IDIAP COMMUNICATION

Internship Report : Summer 2003

Jean-Sébastien Senécal

IDIAP–Com 03-09

September 2003

Dalle Molle Institute for Perceptual Artificial Intelligence • P.O.Box 592 • Martigny • Valais • Switzerland

phone +41 – 27 – 721 77 11
fax +41 – 27 – 721 77 12
e-mail secretariat@idiap.ch
internet http://www.idiap.ch

1 Activity Summary

Here is a brief summary of my activities of this summer.

1. I received corrections to do on my **Master Thesis** from the jury board at University of Montreal. I had to do some major changes to the introduction and to add some explanations in the chapter about algorithms.

2. I **submitted an article to NIPS**, which was finally refused. The article was published as an IDIAP research report [Bengio and Senécal, 2003a].

3. I implemented a **language model library** in Torch, with a n-gram and some connectionist models like the NNLM (neural network language models).

4. I found a way of **quickly estimating the output of a NNLM** with importance sampling (section 3.1).

5. I worked on different approaches to **estimate the gradient of a NNLM with Monte Carlo Markov Chains** (section 3.2).

6. I tried different ways of **guessing the minimal sample size** needed to ensure convergence of the NNLM with importance sampling (section 3.3).

7. I worked on a new acceleration approach based on **clustering** without prior knowledge (section 3.4).

2 Programming

Here is a brief summary of the important classes and programs I programmed. All the classes can be found in directory `/homes/senecal/Torch3/lm`. The corresponding files are named in the usual way i.e. if ClassName is the name of the class, ClassName.h and ClassName.cc are the corresponding files. Programs can be found in `/homes/senecal/Torch3/lm/examples`. **Most** of these programs work fine but just to make sure, it is better to ask me before using them (`senecal@idiap.ch`).

Classes are described in tables 1,2 and 3 and programs in tables 4 and 5.

3 Research

In this section, I describe the different methods I worked on and I give some results. I make the assumption that readers are familiar with the NNLM (neural network language model) architecture as well as with importance sampling methods used to train it. For reference, see [Bengio et al., 2003, Bengio and Senécal, 2003b, Bengio and Senécal, 2003a].

3.1 Quickly Estimate the Ouput of a NNLM with Importance Sampling

Let $\mathcal{V}$ be the vocabulary, $w \in \mathcal{V} = \{1, \ldots, |\mathcal{V}|\}$ the target (next) word and $h \in \mathcal{V}^{n-1}$ the $(n-1)$ words of context, i.e., preceding w. The output of a NNLM has the following form:

$$P(w|h) = \frac{e^{f(w,h)}}{Z(h)} \quad (1)$$

where $f(w,h)$ is the output of the neural net foir word w and $Z(h) = \sum_{w' \in \mathcal{V}} e^{f(w',h)}$ is called the *partition function*.

Class name	Short description
LM	A general class for language models.
EnergyLM	A general class for energy-based language models.
NNLM	The standard neural network language model architecture.
CNNLM	The “cycling” neural network language model architecture.
Ngram	The n-gram language model (as a subclass of LM).
CodeClusterNNLM	A cluster-based NNLM (section 3.4).
LMSampler	A LM that is used as a proposal (sampling) distribution in Monte Carlo approximation.
AdaptiveNgramLMSampler	An n-gram proposal distribution that can be adapted to an energy-based language model in order to get better Monte Carlo estimates (or to pick less samples in order to get a good estimate).

Table 1: Classes : machines and models

Class name	Short description
EnergyLMStochasticGradient	A stochastic gradient algorithm to train energy-based language models.
AISEnergyLMStochasticGradient	A stochastic gradient algorithm to train energy-based language models with adaptive importance sampling.
MCMCEnergyLMStochasticGradient	A stochastic gradient algorithm to train energy-based language models with Monte Carlo Markov Chains (section 3.2).
PerfectAISEnergyLMStochasticGradient	A stochastic gradient algorithm to train energy-based language models with adaptive importance sampling that tries to find the right number of samples with the method described in section 3.3.2.
PerfectMCMCEnergyLMStochasticGradient	A stochastic gradient algorithm to train energy-based language models with Monte Carlo Markov Chains using the “perfect” algorithm described in section 3.3.1.
CodeClusterStochasticGradient	A stochastic gradient algorithm to train a cluster NNLM, as described in section 3.4.

