Optimizing recurrent neural network language model GPU training

Marcin Müller

Master of Science
Artificial Intelligence
School of Informatics
University of Edinburgh
2017

Abstract

In recent years neural language models have consistently out-performed n-gram based statistical language models. Due to their large computational cost the adaptation in production-scale ASR systems is slow. The main bottleneck is the output softmax layer whose computational cost scales linearly with the vocabulary size. This work summarized the large body of research in this field and compared the recently proposed adaptive softmax method to sampling based approaches. An empirical analysis investigated various properties of the adaptive softmax, importance sampling and noise contrastive estimation methods. Experiments were conducted using Tensorflow on the Wikipedia Text8 corpus and a large vocabulary financial domain text. The results show that all three methods can substantially increase training throughput. Adaptive softmax and importance sampling both achieved low perplexity scores on par with full softmax and no noticeable difference could be measured on an ASR re-scoring task. Noise contrastive estimation proved to be sensitive to the chosen normalization constant and in many experiments could not achieve optimal results. The sharing of noise samples within a batch was described as a possible cause. Profiling showed that projection matrices substantially reduce the amount of trainable parameters and memory footprint in adaptive softmax. In other benchmarks switching to the optimized NVIDIA cuDNN LSTM implementation achieved an additional speedup.

Acknowledgements

I would like to extend my most sincere gratitude to my joint supervisors, Prof. Steve Renals and Daniel Renshaw, for their time, valuable advice and guidance and to Quorate Technology Ltd., for the provision of resources without which this project would have not been possible.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Marcin Müller*)

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Language models are an important and integral part of speech recognition and machine translation systems. In recent years neural language models consistently outperformed the previous generation of statistical language models (Mikolov et al., 2010; Sundermeyer et al., 2012; Jozefowicz et al., 2016). But the transition from research into real-word applications is hindered by their large training cost. State-of-the-art models are often trained on a large computing cluster over a period of several weeks (Jozefowicz et al., 2016). The trend towards large datasets with vocabulary sizes exceeding hundreds of thousand words (Chelba et al., 2013) only aggravates the problem. The main bottleneck is the softmax calculation in the output layer whose cost scales linearly with the vocabulary size (Bengio et al., 2003a). Not surprising, a large amount of research effort has been devoted over the last one and a half decades to the engineering of viable alternatives to the full softmax.

The main objective of this project was to critically review recent research related to training large vocabulary neural language models. Emphasis was put on techniques that enable fast training on single graphics processing unit (GPU) systems as access to large scale computing clusters is a resource often not readily available to researchers in academia.

The contribution of this work is two-fold: First, an extensive literature review summarizes the large body of research into alternative softmax methods. Second, an empirical analysis contrasts the recently proposed adaptive softmax method (Grave et al., 2016) with more established sampling based alternatives. Experiments are conducted on two English language corpora with medium and large vocabularies. All models use long short-term memory (LSTM) recurrent neural

network (RNN) and are implemented in Tensorflow. Additionally, an alternative LSTM implementation with GPU specific optimizations was benchmarked.

The rest of this report is structured as follows. Chapter 2 contains a brief introduction to statistical and neural language models which is followed by a comprehensive literature review of alternative methods designed to speed up computation in the softmax output layer. Chapter 3 describes the implementations of the investigated methods and highlights some important implementation details. Chapter 4 introduces the datasets used and formalizes the training procedure. Chapter 5 describes the conducted experiments and presents the results. Accompanying discussions critically analyze the results. Finally, Chapter 6 concludes this work.

Chapter 2

Background and literature review

2.1 Language models

Sentences in natural languages have structure and adhere to grammatical rules. Certain word combinations tend to occur together more frequently. A language model captures such regularities within a language. It can assign a probability to a sequence of characters or words. Language models are often used as a building block within natural language processing (NLP). Applications include automatic speech recognition (ASR) and machine translation (MT) where language models rank and select the most natural sounding phrases. Furthermore, language models help to resolve various kinds of ambiguity pervasive in natural languages. In information retrieval (IR) language models are used to estimate the probability of query sentences. In natural language generation (NLG) possible continuations are chosen according to the probabilities assigned by the language model.

