# Strategies for Training Large Vocabulary Neural Language Models

**Wenlin Chen** **David Grangier** **Michael Auli**
Facebook, Menlo Park, CA

## Abstract

Training neural network language models over large vocabularies is computationally costly compared to count-based models such as Kneser-Ney. We present a systematic comparison of neural strategies to represent and train large vocabularies, including softmax, hierarchical softmax, target sampling, noise contrastive estimation and self normalization. We extend self normalization to be a proper estimator of likelihood and introduce an efficient variant of softmax. We evaluate each method on three popular benchmarks, examining performance on rare words, the speed/accuracy trade-off and complementarity to Kneser-Ney.

## 1 Introduction

Neural network language models (Bengio et al., 2003; Mikolov et al., 2010) have gained popularity for tasks such as automatic speech recognition (Arisoy et al., 2012) and statistical machine translation (Schwenk et al., 2012; Vaswani et al., 2013; Baltescu and Blunsom, 2014). Similar models are also developed for translation (Le et al., 2012; Devlin et al., 2014; Bahdanau et al., 2015), summarization (Chopra et al., 2015) and language generation (Sordoni et al., 2015).

Language models assign a probability to a word given a context of preceding, and possibly subsequent, words. The model architecture determines how the context is represented and there are several choices including recurrent neural networks (Mikolov et al., 2010; Jozefowicz et al., 2016), or log-bilinear models (Mnih and Hinton, 2010). This paper does not focus on architecture or context representation but rather on how to efficiently deal with large output vocabularies, a problem common to all approaches to neural language modeling and related tasks (machine translation, language generation). We therefore experiment with a classical feed-forward neural network model similar to Bengio et al. (2003).

Practical training speed for these models quickly decreases as the vocabulary grows. This is due to three combined factors: (i) model evaluation and gradient computation become more time consuming, mainly due to the need of computing normalized probabilities over a large vocabulary; (ii) large vocabularies require more training data in order to observe enough instances of infrequent words which increases training times; (iii) a larger training set often allows for larger models which requires more training iterations.

This paper provides an overview of popular strategies to model large vocabularies for language modeling. This includes the classical *softmax* over all output classes, *hierarchical softmax* which introduces latent variables, or clusters, to simplify normalization, target sampling which only considers a random subset of classes for normalization, *noise contrastive estimation* which discriminates between genuine data points and samples from a noise distribution, and *infrequent normalization*, also referred as self-normalization, which computes the partition function at an infrequent rate. We also extend self-normalization to be a proper estimator of likelihood. Furthermore, we introduce *differentiated softmax*, a novel variation of softmax which assigns more parameters, or capacity, to frequent words and which we show to be faster and more accurate than softmax (§2).

Our comparison assumes a reasonable budget of one week for training models on a high end GPU (Nvidia K40). We evaluate on three benchmarks differing in the amount of training data and vocabulary size, that is Penn Treebank, Gigaword and the Billion Word benchmark (§3).

Our results show that conclusions drawn from small datasets do not always generalize to larger settings. For instance, hierarchical softmax is less accurate than softmax on the small vocabulary Penn Treebank task but performs best on the very large vocabulary Billion Word benchmark. This is because hierarchical softmax is the fastest method for training and can perform more training updates in the same period of time. Furthermore, our re-

sults with differentiated softmax demonstrate that assigning capacity where it has the most impact allows to train better models in our time budget (§4). Our analysis also shows clearly that traditional Kneser-Ney models are competitive on rare words, contrary to the common belief that neural models are better on infrequent words (§5).

## 2 Modeling Large Vocabularies

We first introduce our model architecture with a classical softmax and then describe various other methods including a novel variation of softmax.

### 2.1 Softmax Neural Language Model

Our feed-forward neural network implements an n-gram language model, i.e., it is a parametric function estimating the probability of the next word $w^t$ given $n-1$ previous context words, $w^{t-1}, \ldots, w^{t-n+1}$. Formally, we take as input a sequence of discrete indexes representing the $n-1$ previous words and output a vocabulary-sized vector of probability estimates, i.e.,

$$f : \{1, \ldots, V\}^{n-1} \rightarrow [0,1]^V,$$

where $V$ is the vocabulary size. This function results from the composition of simple differentiable functions or *layers*.

Specifically, $f$ composes an input mapping from discrete word indexes to continuous vectors, a succession of linear operations followed by hyperbolic tangent non-linearities, plus one final linear operation, followed by a softmax normalization.

The input layer maps each context word index to a continuous $d'_0$-dimensional vector. It relies on a matrix $W^0 \in \mathbb{R}^{V \times d'_0}$ to convert the input

$$x = [w^{t-1}, \ldots, w^{t-n+1}] \in \{1, \ldots, V\}^{n-1}$$

to $n-1$ vectors of dimension $d'_0$. These vectors are concatenated into a single $(n-1) \times d'_0$ matrix,

$$h^0 = [W^0_{w^{t-1}}; \ldots; W^0_{w^{t-n+1}}] \in \mathbb{R}^{n-1 \times d'_0}.$$

This state $h^0$ is considered as a $d_0 = (n-1) \times d'_0$ vector by the next layer. The subsequent states are computed through $k$ layers of linear mappings followed by hyperbolic tangents, i.e.

$$\forall i = 1, \ldots, k, \quad h^i = \tanh(W^i h^{i-1} + b^i) \in \mathbb{R}^{d_i}$$

where $W^i \in \mathbb{R}^{d_i \times d_{i-1}}, b \in \mathbb{R}^{d_i}$ are learnable weights and biases and $\tanh$ denotes the component-wise hyperbolic tangent.

