6 Neural Probabilistic Language Models

Yoshua Bengio[1], Holger Schwenk[2],
Jean-Sébastien Senécal[1], Fréderic Morin[1] and Jean-Luc Gauvain[2]

1. Département d'Informatique et Recherche Opérationnelle,
Université de Montréal, Montréal, Québec, Canada
bengioy@iro.umontreal.ca

2. Groupe Traitement du Langage Parlé
LIMSI-CNRS, Orsay, France
schwenk@limsi.fr

Abstract

A central goal of statistical language modeling is to learn the joint probability function of sequences of words in a language. This is intrinsically difficult because of the **curse of dimensionality**: a word sequence on which the model will be tested is likely to be different from all the word sequences seen during training. Traditional but very successful approaches based on n-grams obtain generalization by concatenating very short overlapping sequences seen in the training set. We propose to fight the curse of dimensionality by **learning a distributed representation for words** which allows each training sentence to inform the model about an exponential number of semantically neighboring sentences. Generalization is obtained because a sequence of words that has never been seen before gets high probability if it is made of words that are similar (in the sense of having a nearby representation) to words forming an already seen sentence. Training such large models (with millions of parameters) within a reasonable time is itself a significant challenge. We report on several methods to speed-up both training and probability computation, as well as comparative experiments to evaluate the improvements brought by these techniques. We finally describe the incorporation of this new language model into a state-of-the-art speech recognizer of conversational speech.

6.1 Introduction

A fundamental problem that makes language modeling and other learning problems difficult is the *curse of dimensionality*. It is particularly obvious in the case when one wants to model the joint distribution between many discrete random variables (such as words in a sentence, or discrete attributes in a data-mining task). For example, if one wants to model the joint distribution of 10 consecutive words in a natural language with a vocabulary V of size 100,000, there are potentially $100\,000^{10} - 1 = 10^{50} - 1$ free parameters. When modeling continuous variables, we obtain generalization more easily (e.g. with smooth classes of functions like multi-layer neural networks or Gaussian mixture models) because the function to be learned can be expected to have some local smoothness properties. For discrete spaces, the generalization structure is not as obvious: any change of these discrete variables may have a drastic impact on the value of the function to be estimated, and when the number of values that each discrete variable can take is large, most observed objects are almost maximally far from each other in Hamming distance.

A useful way to visualize how different learning algorithms generalize, inspired from the view of non-parametric density estimation, is to think of how probability mass that is initially concentrated on the training points (e.g., training sentences) is distributed in a larger volume, usually in some form of neighborhood around the training points. In high dimensions, it is crucial to distribute probability mass where it matters rather than uniformly in all directions around each training point.
A statistical model of language can be represented by the conditional probability of the next word given all the previous ones, since

$$\hat{P}(w_1^T) = \prod_{t=1}^{T} \hat{P}(w_t \mid w_1^{t-1}),$$

where w_t is the t-th word, and writing sub-sequence $w_i^j = (w_i, w_{i+1}, ..., w_{j-1}, w_j)$. Such statistical language models have already been found useful in many technological applications involving natural language, such as speech recognition, language translation, and information retrieval. Improvements in statistical language models could thus have a significant impact on such applications.

When building statistical models of natural language, one considerably reduces the difficulty of this modeling problem by taking advantage of word order, and the fact that temporally closer words in the word sequence are statistically more dependent. Thus, *n-gram* models construct tables of conditional probabilities for the next word, for each one of a large number of *contexts*, i.e. combinations of the last $n-1$ words:

$$\hat{P}(w_t \mid w_1^{t-1}) \approx \hat{P}(w_t \mid w_{t-n+1}^{t-1}).$$

We only consider those combinations of successive words that actually occur in the training corpus, or that occur frequently enough. What happens when a new combination of n words appears that was not seen in the training corpus? We do not want to assign zero probability to such cases, because new combinations are likely to occur, and they will occur even more frequently for larger context sizes. A simple answer is to look at the probability predicted using a smaller context size, as done in back-off trigram models [Katz, 1987] or in smoothed (or interpolated) trigram models [Jelinek and Mercer, 1980]. A way to understand how such models obtain generalization to new sequences of words is to think about a corresponding generative model. Essentially, a new sequence of words is generated by "gluing" very short and overlapping pieces of length 1, 2 ... or up to n words that have been seen frequently in the training data. The rules for obtaining the probability of the next piece are implicit in the particulars of the back-off or interpolated n-gram algorithm. Typically researchers have used $n = 4$, i.e. fourgrams, and obtained state-of-the-art results, but see [Goodman, 2001a] for how combining many tricks can yield to substantial improvements.

However, there is obviously much more information in the sequence that immediately precedes the word to predict than just the identity of the previous couple of words. In addition, this approach does not take advantage of a notion of similarity between words that would go beyond equality of words. For example, having seen the sentence "The cat is walking in the bedroom" in the training corpus should help us generalize to make the sentence "A dog was running in a room" almost as likely, simply because "dog" and "cat" (resp. "the" and "a", "room" and "bedroom", etc...) have similar semantic and grammatical roles.

There are many approaches that have been proposed to address these two issues, and we will briefly explain in Section 6.1.2 the relations between the approach proposed here and some of these earlier approaches. We will first discuss what is the basic idea of the proposed approach. A more formal presentation will follow in Section 6.2, using an implementation of these ideas that relies on shared-parameter multi-layer neural networks. Another very important part of this chapter are methods for efficiently training such very large neural networks (with millions of parameters) for very large data sets (with tens of millions of examples).

Many operations in this paper are in matrix notation, with lower case v denoting a column vector and v' its transpose, A_j the j-th row of a matrix A, and $x \cdot y = x'y$.

6.1.1 Fighting the Curse of Dimensionality with Distributed Representations

In a nutshell, the idea of the proposed approach can be summarized as follows:

1. associate with each word in the vocabulary a distributed *word feature vector* (a real-valued vector in $\mathbb{R}^m$),
2. express the joint *probability function* of word sequences in terms of the feature vectors of these words in the sequence, and
3. learn simultaneously the *word feature vectors* and the parameters of that *probability function*.

The feature vector represents different aspects of the word: each word is associated with a point in a vector space. The number of features (e.g. m=30...100 in the experiments) is much smaller than the size of the vocabulary (e.g. 10 000 to 100 000). The probability function is expressed as a product of conditional probabilities of the next word given the previous ones, (e.g. using a multi-layer neural network to predict the next word given the previous ones, in the experiments). This function has parameters that can be iteratively tuned in order to **maximize the log-likelihood of the training data** or a regularized criterion, e.g. by adding a weight decay penalty[1]. The feature vectors associated with each word are learned, but they could be initialized using prior knowledge of semantic features.

Why does it work? In the previous example, if we knew that dog and cat played similar roles (semantically and syntactically), and similarly for (the,a), (bedroom,room*)*, (is,was*)*, (running,walking*)*, we could naturally generalize (i.e. transfer probability mass) from

The cat is walking in the bedroom

to A dog was running in a room

[1]Like in ridge regression, the squared norm of the parameters is penalized.

and likewise to The cat is running in a room
A dog is walking in a bedroom
The dog was walking in the room
…

and many other combinations. In the proposed model, it will so generalize because "similar" words are expected to have a similar feature vector. Since the probability function is a *smooth* function of these feature values, a small change in the features will induce a small change in the probability. Therefore, the presence of only one of the above sentences in the training data will increase the probability, not only of that sentence, but also of its combinatorial number of "neighbors" in sentence space (as represented by sequences of feature vectors).

6.1.2 Relation to Previous Work

The idea of using neural networks to model high-dimensional discrete distributions has already been found useful to learn the joint probability of $Z_1 \cdots Z_n$, a set of random variables where each is possibly of a different nature [Bengio and Bengio, 2000a,b]. In that model, the joint probability is decomposed as a product of conditional probabilities

$$\hat{P}(Z_1 = z_1, \cdots, Z_n = z_n)$$
$$\overset{def}{=} \prod_i \hat{P}(Z_i = z_i \mid g_i(Z_{i-1} = z_{i-1}, Z_{i-2} = z_{i-2}, \cdots, Z_1 = z_1)),$$

where $g(\cdot)$ is a function represented by a neural network with a special left-to-right architecture, with the i-th output block $g_i()$ computing parameters for expressing the conditional distribution of Z_i given the value of the previous Z's, in some arbitrary order. Experiments on four UCI data sets showed this approach to work comparatively very well [Bengio and Bengio, 2000a,b]. Here, however, we must deal with data of variable length, like sentences, so the above approach must be adapted. Another important difference is that here, all the Z_i (word at i-th position) refer to the same type of object (a word). The model proposed here therefore introduces a sharing of parameters across time – the same g_i is used across time – and across input words at different positions. It is a successful large-scale application of the idea in [Bengio and Bengio, 2000a,b], along with the older idea of learning a distributed representation for symbolic data, that was advocated in the early days of connectionism [Hinton, 1986; Elman, 1990]. More recently, Hinton's approach was improved and successfully demonstrated on learning several symbolic relations [Paccanaro and Hinton, 2000]. The idea of using neural networks for language modeling is not new either, e.g. [Miikkulainen and Dyer, 1991]. In contrast, here we push this idea to a **large scale**, and concentrate on learning a **statistical model** of the distribution of word sequences, rather than learning the role of words in a sentence. The approach proposed here is also related to previous proposals of character-based text compression using neural networks to predict the probability of the next character [Schmidhuber, 1996]. The idea of using a neural network for language modeling has also been independently proposed by [Xu and Rudnicky, 2000], although experiments are with networks without hidden units and a single input word, which limits the model to essentially capturing unigram and bigram statistics.

The idea of discovering some similarities between words to obtain generalization from training sequences to new sequences is not new. For example, it is exploited in approaches that are based on learning a clustering of the words [Brown *et al.*, 1992; Pereira *et al.*, 1993; Niesler *et al.*, 1998; Baker and McCallum, 1998]: each word is associated deterministically or probabilistically with a discrete class, and words in the same class are similar in some respect. In the model proposed here, instead of characterizing the similarity with a discrete random or deterministic variable (which corresponds to a soft or hard partition of the set of words), we use a continuous real-vector for each word, i.e. a **learned distributed feature vector**, to represent similarity between words. The experimental comparisons in this paper include results obtained with class-based n-grams [Brown *et al.*, 1992; Ney and Kneser, 1993; Niesler *et al.*, 1998], which are based on replacing the sequence of past words representing the prediction context by the corresponding sequence of word classes (the class of a word being the identity of its cluster).

The idea of using a vector-space representation for words has been well exploited in the area of *information retrieval*, for example see work by [Schutze, 1993], where feature vectors for words are learned on the basis of their probability of co-occurring in the same documents (see also Latent Semantic Indexing [Deerwester *et al.*, 1990]). An important difference is that here we look for a representation for words that is helpful in representing compactly the probability distribution of word sequences from natural language text. Experiments suggest that learning jointly the representation (word features) and the model is very useful. The idea of using a continuous representation for words has been exploited successfully by [Bellegarda, 1997] in the context of an n-gram based statistical language model, using LSI to dynamically identify the topic of discourse. Finally, the approach discussed here is close in spirit to the more recent research on discovering an *embedding* for words or symbols, in a low-dimensional space, as in [Hinton and Roweis, 2003] and [Blitzer *et al.*, 2005].

The idea of a vector-space representation for symbols in the context of neural networks has also previously been framed in terms of a parameter sharing layer, for protein secondary structure prediction [Riis and Krogh, 1996], and text-to-speech mapping [Jensen and Riis, 2000)].