2.1.1 Statistical language models

A statistical language model is a valid probability distribution over all possible sentences. The probabilities of the language model are usually estimated from a training corpus. Limited data and an infinite number of sentences make it impossible to directly estimate sentence probabilities. Instead, a sentence S is represented as a finite word sequence $w_1, w_2, w_3, ..., w_{|S|}$. Then, the probability of a sentence S can be decomposed into conditional word probabilities by application

of the chain rule

$$\begin{aligned} P(S) &= P(w_1, w_2, ..., w_{|S|}) \\ &= P(w_1)P(w_2|w_1)P(w_3|w_1, w_2)P(w_{|S|}|w_1, w_2, ..., w_{|S|-1}) \\ &= \prod_{t=1}^{|S|} P(w_t|w_1^{t-1}). \end{aligned} \tag{2.1}$$

This decomposition enables estimation of conditional word probabilities with short context sequences. Conditional probabilities with longer contexts still suffer from the curse of dimensionality where the amount of required data grows exponentially with the context length. A common solution is to *incorrectly* assume that the probability of a word is only dependent on its immediate context and not words at the start of the sentence. Such short sequences of n words are referred to as n-grams. Under this assumption Equation 2.1 simplifies to

$$P(S) \approx \prod_{t=1}^{|S|} P(w_t|w_{t-n+1}^{t-1}). \tag{2.2}$$

A straightforward method to estimate the conditional probabilities is maximum likelihood estimation (MLE). Here, the probabilities are obtained with frequency tables that store word counts for different n-gram contexts.

$$P(w_t|w_{t-n+1}^{t-1}) = \frac{count(w_{t-n+1}^{t})}{\sum\limits_{w' \in \mathcal{V}} count(w_{t-n+1}^{t-1}, w')} = \frac{count(w_{t-n+1}^{t})}{count(w_{t-n+1}^{t-1})}. \tag{2.3}$$

Due to a finite training corpus and the curse of dimensionality a majority of possible n-gram contexts w_{t-n+1}^{t-1} will not occur in the training data when $n > 2$. MLE will assign zero probabilities to sentences including such contexts. This outcome is not desirable as it prevents meaningful comparison of sentences. A common solution is to explicitly enforce non-zero probabilities. A variety of smoothing methods have been invented which redistribute some probability mass to events that were not observed in the training data. The most simple smoothing method is add-α where a small constant α is added to each word count.

$$P(w_t|w_{t-n+1}^{t-1}) = \frac{count(w_{t-n+1}^{t}) + \alpha}{\sum\limits_{w' \in \mathcal{V}} count(w_{t-n+1}^{t-1}, w') + \alpha} = \frac{count(w_{t-n+1}^{t}) + \alpha}{count(w_{t-n+1}^{t-1}) + \alpha|\mathcal{V}|}. \tag{2.4}$$

When $\alpha = 1$ the method is also know as add-1 smoothing. Add-α smoothing is rarely used in practice because even a small α redistributes a large probability mass $\alpha|\mathcal{V}|$ that noticeably distorts the distribution. A more common smoothing

method is the interpolation of language models with various n-gram sizes (Jelinek, 1980).

$$P(w_t|w_{t-n+1}^{t-1}) = \sum_{i=1}^{n} \lambda_i P_n(w_t|w_{t-i+1}^{t-1}) \tag{2.5}$$

The mixture weights λ_i specify the importance of each language model and must sum to 1. Some variations allow the mixture weights to be set independently for each word w_t. Other methods are based on back-off (Katz, 1987) where higher n-gram language models are used as long as there is sufficient data for a given context. Otherwise, the system backs off to a lower n-gram. To ensure valid probability distributions back-off systems require more complex normalization. A comprehensive survey of different smoothing methods was conducted by Chen and Goodman (1996). An interpolated version of Kneser-Ney's absolute discounting (Kneser and Ney, 1995) was identified as the overall best method.

2.1.2 Neural language models

In statistical language models words are represented by discrete symbols and probabilities have to be estimated for each word independently. In practice, words are highly interconnected (Kilgarriff, 2000) and multiple words can express a similar meaning. The discrete nature of word symbols does not allow to easily leverage such information. Generalization from common to less frequent words with similar meaning is not possible. This adds to the sparse data issue of long n-gram models. In addition, the amount of memory required by large n-gram models can be prohibitively expensive.

