Università degli Studi di Firenze

Machine Learning - Apprendimento Automatico

Recurrent Neural Network Language Modeling

Authors:
Marco Gualtieri
Maurizio Gullo

Supervisor:
Prof. Paolo Frasconi

Academic year 2014/2015

Contents

Introduction

The goal of language modeling is to learn the joint probability function of sequences of words in a language, in order to predict the next word in textual data given context.
In other words, a **language model** (LM) is a probability distribution over strings that reflects how frequently a string occurs as a sentence.
Language modeling is useful in many fields of natural language processing:

- *speech recognition* and *handwriting recognition*: language model can help to resolve ambiguities (between words that sound similar or are written in a similar way) by providing a context.
- *machine translation*
- *spell correction*
- *intelligent input method*
- *information retrieval*: in query-likelihood models, documents in a collection are ranked following the probability of the query in the document's LM.

A major problem in building a language model is represented by *data sparsity*: just a very small subset of possible sentences will be observed during training. Several kinds of strategies have been adopted to face this problem; in this work we mainly focus on the (recurrent) neural network approach after a general analysis of the previous state-of-art solutions.

This work is structured as follows:

Chapter 1 gives an overview of probabilistic solutions adopted for language modeling, highlighting main features and weaknesses.

Chapter 2 analyses the state-of-art techniques: in particular, we focus on Mikolov's works about recurrent neural networks.

Chapter 3 contains some details about our recurrent neural network framework (implemented from scratch with *Python* and *NumPy*).

Chapter 4 presents some results we obtained with our artefact, comparing them against Mikolov's algorithm.

Chapter 1

Probabilistic Language Modeling

The goal for a LM is to compute the probability of a sentence (sequence of n words):

$$P(W) = P(w_1, w_2, \ldots, w_n)$$

or, similarly, to predict the probability of an upcoming word:

$$P(w_n | w_1, w_2, \ldots, w_{n-1})$$

As a first intuition, one can think to use the *chain rule* for computation:

$$P(w_1, w_2, \ldots, w_n) = P(w_1)P(w_2|w_1) \ldots P(w_n|w_1, \ldots, w_{n-1}) = \prod_i P(w_i|w_1, \ldots, w_{i-1})$$

Note that it is almost impossible to estimate these probabilities using frequency counts:

$$P(w_i|w_1, w_2, \ldots, w_{i-1}) = \frac{C(w_1 w_2 \ldots w_{i-1} w_i)}{C(w_1 w_2 \ldots w_{i-1})}$$

because there are too many possible sentences, and it is unlikely that training set contains enough data.

To avoid this problem of *data sparsity*, we can simplify by assuming the validity of k$^{\text{th}}$-order Markov property:

$$P(x_1, x_2, \ldots, x_n) = \prod_i P(x_i|x_{i-k} \ldots x_{i-1})$$

this means that we can approximate each probability component as:

$$P(w_i|w_1, w_2, \ldots, w_{i-1}) \approx P(w_i|w_{i-k} \ldots w_{i-1})$$

practically, we are considering that the probability of a word in the sentence depends on the previous k words only.

Based on the value of k, we can have **unigram**:

$$P(w_1, w_2, \ldots, w_n) \approx \prod_i P(w_i)$$

bigram:

$$P(w_i|w_1, w_2, \ldots, w_{i-1}) \approx P(w_i|w_{i-1})$$

or, in general, **N-gram** models.
Cutting the context length to N is a strong limitation, since real language has long-distance dependencies, but this often works well enough.

1.1 Problem with generalization

One way to derive probabilities for N-gram is using the Maximum Likelihood Estimate (frequency counts):

$$P(w_i|w_{i-N}, \ldots, w_{i-1}) = \frac{C(w_{i-N}, \ldots, w_{i-1}, w_i)}{C(w_{i-N}, \ldots, w_{i-1})}$$

but, as already mentioned, this approach presents some issues when confronted with any words or N-gram not seen during training.

For instance, if a single word in test set doesn't occur in the training set, we will give probability equal to 0 to the whole test set.
Several smoothing techniques can help here: in a nutshell, the idea is to assign some of the total probability mass to unseen words or N-grams.

Other strategies to achieve generalization are:

- **Back-off** models: the idea is to consider, under certain conditions, models with smaller histories: this means considering the most reliable model depending on the given informations. So if a N-gram has been observed "enough" during training, we use maximum likelihood for conditional probability; otherwise we recursively consider the back-off conditional probability of the (N-1)-gram.

- **Interpolation**: the intention is to combine several models (unigram, bigram, trigram) to obtain better results.

1.2 Evaluation and perplexity

Once we trained a LM over a training set, we need an evaluation metric to check how well the model does on a test set not seen before.
There are two strategies:

- *extrinsic* evaluation of two different models (by comparing accuracy of results obtained with them for a specific task)

- *intrinsic* evaluation on a single model (for instance, by using *perplexity*)

The first approach is usually time consuming, and rely on the ability to easily evaluate accuracy.
The second approach, instead, is usually affected by bad approximation (since test set can be really different from the training one), but can be useful to have a benchmark.

