HW3: (Neural) Language Modeling

Nicolas Drizard
nicolasdrizard@g.harvard.edu

Virgile Audi
vaudi@g.harvard.edu

April 1, 2016

Github: https://github.com/virgodi/cs287/HW3

1 Introduction

This assignment focuses on the task of language modeling, a crucial first-step for many natural language applications. In this report, we will present several count-based multinomial language models with different smoothing methods, an influential neural network based language model from the work of Bengio et al. (2003), and an extension to this language model which learns using noise contrastive estimation, as well as their implementation using Torch. We found this homework more challenging than the previous ones and encountered significant challenges that we will underline in this report.

2 Problem Description

The goal of the language models presented in this report is to learn a distributed representation for words as well as probability distribution for word sequences. Language models are usually represented as the probability of generating a new word conditioned on the preceeding words:

$$P(\boldsymbol{w}_{1:n}) = \prod_{i=1}^{n-1} P(w_{i+1}|w_i)$$

To simplify the analysis, it is common to make the assumption that a word is influenced only by the N words immediately preceeding it, which we call the context. Even with reasonably small values for N, building such models are extremely expensive computationally-wise as well as time-consuming if not ran on GPU. The joint proabibility of a sequence of 6 words taken from a vocabulary of 10 000 words could possibly imply training the model to fit up to $10^{4^6} - 1 = 10^{24} - 1$ parameters.

The objective of this homework was to predict a probability distribution over 50 words at a given place in the sentence and based on the previous words. To do so, we tried implementing N-grams models in an efficient manner. Due to computational limitations (no access to GPUs...), we faced difficulties training the Neural Network and focused our efforts on building strong count-based models to solve the problem of language modeling.

3 Model and Algorithms

We will now dive into more details of two types of models, i.e. count-based models and neural network models.

3.1 Count-based Models

The first models we implemented are based on counts of N-gram features. The main idea is here to implement n-gram multinomial estimates for $p(w_i|w_{i-n+1}, \ldots, w_{i-1})$. We will use different smoothing methods and compare their performance.

Maximum Likelihood Estimation The first approach is just to compute the maximum likelihood estimation $p_{ML}(w|c)$ where c is the context (i.e. N-1 gram) and w the word predicted. This is done by simply counting the frequency of the N-gram: $(c - w)$ in the training.

$$p_{ML}(w|c) = \frac{F_{c,w}}{F_c} = \frac{F_{c,w}}{\sum_i F_{c,w_i}}$$

with $F_{c,w} = \sum_i \mathbb{1}(w_{i-n+1:i-1} = c, w_i = w)$.

We applied this method up to 5-grams. But the results are not very impressive. First, the long tail in the words distribution is not properly represented as we consider the number of occurences of such words as the ground truth. Second, this method works for one context at a time but it should be more interesting to weight and combine them.

Laplace smoohting This approach handles the first issue where we want to have a better representation for the words in the tail. Here we simply add an off-set to each count $\hat{F}_{c,w} = F_{c,w} + \alpha$.

We can use the validation set to tune the α parameter while optimizing its perplexity. We applied this pipeline on the validation set from Kaggle.

Witten-Bell smoothing Here we tackle the second issue of the MLE. The idea is to combine lower order models together to use as much information as possible. The global idea is to use recursivity:

$$p(w|c) = \lambda p_{ML}(w|c) + (1 - \lambda)p(w|c')$$

with c' the lower order context (if c is the N-1 gram, c' is the N-2 gram). We stop it when c' is empty and then use the prior on the words distribution from the train (ie the frequency of the word w only).

Witten and Bell suggested a way to compute the constant: $\lambda = \frac{N1_c}{F_c + N1_c}$ with $N1_c = |\{w : F_{c,w} = 1\}|$

This method is usually called interpolation as we interpolate $p(w|c)$ with its MLE estimations at each lower order context. Another method is known as back-off where we use the lower order context only if $F_{w,c} = 0$. We applied a mixed of the two methods, ie we use interpolation and

jumped directly to the lower order context if the count was 0. Also we applied an alpha smoothing for the 2-gram (ie the lowest order before the word prior). This was motivates by the need to smooth the importance of the 2-grams as the counts are higher for lower order context.

Modified Kneser-Ney smoothing This method also uses a mixture of interpolation and back-off but instead of using the MLE for the current context, it uses absolute discounting. The idea is that we would like to weigh down the count $F_{w,c}$ but differently with regards to their value. As a result, we introduce a discounting parameter which takes discrete values with regards to $F_{w,c}$.

$$p(w|c) = \frac{F_{c,w} - D(F_{c,w})}{F_c} + \gamma(F_{c,w})p(w|c)$$

with :

$$D(F_{c,w}) \begin{cases} 0, & \text{if } F_{c,w} = 0 \\ D_1, & \text{if } F_{c,w} = 1 \\ D_2, & \text{if } F_{c,w} = 2 \\ D_3, & \text{if } F_{c,w} \geq 3 \end{cases}$$

$$\gamma(F_{c,w}) = \frac{D_1 N1_c + D_2 N2_c + D_3 N3_c}{F_c}$$

with $N2_c$ and $N3_c$ similarly defined as $N1_c$.

In the original paper, they used the above definition with fixed $D1$, $D2$ and $D3$ and tuned parameters. We tried both approach and found even better results with manually tuned parameters.

3.2 Neural Networ Models

3.2.1 Regular Models

As in the neural networks build for previous homeworks, the model has for input a window of words preceding the wanted predicted word. It first convert the words in the window of size d_{win} my mapping them into a geometrical space of higher dimension d_{in} (30 in Bengio's paper). It then concatenates the words embeddings into a vector of size $d_{in} \times d_{win}$. This has for advantage of adding information about the position of the words in the window, as opposed to making a bag-of-words assumption. The higher dimensional representation of the window is then fed into a first linear model 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 vocabulary. We then train the model using a Negative Log-Likelihood criterion and stochastic gradient descent.

We can summarize the model in the following formula:

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

where we recall that:

- $\boldsymbol{x} \in \Re^{d_{in} \cdot d_{win}}$ is the concatenated word embeddings
- $\boldsymbol{W} \in \Re^{(d_{in} \cdot d_{win}) \times d_{hid}}$, and $\boldsymbol{b} \in \Re^{d_{hid}}$
- $\boldsymbol{W'} \in \Re^{d_{hid} \times |\mathcal{V}|}$, and $\boldsymbol{b'} \in \Re^{|\mathcal{V}|}$, where $|\mathcal{V}|$ is the size of the vocabulary.

