IDIAP RESEARCH REPORT

Adaptive Importance Sampling to Accelerate Training of a Neural Probabilistic Language Model

Jean-Sébastien Senécal [a] [b] Yoshua Bengio [a]

IDIAP–RR 03-35

September 2003

submitted for publication

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

[a] Département d'Informatique et de Recherche Opérationnelle, Université de Montréal, Montréal, Québec, Canada, H3J 3J7
[b] Institut Dalle Molle d'Intelligence Adaptative et Perceptive, 4, rue du Simplon, 1920 Martigny, Valais, Switzerland

IDIAP Research Report 03-35

Adaptive Importance Sampling to Accelerate Training of a Neural Probabilistic Language Model

Jean-Sébastien Senécal   Yoshua Bengio

September 2003

submitted for publication

Abstract. Previous work on statistical language modeling has shown that it is possible to train a feed-forward neural network to approximate probabilities over sequences of words, resulting in significant error reduction when compared to standard baseline models. However, in order to train the model on the maximum likelihood criterion, one has to make, for each example, as many network passes as there are words in the vocabulary. We introduce adaptive importance sampling as a way to accelerate training of the model. We show that a very significant speed-up can be obtained on standard problems.

1 Introduction

Statistical language modeling focuses on trying to infer the underlying distribution that generated a sequence of words $w_1, \ldots, w_T$ in order to estimate $P(w_1^T)$ [1].

The distribution can be represented by the conditional probability of the next word given all the previous ones:

$$P(w_1^T) = \prod_{t=1}^{T} P(w_t | w_1^{t-1}). \quad (1)$$

In order to reduce the difficulty of the modeling problem, one usually compresses the information brought by the last words by considering only the last $n-1$ words, thus yielding the approximation

$$P(w_1^T) \approx \prod_{t=1}^{T} P(w_t | w_{t-n+1}^{t-1}). \quad (2)$$

The conditional probabilities $P(w_t | w_{t-n+1}^{t-1})$ can be easily modeled by considering sub-sequences of length n, usually referred to as *windows*, and computing the estimated joint probabilities $P(w_{t-n+1}^t)$ and $P(w_{t-n+1}^{t-1})$; the model's conditional probabilities can then be computed as

$$P(w_t | w_{t-n+1}^{t-1}) = \frac{P(w_{t-n+1}^t)}{P(w_{t-n+1}^{t-1})}. \quad (3)$$

Successful traditional approaches, called *n-grams*, consist of simply counting the frequency of appearance of the various windows of words in a training corpus i.e. the conditional probabilities are set as $P(w_t | v_{t-n+1}^{t-1}) = \frac{|w_{t-n+1}^t|}{|w_{t-n+1}^{t-1}|}$ where $|w_i^j|$ is just the frequency of subsequence w_i^j in the training corpus. The main problem with these methods is that they tend to overfit as n becomes large, preventing in practice the use of large windows. In order to smooth the models, they are usually interpolated with lower-order n-grams in order to redistribute some of the probability mass.

1.1 Fighting the Curse of Dimensionality With Word Similarity

The problem faced by n-grams is just a special case of the *curse of dimensionality*. Word vocabularies being usually large i.e. in the order of ten to hundred thousand words, modeling the joint distribution of, say, 10 words potentially requires 10^{40} to 10^{50} free parameters. Since these models do not take advantage of the similarity between words, they tend to redistribute probability mass too blindly, mostly to sentences with a very low probability.

The solution, first proposed in (Bengio et al., 2003), and inspired by previous work on symbolic representation with neural networks, such as (Hinton, 1986), is to map the words in vocabulary $\mathcal{V}$ into a *feature space* $\mathbb{R}^m$ in which the notion of similarity is grasped by a distance measure on word features. That is, to find a mapping C from the *discrete* word space to a *continuous* semantic space. In the proposed approach, this mapping is done by simply assigning a *feature vector* $C(w) \in \mathbb{R}^m$ to each word w of the vocabulary. The vectors are considered as parameters of the model and are thus learned during training, by gradient descent.

