1)-position
        )
185 end
186
187 — Viterbi trigram
188 function predict_count_based_viterbi_trigram(gram_input, F_count,
        ngram_to_ind, N)
189     — Backpointer
190     local score
191     local bp = torch.zeros(gram_input:size(1) + 1, 3)
192     local context = torch.DoubleTensor(2)

```

```

193     local y_hat = torch.DoubleTensor(2)
194     — pi is built as ('char-space', 'char-char', 'space-char')
195     — corresponding index in the context
196     local pi = torch.ones(gram_input:size(1) + 1, 3):mul(-999999999)
197     — Initialization
198     pi[{2,1}] = 0
199     —pi[{2,2}] = 0
200     —pi[{2,3}] = 0
201     — We need to start at the first trigram
202     for i=3,gram_input:size(1)+1 do
203         for c_prev =1,3 do
204             — Precompute y_hat(c_prev)
205             if c_prev == 1 then
206                 context[1] = gram_input[i-2]
207                 context[2] = 1
208             elseif c_prev == 2 then
209                 context[1] = gram_input[i-2]
210                 context[2] = gram_input[i-1]
211             else
212                 context[1] = 1
213                 context[2] = gram_input[i-1]
214             end
215             — Line where the model appears
216             y_hat:copy(compute_probability(context, F_count,
217                                     ngram_to_ind, 1))
218
219             — cannot have 2 spaces in a row: from 1 goes to 3
220             necessarily
221             if c_prev == 1 then
222                 pi[{i, 3}] = pi[{i-1, c_prev}] + math.log(y_hat[2])
223                 bp[{i, 3}] = c_prev
224             else
225                 — last char is necessarily 'char' so
226                 — 1: space predicted (ie 'char-space')
227                 — 2: char predicted (ie 'char-char')
228                 for c_current =1,2 do
229                     score = pi[{i-1, c_prev}] + math.log(y_hat[
230                         c_current])
231                     if score > pi[{i, c_current}] then
232                         pi[{i, c_current}] = score
233                         bp[{i, c_current}] = c_prev
234                     end
235                 end
236             end
237         end
238     end
239 end

```

```

235     end
236     return pi, bp
237 end
238
239 — Building the sequences from the backpointer
240 — We start the sequence by the ('char'-'char') configuration
241 — as we know it's the only one possible
242 function build_sequences_from_bp_trigram(bp, gram_input)
243     local predictions = torch.DoubleTensor(2*gram_input:size(1))
244     — Next position to fill in predictions (have to do it backward)
245     local position = 2*gram_input:size(1)
246     local col = 2
247     — Loop until the 4th position
248     for i=bp:size(1),4,-1 do
249         — coming from a space
250         if bp[i][col] == 1 then
251             predictions[position] = 1
252             position = position - 1
253         end
254         col = bp[i][col]
255         — index i is shifted of 1 wrt local index in gram_input
256         predictions[position] = gram_input[i-1]
257         position = position - 1
258     end
259     — Beginning of gram_input set
260     predictions[position] = gram_input[2]
261     position = position - 1
262     predictions[position] = gram_input[1]
263     position = position - 1
264
265     return predictions:narrow(1,position+1,predictions:size(1)-position
266         )
267 end
268
269 function main()
270     — Parse input params
271     opt = cmd:parse(arg)
272     N = opt.N
273     algo = opt.algo
274
275     — Reading file
276     local file = hdf5.open('data_preprocessed / '..tostring(N)..' - grams.
277         hdf5', 'r')
278     data = file:all()
279     file:close()

```