Table 2: Classes : training algorithms

Class name	Short description
CacheLM	A cache that may be used by language models to store precomputed probabilities in the few last contexts that were seen.
DiskWordDataSet	A dataset used to read text corpus. The text corpus needs to be stored in a simple binary format basicaly just a stream of integers, where each integer is a "word index" whose corresponding word can be retrieved in a vocabulary file).
HashVocabulary	An implementation of the Vocabulary class that uses a hash table to retrieve word indexes. I suggest that the Vocabulary class be replaced by this class one of these days.
Array	A set of macros that implement a dynamical array for any data type. The array automatically resize itself when new elements are added. The interface provides method to append, insert, remove and find elements.
Map	A set of macros that implement an hash table. The interface provides methods to add, get and remove elements.

Table 3: Classes : utility classes

Program name	Short description
nnlm	Train and test a NNLM or CNNLM.
ais_nnlm	Train and test a NNLM or CNNLM with importance sampling.
mcmc_nnlm	Train and test a NNLM or CNNLM with Monte Carlo Markov Chain.
perfect_ais_nnlm	Train and test a NNLM or CNNLM with "perfect" importance sampling.
perfect_mcmc_nnlm	Train and test a NNLM or CNNLM with "perfect" Monte Carlo Markov Chain.
ccnnlm	Train and test a cluster NNLM.

Table 4: Programs : learning algorithms

Program name	Short description
text2bin	Transforms a text file into a binary file that could be read using the DiskWordDataSet class.
wfreq2wfreqbin	Transforms a frequency file (generated using the SRILM or CMU package) into a binary frequency file.

Table 5: Programs : utility programs

The difficulty with computing (1) is that of computing $Z(h)$, since it involves computing $f(w', h)$ for every w' in the vocabulary. Since $\mathcal{V}$ is usually large (several thousands), this is usually much consuming from a computational point of view.

One way to overcome this problem is to approximate (1) by **estimating the partition function**. We will show that we can do that with a sampling approximation, the accuracy of which we can control.

We first show how $Z(h)$ can be seen as an average. Let W be a random variable that takes a value in $\mathcal{V}$ with uniform probability (i.e. $P(W = w) = \frac{1}{|\mathcal{V}|}$). Then

$$Z(h) = |\mathcal{V}| \sum_{w' \in \mathcal{V}} \frac{1}{|\mathcal{V}|} e^{f(w',h)} = |\mathcal{V}| E[e^{f(W,h)}]. \tag{2}$$

Thus, we could estimate $Z(h)$ by simple uniform sampling. That is, let $W_1, \ldots, W_n$ be a random sample from the uniform, then

$$\bar{Z_n}(h) = \frac{|\mathcal{V}|}{n} \sum_{i=1}^{n} e^{f(W_i,h)} \tag{3}$$

is an unbiased estimator of $Z(h)$. However, the problem with this estimator is that it has a big variance when the distribution $P(w|h)$ is very "peaky" (which is the case in a NNLM). The reason is quite obvious. If we sample points uniformly, most of the samples will be in a region of the probability space where the $f(w, h)$ are low. Clearly, we could take advantage of sampling more points in the "importance" region, where the $f(w, h)$ are high: that would reduce the number of samples we have to pick in order to get a good estimate.

Let $Q(w|h)$ be an easy to sample distribution that is close to $P(w|h)$. We call it the *proposal distribution*. Let W a random variable sampled from $Q(W|h)$. See that

$$Z(h) = \sum_{w' \in \mathcal{V}} e^{f(w',h)} = \sum_{w' \in \mathcal{V}} \frac{e^{f(w',h)}}{Q(w'|h)} Q(w'|h) = E\left[\frac{e^{f(W,h)}}{Q(W|h)}\right]. \tag{4}$$

Let $W_1, \ldots, W_n$ be a random sample taken iid from $Q(W|h)$. Then

$$\hat{Z_n}(h) = \frac{1}{n} \sum_{i=1}^{n} \frac{e^{f(W_i,h)}}{Q(W_i|h)} \tag{5}$$

is an unbiased estimator of $Z(h)$, known as *importance sampling*. This estimator has minimum variance when $Q = P$. For finding a good Q, we could use the method described in [Bengio and Senécal, 2003a], that is, adapt a n-gram like distribution to be close to P.