6.2 A Neural Model

The training set is a sequence $w_1 \cdots w_T$ of words $w_t \in V$, where the vocabulary V is a large but finite set. The objective is to learn a good model $f(w_t, \cdots, w_{t-n+1}) = \hat{P}(w_t \mid w_{t-n+1}^{t-1})$, in the sense that it gives high out-of-sample likelihood. In the experiments, we will report the geometric average of $1/\hat{P}(w_t \mid w_{t-n+1}^{t-1})$, also known as *perplexity*, which is also the exponential of the average negative log-likelihood. The only constraint on the model is that for any choice of w_{t-n+1}^{t-1}, $\sum_{i=1}^{|V|} f(i, w_{t-1}, \cdots, w_{t-n+1}) = 1$, with $f > 0$. By the product of these conditional probabilities, one obtains a model of the joint probability of sequences of words.

We decompose the function $f(w_t, \cdots, w_{t-n+1}) = \hat{P}(w_t \mid w_{t-n+1}^{t-1})$ in two parts:

1. A mapping C from any element i of V to a real vector $C(i) \in \mathbb{R}^m$. It represents the *distributed feature vectors* associated with each word in the vocabulary. In practice, C is represented by a $|V| \times m$ matrix of free parameters.
2. The probability function over words, expressed with C: a function g maps an input sequence of feature vectors for words in context, $(C(w_{t-n+1}), ..., C(w_{t-1}))$ and optionally the feature vector for the next word, $(C(w_{t-n+1}), ..., C(w_{t-1}))$ C(w_t) to a conditional probability distribution over words in V for the next word w_t. The output of g is a vector whose i-th element estimates the probability $\hat{P}(w_t = i \mid w_{t-n+1}^{t-1})$ as in Figure 6.1:

$$f(i, w_{t-1}, \cdots, w_{t-n+1}) = g(i, C(w_{t-1}), \cdots, C(w_{t-n+1})).$$

Alternatively, the energy minimization network used with some of the speeding up techniques described later (section 6.4) is based on

$$f(i, w_{t-1}, \cdots, w_{t-n+1}) = \frac{g(C(i), C(w_{t-1}), \cdots, C(w_{t-n+1}))}{\sum_j g(C(j), C(w_{t-1}), \cdots, C(w_{t-n+1}))}.$$

The function f is a composition of these two mappings (C and g), with C being *shared* across all the words in the context. With each of these two parts are associated some parameters. The parameters of the mapping C are simply the feature vectors themselves, represented by a $|V| \times m$ matrix C whose row i is the feature vector $C(i)$ for word i. The function g may be implemented by a feed-forward or recurrent neural network or another parameterized function, with parameters ω. The overall parameter set is $\theta = (C, \omega)$.

Training is achieved by looking for θ that maximizes the training corpus penalized log-likelihood:

$$L = \frac{1}{T} \sum_t \log f(w_t, w_{t-1}, \cdots, w_{t-n+1}; \theta) + R(\theta),$$

where $R(\theta)$ is a regularization term. For example, in our experiments, R is a weight decay penalty applied only to the weights of the neural network and to the C matrix, not to the biases.[2]

[2]The *biases* are the additive parameters of the neural network, such as ***b*** and ***d*** in equation 6.1 below.

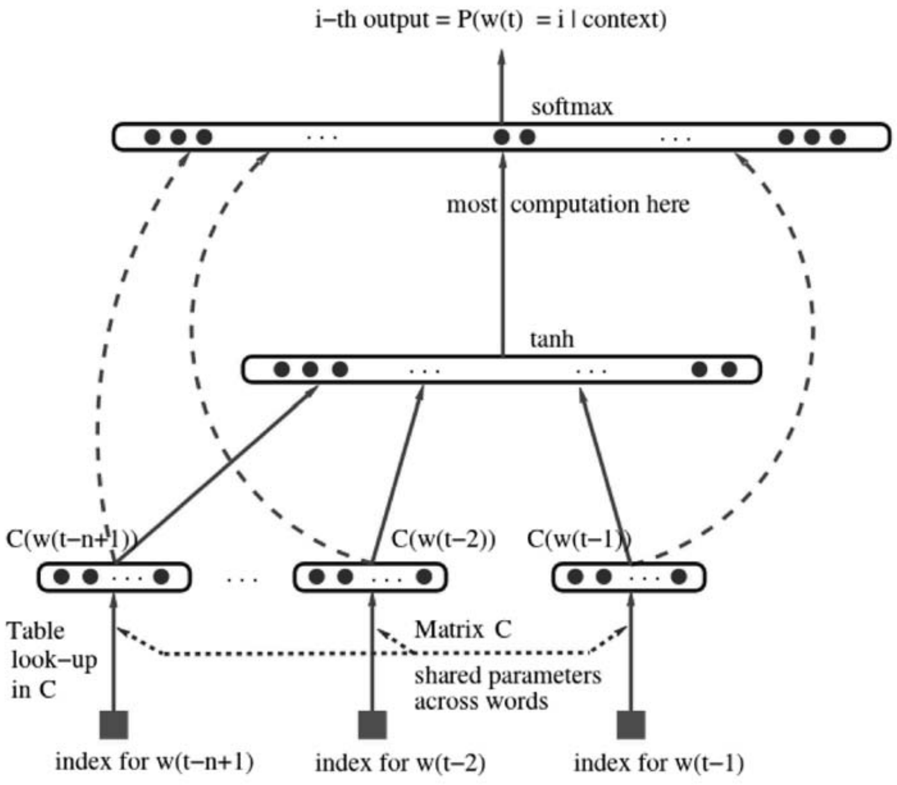


Fig 6.1 Neural architecture: $f(i, w_{t-1}, \cdots, w_{t-n+1}) = g(i, C(w_{t-1}), ..., C(w_{t-n+1}))$ where g is the neural network and $C(i)$ is the i-th word feature vector.

In the above model, the number of free parameters **only scales linearly** with $|V|$, the number of words in the vocabulary. It also **only scales linearly** with the order n: the scaling factor could be reduced to sub-linear if more sharing structure were introduced, e.g. using a time-delay neural network or a recurrent neural network (or a combination of both).

In most experiments below, the neural network has one hidden layer beyond the word features mapping, and optionally, direct connections from the word features to the output. Therefore there are really two hidden layers: the shared word features layer C, which has no non-linearity (it would not add anything useful), and the ordinary hyperbolic tangent hidden layer

More precisely, the neural network computes the following function, with a *softmax* output layer, which guarantees positive probabilities summing to 1:

$$\hat{P}(w_t \mid w_{t-1}, \cdots w_{t-n+1}) = \frac{e^{y_{w_t}}}{\sum_i e^{y_i}}.$$

The y_i are the unnormalized log-probabilities for each possible target word i, computed as follows, with parameters b, W, U, d and H:

$$y = b + Wx + U \tanh(d + Hx) \tag{6.1}$$

where the hyperbolic tangent ($\tanh$) is applied element by element, W is optionally zero (no direct connections), and x is the word features layer activation vector, which is the concatenation of the input word features from the matrix C:

$$x = (C(w_{t-1}), C(w_{t-2}), \cdots, C(w_{t-n+1})). \tag{6.2}$$

Let h be the number of hidden units, and m the number of features associated with each word. When no direct connections from word features to outputs are desired, the matrix W is set to 0. The free parameters of the model are the output biases b (with $|V|$ elements), the hidden layer biases d (with h elements), the hidden-to-output weights U (a $|V| \times h$ matrix), the word features to output weights W (a $|V| \times (n-1)m$ matrix), the hidden layer weights H (an $h \times (n-1)m$ matrix), and the word features C (a $|V| \times m$ matrix):

$$\theta = (b, d, W, U, H, C).$$

The number of free parameters is $|V|(1 + nm + h) + h(1 + (n-1)m)$. The dominating factor is $|V|(nm + h)$. Note that in theory, if there is a weight decay on the weights W and H but not on C, then W and H could converge towards zero while C would blow up. In practice we did not observe such behavior when training with stochastic gradient ascent.

Stochastic gradient ascent on the neural network consists in performing the following iterative update after presenting the t-th word of the training corpus:

$$\theta \leftarrow \theta + \varepsilon \frac{\partial \log \hat{P}(w_t \mid w_{t-1}, \cdots w_{t-n+1})}{\partial \theta},$$

where ε is the "learning rate" (chosen as large as possible while allowing the average log-likelihood of the training set to continuously increase).

Note that a large fraction of the parameters needs not be updated or visited after each example: the word features $C(j)$ of all words j that do not occur in the input window.

Mixture of models. In our experiments (see Section 6.3) we have found improved performance by combining the probability predictions of the neural network with those of an interpolated n-gram model, either with a simple fixed weight of 0.5, a learned weight (maximum likelihood on the validation set) or a set of weights that are conditional on the frequency of the context (using the same procedure that combines trigram, bigram, and unigram in the interpolated trigram, which is a mixture).

6.3 First Experimental Results

Comparative experiments were performed on the Brown corpus which is a stream of 1,181,041 words, from a large variety of English texts and books. The first 800,000 words were used for training, the following 200,000 for validation (model selection: number of neurons, weight decay, early stopping) and the remaining 181,041 for testing. The number of different words is $47,578$ (including punctuation, distinguishing between upper and lower case, and including the syntactical marks used to separate texts and paragraphs). Rare words with frequency ≤ 3 were merged into a single symbol, reducing the vocabulary size to $|V| = 16,383$.

An experiment was also run on text from the Associated Press (AP) News from 1995 and 1996. The training set is a stream of about 14 million (13,994,528) words, the validation set is a stream of about 1 million (963,138) words, and the test set is also a stream of about 1 million (963,071) words. The original data has 148,721 different words (including punctuation), which was reduced to $|V| = 17964$ by keeping only the most frequent words (and keeping punctuation), mapping upper case to lower case, mapping numeric forms to special symbols, mapping rare words to a special symbol and mapping proper nouns to another special symbol.

For training the neural networks, the initial learning rate was set to $\varepsilon_o = 10^{-3}$ (after a few trials with a tiny data set), and gradually decreased according to the following schedule: $\varepsilon_t = \frac{\varepsilon_o}{1+rt}$ where t represents the number of parameter updates done and r is a decrease factor that was heuristically chosen to be $r = 10^{-8}$.

6.3.1 Comparative Results

The first benchmark against which the neural network was compared is an interpolated or smoothed trigram model [Jelinek and Mercer, 1980]. Comparisons were also made with other state-of-the-art n-gram models: back-off n-gram models with the *Modified Kneser-Ney* algorithm [Kneser and Ney, 1995; Chen and Goodman, 1999], as well as class-based n-gram models [Brown *et al.*, 1992; Ney and Kneser, 1993; Niesler *et al.*, 1998]. The validation set was used to choose the order of the n-gram and the number of word classes for the class-based models. We used the implementation of these algorithms in the SRI Language Modeling toolkit, described by [Stolcke, 2002] and in www.speech.sri.com/projects/srilm/

In Tables 6.1 and 6.2 are measures of test set perplexity (geometric average of $1/\hat{P}(w_t \mid w_{t-n+1}^{t-1})$) for different models $\hat{P}$. Apparent convergence of the stochastic gradient ascent procedure was obtained after around 10 to 20 epochs for the Brown corpus. On the AP News corpus we were not able to see signs of over-fitting (on the validation set), possibly because we ran only 5 epochs. Early stopping on the validation set was used, but was necessary only in our Brown experiments. A weight decay penalty of 10^{-4} was used in the Brown experiments and a weight decay of 10^{-5} was used in the APNews experiments (selected by a few trials, based on validation set perplexity). Table 6.1 summarizes the results obtained on the Brown corpus. All the back-off models of the table are modified Kneser-Ney n-grams, which worked significantly better than standard back-off models. When c is specified for a back-off model in the table, a class-based n-gram is used (c is the number of word classes). Random initialization of the word features was done (similarly to initialization of neural network weights).