We give a diagram of the model to better illustrate it:

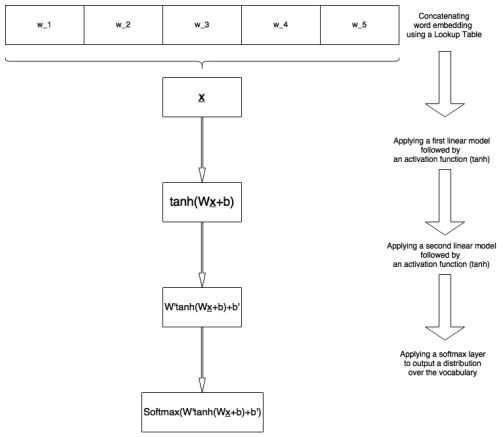


Figure 1: Neural Language Model (Bengio,2003)

We then implemented a variant of the model using a skip-layer that concatenates the output of the tanh layer again with the original embeddings. The updated formula for the model is:

$$nnlm_2(\boldsymbol{x}) = [\tanh(\boldsymbol{x}\boldsymbol{W} + \boldsymbol{b}), \boldsymbol{x}]\boldsymbol{W'} + \boldsymbol{b'}$$

where this time:

- $\boldsymbol{W'} \in \Re^{(d_{hid} + d_{in} \cdot d_{win}) \times |\mathcal{V}|}$, and $\boldsymbol{b'} \in \Re^{|\mathcal{V}|}$

The updated diagram is as follows:

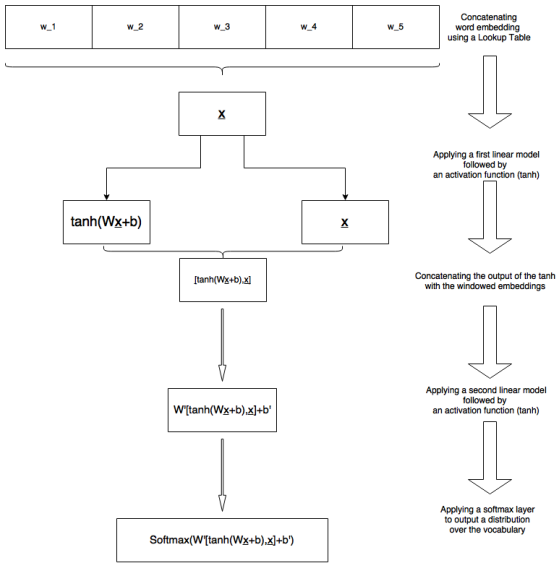


Figure 2: Skip-Layer Model

We now show the pseudo code for training these NNLMs using batch stochastic gradient descent:

procedure $NNLM_i(win_1, ..., win_n$,MaxEpoch,BatchSize,LearningRate)
  for epoch = 1,MaxEpoch **do**
    for batch = 1, |train|/BatchSize **do**
      for win in batch **do**
        Call $NNLM_i$:forward(win)
        Evalute the loss
        Evaluate derivatives of the loss
        Backprop through $NNLM_i$
      Update Parameters with LearningRate

3.2.2 Noise Contrastive Estimation

As mentioned earlier, training such model is extremely expensive in computation, especially on CPUs. The issue comes from the use of the softmax in the last layer of the model in order to obtain a distribution on a large vocabulary. In order to speed up the training time and reduce compution, we tried to implement NCE, which is a method used to fit unnomarlised method and therefore avoids using the last softmax layer.

The principle behind NCE is to sample for every context in the training data K wrong words that do not appear next in the windows using a noise distribution such a multinomial of the vocabulary simplex. We then apply a log-bilinear model. For a given context $\boldsymbol{x}$ and target y, the probability of y being a correct word for this context is given by:

$$p(D = 1|\boldsymbol{x}, \boldsymbol{y}) = \sigma(\log p(\boldsymbol{y}|D = 1, \boldsymbol{x}) - \log Kp(\boldsymbol{y}|D = 0, \boldsymbol{x}))$$

where

$$p(D = 1|\boldsymbol{x}, \boldsymbol{y}) = \sigma(log p(\boldsymbol{y}|D = 1, \boldsymbol{x}) - log(Kp(\boldsymbol{y}|D = 0, \boldsymbol{x})))$$

If the $p(y|D = 1, x)$ term still forces some normalisation in the model, thanks to the contribution of Mnih and Teh (2012), we can estimate the normalisation constant as a parameter in the model and even set to equal to 1. We can therefore replace this term by the score outpute by the linear model $z_{\boldsymbol{x_i}, w_i}$. The objective function that needs to be minimised becomes:

$$\mathcal{L}(\theta) = \sum_i \log \sigma(z_{\boldsymbol{x_i}, w_i}) - \log(Kp_{ML}(w_i))) + \sum_{k=1}^{K} \log(1 - \sigma(z_{\boldsymbol{x_i}, s_k}) - \log Kp_{ML}(s_k))$$

where p_{ML} is the pdf of the noise distribution, and s_k for $k \in \{1, ..., K\}$ are samples from the noise distribution.

4 Experiments

We now present the results of our experiments. We will first talk about the preprocessing and then continue with a comparison of the different models.

4.1 Data and Preprocessing

To complete this homework, we were given 3 datasets, one for training, one for validation and one for testing. The train set consisted of sentences with a total of over eight hundred thousands words from a vocabulary of ten thousands words. The validation set consisted of 70391 words. The particularity of the issue at hand consisted in the fact that we had to only predict a probability distribution over 50 words and not on the entire vocabulary. This is why we were provided the same validation set in the same format as for the test set. We could therefore predict 3370 words on the validation set to help us predict the 3361 words of the test set.

Most of the preprocessing was about building the N-grams for the different sets. We included in the preprocess.py file different functions to evaluate the windows as well as counting the different occurences of each N-grams. For instance, looking at 6-grams gave:

- 772 670 unique 6-grams on the training set,
- 887 522 6-grams in total,
- 70 391 6-grams on the validation set,
- 3 370 words to predict on the validation set,
- and 3 361 words of the test set

4.2 Evaluation

To evaluate the models, we will use the perplexity measure. For a set of m N-grams, $\boldsymbol{w}_1, ..., \boldsymbol{w}_m$, it is defined to be:

$$P(w_1, ..., w_m) = \exp\left(-\frac{1}{m}\sum_{i=1}^{m}\log P(w_N^i | w_{N-1}^i, ..., w_1^i)\right)$$

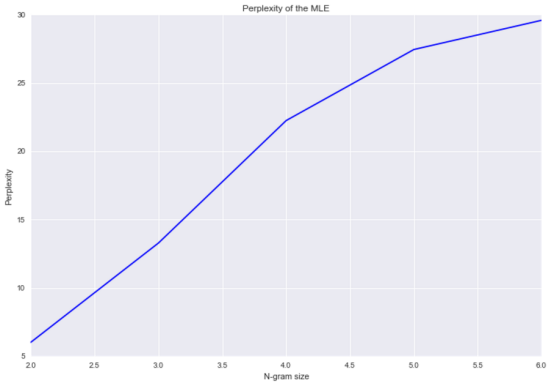
In other words, the perplexity translates how likely is the predicted word given the previous N-1 words. In this report, we will evaluated perplexity both on the entire vocabulary but also on the reduced 50 words to predict from. Values between these two "different" perplexities wil range from 3 to 1000.