Finally, the last layer performs a linear operation followed by a softmax normalization, i.e.,

$$\begin{aligned} & h^{k+1} = W^{k+1} h^k + b^{k+1} \in \mathbb{R}^V \\ \text{and} \quad & y = \tfrac{1}{Z} \exp(h^{k+1}) \in [0,1]^V \end{aligned} \tag{1}$$

where $Z = \sum_{j=1}^{V} \exp(h^{k+1}_j)$ and $\exp$ denotes the component-wise exponential. The network output $y$ is therefore a vocabulary-sized vector of probability estimates. We use the standard cross-entropy loss with respect to the computed log probabilities

$$\frac{\partial \log y_i}{\partial h^{k+1}_j} = \delta_{ij} - y_j$$

where $\delta_{ij} = 1$ if $i = j$ and 0 otherwise The gradient update therefore increases the score of the correct output $h^{k+1}_i$ and decreases the score of all other outputs $h^{k+1}_j$ for $j \neq i$.

A downside of the classical softmax formulation is that it requires computation of the activations for *all output words*, Eq. (1). The output layer with $V$ activations is much larger than any other layer in the network and its matrix multiplication dominates the complexity of the entire network.

### 2.2 Hierarchical Softmax

Hierarchical Softmax (HSM) organizes the output vocabulary into a tree where the leaves are the words and the intermediate nodes are latent variables, or *classes* (Morin and Bengio, 2005). The tree has potentially many levels and there is a unique path from the root to each word. The probability of a word is the product of the probabilities of the latent variables along the path from the root to the leaf, including the probability of the leaf.

We follow Goodman (2001) and Mikolov et al. (2011b) and model a two-level tree. Given context $x$, HSM predicts the *class* of the next word $c^t$ and the actual word $w^t$

$$p(w^t|x) = p(c^t|x)\, p(w^t|c^t, x) \tag{2}$$

If the number of classes is $\mathcal{O}(\sqrt{V})$ and classes are balanced, then we only need to compute $\mathcal{O}(2\sqrt{V})$ outputs. In practice, this strategy results in weight matrices whose largest dimension is $< 1,000$, a setting for which GPU hardware is fast.

A popular strategy is *frequency* clustering. It sorts the vocabulary by frequency and then forms clusters of words with similar frequency. Each cluster contains an equal share of the total unigram probability. We compare this strategy to random class assignment and to clustering based on word

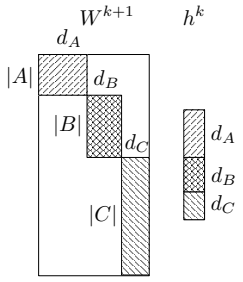


Figure 1: Output weight matrix $W^{k+1}$ and hidden layer $h^k$ for differentiated softmax for vocabulary partitions $A, B, C$ with embedding dimensions $d_A, d_B, d_C$; non-shaded areas are zero.

contexts, relying on PCA (Lebret and Collobert, 2014). A full comparison of context-based clustering is beyond the scope of this work (Brown et al., 1992; Mikolov et al., 2013).

### 2.3 Differentiated Softmax

This section introduces a novel variation of softmax that assigns a variable number of parameters to each word in the output layer. The weight matrix of the final layer $W^{k+1} \in \mathbb{R}^{d_k \times V}$ stores *output embeddings* of size $d_k$ for the $V$ words the language model may predict: $W_1^{k+1}; \ldots; W_V^{k+1}$. Differentiated softmax (D-Softmax) varies the dimension of the output embeddings $d_k$ across words depending on how much model capacity, or parameters, are deemed suitable for a given word. We assign more parameters to frequent words than to rare words since more training occurrences allow for fitting more parameters.

We partition the output vocabulary based on word frequency and the words in each partition share the same embedding size. Partitioning the vocabulary in this way results in a sparse final weight matrix $W^{k+1}$ which arranges the embeddings of the output words in blocks, each block corresponding to a separate partition (Figure 1). The size of the final hidden layer $h^k$ is the sum of the embedding sizes of the partitions. The final hidden layer is effectively a concatenation of separate features for each partition which are used to compute the dot product with the corresponding embedding type in $W^{k+1}$. In practice, we efficiently compute separate matrix-vector products, or in batched form, matrix-matrix products, for each partition in $W^{k+1}$ and $h^k$.

Overall, differentiated softmax can lead to large speed-ups as well as accuracy gains since we can greatly reduce the complexity of computing the output layer. Most significantly, this strategy speeds up *both* training and inference. This is in contrast to hierarchical softmax which is fast during training but requires even more effort than softmax for computing the most likely next word.

### 2.4 Target Sampling

Sampling-based methods approximate the softmax normalization, Eq. (1), by summing over a sub-sample of impostor classes. This can significantly speed-up each training iteration, depending on the size of the impostor set. Target sampling builds upon the importance sampling work of Bengio and Senécal (2008). We follow Jean et al. (2014) who choose as impostors all positive examples in a mini-batch as well as a subset of the remaining words. This subset is sampled uniformly and its size is chosen by validation.