```

278
279 F_train = data['F_train']
280 input_data_valid = data['input_data_valid']
281 input_data_train = data['input_data_train']
282 input_data_test = data['input_data_test']
283 input_data_valid_nospace = data['input_data_valid_nospace']
284
285 — Building the model
286 F_count, ngram_to_ind = get_F_count(F_train, N)
287 print('Ngram size '..tostring(N))
288 print('Train Perplexity')
289 print(compute_perplexity(input_data_train, F_count, ngram_to_ind, N
    ))
290 print('Valid Perplexity')
291 print(compute_perplexity(input_data_valid, F_count, ngram_to_ind, N
    ))
292
293 — Prediction
294 if (algo == 'greedy') then
295     predictions_test = predict_count_based_greedy(input_data_test,
        F_count, ngram_to_ind, N)
296 elseif (algo == 'viterbi') then
297     if (N == 2) then
298         pi, bp = predict_count_based_viterbi(input_data_test,
            F_count, ngram_to_ind, N)
299         predictions_test = build_sequences_from_bp(bp,
            input_data_test)
300     elseif (N == 3) then
301         pi_tri, bp_tri = predict_count_based_viterbi_trigram(
            input_data_test, F_count, ngram_to_ind, N)
302         predictions_test = build_sequences_from_bp_trigram(bp_tri,
            input_data_test)
303     else
304         error("invalid N for Viterbi")
305     end
306 else
307     error("invalid algorithm input")
308 end
309
310 — Kaggle format
311 num_spaces = get_kaggle_format(predictions_test, N)
312
313 — Saving the Kaggle format output
314 myFile = hdf5.open('submission/'..'opt.f', 'w')
315 myFile:write('num_spaces', num_spaces)

```

```

316     myFile: close ()
317 end
318
319 main ()

```

## NNLM:

```

1  require 'hdf5';
2  require 'nn';
3  require 'helper.lua';
4
5  cmd = torch.CmdLine()
6
7  — Cmd Args
8  cmd:option('-N', 5, 'Ngram size for the input')
9  cmd:option('--embed', 16, 'Embedding size of characters')
10 cmd:option('--hid', 80, 'Hidden layer dimension')
11 cmd:option('--eta', 0.01, 'Learning rate')
12 cmd:option('--batch', 10, 'Batchsize')
13 cmd:option('--Ne', 20, 'Number of epochs')
14 cmd:option('--algo', 'greedy', 'Algorithm to use: either greedy or
    viterbi')
15 cmd:option('-f', 'pred_test.f5', 'File name for the predictions on the
    test')
16
17 function build_model(dwin, nchar, nclass, hid1, hid2)
18     — Model with skip layer from Bengio, standards parameters
19     — should be:
20     — dwin = 5
21     — hid1 = 30
22     — hid2 = 100
23
24     — To store the whole model
25     local dnnlm = nn.Sequential()
26
27     — Layer to embedd (and put the words along the window into one
        vector)
28     local LT = nn.Sequential()
29     local LT_ = nn.LookupTable(nchar, hid1)
30     LT:add(LT_)
31     LT:add(nn.View(-1, hid1*dwin))
32
33     dnnlm:add(LT)
34
35     local concat = nn.ConcatTable()
36

```

```

37     local lin_tanh = nn.Sequential()
38     lin_tanh:add(nn.Linear(hid1*dwin, hid2))
39     lin_tanh:add(nn.Tanh())
40
41     local id = nn.Identity()
42
43     concat:add(lin_tanh)
44     concat:add(id)
45
46     dnnlm:add(concat)
47     dnnlm:add(nn.JoinTable(2))
48     dnnlm:add(nn.Linear(hid1*dwin + hid2, nclass))
49     dnnlm:add(nn.LogSoftMax())
50
51     — Loss
52     local criterion = nn.ClassNLLCriterion()
53
54     return dnnlm, criterion
55 end
56
57
58 function train_model(train_input, train_output, dnnlm, criterion, dwin,
59     nclass, eta, nEpochs, batchSize)
60     — Train the model with a mini batch SGD
61     — standard parameters are
62     — nEpochs = 1
63     — batchSize = 32
64     — eta = 0.01
65
66     — To store the loss
67     local av_L = 0
68
69     — Memory allocation
70     local inputs_batch = torch.DoubleTensor(batchSize, dwin)
71     local targets_batch = torch.DoubleTensor(batchSize)
72     local outputs = torch.DoubleTensor(batchSize, nclass)
73     local df_do = torch.DoubleTensor(batchSize, nclass)
74
75     for i = 1, nEpochs do
76         — timing the epoch
77         local timer = torch.Timer()
78
79         av_L = 0
80
81         — max renorm of the lookup table

```