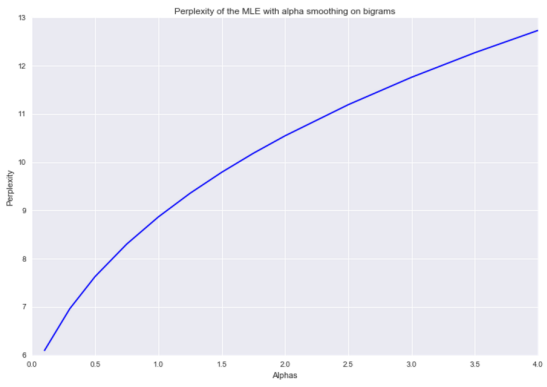
4.3 Count-based Models

We first present the results of the different count-based models:

Maximum Likelihood Estimation For this first model, the only hyperparameter is the number of N-grams in the feature. We applied this method on different N-gram models and provides the results. The computation time remains very low for each model; the only issue as we increase the size of the N-gram is the memory footprint.

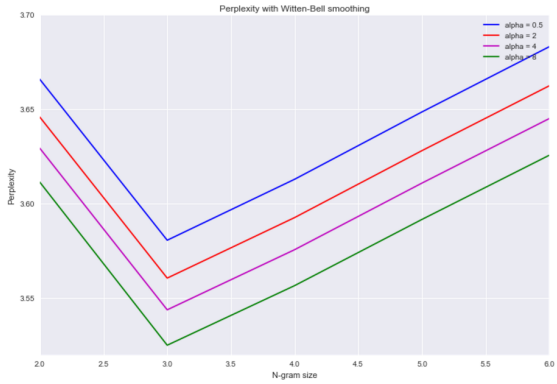
Laplace Smoothing We studied the impact of the alpha parameter and the best value with regard to the perlexity on the Kaggle development set.





Witten-Bell Smoothing For the Witten-Bell Smoothing, we benchmarked both the alpha parameter and the context from which to start the interpolation.

First, we applied a different normalization when we were using our model for the Kaggle dataset. For each entry in this dataset, we build a language model on a really small subset of the training dictionary (50 out of 10 001 words). As a result, only a small fraction of all the N-grams can be considered and we decided to compute the factor $N1$ and F_c only among the output vocabulary (50 words). This change lead to better results, we directly applied it on the modified Kneser-Ney.

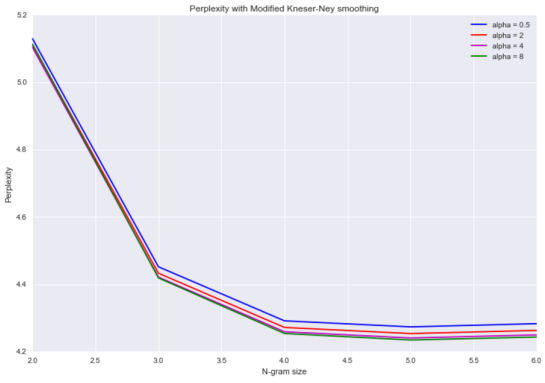


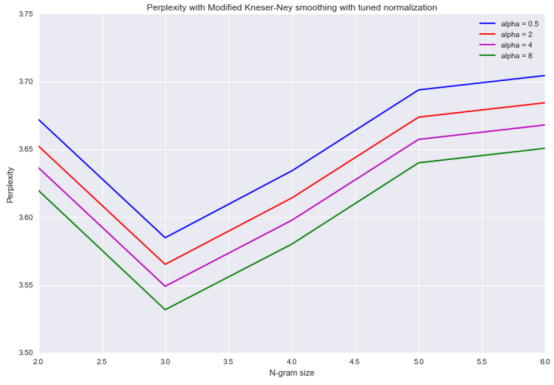
This plots shows that the Witten-Bell smoothing does not manage to successfully balance the different orders of the models when they are too many. The best perplexity is always reached for the tri-grams. This may be cause by the fact that with large context (N¿2), the probability may be disturbed by a lower order model very present even though the rest of the context is not (see the example of SAN FRANCISCO in the paper).

Moreover, we see that the perplexity gets smaller as α increases. This is because we may want a strong smoothing of the lower order model (here we use alpha only on the lower order model computed). The relative differences are bigger in the lower order models as the contexts are there more frequent (because smaller) so large smoothing may decrease their influence and favorise the long tail.

Modified Kneser-Ney This model introduces different hyperparameters. The absolute discounting D induces 2 kinds of hyperparameters: the number of values it takes and the values. We first applied those suggested in the paper but also decided to tune them manually as the paper provides better results in that case. We also observed this result.

We plot the results with a model with fixed parameters D as suggested in the paper. As with the Witten-Bell smoothing, we evaluated the models against different starting contexts. In that case, the best results were reached for the 3-grams.





The best perplexity was reached with a manually tuned D (on the Kaggle development set with the perplexity). The perplexity reached is 3.476.