However, we don't know how many points n we should sample to get a good approximation. I propose a simple method to estimate the good sample size. Suppose one wants to estimate the negative log-likelihood over a point of data (w, h). We can see that the negative log-likelihood decomposes into two parts:

$$NLL = \log P(w|h) = f(w, h) - \log Z(h). \tag{6}$$

Now, what interests us is to get an estimate of $\log Z(h)$ so as to approximate NLL. Let $\hat{Z_n}(h)$ the importance sampling estimator (Eq. (5)). An estimate of NLL would be

$$\widehat{NLL} = f(w, h) - \log \hat{Z_n}(h) \tag{7}$$

such that

$$\log Z(h) = \log \hat{Z_n}(h) \pm 1.96\sqrt{Var(\log \hat{Z_n}(h))}$$

for a confidence interval of 95%. What I propose is to increase the number of samples n until $1.96\sqrt{Var(\hat{Z}_n(h))}$ becomes less than some threshold ϵ, so that $\log Z(h) \in [\hat{Z}_n(h) - \epsilon, \hat{Z}_n(h) + \epsilon]$, 95% of the time.

Applying the delta rule, we can get an estimate of $Var(\log \hat{Z}_n(h))$:

$$\left(\frac{1}{\hat{Z}_n(h)}\right)^2 \frac{1}{n} CV\left(\frac{e^{f(W,h)}}{Q(W|h)}\right) \tag{8}$$

where $CV(X) = \frac{1}{n-1}\frac{\sum_{i=1}^{n}(X_i - \bar{x})^2}{\bar{x}^2}$ is the *coefficient of variation* ($\bar{x} = \frac{1}{n}\sum_{i=1}^{n} X_i$) [1].

The resulting algorithm is presented right here (Alg. 1).

Algorithm 1 Estimation of the partition function

function *estimateZ()*
  fpropInput(h)
  $i \leftarrow 1$
  while $i \leq n_0$ **do**
    $w_i \leftarrow$ *proposal(h)* {Pick a point from Q}
    $f \leftarrow$ *fprop*(w_i)
    $o_i \leftarrow e^f$
    $i \leftarrow i + 1$
  $\sigma^2 \leftarrow$ *variance*$((o_j)_{j=1}^{i})$ {Estimate variance}
  while $1.96\sqrt{\sigma^2} > \epsilon$ **do**
    $w_i \leftarrow$ *proposal(h)* {Pick a point from Q}
    $f \leftarrow$ *fprop*(w_i)
    $o_i \leftarrow e^f$
    $\sigma^2 \leftarrow$ *variance*$((o_j)_{j=1}^{i})$
    $i \leftarrow i + 1$
  $n \leftarrow i - 1$
  $Z \leftarrow \frac{1}{n}\sum_{j=1}^{n} o_j$
  return Z

3.2 Adaptive Monte Carlo Markov Chains

There exist other sampling methods than importance sampling to estimate averages. One of these is known as *Monte Carlo Markov Chains*, or MCMC for short. In our framework, an MCMC is a Markov chain with finite states $\mathcal{V}$ (one state per word) and probabilities of transition $T(W_{t+1}|W_t, h)$ such that

$$\sum_{w' \in \mathcal{V}} T(w|w', h) P(w'|h) = P(w|h). \tag{9}$$

Given T has some extra properties, theory says that in whatever starting state the chain is, if we run it for enough steps – at each step selecting the next state W_{t+1} given the previous state W_t according to $T(W_{t+1}|W_t)$ – the sequence of generated states will tend to be distributed according to $P(W|h)$. We say that the chain has *converged* to its stability distribution. From this point, we could thus use the next W_t's sampled from the chain to approximate an average over some random variable of $P(W|h)$. That is, suppose after n_0 steps of running the chain, it has converged to its

[1] Here the X_i are iid samples of $Q(W|h)$.

stability distribution $P(W|h)$. Then if we continue sampling n points $W_{n_0+1}, \dots, W_{n_0+n}$ we could approximate $E_P[\frac{\partial f(W,h)}{\partial \theta}]$ by

$$\hat{I_n}(h) = \frac{1}{n} \sum_{t=n_0+1}^{n_0+n} \frac{\partial f(W_t, h)}{\partial \theta}. \tag{10}$$

