HW4: Word Segmentation

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

April 2, 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:

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

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 ϵ
  $\pi[0, \langle s\rangle] = 0$
  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 $_{-\mathbb{1}}$ and the output layer producing the output at each timestep. We used Elman tanh layer for the output.

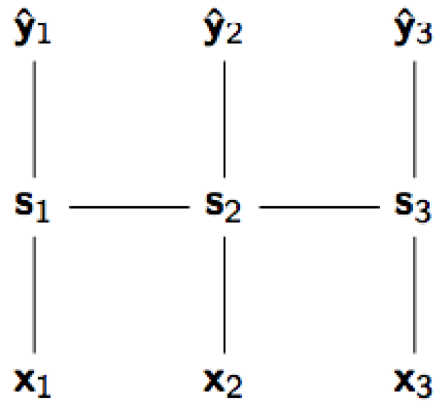

Figure 1: Transducer Architecture

Formally:

$$\begin{aligned} \hat{} &= softmax(\mathbb{W} +) \\ &= tanh([, _{-1}\mathbb{W} +) \end{aligned}$$

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 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.

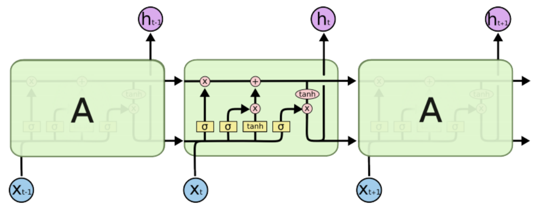

Figure 3: LSTM Architecture

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 gienve 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 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:

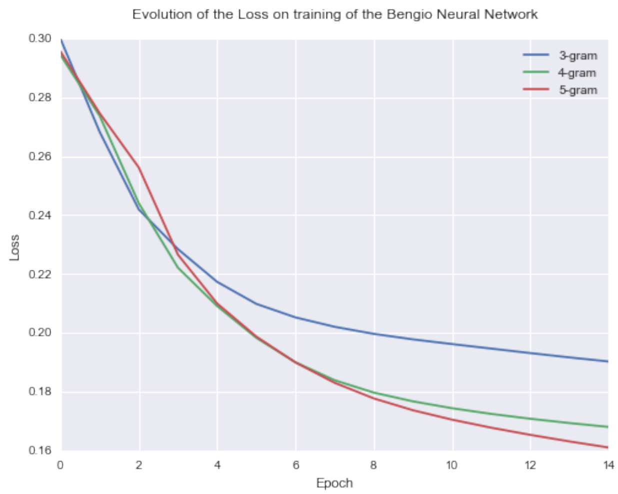

Figure 6: Training Loss

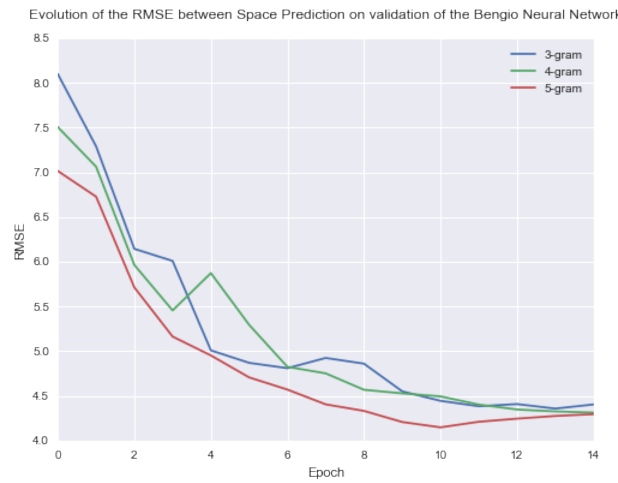

Figure 7: RMSE on Validation set

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

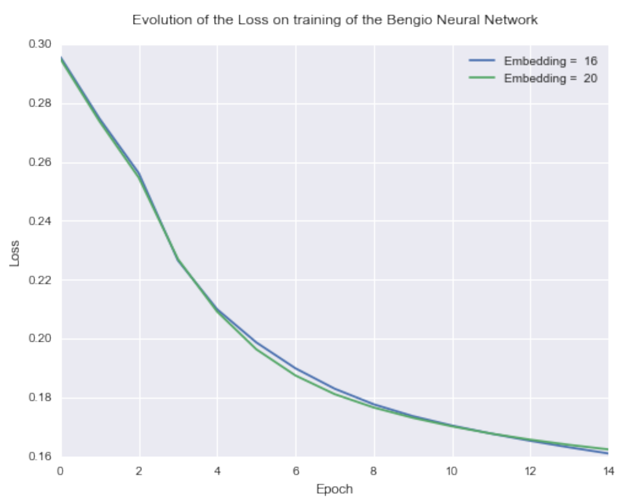

Figure 8: Training Loss

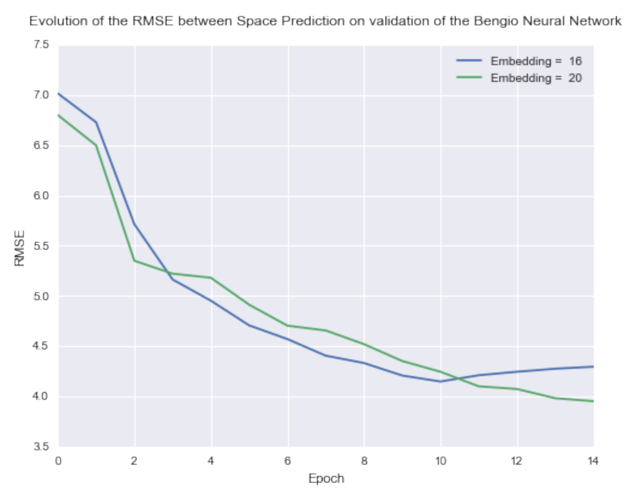

Figure 9: RMSE on Validation set

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 threshhold instead of using argmax prediction. The ratio of spaces to characters in the training set being relatively little, by specifying a probability smaller than 0.5 above which we generate a space could help the performance of the greedy algorithm. We present results for a cutoff probability ranging from 0.2 to 0.5:

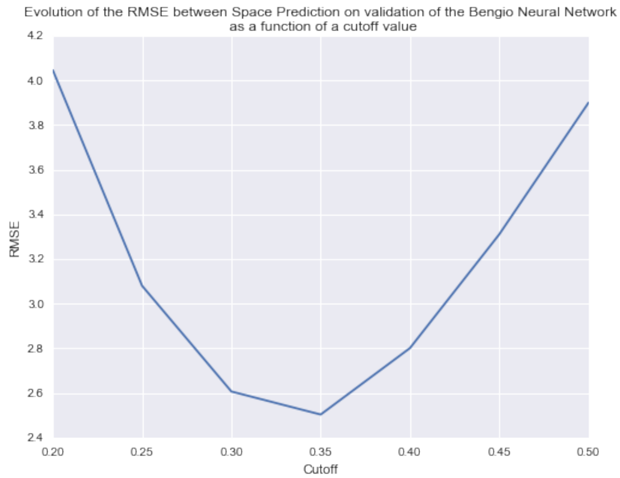

Figure 10: Perplexity evolution for the RNN

Results improve by 35% using the cutoff trick, and the results on Kaggle are also much better:

$$RMSE_{cutoff} = \sqrt{6.09} = 2.47$$

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 prior explorations also.

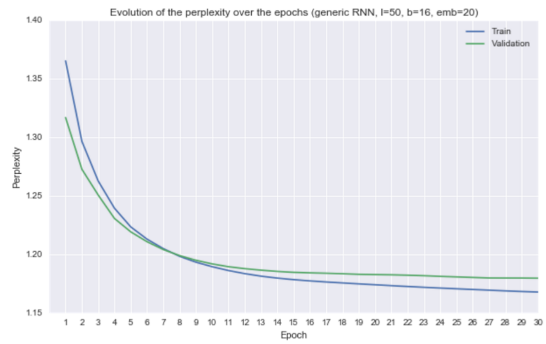

Figure 11: Perplexity evolution for the RNN

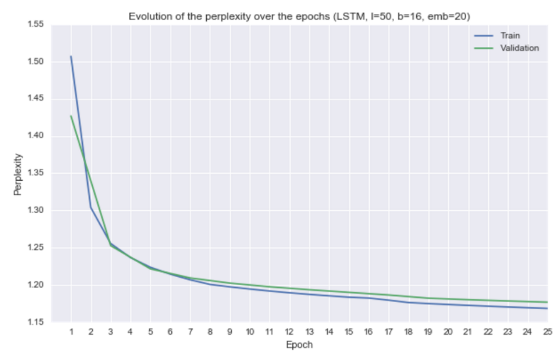

Figure 12: Perplexity evolution for the LSTM

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).

We also applied the cutoff trick on the GRU to refine the sequence generation for the Kaggle competition. Here the metric we care about is the RMSE on the number of spaces per sentence.

We tuned the cutoff value on the validation set. We kept the cutoff value reaching the minimum of RMSE for both the GRU (0.325) and the RNN (0.275) and combined them with a weighted sum to obtain our best result on Kaggle.

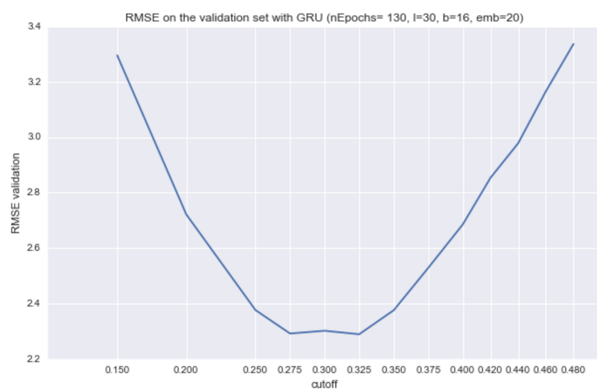

Figure 14: RMSE on the validation with the GRU with cutoff

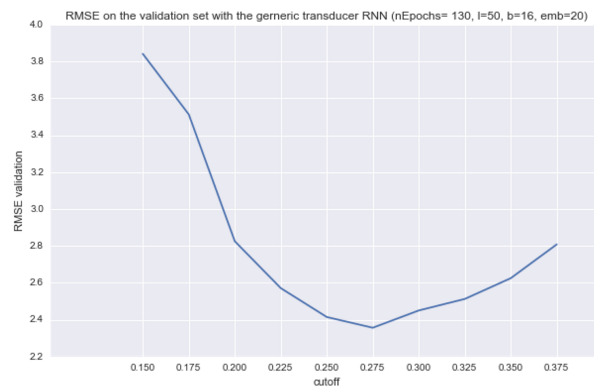

Figure 15: RMSE on the validation with the RNN with cutoff

4.4 Model performance summary

Here we summarize the performance of our different models. We reported the perplexity on the validation set computed from the model and the RMSE computed by Kaggle on the sequence predicted with our chosen algorithm.

First, we observed that the count 5gram count based model still provides a better sequence generated with the greedy algorithm as the 3gram one generated with Viterbi. We also have a notable difference for the reccurent networks with the RMSE computed on Kaggle even though we have similar perplexity. The large value of RMSE on the validation could be decreased with an adaptive cutoff as we showed it for our best model.

Table 1: Summary of the results

Model	Sequence generation algorithm	Perplexity on validation	MSE Kaggle
count based 5gram	Greedy	1.1467	17.88
count based 3gram	Viterbi	1.2780	56.27
NN	Greedy	1.156	13.37926
NN with cutoff	Greedy	1.156	6.09
RNN	Greedy	1.1746	33.13
LSTM	Greedy	1.1766	18.95
GRU	Greedy	1.1513	10.94
GRU + RNN with cutoof	Greedy	-	**5.39**

5 Conclusion

This segmentation task gave us the opportunity to implement different recurrent neural network architectures but also to compare them with more traditionnal method. Whereas the count based and even the simple neural network models are pretty fast to train they still provide interesting results. The results provided by the three variants of RNN were interesting to illustrate the influence of gates and memory in such networks. The gated reccurent network ended as the best model on this task. One future work could be to stack more layers to our reccurent architecture or to implement a network with a dynamic memory part to give more flexibility in how the model uses the information it already processed.