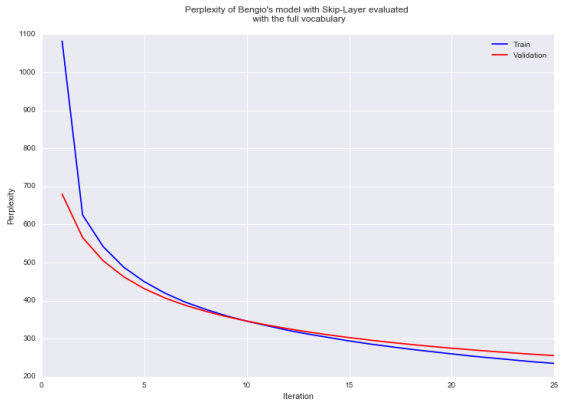
4.4 Neural Models

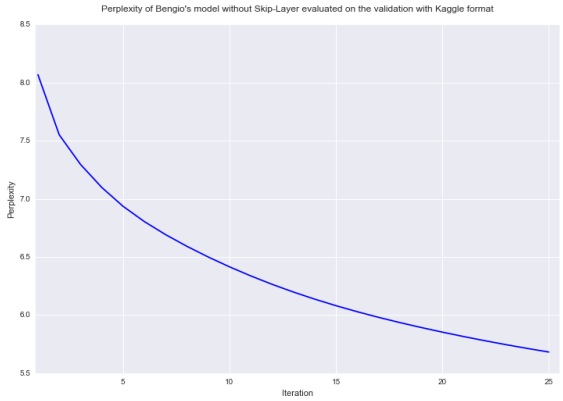
The main issue we faced with Bengio's model was training time. Even we managed to have a working model with a smaller vocabulary, we struggled at first to get a code that ran fast enough to experiment extensively with different paremetrisations. Our original code ran one epoch in about one hour for the paremetrisation. While trying to code the NCE approximation, we nevertheless managed to cut the training time to about 14-15 minutes.

We started to train the more simple neural network i.e. the one without the skip-layer, with the parametrisation suggested by Bengio:

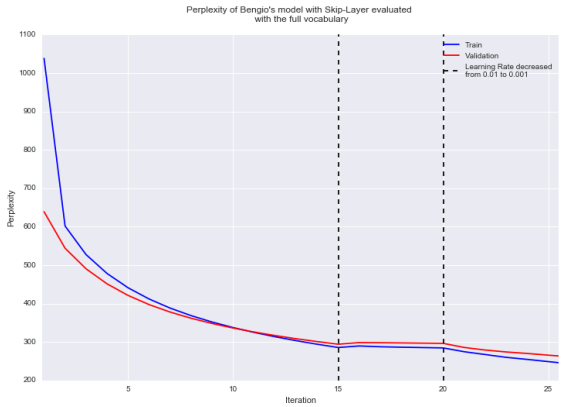
- Window size: $d_{win} = 5$
- Dimension of the embeddings: $d_{in} = 30$
- Hidden dimension: $d_{hid} = 100$

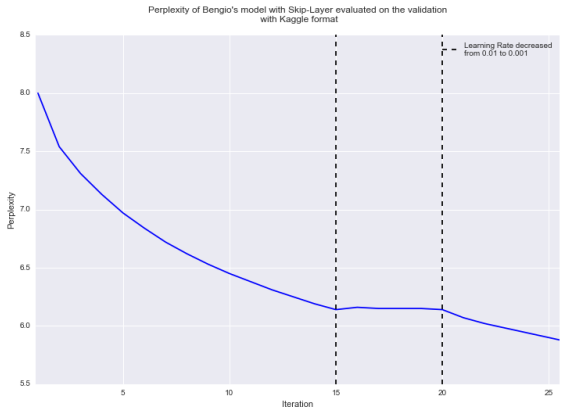
We summarize the results in the graphs below:





Before making a submission on kaggle, we decided to compare the encouraging results with the Skip-Layer model. We ran the experiment 5 epochs at a time to give us control on the learning rate. We thought that after the 15 epochs, the model was close to convergence, and decided to decrease the learning rate from 0.01 to 0.001. We obtain the following results:





As one can see, changing the learning rate negatively impacted the results. It plateaued but perplexity started decreasing again as soon as we re-up the learning rate. It also unclear how much improvement the skip-layer model brings compared to the original Bengio model, and this even without the change in learning rate. Based on these observation, we decided to submit to Kaggle the results of the simple model. We obtained:

$$Perp^{test}_{nnlm} = 5.47$$

Nevertheless, a final observation on the training of these NNLMs is that the models haven't seem to converge fully after 25 epochs. Running the algorithm for 5-10 extra epochs would most probably yielded better results and helped us reach the level of the count-base models.

4.5 NCE

We unfortunately did not succeed to implement a valid version of the Noise Contrastive Estimation. We did not managed to have a speed improvement and even worse observed that the perplexity on the validation was increasing instead of decreasing.

4.6 Mixtures of models

In order to increase our score, we decided to combine the differente approaches by averaging the results over the distributions outputted by various models.

We managed to reach our best perplexity with a weigthed mean of the Witten-Bell smoothing, the modified Kneser-Kney with tuned parameters and local normalization and the Neural Network.

Formally, the output is built as follows:

$$p(w|c) = \frac{2p_{WB}(w|c) + p_{mKN}(w|c) + p_{NN}(w|c)}{5}$$

We reached on the Kaggle development set: $perp_{Kaggle} = 3.36$
We provide here a summary of our models on the Kaggle development set:

Table 1: Final Performances

Model	Perplexity (dev Kaggle)
MLE with Laplace smoothing	6.01
Witten Bell (trigram)	3.52
Modified Kneser Kney	3.47
Neural Network	5.68
Neural Network with skip Layer	5.66
Ensemble (WB + mKN + NNsl)	**3.36**

5 Conclusion

In this homework we successfully build very efficient count-based Language models that yield great results. On the other hand, we were surprised to see that these models were beating in performance the model presented by Bengio which is much more elaborated. Nevertheless, we obtained the best results by taking a mixture of the two different types of results. These could be explained by the fact that the stochastic gradient descent may converge to different local minimums.

6 References

- Bengio, Y., Ducharme, R., Vincent, P., and Janvin, C. (2003). A neural probabilistic language model. *Journal of Machine Learning Research*, 3:11371155.
- Mnih, A. and Kavukcuoglu, K. (2013). Learning word embeddings efficiently with noise-contrastive estimation. *In Advances in Neural Information Processing Systems*, pages 22652273.
- Chen, S. and Goodman, J. (1998). An Empirical Study of Smoothing Techniques for Language. *Technical Report TR-10-98*, Computer Science Group, Harvard University.