We can say that the best language model is the one that best predicts an unseen test set, that is giving to it the highest probability.
Perplexity is defined as the inverse probability of the test set, normalized by the number of words; so minimizing the perplexity corresponds to maximizing the probability.
It is more convenient to work in **log** space, since this is help to avoid underflow (some probabilities can have really low values):

$$PPL = P(w_1, w_2, \dots, w_n)^{-\frac{1}{n}} = 2^{-\frac{1}{n}\sum_{i=1}^{n}\log_2 p(w_i)}$$

Chapter 2

Neural Language Modeling

The N-gram technique tries to obtain generalization by concatenating very short overlapping sequences seen in the training set.
This has two major limitations, even when considering back-off models:

- the context is usually no longer than 2/3 words
- the syntactic and semantic similarity between words is not taken into account

2.1 Bengio's Neural Model

The work presented in 2003 by Bengio ([1], *A Neural Probabilistic Language Model*) aims to solve the *curse of dimensionality* with the following ideas:

1 Associate with each word in the vocabulary a distributed *word feature vector*, that is a real-valued vector in $\mathbb{R}^m$ (with m much smaller than vocabulary size). This could be initialized using prior knowledge of semantic features.

2 Express the joint probability of words in a sequence in terms of their feature vector representation.

3 Learn simultaneously the *word feature vectors* and the parameters of the joint probability function.

Similar feature vectors are expected for similar words, and since the probability function is a smooth function over them, a small change in the feature will result

in a small change on probability.
Hence each sentence in the training set will increase probability not only of the sentence itself, but also of several other sentences that have a near representation.

Given a sequence $w_1...w_n$ of words $w_i \in V$, the objective is to learn a good model that gives out-of-sample likelihood:

$$f(w_{i-C}, \dots, w_{i-1}, w_i) \approx P(w_i | w_{i-C}, \dots, w_{i-1})$$

(thereafter C is the size of context we are considering).

The function f can be decomposes as:

$$f(w_{i-C}, \dots, w_{i-1}, w_i) = g(M(w_{i-C}), \dots, M(w_{i-1}), M(w_i))$$

where:

- M (shared across all words in context) is a mapping from V to R^m (distributed features vector); it consists of a $|V| \times m$ matrix where the row $M(i)$ is the feature vector for word i.
- g is the probability function over words (expressed using M), this function may be implemented with a feed-forward neural network with a set ω of parameters.

The overall parameter set is then $\theta = (M, \omega)$, and training means to find θ that minimizes the log-likelihood:

$$L = \frac{1}{n} \sum_i log_2 f(w_{i-C}, \dots, w_{i-1}, w_i \ ; \ \theta)$$

It is important to consider that in this model the number of free parameters only scales linearly both with V (vocabulary size) and C (context size).