Bengio et al. (2003a) proposed to use neural networks to learn the output word probabilities. Usually, discrete word symbols need to be converted into 1-hot $|\mathcal{V}|$ dimensional boolean vectors before they are fed into a neural network. An alternative continuous representation are word embeddings which represent words as continuous vectors $\psi(w) \in \mathcal{R}^d$. Word embeddings can be interpreted as a projection of word symbols onto the subspace $\mathcal{R}^d$. Related words tend to have similar embedding vectors. Mikolov et al. (2013a,b) demonstrated that semantic meaning can be attributed to certain directions in the embedding space.

The proposed neural probabilistic language model (NPLM) architecture by Bengio et al. (2003a) consists of a single layer feed-forward neural network that maps a series of input word embeddings to a vector of output scores in $\mathcal{R}^{|\mathcal{V}|}$. The input is formed by concatenating the embedding vector $\psi(w)$ for each word in

an n-gram context H. The hidden layer has a non-linear activation function and computes an embedding $\phi(H)$ for the predicted word. The output layer compares $\phi(H)$ to each word embedding $\psi(w)$ over the entire vocabulary. The similarity is measured by the dot product $\phi(H)^T \cdot \psi(w)$. Larger scores indicate a greater likelihood of the output word.

In practice, the word embeddings are stored as a matrix Ψ and the output scores s_θ are obtained by

$$\underbrace{\begin{bmatrix} s_\theta(w_1, H) \\ s_\theta(w_2, H) \\ \vdots \\ s_\theta(w_{|\mathcal{V}|}, H) \end{bmatrix}}_{s_\theta(H)} = \underbrace{\begin{bmatrix} \psi(w_1)_1^T & \psi(w_1)_2^T & \dots & \psi(w_1)_d^T \\ \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \\ \psi(w_{|\mathcal{V}|})_1^T & \psi(w_{|\mathcal{V}|})_2^T & \dots & \psi(w_{|\mathcal{V}|})_d^T \end{bmatrix}}_{\Psi} \underbrace{\begin{bmatrix} \phi(H)_1 \\ \phi(H)_2 \\ \vdots \\ \phi(H)_d \end{bmatrix}}_{\phi(H)}$$

where θ denotes the parameters of the model. Often separate embedding matrices are used in the input and output layers.

The output scores can be transformed into valid probabilities using the softmax function. The probability distribution over the next word w_t is given by

$$P_\theta(w_t|H) = \frac{e^{s_\theta(w_t, H)}}{\sum\limits_{w' \in V} e^{s_\theta(w', H)}}. \tag{2.6}$$

The term in the denominator is a normalizing constant that ensures P_θ is valid probability density function where $\sum_{w \in \mathcal{V}} P_\theta(w|H) = 1$.

2.1.3 RNN language models

Feed-forward neural networks are stateless models that can approximate arbitrary functions given enough parameters (Bishop, 1995). RNNs are a state-full extension where connections are allowed to point back in time and form delay loops. The recurrent connections enable the network to access activations at previous time steps thus maintaining an internal state over time. The output h_t at time step t of an RNN network can be computed as

$$h_t = g([W_{hh} W_{hx}][h_{t-1}^T x_t^T]^T + b_h) = g(W_{hh} h_{t-1} + W_{hx} x_t + b_h) \tag{2.7}$$

where x_t represents the current input vector, h_{t-1} is network's output at the previous time step and W_{hx}, W_{hh} are weight matrices for the forward and recurrent

connection, respectively (Elman, 1990). $g(.)$ is a point-wise non-linear function, i.e. $tanh(.)$

Arbitrarily long input sequences can be processed in a serial fashion by presenting one token at a time. This property removes the limitation of finite context lengths in n-gram models and NPLMs. In theory, RNN-based language models are capable of learning long-range dependencies and therefore seem to be in a better position for approximating the probability $P(w_t|w_1^{t-1})$ in Equation 2.1. Mikolov et al. (2010, 2011) reported the first successful application of RNNs to the task of language modelling.

For a long time a major obstacle to successful applications of RNNs was the lack of adequate training procedures. Rumelhart et al. (1985) and Werbos (1988) noticed that a RNN can be treated like a standard feed-forward neural network if the recurrent connections are unrolled over time. Then, the unrolled feed-forward network can be trained with standard gradient back-propagation methods. This technique is referred to as back-propagation through time (BPTT). In practice, truncated BPTT is used to restrict the amount of computation required. In truncated BPTT the number of steps a gradient signal is propagated back through time is artificially limited (Zipser, 1990).