### 2.5 Noise Contrastive Estimation

Noise contrastive estimation (NCE) is another sampling-based technique (Hyvärinen, 2010; Mnih and Teh, 2012; Chen et al., 2015). Contrary to target sampling, it does not maximize the training data likelihood directly. Instead, it solves a two-class problem of distinguishing genuine data from noise samples. The training algorithm samples a word $w$ given the preceding context $x$ from a mixture

$$p(w|x) = \frac{1}{k+1} p_{\text{train}}(w|x) + \frac{k}{k+1} p_{\text{noise}}(w|x)$$

where $p_{\text{train}}$ is the empirical distribution of the training set and $p_{\text{noise}}$ is a known noise distribution which is typically a context-independent unigram distribution. The training algorithm fits the model $\hat{p}(w|x)$ to recover whether a mixture sample came from the data or the noise distribution, this amounts to minimizing the binary cross-entropy $-y \log \hat{p}(y=1|w,x) - (1-y) \log \hat{p}(y=0|w,x)$ where $y$ is a binary variable indicating where the current sample originates from

$$\begin{cases} \hat{p}(y=1|w,x) = \frac{\hat{p}(w|x)}{\hat{p}(w|x) + k p_{\text{noise}}(w|x)} & \text{(data)} \\ \hat{p}(y=0|w,x) = 1 - \hat{p}(y=1|w,x) & \text{(noise).} \end{cases}$$

This formulation still involves a softmax over the vocabulary to compute $\hat{p}(w|x)$. However, Mnih and Teh (2012) suggest to forego normalization and replace $\hat{p}(w|x)$ with unnormalized exponentiated scores. This makes the training complexity independent of the vocabulary size. At test time, softmax normalization is reintroduced to get a proper distribution. We also follow Mnih and Teh (2012) recommendations for $p_{\text{noise}}$ and rely on a unigram distribution of the training set.

### 2.6 Infrequent Normalization

Devlin et al. (2014), followed by Andreas and Klein (2015), proposed to relax score normalization. Their strategy (here referred to as WeaknormSQ) associates unnormalized likelihood maximization with a penalty term that favors normalized predictions. This yields the following loss over the training set $T$

$$L_{\alpha}^{(2)} = -\sum_{(w,x)\in T} s(w|x) + \alpha \sum_{(w,x)\in T} (\log Z(x))^2$$

where $s(w|x)$ refers to the unnormalized score of word $w$ given context $x$ and $Z(x) = \sum_w \exp(s(w|x))$ refers to the partition function for context $x$. This strategy therefore pushes the log partition towards zero. For efficient training, the second term can be down-sampled

$$L_{\alpha,\gamma}^{(2)} = -\sum_{(w,x)\in T} s(w|x) + \frac{\alpha}{\gamma} \sum_{(w,x)\in T_\gamma} (\log Z(x))^2$$

where $T_\gamma$ is the training set sampled at rate $\gamma$. A small rate implies computing the partition function only for a small fraction of the training data.

We extend this strategy to the case where the log partition term is not squared (Weaknorm), i.e.,

$$L_{\alpha,\gamma}^{(1)} = -\sum_{(w,x)\in T} s(w|x) + \frac{\alpha}{\gamma} \sum_{(w,x)\in T_\gamma} \log Z(x)$$

For $\alpha = 1$, this loss is an unbiased estimator of the negative log-likelihood of the training data $L_1^{(2)} = -\sum_{(w,x)\in T} s(w|x) + \log Z(x)$.

## 3 Experimental Setup

**Datasets** We run experiments over three news datasets of different sizes: Penn Treebank (PTB), WMT11-lm (billionW) and English Gigaword, version 5 (gigaword). Penn Treebank (Marcus et al., 1993) is the smallest corpus with 1M tokens and we use a vocabulary size of 10k (Mikolov et al., 2011a). The billion word benchmark (Chelba et al., 2013) comprises almost one billion tokens and a vocabulary of about 800k words[1]. Gigaword (Parker et al., 2011) is even larger with 5 billion tokens and was previously used for language modeling (Heafield, 2011) but there is no standard train/test split or vocabulary for this set. We split according to time: training covers 1994–2009 and test covers 2010. The vocabulary comprises the 100k most frequent words in train. Table 1 summarizes the data statistics.

[1]T. Robinson version `http://tiny.cc/1billionLM`.

| Dataset | Train | Test | Vocab | OOV |
|---|---|---|---|---|
| PTB | 1M | 0.08M | 10k | 5.8% |
| gigaword | 4,631M | 279M | 100k | 5.6% |
| billionW | 799M | 8.1M | 793k | 0.3% |

Table 1: Dataset statistics. Number of tokens for train and test, vocabulary size, fraction of OOV.

**Evaluation** We measure perplexity on the test set. For PTB and billionW, we report results on a per sentence basis, i.e., models do not use context words across sentence boundaries and we score end-of-sentence markers. This is the standard setting for these benchmarks and allows comparison with other work. On gigaword, we use contexts across sentence boundaries and evaluation does not include end-of-sentence markers.

Our baseline is an interpolated Kneser-Ney (KN) model. We use KenLM (Heafield, 2011) to train 5-gram models without pruning. For neural models, we train 11-gram models for gigaword and billionW; for PTB we train a 6-gram model. The model parameters (weights $W^i$ and biases $b^i$ for $i = 0, \ldots, k+1$) are learned to maximize the training log-likelihood relying on stochastic gradient descent (SGD; LeCun et al.. 1998).