```

81     dnnlm: get(1): get(1).weight: renorm(2,1,1)
82
83     — mini batch loop
84     for t = 1, train_input:size(1), batchSize do
85         — Mini batch data
86         local current_batch_size = math.min(batchSize, train_input:
            size(1)-t)
87         inputs_batch:narrow(1,1,current_batch_size):copy(
            train_input:narrow(1,t,current_batch_size))
88         targets_batch:narrow(1,1,current_batch_size):copy(
            train_output:narrow(1,t,current_batch_size))
89
90         — reset gradients
91         dnnlm:zeroGradParameters()
92         —gradParameters:zero()
93
94         — Forward pass (selection of inputs_batch in case the
            batch is not full, ie last batch)
95         outputs:narrow(1,1,current_batch_size):copy(dnnlm:forward(
            inputs_batch:narrow(1,1,current_batch_size)))
96
97         — Average loss computation
98         local f = criterion:forward(outputs:narrow(1,1,
            current_batch_size), targets_batch:narrow(1,1,
            current_batch_size))
99         av_L = av_L +f
100
101         — Backward pass
102         df_do:narrow(1,1,current_batch_size):copy(criterion:
            backward(outputs:narrow(1,1,current_batch_size),
            targets_batch:narrow(1,1,current_batch_size)))
103         dnnlm:backward(inputs_batch:narrow(1,1,current_batch_size),
            df_do:narrow(1,1,current_batch_size))
104         dnnlm:updateParameters(eta)
105
106     end
107
108     print('Epoch '..i..'': '..timer:time().real)
109     print('Average Loss: '..av_L/math.floor(train_input:size(1)/
        batchSize))
110
111 end
112
113 end
114

```

```

115
116 — Compute perplexity on entry with space
117 function compute_perplexity(gram_input, nnlm, N)
118     local perp = 0
119     local context = torch.zeros(N-1)
120     local probability = torch.zeros(2)
121     — Do not predict for the last char
122     —for i=1,gram_input:size(1)-N do
123     local size=gram_input:size(1) - (N-1)
124     for i=1,size do
125         context:copy(gram_input:narrow(1,i,N-1))
126         — Line where the model appears
127         probability:copy(nnlm:forward(context))
128         if gram_input[i+(N-1)] == 1 then
129             right_proba = probability[1]
130         else
131             right_proba = probability[2]
132         end
133         perp = perp + right_proba
134     end
135     perp = math.exp(-perp/size)
136     return perp
137 end
138
139
140 — Greedy algorithm to predict a sequence from gram_input with a count
141 — based probability model
142 function predict_NN_greedy(gram_input, nnlm, N)
143     — Next Position to fill in predictions
144     local position = N
145     — We allocate the maximum of memory that could be needed
146     — Default value is -1 (to know where predictions end afterwards)
147     local predictions = torch.ones(2*(gram_input:size(1) - N)):mul(-1)
148     — Copy the first (N-1) gram
149     predictions:narrow(1,1,N-1):copy(gram_input:narrow(1,1,N-1))
150     local probability = torch.zeros(2)
151     local context = torch.zeros(N-1)
152
153     — Build mapping
154     for i=1,gram_input:size(1)-N do
155         — Compute proba for next char
156         context:copy(predictions:narrow(1,position-(N-1),N-1))
157         — Line where the model appears
158         probability:copy(nnlm:forward(context))
159         m,a = probability:max(1)

```



```

160
161     — Case space predicted
162     if (a[1] == 1) then
163         predictions[position] = 1
164         position = position +1
165     end
166
167     — Copying next character
168     predictions[position] = gram_input[i+N-1]
169     position = position +1
170 end
171 — Adding last character (</s>)
172 predictions[position] = gram_input[gram_input:size(1)]
173 — Cutting the output
174 return predictions:narrow(1,1,position)
175 end
176
177 function predict_NN_greedy(gram_input, nnlm, N)
178     — Next Position to fill in predictions
179     local position = N
180     — We allocate the maximum of memory that could be needed
181     — Default value is -1 (to know where predictions end afterwards)
182     local predictions = torch.ones(2*(gram_input:size(1) - N)):mul(-1)
183     — Copy the first (N-1) gram
184     predictions:narrow(1,1,N-1):copy(gram_input:narrow(1,1,N-1))
185     local probability = torch.zeros(2)
186     local context = torch.zeros(N-1)
187
188     — Build mapping
189     for i=1,gram_input:size(1)-N do
190         — Compute proba for next char
191         context:copy(predictions:narrow(1,position-(N-1),N-1))
192         — Line where the model appears
193         probability:copy(nnlm:forward(context))
194         m,a = probability:max(1)
195
196         — Case space predicted
197         if (a[1] == 1) then
198             predictions[position] = 1
199             position = position +1
200         end
201
202         — Copying next character
203         predictions[position] = gram_input[i+N-1]
204         position = position +1

```