Even with BPTT, RNNs were often not capable of learning dependencies that span more than a few time steps. Bengio et al. (1994) showed that during BPTT the gradient signal is repeatedly multiplied with the same weight matrix W_{hh}. After a few time steps this can result in vanishing or exploding gradients. Different solutions were proposed for both conditions. Exploding gradients can be prevented by constraining or *clipping* the gradient signal (Pascanu et al., 2012; Bengio et al., 2013). To combat vanishing gradients Hochreiter and Schmidhuber (1997) proposed a more elaborate design by introducing an additional cell state that can flow unmodified through many time steps. Modifications to this cell state are performed exclusively through special gates that allow only some parts of the input through. This RNN design is referred to as long short-term memory (LSTM). Over the last two decades a plethora of modifications and alternative designs was introduced. Greff et al. (2016) chronologically listed the major contributions that led to the current LSTM cell implementation. Noticeable innovations include the forget gate (Gers et al., 1999) and peephole connections (Gers and Schmidhuber, 2000). A more radical change was introduced with the gated recurrent unit (GRU) (Cho et al., 2014). GRUs remove the additional cell

state and employ only gates to tackle the vanishing gradient problem leading to a simplified design.

RNNs language models were applied to ASR (Graves et al., 2013) and are an integral part of neural MT models (Sutskever et al., 2014). Another popular application of RNNs language models can be found in the field of NLG (Sutskever et al., 2011; Graves, 2013; Karpathy, 2016). Sundermeyer et al. (2012) measured the advantage of LSTMs compared to standard RNNs on language modeling tasks. They report that using a LSTM layer improved the perplexity by 8%. Additionally, interpolating the LSTM-based language model with a large n-gram model yields an additional improvement (Sundermeyer et al., 2012; Jozefowicz et al., 2016).

2.2 Alternative softmax methods

Bengio et al. (2003a) identified the softmax computation (Equation 2.6) in the output layer as the main bottleneck in neural language models. The computation of $P_\theta(w|H)$ for a single word needs to evaluate scores over the entire vocabulary. The same issue is also present in the gradient calculation. Thus, the computation cost during inference and training scales with the size of the vocabulary. With vocabulary sizes approaching the order of a million words in recent corpora (Parker et al., 2011; Chelba et al., 2013), computing the full softmax during training becomes prohibitively expensive.

This problem has attracted substantial research efforts over the last one and a half decades which resulted in a sizable number of publications. The following literature review presents a systematic review of this work. Many of the established methods can be classified into one of two main categories. Factorization techniques divide the vocabulary into smaller groups which can be evaluated independently. Sampling based techniques estimate the loss gradient or word probability using a small randomly chosen subset of the entire vocabulary. In related work, Ruder (2016) compiled an informal review of various alternative methods. The literature review presented here extends this work and provides a more detailed approach. An attempt is made to systematically cover the published research in this field.

The literature review is structured as follows. First, shortlists are introduced which are the simplest solution for reducing the amount of computation. Next,

class-based and hierarchical factorization techniques are discussed including the recent differentiated and adaptive softmax variations. The section on sampling methods covers importance sampling and noise contrastive estimation in great detail. Finally, current research into other alternative methods is presented.

2.2.1 Shortlist

In the initial work on NPLM (Bengio et al., 2003a) the vocabulary was reduced to a manageable size by replacing all words that occurred less than 4 times in the training corpus with a special out-of-vocabulary (OOV) token. Subsequent work (Schwenk, 2004; Schwenk and Gauvain, 2004, 2005) investigated the utility of NPLMs in ASR decoding and lattice rescoring. To reduce the computational cost of training and predictions at evaluation time the NPLM output layer was reduced to a small shortlist $\mathcal{V}_S$ containing only the most frequent words. Due to the Zipfian nature of natural language (Zipf, 1949), a shortlist of 2000 words is sufficient to cover $\approx 89\%$ of a $65k$ vocabulary (Schwenk and Gauvain, 2004). Similar coverage was achieved with a $12k$ shortlist when the vocabulary size was increased to $200k$ (Schwenk, 2007). Subsequently, the reduced NPLM output was combined with a standard backoff n-gram model. NPLM output predicted the probabilities of the shortlisted words and the n-gram language model was consulted for the rest of the vocabulary. Because the NPLM output does not cover the entire vocabulary, its output probability cannot be used directly. Instead, the probability of a word w given the n-gram context H is computed as follows