The **main result** is that significantly better results can be obtained when using the neural network, in comparison with the best of the n-grams, with a test perplexity difference of about 24% on Brown and about 8% on AP News, when taking the MLP versus the n-gram that worked best on the validation set. The table also suggests that the neural network was able to take advantage of more context (on Brown, going from 2 words of context to 4 words brought improvements to the neural network, not to the n-grams). It also shows that the hidden units are useful (MLP3 vs MLP1 and MLP4 vs MLP2), and that mixing the output probabilities of the neural network with the interpolated trigram always helps to reduce perplexity. The fact that simple averaging helps suggests that the neural network and the trigram make errors (i.e. low probability given to an observed word) in different places.

	ɪ	c	h	m	direct	mix	Train	Valid	test
MLP1	5		50	60	yes	no	182	284	268
MLP2	5		50	60	yes	yes		275	257
MLP3	5		0	60	yes	no	201	327	310
MLP4	5		0	60	yes	yes		286	272
MLP5	5		50	30	yes	no	209	296	279
MLP6	5		50	30	yes	yes		273	259
MLP7	3		50	30	yes	no	210	309	293
MLP8	3		50	30	yes	yes		284	270
MLP9	5		100	30	no	no	175	280	276
MLP10	5		100	30	no	yes		265	252
Del. Int.	3						31	352	336
Kneser-Ney back-off	3							334	323
Kneser-Ney back-off	4							332	321
Kneser-Ney back-off	5							332	321
Class-based back-off	3	150						348	334
Class-based back-off	3	200						354	340
Class-based back-off	3	500						326	312
Class-based back-off	3	1000						335	319
Class-based back-off	3	2000						343	326
Class-based back-off	4	500						327	312
Class-based back-off	5	500						327	312

Table 6.1 Comparative results on the Brown corpus. The deleted interpolation trigram has a test perplexity that is 33% above that of the neural network with the lowest validation perplexity. The difference is 24% in the case of the best n-gram (a class-based model with 500 word classes). ***n*** **:**order of the model. ***c:*** number of word classes in class-based n-grams. ***h:*** number of hidden units. ***m:*** number of word features for MLPs. *direct*: whether there are direct connections from word features to outputs. *mix*: whether the output probabilities of the neural network are mixed with the output of the trigram (with a weight of 0.5 on each). The last three columns give perplexity on the training, validation and test sets.

	n	h	m	direct	Mix	Valid.	Test
MLP10	6	60	100	yes	yes	104	109
Del. Int.	3					126	132
Back-off KN	3					121	127
Back-off KN	4					113	119
Back-off KN	5					112	117

Table 6.2. Comparative results on AP News corpus. See table 1 for the column labels. KN stands for Kneser-Ney.

Table 6.2 gives similar results on the larger corpus (AP News), albeit with a smaller difference in perplexity (8%). Only 5 epochs were performed because of the computational load, but much more efficient implementations are described in the next few sections. The class-based model did not appear to help the n-gram models in this case, but the high-order modified Kneser-Ney back-off model gave the best results among the n-gram models. The MLP10 is mixed with the interpolated trigram, and it gave the best results (although better results might have been obtained by mixing with the back-off KN 5 model).

6.4 Architectural Extension: Energy Minimization Network

A variant of the above neural network can be interpreted as an energy minimization model following Hinton's work on products of experts [Hinton, 2000]. In the neural network described in the previous sections the distributed word features are used only for the "input" words and not for the "target" word (next word). Furthermore, a very large number of parameters (the majority) are expanded in the output layer: the semantic or syntactic similarities between target words are not exploited. In the variant described here, the target word is also represented by its feature vector. The network takes in input a sub-sequence of words (mapped to their feature vectors) and outputs an energy function E which is low when the words form a likely sub-sequence, high when it is unlikely. For example, the network outputs an "energy" function

$$E(w_{t-n+1},\cdots,w_t) = v \cdot \tanh(d + Hx) + \sum_{i=0}^{n-1} b_{w_{t-i}} \tag{6.3}$$

where b is the vector of biases (which correspond to unconditional probabilities), d is the vector of hidden units biases, v is the output weight vector, and H is the hidden layer weight matrix, and unlike in the previous model, input and target words contribute to x:

$$x = (C(w_t), C(w_{t-1}), C(w_{t-2}), \cdots, C(w_{t-n+1})).$$

The energy function $E(w_{t-n+1},\cdots,w_t)$ can be interpreted as an unnormalized log-probability for the joint occurrence of $(w_{t-n+1},\cdots,w_t)$. To obtain a conditional probability $\hat{P}(w_t \mid w_{t-n+1}^{t-1})$ it is enough (but costly) to normalize over the possible values of w_t with a *softmax*, as follows:

$$\hat{P}(w_t \mid w_{t-1},\cdots,w_{t-n+1}) = \frac{e^{-E(w_{t-n+1},\cdots,w_t)}}{\sum_i e^{-E(w_{t-n+1},\cdots,w_{t-1},i)}}.$$

Note that the total amount of computation is comparable to the architecture presented earlier, and the number of parameters can also be matched if the v parameter takes different values for each target word w_t. Note that only b_{w_t} remains after the above softmax normalization (any linear function of the w_{t-i} for $i > 0$ is canceled by the softmax normalization). As before, the parameters of the model can be tuned by stochastic gradient ascent on $\log \hat{P}(w_t \mid w_{t-1},\cdots,w_{t-n+1})$, using similar computations.

In the products-of-experts framework, the hidden units can be seen as the experts: the joint probability of a sub-sequence $(w_{t-n+1},\cdots,w_t)$ is proportional to the exponential of a sum of terms associated with each hidden unit j, $v_j \tanh(d_j + H_j x)$. Note that because we have chosen to decompose the probability of a whole sequence in terms of conditional probabilities for each element, the computation of the gradient is tractable. This is not the case for example with products-of-HMMs [Brown and Hinton, 2000], in which the product is over experts that view the whole sequence, and which can be trained with approximate gradient algorithms such as the contrastive divergence algorithm [Brown and Hinton, 2000]. Note also that this architecture and the products-of-experts formulation can be seen as extensions of the very successful **Maximum Entropy** models [Berger *et al.,* 1996], but where the basis functions (or "features", here the hidden units activations) are learned by penalized maximum likelihood at the same time as the parameters of the features linear combination, instead of being learned in an outer loop, with greedy feature subset selection methods.

In our experiments, we did not find significant generalization improvements with this architecture, but it lends itself more naturally to the speed-up techniques described below.

6.5 Speeding-up Training by Importance Sampling

In the above energy-based model, the probability distribution of a random variable X over some set $\mathcal{X}$ is expressed as

$$P(X = x) = \frac{e^{-\mathcal{E}(x)}}{Z} \tag{6.4}$$

where $\mathcal{E}(\cdot)$ is a parameterized *energy* function which is low for plausible configurations of x, and high for improbable ones, and where $Z = \sum_{x \in X} e^{-\mathcal{E}(x)}$ is called the *partition function*. In the case that interests us, the partition function depends on the context $h_t = w_{t-n+1}^{t-1}$ because we estimate a conditional probability.

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 the parameters θ of the energy function. 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 the negative logarithm of (4) with respect to θ):

$$\nabla_\theta\left(-\log P(x)\right) = \nabla_\theta\left(\mathcal{E}(x)\right) - \sum_{x' \in \chi} P(x')\nabla_\theta\left(\mathcal{E}(x')\right). \tag{6.5}$$

Clearly, the difficulty here is to compute the negative reinforcement when $|\chi|$ 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

$$E_P\left[\nabla_\theta\left(\mathcal{E}(X)\right)\right]. \tag{6.6}$$

In [Hinton, 2002], it is proposed to estimate this average with a Gibbs sampling method, using a Markov Chain Monte-Carlo process. 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).

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

If one could sample from $P(\cdot)$, a simple way to estimate (Eq. 6.6) would consist in sampling M points $x_1, \ldots, x_M$ from the network's distribution $P(\cdot)$ and to approximate (Eq. 6.6) by the average

$$\frac{1}{M}\sum_{i=1}^{M} \nabla_\theta\left(\mathcal{E}(x_i)\right). \tag{6.7}$$

This method, known as *classical Monte-Carlo* allows estimating the gradient of the log-likelihood (Eq. 6.5). The maximum speed-up that could be achieved with such a procedure would be $|\chi|/M$. In the case of the language modeling application we are considering, that means a potential for a huge speed-up, since $|\chi|$ is typically in the tens of thousands and M could be quite small; in fact, Hinton found $M=1$ to be a good choice with the contrastive divergence method [Hinton, 2002].

However, this method requires to sample from distribution $P(\cdot)$, but it is not clear how to do this efficiently without computing $\mathcal{E}(x)$ for each $x \in \chi$.

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 (Eq. 6.6).

Classical Importance Sampling

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

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

Thus, if we take M independent samples $x_1,\dots,x_M$ from Q and apply classical Monte-Carlo to estimate $E_Q\left[\frac{P(X)}{Q(X)}\nabla_\theta(\mathcal{E}(X))\right]$, we obtain the following estimator known as *importance sampling* [Robert and Casella, 2000]:

$$\frac{1}{M}\sum_{i=1}^{M}\frac{P(x_i)}{Q(x_i)}\nabla_\theta(\mathcal{E}(x_i)). \qquad (6.8)$$

Clearly, that does not solve the problem: although we do not need to sample from P anymore, the $P(x_i)$'s still need to be computed, which cannot be done without explicitly computing the partition function. Back to square one.

Biased Importance Sampling

Fortunately, there is a way to estimate (Eq. 6.6) 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 $\frac{1}{W}w(x_i)$ to weight the $\nabla_\theta\left(\mathcal{E}(x_i)\right)$, with $w(x) = \frac{e^{-\mathcal{E}(x)}}{Q(x)}$ and $W = \sum_{j=1}^{M} w(x_j)$, thus yielding the estimator [Liu, 2001]

$$\frac{1}{W}\sum_{i=1}^{M} w(x_i)\nabla_\theta\left(\mathcal{E}(x_i)\right). \tag{6.9}$$

Though this estimator is biased, its bias decreases as M increases. It can be shown to converge to the true average (Eq. 6.6) as $M \to \infty$ [3].

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.

Adapting the Sample Size

Preliminary experiments with biased importance sampling 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 (increasing 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 biased importance sampling for the neural language model that makes use of a diagnostic, called *effective sample size* [Kong, 1992; Kong *et al.,* 1994]. For a sample $x_1,\ldots,x_M$ taken from proposal distribution Q, the effective sample size is defined by

$$ESS = \frac{\left(\sum_{j=1}^{M} w(x_j)\right)^2}{\sum_{j=1}^{M} w(x_j)^2} \tag{6.10}$$

[3]However, this does not guarantee that the variance of the estimator remains bounded. We have not dealt with the problem yet, but see [Luis and Leslie, 2000].

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 the one yielded by the biased importance sampling estimator with sample $x_1,\ldots,x_m$.

We can use this measure to diagnose whether we have sampled enough points. In order to do that, we fix a baseline sample size l. 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 than the target l. If the number of samples becomes too large, we switch back to a full back-propagation (i.e. we compute the true negative gradient).

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 fixed 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[4]. In order to do that, we propose to *redistribute the probability mass* of the sampled points in the n-gram to track P. This is achieved with n-gram tables Q_k which are estimated with the goal of matching the order k conditional probabilities of samples from our model P when the history is sampled from the empirical distribution. The tables corresponding to different values of k are interpolated in the usual way to form a predictive model, from which it is easy to sample.

Let us thus define the *adaptive* n*-gram* as follows:

$$Q(w_t \mid h_t) = \sum_{k=1}^{n} \alpha_k(h_t) Q_k(w_t \mid w_{t-k+1}^{t-1}) \qquad (6.11)$$

where $h_t = w_{t-n+1}^{t-1}$ is the context, the Q_k are the sub-models that we wish

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

to estimate, and $\alpha_k(h_t)$ is a mixture function such that $\sum_{k=1}^{n} \alpha_k(h_t) = 1$. Usually, for obvious reasons of memory constraints, the probabilities given by an n-gram are non-null only for those sequences that are observed. Mixing with lower-order models allows to give some probability mass to unseen word sequences.