```

205     end
206     — Adding last character (</s>)
207     predictions[position] = gram_input[gram_input:size(1)]
208     — Cutting the output
209     return predictions:narrow(1,1,position)
210 end
211
212 function predict_NN_greedy_cutoff(gram_input, nnlm, N, cut)
213     — Next Position to fill in predictions
214     local position = N
215     — We allocate the maximum of memory that could be needed
216     — Default value is -1 (to know where predictions end afterwards)
217     local predictions = torch.ones(2*(gram_input:size(1) - N)):mul(-1)
218     — Copy the first (N-1) gram
219     predictions:narrow(1,1,N-1):copy(gram_input:narrow(1,1,N-1))
220     local probability = torch.zeros(2)
221     local context = torch.zeros(N-1)
222
223     — Build mapping
224     for i=1,gram_input:size(1)-N do
225         — Compute proba for next char
226         context:copy(predictions:narrow(1,position-(N-1),N-1))
227         — Line where the model appears
228         probability:copy(nnlm:forward(context))
229         — Case space predicted
230         if probability[1] > math.log(cut) then
231             predictions[position] = 1
232             position = position + 1
233         end
234
235         — Copying next character
236         predictions[position] = gram_input[i+N-1]
237         position = position + 1
238     end
239     — Adding last character (</s>)
240     predictions[position] = gram_input[gram_input:size(1)]
241     — Cutting the output
242     return predictions:narrow(1,1,position)
243 end
244
245 — Viterbi algorithm to predict a sequence from gram_input with a count
246 — based probability model
247 — pi matrix format (col1: space; col2: char)
248 function predict_NN_viterbi(gram_input, nnlm, N)
249     — Backpointer

```

```

250     local score
251     local bp = torch.zeros(gram_input:size(1) + 1, 2)
252     local context = torch.DoubleTensor(1)
253     local y_hat = torch.DoubleTensor(2)
254     local pi = torch.ones(gram_input:size(1) + 1, 2):mul(-9999)
255     — Initialization
256     pi[{1,1}] = 0
257     — i is shifted
258     for i=2,gram_input:size(1)+1 do
259         for c_prev =1,2 do
260             — Precompute y_hat(c_prev)
261             if c_prev == 1 then
262                 context[1] = c_prev
263             else
264                 context[1] = gram_input[i-1]
265             end
266             — Line where the model appears
267             y_hat:copy(nnlm:forward(context))
268
269             for c_current =1,2 do
270                 score = pi[{i-1, c_prev}] + y_hat[c_current]
271                 if score > pi[{i, c_current}] then
272                     pi[{i, c_current}] = score
273                     bp[{i, c_current}] = c_prev
274                 end
275             end
276         end
277     end
278     return pi, bp
279 end
280
281 — Building the sequences from the backpointer
282 function build_sequences_from_bp(bp, gram_input)
283     local predictions = torch.DoubleTensor(2*gram_input:size(1))
284     — Next position to fill in predictions (have to do it backward)
285     local position = 2*gram_input:size(1)
286     local col = 2
287     — Loop until the 3rd position (because 2nd is the first one, could
        be set by hand)
288     for i=bp:size(1),3,-1 do
289         — coming from a space
290         if bp[i][col] == 1 then
291             predictions[position] = 1
292             position = position - 1
293             col = 1

```

```

294         else
295             col = 2
296         end
297         — index i is shifted of 1 wrt local index in gram_input
298         predictions[position] = gram_input[i-1]
299         position = position - 1
300     end
301     — Beginnning of gram_input set
302     predictions[position] = gram_input[1]
303     position = position - 1
304
305     return predictions:narrow(1,position+1,predictions:size(1)-position
306         )
307 end
308 function main()
309     — Parse input params
310     opt = cmd:parse(arg)
311     N = opt.N
312     algo = opt.algo
313     eta = opt.eta
314     hid = opt.hid
315     embed = opt.embed
316     batchsize = opt.batch
317     Ne = opt.Ne
318
319
320     — Reading file
321     local file = hdf5.open('data_preprocessed / '..tostring(N)..'-grams.
322         hdf5', 'r')
323     data = file:all()
324     file:close()
325
326     train_input = data['input_matrix_train']
327     train_output = data['output_matrix_train']
328     input_data_train = data['input_data_train']
329
330     input_data_valid = data['input_data_valid_nospace']:clone()
331
332     input_data_test = data['input_data_test']:clone()
333
334     — Building the model
335     torch.manualSeed(1)
336
337     nnlm1, crit = build_model(N-1, 49, 2, embed, hid)

```