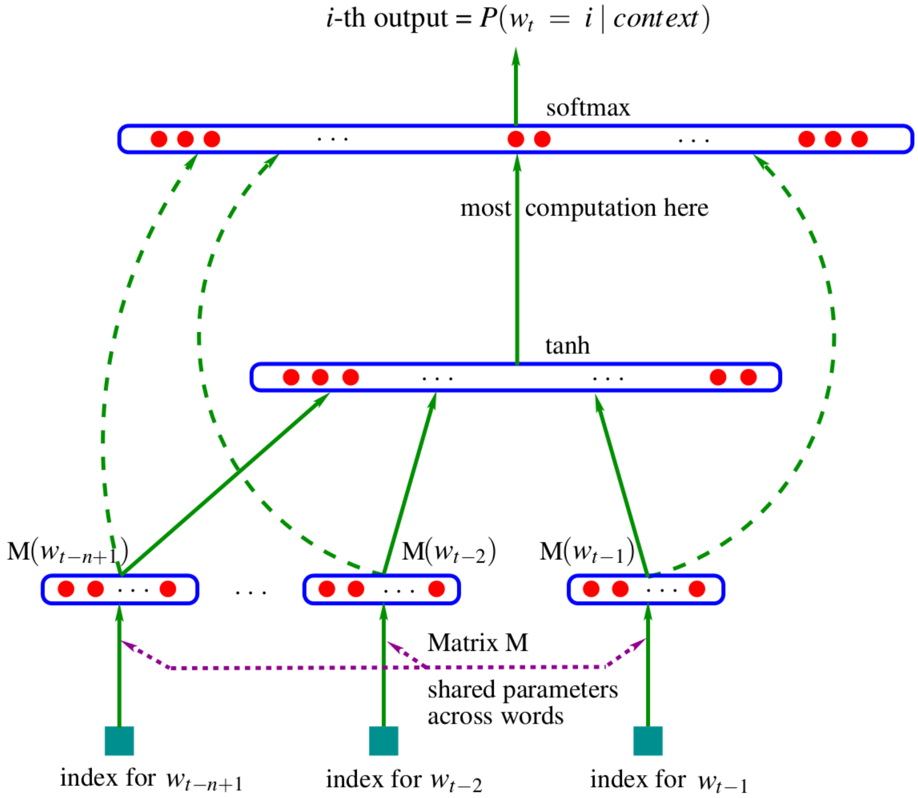


Figure 2.1: Structure of feed-forward Bengio's neural network

2.2 Recurrent Neural Network language model

A major deficiency in the feed-forward structure proposed by Bengio is to be limited to a fixed length context (usually 5-10 words), that has to be set ad-hoc before training.

In his work *Recurrent neural network based language model* ([2]), Mikolov proposed a *simple recurrent neural network* structure to overcome this:

- History length is unlimited (the recurrent connections allow the information to cycle in the network for arbitrarily long time).
- RNN can compress the whole history in a low dimensional space, while feed-forward projects just single words.
- RNN can form short-term memory (they can act like cache model techniques).

The network is composed by:

- input layer x which receives words in 1-of-V coding;
- hidden (or *context*) layer s, which has a sigmoid activation function in its nodes;
- output layer y, with the same size of input layer, whose nodes are activated by a softmax function.

The input of the network at time t is the vector representing the current word, concatenated with values of context layer at time $t-1$.

The RNN is trained with a standard backpropagation method with gradient descent; convergence is usually reached in 10-20 epochs.
Experiments show that regularization of weights is not needed, since the network does not overtrain even with huge datasets.

The rule in statistical language modeling is to update the model only during training phase, not in test step.
Mikolov, instead, affirms that the long-term memory is not in the context layer's neurons (as these change rapidly), but rather in synapses: therefore the training should continue even during testing (with a fixed learning rate). These models are called *dynamics.*

Experiments show that RNNs outperform state-of-art backoff models (in terms of perplexity achieved), even when the latter are trained on bigger dataset.

2.3 Strategies for training large scale NN

The main difficulty for NN and RNN approach is related to computational performance, most of all when processing huge datasets.

The training complexity of a N-gram NN LM is proportional to:

$$I \times W \times ((N-1) \times D \times H + H \times V)$$

where:

I: is the number of training epochs before convergence

W: is the number of tokens in the training set

N: is the N-gram order

D: is the size of each word feature vector

H: is the size of the hidden layer

V: is the vocabulary size

The RNN LM has instead the following computational complexity:

$$I \times W \times (H \times H + H \times V)$$

It can be seen that for increasing order N, the complexity of the feedforward architecture increases linearly, while it remains constant for the recurrent one (N has no meaning in RNN LM).

Several techniques can be adopted in order to improve the training performance.

Reduction of number of training tokens

According to statistics, *out-of-domain* data usually represents 90% of the size of the training corpora, but their weight in the final model is relatively low.
Therefore NN LM can be trained on *in-domain* data only, or considering just some randomly subsampled part of *out-of-domain* data chosen at the start of each training epoch.

This subsampling can lead to major results degradation, against a model that is trained on all the data available.

Automatic data selection and sorting

Usually, stochastic gradient descent is used for training neural networks and this assumes randomized order of training data before start of each epoch.
In RRN we hope that the model will be able to find complex patterns in the data, that are based on simpler patterns. Intuitively, these simpler patterns have to be learned before complex patterns can be learned: this approach is called *incremental learning.*
Starting from this, we let the *out-of-domain* data to be processed first, while the most important *in-domain* data are observed at the end.
This is also useful because data processed at the end of the training will have higher weights, as the update of parameters is done on-line.
Practically, the training set is divided into chunks, then they are sorted by their perplexity (computed with a bigram model). Chunks containing noisy data (i.e. with perplexity over a certain threshold) could be discarded.

We can observe that the most of the computational complexity (both for NN LM and RNN LM) is due to the huge term $H \times V$. (usual values are between 100 and 500 neurons for H and between 50k and 300k words for V).
The following three techniques try to reduce the impact of this term.

Reduction of the hidden layer

Hidden layer size heavily affects computational performance, however it can't be reduced too much if we still want to obtain good results (for instance, $H = 100$ is insufficient to well perform on a large dataset of 600M words).

Reduction of vocabulary size

A successful approach is based on Goodman's trick ([7]) for speeding up maximum entropy models. Each word from the vocabulary is assigned to a class, and only the probability distribution over classes is computed. In the second step we then compute the probability distribution over words that are members of a particular class.

Briefly, if we assume that each word belongs to exactly one class, we can first estimate the probability distribution over the classes using RNN, and then compute the probability of a particular word from the desired class by assuming unigram distribution of words within the class:

$$P(w_i|history) = P(c_i|history)P(w_i|c_i)$$

This reduces the computational complexity to:

$$I \times W \times (H \times H + H \times C)$$

where C is the number of classes. Obviously C can be order of magnitude smaller than V without sacrificing much of accuracy.
On the other hand, performance heavily depends on the ability to estimate classes precisely.