1.2 An Energy-based Neural Network for Language Modeling

Many variants of this neural network language model exist, as presented in (Bengio et al., 2003). Here we formalize a particular one.

The output of the neural network depends on the next word w_t and the history $h_t = w_{t-n+1}^{t-1}$ as follows. In the *features layer*, one maps each word w_{t-i} in w_{t-n+1}^t to a lower-dimensional continuous

[1] In this paper, we refer to sub-sequence $w_i, \ldots, w_j$ as w_i^j, for simplicity.

subspace $\mathbf{z}_i$. In the framework we propose, the target (next) word is mapped to a different feature space than the context (last) words.

$$
\begin{aligned}
\mathbf{z}_i &= \mathbf{C}_{w_{t-i}}, \quad i = 1, \ldots, n-1, \\
\mathbf{z}_0 &= \mathbf{D}_{w_t}, \\
\mathbf{z} &= (\mathbf{z}_0, \ldots, \mathbf{z}_{n-1})
\end{aligned}
\tag{4}
$$

where $\mathbf{C}_j$ is the j-th column of the *word features* matrix $\mathbf{C}$ of free parameters for the context words w_{t-n+1}^{t-1} and $\mathbf{D}_j$ is the j-th column of the word features matrix $\mathbf{D}$ for the target word w_t. The resulting vector $\mathbf{z}$ (the concatenation of the projections $\mathbf{z}_i$) is the input vector for the next layer, the hidden layer:

$$\mathbf{a} = \tanh(\mathbf{d} + \mathbf{W}\mathbf{z}) \tag{5}$$

where $\mathbf{d}$ is a vector of free parameters (hidden units biases), $\mathbf{W}$ is a matrix of free parameters (hidden layer weights) and $\mathbf{a}$ is a vector of hidden units activations. Finally the output is a scalar energy function

$$\mathcal{E}(w_t, h_t) = b_{w_t} + \mathbf{V}_{w_t} \cdot \mathbf{a} \tag{6}$$

where $\mathbf{b}$ is a vector of free parameters (called biases), and $\mathbf{V}$ (hidden to output layer weights) is a matrix of free parameters with one column $\mathbf{V}_i$ per word.

To obtain the joint probability of (w_t, h_t), we normalize the exponentiated energy $e^{-\mathcal{E}(w_t,h_t)}$ by dividing it by the normalizing function $Z = \sum_{(w,h)\in\mathcal{V}^n} e^{-\mathcal{E}(w,h)}$:

$$P(w_t, h_t) = \frac{e^{-\mathcal{E}(w_t,h_t)}}{Z}, \tag{7}$$

The normalization Z is extremely hard to compute, since it requires an exponential number of passes i.e. computations of $\mathcal{E}(\cdot)$. Luckily, we don't need the joint probability but the conditional probability $P(w_t|h_t)$. By observing that $P(h_t) = \sum_{w\in\mathcal{V}} P(w, h_t)$ and plugging it and (7) in (3), the normalization function vanishes and one obtains

$$P(w_t|h_t) = \frac{e^{-\mathcal{E}(w_t,h_t)}}{Z(h_t)}, \tag{8}$$

where $Z(h_t) = \sum_{w\in\mathcal{V}} e^{-\mathcal{E}(w,h_t)}$ is tractable, tough still hard to compute because $|\mathcal{V}|$ is usually large and the activation $\mathcal{E}(\cdot)$ is hard to compute.