```

337
338     print('-> Training the model')
339     train_model(train_input, train_output, nnlm1, crit, N-1, 2, eta, Ne
        , batchsize)
340
341     print('Ngram size '..tostring(N))
342     print('Train Perplexity ')
343     print(compute_perplexity(input_data_train, nnlm1, N))
344     print('Valid Perplexity ')
345     print(compute_perplexity(input_data_valid, nnlm1, N))
346
347     — Prediction
348     if (algo == 'greedy') then
349         predictions_test = predict_NN_greedy(input_data_test, nnlm1, N)
350     elseif (algo == 'viterbi') then
351         pi, bp = predict_count_based_viterbi(input_data_test, nnlm1, N)
352         predictions_test = build_sequences_from_bp(bp, input_data_test)
353     else
354         error("invalid algorithm input")
355     end
356
357     — Kaggle format
358     num_spaces = get_kaggle_format(predictions_test, N)
359
360     print(num_spaces:narrow(1,1,10))
361
362     — — Saving the Kaggle format output
363     — myFile = hdf5.open('submission/'..'opt.f', 'w')
364     — myFile:write('num_spaces', num_spaces)
365     — myFile:close()
366 end
367
368 main()

```

## **RNN:**

```

1 — Documentation:
2 — ——— How to call it from the command line?
3 — For example:
4 — $ th count_based.lua -N 5
5 — Other argument possible (see below)
6 —
7 — ——— Is there an Output?
8 — By default, the predictions on the test set are saved in hdf5 format
   as classifier .. opt.f
9

```

```

10 — Only requirements allowed
11 require("hdf5")
12 require("rnn")
13 require 'helper.lua';
14
15 cmd = torch.CmdLine()
16
17 — Cmd Args
18 cmd:option('-l', 30, 'Length size for the training sequence')
19 cmd:option('-b', 16, 'Batch-size for the training')
20 cmd:option('-edim', 20, 'Embed dimension for the characters embeddings
    ')
21 cmd:option('-eta', 0.5, 'Learning rate')
22 cmd:option('-ne', 4, 'Number of epochs for the training')
23 cmd:option('-s', 1, 'Step size for the adaptive eta changes')
24 cmd:option('-f', 'pred_test_rnn.f5', 'File name for the predictions on
    the test')
25 cmd:option('-model', 'RNN', 'Recurrent model to be used (RNN, LSTM or
    GRU')
26
27
28 — Formating the input
29 — input is a 1d tensor
30 function get_train_input(input, len, batch_size)
31     — Building output (we put predict a padding at the end)
32     local n = input:size(1)
33
34     — Get the closer multiple of batch_size*len below n
35     local factor = -math.floor(-n/(len*batch_size))
36     local n_new = factor*len*batch_size
37     local input_new = torch.DoubleTensor(n_new)
38     local t_input, t_output
39     input_new:narrow(1,1,n):copy(input)
40     input_new:narrow(1,n,n_new-n+1):fill(2) — Filling with padding
41
42     — Building output
43     local output = get_output(input_new)
44
45     — Issue with last sequence if batch_size does not divide n
46     t_input = torch.split(input_new:view(batch_size,n_new/batch_size),
        len, 2)
47     t_output = torch.split(output:view(batch_size,n_new/batch_size),len
        , 2)
48     return t_input, t_output
49 end

```

```

50
51 function get_output(input)
52     local n = input:size(1)
53     local output = torch.DoubleTensor(n)
54     for i=2, n do
55         if input_new[i] ~= 1 then
56             output[i-1] = 2
57         else
58             output[i-1] = input[i]
59         end
60     end
61     output[n] = 2
62     return output
63 end
64
65 — Methods to build the model
66 function build_RNN(embed_dim, rho)
67     return nn.Recurrent(embed_dim, nn.Linear(embed_dim, embed_dim), nn.
        Linear(embed_dim, embed_dim), nn.Tanh(), rho)
68 end
69
70 function build_LSTM(embed_dim, rho)
71     return nn.FastLSTM(embed_dim, embed_dim, rho)
72 end
73
74 function build_GRU(embed_dim, rho, dropout_p)
75     return nn.GRU(embed_dim, embed_dim, rho, dropout_p)
76 end
77
78 function build_rnn(embed_dim, vocab_size, batch_size, recurrent_model,
    len)
79     local batchRNN
80     local params
81     local grad_params
82     — generic RNN transduced
83     batchRNN = nn.Sequential()
84         :add(nn.LookupTable(vocab_size, embed_dim))
85         :add(nn.SplitTable(1, batch_size))
86     local rec = nn.Sequencer(recurrent_model)
87     rec:remember('both')
88
89     batchRNN:add(rec)
90
91 — Output
92 batchRNN:add(nn.Sequencer(nn.Linear(embed_dim, 2)))

```