$$P(w|H) = \begin{cases} P_{NN}(w|H)P_{\mathcal{V}_S}(H) & \text{if } w \in \mathcal{V}_S \\ P_{NGRAM}(w|H) & \text{otherwise} \end{cases} \tag{2.8}$$

where P_{NN} and P_{NGRAM} are the probabilities computed by the NPLM and n-gram model, respectively and

$$P_{\mathcal{V}_S}(H) = \sum_{w \in \mathcal{V}_S} P_{NGRAM}(w|H). \tag{2.9}$$

In the above Equations the NPLM probabilities are scaled by the n-gram probability mass of the shortlisted words $P_{\mathcal{V}_S}(H)$. This approach was criticized by Park et al. (2010) who argues that it creates a bias towards shortlisted words and fails to utilize the portions of training data that contains out-of-shortlist (OOS) words. To alleviate the problem, Park et al. (2010) proposed to add an explicit

OOS node to the NPLM output layer. Then, the merged word probabilities are computed by

$$P(w|H) = \begin{cases} P_{NN}(w|H) & \text{if } w \in \mathcal{V}_S \\ \dfrac{P_{NGRAM}(w|H)}{\sum\limits_{w' \notin \mathcal{V}_S} P_{NGRAM}(w'|H)} P_{NN}(w_{OOS}|H) & \text{otherwise.} \end{cases} \tag{2.10}$$

To avoid costly queries to the NPLM at decoding time Equation 2.8 can be used to prepare a modified n-gram model within a preprocessing stage (Schwenk, 2004). Then, the new n-gram model is used during decoding without any overhead.

2.2.2 Factorization

The idea of factorization of word probabilities was first explored by Brown et al. (1992) in the context of statistical language models. The motivation was to exploit similarities between words to improve predictions for rare words. When each word w_i is mapped to a unique class c_{w_i}, the conditional word probability can be decomposed into

$$P(w_i|H) = P(w_i|c_{w_i})P(c_{w_i}|H) \tag{2.11}$$

where $P(c_{w_i}|H)$ is the conditional probability of class c_{w_i} given the current history H and $P(w|c_{w_i})$ is the unigram probability of that word within its class. This factorization is only valid when the classes are mutually exclusive which implies $P(w_i|c_j) = 0$ for all $w_i \notin c_j$. Morin and Bengio (2005) showed that any valid clustering scheme $c(w_i)$ will yield sound probabilities but might effect the performance and generalization properties of the language model.

$$\begin{aligned} P(w_i|H) &= \sum_i P(w_i, c_i|H) \\ &= \sum_i P(w_i|c_i, H)P(c_i|H) \\ &= P(w_i|c(w_i), H)P(c(w_i)|H) \end{aligned} \tag{2.12}$$

Brown et al. (1992) investigated several statistical techniques based on mutual information for grouping words into syntactic or semantic classes. No improvement in perplexity could be achieved but using word classes reduced the storage

requirements of n-gram models. Kneser and Ney (1993) used maximum likelihood to find good class assignments which minimized the overall perplexity of the language model.

2.2.2.1 Class-based softmax

Goodman (2001) applied the mechanics of class-based factorization to log-linear language models with the intent of reducing the computation in the softmax output layer. But instead of using the unigram probability $P(w_i|c_{w_i})$ of words in a given cluster, Goodman (2001) trained two separate log-linear models where each model was conditioned on the current context H. Each model computed $P(c_{w_i}|H)$ and $P(w_i|c_{w_i}, H)$, respectively. The final output was computed as

$$P(w_i|H) = P(w_i|c_{w_i}, H)P(c_{w_i}|H). \tag{2.13}$$

Goodman (2001) observed that the maximum speed up achieved by this technique is $\mathcal{O}(\frac{|\mathcal{V}|}{\sqrt{|\mathcal{V}|}}) = \mathcal{O}(\sqrt{|\mathcal{V}|})$ when the vocabulary is divided evenly into $\sqrt{|\mathcal{V}|}$ classes. Mikolov et al. (2011) applied a similar idea to training of RNN language models. Instead of training two separate networks the same RNN output was used to compute both conditional probabilities $P(w_i|c_{w_i}, H)$ and $P(c_{w_i}|H)$. This was achieved by having multiple softmax output layers on top of the RNN layer, one for computing class probabilities $P(c_{w_i}|H)$ and the rest for computing the conditional word probabilities $P(w_i|c_{w_i}, H)$. During training the gradients of the individual output layers are combined before being back-propagated through the RNN.