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

$$Q_k(w \mid w_{t-k+1}^{t-1}) \leftarrow (1-\lambda) Q_k(w \mid w_{t-k+1}^{t-1}) + \lambda \overline{q}_k \tilde{P}(w \mid h_t) \qquad (6.12)$$

where λ is a kind of "learning rate". The parameters of functions $\alpha_k(\cdot)$ are updated so as to minimize the Kullback-Leibler divergence $\sum_{w \in W} \tilde{P}(w \mid h_t) \log \frac{\tilde{P}(w|h_t)}{Q(w|h_t)}$ by gradient descent.

We describe here the method we used to train the α_k's in the case of a bigram interpolated with a unigram, i.e. $n = 2$ above. In our experiments, the α_k's were a function of the frequency of the last word w_{t-1}.

The words were first clustered in C frequency bins $B_c, c = 1,\dots,C$. Those bins were built so as to *group words with similar frequencies in the same bin while keeping the bins balanced*[5].

Then, an "energy" value $a(c)$ was assigned for $c = 1,\dots,C$. We set $\alpha_1(h_t) = \sigma(a(h_t))$ and $\alpha_2(h_t) = 1 - \alpha_1(h_t)$ where $\sigma(z) = 1/(1+e^{-z})$ 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:

[5]By *balanced*, we mean that the sum of word frequencies does not vary a lot between two bins. That is, let $|w|$ be the frequency of word $|w|$ in the training set, then we wish that $\forall i, j, \sum_{w \in B_i} |w| \approx \sum_{w \in B_j} |w|$

$$a(h_t) \leftarrow a(h_t) - \eta\alpha_1(h_t)\alpha_2(h_t)\sum_{w\in W}\tilde{P}(w \mid h_t)\frac{Q(w \mid h_t)}{Q_2(w \mid w_{t-1}) - Q_1(w)} \tag{6.13}$$

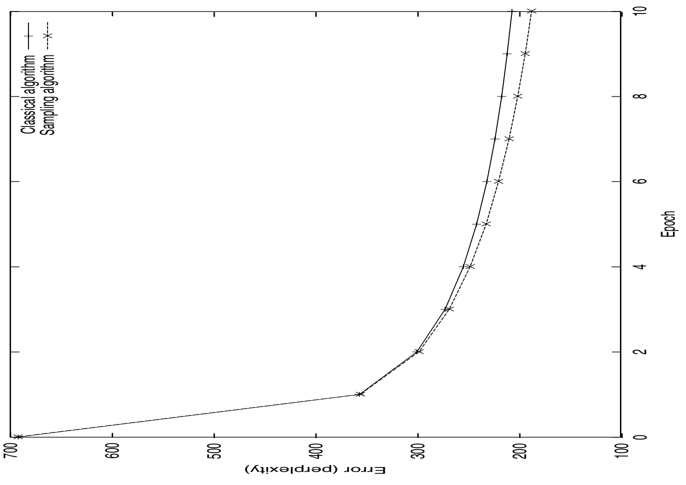
where η is a learning rate.

6.5.2 Experimental Results

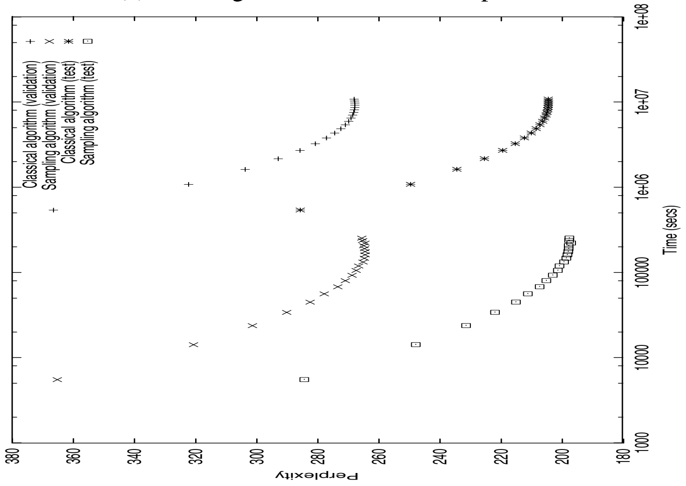
We ran some experiments on the Brown corpus, with different configurations. For these experiments the vocabulary was truncated by mapping all "rare" words (words that appear 3 times or less in the corpus) into a single special word. The resulting vocabulary contains 14,847 words. The preprocessing being a bit different, the numerical results are not directly comparable with those of Section 6.3.1.

On this dataset, a simple interpolated trigram, serving as our baseline, achieves a perplexity of 253.8 on the test set [6]. 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. Initial learning rate for the neural network was set to 3.0^{-3}, decrease constant to 10^{-8} and we used a weight decay of 10^{-4}. The output biases b_{w_t} in (3) were manually initialized so that the neural network's initial distribution is equal to the unigram. This setting is the same as that of the neural network that achieved the best results on Brown, as described in Section 6.3.1. 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. In the adaptive bigram algorithm, the parameters λ in (Eq. 6.12) and η in (Eq. 6.13) where both set to 10^{-3}. The $a(h_t)$ were initially set to $\sigma^{-1}(0.9)$ so that $\alpha_1(h_t) = 0.9$ and $\alpha_2(h_t) = 0.1$ for all h_t; this way, at the start of training, the target (neural network) and the proposal distribution are close to each other (both are close to the unigram).

[6]Better results can be achieved with a Knesser-Ney back-off trigram, but it has been shown before that a neural network converges to a lower perplexity on Brown. Furthermore, the neural network can be interpolated with the trigram for even larger perplexity reductions.



(a) Training error wrt number of epochs.



•

. (b) validation and test errors wrt CPU time

Fig 6.2 Comparison of errors between a model trained with the classical algorithm and a model trained by adaptive importance sampling

Figure 6.2(a) plots the training error at every epoch for the network trained without sampling (Section 6.4) 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 networks is similar. The same holds for validation and test errors, as is shown in Figure 6.2(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 ordinary one (trained with the exact gradient). After 9 epochs (26 hours), its perplexity over the test set is equivalent to that of the one trained with exact gradient at its overfitting point (18 epochs, 113 days). The sampling approximation thus allowed a **100-fold speed-up.**

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 with respect to the trigram baseline is 29%.

An important detail is worth mentioning here. Since $|V|$ is large, we first thought that there was too small a chance to sample the same word w twice from the proposal Q at each step to really worry about it. However, we found out the chance of picking twice the same w to be quite high in practice (with an adaptive bigram proposal distribution). We think this is due to the particular look of our proposal distribution (Eq. 6.11). The bigram part $Q_2(w \mid w_{t-1})$ of that distribution being non-null for only those words w for which $|w_{t-1}w| > 0$, there are contexts w_{t-1} in which the number of candidate words w for which $Q_2(w \mid w_{t-1}) > 0$ is small, thus there are actually good chances to pick twice the same word. Knowing this, one can save much computation time by avoiding computing the energy function $\mathcal{E}(w, h_t)$ many times for the same word w. Instead, the values of the energy functions for sampled words is kept in memory. When a word is first picked, its energy $\mathcal{E}(w, h_t)$ is computed in order to calculate the sampling weights. The value of the sampling weight is kept in memory so that, whenever the same word is picked during the same iteration, all that needs to be done is to use the copied weight, thus saving one full propagation of the energy function. **This trick increases the speed-up from a factor of 100 to a factor of 150.**

6.6 Speeding-up Probability Computation by Hierarchical Decomposition

In this section we consider a very fast variant of the neural probabilistic language model, in which not just the training algorithm but also the model itself is different. It is based on an idea that could in principle deliver close to exponential speed-up with respect to the number of words in the vocabulary. The computations required during training and during probability prediction with the regular model are a small constant plus a factor linearly proportional to the number of words $|V|$ in the vocabulary V. The approach proposed here can yield a speed-up of order $O\left(\frac{|V|}{\log|V|}\right)$ for the second term. It follows up on a proposal made in [Goodman, 2001b] to rewrite a probability function based on a partition of the set of words. The basic idea is to form a hierarchical description of a word as a sequence of $O(\log|V|)$ decisions, and to learn to take these probabilistic decisions instead of directly predicting each word's probability. Another important idea presented here is to reuse the same model (i.e. the same parameters) for all those decisions (otherwise a very large number of models would be required), using a special symbolic input that characterizes the nodes in the tree of the hierarchical decomposition. Finally, we use prior knowledge in the WordNet lexical reference system to help define the hierarchy of word classes.

6.6.1. Hierarchical Decomposition Can Provide Exponential Speed-up

In [Goodman, 2001b] it is shown how to speed-up a maximum entropy class-based statistical language model by using the following idea. Instead of computing directly $P(Y \mid X)$ (which involves normalization across all the values that Y can take), one defines a clustering partition for the Y (into the word classes C, such that there is a deterministic function from Y to C), so as to write

$$P(Y=y \mid X=x) = P(Y=y \mid C=c(y), X=x)P(C=c(y) \mid X=x).$$

This is always true for any function $c(y)$ because

$$P(Y \mid X) = \sum_c P(Y, C=c \mid X) = \sum_c P(Y \mid C=c, X)P(C=c \mid X)$$
$$= P(Y \mid C=c(Y), X)P(C=c(Y) \mid X)$$

since only one value of C is compatible with the value of Y. However, generalization could be better for choices of word classes that "make sense", i.e. make it easier to learn the $P(C=c(y) \mid X=x)$. If Y can take 10000 values and we have 100 classes with 100 words y in each class, then instead of doing normalization over 10000 choices we only need to do two normalizations, each over 100 choices. If computation of conditional probabilities is proportional to the number of choices then the above would reduce computation by a factor 50. This is approximately what is gained according to the measurements reported in [Goodman, 2001b]. The same paper suggests that one could introduce more levels to the decomposition and here we push this idea to the limit. Indeed, whereas a one-level decomposition should provide a speed-up on the order of $\frac{|V|}{\sqrt{|V|}} = \sqrt{|V|}$, a hierarchical decomposition represented by a balanced binary tree should provide an exponential speed-up, on the order of $\frac{|V|}{\log_2 |V|}$ (at least for the part of the computation that is linear in the number of choices).

Each word v must be represented by a bit vector $(k_1(v), \ldots k_m(v))$ (where m depends on v). This can be achieved by building a binary hierarchical clustering of words, and a method for doing so is presented in the next section. For example, $k_1(v) = 1$ indicates that v belongs to the top-level group 1 and $k_2(v) = 0$ indicates that it belongs to the sub-group 0 of that top-level group.

The next-word conditional probability can thus be represented and computed as follows:

$$\begin{aligned} P(v \mid w_{t-1}, \ldots) = {} & P(k_1(v) \mid w_{t-1}, \ldots) P(k_2(v) \mid k_1(v), w_{t-1}, \ldots) \\ & \ldots P(k_m(v) \mid k_1(v), \ldots k_{m-1}(v), w_{t-1}, \ldots). \end{aligned} \tag{6.14}$$

This can be interpreted as a series of binary decisions associated with nodes of a binary tree. Each node is indexed by a bit vector corresponding to the path from the root to the node (append 1 or 0 according to whether the left or right branch of a decision node is followed). Each leaf corresponds to a word. If the tree is balanced then the maximum length of the bit vector is $\lceil \log_2 |V| \rceil$. Note that we could further reduce computation by looking for an encoding that takes the frequency of words into account, to reduce the average bit length to the unconditional entropy of words. For example with the corpus used in our experiments, $|V| = 10000$ so $\log_2 |V| \approx 13.3$ while the unigram entropy is about 9.16, i.e. there is a possible speed-up of 31%. The gain would be greater for larger vocabularies, but not a very significant improvement over the major one obtained by using a simple balanced hierarchy.