Appendices

Preprocessing:

```
import numpy as np
import h5py
import argparse
import sys
import re
import codecs

from collections import Counter


FILE_PATHS = ("data/train_chars.txt",
              "data/valid_chars.txt",
              "data/test_chars.txt")


def get_input(filename, n, char_to_ind=None):
    # Contain the list of characters indices in the data
    # initialized with a padding
    if n > 2:
        input_data = [2]*(n-2)
    else:
        input_data = []
    if char_to_ind is None:
        # Map each character to an index with
        # Index of <space> set to 1
        char_to_ind = {'<space>': 1, '</s>': 2}
        count = 3
    with open(filename, 'r') as f:
        # Loop to index the char and store them inside the input
```

```
        for line in f:
            for c in line[:-1].split(' '):
                # Input data
                if c in char_to_ind:
                    input_data.append(char_to_ind[c])
                else:
                    char_to_ind[c] = count
                    count += 1
                    input_data.append(char_to_ind[c])
    return input_data, char_to_ind


def build_train_data(input_data, n):
    # Build the input matrix: (num_records, n-1)
    # and the output vector (num_records,1)
    # which stores the output for the given (n-1)gram
    input_matrix = np.zeros((len(input_data)-n, n-1))
    output_matrix = np.zeros(len(input_data)-n)
    for i in xrange(len(input_data)-n):
        # Countext is a (n-1)gram
        w = input_data[i:i+(n-1)]
        input_matrix[i, :] = w
        output_matrix[i] = (1 if input_data[i+(n-1)] == 1 else 2)
    return input_matrix, output_matrix


def build_count_matrix(input_matrix, output_matrix, n):
    count_matrix_raw = np.concatenate((input_matrix,
                                       output_matrix.reshape(
                                           output_matrix.shape[0], 1)),
                                           axis=1)

    num_rows = len(set([tuple(s) for s in input_matrix]))
    count = Counter([tuple(s) for s in count_matrix_raw])

    # count matrix: (num_(n-1grams, 2))
    F = np.zeros((num_rows, n + 1))
    gram_to_ind = {}
    i = 0
    for k, v in count.iteritems():
        gram = k[:(n-1)]
        if gram not in gram_to_ind:
            gram_to_ind[gram] = i
            i += 1
        F[gram_to_ind[gram], n-1 + int(k[-1]) - 1] = v
```

```
        F[gram_to_ind[gram], :n-1] = list(gram)

    return F


def main(arguments):
    global args
    parser = argparse.ArgumentParser(
        description=__doc__,
        formatter_class=argparse.RawDescriptionHelpFormatter)

    parser.add_argument('--N', default=2, type=int, help='Ngram size')
    args = parser.parse_args(arguments)
    N = args.N

    train, valid, test = FILE_PATHS

    # Train
    input_data_train, char_to_ind = get_input(train, N)
    input_matrix_train, output_matrix_train = build_train_data(
        input_data_train, N)
    F_train = build_count_matrix(input_matrix_train, output_matrix_train, N)

    # Valid
    input_data_valid, char_to_ind = get_input(valid, N, char_to_ind)
    input_data_valid_nospace = filter(lambda a: a != 1, input_data_valid)

    # Test
    input_data_test, char_to_ind = get_input(test, N, char_to_ind)

    filename = 'data_preprocessed/' + str(N) + '-grams.hdf5'
    with h5py.File(filename, "w") as f:
        # Stores a matrix (num_records, N-1) with at each row
        # the (N-1) grams appearing in the input data
        f['input_matrix_train'] = input_matrix_train
        f['F_train'] = F_train
        # Vector (num_records) storing the class of the next word
        # after the (N-1) gram stored at the same index in input_matrix
        # 1 is space; 2 is character
        f['output_matrix_train'] = output_matrix_train
        # Stores the list of consecutives character (or space) as their
        # index from the mapping char_to_ind
        f['input_data_train'] = np.array(input_data_train)
```

```
        f['input_data_valid'] = np.array(input_data_valid)
        f['input_data_valid_nospace'] = np.array(input_data_valid_nospace)
        f['input_data_test'] = np.array(input_data_test)


if __name__ == '__main__':
    sys.exit(main(sys.argv[1:]))
```

Count-Based Models:

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

-- Only requirements allowed
require("hdf5")
require 'helper.lua';

cmd = torch.CmdLine()

-- Cmd Args
cmd:option('-N', 2, 'Ngram size for the input')
cmd:option('-algo', 'greedy', 'Algorithm to use: either greedy or viterbi')
cmd:option('-f', 'pred_test.f5', 'File name for the predictions on the test')

-- Build the mapping from (N-1)gram to row index
-- and the count matrix F_count: (num_context, 2)
function get_F_count(F, N)
    local ngram_to_ind = {}
    local key
    for i=1,F:size(1) do
        key = tostring(F[{i,1}])
        -- Building key
        for k = 2,N-1 do
            key = key .. '-' .. tostring(F[{i,k}])
        end
        ngram_to_ind[key] = i
```

```
    end
    return F:narrow(2,N,2), ngram_to_ind
end

-- Compute proba distribution over (space, char) for the context
-- F is here the count matrix (num_context, 2)
function compute_count_based_probability(context, F_count, ngram_to_ind, alpha)
    local probability = torch.zeros(2)
    -- Building key, ie (N-1)gram (from i to i+(N-2))
    local key = tostring(context[1])
    for k = 2,context:size(1) do
        key = key .. '-' .. tostring(context[k])
    end
    -- If (N-1)gram never seen, prior distribution
    if (ngram_to_ind[key] ~= nil) then
        -- index of the current (n-1)gram in the F matrix
        local index = ngram_to_ind[key]
        probability:copy(F_count:narrow(1,index,1))
        -- Adding smoothing
        probability:add(alpha)
    -- Case unseen context
    else
        -- Prior
        probability:copy(torch.DoubleTensor({F_count:narrow(2,1,1):sum(), F_count:narrow(2,2,1):sum()}))
    end
    return probability:div(probability:sum())
end

-- Compute perplexity on entry with space
function compute_perplexity(gram_input, F_count, ngram_to_ind, N)
    local perp = 0
    local context = torch.zeros(N-1)
    local probability = torch.zeros(2)
    -- Do not predict for the last char
    --for i=1,gram_input:size(1)-N do
    local size=gram_input:size(1) - (N-1)
    for i=1,size do
        context:copy(gram_input:narrow(1,i,N-1))
        -- Line where the model appears
        probability:copy(compute_count_based_probability(context, F_count, ngram_to_ind, 1))
        if gram_input[i+(N-1)] == 1 then
            right_proba = probability[1]
```