Let us specify a probability of transition that has the required properties. Let $Q(W|h)$ be a probability distribution from which it is easy to sample points (the proposal distribution). Then

$$T(W_{t+1}|W_t, h) = Q(W_{t+1}|h) \min\left(1, \frac{e^{f(W_{t+1},h)}Q(W_t|h)}{e^{f(W_t,h)}Q(W_{t+1}|h)}\right)$$

has the required properties. The resulting algorithm, which consists of iterating through the Markov Chain using this transition function, is called the *independent Metropolis-Hastings* algorithm. The algorithm for the transition step is shown below (Alg. 2).

Preliminary experiments with a fixed proposal (e.g. unigram distribution) had yielded poor results: the chain wouldn't converge, even after waiting for thousands of steps. We thought the reason was that we used a fixed proposal distribution (the unigram). It is well known that the independent Metropolis-Hastings algorithm suffers drasticaly from a proposal distribution that is too far from the actual – target – distribution $P(W|h)$. However, we had not tried it with an adaptive proposal, as was succesfully done in [Bengio and Senécal, 2003a] for importance sampling.

As experiments with importance sampling has shown, it is better to use an *adaptive* proposal distribution, i.e., one that is adapted to be as close as possible to the target – NNLM – distribution. In [Bengio and Senécal, 2003a] we explain how to implement and adapt such a proposal distribution, in the importance sampling framework. However, what could be done with importance sampling could be done with a MCMC. All one needs in order to adapt the proposal is a set of words w for which $f(w, h)$ is known. We can have such a set simply by reusing the points that were sampled during Metropolis-Hastings, for which we had to evaluate the activations $f(w, h)$ anyway.

Overall, this approach still yielded poor results compared to importance sampling. The main problem, I think, is that we are never sure that the chain we are sampling from has converged to its stability distribution, so there is bias and it hampers learning.

Algorithm 2 Independent Metropolis-Hastings

function *metropolisHastings(W_t)*
  Sample Y from proposal $Q(Y|h)$
  Sample U from uniform $U(0,1)$

$$W_{t+1} = \begin{cases} Y, & \text{if } U \leq r(W_t, Y) \\ W_t, & \text{else} \end{cases}$$

  where $r(W, Y) = \min\left\{1, \frac{e^{f(Y,h)}Q(W|h)}{e^{f(W,h)}Q(Y|h)}\right\}$.
  return W_{t+1}

3.3 Perfect Sampling Methods

We worked on ways to find the minimal number of samples we have to pick in order to get a good estimator (either with MCMC or importance sampling).

3.3.1 Perfect Sampling with Monte Carlo Markov Chains

We thought the main problem with MCMC's is that we never know wether the chain has converged or not. If we use samples before convergence has occured, the estimate will be biased and learning will be impaired. On the other hand, if we sample too much points in order to ensure convergence, performance is impaired. How can we assess for convergence of the chain?

Several methods have been proposed to deal with the convergence problem. Some of them *guarantee* that the chain has converged and are thus called *perfect sampling methods*. In [Corcoran and Tweedie, 2002], the authors propose a simple method, known as *backward coupling* for perfect sampling in independent Metropolis-Hastings chains. Let $\beta_0(h) = \min_{w\in\mathcal{V}} \frac{Q(w|h)}{e^{f(w,h)}}$. Then the following algorithm ensures that the chain has converged when the point is returned, i.e., that the returned point can be considered as being sampled from $P(W|h)$. This point can thus be used to initialize an MCMC (see Alg. 2); all subsequent points drawn from this MCMC can then also be considered as points sampled from $P(W|h)$.

Algorithm 3 Perfect Independent Metropolis-Hastings

```
function perfectMetropolisHastings()
  loop
    Sample Y from proposal Q(Y|h)
    Sample U from uniform U(0,1)
    if U ≤ β_0(h) e^{f(Y,h)} / Q(Y|h) then
      return Y
```

Now, the problem we faced with this algorithm is that we can't have $\beta_0(h)$ unless we compute $\frac{Q(w|h)}{e^{f(w,h)}}$ for all w in $\mathcal{V}$ (which we don't want to do). We know that as long as we choose $\beta_0(h) \leq \min_{w\in\mathcal{V}} \frac{Q(w|h)}{e^{f(w,h)}}$, the "perfectness property" will be kept. However, the smallest the $\beta_0(h)$, the longest the computation time. So we face the following dilemna: we want $\beta_0(h)$ to be as big as possible, but never bigger than $\min_{w\in\mathcal{V}} \frac{Q(w|h)}{e^{f(w,h)}}$.