The above architecture can be seen as an *energy-based model*, that is, a probabilistic model based on the Boltzmann energy distribution. In an energy-based model, the probability distribution of a random variable X over some set $\mathcal{X}$ is expressed as

$$P(X = x) = \frac{e^{-\mathcal{E}(x)}}{Z} \tag{9}$$

where $\mathcal{E}(\cdot)$ is a parametrized *energy* function which is low for plausible configurations of x, and high for improbable ones, and where $Z = \sum_{x\in\mathcal{X}} e^{-\mathcal{E}(x)}$ (or $Z = \int_{x\in\mathcal{X}} e^{-\mathcal{E}(x)}$ in the continuous case), is called the *partition function*. In the case that interests us, the partition function depends of the context h_t, as seen in (8).

The main step in a gradient-based approach to train such models involves computing the gradient of the log-likelihood $\log P(X = x)$ with respect to parameters θ. The gradient can be decomposed in *two parts*: *positive reinforcement* for the observed value $X = x$ and *negative reinforcement* for every x', weighted by $P(X = x')$, as follows (by differentiating (9)):

$$\nabla_\theta \left(-\log P(x)\right) = \nabla_\theta \left(\mathcal{E}(x)\right) - \sum_{x'\in\mathcal{X}} P(x') \nabla_\theta \left(\mathcal{E}(x')\right). \tag{10}$$

Clearly, the difficulty here is to compute the negative reinforcement when $|\mathcal{X}|$ is large (as is the case in a language modeling application). However, as is easily seen, the negative part of the gradient is nothing more than the average

$$E_P\left[\nabla_\theta\left(\mathcal{E}(X)\right)\right]. \tag{11}$$

In (Hinton, 2002), the author proposes to estimate this average with a sampling method known as a Monte-Carlo Markov Chain (Gibbs sampling). Unfortunately, this technique relies on the particular form of the energy function in the case of products of experts, which lends itself naturally to Gibbs sampling (using the activities of the hidden units as one of the random variables, and the network input as the other one).

2 Approximation of the Log-Likelihood Gradient by Biased Importance Sampling

A traditional way to estimate (11) would consist of sampling n points $x_1, \ldots, x_n$ from the network's distribution $P(\cdot)$ and to approximate (11) by the average

$$\frac{1}{n}\sum_{i=1}^{n}\nabla_\theta\left(\mathcal{E}(x_i)\right). \tag{12}$$

This method, known as *classical Monte-Carlo*, yields Algorithm 1 for estimating the gradient of the log-likelihood (10). The maximum speed-up that could be achieved with such a procedure would be $|\mathcal{X}|/n$. In the case of the language modeling application we are considering, that means a potential for a huge speed-up, since $|\mathcal{X}|$ is typically in the tens of thousands and n could be quite small; in fact, Hinton found $n = 1$ to be a good choice with the contrastive divergence method (Hinton, 2002).

Algorithme 1 Classical Monte-Carlo Approximation of the Gradient

$\nabla_\theta\left(-\log P(x)\right) \leftarrow \nabla_\theta\left(\mathcal{E}(x)\right)$ {Add positive contribution}
for $k \leftarrow 1$ **to** n **do** {Estimate negative contribution}
  $x' \sim P(\cdot)$ {Sample negative example}
  $\nabla_\theta\left(-\log P(x)\right) \leftarrow \nabla_\theta\left(-\log P(x)\right) - \frac{1}{n}\nabla_\theta\left(\mathcal{E}(x')\right)$ {Add negative contribution}
end for

However, this method requires to sample from distribution $P(\cdot)$, which we can't do without having to compute $P(x)$ explicitly. That means we have to compute the partition function Z, which is still hard because we have to compute $\mathcal{E}(x)$ for each $x \in \mathcal{X}$.

Fortunately, in many applications, such as language modeling, we can use an alternative, ***proposal*** distribution Q from which it is cheap to sample. In the case of language modeling, for instance, we can use n-gram models. There exist several Monte-Carlo algorithms that can take advantage of such a distribution to give an estimate of (11).