```
            --print('space')
            --print(right_proba)
        else
            right_proba = probability[2]
        end
        perp = perp + math.log(right_proba)
    end
    perp = math.exp(-perp/size)
    --perp = math.exp(-perp/(gram_input:size(1)-N))
    return perp
end

-- Greedy algorithm to predict a sequence from gram_input with a count
-- based probability model
function predict_count_based_greedy(gram_input, F_count, ngram_to_ind,
    N)
    -- Next Position to fill in predictions
    local position = N
    -- We allocate the maximum of memory that could be needed
    -- Default value is -1 (to know where predictions end afterwards)
    local predictions = torch.ones(2*(gram_input:size(1) - N)):mul(-1)
    -- Copy the first (N-1) gram
    predictions:narrow(1,1,N-1):copy(gram_input:narrow(1,1,N-1))
    local probability = torch.zeros(2)
    local context = torch.zeros(N-1)

    -- Build mapping
    for i=1,gram_input:size(1)-N do
        -- Compute proba for next char
        context:copy(predictions:narrow(1,position-(N-1),N-1))
        -- Line where the model appears
        probability:copy(compute_count_based_probability(context,
            F_count, ngram_to_ind, 1))
        m,a = probability:max(1)

        -- Case space predicted
        if (a[1] == 1) then
            predictions[position] = 1
            position = position +1
        end

        -- Copying next character
        predictions[position] = gram_input[i+N-1]
        position = position +1
    end
```

```
    -- Adding last character (</s>)
    predictions[position] = gram_input[gram_input:size(1)]
    -- Cutting the output
    return predictions:narrow(1,1,position)
end

-- Viterbi algorithm to predict a sequence from gram_input with a count
-- based probability model
-- pi matrix format (col1: space; col2: char)
function predict_count_based_viterbi(gram_input, F_count, ngram_to_ind, N)
    -- Backpointer
    local score
    local bp = torch.zeros(gram_input:size(1) + 1, 2)
    local context = torch.DoubleTensor(1)
    local y_hat = torch.DoubleTensor(2)
    local pi = torch.ones(gram_input:size(1) + 1, 2):mul(-9999)
    -- Initialization
    pi[{1,1}] = 0
    -- i is shifted
    for i=2,gram_input:size(1)+1 do
        for c_prev =1,2 do
            -- Precompute y_hat(c_prev)
            if c_prev == 1 then
                context[1] = c_prev
            else
                context[1] = gram_input[i-1]
            end
            -- Line where the model appears
            y_hat:copy(compute_probability(context, F_count, ngram_to_ind, 1))

            for c_current =1,2 do
                score = pi[{i-1, c_prev}] + math.log(y_hat[c_current])
                if score > pi[{i, c_current}] then
                    pi[{i, c_current}] = score
                    bp[{i, c_current}] = c_prev
                end
            end
        end
    end
    return pi, bp
end

-- Building the sequences from the backpointer
```

```lua
function build_sequences_from_bp(bp, gram_input)
    local predictions = torch.DoubleTensor(2*gram_input:size(1))
    -- Next position to fill in predictions (have to do it backward)
    local position = 2*gram_input:size(1)
    local col = 2
    -- Loop until the 3rd position (because 2nd is the first one, could be set by hand)
    for i=bp:size(1),3,-1 do
        -- coming from a space
        if bp[i][col] == 1 then
            predictions[position] = 1
            position = position - 1
            col = 1
        else
            col = 2
        end
        -- index i is shifted of 1 wrt local index in gram_input
        predictions[position] = gram_input[i-1]
        position = position - 1
    end
    -- Beginnning of gram_input set
    predictions[position] = gram_input[1]
    position = position - 1

    return predictions:narrow(1,position+1,predictions:size(1)-position)
end

-- Viterbi trigram
function predict_count_based_viterbi_trigram(gram_input, F_count, ngram_to_ind, N)
    -- Backpointer
    local score
    local bp = torch.zeros(gram_input:size(1) + 1, 3)
    local context = torch.DoubleTensor(2)
    local y_hat = torch.DoubleTensor(2)
    -- pi is built as ('char-space', 'char-char', 'space-char')
    -- corresponding index in the context
    local pi = torch.ones(gram_input:size(1) + 1, 3):mul(-999999999)
    -- Initialization
    pi[{2,1}] = 0
    --pi[{2,2}] = 0
    --pi[{2,3}] = 0
    -- We need to start at the first trigram
    for i=3,gram_input:size(1)+1 do
```

```
        for c_prev =1,3 do
            -- Precompute y_hat(c_prev)
            if c_prev == 1 then
                context[1] = gram_input[i-2]
                context[2] = 1
            elseif c_prev == 2 then
                context[1] = gram_input[i-2]
                context[2] = gram_input[i-1]
            else
                context[1] = 1
                context[2] = gram_input[i-1]
            end
            -- Line where the model appears
            y_hat:copy(compute_probability(context, F_count, ngram_to_ind, 1))

            -- cannot have 2 spaces in a row: from 1 goes to 3 necessarily
            if c_prev == 1 then
                pi[{i, 3}] = pi[{i-1, c_prev}] + math.log(y_hat[2])
                bp[{i, 3}] = c_prev
            else
                -- last char is necessarily 'char' so
                -- 1: space predicted (ie 'char-space')
                -- 2: char predicted (ie 'char-char')
                for c_current =1,2 do
                    score = pi[{i-1, c_prev}] + math.log(y_hat[c_current])
                    if score > pi[{i, c_current}] then
                        pi[{i, c_current}] = score
                        bp[{i, c_current}] = c_prev
                    end
                end
            end
        end
    end
    return pi, bp
end

-- Building the sequences from the backpointer
-- We start the sequence by the ('char'-'char') configuration
-- as we know it's the only one possible
function build_sequences_from_bp_trigram(bp, gram_input)
    local predictions = torch.DoubleTensor(2*gram_input:size(1))
    -- Next position to fill in predictions (have to do it backward)
```