The "target class" (0 or 1) for each node is obtained directly from the target word in each context, using the bit encoding of that word. Note also that there is a target (and gradient propagation) only for the nodes on the path from the root to the leaf associated with the target word. This is the major source of savings during training.

During recognition and testing, there are two main cases to consider: (1) one needs the probability of only one word, e.g. the observed word, (or very few), or (2) one needs the probabilities of all the words. In the first case (which occurs during testing on a corpus) we still obtain the exponential speed-up. In the second case, we are back to $O(|V|)$ computations (with a constant factor overhead). For the purpose of estimating generalization performance (out-of-sample log-likelihood) only the probability of the observed next word is needed. And in practical applications such as speech recognition, we are only interested in discriminating between a few alternatives, e.g. those that are consistent with the acoustics, and represented in a trellis of possible word sequences.

This speed-up should be contrasted with the one provided by the importance sampling method described above (Section 6.5), which leads to significant speed-up during training, but because the architecture is unchanged, probability computation during recognition and test still requires $O(|V|)$ computations for each prediction. Instead, the architecture proposed here gives significant speed-up both during training and test / recognition.

6.6.2 Sharing Parameters Across the Hierarchy

If a separate predictor is used for each of the nodes in the hierarchy, about $2|V|$ predictors are needed. This represents a huge capacity since each predictor maps from the context words to a single probability. This might create problems in terms of computer memory (not all the models would fit at the same time in memory) as well as overfitting. Therefore we have chosen to build a model in which parameters are shared across the hierarchy. There are clearly many ways to achieve such sharing, and alternatives to the architecture presented here should motivate further study.

Based on our discussion in the introduction, it makes sense to force the word embedding to be shared across all nodes. This is important also because the matrix of word features C is the largest component of the parameter set. Since each node in the hierarchy has a semantic meaning (being associated with a group of hopefully similar-meaning words) it makes sense to also associate each node with a feature vector. Without loss of generality, we can consider the model to predict $P(k \mid node, w_{t-1}, ..., w_{t-n+1})$, where $node$ corresponds to a sequence of bits specifying a node in the hierarchy and k is the next bit (0 or 1), corresponding to one of the two children of $node$. This can be represented by a model similar to the one described in Section 6.2, but with two kinds of symbols in input: the context words and the current node. We allow the embedding parameters for word cluster nodes to be different from those for words. Otherwise the architecture is the same, with the difference that there are only two choices to predict, instead of $|V|$ choices.

More precisely, the specific predictor used in our experiments is the following:

$$P(k=1 \mid node, w_{t-1}, ..., w_{t-n+1}) = \sigma(\gamma_{node} + \beta \cdot \tanh(d + Hx + UN_{node}))$$

where x is the concatenation of context word features as in Eq. 6.2, σ is the sigmoid function, γ_{node} is a bias parameter playing the same role as b_{w_t} in Eq. 6.3, β is a weight vector playing the same role as v in Eq. 6.3, d and H play the same role as in Eq. 6.3, and N gives feature vector embeddings for nodes, with U playing the same role as H for the node feature vector embedding.

6.6.3 Using WordNet to Build the Hierarchical Decomposition

A very important component of the whole model is the choice of the words binary encoding, i.e. of the hierarchical word clustering. In this paper we combine empirical statistics with prior knowledge from the WordNet resource [Fellbaum, 1998]. Again there are many other choices that could have been made, one extreme being a purely data-driven clustering of words.

The IS-A taxonomy in WordNet organizes semantic concepts associated with senses in a graph that is almost a tree. For our purposes we need a tree, so we have manually selected a parent for each of the few nodes that have more than one. The leaves of the WordNet taxonomy are senses and each word can be associated with more than one sense. Words sharing the same sense are considered to be synonymous (at least in one of their uses). For our purpose we have to choose one of the senses for each word (to transform the hierarchy over senses into a hierarchy over words) and we selected the most frequent sense. In addition, this WordNet tree is not binary: each node may have many more than two children (this is particularly a problem for verbs and adjectives, for which WordNet is shallow and incomplete). To transform this hierarchy into a binary tree we perform a binary hierarchical clustering of the children associated with each node, as illustrated in Figure 6.3. The K-means algorithm is used at each step to split each cluster. To compare nodes, we associate each node with the subset of words that it covers. Each word is associated with a TF/IDF [Salton and Buckley, 1988] vector of document/word occurrence counts, where each "document" is a paragraph in the training corpus. Each node is associated with the dimension-wise median of the TF/IDF scores. Each TF/IDF score is the occurrence frequency of the word in the document times the logarithm of the ratio of the total number of documents by the number of documents containing the word.

6.6.4 Comparative Results

Experiments were performed to evaluate the speed-up and any change in generalization error. The experiments also compared an alternative speed-up technique described in Section 6.5, based on importance sampling. The experiments were performed on the Brown corpus, with a reduced vocabulary size of 10,000 words (the most frequent ones). The corpus was split into 3 sets: 900,000 for training, 100,000 for validation (model selection), and 105,515 for testing. Again, the absolute perplexity numbers cannot be directly compared to those given earlier. The validation set was used to select a small number of choices for the size of the embeddings and the number of hidden units.

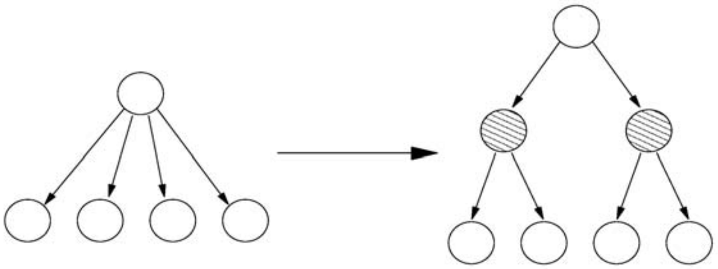


Fig 6.3 WordNet's IS-A hierarchy is not a binary tree: most nodes have many children. Binary hierarchical clustering of these children is performed.

The results in terms of raw computations (time to process one example), either during training or during test are shown respectively in Tables 6.3 and 6.4. The computations were performed on Athlon processors with a 1.2 GHz clock. The speed-up during training is by a factor greater than 250 and during test by a factor close to 200. These are impressive but less than the $|V|/\log_2|V| \approx 750$ that could be expected if there was no overhead and no constant term in the computational cost.

Architecture	Time per epoch (set)	Time per example (ms)	Speed-up
Original neural net	416 300	462.6	1
Importance sampling	6 092	6.73	68.7
Hierarchical model	1 609	1.79	258

Table 6.3 Training time per epoch (going once through all the training examples) and per example. The original neural net is as described in section 6.2 The importance sampling algorithm trains the same model faster. The hierarchical model is the one proposed here.

Architecture	Time per example (ms)	Speed-up
Original neural net	270.7	1
Importance sampling	221.3	1.22
Hierarchical model	1.4	193

Table 6.4 Test time per example for the different algorithms. See Table 6.3's caption.

It is also important to verify that learning still works and that the model generalizes well. Training is performed over about 20 to 30 epochs according to validation set perplexity (early stopping). Table 6.5 shows the comparative generalization performance of the different architectures, along with that of an interpolated 3-gram (same procedure as in [Bengio *et al.*, 2003], which follows [Jelinek and Mercer, 1980]). Note that better performance should be obtainable with some of the tricks in [Goodman, 2001a]. Combining the neural network with a trigram should also decrease its perplexity, as already shown earlier (see Section 6.3.1).

	Validation perplexity	Test Perplexity
Interpolated trigram	299.4	268.7
Original neural net	213.2	195.3
Importance sampling	209.4	192.6
Hierarchical model	241.6	220.7

Table 6.5 Test perplexity for the different architectures and for an interpolated trigram.

As shown in Table 6.5, the hierarchical model does not generalize as well as the original neural network, but the difference is not very large and still represents an improvement over the basic interpolated trigram model. Given the very large speed-up, it is certainly worth investigating variations of the hierarchical model proposed here (in particular how to define the hierarchy) for which generalization could be better. Note also that the speed-up would be greater for larger vocabularies (e.g. 50,000 is not uncommon in speech recognition systems).

6.7 Short Lists and Speech Recognition Applications

The standard measure of the prediction capability of language models is perplexity, as used in our experiments described above. Often language models are however part of a larger system and improvements in perplexity do not necessarily lead to better performance of the overall system. An important application domain of language models is continuous speech recognition and many new language modeling techniques have been in fact first introduced in the context of this domain.

In continuous speech recognition we are faced with the problem to find the word sequence w^* corresponding to a given acoustic signal x. Using Bayes' rule,

$$\begin{aligned} w^* &= \operatorname*{argmax}_w \hat{P}(w \mid x) \\ &= \operatorname*{argmax}_w \frac{f(x \mid w)\hat{P}(w)}{P(x)} \\ &= \operatorname*{argmax}_w f(x \mid w)\hat{P}(w) \end{aligned} \tag{6.15}$$

where $f(x \mid w)$ is the so called acoustic model and $\hat{P}(w)$ the language model. The problem of actually finding w^* that maximizes the equation is known as the decoding problem. To the best of our knowledge, all state-of-the-art large vocabulary continuous speech recognizers (LVCSR) used hidden Markov models (with various methods for parameter sharing and adaptation) for acoustic modeling and n-gram back-off language models to estimate $\hat{P}(w)$.

In the following we study the application of the neural network language model to conversational telephone speech recognition in a

LVCSR. Recognition of conversational speech is a very challenging tasks with respect to acoustic modeling (bad signal quality, highly varying pronunciations, ...) as well as language modeling (unconstrained speaking style, frequent grammatical errors, hesitations, start-overs, ..). This is illustrated by the following examples that have been extracted from the training material:

- *But yeah so that is that is what I do.*
- *I I have to say I get time and I get a few other.*
- *Possibly I don't know I mean I had one years ago so.*
- *Yeah yeah well it's it's.*

In addition, language modeling for conversational speech suffers from a lack of adequate training data since the main source is audio transcriptions, in contrast to the broadcast news task for which other news sources are readily available. Unfortunately, collecting large amounts of conversational data and producing detailed transcriptions is very costly. One possibility is to increase the amount of training data by selecting conversational-like sentences in broadcast news material and on the Internet, or by transforming other sources to be more conversational-like. Here we use the neural network language model in order to see if it can take better advantage from the limited amount of training data. Initial work has in fact shown that the neural network language model (LM) can be used to reduce the word error rate in a speech recognizer [Schwenk and Gauvain, 2002]. These results were later confirmed with an improved speech recognizer [Gauvain *et al.,* 2002; Schwenk and Gauvain, 2003]. A neural network LM has also been used with a Syntactical Language Model showing perplexity and word error improvements on the Wall Street Journal corpus [Emami *et al.,* 2003].

In our previous work the neural network LM was trained on about 6M words and the word error rates were in the range of 25%. Since then, large amounts of conversational data have been collected in the DARPA EARS program and quick transcriptions were made available (total of about 2200h, 27.2M words), helping to build better speech recognizers. This amount of LM training data enables better training of the back-off 4-gram LM and one may wonder if the neural network LM, a method developed for probability estimation of sparse data, still achieves an improvement. In the following sections we show that the neural network LM continues to achieve consistent word error reductions with respect to a carefully tuned back-off 4-gram LM. Detailed results are provided as a function of the size of the LM training data. The described systems have successfully participated in speech recognition evaluations organized by the

National Institute of Standards and Technology NIST and DARPA [Lee *et al.,* 2003; Fiscus *et al.,* 2004]. All results reported in the following sections use the official test sets of these evaluations.