Appendices

Preprocessing:

```
#!/usr/bin/env python

"""Language modeling preprocessing
"""

import numpy as np
import h5py
import argparse
import sys
import re
import codecs
from collections import Counter

# Your preprocessing, features construction, and word2vec code.


def get_words2index(filename):
    '''
    Loading the tags to index mapping
    '''
    words2index = {}
    with open(filename) as f:
        for line in f:
            (val, key, num) = line.split()
            words2index[key] = int(val)
    return words2index


def get_index2words(filename):
    '''
    Loading the tags to index mapping
    '''
    index2words = {}
    with open(filename) as f:
        for line in f:
            (val, key, num) = line.split()
            index2words[int(val)] = key
    return index2words

index2words = get_index2words('data/words.dict')
index2words1000 = get_index2words('data/words.1000.dict')
```

```
42
43
44 def valid_test_Ngram(filepath, words2index, N, test=False):
45     results = []
46     if test == False:
47         with open(filepath) as f:
48             i = 1
49             for line in f:
50                 lsplit = line.split()
51                 if lsplit[0] == 'Q':
52                     topredict = np.array([words2index[x] for x in
                           lsplit[1:]])
53                 if lsplit[0] == 'C':
54                     l = np.append(
55                         np.repeat(words2index['<s>'], N-1), [
                               words2index[x] for x in lsplit[1:-1]])
56                     lastNgram = l[-N+1:]
57                     results.append((lastNgram, topredict))
58     else:
59         with open(filepath) as f:
60             i = 1
61             for line in f:
62                 lsplit = line.split()
63                 if lsplit[0] == 'Q':
64                     topredict = np.array([words2index[x] for x in
                           lsplit[1:]])
65                 if lsplit[0] == 'C':
66                     l = np.append(
67                         np.repeat(words2index['<s>'], N-1), [
                               words2index[x] for x in lsplit[1:-1]])
68                     lastNgram = l[-N+1:]
69                     results.append((lastNgram, topredict))
70     return results
71
72
73 def train_get_ngram(filename, words2index, N):
74     '''
75     Generating N-grams
76     '''
77     results = []
78     with open(filename) as f:
79         for line in f:
80             lsplit = [words2index[x] for x in line.split()]
81             l = np.append(np.repeat(words2index['<s>'], N-1), lsplit)
82
```

```
83              for i in range(len(lsplit)):
84                  g = l[i:N-1+i]
85                  v = lsplit[i]
86                  results.append((g, v))
87          results.append((l[-N+1:], words2index['</s>']))
88      return results
89
90  def tomatrix(results, train=True, count = True):
91
92      N = len(results[0][0])+1
93
94      if train:
95          if count:
96              tuplelist = []
97              for i in range(len(results)):
98                  tuplelist.append(
99                      tuple(list(np.append(results[i][0], results[i][1]))
                            ))
100             Count = Counter(tuplelist).most_common()
101             tooutput = np.empty((len(Count), N+1))
102
103             for i in range(len(Count)):
104                 tooutput[i, :] = np.append(np.array(Count[i][0]), Count
                        [i][1])
105
106             return tooutput.astype(int)
107
108         else:
109             tooutput_ = np.empty((len(results),N))
110             for i in range(len(results)):
111                 tooutput_[i,:] = np.append(results[i][0],results[i][1])
112             return tooutput_
113
114     else:
115         tooutput = np.empty((len(results), 50+N-1))
116
117         for i in range(len(results)):
118             tooutput[i, :] = np.hstack((results[i][1], results[i][0]))
119
120         return tooutput.astype(int)
121
122 def validation_kaggle(filepath):
123     it = 0
124     results = []
125     with open(filepath) as f:
```

```
126         for line in f:
127             if it == 0 :
128                 it+=1
129             else:
130                 lsplit = line.split(',')
131                 l = [int(x.rstrip()) for x in lsplit[1:]]
132                 results.append(l)
133     return np.array(results)
134
135
136 def get_prior(filepath, words2index):
137     '''
138     Case N=1: ie prior on the word distribution from the train text
139     '''
140     counter = Counter()
141     with open(filepath) as f:
142         lines = f.readlines()
143         for line in lines:
144             # Adding the end of line prediction
145             lsplit = line.split() + ['</s>']
146             counter.update(lsplit)
147     # Build count matrix: (N_words, 2), col 1: word index, col2: word
        cout
148     count_matrix = np.zeros((len(counter), 2), dtype=int)
149
150     for i,t in enumerate(counter.iteritems()):
151         k, v = t
152         count_matrix[i, 0] = words2index[k]
153         count_matrix[i, 1] = v
154     return count_matrix
155
156
157 FILE_PATHS = ("data/train.txt",
158                         "data/train.1000.txt",
159               "data/valid.txt",
160               "data/valid_blanks.txt",
161               "data/test_blanks.txt",
162               "data/words.dict",
163               "data/words.1000.dict",
164               "data/valid_kaggle.txt")
165 args = {}
166
167
168 def main(arguments):
169     global args
```

```
170     parser = argparse.ArgumentParser(
171         description=__doc__,
172         formatter_class=argparse.RawDescriptionHelpFormatter)
173     parser.add_argument('--N', default=3, type=int, help='Ngram size')
174     args = parser.parse_args(arguments)
175     N = args.N
176     train, train1000, valid_txt, valid, test, word_dict, word_dict_1000
            , kaggle = FILE_PATHS
177
178     words2index = get_words2index(word_dict)
179     words2index1000 = get_words2index(word_dict_1000)
180     index2words = get_index2words(word_dict)
181
182     train_list = train_get_ngram(train, words2index, N)
183     train_matrix_count = tomatrix(train_list)
184     train_matrix = tomatrix(train_list,True,False)
185
186     train_list_1000 = train_get_ngram(train1000, words2index1000, N)
187     train_matrix_1000_count = tomatrix(train_list_1000,True,False)
188     train_matrix_1000 = tomatrix(train_list_1000)
189
190     valid_txt_list = train_get_ngram(valid_txt, words2index, N)
191     valid_txt_matrix= tomatrix(valid_txt_list,True,False)
192
193     valid_list = valid_test_Ngram(valid, words2index, N)
194     valid_matrix = tomatrix(valid_list, False)
195
196     test_list = valid_test_Ngram(test, words2index, N, True)
197     test_matrix = tomatrix(test_list, False)
198
199     valid_kaggle = validation_kaggle(kaggle)
200
201     filename = str(N) + '-grams.hdf5'
202     with h5py.File(filename, "w") as f:
203
204         f['train'] = train_matrix_count
205         f['train_1000_nocounts'] = train_matrix_1000
206         f['train_1000'] = train_matrix_1000_count
207         f['train_nocounts'] = train_matrix
208         f['valid_txt'] = valid_txt_matrix
209         f['valid'] = valid_matrix
210         f['valid_output'] = valid_kaggle
211         f['test'] = test_matrix
212         f['nwords'] = np.array([np.max(index2words.keys())])
213
```