```
    local position = 2*gram_input:size(1)
    local col = 2
    -- Loop until the 4th position
    for i=bp:size(1),4,-1 do
        -- coming from a space
        if bp[i][col] == 1 then
            predictions[position] = 1
            position = position - 1
        end
        col = bp[i][col]
        -- index i is shifted of 1 wrt local index in gram_input
        predictions[position] = gram_input[i-1]
        position = position - 1
    end
    -- Beginnning of gram_input set
    predictions[position] = gram_input[2]
    position = position - 1
    predictions[position] = gram_input[1]
    position = position - 1

    return predictions:narrow(1,position+1,predictions:size(1)-position)
end

function main()
    -- Parse input params
    opt = cmd:parse(arg)
    N = opt.N
    algo = opt.algo

    -- Reading file
    local file = hdf5.open('data_preprocessed/'..tostring(N)..'-grams.hdf5', 'r')
    data = file:all()
    file:close()

    F_train = data['F_train']
    input_data_valid = data['input_data_valid']
    input_data_train = data['input_data_train']
    input_data_test = data['input_data_test']
    input_data_valid_nospace = data['input_data_valid_nospace']

    -- Building the model
    F_count, ngram_to_ind = get_F_count(F_train, N)
    print('Ngram size '..tostring(N))
```

```
    print('Train Perplexity')
    print(compute_perplexity(input_data_train, F_count, ngram_to_ind, N))
    print('Valid Perplexity')
    print(compute_perplexity(input_data_valid, F_count, ngram_to_ind, N))

    -- Prediction
    if (algo == 'greedy') then
        predictions_test = predict_count_based_greedy(input_data_test, F_count, ngram_to_ind, N)
    elseif (algo == 'viterbi') then
        if (N == 2) then
            pi, bp = predict_count_based_viterbi(input_data_test, F_count, ngram_to_ind, N)
            predictions_test = build_sequences_from_bp(bp, input_data_test)
        elseif (N == 3) then
            pi_tri, bp_tri = predict_count_based_viterbi_trigram(input_data_test, F_count, ngram_to_ind, N)
            predictions_test = build_sequences_from_bp_trigram(bp_tri, input_data_test)
        else
            error("invalid N for Viterbi")
        end
    else
        error("invalid algorithm input")
    end

    -- Kaggle format
    num_spaces = get_kaggle_format(predictions_test, N)

    -- Saving the Kaggle format output
    myFile = hdf5.open('submission/'..opt.f, 'w')
    myFile:write('num_spaces', num_spaces)
    myFile:close()
end

main()
```

NNLM:

```
require 'hdf5';
require 'nn';
require 'helper.lua';

```

```
cmd = torch.CmdLine()

-- Cmd Args
cmd:option('-N', 5, 'Ngram size for the input')
cmd:option('--embed', 16, 'Embedding size of characters')
cmd:option('--hid', 80, 'Hidden layer dimension')
cmd:option('--eta', 0.01, 'Learning rate')
cmd:option('--batch', 10, 'Batchsize')
cmd:option('--Ne', 20, 'Number of epochs')
cmd:option('-algo', 'greedy', 'Algorithm to use: either greedy or viterbi')
cmd:option('-f', 'pred_test.f5', 'File name for the predictions on the test')

function build_model(dwin, nchar, nclass, hid1, hid2)
    -- Model with skip layer from Bengio, standards parameters
    -- should be:
    -- dwin = 5
    -- hid1 = 30
    -- hid2 = 100

    -- To store the whole model
    local dnnlm = nn.Sequential()

    -- Layer to embedd (and put the words along the window into one vector)
    local LT = nn.Sequential()
    local LT_ = nn.LookupTable(nchar,hid1)
    LT:add(LT_)
    LT:add(nn.View(-1, hid1*dwin))

    dnnlm:add(LT)

    local concat = nn.ConcatTable()

    local lin_tanh = nn.Sequential()
    lin_tanh:add(nn.Linear(hid1*dwin,hid2))
    lin_tanh:add(nn.Tanh())

    local id = nn.Identity()

    concat:add(lin_tanh)
    concat:add(id)

    dnnlm:add(concat)
```

```
    dnnlm:add(nn.JoinTable(2))
    dnnlm:add(nn.Linear(hid1*dwin + hid2, nclass))
    dnnlm:add(nn.LogSoftMax())

    -- Loss
    local criterion = nn.ClassNLLCriterion()

    return dnnlm, criterion
end


function train_model(train_input, train_output, dnnlm, criterion, dwin, nclass, eta, nEpochs, batchSize)
    -- Train the model with a mini batch SGD
    -- standard parameters are
    -- nEpochs = 1
    -- batchSize = 32
    -- eta = 0.01

    -- To store the loss
    local av_L = 0

    -- Memory allocation
    local inputs_batch = torch.DoubleTensor(batchSize,dwin)
    local targets_batch = torch.DoubleTensor(batchSize)
    local outputs = torch.DoubleTensor(batchSize, nclass)
    local df_do = torch.DoubleTensor(batchSize, nclass)

    for i = 1, nEpochs do
        -- timing the epoch
        local timer = torch.Timer()

        av_L = 0

        -- max renorm of the lookup table
        dnnlm:get(1):get(1).weight:renorm(2,1,1)

        -- mini batch loop
        for t = 1, train_input:size(1), batchSize do
            -- Mini batch data
            local current_batch_size = math.min(batchSize,train_input:size(1)-t)
            inputs_batch:narrow(1,1,current_batch_size):copy(train_input:narrow(1,t,current_batch_size))
            targets_batch:narrow(1,1,current_batch_size):copy(
```

```
                train_output:narrow(1,t,current_batch_size))

            -- reset gradients
            dnnlm:zeroGradParameters()
            --gradParameters:zero()

            -- Forward pass (selection of inputs_batch in case the
                batch is not full, ie last batch)
            outputs:narrow(1,1,current_batch_size):copy(dnnlm:forward(
                inputs_batch:narrow(1,1,current_batch_size)))

            -- Average loss computation
            local f = criterion:forward(outputs:narrow(1,1,
                current_batch_size), targets_batch:narrow(1,1,
                current_batch_size))
            av_L = av_L +f