Incorporating this approach into a heavily tuned LVCSR needs however several modifications of the basic architecture described above. Current decoding algorithms suppose in particular that a request of a LM probability takes basically no time[7] and many thousand LM probabilities may be requested to decode a sentence of several seconds. Fast training is also important for system development. The necessary modifications of the neural network language model are described in the following sections. We first present another approach for fast training of the neural networks and efficient incorporation into a speech recognition system [Schwenk, 2004]. We then summarize the principles of the reference speech recognizer.

6.7.1 Fast Recognition

Language models play an important role during decoding of continuous speech since the information provided about the most probable set of words given the current context is used to limit the search space. Using the neural LM directly during decoding imposes an important burden on search space organization since a context of three words must be kept. This led to long decoding times in our first experiments when the neural LM was used directly during decoding [Schwenk and Gauvain, 2002]. In order to make the model tractable for LVCSR the following techniques have been applied (more details below):

1. **Lattice rescoring**: decoding is done with a standard back-off LM and a lattice is generated. The neural network LM is then used to rescore the lattice, i.e. the LM probabilities are changed.
2. **Shortlists**: the neural network is only used to predict the LM probabilities of a subset of the whole vocabulary.
3. **Regrouping**: all LM probability requests in one lattice are collected and sorted. By these means all LM probability requests with the same context h_t lead to only one forward pass through the neural network.
4. **Block mode**: several examples are propagated at once through the neural network, allowing the use of faster matrix/matrix operations.

[7]For a back-off LM this is a table look-up using hashing techniques.

5. **CPU optimization**: machine specific BLAS libraries are used for fast matrix and vector operations.

Normally, the output of a speech recognition system is the most likely word sequence given the acoustic signal (see Eq. 6.15), but it is often advantageous to preserve more information for subsequent processing steps. This is usually done by generating a lattice which is a graph of possible solutions where each edge corresponds to a hypothesized word with its acoustic and language model scores. These graphs are called lattices (see Figure 6.4 for an example). In the context of this work, the lattice is used to replace the LM probabilities by those calculated by the neural network LM (*lattice rescoring*).

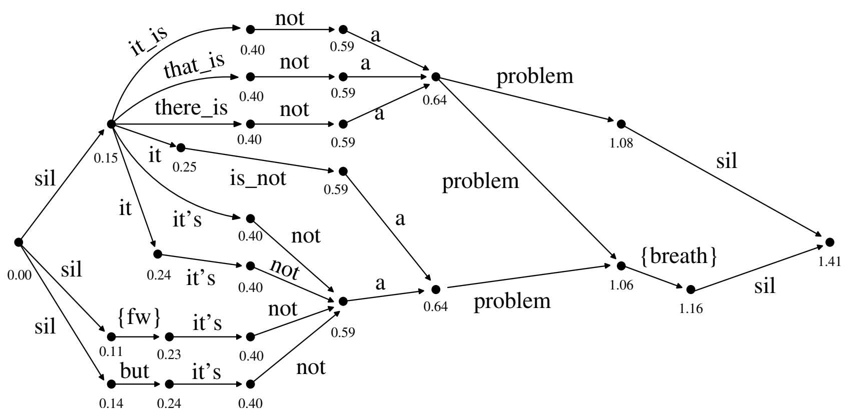


Fig 6.4 Example of a lattice produced by the speech recognizer using a 3-gram language model. A lattice is a graph of possible solutions where each edge corresponds to a hypothesized word with its acoustic and language model scores (for clarity these scores are not shown in the figure). {*fw*} stands for a filler word and {*breath*} is breath noise.

It has been demonstrated in Section 6.2 that most calculations are due to the large size of the output layer. Remember that all outputs need to be calculated in order to perform the softmax normalization even though only one LM probability is needed. Experiments using lattice rescoring with normalized LM scores led to much higher word error rates. One may argue that it is not very reasonable to spend a lot of time to get the LM probabilities of words that do not appear very often. Therefore, we chose to limit the output of the neural network to the s most frequent words, $s = |V|$, referred to as a *shortlist* in the following discussion. All words in V are still considered for the input of the neural network. The LM probabilities of words in the shortlist ($\hat{P}_N$) are calculated by the neural network and the LM probabilities of the remaining words ($\hat{P}_B$) are obtained from a standard 4-gram back-off LM:

$$\hat{P}(w_t \mid h_t) = \begin{cases} \hat{P}_N(w_t \mid h_t) \cdot P_S(h_t) & \text{if } w_t \in \text{shortlist} \\ \hat{P}_B(w_t \mid h_t) & \text{else} \end{cases}$$

$$P_S(h_t) = \sum_{w \in shortlist(h_t)} \hat{P}_B(w \mid h_t)$$

It can be considered that the neural network redistributes the probability mass of all the words in the shortlist[8]. This probability mass is precalculated and stored in the data structures of the standard 4-gram LM. A back-off technique is used if the probability mass for a requested input context is not directly available. Table 6.6 gives the coverage, i.e. the percentage of LM probabilities that are effectively calculated by the neural network when evaluating the perplexity on a development set of 56k words ot when rescoring lattices.

Shortlist size	1024	2000	4096	8192
Development set	89.3%	93.6%	96.8%	98.5%
Lattice rescoring	88.5%	89.9%	90.4%	91.0%

Table 6.6 Coverage for different shortlist sizes, i.e. percentage of 4-grams that are actually calculated by the neural LM. The vocabulary size is about 51k.

During lattice rescoring LM probabilities with the same context h_t are often requested several times on potentially different nodes in the lattice (for instance the trigram "*not a problem*" in the lattice shown in Figure 6.4). Collecting and regrouping all these calls prevents multiple forward passes since all LM predictions for the same context are immediately available (see Figure 6.1). Further improvements can be obtained by propagating several examples at once through the network, which is also known as bunch mode [Bilmes *et al.,* 1997]. In comparison to equation 1, this results in using matrix/matrix instead of matrix/vector operations:

[8]Note that the sum of the probabilities of the words in the shortlist for a given context is normalized $\sum_{w \in shortlist} \hat{P}_N(w \mid h_t) = 1$ ·

$$Z = \tanh\left(HX + D\right)$$
$$Y = UZ + B$$

where D and B are obtained by duplicating the bias d and b respectively for each line of the matrix. The tanh-function is performed element-wise. These matrix/matrix operations can be aggressively optimized on current CPU architectures, e.g. using SSE2 instructions for Intel processors [Intel's MKL, 2004; ATLAS, 2004]. Although the number of floating point operations to be performed is strictly identical to single example mode, an up to five times faster execution can be observed depending on the sizes of the matrices.

The test set of the 2001 NIST speech recognition evaluation consists in 6h of speech comprised of 5895 conversations sides. The lattices generated by the speech recognizer for this test set contain on average 511 nodes and 1481 arcs per conversation side. In total 3.8 million 4-gram LM probabilities were requested out of which 3.4 million (89.9%) have been processed by the neural network, i.e. the word to be predicted is among the 2000 most frequent words (the shortlist). After collecting and regrouping all LM calls in each lattice, only 1 million forward passes though the neural network have been performed, giving a cache hit rate of about 70%. Using a bunch size of 128 examples, the total processing time took less than 9 minutes on a Intel Xeon 2.8GHz processor, i.e. in 0.03 times real time (xRT).[9] This corresponds to about 1.7 billion floating point operations per second (1.7 GFlops). Lattice rescoring without bunch mode and regrouping of all calls in one lattice is about ten times slower.

6.7.2 Fast Training

Language models for LVCSR are usually trained on text corpora of several million words. With a vocabulary size of 51k words, standard back-propagation training would take a very long time. In addition to the speed-up techniques of importance sampling and hierarchical decomposition described in the previous sections, a third method has been developed. Optimized floating point operations are much more efficient if they are applied to data that is stored in contiguous locations in memory, making a better use of cache and data prefetch capabilities of processors. This is

[9]In speech recognition, processing time is measured in multiples of the length of the speech signal, the real time factor xRT. For a speech signal of 2h, a processing time of 2xRT corresponds to 4h of calculation.

more difficult to obtain for resampling techniques. Therefore, a fixed size output layer was used and the words in the shortlist were rearranged in order to occupy contiguous locations in memory.

In our initial implementation we used standard stochastic backpropagation and double precision for the floating point operations in order to ensure good convergence. Despite careful coding and optimized BLAS libraries [Intel's MKL, 2004; ATLAS, 2004] for the matrix/vector operations, one epoch through a training corpus of 12.4M examples took about 47 hours on a Pentium Xeon 2.8 GHz processor. This time was reduced by a factor of more than 30 using the following techniques [Schwenk, 2004]:

- **Floating point precision** (1.5 times faster). Only a slight decrease in performance was observed due to the lower precision.
- **Suppression of intermediate calculations** when updating the weights (1.3 times faster).
- **Bunch mode**: forward and back-propagation of several examples at once (up to 10 times faster).
- **Multi-processing**: use of SMP-capable BLAS libraries for off-the-shelf bi-processor machines (1.5 times faster).

Most of the improvement was obtained by using bunch mode in the forward and backward pass. After calculating the derivatives of the error function ΔB at the output layer, the following equations were used (similar to [Bilmes *et al.,* 1997]):

$$b = b - \lambda \Delta B i \tag{6.16}$$

$$\Delta D = U^T \Delta B \tag{6.17}$$

$$U = -\lambda \Delta B Z^T + \alpha U \tag{6.18}$$

$$\Delta D = \Delta D \odot (1 - Z \odot Z) \tag{6.19}$$

$$d = d - \lambda \Delta D i \tag{6.20}$$

$$\Delta X = H^T \Delta D \tag{6.21}$$

$$H = -\lambda \Delta D X^T + \alpha H \tag{6.22}$$

where $i = (1, 1, \ldots 1)^T$, with dimension of the bunch size. The symbol $\odot$ denotes element-wise multiplication. Note that the backpropagation and weight update step, including weight decay, is done in one operation using the GEMM function of the BLAS library (Eq. 6.18 and 6.22). For this, the weight decay factor ε is incorporated into $\alpha = 1 - \lambda\varepsilon$. The update step of the projection matrix is not shown, for clarity.

Table 6.7 summarizes the effect of the different techniques to speed up training. Extensive experiments were first done with a training corpus of 1.1M

Size of Training data	**Double Prec.**	**Float Prec.**	**Bunch mode 2**	**4**	**8**	**16**	**32**	**128**	**SMP 128**
1.1M words	2h	1h16	37m	31m	24m	14m	11m	8m18	5m50
12.4M words	47h	30h	10h12	8h18	6h51	4h01	2h51	2h09	1h27

Table 6.7 Training times (for one epoch) reflecting the different improvements (on a Intel Pentium CPU at 2.8 GHz)

words and then applied to a larger corpus of 12.4M words. Bilmes et al. reported that the number of epochs needed to achieve the same MSE increases with the bunch size [Bilmes *et al.,* 1997]. In our experiments the convergence behavior also changed with the bunch size, but after adapting the learning parameters of the neural network only small losses in perplexity were observed, and there was no impact on the word error when the neural LM was used in lattice rescoring.

6.7.3 Regrouping of training examples

The above described bunch mode optimization is generic and can be applied to any neural network learning problem, although it is most useful for large tasks like this one. In addition we propose new techniques that rely on the particular characteristics of the language model training corpus. A straightforward implementation of stochastic backpropagation is to cycle through the training corpus, in random order, and to perform a forward/backward pass and weight update for each 4-gram. However, in large texts it is frequent to encounter some 4-grams several times. This means that identical examples are trained several times. This is different from other pattern recognition tasks, for instance optical character recognition, for which it is unlikely to encounter twice the exact same example since the inputs are usually floating point numbers. In addition, for the LM task, we will find many occurrences of the same context in the training corpus for which several different target words should be predicted. In other words, we can say that for each trigram there are usually several corresponding 4-grams. This fact can be used to substantially decrease the number of operations. The idea is to regroup these examples and to perform only one forward and backward pass though the neural network. The only difference is that there are now multiple output targets for each input context. Furthermore, 4-grams appearing multiple times can be learned at once by multiplying the corresponding gradients by the number of occurrences. This is equivalent to using bunch mode where each bunch includes all examples with the same trigram context. Alternatively, the cross-entropy targets can also be set to the empirical posterior probabilities (rather than 0 or 1), i.e. the relative frequencies of the 4-grams with a common context.