We have tried different ways to approximate $\beta_0(h)$.

The first thing we tried is to keep a fixed β_0 for all contexts h and try to make sure $\beta_0 \leq \beta_0(h) \forall h$. To do that, we just start with an initial β_0. As soon as we find a ratio $\frac{Q(w|h)}{e^{f(w,h)}}$ that is smaller than β_0, we just let $\beta_0 = \frac{Q(w|h)}{e^{f(w,h)}}$. There are two problems with this approach. The first problem is that we are still never sure that our β_0 is small enough. In fact, when we find that it is **not** small enough and we replace it, we know we have made a mistake and that our previous estimates of the gradient might be biased (because previous chains might not have converged). The second problem is that the β_0, which is independent of context, might be too conservative in some contexts (too small). In practice, we find that this method not only yields poor results but is usually much time-consuming.

The second thing we tried is to look at the distribution of the $\frac{Q(w|h)}{e^{f(w,h)}}$'s in order to infer at what point the minimum usually is. We've found a correlation between the point w_{max} where $Q(w_{max}|h)$ is maximal and $min_{w\in\mathcal{V}} \frac{Q(w|h)}{e^{f(w,h)}}$. What we've found is not that these points always correspond, but that they often correspond and that, otherwise, $\frac{Q(w_{max}|h)}{e^{f(w_{max},h)}}$ is usualy close to the minimum. Though this looks strange, we may have an explanation for this. The thing is, the points where $Q(w|h)$ is high usualy correspond to points where $P(w|h)$ is high as well. However, we know that $P(w|h)$ is very "peaky", whereas $Q(w|h)$ is smoother. So, while it is true that $Q(w|h)$ is maximal at $w = w_{max}$, it might be that in the majority of the case, $e^{f(w_{max},h)}$ is so high at this point that $\frac{Q(w_{max}|h)}{e^{f(w_{max},h)}}$ is small (or minimal). We tried to use this prior information to estimate the right $\beta_0(h)$. However, it seems in fact to be a poor estimate of the minimal weight, so we abandoned this idea as well.

The main problem with this method is that finding a good estimate of $\beta_0(h)$ requires some knowledge of both $Q(w|h)$ and $P(w|h)$. The distribution of the weights varies greatly depending on the context.

3.3.2 Perfect Biased Importance Sampling

In [Bengio and Senécal, 2003b] and [Bengio and Senécal, 2003a] we describe a way to accelerate training of a NNLM using importance sampling. We use a heuristical metric, called *effective sample size*, to "automatically" find the proper number of samples at each step. The metric estimates the number of samples that is required for the importance sampling estimator to have the same variance as the classical Monte Carlo estimator with n samples [2]. However, that tells us nothing about how many n samples we would need with the classical Monte Carlo estimator. Furthermore, it doesn't acount for the bias in our importance sampling estimator. The problem is that if we choose n too small, the variance of the estimator might still be high and convergence will be slowed down.

With all these considerations in mind, we tried a new way of estimating the right number of samples. The idea is very simple. Let $\tilde{I}_n$ be the biased importance sampling estimator of the gradient [Bengio and Senécal, 2003b]. Recall that this is a vector (since the gradient is a vector). Let $\tilde{I}_{n_0}$ be a first estimate of the gradient. Then, we increase the sample size (that is, we pick say k more sample at each step n) until $\frac{\|\tilde{I}_{n_0+k(n+1)}-\tilde{I}_{n_0+kn}\|}{\|\tilde{I}_{n_0+kn}\|} < \epsilon$, i.e., the importance sampling estimator has "converged". Here $\epsilon > 0$ is a value that controls accuracy and $k \geq 1$ is the number of samples that is picked at a time.