            -- Backward pass
            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)))
            dnnlm:backward(inputs_batch:narrow(1,1,current_batch_size),
                 df_do:narrow(1,1,current_batch_size))
            dnnlm:updateParameters(eta)

        end

        print('Epoch '..i..': '..timer:time().real)
        print('Average Loss: '..av_L/math.floor(train_input:size(1)/
            batchSize))

    end

end


-- Compute perplexity on entry with space
function compute_perplexity(gram_input, nnlm, N)
    local perp = 0
    local context = torch.zeros(N-1)
    local probability = torch.zeros(2)
    -- Do not predict for the last char
    --for i=1,gram_input:size(1)-N do
    local size=gram_input:size(1) - (N-1)
    for i=1,size do
```

```
            context:copy(gram_input:narrow(1,i,N-1))
            -- Line where the model appears
            probability:copy(nnlm:forward(context))
            if gram_input[i+(N-1)] == 1 then
                right_proba = probability[1]
            else
                right_proba = probability[2]
            end
            perp = perp + right_proba
        end
        perp = math.exp(-perp/size)
        return perp
end


-- Greedy algorithm to predict a sequence from gram_input with a count
-- based probability model
function predict_NN_greedy(gram_input, nnlm, N)
        -- Next Position to fill in predictions
        local position = N
        -- We allocate the maximum of memory that could be needed
        -- Default value is -1 (to know where predictions end afterwards)
        local predictions = torch.ones(2*(gram_input:size(1) - N)):mul(-1)
        -- Copy the first (N-1) gram
        predictions:narrow(1,1,N-1):copy(gram_input:narrow(1,1,N-1))
        local probability = torch.zeros(2)
        local context = torch.zeros(N-1)

        -- Build mapping
        for i=1,gram_input:size(1)-N do
            -- Compute proba for next char
            context:copy(predictions:narrow(1,position-(N-1),N-1))
            -- Line where the model appears
            probability:copy(nnlm:forward(context))
            m,a = probability:max(1)

            -- Case space predicted
            if (a[1] == 1) then
                predictions[position] = 1
                position = position +1
            end

            -- Copying next character
            predictions[position] = gram_input[i+N-1]
            position = position +1
```

```
    end
    -- Adding last character (</s>)
    predictions[position] = gram_input[gram_input:size(1)]
    -- Cutting the output
    return predictions:narrow(1,1,position)
end

function predict_NN_greedy(gram_input, nnlm, N)
    -- Next Position to fill in predictions
    local position = N
    -- We allocate the maximum of memory that could be needed
    -- Default value is -1 (to know where predictions end afterwards)
    local predictions = torch.ones(2*(gram_input:size(1) - N)):mul(-1)
    -- Copy the first (N-1) gram
    predictions:narrow(1,1,N-1):copy(gram_input:narrow(1,1,N-1))
    local probability = torch.zeros(2)
    local context = torch.zeros(N-1)

    -- Build mapping
    for i=1,gram_input:size(1)-N do
        -- Compute proba for next char
        context:copy(predictions:narrow(1,position-(N-1),N-1))
        -- Line where the model appears
        probability:copy(nnlm:forward(context))
        m,a = probability:max(1)

        -- Case space predicted
        if (a[1] == 1) then
            predictions[position] = 1
            position = position +1
        end

        -- Copying next character
        predictions[position] = gram_input[i+N-1]
        position = position +1
    end
    -- Adding last character (</s>)
    predictions[position] = gram_input[gram_input:size(1)]
    -- Cutting the output
    return predictions:narrow(1,1,position)
end

function predict_NN_greedy_cutoff(gram_input, nnlm, N, cut)
    -- Next Position to fill in predictions
    local position = N
```

```
    -- We allocate the maximum of memory that could be needed
    -- Default value is -1 (to know where predictions end afterwards)
    local predictions = torch.ones(2*(gram_input:size(1) - N)):mul(-1)
    -- Copy the first (N-1) gram
    predictions:narrow(1,1,N-1):copy(gram_input:narrow(1,1,N-1))
    local probability = torch.zeros(2)
    local context = torch.zeros(N-1)

    -- Build mapping
    for i=1,gram_input:size(1)-N do
        -- Compute proba for next char
        context:copy(predictions:narrow(1,position-(N-1),N-1))
        -- Line where the model appears
        probability:copy(nnlm:forward(context))
        -- Case space predicted
        if probability[1] > math.log(cut) then
            predictions[position] = 1
            position = position +1
        end

        -- Copying next character
        predictions[position] = gram_input[i+N-1]
        position = position +1
    end
    -- Adding last character (</s>)
    predictions[position] = gram_input[gram_input:size(1)]
    -- Cutting the output
    return predictions:narrow(1,1,position)
end

-- Viterbi algorithm to predict a sequence from gram_input with a count
-- based probability model
-- pi matrix format (col1: space; col2: char)
function predict_NN_viterbi(gram_input, nnlm, N)
    -- Backpointer
    local score
    local bp = torch.zeros(gram_input:size(1) + 1, 2)
    local context = torch.DoubleTensor(1)
    local y_hat = torch.DoubleTensor(2)
    local pi = torch.ones(gram_input:size(1) + 1, 2):mul(-9999)
    -- Initialization
    pi[{1,1}] = 0
    -- i is shifted
    for i=2,gram_input:size(1)+1 do
        for c_prev =1,2 do
```

```
            -- Precompute y_hat(c_prev)
            if c_prev == 1 then
                context[1] = c_prev
            else
                context[1] = gram_input[i-1]
            end
            -- Line where the model appears
            y_hat:copy(nnlm:forward(context))

            for c_current =1,2 do
                score = pi[{i-1, c_prev}] + y_hat[c_current]
                if score > pi[{i, c_current}] then
                    pi[{i, c_current}] = score
                    bp[{i, c_current}] = c_prev
                end
            end
        end
    end
    return pi, bp
end

-- Building the sequences from the backpointer
function build_sequences_from_bp(bp, gram_input)
    local predictions = torch.DoubleTensor(2*gram_input:size(1))
    -- Next position to fill in predictions (have to do it backward)
    local position = 2*gram_input:size(1)
    local col = 2
    -- Loop until the 3rd position (because 2nd is the first one, could be set by hand)
    for i=bp:size(1),3,-1 do
        -- coming from a space
        if bp[i][col] == 1 then
            predictions[position] = 1
            position = position - 1
            col = 1
        else
            col = 2
        end
        -- index i is shifted of 1 wrt local index in gram_input
        predictions[position] = gram_input[i-1]
        position = position - 1
    end
    -- Beginnning of gram_input set
    predictions[position] = gram_input[1]
    position = position - 1
```