2.1 Classical Importance Sampling

One well-known statistical method that can make use of a proposal distribution Q in order to approximate the average $E_P[\nabla_\theta\left(\mathcal{E}(X)\right)]$ is based on a simple observation. In the discrete case

$$E_P[f(X)] = \sum_{x\in\mathcal{X}} P(x)\nabla_\theta\left(\mathcal{E}(x)\right) = \sum_{x\in\mathcal{X}} Q(x)\frac{P(x)}{Q(x)}\nabla_\theta\left(\mathcal{E}(x)\right) = E_Q\left[\frac{P(X)}{Q(X)}\nabla_\theta\left(\mathcal{E}(X)\right)\right]. \tag{13}$$

Thus, if we take m independent samples $y_1, \ldots, y_n$ from Q and apply classical Monte-Carlo to estimate $E_Q\left[\frac{P(X)}{Q(X)}\nabla_\theta\left(\mathcal{E}(X)\right)\right]$, we obtain the following estimator known as *importance sampling*:

$$\frac{1}{m}\sum_{i=1}^{m}\frac{P(y_i)}{Q(y_i)}\nabla_\theta\left(\mathcal{E}(y_i)\right). \tag{14}$$

Clearly, that does not rule the problem out because though we don't need to sample from P anymore, the $P(y_i)$'s still need to be computed, which cannot be done without explicitly computing the partition function. Back to square one.

2.2 Biased Importance Sampling

By chance, there is a way to estimate (11) without sampling from P nor having to compute the partition function. The proposed estimator is a biased version of classical importance sampling (Kong et al., 1994). It can be used when $P(x)$ can be computed explicitly up to a multiplicative constant: in the case of energy-based models, this is clearly the case since $P(x) = Z^{-1}e^{-\mathcal{E}(x)}$. The idea is to use $\frac{1}{W}w(y_i)$ to weight the $\nabla_\theta\left(\mathcal{E}(y_i)\right)$, with $w(x) = \frac{e^{-\mathcal{E}(x)}}{Q(x)}$ and $W = \sum_{j=1}^{m} w(y_i)$, thus yielding the estimator

$$\frac{1}{W}\sum_{i=1}^{m} w(y_i)\nabla_\theta\left(\mathcal{E}(y_i)\right). \tag{15}$$

Though this estimator is biased, its bias decreases as m increases. It can be shown to converge to the true average (11) as $m \to \infty$.

The advantage of using this estimator over classical importance sampling is that we no more need to compute the partition function: we just need to compute the energy function for the sampled points. The procedure is summarized in Algorithm 2.

Algorithme 2 Biased Importance Sampling Approximation of the Gradient

$\nabla_\theta\left(-\log P(x)\right) \leftarrow \nabla_\theta\left(\mathcal{E}(x)\right)$ {Add positive contribution}
vector $\mathbf{g} \leftarrow \mathbf{0}$
$W \leftarrow 0$
for $k \leftarrow 1$ **to** m **do** {Estimate negative contribution}
  $y' \sim Q(\cdot)$ {Sample negative example}
  $w \leftarrow \frac{e^{-\mathcal{E}(y')}}{Q(y')}$
  $\mathbf{g} \leftarrow \mathbf{g} + w\nabla_\theta\left(\mathcal{E}(y')\right)$
  $W \leftarrow W + w$
end for
$\nabla_\theta\left(-\log P(x)\right) \leftarrow \nabla_\theta\left(-\log P(x)\right) - \frac{1}{W}\mathbf{g}$ {Add negative contributions}

3 Adapting the Sample Size

Preliminary experiments with Algorithm 2 using the unigram distribution showed that whereas a small sample size was appropriate in the initial training epochs, a larger sample size was necessary later to avoid divergence (increases in training error). This may be explained by a too large bias – because the network's distribution diverges from that of the unigram, as training progresses – and/or by a too large variance in the gradient estimator.