**Validation** Hyper-parameters are the number of layers $k$ and the dimension of each layer $d_i, \forall i = 0, \ldots, k$. We tune the following settings for each technique on the validation set: the number of clusters, the clustering technique for hierarchical softmax, the number of frequency bands and their allocated capacity for differentiated softmax, the number of distractors for target sampling, the noise/data ratio for NCE, as well as the regularization rate and strength for infrequent normalization. Similarly, SGD parameters (learning rate and mini-batch size) are set to maximize validation likelihood. We also tune the dropout rate (Srivastava et al., 2014); dropout is employed after each tanh non-linearity.[2]

**Training Time** We train for 168 hours (one week) on the large datasets (billionW, gigaword) and 24 hours (one day) for Penn Treebank. All experiments are performed on the same hardware, a single K40 GPU. We select the hyper-parameters which yield the best validation perplexity after the allocated time and report the perplexity of the resulting model on the test set. This training time

[2]More parameter settings are available in an extended version of the paper at `http://arxiv.org/abs/1512.04906`.

is a trade-off between being able to do a comprehensive exploration of the various settings for each method and good accuracy. The chosen training times are not long enough to observe over-fitting, i.e. validation performance is still improving – albeit very slowly – at the end of the training session. As a general observation, even on the small PTB where 24 hours is rather long, we always found better results using the full training time, possibly increasing the dropout rate.

A concern may be that a fixing the training time favors models with better implementations. However, all models are very similar and their core computations are always matrix/matrix products. Training differs mostly in the size and frequency of large matrix/matrix products. Matrix products rely on CuBLAS[3], using torch[4]. For the matrix sizes involved ($> 500 \times 1,000$), the time complexity of matrix product is linear in each dimension, both on CPU (Intel MKL[5]) and GPU (CuBLAS), with a $10X$ speedup for GPU (Nvidia K40) compared to CPU (Intel Xeon E5-2680). Therefore, the speed trade-off applies to both CPU and GPU hardware, albeit with a different time scale.

## 4 Results

The test perplexities (Table 2) and validation learning curves (Figures 2, 3, and 4) show that the competitiveness of softmax diminishes with larger vocabularies. Softmax does well on the small vocabulary PTB but poorly on the large vocabulary billionW corpus. Faster methods such as sampling, hierarchical softmax, and infrequent normalization (Weaknorm, WeaknormSQ) are much better in the large vocabulary setting of billionW.

D-Softmax is performing well on all sets and shows that assigning higher capacity where it benefits most results in better models. Target sampling performs worse than softmax on gigaword but better on billionW. Hierarchical softmax performs poorly on Penn Treebank which is in stark contrast to billionW where it does well. Noise contrastive estimation has good accuracy on billionW, where speed is essential to achieving good accuracy.

Of all the methods, hierarchical softmax processes most training examples in a given time frame (Table 3). Our test time speed comparison assumes that we would like to find the highest scoring next word rather than rescoring an existing string. This scenario requires scoring all output words and D-Softmax can process nearly twice as many tokens per second than the other methods whose complexity is similar to softmax.

| | PTB | gigaW | billionW |
|---|---|---|---|
| KN | 141.2 | 57.1 | 70.2[6] |
| Softmax | 123.8 | 56.5 | 108.3 |
| D-Softmax | **121.1** | **52.0** | 91.2 |
| Sampling | 124.2 | 57.6 | 101.0 |
| HSM | 138.2 | 57.1 | **85.2** |
| NCE | 143.1 | 78.4 | 104.7 |
| Weaknorm | 124.4 | 56.9 | 98.7 |
| WeaknormSQ | 122.1 | 56.1 | 94.9 |
| KN+Softmax | 108.5 | 43.6 | 59.4 |
| KN+D-Softmax | **107.0** | **42.0** | 56.3 |
| KN+Sampling | 109.4 | 43.8 | 58.1 |
| KN+HSM | 115.0 | 43.9 | **55.6** |
| KN+NCE | 114.6 | 49.0 | 58.8 |
| KN+Weaknorm | 109.2 | 43.8 | 58.1 |
| KN+WeaknormSQ | 108.8 | 43.8 | 57.7 |

Table 2: Test perplexity of individual models and interpolation with Kneser-Ney.

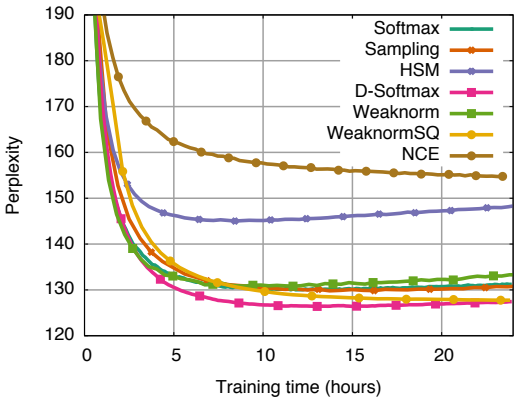


Figure 2: PTB validation learning curve.

### 4.1 Softmax

Despite being our baseline, softmax ranks among the most accurate methods on PTB and it is second best on gigaword after D-Softmax (with WeaknormSQ performing similarly). For billionW, the extremely large vocabulary makes softmax training too slow to compete with faster alterna-

[3] http://docs.nvidia.com/cuda/cublas/

[4] http://torch.ch

[5] https://software.intel.com/en-us/intel-mkl

[6] This perplexity is higher than reported in (Chelba et al., 2013), in which Kneser Ney is not trained on the 800m token training set, but on a larger corpus of 1.1B tokens.

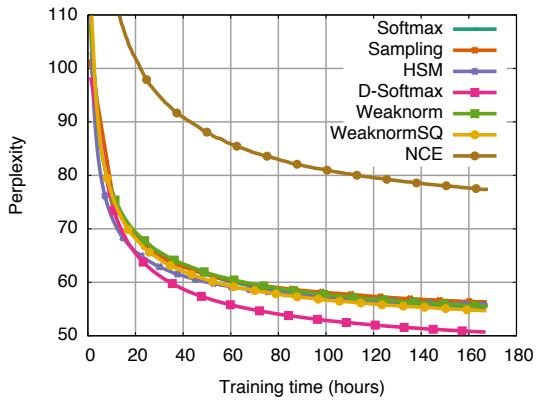


Figure 3: Gigaword validation learning curve.