This method yielded poor results. It is harder to compute that effective sample size because we have to compute gradients norms. It also seems to require much more samples. This is because convergence of the estimator is sufficient but unnecessary for convergence of the gradient. Waiting for convergence of the estimator surely makes sure the variance of the estimator is low. However, at start of training we don't require a nonnoisy estimator. We're still able to converge if the estimator is noisy, as long as it is rather unbiased – which is not the case with effective sample size **nor** with the proposed algorithm – so at start of training we're likely to pick way too much samples with this method. I'd say this still requires some investigation, but I'm not confident that it may yield better results.

3.4 Clustering Acceleration Without Semantical Prior Knowledge

One way to accelerate training of energy-based models like NNLM is to do class clustering. Remind that the NNLM's output is a softmax probability distribution (Eq. (1)). Let the outputs $f(w, h)$ have the following form:

$$f(w,h) = \sum_{i=1}^{n_h} \alpha_{wi} h_i(h) + \beta_w \tag{11}$$

where α_{wi} is the i-th output weight for word w, β_w is the output bias of word w and $h_i(h)$ is the i-th hidden unit (with context h as input).

What we propose is to consider the output weights α_{wi} as a *semantical representation* of target word w. This rather strange idea stems from the following remark. In some given context h, the hidden units will all have values in $\{-1, +1\}$ – well it's not exactly true, but given the form of the $\tanh(\cdot)$ function, it will be close to that – so we could say that the hidden units give us a *binary representation* of the current context. But it can also be seen as a binary representation, in a semantical space, of the word that is the most likely to follow the current context. Say, each unit might represent some characteristic of the word: is it a verb or not? is it an animal or not? is it a noun or not? Now the output weights might then be seen as an answer to these questions for some word. Let's say one hidden unit tells wether the next word is a verb (+1) or not (−1). Then the words "eat" and "bring" would have a strong positive weight, whereas words "coffin" and "the" would have a strong negative weight. The words "take" would have a small positive weight, since it can also be a noun.

[2] The classical Monte Carlo estimator with n samples is just the average of n points sampled from the target $P(W|h)$ distribution. That is, let $X_1, \ldots, X_n$ an iid sample of $P(W|h)$. Then the classical Monte Carlo estimator of the gradient $E_P[\frac{\partial f(W,h)}{\partial \theta}]$ is $\hat{I}_n = \frac{1}{n}\sum_{i=1}^{n} \frac{\partial f(W_i,h)}{\partial \theta}$.

Another way to see that is to notice that two words that have close output weights – in an euclidian way – will also have close outputs $f(w,h)$ in the same contexts if we forget about the biases (that don't vary depending on contexts). That is, if words w_1 and w_2 always appear after similar contexts, let $\alpha_{w_1 i} \approx \alpha_{w_2 i}\ \forall i$, then $f(w_1,h) - f(w_2,h) \approx \beta_{w_2} - \beta_{w_1}$ so the difference is just a constant independent of context.

Now the idea is to use this information to accelerate training and activation (propogation) by a very simple clustering technique. The advantage of this technique is that it doesn't require any prior knowledge in order to do the actual clustering: it is done by clustering the parameters (output weights) that are learned during training in a very simple way.

Let us design a new NNLM architecture. Let $\mathcal{V} = \{1,\ldots,|\mathcal{V}|\}$ be the vocabulary and $\mathcal{C} = \{|\mathcal{V}|+1,\ldots,|\mathcal{V}|+|\mathcal{C}|\}$ be a "cluster vocabulary". You can see the elements in $\mathcal{C}$ as "concepts" or "senses", whatever you like. Let $c_w \in \mathcal{C}$ be the cluster attached to word $w \in \mathcal{V}$ and let $\mathcal{W}_c = \{w \in \mathcal{V} | c_w = c\}$. Now, let the probability of the neural net be

$$P(w|h) = P(w|c_w,h)P(c_w|h) \tag{12}$$

where

$$P(w|c_w,h) = \frac{e^{f(w,h)}}{\sum_{w' \in \mathcal{W}_{c_w}} e^{f(w',h)}} \tag{13}$$

and

$$P(c_w|h) = \frac{e^{f(c_w,h)}}{\sum_{c' \in \mathcal{C}} e^{f(c',h)}}. \tag{14}$$