```
    return predictions:narrow(1,position+1,predictions:size(1)-position
        )
end

function main()
    -- Parse input params
    opt = cmd:parse(arg)
    N = opt.N
    algo = opt.algo
    eta = opt.eta
    hid = opt.hid
    embed = opt.embed
    batchsize = opt.batch
    Ne = opt.Ne


    -- Reading file
    local file = hdf5.open('data_preprocessed/'..tostring(N)..'-grams.
        hdf5', 'r')
    data = file:all()
    file:close()

    train_input = data['input_matrix_train']
    train_output = data['output_matrix_train']
    input_data_train = data['input_data_train']

    input_data_valid = data['input_data_valid_nospace']:clone()

    input_data_test = data['input_data_test']:clone()

    -- Building the model
    torch.manualSeed(1)

    nnlm1, crit = build_model(N-1, 49, 2, embed, hid)

    print('-> Training the model')
    train_model(train_input, train_output, nnlm1, crit, N-1, 2, eta, Ne
        , batchsize)

    print('Ngram size '..tostring(N))
    print('Train Perplexity')
    print(compute_perplexity(input_data_train, nnlm1, N))
    print('Valid Perplexity')
    print(compute_perplexity(input_data_valid, nnlm1, N))
```

```

    -- Prediction
    if (algo == 'greedy') then
        predictions_test = predict_NN_greedy(input_data_test, nnlm1, N)
    elseif (algo == 'viterbi') then
        pi, bp = predict_count_based_viterbi(input_data_test, nnlm1, N)
        predictions_test = build_sequences_from_bp(bp, input_data_test)
    else
        error("invalid algorithm input")
    end

    -- Kaggle format
    num_spaces = get_kaggle_format(predictions_test, N)

    print(num_spaces:narrow(1,1,10))

    -- -- Saving the Kaggle format output
    -- myFile = hdf5.open('submission/'..opt.f, 'w')
    -- myFile:write('num_spaces', num_spaces)
    -- myFile:close()
end

main()
```

<u>RNN:</u>

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

-- Only requirements allowed
require("hdf5")
require("rnn")
require 'helper.lua';

cmd = torch.CmdLine()

-- Cmd Args
cmd:option('-l', 30, 'Length size for the training sequence')
cmd:option('-b', 16, 'Batch-size for the training')
```

```lua
cmd:option('-edim', 20, 'Embed dimension for the characters embeddings')
cmd:option('-eta', 0.5, 'Learning rate')
cmd:option('-ne', 4, 'Number of epochs for the training')
cmd:option('-s', 1, 'Step size for the adaptive eta changes')
cmd:option('-f', 'pred_test_rnn.f5', 'File name for the predictions on the test')
cmd:option('-model', 'RNN', 'Recurrent model to be used (RNN, LSTM or GRU')


-- Formating the input
-- input is a 1d tensor
function get_train_input(input, len, batch_size)
    -- Building output (we put predict a padding at the end)
    local n = input:size(1)

    -- Get the closer multiple of batch_size*len below n
    local factor = -math.floor(-n/(len*batch_size))
    local n_new = factor*len*batch_size
    local input_new = torch.DoubleTensor(n_new)
    local t_input, t_output
    input_new:narrow(1,1,n):copy(input)
    input_new:narrow(1,n,n_new-n+1):fill(2) -- Filling with padding

    -- Building output
    local output = get_output(input_new)

    -- Issue with last sequence if batch_size does not divide n
    t_input = torch.split(input_new:view(batch_size,n_new/batch_size), len, 2)
    t_output = torch.split(output:view(batch_size,n_new/batch_size),len, 2)
    return t_input, t_output
end

function get_output(input)
    local n = input:size(1)
    local output = torch.DoubleTensor(n)
    for i=2, n do
        if input_new[i] ~= 1 then
            output[i-1] = 2
        else
            output[i-1] = input[i]
        end
```

```
    end
    output[n] = 2
    return output
end

-- Methods to build the model
function build_RNN(embed_dim, rho)
    return nn.Recurrent(embed_dim, nn.Linear(embed_dim, embed_dim),nn.Linear(embed_dim, embed_dim), nn.Tanh(), rho)
end

function build_LSTM(embed_dim, rho)
    return nn.FastLSTM(embed_dim, embed_dim, rho)
end

function build_GRU(embed_dim, rho, dropout_p)
    return nn.GRU(embed_dim, embed_dim, rho,dropout_p)
end

function build_rnn(embed_dim, vocab_size, batch_size, recurrent_model, len)
    local batchRNN
    local params
    local grad_params
    -- generic RNN transduced
    batchRNN = nn.Sequential()
        :add(nn.LookupTable(vocab_size, embed_dim))
        :add(nn.SplitTable(1, batch_size))
    local rec = nn.Sequencer(recurrent_model)
    rec:remember('both')

    batchRNN:add(rec)

    -- Output
    batchRNN:add(nn.Sequencer(nn.Linear(embed_dim, 2)))
    batchRNN:add(nn.Sequencer(nn.LogSoftMax()))

    -- Retrieve parameters (To do only once!!!)
    params, grad_params = batchRNN:getParameters()
    -- Initializing all the parameters between -0.05 and 0.05
    for k=1,params:size(1) do
        params[k] = torch.uniform(-0.05,0.05)
    end

    return batchRNN, params, grad_params
```

```
end

function train_model_with_perp(t_input, t_output, model, model_flattened, params_flattened,
        params, grad_params, criterion, eta, nEpochs, batch_size, len, n, input_valid, output_valid, step)
    -- Train the model with a mini batch SGD
    -- Uses an adaptive learning rate eta computed each cycle of step iterations from the
    -- evolution of the perplexity on the validation set (compute with the model_flattened)
    local timer
    local pred
    local loss
    local dLdPred
    local t_inputT = torch.DoubleTensor(len,batch_size)
    local t_output_table
    local size

    -- To store the loss
    local av_L = 0
    local perp = 0
    local old_perp = 0

    for i = 1, nEpochs do
        -- timing the epoch
        timer = torch.Timer()
        old_L = av_L
        old_perp = perp
        av_L = 0

        -- mini batch loop
        for k = 1, n/(batch_size * len) do
            -- Mini batch data

            t_inputT:copy(t_input[k]:t())
            t_output_table = torch.split(t_output[k],1,2)
            --format the output
            for j=1,len do
                t_output_table[j] = t_output_table[j]:squeeze()
            end

            -- reset gradients
            grad_params:zero()