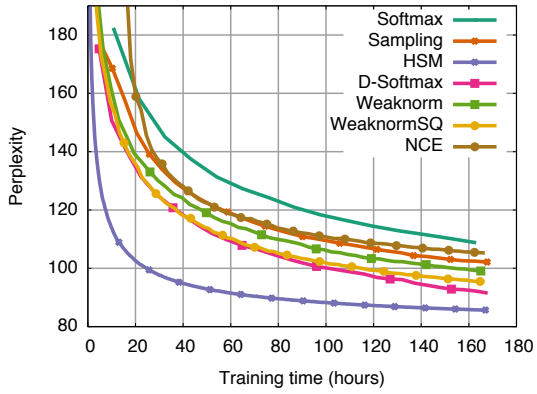


Figure 4: Billion Word validation learning curve.

| | train | test |
|---|---|---|
| Softmax | 510 | 510 |
| D-Softmax | 960 | **960** |
| Sampling | 1,060 | 510 |
| HSM | **12,650** | 510 |
| NCE | 4,520 | 510 |
| Weaknorm | 1,680 | 510 |
| WeaknormSQ | 2,870 | 510 |

Table 3: Training and test speed on billionW in tokens per second for generation of the next word. Most techniques are identical to softmax at test time. HSM can be faster for rescoring.

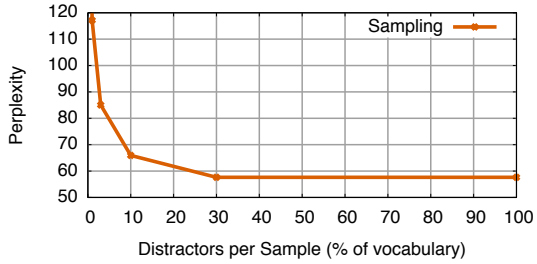


Figure 5: Number of Distractors versus Perplexity for Target Sampling over Gigaword

tives. However, of all the methods softmax has the simplest implementation and it has no additional hyper-parameters compared to other methods.

### 4.2 Target Sampling

Figure 5 shows that target sampling is most accurate for distractor sets that amount to a large fraction of the vocabulary, i.e. $> 30\%$ on gigaword (billionW best setting $> 50\%$ is even higher). Target sampling is faster and performs more iterations than softmax in the same time. However, its perplexity reduction per iteration is less than softmax. Overall, it is not much better than softmax. A reason might be that sampling chooses distractors independently from context and current model performance. This does not favor distractors the model incorrectly considers likely for the current context. These distractors would yield higher gradients that could update the model faster.

### 4.3 Hierarchical Softmax

Hierarchical softmax is very efficient for large vocabularies and it is the best method on billionW. On the other hand, HSM does poorly on small vocabularies as seen on PTB. We found that a good word clustering structure is crucial: when clusters gather words occurring in similar contexts, cluster likelihoods are easier to learn; when the cluster structure is uninformative, cluster likelihoods converge to the uniform distribution. This affects accuracy since words cannot have higher probability than their clusters, Eq. (2).

Our experiments organize words into a two level hierarchy and compare four clustering strategies on billionW and gigaword (§2.2). Random clustering shuffles the vocabulary and splits it into equally sized partitions. Frequency-based clustering first orders words based on their frequency and assigns words to clusters such that each cluster represents an equal share of the total frequency (Mikolov et al., 2011b). K-means runs the well-known clustering algorithm on Hellinger PCA word embeddings. Weighted k-means weights each word by its frequency.[7]

Random clusters perform worst (Table 4) followed by frequency-based clustering but k-means does best; weighted k-means performs similarly to its unweighted version. In earlier experiments, plain k-means performed very poorly since the most frequent cluster captured up to $40\%$ of the

[7]The time to compute the clustering (multi-threaded word co-occurrence counts, PCA and k-means) is under one hour, which is negligible given a one week training budget.

| | billionW | gigaword |
|---|---|---|
| random | 98.51 | 62,27 |
| frequency-based | 92.02 | 59.47 |
| k-means | 85.70 | 57.52 |
| weighted k-means | 85.24 | 57.09 |

Table 4: HSM with different clustering.

token occurrences. We then explicitly capped the frequency budget of each cluster to $10\%$ which brought k-means on par with weighted k-means.

### 4.4 Differentiated Softmax

D-Softmax is the best technique on gigaword and second best on billionW after HSM. On PTB it ranks among the best techniques whose perplexities cannot be reliably distinguished. The variable-capacity scheme of D-Softmax can assign large embeddings to frequent words, while keeping computational complexity manageable through small embeddings for rare words.

Unlike for hierarchical softmax, NCE or Weaknorm, the computational advantage of D-Softmax is preserved at test time (Table 3). D-Softmax is the fastest technique at test time, while ranking among the most accurate methods. This speed advantage is due to the low dimensional representation of rare words which negatively affects the model accuracy on these words (Table 5).

### 4.5 Noise Contrastive Estimation

Although we report better perplexities than the original NCE paper on PTB (Mnih and Teh, 2012), we found NCE difficult to use for large vocabularies. In order to work in this setting where models are larger, we had to dissociate the number of noise samples from the data to noise ratio in the modeled mixture. For instance, a data/noise ratio of $1/50$ gives good performance in our experiments but estimating only $50$ noise sample posteriors per data point is wasteful given the cost of network evaluation. Moreover, $50$ samples do not allow frequent sampling of every word in a large vocabulary. Our setting considers more noise samples and up-weights the data sample. This allows to set the data/noise ratio independently from the number of noise samples.