Notice that in (14) the $f(c,h)$ for clusters $c \in \mathcal{C}$ has the same form as (11), i.e., each cluster has an associated output layer plus a bias. Now, the idea is that $P(c_w|h)$ gives kind of a first evaluation of the probability of word w (the probability that next word belongs to the class c_w) and $P(w|c_w,h)$ gives a second, more precise evaluation of its probability compared to other words in the same cluster. The advantage is that we can reduce the time complexity – which is usualy $O(|\mathcal{V}|)$ in a NNLM – to $O(\sqrt{|\mathcal{V}|})$ if we keep balanced clusters – i.e. $|\mathcal{W}_c| \approx |\mathcal{W}_{c'}|$ for all $c,c' \in \mathcal{C}$ – because we don't need to compute the partition function $Z(h)$.

The gradient of the negative log-likelihood of (12) has the very nice decomposition

$$\frac{\partial - \log P(w|h)}{\partial \theta} = -\frac{\partial \log P(w|c_w,h)}{\partial \theta} - \frac{\partial \log P(c_w|h)}{\partial \theta} \tag{15}$$

which allows us to backpropagate separately on $\log P(w|c_w,h)$ and $\log P(c_w|h)$. Again, backpropagation stays in $O(\sqrt{|\mathcal{V}|})$ (Alg. 4).

The algorithm we propose trains the network and find the clusters in an iterative way, without the need of any prior knowledge. The idea is that once in a while – say, after each epoch – we re-clusterize the word w to the cluster c which output layer is closer to that of w, in euclidian distance. We use a very simple trick to keep the clusters balanced.

We have tried this method on both small and large-scale problems. Our small-scale problem is a database generated by a probabilistic grammar containing sentences such as "the big dog eats a small cat", "a(n) ugly cat annoys a dog" and "the cat kills the cat" (in fact the text is in french, but this is approximately the king of sentences there is). The vocabulary consists of 15 words, including start and end of sentence symbols, plus an "unknown" word symbol and a "pause" symbol which don't appear in the training set. What is fun is that we can see that after 7 epoch the clusters fully stabilize to clusters that make sense from a semantical and syntaxical point of view. This is the words the different clusters contained:

1. "(pause)" "(unknown)" "(start)" "(end)"

Algorithm 4 Training of a cluster NNLM by stochastic gradient descent

function *clusterNNLM*($D_N = \{(w_t, h_t)\}_{t=1}^{N}$)
  $\theta_0 \leftarrow$ *init()* {Initialize parameters}
  $a \leftarrow 0$ {Network's age (total number of examples seen)}
  $e \leftarrow 0$ {Number of epochs}
  $r_0 \leftarrow \infty$ {Empirical risk (neg. log-likelihood)}
  repeat
    {Clusterize the words}
    $b_{max} \leftarrow |\mathcal{V}| \mod |\mathcal{C}|$ {Max. number of big clusters (clusters with s_{max} words)}
    $s_{max} \leftarrow |\mathcal{V}|/|\mathcal{C}|$ {Max. number of words per cluster}
    if $b_{max} > 0$ **then**
      $s_{max} \leftarrow s_{max} + 1$
    $b \leftarrow 0$ {Current number of big clusters}
    forall $c \in \mathcal{C}$ **do**
      $\mathcal{W}_c \leftarrow \emptyset$
    forall $w \in \mathcal{V}$ **do**
      $c \leftarrow argmin_{c' \in \mathcal{C}:|\mathcal{W}_{c'}| < s_{max}} d(\alpha_w, \alpha_{c'})$
      $c_w \leftarrow c$
      $\mathcal{W}_c \leftarrow \mathcal{W}_c \bigcup \{w\}$
      if $|\mathcal{W}_c| = s_{max}$ **then** {This is a big cluster}
        $b \leftarrow b + 1$
        if $b = b_{max}$ **then** {Max. number of big clusters reached}
          $s_{max} \leftarrow s_{max} - 1$
    {Train for an epoch}
    $r_e \leftarrow 0$
    for $t \leftarrow 1$ **to** N **do**
      $\theta_{a+1} \leftarrow \theta_a - \eta_a \left[-\frac{\partial \log P(w_t|c_{w_t}, h_t)}{\partial \theta} - \frac{\partial \log P(c_{w_t}|h_t)}{\partial \theta} \right]$
      $r_e \leftarrow r_e - \log P(w_t|c_{w_t}, h_t) - \log P(c_{w_t}|h_t)$
      $a \leftarrow a + 1$
    $r_e \leftarrow r_e/N$
    $e \leftarrow e + 1$
  until $|r_{e-1} - r_{e-2}| < \epsilon$