Mikolov et al. (2011) described a simple but efficient scheme for assigning words to clusters. In frequency binning the vocabulary is first sorted by frequency and then words are added to a class until its probability mass exceeds $\frac{1}{|C|}$ where $|C|$ is the number of classes. Due to Zipf's law (Zipf, 1949) the first class might only contain a single word (e.g. `the`) whereas classes representing rare words will contain thousands of members. The total number of classes can be empirically determined by cross validation using a held-out dataset. A variant of this technique uses the square root of the frequencies (Zweig and Makarychev, 2013).

In speed regularized likelihood classing Zweig and Makarychev (2013) proposed to add a regularization term to the maximum likelihood technique used in class-based statistical language models (Kneser and Ney, 1993). The aim of the

regularization term is to penalize class assignments which increase the computational cost.

2.2.2.2 Hierarchical softmax

The maximum theoretical speed up of $\mathcal{O}(\sqrt{\mathcal{V}})$ achieved by class-based factorization can be improved by further decomposition. Morin and Bengio (2005) developed this idea to its extreme and proposed a binary tree decomposition of the softmax layer. Intermediate tree nodes can be interpreted as a taxonomy of binary concepts that categorize words contained in the leaves. The probability of a word w_i is then equivalent to the probabilities along a path through the tree from the root node to the corresponding leave node. This can also be thought of as a sequence of $D = \lceil \log_2 |\mathcal{V}| \rceil$ binary decisions where each decision has the probability $P(c_d|c_{d-1}, H)$. Hence, the path from the root to each word can be stored as a D dimensional bit vector. The output probability is the product of all probabilities along a path.

$$\begin{aligned} P_\theta(w_i|H) &= P_\theta(c_1(w_i)|H) \prod_{d=2}^{D} P_\theta(c_d(w_i)|c_{d-1}(w_i), H) \\ &= \prod_{d=1}^{D} \sigma(s_\theta(c_d(w_i), H) \end{aligned} \tag{2.14}$$

Each binary decision is computed using the logistic function $\sigma(x) = \frac{1}{1+e^{-x}}$. The theoretical speed up obtained with this decomposition is $\mathcal{O}(\frac{|\mathcal{V}|}{\log_2 |\mathcal{V}|})$ and grows exponentially with the vocabulary size.

A major part of any hierarchical method is the clustering of words. Morin and Bengio (2005) used the IS-A taxonomy in WordNet (Kilgarriff, 2000) as the initial tree structure. The binary tree constraint was ensured by splitting up tree nodes with more than two children. This was done by K-means clustering based on a TF/IDF similarity metric. Empirical experiments showed that the proposed hierarchical decomposition achieved a ~ 200 fold speed up but could not match the perplexity of the full softmax, especially on small vocabulary datasets. Further, the exact choice of the clustering algorithm has a large impact on the achieved perplexity of the model (Morin and Bengio, 2005; Mnih and Hinton, 2009; Zweig and Makarychev, 2013).

Mnih and Hinton (2009) devised an alternative word clustering algorithm that is purely data-driven without any dependencies to external linguistic resources. The main idea is to recursively split a list of words into two classes. In

the following clustering algorithm words are represented by continuous feature vectors that are first computed during a bootstrapping phase. After training a model using a random binary tree, word representations are computed by averaging the output of the neural network layer $\phi(H)$ over all possible n-gram contexts $H = w_{t-n+1}^{1}$ that can precede a word. The clustering is then performed using a Gaussian mixture model with 2 mixtures that is trained with the expectation-maximization (EM) algorithm (Dempster et al., 1977). Words are recursively subdivided into one of two classes based on the mixture coefficients. Due to the fixed size of word representations $\mathcal{R}^d$ the run time of the clustering algorithm is $\mathcal{O}(|\mathcal{V}|)$. Mnih and Hinton (2009) experimented with 3 different strategies for constructing the binary tree. A perfectly balanced tree can be obtained by ranking words according to their mixture weights and inserting the split point in the middle. This approach will assign words to a cluster regardless of the respective Gaussian component weight. An alternative approach is to assign words to clusters where the mixture weight is > 0.5 which will produce unbalanced trees. Mnih and Hinton (2009) also tried assigning a word to both clusters if there is no clear preference towards one of the classes. As expected the highest speed up was achieved with perfectly balanced trees but empirical results showed that lower perplexity could be achieved with clustering schemes that put more emphasis on the mixture weights.