Using a Compression Layer

As proposed in [4], we can think to reduce complexity by inserting a compression layer between hidden and output layers (this will have a sigmoid activation).
This reduces the total amount of parameters: in fact, given P the size of this layer, the number of weights between hidden and output becomes:

$$H \times V \rightarrow H \times P + P \times V$$

2.4 Deep learning via Hessian-free optimization

Backpropagation algorithm with a gradient-descent scheme works efficiently to learn the weights of a network with multiple layers of non-linear hidden units. Unfortunately, this technique does not generalize well with networks that have many hidden layers (deep networks): the common experience is that gradient-descent progresses extremely slowly on deep nets, seeming to halt altogether before making significant progress, resulting in poor performance on the training set (under-fitting).

Also, the gradient descent is unsuitable for objectives that exhibit pathological curvature: here a 2$^{\mathrm{nd}}$-order optimization method can help.
Not accounting for the curvature when computing search directions can lead to many undesirable scenarios. First, the sequence of search directions might move too far in directions of high curvature, causing an unstable "bouncing", often observed with gradient descent. Second, directions of low curvature will be explored much more slowly (a problem exacerbate by lowering the learning rate). Lastly, if the only directions of significant decrease in f are ones with low curvature, the optimization may become too slow and creating a false impression of a local minimum.

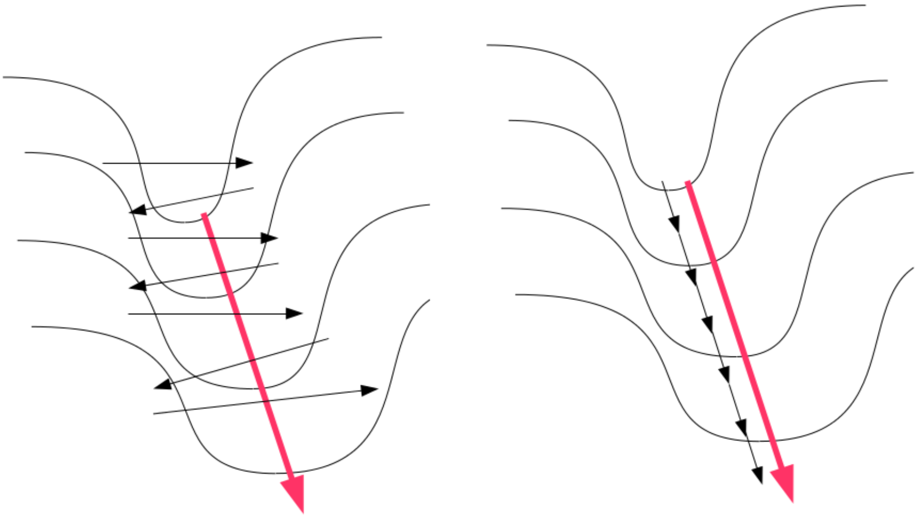


Figure 2.2: Optimization in a long narrow valley

Fig. 2.2 shows a "pathological curvature scenario", where objective function locally resembles a long narrow valley.
The smaller arrows represent the steps taken by the gradient descent with different learning rates, while the large arrow along the base of the valley represents the step computed, for example, by Newton's method. What makes this scenario "pathological" is not the presence of merely low or high curvature directions, but the mixture of both.

2.4.1 Newton's method

Newton's method, like the gradient descent, is an optimization algorithm that iteratively updates the parameters $\theta \in \mathbb{R}^N$ of an objective function f by computing search directions p and updating θ as $\theta + \alpha p$.
The main idea of the Newton's method is that f can be approximated around each θ by the quadratic:

$$f(\theta + p) \approx q_\theta(p) \equiv f(\theta) + \nabla f(\theta)^\intercal p + \frac{1}{2} p^\intercal B p$$

where $B = H(\theta)$ is the Hessian matrix of f at θ.

Finding a good search direction is then equivalent to minimizing this quadratic with respect to p.
This is reduced to solve the system $Bp = -\nabla f(\theta)$, that involves the computation of the $N \times N$ matrix B. This quadratic relationship between the size of the Hessian and the number of parameters in the model makes this strategy unfeasible.

2.4.2 Hessian-free optimization

Hessian-free methods aim to optimize $q_\theta(p)$ by exploiting two simple ideas.
The first is that, for an N-dimensional vector d, Hd can be easily computed using finite differences at the cost of a single extra gradient evaluation:

$$Hd = \lim_{\epsilon \to 0} \frac{\nabla f(\theta + \epsilon d) - \nabla f(\theta)}{\epsilon}$$

The second includes the usage of the linear conjugate gradient algorithm (CG), which is effective for optimizing quadratic objectives (such as $q_\theta(p)$) requiring only matrix-vector products with B.