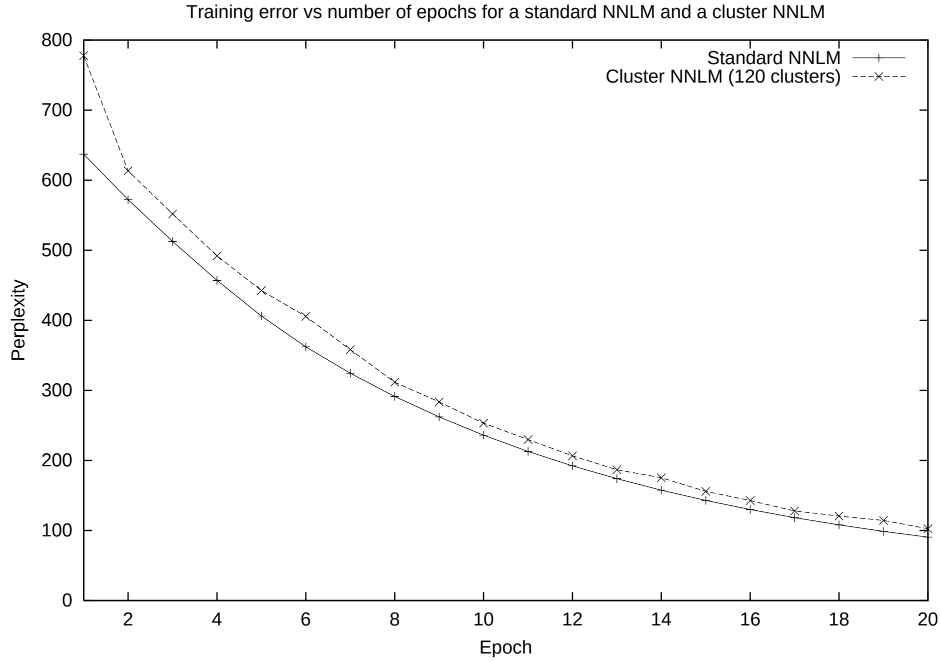

Figure 1: Comparison of convergence of the standard NNLM model and the cluster NNLM on 10K examples of Brown. Both models had 40 hidden units and 20 code features for word/cluster encoding.

2. "annoys" "eats" "small" "kills"
3. "cat" "dog" "big" "ugly"
4. "hastily" "the" "a(n)".

Now, if we run on a large-scale problem (Brown corpus, 800K training examples), we can see that the perplexity decreases from one epoch to the other. However, it does not seem to decrease as fast as the normal NNLM but we have not tuned it yet so it is difficult for now to tell wether it works or not. But since it goes – at least theoreticaly – 100 times faster than the normal algorithm, even if it does not converge as fast, it might still converge to a good local minimum in less time.

Just to give an idea, let us look at the convergence rate of the standard NNLM and a cluster NNLM with 120 classes on a subset of Brown (10K training examples) (Fig. 1). We can see that the cluster NNLM seems to converge approximately as fast as the standard NNLM. We still have to investigate on this.

References

[Bengio et al., 2003] Bengio, Y., Ducharme, R., Vincent, P., and Jauvin, C. (2003). A neural probabilistic language model. Journal of Machine Learning Research.

[Bengio and Senécal, 2003a] Bengio, Y. and Senécal, J.-S. (2003a). Adaptive importance sampling to accelerate training of a neural probabilistic language model. IDIAP-RR 35, IDIAP.

[Bengio and Senécal, 2003b] Bengio, Y. and Senécal, J.-S. (2003b). Quick training of probabilistic neural nets by sampling. In *Proceedings of the Ninth International Workshop on Artificial Intelligence and Statistics*, volume 9, Key West, Florida. AI and Statistics.

[Corcoran and Tweedie, 2002] Corcoran, J. and Tweedie, R. (2002). Perfect sampling from independent metropolis-hastings chains. *Jour. Stat. Plan. Infer.*, 2(104):297–314.