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

Count-Based Models:

```
 1 ----------------
 2 -- helper
 3 ----------------
 4
 5 -- Loading train of the gram_size N
 6 function get_train(N)
 7     local filename = N .. '-grams.hdf5'
 8     --print(filename)
 9     myFile = hdf5.open(filename,'r')
10     train = myFile:all()['train']
11     myFile:close()
12     return train
13 end
14
15 function perplexity(distribution, true_words)
16     -- exp of the average of the cross entropy of the true word for
          each line
17     -- true words (N_words to predict, one hot true value among 50)
18     local perp = 0
19     local N = true_words:size(1)
20     for i = 1,N do
21         mm,aa = true_words[i]:max(1)
22         perp = perp + math.log(distribution[{i, aa[1]}])
23     end
24     perp = math.exp(- perp/N)
25     return perp
26 end
27
28
29 function build_context_count(count_tensor)
30     local indexes
31     local indexN
32     -- Ngram count (depend on w and context)
33     -- {'index1-...-indexN-1': {'indexN' : count}}
34     local F_c_w = {}
35     -- F_c dict (independent of w, only context based)
36     -- {index1-...-indexN-1 : count all words in c}
37     local F_c = {}
38     -- N_c dict (independent of w, only context based)
39     -- {index1-...-indexN-1 : count unique type of words in c}
40     local N_c = {}
41
```

```
42      local N = count_tensor:size(1)
43      local M = count_tensor:size(2)
44
45      for i=1, N do
46          indexN = count_tensor[{i,M-1}]
47
48          -- build the key index1-...-indexN-1
49          indexes = tostring(count_tensor[{i,1}])
50          for j=2, M - 2 do
51              indexes = indexes .. '-' .. tostring(count_tensor[{i,j}])
52          end
53
54          -- Filling F_c_w
55          if F_c_w[indexes] == nil then
56              F_c_w[indexes] = {[indexN] = count_tensor[{i, M}]}
57          else
58              F_c_w[indexes][indexN] = count_tensor[{i, M}]
59          end
60
61          -- Updating F_c and F_c
62          if F_c[indexes] == nil then
63              F_c[indexes] = count_tensor[{i, M}]
64              N_c[indexes] = 1
65          else
66              F_c[indexes] = count_tensor[{i, M}] + F_c[indexes]
67              N_c[indexes] = 1 + N_c[indexes]
68          end
69      end
70
71      return F_c_w, F_c, N_c
72  end
73
74  ------------------
75  -- Maximum Likekihood Estimation
76  ------------------
77
78  function compute_mle_line(N, entry, F_c_w, alpha)
79      -- Compute the maximum likelihood estimation with alpha smoothing
            on the
80      -- input in entry,
81      --
82      -- Return vector (50) predicting the distribution from entry
83      -- N represent the Ngram size used in the prediction so context is
            N-1 gram
84      local prediction = torch.zeros(50)
```

```
85      local indexN
86
87      -- context (at least with one element)
88      local indexes = tostring(entry[{1, entry:size(2)}])
89      for j=entry:size(2) - 1, entry:size(2) - 1 - (N-3), -1 do
90          indexes = tostring(entry[{1, j}]) .. '-' .. indexes
91      end
92      -- check if context is unseen, otherwise go to next context
93      if F_c_w[indexes] == nil then
94          --print('unseen context')
95          prediction:fill(alpha)
96      else
97          -- Compute MLE for each word
98          for j=1,50 do
99              indexN = entry[{1, j}]
100             if F_c_w[indexes][indexN] ~= nil then
101                 prediction[j] = F_c_w[indexes][indexN] + alpha
102             else
103                 --print('unseen word')
104                 prediction[j] = alpha
105             end
106         end
107     end
108
109     return prediction:div(prediction:sum())
110 end
111
112 -- Prediction with the MLE (with Laplace smoothing, no back-off and
        interpolation)
113
114 function mle_proba(N, data, alpha)
115     -- Output format: distribution predicted for each N word along the
116     -- 50 possibilities
117     local N_data = data:size(1)
118
119     -- Train model
120     local train = get_train(N)
121     local F_c_w = build_context_count(train)
122
123     -- Prediction
124     local distribution = torch.zeros(N_data, 50)
125     for i=1, N_data do
126         distribution:narrow(1, i, 1):copy(compute_mle_line(N, data:
              narrow(1,i,1), F_c_w, alpha))
127     end
```

```
128
129     return distribution
130 end
131
132 ------------------
133 -- Witten-Bell
134 ------------------
135
136 function compute_wb_line(N, entry, F_c_w_table, alpha)
137     -- Compute the interpolated Witten-Bell model where we jump tp
        lower
138     -- order models if the context count is 0 or all the words counts
        in that
139     -- context is 0 also.
140     --
141     -- Return vector (50) predicting the distribution from entry
142     -- N represent the Ngram size used in the prediction so context is
        N-1 gram
143     -- alpha is only used for the MLE without any context
144     --
145     -- NB: the normalization is done based on the words contained in
        the first 50
146     -- columns of the entry as we are building a distribution on a sub
        sample of a
147     -- dictionnary (so we are using the count only of these words to
        normalize).
148     -- Hence the variable denom and N_c_local
149     local prediction = torch.zeros(50)
150     local indexN
151     local indexes
152     local denom
153     local N_c_local
154
155     -- case where computation only on the prior
156     if N == 1 then
157         for j=1,50 do
158             indexN = entry[{1, j}]
159             -- Corner case when prediction on words not on the dict (
                case for <s>)
160             if F_c_w_table[1][tostring(indexN)] == nil then
161                 prediction[j] = 0
162             else
163                 prediction[j] = F_c_w_table[1][tostring(indexN)][indexN
                    ] + alpha
164             end
```

```
165         end
166         -- Normalizing
167         return prediction:div(prediction:sum(1)[1])
168     else
169         -- Compute the MLE for current N
170         -- context (at least with one element)
171         indexes = tostring(entry[{1, entry:size(2)}])
172         for j=entry:size(2) - 1, entry:size(2) - 1 - (N-3), -1 do
173             indexes = tostring(entry[{1, j}]) .. '-' .. indexes
174         end
175
176         -- check if context is unseen, otherwise go to next context
177         if F_c_w_table[N][indexes] == nil then
178             --print('unseen context')
179             return compute_wb_line(N-1, entry, F_c_w_table, alpha)
180         end
181
182         -- local variable initialization
183         denom = 0
184         N_c_local = 0
185         -- Compute MLE for each word
186         for j=1,50 do
187             indexN = entry[{1, j}]
188             if F_c_w_table[N][indexes][indexN] ~= nil then
189                 prediction[j] = F_c_w_table[N][indexes][indexN]
190                 denom = denom + F_c_w_table[N][indexes][indexN] + 1
191                 N_c_local = N_c_local + 1
192             end
193         end
194
195         -- Check that MLE predicted at least one words, otherwise go to
                next context
196         if prediction:sum(1)[1] == 0 then
197             --print('unseen words')
198             return compute_wb_line(N-1, entry, F_c_w_table, alpha)
199         end
200
201         -- Combining with next context
202         prediction:add(compute_wb_line(N-1, entry, F_c_w_table, alpha):
               mul(N_c_local)):div(denom)
203         return prediction
204     end
205 end
206
207 -- Witten Bell: new version, computation done at once line by line
```

```
208 --
209 -- p_wb(w|c) = (F_c_w + N_c_. * p_wb(w|c'))/(N_c_. + F_c_.)
210 function distribution_proba_WB(N, data, alpha)
211     local N_data = data:size(1)
212     local M = data:size(2)
213
214     -- Building the count matrix for each ngram size lower than N.
215     local F_c_w_table = {}
216     for i=1,N do
217         train = get_train(i)
218         F_c_w_table[i] = build_context_count(train)
219     end
220
221     -- Vector initialisation
222     local distribution = torch.zeros(N_data, 50)
223     for i=1,N_data do
224         -- Compute witten bell for the whole line i
225         distribution:narrow(1, i, 1):copy(compute_wb_line(N, data:
              narrow(1,i,1), F_c_w_table, alpha))
226     end
227     return distribution
228 end
229
230
231 ------------------
232 --- Modified Kneser Ney
233 ------------------
234
235 -- Version tailored for modified Kneser-Ney:
236 -- Modif: now we enable a local computation of D