Words in corpus	**229k**	**1.1M**	**12.4M**
Random presentation:			
# 4-grams	162k	927k	11.0M
Training time	144s	11m	171m
Regrouping:			
# distinct 3-grams	124k	507k	3.3M
Training time	106s	6m35s	58m

Table 6.8 Training times for stochastic backpropagation using random presentation of all 4-grams in comparison to regrouping all 4-grams that have a common trigram context (bunch size=32).

In stochastic backpropagation with random presentation order the number of forward and backward passes corresponds to the total number of 4-grams in the training corpus which is roughly equal to the number of words[10]. With the new algorithm the number of forward and backward passes is equal to the number of *distinct trigrams* in the training corpora. One major advantage of this approach is that the expected gain, i.e. the relation between the total number of 4-grams and distinct trigrams, increases with the size of the training corpus. Although this approach is particularly interesting for large training corpora (see Table 6.8), we were not able to achieve the same convergence as with random presentation of individual 4-grams. Overall the perplexities obtained with the regrouping algorithm were slightly higher.

6.7.4 Baseline Speech Recognizer

The above described neural network language model is evaluated in a state-of-the-art speech recognizer for conversational telephone speech

[10]The sentence were surrounded by begin and end of sentence markers. The first two words of a sentence do not form a full 4-gram, i.e. a sentence of 3 words has only two 4-grams.

[Gauvain *et al.,* 2003]. This task is known to be significantly more difficult than the recognition of Wall Street journal or of broadcast news data. Based on the NIST speech recognition benchmarks [Lee *et al.,* 2003; Fiscus *et al.,* 2004], current best broadcast news transcription systems achieve word error rates that approach 10% in 10xRT (10 times real time) while the word error rate for the DARPA conversational telephone speech recognition task is about 20% using more computational resources (20xRT). A large amount of this difference can of course be attributed to the difficulties in acoustic modeling, but language modeling of conversational speech also faces problems that are much less important in broadcast news data such as unconstrained speaking style, hesitations, etc.

The word recognizer uses continuous density hidden Markov models (HMM) with Gaussian mixtures for acoustic modeling and n-gram statistics estimated on large text corpora for language modeling. Each context-dependent phone model is a tied-state left-to-right context dependant HMM with Gaussian mixture observation densities where the tied states are obtained by means of a decision tree. The acoustic feature vector has 39 components including 12 cepstrum coefficients and the log energy, along with their first and second derivatives. Cepstral mean and variance normalization are carried out on each conversation side. The acoustic models are trained using the maximum mutual information criterion on up to 2200 hours of data.

Decoding is carried out in three passes. In the first pass the speaker gender for each conversation side is identified using Gaussian mixture models, and a fast trigram decode is performed to generate approximate transcriptions. These transcriptions are used to compute the vocal tract length normalization (VTLN) warp factors for each conversation side and to adapt the speaker adaptive (SAT) models that are used in the second pass. Passes 2 and 3 make use of the VTLN-warped data to generate a trigram lattice per speaker turn which is expanded with the 4-gram baseline back-off LM and converted into a confusion network with posterior probabilities. The best hypothesis in the confusion network is used in the next decoding pass for unsupervised maximum likelihood linear regression (MLLR) adaptation of the acoustic models (constraint and unconstrained). The third pass is similar to the second one but more phonemic regression classes are used and the search space is limited to the word graph obtained in the second pass. The overall run time is about 19xRT.

6.7.5 Language model training

The main source for training a language model for conversational speech are the transcriptions of the audio training corpora. Three different *in-domain* data corpora have been used:

- **7.2M words:** Our first experiments were carried out with the initial release of transcriptions of acoustic training data for the Switchboard (SWB) task, namely the *careful* transcriptions of the SWB corpus distributed by LDC (2.7M words) and by the Mississippi State University (ISIP) (2.9M words), the Callhome corpus (217k words), some SWB cellular data (230k words) and *fast* transcriptions of a previously unused part of the SWB2 corpus (80h, 1.1M word).
- **12.3M words:** In 2003 LDC changed the data collection paradigm of conversational data to the so called "Fisher-protocol". Fast transcriptions of 520h of such data were available for this work.
- **27.2M words:** In a second release, additional fast transcriptions have been made available, doubling the total amount of language model training data.

All this data is available from the linguistic data consortium (LDC).[11] In addition to these in-domain corpora the following texts have been used to train the 4-gram back-off LM:

- 240M words of commercially produced broadcast news transcripts,
- 80M words of CNN television broadcast news transcriptions,
- up to 500M words of conversational like data that was collected from the Internet[12].

We refer to these data as the broadcast news corpus. Adding other sources, in particular newspaper text, did not turn out to be useful. The LM vocabulary contains 51k words and the fraction of test words not in the training set is 0.2%. The baseline LM is constructed as follows. Separate back-off n-gram language models are estimated on all the above corpora using the modified version of Kneser-Ney smoothing as implemented in

[11]http://www.ldc.upenn.edu/

[12]This data has been provided by the University of Washington.

the SRI LM toolkit [Stolcke, 2002]. A single back-off LM was built by merging these models, estimating the interpolation coefficients with an EM procedure. Table 6.9 gives some statistics about the reference language models.

In-domain data [words] (+broadcast news corpus)	7.2M	12.3M	27.2M
Number of 2-grams	20.1M	20.1M	26.1M
3-grams	38.8M	31.5M	40.3M
4-grams	24.0M	24.3M	39.4M
Perplexity on Eval03 test	53.0	51.5	47.5

Table 6.9 Statistics for the back-off 4-gram reference LM built using in-domain training data of varying sizes and more than 500M words of broadcast news data.

6.7.6 Experimental results

The neural network LM was trained only on the in-domain corpora (7 M, 12M and 27M words respectively). Two experiments have been conducted:

1. The neural network LM is interpolated with a back-off LM that was trained only on the in-domain copora and compared to this LM,
2. The neural network LM is interpolated with the full back-off LM (in-domain and broadcast news data) and compared to this full LM.

The first experiment allows us to assess the real benefit of the neural LM since the two smoothing approaches (back-off and neural network) are compared on the same data. In the second experiment all the available data are used to obtain the overall best results. The perplexities of the neural network and the back-off LM are given in Table 6.10.

A perplexity reduction of about 9% relative is obtained independently of the size of the LM training data. This gain decreases to approximately 6% after interpolation with the back-off LM trained on the additional broadcast news corpus of out-of domain data. It can been seen that the perplexity of the neural network LM trained only on the in-domain data is better than that of the back-off reference LM trained on all data (45.5 with respect to 47.5). Despite these rather small gains in perplexity, consistent word error reductions were observed (see Figure 6.4). The first system is that described in [Gauvain *et al.,* 2003].

In-domain data [words]	7.2M	12.3M	27.2M
In-domain data only:			
Back-off LM	62.4	55.9	50.1
Neural LM	57.0	50.6	45.5
Interpolated with all data:			
Back-off LM	53.0	51.1	47.5
Neural LM	50.8	48.0	44.2

Table 6.10 Eval03 test set perplexities for the back-off and neural LM as a function of the size of the in-domain training data.

The second system has a much lower word error rate than the first one due to several improvements of the acoustics models, and the availability of more acoustic training data. The third system differs from the second one again by the amount of training data used for the acoustic and the language model.

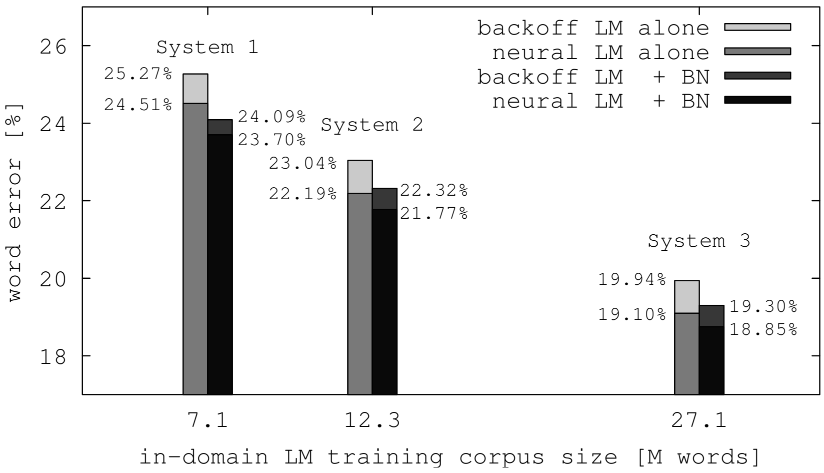


Fig 6.5 Word error rates on the Eval03 test set for the back-off LM and the neural network LM, trained only on in-domain data (left bars for each system) and interpolated with the broadcast news LM (right bars for each system).

Although the size of the LM training data has almost quadrupled from 7.2M to 27.2M words, a consistent absolute word error reduction of 0.55% can be observed [Schwenk and Gauvain, 2004]. In all these experiments, it seems that the word error reductions brought by the neural network LM are independent of the other improvements, in particular those obtained by better acoustic modeling and by adding more language model training data. When only in-domain data is used (left bars for each system in Figure 6.4) the neural network LM achieves an absolute word error reduction of about 0.8%. Note also that the neural network LM trained on at least 12.3M words is better than the back-off LM that uses in addition more than 500M words of the broadcast news corpus (system 2: 22.19 with respect to 22.32%, and system 3: 19.10 with respect to 19.30%). In a contrastive experiment, back-off and neural LMs were trained on a random 400k word subset of the 27.2M corpus, simulating for instance a domain specific task with very little LM training data. In this case, the neural network LM decreased the perplexity from 84.9 to 75.6 and the word error rate from 25.60% to 24.67%.

6.7.7 Interpolation and ensembles

When building back-off LM for a collection of training corpora, better results are usually obtained by first building separate language models for each corpus and then merging them together using interpolation coefficients obtained by optimizing the perplexity on some development data [Jelinek and Mercer, 2000]. This procedure has been used for all our back-off language models. It is, however, not so straightforward to apply this technique to a neural network LM since individually trained language models can not be merged together, and they need to be interpolated on the fly. In addition, it may be sub-optimal to train the neural network language models on subparts of the training corpus since the continuous word representations are not learned on all data. For this reason, in the above described experiments only one neural network was trained on all data.

Training large neural networks (2M parameters) on a lot of data (27.2M examples) with stochastic backpropagation may pose convergence problems and we believe that the neural network underfits the training data. Unfortunately, more sophisticated techniques than stochastic gradient descent are not tractable for problems of this size. On the other hand, several well known ensemble learning algorithms used in the machine learning community can be applied to neural networks, in particular Bagging and AdaBoost. Following to the bias/variance decomposition of error, Bagging improves performance by minimizing the variance of the classifiers [Breiman, 1994], while AdaBoost sequentially constructs classifiers that try to correct the errors of the previous ones [Freund, 1995]. In this work we have evaluated another variance reduction method: several networks are trained on all the data, but after each epoch the training data is randomly shuffled.

# hidden units	1024	1280	1560	1600	2000
One network	22.19	22.22	22.18	-	-
Ensembles	2x500	3x400	3x500	4x400	4x500
	22.15	22.12	22.07	22.09	22.09

Table 6.11 Word error rates for one large neural network and ensembles, trained on the 12.4M word in-domain corpus only.