Overall, NCE results are better than softmax only for billionW, a setting for which softmax is very slow due to the very large vocabulary. Why does NCE perform so poorly? Figure 6 shows entropy on the validation set versus the NCE loss for several models. The results clearly show that similar NCE loss values can result in very different validation entropy. Although NCE might make sense for other metrics such as BLEU (Baltescu and Blunsom, 2014), it is not among the best techniques for minimizing perplexity. Jozefowicz et al. (2016) recently drew similar conclusions.

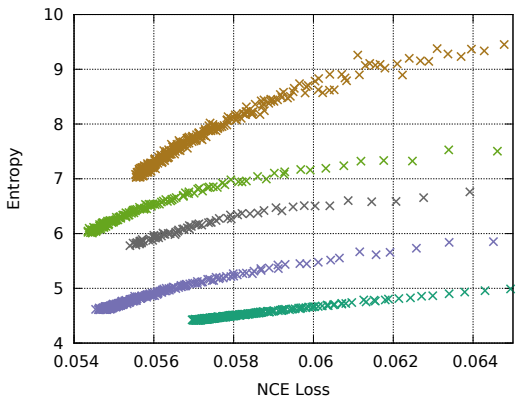


Figure 6: Validation entropy versus NCE loss on gigaword for experiments differing only in learning rates and initial weights. Each color corresponds to one experiment, with one point per hour.

### 4.6 Infrequent Normalization

Infrequent normalization (Weaknorm and WeaknormSQ) performs better than softmax on billionW and comparably to softmax on Penn Treebank and gigaword (Table 2). The speedup from skipping partition function computations is substantial. For instance, WeaknormSQ on billionW evaluates the partition only on $10\%$ of the examples. In one week, the model is evaluated and updated on 868M tokens (with 86.8M partition evaluations) compared to 156M tokens for softmax.

Although referred to as self-normalizing (Andreas and Klein, 2015), the trained models still need normalization after training. The partition varies greatly between data samples. On billionW, the partition ranges between $9.4$ to $10.3$ in log scale for 10th to 90th percentile, i.e. a ratio of $2.5$.

We observed the squared version (WeaknormSQ) to be unstable at times. Regularization strength could be found too low (collapse) or too high (blow-up) after a few days of training. We added an extra unit to bound unnormalized predictions $x \rightarrow 10 \tanh(x/5)$, which yields stable training and better generalization. For the non-squared Weaknorm, stability was not an issue. A regularization strength of $1$ was the best setting for Weaknorm. This choice makes the loss an unbiased estimator of the data likelihood.

| | 1-4K | 4-20K | 20-40K | 40-70K | 70-100K |
|---|---|---|---|---|---|
| Kneser-Ney | 3.48 | 7.85 | 9.76 | 10.76 | 11.57 |
| Softmax | 3.46 | 7.87 | 9.76 | 11.09 | 12.39 |
| D-Softmax | **3.35** | 7.79 | 10.13 | 12.22 | 12.69 |
| Target sampling | 3.51 | **7.62** | 9.51 | 10.81 | 12.06 |
| HSM | 3.49 | 7.86 | **9.38** | **10.30** | **11.24** |
| NCE | 3.74 | 8.48 | 10.60 | 12.06 | 13.37 |
| Weaknorm | 3.46 | 7.86 | 9.77 | 11.12 | 12.40 |
| WeaknormSQ | 3.46 | 7.79 | 9.67 | 10.98 | 12.32 |

Table 5: Test entropy on gigaword over subsets of the frequency ranked vocabulary; rank 1 is the most frequent word.

## 5 Analysis

### 5.1 Model Capacity

Training neural language models over large corpora highlights that training time, not training data, is the main factor limiting performance. The learning curves on gigaword and billionW indicate that most models are still making progress after one week. Training time has therefore to be taken into account when considering increasing capacity. Figure 7 shows validation perplexity versus the number of iterations for a week of training. This figure shows that a softmax model with 1024 hidden units in the last layer could perform better than the 512-hidden unit model with a longer training horizon. However, in the allocated time, 512 hidden units yield the best validation performance. D-softmax shows that it is possible to *selectively* increase capacity, i.e., to allocate more hidden units to the most frequent words at the expense of rarer words. This captures most of the benefit of a larger softmax model while staying within a reasonable training budget.

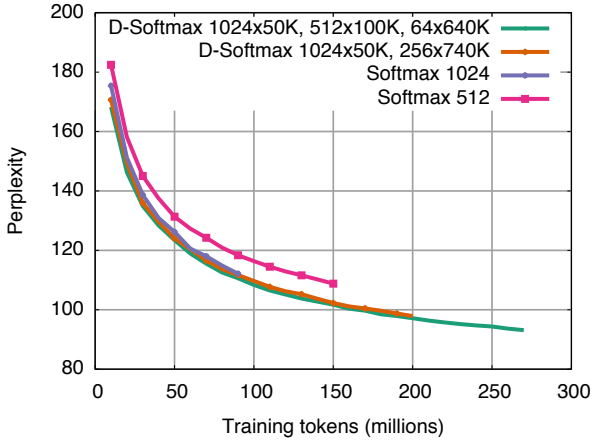


Figure 7: Validation perplexity per iteration on billionW for softmax and D-softmax. Softmax uses the same number of units for all words. The first D-Softmax experiment uses 1024 units for the 50K most frequent words, 512 for the next 100K, and 64 units for the rest; similarly for the second experiment. All experiments end after one week.