237 -- (that will be based on the sub vocabulary used in the validation and
      tesst)
238
239 function build_context_count_split(count_tensor, K)
240     -- count_tensor in format (N_words, N + 1):
241     -- col1, ..., colN = indexes for the Ngram, colN+1 = N_gram count
242     -- K: number of count separate cases (need K > 1, usually K = 3)
243     --
244     -- Ngram count (depend on w and context)
245     -- {'index1-...-indexN-1': {'indexN' : count}}
246     local F_c_w = {}
247     -- n_table: stores the total number of N_grams ending with indexN
248     -- with exact number of occurences stored in their key k:
249     -- {k : {'indexN': # N_grams ending with indexN with exactly k
          occurences}}
```

```
250     local n_table = {}
251     for j=1,K+1 do
252         n_table[j] = {}
253     end
254
255     local N = count_tensor:size(1)
256     local M = count_tensor:size(2)
257
258     for i=1, N do
259         local indexN = count_tensor[{i,M-1}]
260
261         -- build the key index1-...-indexN-1
262         indexes = tostring(count_tensor[{i,1}])
263         for j=2, M - 2 do
264             indexes = indexes .. '-' .. tostring(count_tensor[{i,j}])
265         end
266
267         -- Filling F_c_w
268         if F_c_w[indexes] == nil then
269             F_c_w[indexes] = {[indexN] = count_tensor[{i, M}]}
270         else
271             F_c_w[indexes][indexN] = count_tensor[{i, M}]
272         end
273
274         -- Building the key to update the corresponding part of n_table
275         if count_tensor[{i, M}] > K then
276             key_N_c = K
277         else
278             key_N_c = count_tensor[{i, M}]
279         end
280
281         -- Updating n_table
282         if count_tensor[{i, M}] <= K + 1 then
283             if n_table[count_tensor[{i, M}]][indexN] == nil then
284                 n_table[count_tensor[{i, M}]][indexN] = 1
285             else
286                 n_table[count_tensor[{i, M}]][indexN] = n_table[
                        count_tensor[{i, M}]][indexN] + 1
287             end
288         end
289     end
290
291     return F_c_w, n_table
292 end
293
```

```
294 -- V2: with local normalization on the validation sub vocabulary
295
296 function compute_mkn_line(N, entry, F_c_w_table, n_table, alpha, K, D)
297     -- Compute the Modified Kneser Ney model where we jump to lower
298     -- order models if the context count is 0 or all the words counts
            in that
299     -- context is 0 also.
300     --
301     -- Return vector (50) predicting the distribution from entry
302     -- N represent the Ngram size used in the prediction so context is
            N-1 gram
303     -- alpha is only used for the MLE without any context
304     local prediction = torch.zeros(50)
305     local indexN
306     local F_local
307     local N_c_local = {}
308     for k=1,K do
309         N_c_local[k] = 0
310     end
311     local n_table_local = {}
312     for k=1,K+1 do
313         n_table_local[k] = 0
314     end
315
316     -- case where computation only on the prior
317     if N == 1 then
318         for j=1,50 do
319             indexN = entry[{1, j}]
320             -- Corner case when prediction on words not on the dict (
                    case for <s>)
321             if F_c_w_table[1][tostring(indexN)] == nil then
322                 prediction[j] = 0
323             else
324                 prediction[j] = F_c_w_table[1][tostring(indexN)][indexN
                        ] + alpha
325             end
326         end
327         -- Normalizing
328         return prediction:div(prediction:sum(1)[1])
329     else
330         -- Compute the MLE for current N
331         -- context (at least with one element)
332         local indexes = tostring(entry[{1, entry:size(2)}])
333         for j=entry:size(2) - 1, entry:size(2) - 1 - (N-3), -1 do
334             indexes = tostring(entry[{1, j}]) .. '-' .. indexes
```

```
335         end
336         -- check if context is unseen, otherwise go to next context
337         if F_c_w_table[N][indexes] == nil then
338             --print('unseen context')
339             return compute_mkn_line(N-1, entry, F_c_w_table, n_table,
                    alpha, K, D)
340         end
341
342         -- Building local n_table
343         for j=1,50 do
344             indexN = entry[{1, j}]
345             -- Updating local n_table
346             for k=1,K+1 do
347                 -- Possible Case where there is no Ngrams ending with
                        indexN with count of K
348                 if n_table[N][k][indexN] ~= nil then
349                     n_table_local[k] = n_table_local[k] + n_table[N][k
                            ][indexN]
350                 end
351             end
352         end
353
354         -- Check no 0 in n_table_local
355         for k=1,K+1 do
356             if n_table_local[k] == 0 then
357                 print('0 count in n_table_local for ', indexN, k, N)
358                 n_table_local[k] = 1
359             end
360         end
361
362         -- Building D (needed to compute prediction rows)
363         -- Computing local D
364
365         if D == nil then
366             local Y = n_table_local[1]/(n_table_local[1] + 2*
                        n_table_local[2])
367             D = {}
368             for k=1,K do
369                 D[k] = k - (1 + k)*Y*n_table_local[1 + k]/n_table_local[
                            k]
370             end
371         end
372
373         F_local = 0
374         -- Compute curent order level with modified absolute discouting
```

```
            for each word
375         for j=1,50 do
376             indexN = entry[{1, j}]
377             -- case word seen
378             if F_c_w_table[N][indexes][indexN] ~= nil then
379                 -- Building the key for the different case of absolute
                        discounting
380                 if F_c_w_table[N][indexes][indexN] > K then
381                     key_N_c = K
382                 else
383                     key_N_c = F_c_w_table[N][indexes][indexN]
384                 end
385                 prediction[j] = F_c_w_table[N][indexes][indexN] - D[
                        key_N_c]
386                 F_local = F_local + F_c_w_table[N][indexes][indexN]
387                 N_c_local[key_N_c] = N_c_local[key_N_c] + 1
388             end
389         end
390
391         -- Check that MLE predicted at least one words, otherwise go to
                next context
392         if prediction:sum(1)[1] == 0 then
393             --print('unseen words')
394             return compute_mkn_line(N-1, entry, F_c_w_table, n_table,
                        alpha, K, D)
395         end
396
397         -- Computing factor of lower order model (no denominator
                because we normalize afterwards)
398         local gamma = 0
399         for k=1,K do
400             if N_c_local[k] ~= nil then
401                 gamma = gamma + D[k]*N_c_local[k]
402             end
403         end
404         if gamma < 0 then
405             --print('gamma error')
406             return compute_mkn_line(N-1, entry, F_c_w_table, n_table,
                        alpha, K, D)
407         end
408         -- Combining with next context
409         prediction:add(compute_mkn_line(N-1, entry, F_c_w_table,
                n_table, alpha, K, D):mul(gamma)):div(F_local)
410         -- Normalization
411         -- TODO: why??? We normalize at the end
```

```
412             -- prediction:div(prediction:sum(1)[1])
413             return prediction
414         end
415     end
416
417     -- Modified Kneser Ney: computation done at once line by line
418     --
419     -- p_wb(w|c) = (F_c_w + N_c_. * p_wb(w|c'))/(N_c_. + F_c_.)
420     function distribution_proba_mKN(N, data, alpha, K, D)
421         local N_data = data:size(1)
422         local M = data:size(2)
423
424         -- Building the count matrix for each ngram size lower than N.
425         local F_c_w_table = {}
426         local n_table = {}
427         for i=1,N do
428             train = get_train(i)
429             F_c_w_table[i], n_table[i] = build_context_count_split2(train,
                    K)
430         end
431
432         -- Vector initialisation
433         local distribution = torch.zeros(N_data, 50)
434         for i=1,N_data do
435             -- Compute witten bell for the whole line i
436             distribution:narrow(1, i, 1):copy(compute_mkn_line2(N, data:
                    narrow(1,i,1), F_c_w_table, n_table, alpha, K, D))
437         end
438         --distribution:cdiv(distribution:sum(2):expand(distribution:size(1)
                , distribution:size(2)))
439         return distribution
440     end
```