As can been seen in Table 6.11 increasing the number of parameters of one large neural network does not lead to improvements for more than 1000 hidden units, but using ensembles of several neural networks with the same total number of parameters results in a slight decrease in the word error rate. As a side effect, training is also faster since the individual smaller networks can be trained in parallel on several machines, without any communication overhead.

6.7.8 Evaluation on other data

The neural network language model was also evaluated on other languages and tasks, in particular English and French broadcast news (BN) and English and Spanish parliament speeches (PS) [Schwenk and Gauvain, 2005]. In all cases it achieved significant word error reductions with respect to the reference back-off language model (see Table 6.12) although it was trained on less data. For each task the amount of available language model training data, the word error rate and the processing time are given.

Task	**Back-off LM**			**Neural network LM**		
	LM data	**Werr**	**Runtime**	**LM data**	**Werr**	**Addtl. Runtime**
French BN	523M	10.7%	8xRT	22M	10.4%	0.30xRT
English PS	325M	11.5%	10xRT	34M	10.8%	0.10xRT
Spanish PS	36M	10.7%	10xRT	34M	10.0%	0.07xRt

Table 6.12 Evaluation of the neural network language model in a speech recognizer for broadcast news (BN) and Parliament speeches (PS).

6.8 Conclusions and Future Work

This chapter has introduced an approach to learn a distributed representation for language models using artificial neural networks, a number of techniques to speed up training and testing of the model, and successful applications of the model, notably for speech recognition.

The basic idea behind this model is that of learning an embedding for words that allows to generalize to new sentences composed of words that have some similarity to words in training sentences. The embedding is continuous and the model learns a smooth map from the embeddings of the words in the sentence to the conditional probability of the last word. Hence when a word is replaced by a similar one, a similar conditional probability function is computed. Instead of distributing probability mass according to exact match of subsequences, as in n-gram models, a more subtle notion of similarity in the space of word sequences is learnt and applied.

One of the difficulty with this approach, however, compared to n-gram models, is that it requires a large number of multiply-adds for each prediction, whereas n-gram models only require a few table look-ups. A good deal of the chapter therefore explores different methods to speed-up the algorithms. The computational bottleneck involves the loop or the sum over the set of words over which the conditional probability is to be computed. The three different methods all aim at reducing this computation, either by sampling, by hierar-chical decomposition of the probabilities, or by focusing on the most frequent words. Speed-up on the order of 100-fold are thus obtained and a large-scale application of the approach on a state-of-the-art speech recognition system shows that it can be used to improve on the state-of-the-art performance.

Acknowledgments

The authors would like to thank Pascal Vincent, Réjean Ducharme and Christian Jauvin for their collaboration on the neural language model, and Léon Bottou, Yann Le Cun and Geoffrey Hinton for useful discussions. The authors would also like to recognize the contributions of Lori Lamel, Gilles Adda and Fabrice Lefèvre for their involvement in the development of the LIMSI conversational speech recognition system from which part of this work is based. Part of this research was made possible by funding to Yoshua Bengio from the NSERC granting agency, as well as the MITACS and IRIS networks.

References

Automatically tuned linear algebra software, https://sourceforge.net/projects/math-atlas/atlas

Baker, D. and McCallum, A. (1998). Distributional clustering of words for text classification. In SIGIR'98.

Bellegarda, J. (1997). A latent semantic analysis framework for large–span language modeling. In Proceedings of Eurospeech 97, pages 1451–1454, Rhodes, Greece.

Bengio, S. and Bengio, Y. (2000a). Taking on the curse of dimensionality in joint distributions using neural networks. IEEE Transactions on Neural Networks, special issue on Data Mining and Knowledge Discovery, 11(3), 550–557.

Bengio, Y. and Bengio, S. (2000b). Modeling high-dimensional discrete data with multi-layer neural networks. In S. Solla, T. Leen, and K.-R. Muller, editors, Advances in Neural Information Processing Systems 12, pages 400–406. MIT Press.

Bengio, Y. and Senecal, 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. 38 Authors Suppressed Due to Excessive Length

Bengio, Y., Ducharme, R., Vincent, P., and Jauvin, C. (2003). A neural probabilistic language model. Journal of Machine Learning Research, 3, 1137–1155.

Berger, A., Della Pietra, S., and Della Pietra, V. (1996). A maximum entropy approach to natural language processing. Computational Linguistics, 22, 39–71.

Bilmes, J., Asanovic, K., Chin, C.-W., and Demmel, J. (1997). Using phipac to speed error back-propagation learning. In International Conference on Acoustics, Speech, and Signal Processing, pages V: 4153–4156.

Blitzer, J., K.Q.Weinberger, Saul, L., and Pereira, F. (2005). Hierarchical distributed representations for statistical language models. In L. Saul, Y. Weiss, and L. Bottou, editors, Advances in Neural Information Processing Systems 17. MIT Press.

Breiman, L. (1994). Bagging predictors. Machine Learning, 24(2), 123–140.

Brown, A. and Hinton, G. (2000). Products of hidden markov models. Technical Report GCNU TR 2000-004, Gatsby Unit, University College London.

Brown, P., Pietra, V. D., DeSouza, P., Lai, J., and Mercer, R. (1992). Class-based n-gram models of natural language. Computational Linguistics, 18, 467–479.

Chen, S. F. and Goodman, J. T. (1999). An empirical study of smoothing techniques for language modeling. Computer, Speech and Language, 13(4), 359–393.

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.

Deerwester, S., Dumais, S., Furnas, G., Landauer, T., and Harshman, R. (1990). Indexing by latent semantic analysis. Journal of the American Society for Information Science, 41(6), 391–407.

Elman, J. (1990). Finding structure in time. Cognitive Science, 14, 179–211.

Emami, A., Xu, P., and Jelinek, F. (2003). Using a connectionist model in a syntactical based language model. In International Conference on Acoustics, Speech, and Signal Processing, pages I: 272–375.

Fellbaum, C. (1998). WordNet: An Electronic Lexical Database. MIT Press.

Fiscus, J., Garofolo, J., Lee, A., Martin, A., Pallett, D., Przybocki, M., and Sanders, G. (Nov 2004). Results of the fall 2004 STT and MDE evaluation. In DARPA Rich Transcription Workshop, Palisades NY.

Freund, Y. (1995). Boosting a weak learning algorithm by majority. Information and Computation, 121(2), 256–285.

Gauvain, J.-L., Lamel, L., Schwenk, H., Adda, G., Chen, L., and Lefevre, F. (2003). Conversational telephone speech recognition. In International Conference on Acoustics, Speech, and Signal Processing, pages I: 212–215.

Goodman, J. (2001a). A bit of progress in language modeling. Technical Report MSR-TR-2001-72, Microsoft Research.

Goodman, J. (2001b). Classes for fast maximum entropy training. In International Conference on Acoustics, Speech, and Signal Processing, Utah.

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

Hinton, G. (2000). Training products of experts by minimizing contrastive divergence. Technical Report GCNU TR 2000-004, Gatsby Unit, University College London.

Hinton, G. (2002). Training products of experts by minimizing contrastive divergence. Neural Computation, 14(8), 1771–1800. 1 Neural Probabilistic Language Models 39

Hinton, G. and Roweis, S. (2003). Stochastic neighbor embedding. In S. Becker, S. Thrun, and K. Obermayer, editors, Advances in Neural Information Processing Systems 15. MIT Press, Cambridge, MA.

Jelinek, F. and Mercer, R. (2000). Interpolated estimation of markov source parameters from sparse data. Pattern Recognition in Practice, pages 381–397.

Jelinek, F. and Mercer, R. L. (1980). Interpolated estimation of Markov source parameters from sparse data. In E. S. Gelsema and L. N. Kanal, editors, Pattern Recognition in Practice. North-Holland, Amsterdam.

Jensen, K. and Riis, S. (2000). Self-organizing letter code-book for text-to-phoneme neural network model. In Proceedings ICSLP.

Katz, S. M. (1987). Estimation of probabilities from sparse data for the language model component of a speech recognizer. IEEE Transactions on Acoustics, Speech, and Signal Processing, ASSP-35(3), 400–401.

Kneser, R. and Ney, H. (1995). Improved backing-off for m-gram language modeling. In International Conference on Acoustics, Speech, and Signal Processing, pages 181–184.

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.

Lee, A., Fiscus, J., Garofolo, J., Przybocki, M., Martin, A., Sanders, G., and Pallett, D. (May 2003). Spring speech-to-text transcription evaluation results. In Rich Transcription Workshop, Boston.

Liu, J. S. (2001). Monte Carlo Strategies in Scientific Computing. Springer.

Luis, O. and Leslie, K. (2000). Adaptive importance sampling for estimation in structured domains. In Proceedings of the 16th Annual Conference on Uncertainty in Artificial Intelligence (UAI-00), pages 446–454.

Intel math kernel library (2004)., http://www.intel.com/software/products/mkl/.

Miikkulainen, R. and Dyer, M. (1991). Natural language processing with modular neural networks and distributed lexicon. Cognitive Science, 15, 343–399.

Ney, H. and Kneser, R. (1993). Improved clustering techniques for class-based statistical language modeling. In European Conference on Speech Communication and Technology (Eurospeech), pages 973–976, Berlin.

Niesler, T., Whittaker, E., and Woodland, P. (1998). Comparison of part-of-speech and automatically derived category-based language models for speech recognition. In International Conference on Acoustics, Speech, and Signal Processing, pages 177–180.

Paccanaro, A. and Hinton, G. (2000). Extracting distributed representations of concepts and relations from positive and negative propositions. In Proceedings of the International Joint Conference on Neural Network, IJCNN'2000, Como, Italy. IEEE, New York.

Pereira, F., Tishby, N., and Lee, L. (1993). Distributional clustering of English words. In 30th Annual Meeting of the Association for Computational Linguistics, pages 183–190, Columbus, Ohio.

Riis, S. and Krogh, A. (1996). Improving protein secondary structure prediction using structured neural networks and multiple sequence profiles. Journal of Computational Biology, pages 163–183. 40 Authors Suppressed Due to Excessive Length

Robert, C. P. and Casella, G. (2000). Monte Carlo Statistical Methods. Springer. Springer texts in statistics.

Salton, G. and Buckley, C. (1988). Term weighting approaches in automatic text retrieval. Information Processing and Management, 24(5), 513–523.

Schmidhuber, J. (1996). Sequential neural text compression. IEEE Transactions on Neural Networks, 7(1), 142–146.

Schutze, H. (1993). Word space. In C. Giles, S. Hanson, and J. Cowan, editors, Advances in Neural Information Processing Systems 5, pages pp. 895–902, San Mateo CA. Morgan Kaufmann.

Schwenk, H. and Gauvain, J.-L. (2002). Connectionist language modeling for large vocabulary continuous speech recognition. In International Conference on Acoustics, Speech, and Signal Processing, pages I: 765–768.

Schwenk, H. and Gauvain, J.-L. (2003). Using continuous space language models for conversational speech recognition. In ISCA & IEEE Workshop on Spontaneous Speech Processing and Recognition, Tokyo.

Schwenk, H. (2004). Efficient training of large neural networks for language modeling. In IEEE joint conference on neural networks, pages 3059–3062.

Schwenk, H. and Gauvain, J.-L. (2004). Neural network language models for conversational speech recognition. In International Conference on Speech and Language Processing, pages 1215–1218.

Schwenk, H. and Gauvain, J.-L. (2005). Building Continuous Language Models for Transcribing European Languages. In Eurospeech. To appear.

Stolcke, A. (2002). SRILM - an extensible language modeling toolkit. In Proceedings of the International Conference on Statistical Language Processing, Denver, Colorado.

Xu, W. and Rudnicky, A. (2000). Can artificial neural network learn language models? In International Conference on Statistical Language Processing, pages M1–13, Beijing, China.