Algorithm 1 Hessian-free optimization method

```
for $n = 1, 2, \ldots$ do
    $g_n \leftarrow \nabla f(\theta_n)$
    compute/adjust $\lambda$ by some method
    define function $B_n(d) = H(\theta_n)d + \lambda d$
    $p_n \leftarrow$ CG-minimize$(B_n, -g_n)$
    $\theta_{n+1} \leftarrow \theta_n + p_n$
end for
```

Hessian-free is appealing because, unlike many other quasi-Newton methods, it does not make any approximation to the Hessian, but it relies on the accurate computation of the Hd product by the finite differences method.
Moreover, Hessian-free differs from Newton's method only because it is performing an incomplete optimization (via un-converged CG) of $q_\theta(p)$ instead of doing a full matrix inversion.

Chapter 3

Implementation notes

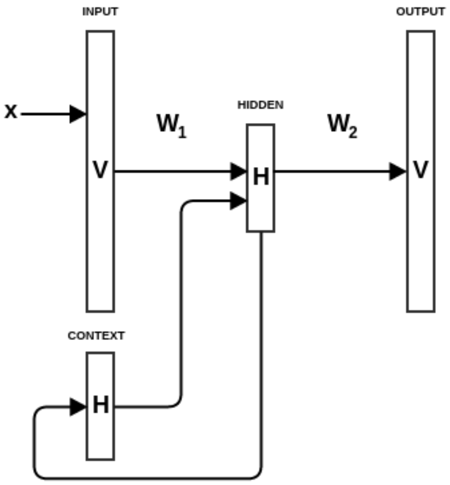


Figure 3.1: Structure of the RNN

As shown in fig. 3.1, the structure of the network is similar to a standard feedforward architecture; with the insertion of an extra context-layer appended to the input (it contains the values of the hidden layer at the previous iteration, i.e. the hidden layer is delayed for one step).

Nodes in hidden layer have a standard sigmoid function:

$$f(x) = \frac{1}{1 + e^{-x}}$$

3.1 Softmax and error gradient

The nodes in the output layer are activated with a function called **softmax**; this forces all the values in output nodes to lie between 0 and 1, and to sum to one:

$$g(z_j) = \frac{e^{z_j}}{\sum_k e^{z_k}}$$

this allows to model a joint distribution over the output variables.

The derivative of softmax has the form:

$$\frac{\partial g_j}{\partial z_k} = g_j \delta_{kj} - g_j g_k$$

where δ_{kj} is the Kronecker Delta.

Let's consider now p and q, discrete probability distributions over the same set of events, with q intended as a model to be compared against the reference distribution p.

Their **cross-entropy** is defined as:

$$H(p,q) = -\sum_x p(x) \, \log_2 q(x)$$

and it represents a measure for similarity between the two distributions.

Therefore, when $p = q$, the cross-entropy takes its minimal value, equal to the entropy of the fixed distribution p: $H(p) = -\sum_x p(x) log_2 p(x)$.

In our scenario, we have a set of N data samples drawn from V different classes, so we can consider the following cross-entropy cost function:

$$E = -\sum_{n=1}^{N} \sum_{k=1}^{V} t_k^n \log_2 g_k(z^n)$$

where t_k^n is the value of the *k-th* component of the target for the *n-th* sample, and g is the softmax described above.

Since we are using a 1-of-V coding scheme, t^n will have all components equal to zero, except for the component representing its corresponding class, which is one.

Given that t is constant, then minimizing the cross-entropy cost is equivalent to minimize this functional:

$$\widetilde{E} = -\sum_{n=1}^{N}\sum_{k=1}^{V} t_k^n \log_2 g_k(z^n) + \sum_{n=1}^{N}\sum_{k=1}^{V} t_k^n \log_2 t_k^n = -\sum_{n=1}^{N}\sum_{k=1}^{V} t_k^n \log_2 \frac{g_k(z^n)}{t_k^n}$$

which has the advantage that the Jacobian takes the convenient form

$$\frac{\partial E}{\partial z_j} = g_j - t_j \quad \forall j = 1, ..., V$$

3.2 Stopping criterion and adaptive learning rate

At the end of each epoch, we evaluate the trend of the learning by calculating the measure of *perplexity* on a validation set:

$$PPL = 2^{-LOGP/N}$$

where

$$LOGP = \sum_{i=1}^{N} \log_2 p(x_i)$$

($p(x_i)$ is the probability associated with the i-th sample of the validation set, as observed in the output layer of the network).

Since probability of each word is in the $[0, 1]$ interval, $LOGP$ has a negative value (this should be closer to zero as possible in order to have a low perplexity). Therefore, if $LOGP$ has decreased compared to the previous iteration, we restore the weights of the network to the values at the previous step.
Also, whether this value is decreased or just reported a not significant improvement (we set a 3‰ threshold), we halve the value of the learning rate and keep on halving it in the following epochs. If the "halving condition" is encountered again, we stop the execution of the algorithm.