In (Bengio and Senécal, 2003), we presented an improved version of Algorithm 2 that makes use of a diagnostic, called *effective sample size* (Kong, 1992; Kong et al., 1994). For a sample $y_1, \ldots, y_m$

taken from proposal distribution Q, the effective sample size is given by

$$ESS = \frac{(\sum_{j=1}^{m} w(y_i))^2}{\sum_{j=1}^{m} w^2(y_j)}. \tag{16}$$

Basically, this measure approximates the number of samples from the target distribution P that would have yielded, with classical Monte-Carlo, the same variance as that yielded by the biased importance sampling estimator with sample $y_1, \ldots, y_m$.

We can use this measure to diagnose whether we have sampled enough points. In order to do that, we fix a baseline sample size n. This baseline is the number of samples we would sample in a classical Monte-Carlo scheme, were we able to do it. We then sample points from Q by "blocks" of size $m_b \geq 1$ until the effective sample size becomes larger that target n. If the number of samples becomes too large, we switch back to a full back-propagation (i.e. we compute the true negative gradient).

4 Adapting the Proposal Distribution

The method was used with a simple unigram proposal distribution to yield significant speed-up on the Brown corpus (Bengio and Senécal, 2003). However, the required number of samples was found to increase quite drastically as training progresses. This is because the unigram distribution stays fix while the network's distribution changes over time and becomes more and more complex, thus diverging from the unigram. Switching to a bigram or trigram during training actually worsens even more the training, requiring even larger samples.

Clearly, using a proposal distribution that stays "close" to the target distribution would yield even greater speed-ups, as we would need less samples to approximate the gradient. We propose to use a n-gram model that is *adapted* during training to fit to the target (neural network) distribution P [2] In order to do that, we propose to *redistribute the probability mass* of the sampled points in the n-gram to track P.

Let us consider the following *adaptive n-gram*:

$$Q(w_t|h_t) = \sum_{k=1}^{n} \alpha_k(h_t) Q_k(w_t|w_{t-k+1}^{t-1}) \tag{17}$$

where the Q_k are the sub-models and $\alpha_k(h_t)$ is a mixture function such that $\sum_{k=1}^{n} \alpha_k(h_t) = 1$ [3].

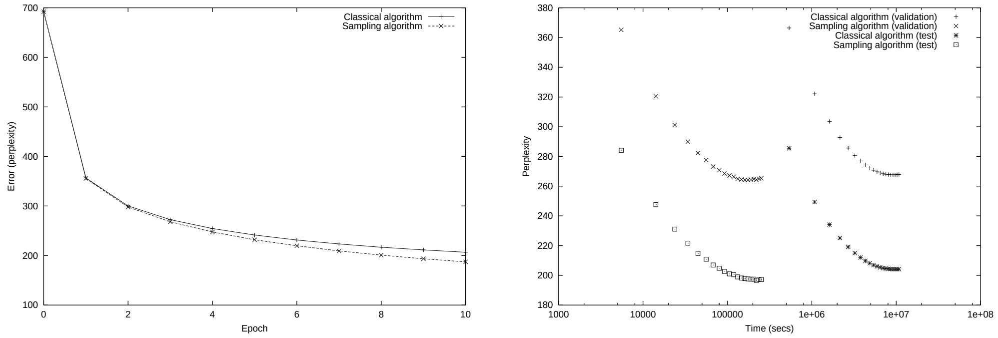
Let $\mathcal{W}$ be the set of m words sampled from Q. Let $\bar{q_k} = \sum_{w \in \mathcal{W}} Q_k(w|w_{t-k+1}^{t-1})$ be the total probability mass of the sampled points in k-gram Q_k and $\bar{p} = \sum_{w \in \mathcal{W}} e^{-\mathcal{E}(w,h_t)}$ the unnormalized probability mass of these points in P. Let $\tilde{P}(w|h_t) = \frac{e^{-\mathcal{E}(w,h_t)}}{\bar{p}}$ for each $w \in \mathcal{W}$. For each k and for each $w \in \mathcal{W}$, the values in Q_k are updated as follows:

$$Q_k(w|w_{t-k+1}^{t-1}) \leftarrow (1-\lambda) Q_k(w|w_{t-k+1}^{t-1}) + \lambda \bar{q_k} \tilde{P}(w|h_t)$$

where λ is a kind of "learning rate". The parameters of function $\alpha(\cdot)$ are updated so as to minimize the Kullback-Leibler divergence $\sum_{w \in \mathcal{W}} \tilde{P}(w|h_t) \log \frac{\tilde{P}(w|h_t)}{Q(w|h_t)}$ by gradient descent. We describe here the method we used to train the α's in the case of a bigram interpolated with a unigram. In our experiments, the α's were a function of the frequency of the last word w_{t-1}. The words were first clustered in C frequency bins $\mathcal{B}_c, c = 1, \ldots, C$ such that $\forall i, j \sum_{w \in \mathcal{B}_i} |w| \approx \sum_{w \in \mathcal{B}_j} |w|$. Then, an "energy" value $a(c)$ was assigned for $c = 1, \ldots, C$. We set $\alpha_1(h_t) = \sigma(a(h_t))$ and $\alpha_2(h_t) = 1 - \alpha_1(h_t)$

[2] A similar approach was proposed in (Cheng and Druzdzel, 2000) for Bayesian networks.

[3] Usually, for obvious reasons of memory constraints, the probabilities given by a n-gram will be non-null only for those sequences that were observed in the training set. Mixing with lower-order models allows to give some probability mass to unseen word sequences.



(a) Training error wrt number of epochs

(b) Validation and test errors wrt CPU time

Figure 1: Comparison of errors between a model trained with the classical algorithm and a model trained by adaptive importance sampling.

where σ is the sigmoid function and $a(h_t) = a(w_{t-1}) = a(c)$, c being the class (bin) of w_{t-1}. The energy $a(h_t)$ is thus updated with the following rule:

$$a(h_t) \leftarrow a(h_t) - \alpha(h_t)(1 - \alpha(h_t)) \sum_{w \in \mathcal{W}} \tilde{P}(w|h_t) \frac{Q(w|h_t)}{Q_2(w|w_{t-1}) - Q_1(w)}.$$

5 Experimental Results

We ran some experiments on the Brown corpus, with different configurations. The Brown corpus consists of 1,181,041 words from various American English documents. The corpus was divided in train (800K words), validation (200K words) and test (the remaining $\approx$ 180K words) sets. The vocabulary was truncated by mapping all "rare" words (words that appear 3 times or less in the corpus) into a single special word. The resulting vocabulary contains 14,847 words.

On this dataset, a simple interpolated trigram, serving as our baseline, achieves a perplexity of 253.8 on the test set [4].

In all settings, we used 30 word features for both context and target words, and 80 hidden neurons. The number of context words was 3. This setting is the same as that of the neural network that achieved the best results on Brown, as described in (Bengio et al., 2003). In this setting, a classical neural network – one that doesn't make a sampling approximation of the gradient – converges to a perplexity of 204 in test, after 18 training epochs.

Figure 1(a) plots the training error at every epoch for the network trained without sampling and a network trained by importance sampling, using an adaptive bigram with a target effective sample size of 50. The number of frequency bins used for the mixing variables was 10. It shows that the convergence of both network is similar. The same holds for validation and test errors, as is shown in figure 1(b). In this figure, the errors are plotted wrt computation time on a Pentium 4 2 GHz. As can be seen, the network trained with the sampling approximation converges before the network trained classically even finishes to complete one full epoch.

Quite interestingly, the network trained by sampling converges to an even lower perplexity than the classical one. After 9 epochs (26 hours), it's perplexity over the test set is equivalent to that of the other at its overfitting point (18 epochs, 113 days). The sampling approximation thus allowed a **100-fold speed-up**.