```

93     batchRNN:add(nn.Sequencer(nn.LogSoftMax()))
94
95     — Retrieve parameters (To do only once!!!)
96     params, grad_params = batchRNN:getParameters()
97     — Initializing all the parameters between -0.05 and 0.05
98     for k=1,params:size(1) do
99         params[k] = torch.uniform(-0.05,0.05)
100     end
101
102     return batchRNN, params, grad_params
103 end
104
105 function train_model_with_perp(t_input, t_output, model,
    model_flattened, params_flattened,
106     params, grad_params, criterion, eta, nEpochs, batch_size, len,
    n, input_valid, output_valid, step)
107     — Train the model with a mini batch SGD
108     — Uses an adaptive learning rate eta computed each cycle of step
    iterations from the
109     — evolution of the perplexity on the validation set (compute with
    the model_flattened)
110     local timer
111     local pred
112     local loss
113     local dLdPred
114     local t_inputT = torch.DoubleTensor(len, batch_size)
115     local t_output_table
116     local size
117
118     — To store the loss
119     local av_L = 0
120     local perp = 0
121     local old_perp = 0
122
123     for i = 1, nEpochs do
124         — timing the epoch
125         timer = torch.Timer()
126         old_L = av_L
127         old_perp = perp
128         av_L = 0
129
130         — mini batch loop
131         for k = 1, n/(batch_size * len) do
132             — Mini batch data
133

```



```

134         t_inputT:copy(t_input[k]:t())
135         t_output_table = torch.split(t_output[k],1,2)
136         —format the output
137         for j=1,len do
138             t_output_table[j] = t_output_table[j]:squeeze()
139         end
140
141         — reset gradients
142         grad_params:zero()
143
144         — Forward loop
145         pred = model:forward(t_inputT)
146         loss = criterion:forward(pred, t_output_table)
147         av_L = av_L + loss
148
149         — Backward loop
150         dLdPred = criterion:backward(pred, t_output_table)
151         model:backward(t_inputT, dLdPred)
152
153         — gradient normalization with max norm 5 (l2 norm)
154         grad_params:view(grad_params:size(1),1):renorm(1,2,5)
155         model:updateParameters(eta)
156
157     end
158
159     print('Epoch '..i..'': '..timer:time().real)
160     print('Average Loss: '..av_L/math.floor(n/batch_size))
161     — Print perplexity validity every step of iteration
162     if (i%step == 0) then
163         size = input_valid:size(1) - 1
164         params_flattened:copy(params)
165         perp = compute_perplexity(input_valid:narrow(1,1,size):view
            (size,1), output_valid, model_flattened)
166         print('Valid perplexity: '..perp)
167
168         if old_perp - perp < 0 then
169             eta = eta/2
170         end
171
172         if (eta < 0.0001) then eta = 0.1 end
173
174     end
175 end
176 end
177

```

```

178 

---


179 

---

 Methods for prediction 

---


180 

---


181
182 function compute_probability_model(model, input)
183     return model:forward(input:view(input:size(1), 1))
184 end
185
186 — Method to compute manually the perplexity
187 function compute_perplexity(input, output, model)
188     — Last Position filled in predictions
189     — Position to predict in input
190     local position_input = 1
191     local probability = torch.DoubleTensor(2)
192     local probability_table
193     local perp = 0
194
195     — Build mapping
196     for i = 1,input:size(1) do
197         — Line where the model appears
198         — The model remember the states before, just need to feed into
           it a character
199         probability_table = compute_probability_model(model, input:
           narrow(1,i,1))
200         probability:copy(probability_table[1])
201         perp = perp + probability[output[i]]
202     end
203     — Cutting the output
204     return math.exp(-perp/input:size(1))
205 end
206
207 — Prediction with greedy algorithm
208 function predict_rnn_greedy(input, len, model)
209     — Last Position filled in predictions
210     local position_prediction = 1
211     — Position to predict in input
212     local position_input = 1
213     — We allocate the maximum of memory that could be needed
214     — Default value is -1 (to know where predictions end afterwards)
215     local predictions = torch.ones(2*input:size(1)):mul(-1)
216     — Copy the first entry
217     predictions[position_prediction] = input[position_input]
218     local probability = torch.zeros(2)
219     local probability_table
220

```