Chapter 4

Results

This chapter contains some results obtained with different configurations of the network. In order to have a meaningful baseline, they are compared against results obtained with the C++ toolkit released by T. Mikolov ([5], `http://rnnlm.org/`).

We will use the following labels to denote each neural network instance:

- **mklv_n**: the Mikolov's toolkit with n hidden nodes;
- **nmprnn_n**: our implementation of the recurrent neural network, with n hidden nodes;
- **nmprnn_n_rc**: same as above, in which we reset the context layer before processing each new sentence (i.e. a new line).

For each test, we collected these properties:

- **iter**: the number of training epochs to converge;
- **iter_t**: the average time needed to perform a single training epoch;
- **v_ppl**: the perplexity computed on validation set on the last epoch;
- **t_ppl**: the perplexity evaluated on test set;
- **t_wer**: the *Work Error Rate* on test set.

Note 1: the execution times are reported just to give a rough estimate, without claiming to be complete or accurate.
Tests are performed on a MAC OSX machine with 16GB of memory and a processor `Intel Core i7 2.8GHz`, in which `Python` and `NumPy/SciPy` are installed using `Anaconda` distribution, with the `MKL Optimizations` add-on to improve computational performances.

Note 2: to be sure that the C++ toolkit and our system work on the same data, we feed them with plain text files in which punctuation is removed and all the content is lower case.

4.1 Small dataset

This set is included as a sample in the Mikolov's toolkit, it consists of:

- Training $\sim$ 71350 words
- Validation $\sim$ 6100 words
- Test $\sim$ 7500 words
- Vocabulary size $\sim$ 3700

	iter	**iter_t**	**v_ppl**	**t_ppl**	**t_wer**
mklv_15	12	15s	74.6	75.5	73.3 %
nmprnn_15	12	21s	74.0	73.8	76.5 %
nmprnn_15_rc	12	24s	73.9	73.8	76.3 %
mklv_50	12	40s	70.7	70.8	73.0 %
nmprnn_50	12	62s	69.9	70.7	75.8 %
nmprnn_50_rc	11	75s	71.0	71.0	76.3 %
mklv_100	11	98s	70.9	70.2	72.9 %
nmprnn_100	11	150s	70.4	69.5	75.5 %
nmprnn_100_rc	11	163s	71.9	70.5	76.0 %

Table 4.1: Comparison on small dataset

4.2 Medium dataset

This corpus is built from English version of Wikipedia, gathering articles from different categories (Technology, Science, Cinema and Art, Sport...), for a total of:

- Training $\sim$ 416000 words
- Validation $\sim$ 105000 words
- Test $\sim$ 95000 words
- Vocabulary size $\sim$ 25400

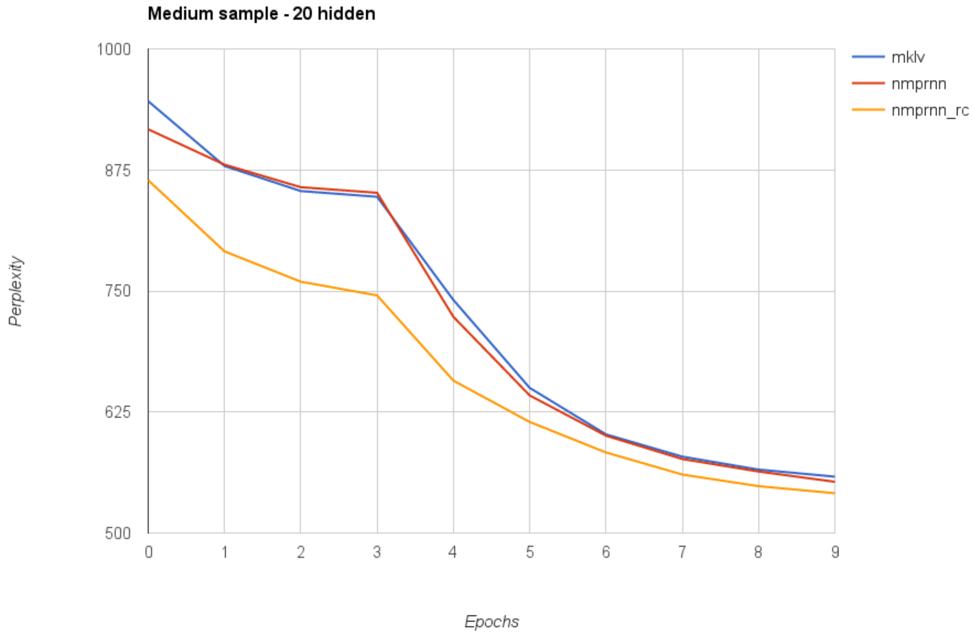
	iter	**iter_t**	**v_ppl**	**t_ppl**	**t_wer**
mklv_20	10	13m	558.2	602.9	82.0 %
nmprnn_20	10	18m	552.9	603.7	82.7 %
nmprnn_20_rc	10	18m	541.1	594.4	82.1 %
mklv_100	9	118m	579.9	642.5	82.0 %
nmprnn_100	10	126m	546.4	607.6	82.3 %
nmprnn_100_rc	9	130m	546.5	602.0	82.4 %
mklv_200	9	210m	629.5	690.3	82.2 %
nmprnn_200	10	250m	590.2	662.6	82.6 %
nmprnn_200_rc	9	255m	569.2	640.4	82.3 %