NNLM:

```
1   function build_model(dwin, nwords, hid1, hid2)
2       -- Model with skip layer from Bengio, standards parameters
3       -- should be:
4       -- dwin = 5
5       -- hid1 = 30
6       -- hid2 = 100
7
8       -- To store the whole model
9       dnnlm = nn.Sequential()
10
11      -- Layer to embedd (and put the words along the window into one
            vector)
```

```
12      LT = nn.Sequential()
13      LT_ = nn.LookupTable(nwords,hid1)
14      LT:add(LT_)
15      LT:add(nn.View(-1, hid1*dwin))
16
17      dnnlm:add(LT)
18
19      concat = nn.ConcatTable()
20
21      lin_tanh = nn.Sequential()
22      lin_tanh:add(nn.Linear(hid1*dwin,hid2))
23      lin_tanh:add(nn.Tanh())
24
25      id = nn.Identity()
26
27      concat:add(lin_tanh)
28      concat:add(id)
29
30      dnnlm:add(concat)
31      dnnlm:add(nn.JoinTable(2))
32      dnnlm:add(nn.Linear(hid1*dwin + hid2, nwords))
33      dnnlm:add(nn.LogSoftMax())
34
35      -- Loss
36      criterion = nn.ClassNLLCriterion()
37
38      return dnnlm, criterion
39  end
40
41
42  function train_model(train_input, dnnlm, criterion, dwin, nwords, eta,
        nEpochs, batchSize)
43      -- Train the model with a mini batch SGD
44      -- standard parameters are
45      -- nEpochs = 1
46      -- batchSize = 32
47      -- eta = 0.01
48
49      -- To store the loss
50      av_L = 0
51
52      -- Memory allocation
53      inputs_batch = torch.DoubleTensor(batchSize,dwin)
54      targets_batch = torch.DoubleTensor(batchSize)
55      outputs = torch.DoubleTensor(batchSize, nwords)
```

```
56      df_do = torch.DoubleTensor(batchSize, nwords)
57
58      for i = 1, nEpochs do
59          -- timing the epoch
60          timer = torch.Timer()
61          av_L = 0
62
63          -- max renorm of the lookup table
64          dnnlm:get(1):get(1).weight:renorm(2,1,1)
65
66          -- mini batch loop
67          for t = 1, train_input:size(1), batchSize do
68              -- Mini batch data
69              current_batch_size = math.min(batchSize,train_input:size(1)
                    -t)
70              inputs_batch:narrow(1,1,current_batch_size):copy(
                    train_input:narrow(1,t,current_batch_size))
71              targets_batch:narrow(1,1,current_batch_size):copy(
                    train_output:narrow(1,t,current_batch_size))
72
73              -- reset gradients
74              dnnlm:zeroGradParameters()
75              --gradParameters:zero()
76
77              -- Forward pass (selection of inputs_batch in case the
                    batch is not full, ie last batch)
78              outputs:narrow(1,1,current_batch_size):copy(dnnlm:forward(
                    inputs_batch:narrow(1,1,current_batch_size)))
79
80              -- Average loss computation
81              f = criterion:forward(outputs:narrow(1,1,current_batch_size
                    ), targets_batch:narrow(1,1,current_batch_size))
82              av_L = av_L +f
83
84              -- Backward pass
85              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)))
86              dnnlm:backward(inputs_batch:narrow(1,1,current_batch_size),
                     df_do:narrow(1,1,current_batch_size))
87              dnnlm:updateParameters(eta)
88
89          end
90
91          print('Epoch '..i..': '..timer:time().real)
```

```
92            print('Average Loss: '..av_L/math.floor(train_input:size(1)/
                  batchSize))
93
94        end
95
96        return dnnlm
97
98  end
```