### 5.2 Effect of Initialization

We consider initializing both the input word embeddings and the output matrix from Hellinger PCA embeddings. Several alternative techniques for pre-training embeddings have been proposed (Mikolov et al., 2013; Lebret and Collobert, 2014; Pennington et al., 2014). Our experiment highlights the advantage of initialization and do not aim to compare embedding techniques.

Figure 8 shows that PCA is better than random for initializing both input and output word representations; initializing both from PCA is even better. We see that even after long training sessions, the initial conditions still impact the validation perplexity. We observed this trend also with other strategies than softmax. After one week of training, HSM is the only method which can reach comparable accuracy to PCA initialization when the output matrix is randomly initialized.

### 5.3 Training Set Size

Large training sets and a fixed training time introduce competition between slower models with more capacity and observing more training data. This trade-off only applies to iterative SGD optimization and does not apply to classical count-based models, which visit the training set once and then solve training in closed form.

We compare Kneser-Ney and softmax, trained for one week, with gigaword on differently sized subsets of the training data. For each setting we take care to include all data from the smaller subsets. Figure 9 shows that the performance of the neural model improves very little on more than

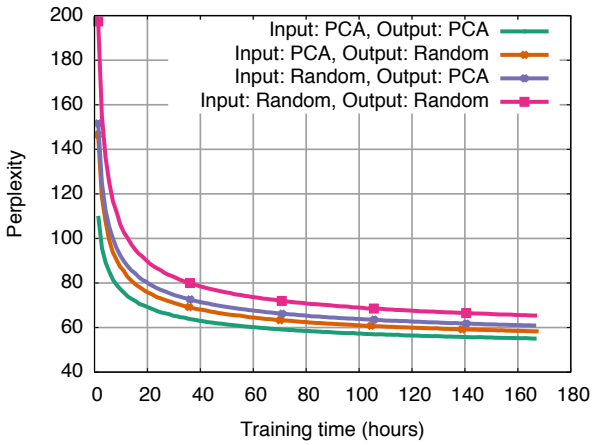


Figure 8: Effect of random initialization and with Hellinger PCA on gigaword for softmax.

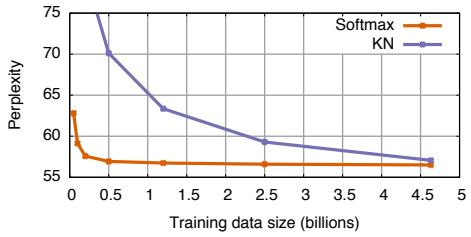


Figure 9: Effect of training set size measured on test of gigaword for Softmax and Kneser-Ney.

500M tokens. In order to benefit from the full training set we would require a much higher training budget, faster hardware, or parallelization.

Scaling training to large datasets can have a significant impact on perplexity, even when data from the distribution of interest is limited. As an illustration, we adapted a softmax model trained on billionW to Penn Treebank and achieved a perplexity of 96 - a far better result than with any model we trained from scratch on PTB (cf. Table 2).

### 5.4 Rare Words

How well do neural models perform on rare words? To answer this question, we computed entropy across word frequency bands for Kneser-Ney and neural models. Table 5 reports entropy for the $4,000$ most frequent words, then the next most frequent $16,000$ words, etc. For frequent words, neural models are on par or better than Kneser-Ney. For rare words, Kneser-Ney is very competitive. Although neural models might eventually close this gap with much longer training, one should consider that Kneser-Ney trains on gigaword in only $8$ hours on CPU which contrasts with $168$ hours of training for neural models on high end GPUs. This result highlights the complementarity of both approaches, as observed in our interpolation experiments (Table 2).

For neural models, D-Softmax excels on frequent words but performs poorly on rare ones. This is because D-Softmax assigns more capacity to frequent words at the expense of rare words. Overall, hierarchical softmax is the best neural technique for rare words. HSM does more iterations than any other technique and so it can observe every rare word more often.

## 6 Conclusions

This paper presents a comprehensive analysis of strategies to train neural language models with large vocabularies. This setting is very challenging for neural networks as they need to compute the partition function over the entire vocabulary at each evaluation.

We compared classical softmax to hierarchical softmax, target sampling, noise contrastive estimation and infrequent normalization, commonly referred to as self-normalization. Furthermore, we extend infrequent normalization to be a proper estimator of likelihood and we introduce differentiated softmax, a novel variant of softmax assigning less capacity to rare words to reduce computation.

Our results show that methods which are effective on small vocabularies are not necessarily equally so on large vocabularies. In our setting, target sampling and noise contrastive estimation failed to outperform the softmax baseline. Overall, differentiated softmax and hierarchical softmax are the best strategies for large vocabularies. Compared to classical Kneser-Ney models, neural models are better at modeling frequent words, but are less effective for rare words. A combination of the two is therefore very effective.

We conclude that there is a lot to explore in training from a combination of normalized and unnormalized objectives. An interesting future direction is to combine complementary approaches, either through combined parameterization (e.g. hierarchical softmax with differentiated capacity per word) or through a curriculum (e.g. transitioning from target sampling to regular softmax as training progresses). Further promising areas are parallel training as well as better rare word modeling.

## References

Jacob Andreas and Dan Klein. 2015. When and why are log-linear models self-normalizing? In *Proc. of NAACL*.

Ebru Arisoy, Tara N. Sainath, Brian Kingsbury, and Bhuvana Ramabhadran. 2012. Deep Neural Network Language Models. In *NAACL-HLT Workshop on the Future of Language Modeling for HLT*, pages 20–28, Stroudsburg, PA, USA. Association for Computational Linguistics.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *Proc. of ICLR*. Association for Computational Linguistics, May.