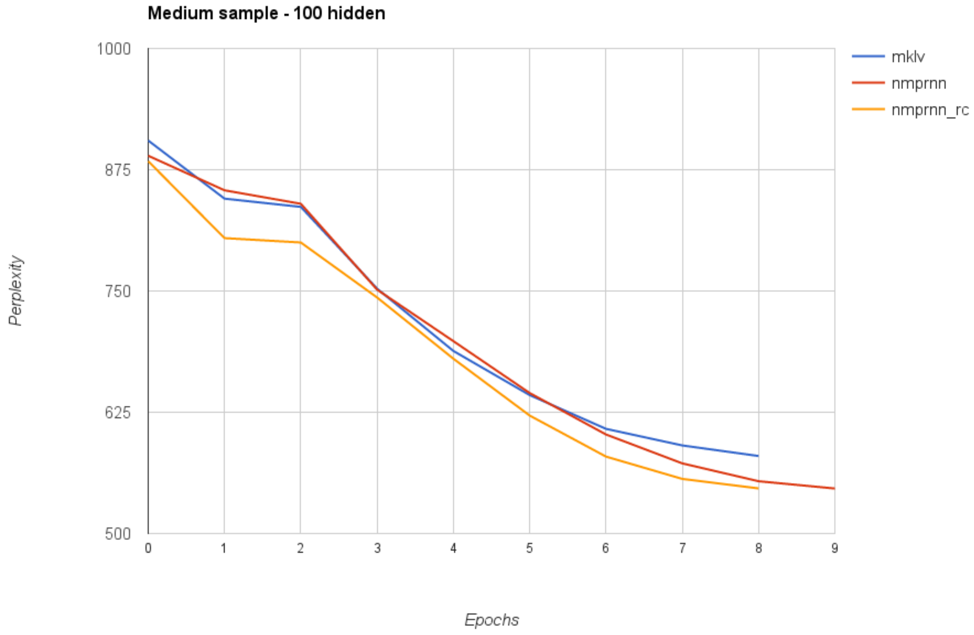
Table 4.2: Comparison on medium dataset

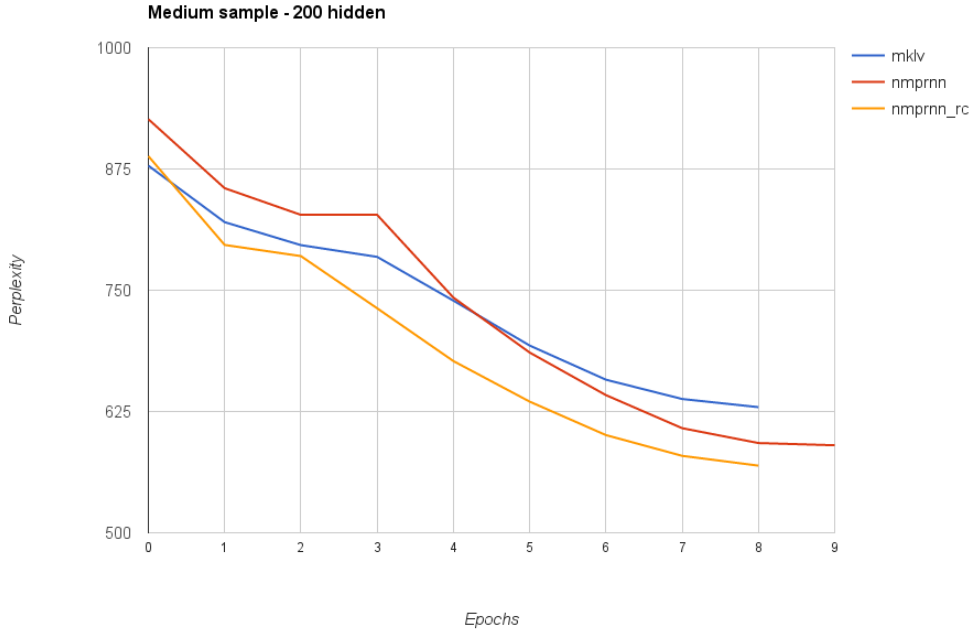
In general, resetting the context layer at each new sentence seems to produce better results.
About the differences between our model and Mikolov's one, they has to be searched mainly in the activation functions implementation: both for sigmoid and softmax, Mikolov uses a `fastexp` function, that has a lower precision for the benefit of computational times.