```

```lua
            -- Forward loop
            pred = model:forward(t_inputT)
            loss = criterion:forward(pred, t_output_table)
            av_L = av_L + loss

            -- Backward loop
            dLdPred = criterion:backward(pred, t_output_table)
            model:backward(t_inputT, dLdPred)

            -- gradient normalization with max norm 5 (l2 norm)
            grad_params:view(grad_params:size(1),1):renorm(1,2,5)
            model:updateParameters(eta)

        end

        print('Epoch '..i..': '..timer:time().real)
        print('Average Loss: '..av_L/math.floor(n/batch_size))
        -- Print perplexity validity every step of iteration
        if (i%step == 0) then
            size = input_valid:size(1) - 1
            params_flattened:copy(params)
            perp = compute_perplexity(input_valid:narrow(1,1,size):view(size,1), output_valid, model_flattened)
            print('Valid perplexity: '..perp)

            if old_perp - perp < 0 then
                eta = eta/2
            end

            if (eta < 0.0001) then eta = 0.1 end

        end
    end
end

-----------------------------------------
----- Methods for prediction
-----------------------------------------

function compute_probability_model(model, input)
    return model:forward(input:view(input:size(1), 1))
end

-- Method to compute manually the perplexity
function compute_perplexity(input, output, model)
```

```lua
    -- Last Position filled in predictions
    -- Position to predict in input
    local position_input = 1
    local probability = torch.DoubleTensor(2)
    local probability_table
    local perp = 0

    -- Build mapping
    for i = 1,input:size(1) do
        -- Line where the model appears
        -- The model remember the states before, just need to feed into it a character
        probability_table = compute_probability_model(model, input:narrow(1,i,1))
        probability:copy(probability_table[1])
        perp = perp + probability[output[i]]
    end
    -- Cutting the output
    return math.exp(-perp/input:size(1))
end

-- Prediction with greedy algorithm
function predict_rnn_greedy(input, len, model)
    -- Last Position filled in predictions
    local position_prediction = 1
    -- Position to predict in input
    local position_input = 1
    -- We allocate the maximum of memory that could be needed
    -- Default value is -1 (to know where predictions end afterwards)
    local predictions = torch.ones(2*input:size(1)):mul(-1)
    -- Copy the first entry
    predictions[position_prediction] = input[position_input]
    local probability = torch.zeros(2)
    local probability_table

    -- Build mapping
    while position_input < input:size(1) do
        -- Line where the model appears
        -- The model remember the states before, just need to feed into it a character
        probability_table = compute_probability_model(model, predictions:narrow(1,position_prediction, 1))
        probability:copy(probability_table[1])

        m,a = probability:max(1)
```

```
        -- Case space predicted
        position_prediction = position_prediction +1
        if (a[1] == 1) then
            predictions[position_prediction] = 1
        else
            -- Copying next character
            position_input = position_input + 1
            predictions[position_prediction] = input[position_input]
        end
    end
    -- Cutting the output
    return predictions:narrow(1,1,position_prediction)
end

function main()
    -- Parse input params
    opt = cmd:parse(arg)

    -- Reading file
    N = 2
    local data = hdf5.open('../data_preprocessed/'..tostring(N)..'-
        grams.hdf5','r'):all()
    F_train = data['F_train']
    input_data_valid = data['input_data_valid']
    input_matrix_train = data['input_matrix_train']
    input_data_train = data['input_data_train']
    input_data_valid_nospace = data['input_data_valid_nospace']
    input_data_test = data['input_data_test']
    myFile:close()

    F_train = data['F_train']
    input_data_valid = data['input_data_valid']
    input_data_train = data['input_data_train']
    input_data_test = data['input_data_test']
    input_data_valid_nospace = data['input_data_valid_nospace']

    -- Model parameters
    len = opt.l
    batch_size = opt.b
    vocab_size = 49
    embed_dim = oopt.edim
    eta = opt.eta
    nEpochs = opt.ne
    step = opt.s
```

```

    -- Formating data
    t_input_new, t_output_new = get_train_input(input_data_train, len, batch_size)
    output_valid = get_output(input_data_valid)
    n_new = len * batch_size *(#t_input_new)

    -- Building model
    model, params, grad_params = build_rnn(embed_dim, vocab_size, batch_size, build_RNN(embed_dim, len), len)
    model_valid, params_valid, grad_params_valid = build_rnn(embed_dim, vocab_size, 1,build_RNN(embed_dim))

    crit = nn.SequencerCriterion(nn.ClassNLLCriterion())

    -- Training model
    train_model_with_perp(t_input_new, t_output_new, model, model_valid, params_valid,
        params, grad_params, crit, eta, nEpochs, batch_size, len, n_new, input_data_valid, output_valid, step)
    print('here')

    -- -- Computing RMSE on valid
    -- kaggle_true_valid = get_kaggle_format(input_data_valid,2)

    -- timer = torch.Timer()
    -- pred_valid = predict_rnn_greedy(input_data_valid_nospace:narrow(1,1,input_data_valid_nospace:size(1)), len, model_valid)
    -- print('Greedy prediction on validation set (Time elasped : '..timer:time().real..' )')
    -- kaggle_model_valid = get_kaggle_format(pred_valid,2)
    -- print('RMSE')
    -- rsme = compute_rmse(kaggle_true_valid, kaggle_model_valid)
    -- print(rsme)

    -- -- Prediction on test
    -- timer = torch.Timer()
    -- size = input_data_test:size(1)
    -- pred_test = predict_rnn_greedy(input_data_test:narrow(1,1,size), len, model_valid)
    -- print('Greedy prediction on test set (Time elasped : '..timer:time().real..' )')
    -- kaggle_test = get_kaggle_format(pred_test,2)

    -- -- Saving the Kaggle format output
```

```
    -- myFile = hdf5.open('../submission/'..opt.f, 'w')
    -- myFile:write('num_spaces', kaggle_test)
    -- myFile:close()
end
```