Paul Baltescu and Phil Blunsom. 2014. Pragmatic neural language modelling in machine translation. Technical Report arXiv 1412.7119.

Yoshua Bengio and Jean-Sébastien Senécal. 2008. Adaptive importance sampling to accelerate training of a neural probabilistic language model. *IEEE Transactions on Neural Networks*.

Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. 2003. A Neural Probabilistic Language Model. *Journal of Machine Learning Research*, 3:1137–1155.

Peter F. Brown, Peter V. deSouza, Robert L. Mercer, Vincent J. Della Pietra, and Jenifer C. Lai. 1992. Class-based $n$-gram models of natural language. *Computational Linguistics*, 18(4):467–479, Dec.

Ciprian Chelba, Tomáš Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, Phillipp Koehn, and Tony Robinson. 2013. One billion word benchmark for measuring progress in statistical language modeling. Technical report, Google.

Xie Chen, Xunying Liu, MJF Gales, and PC Woodland. 2015. Recurrent neural network language model training with noise contrastive estimation for speech recognition. In *Acoustics, Speech and Signal Processing (ICASSP)*.

Sumit Chopra, Jason Weston, and Alexander M. Rush. 2015. Tuning as ranking. In *Proc. of EMNLP*. Association for Computational Linguistics, Sep.

Jacob Devlin, Rabih Zbib, Zhongqiang Huang, Thomas Lamar, Richard Schwartz, , and John Makhoul. 2014. Fast and Robust Neural Network Joint Models for Statistical Machine Translation. In *Proc. of ACL*. Association for Computational Linguistics, June.

Joshua Goodman. 2001. Classes for Fast Maximum Entropy Training. In *Proc. of ICASSP*.

Kenneth Heafield. 2011. KenLM: Faster and Smaller Language Model Queries. In *Workshop on Statistical Machine Translation*, pages 187–197.

Michael Gutmann Aapo Hyvärinen. 2010. Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. In *Proc. of AISTATS*.

Sébastien Jean, Kyunghyun Cho, Roland Memisevic, and Yoshua Bengio. 2014. On Using Very Large Target Vocabulary for Neural Machine Translation. *CoRR*, abs/1412.2007.

Rafal Jozefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. 2016. Exploring the limits of language modeling. Technical Report arXiv 1602.02410.

Hai-Son Le, Alexandre Allauzen, and François Yvon. 2012. Continuous Space Translation Models with Neural Networks. In *Proc. of HLT-NAACL*, pages 39–48, Montréal, Canada. Association for Computational Linguistics.

Remi Lebret and Ronan Collobert. 2014. Word Embeddings through Hellinger PCA. In *Proc. of EACL*.

Yann LeCun, Leon Bottou, Genevieve Orr, and Klaus-Robert Mueller. 1998. Efficient BackProp. In Genevieve Orr and Klaus-Robert Muller, editors, *Neural Networks: Tricks of the trade*. Springer.

Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. 1993. Building a Large Annotated Corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):314–330, Jun.

Tomáš Mikolov, Karafiát Martin, Lukáš Burget, Jan Cernocký, and Sanjeev Khudanpur. 2010. Recurrent Neural Network based Language Model. In *Proc. of INTERSPEECH*, pages 1045–1048.

Tomáš Mikolov, Anoop Deoras, Stefan Kombrink, Lukas Burget, and Jan Honza Cernocky. 2011a. Empirical Evaluation and Combination of Advanced Language Modeling Techniques. In *Interspeech*.

Tomáš Mikolov, Stefan Kombrink, Lukáš Burget, Jan Cernocký, and Sanjeev Khudanpur. 2011b. Extensions of Recurrent Neural Network Language Model. In *Proc. of ICASSP*, pages 5528–5531.

Tomáš Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. *CoRR*, abs/1301.3781.

Andriy Mnih and Geoffrey E. Hinton. 2010. A Scalable Hierarchical Distributed Language Model. In *Proc. of NIPS*.

Andriy Mnih and Yee Whye Teh. 2012. A fast and simple algorithm for training neural probabilistic language models. In *Proc. of ICML*.

Frederic Morin and Yoshua Bengio. 2005. Hierarchical Probabilistic Neural Network Language Model. In *Proc. of AISTATS*.

Robert Parker, David Graff, Junbo Kong, Ke Chen, and Kazuaki Maeda. 2011. English Gigaword Fifth Edition. Technical report, Linguistic Data Consortium.

Jeffrey Pennington, Richard Socher, and Christopher D Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the Empiricial Methods in Natural Language Processing*.

Holger Schwenk, Anthony Rousseau, and Mohammed Attik. 2012. Large, Pruned or Continuous Space Language Models on a GPU for Statistical Machine Translation. In *NAACL-HLT Workshop on the Future of Language Modeling for HLT*, pages 11–19. Association for Computational Linguistics.

Alessandro Sordoni, Michel Galley, Michael Auli, Chris Brockett, Yangfeng Ji, Margaret Mitchell, Jian-Yun Nie1, Jianfeng Gao, and Bill Dolan. 2015. A Neural Network Approach to Context-Sensitive Generation of Conversational Responses. In *Proc. of NAACL*. Association for Computational Linguistics, May.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*.

Ashish Vaswani, Yinggong Zhao, Victoria Fossum, and David Chiang. 2013. Decoding with Large-scale Neural Language Models improves Translation. In *Proc. of EMNLP*. Association for Computational Linguistics, October.