```

221 — Build mapping
222 while position_input < input:size(1) do
223     — Line where the model appears
224     — The model remember the states before , just need to feed into
        it a character
225     probability_table = compute_probability_model(model,
        predictions:narrow(1,position_prediction , 1))
226     probability:copy(probability_table[1])
227
228     m,a = probability:max(1)
229
230     — Case space predicted
231     position_prediction = position_prediction +1
232     if (a[1] == 1) then
233         predictions[position_prediction] = 1
234     else
235         — Copying next character
236         position_input = position_input + 1
237         predictions[position_prediction] = input[position_input]
238     end
239 end
240 — Cutting the output
241 return predictions:narrow(1,1,position_prediction)
242 end
243
244 function main()
245     — Parse input params
246     opt = cmd:parse(arg)
247
248     — Reading file
249     N = 2
250     loacal data = hdf5.open( '../ data_preprocessed / '.. tostring(N)..' —
        grams.hdf5' , 'r') : all()
251     F_train = data['F_train']
252     input_data_valid = data['input_data_valid']
253     input_matrix_train = data['input_matrix_train']
254     input_data_train = data['input_data_train']
255     input_data_valid_nospace = data['input_data_valid_nospace']
256     input_data_test = data['input_data_test']
257     myFile:close()
258
259     F_train = data['F_train']
260     input_data_valid = data['input_data_valid']
261     input_data_train = data['input_data_train']
262     input_data_test = data['input_data_test']

```

```

263     input_data_valid_nospace = data['input_data_valid_nospace']
264
265     — Model parameters
266     len = opt.l
267     batch_size = opt.b
268     vocab_size = 49
269     embed_dim = oopt.edim
270     eta = opt.eta
271     nEpochs = opt.ne
272     step = opt.s
273
274     — Formating data
275     t_input_new, t_output_new = get_train_input(input_data_train, len,
        batch_size)
276     output_valid = get_output(input_data_valid)
277     n_new = len * batch_size * (#t_input_new)
278
279     — Building model
280     model, params, grad_params = build_rnn(embed_dim, vocab_size,
        batch_size, build_RNN(embed_dim, len), len)
281     model_valid, params_valid, grad_params_valid = build_rnn(embed_dim,
        vocab_size, 1, build_RNN(embed_dim))
282
283     crit = nn.SequencerCriterion(nn.ClassNLLCriterion())
284
285     — Training model
286     train_model_with_perp(t_input_new, t_output_new, model, model_valid
        , params_valid,
287         params, grad_params, crit, eta, nEpochs, batch_size, len,
        n_new, input_data_valid, output_valid, 5)
288
289     — Computing RMSE on valid
290     kaggle_true_valid = get_kaggle_format(input_data_valid, 2)
291
292     timer = torch.Timer()
293     pred_valid = predict_rnn_greedy(input_data_valid_nospace:narrow
        (1,1,input_data_valid_nospace:size(1)), len, model_valid)
294     print('Greedy prediction on validation set (Time elapsed : '..timer
        :time().real..' )')
295     kaggle_model_valid = get_kaggle_format(pred_valid, 2)
296     print('RMSE')
297     rsme = compute_rmse(kaggle_true_valid, kaggle_model_valid)
298     print(rsme)
299
300     — Prediction on test

```

```
301     timer = torch.Timer()
302     size = input_data_test:size(1)
303     pred_test = predict_rnn_greedy(input_data_test:narrow(1,1,size),
        len, model_valid)
304     print('Greedy prediction on test set (Time elapsed : '..timer:time
        ()).real..' ')
305     kaggle_test = get_kaggle_format(pred_test,2)
306
307     — Saving the Kaggle format output
308     myFile = hdf5.open('../submission/'..'opt.f', 'w')
309     myFile:write('num_spaces', kaggle_test)
310     myFile:close()
311 end
```