Another interesting aspect, even if our datasets are too short to give a reliable indication, is that over a certain value (related to vocabulary size and number of samples) increasing the number of hidden nodes does not result in improvements of perplexity.

Following graphics show the trend of validation perplexity at the end of each training epoch.







4.2.1 Experiments with learning-rate smoothing

Starting from the configuration (**nmprnn_100_rc**), we tried to use a different learning rate for each word in the training set, depending on its frequency:

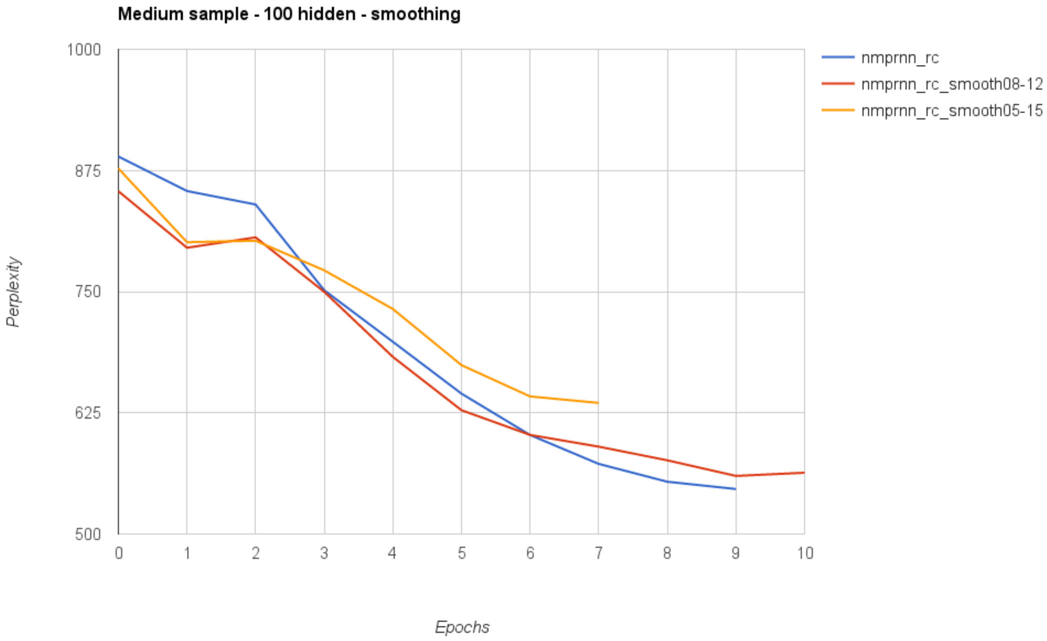
$$\alpha_i = log_2(W) - log_2(count(w_i))$$

normalized between $(0.8, 1.2)$ and $(0.5, 1.5)$ ranges.

	iter	**iter_t**	**v_ppl**	**t_ppl**	**t_wer**
nmprnn_100_rc	9	130m	546.5	602.0	82.4 %
nmprnn_100_rc_0.8,1.2	11	130m	559.9	621.0	82.3 %
nmprnn_100_rc_0.5,1.5	8	130m	635.2	670.0	83.9 %

Table 4.3: Comparison on smoothing

This strategy doesn't seem to help in producing a better generalization.


Medium sample - 100 hidden - smoothing
1000
875
750
625
500
Perplexity
0
1
2
3
4
5
6
7
8
9
10
Epochs
nmprnn_rc
nmprnn_rc_smooth08-12
nmprnn_rc_smooth05-15

References

[1] Bengio Y., Ducharme R., Vincent P., Jauvin C.
A Neural Probabilistic Language Model
Université de Montréal, Département d'Informatique et Recherche Opèrationnelle
Journal of Machine Learning Research, 2003.

[2] Mikolov T., Karafìat M., Burget L., Cernocky J., Khudanpur S.
Recurrent neural network based language model
Brno University of Technology
Interspeech, 2010.

[3] Mikolov T., Deoras A., Povey D., Burget L., Cernocky J.
Strategies for Training Large Scale Neural Network Language Models
Brno University of Technology
Proceedings of ASRU, 2011.

[4] Mikolov T., Kombrink S., Burget L., Cernocky J., Khundanpur S.
Extensions of Recurrent Neural Network Language Model
Brno University of Technology
ICASSP, 2011.

[5] Mikolov T., Kombrink S., Deoras A., Burget L., Cernocky J.
Recurrent neural network language modeling toolkit
Brno University of Technology
ASRU Demo Session, 2011.

[6] Martens J.
Deep learning via Hessian-free optimization
University of Toronto
ICML, 2010.

[7] J.Goodman
Classes for fast maximum entropy training
Microsoft Research, Redmond
ICASSP, 2001.