[4] Better results can be achieved with a Knesser-Ney back-off trigram, but it has been shown in (Bengio et al., 2003) that a neural network still converges to a lower perplexity on Brown. Furthermore, the neural network can be interpolated with the trigram for even bigger perplexity reductions.

Surprisingly enough, if we let the sampling-trained model converge, it starts to overfit at epoch 18 – as for classical training – but with a lower test perplexity of 196.6, a 3.8% improvement. Total improvement in test perplexity wrt the trigram baseline is 29%.

Apart from the speed-up, the other interesting thing if we compare the results with those obtained by using a non-adaptive proposal distribution (Bengio and Senécal, 2003) is that the mean number of samples required in order to ensure convergence seems to grow almost linearly with time, whereas the required number of samples with the non-adaptive unigram was growing exponentially.

6 Future Work

Previous work (Bengio et al., 2003) used a parallel implementation in order to speed-up training and testing. Although our sampling algorithm works very well on a single machine, we had much trouble making an efficient parallel implementation of it. The reason is that the parallelization has to be done on the hidden layer; thus for each back-propagation, we have to accumulate the gradient wrt the feature parameters (the $\mathbf{z}_i$'s) for each processor and then share the gradients. The process of sharing the gradient necessitates huge resources in terms of data transmission, which we have found to take up to 60% of the back-propagation time. One way to deal with the problem is to desynchronize the sharing of the parameters on the feature vectors i.e. allowing the computations to continue while the messages are transmitted. Since the changes in the feature vectors are quite small, this should not affect convergence.

The other problem we face is that of choosing the target effective sample size. Currently, we have to choose it conservatively enough to guarantee convergence. In fact, we could achieve the same convergence by adapting it to the gradient's variance: as training progresses, the gradient is likely to become noisier, thus necessitating a bigger number of samples for even the classical Monte-Carlo estimate to yield a good approximation. We could same even more computations by targeting a smaller effective sample size at the start of training and increasing it after.

7 Conclusion

In this paper, we proposed a simple method to efficiently train a probabilistic energy-based neural network. Though the application was to language modeling with a neural network, the method could in fact be used to train arbitrary energy-based models as well.

The method is based on the idea that the gradient of the log-likelihood can be decomposed in two parts: positive and negative contributions. The negative contribution is usually hard to compute because it involves a number of passes through the network equivalent to the size of the vocabulary. Luckily, it can be estimated efficiently by importance sampling.

We had already argued for such a method in (Bengio and Senécal, 2003), achieving a significant 19-fold speed-up on a standard problem (Brown). Our new contribution is to *adapt* the proposal distribution as training progresses so that it stays as close as possible to the network's distribution. We have showed that it is possible to do it efficiently by *reusing the sampled words* to re-weight the probabilities given by a n-gram. With the new method, we were able to achieve an even more significant 100-fold speed-up on the same problem. Analysis of the required sample size through time also suggest that the algorithm will scale with more difficult problems, since the mean sample size is proportional to the number of epochs.

Acknowledgments

The authors would like to thank Geoffrey Hinton for fruitful discussions, and the following funding organizations: NSERC, MITACS, and the Canada Research Chairs.

References

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

Bengio, Y. and Senécal, J.-S. (2003). 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.

Cheng, J. and Druzdzel, M. J. (2000). Ais-bn: An adaptive importance sampling algorithm for evidential reasoning in large bayesian networks. *Journal of Artificial Intelligence Research*, 13:155–188.

Hinton, G. (1986). Learning distributed representations of concepts. In *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*, pages 1–12, Amherst. Lawrence Erlbaum, Hillsdale.

Hinton, G. (2002). Training products of experts by minimizing contrastive divergence. *Neural Computation*, 14(8):1771–1800.

Kong, A. (1992). A note on importance sampling using standardized weights. Technical Report 348, Department of Statistics, University of Chicago.

Kong, A., Liu, J. S., and Wong, W. H. (1994). Sequential imputations and bayesian missing data problems. *Journal of the American Statistical Association*, 89:278–288.