The speed up can be further maximized by taking word frequencies into account. Mikolov et al. (2013a) used a binary Huffman tree that assigns shorter codes (and thus shorter paths) to more frequent words. Mikolov et al. (2013a) compared hierarchical softmax against other sampling based methods on an word analogy benchmark and found that hierarchical softmax gave a similar speed up but could not match the accuracy of the sampled methods.

Le et al. (2011a,b) proposed a more general hierarchical decomposition of the output layer which removes the binary tree constraint of previous hierarchical softmax methods (Morin and Bengio, 2005; Mnih and Hinton, 2009). The structured output layer (SOUL) can be seen as an extension of class-based factorization (Goodman, 2001; Mikolov et al., 2011) with additional layers. New layers are introduced by splitting up each class into subclasses which can be broken down further into sub-subclasses and so on up to an arbitrary depth D. If the depth is constrained to 2, the architecture is equivalent to class-based factorization. Splitting each class recursively into two subclasses yields hierarchical

softmax. Analogous to hierarchical softmax, nodes within intermediate layers represent word classes and nodes in the bottom layer individual words. Unlike in hierarchical softmax with binary trees, the probability of nodes within a cluster is computed using the softmax function. Again, the probability of a word can be computed as the product of all class probabilities along the path through the tree.

$$P(w_i|H) = P(c_1(w_i)|H)(\prod_{d=2}^{D-1} P(c_d(w_i)|c_{d-1}(w_i), H))P(w_i|c_{D-1}(w_i), H) \quad (2.15)$$

A further innovation in SOUL is the direct inclusion of a small set of frequent words within the root layer. This is similar to the notion of shortlists (Schwenk, 2004) and avoids the loss in precision due to accumulating errors along the path.

Le et al. (2011a,b) introduced an alternative clustering algorithm that is based on word embeddings. The first step is to train the language model using only the vocabulary in the shortlist. Next, the input word embeddings are extracted and reduced in dimensionality using principle component analysis (PCA). The hierarchy of word clusters is computed using a recursive application of K-means clustering. SOUL with 3 layers has been applied to MT (Le et al., 2012) and ASR (Le et al., 2013).

2.2.2.3 Differentiated softmax

Traditionally, the same dimension d is used for all output embedding vectors $\psi(w) \in \mathcal{R}^d$. The output scores in the softmax output layer are computed by a large matrix-vector multiplication $W \times h$ where W is the word embedding matrix and h the output of the neural network layer $\phi(H)$. As previously mentioned the cost of this operation is $\mathcal{O}(d \times V)$. Chen et al. (2015a) argued that because rare words have by definition fewer training examples the number of parameters assigned to those words should be reduced. This can avoid issues with over-fitting the data and lower the memory and computational cost. Chen et al. (2015a) designed a novel variation of the output layer called differentiated softmax. The output of the neural network layer $h = \phi(H) \in \mathcal{R}^d$ is partitioned into several blocks h_i of various dimensions d_i such that $\sum_i d_i = d$. Frequent words are assigned to wider blocks whereas rare words are put into narrow blocks. Word scores over the whole vocabulary are computed efficiently by concatenating the results of the smaller matrix-vector multiplications $W_i \times h_i$. The total cost of these operations is $\mathcal{O}(\sum_i d_i \times V_i)$ which can be noticeably smaller than $\mathcal{O}(d \times V)$. It should be noted

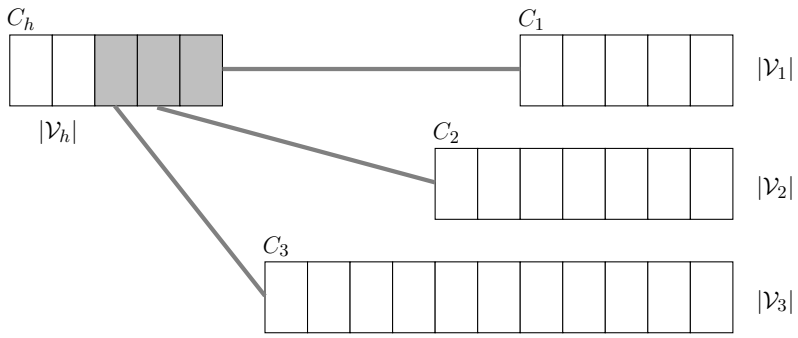


Figure 2.1: Illustration of adaptive softmax including the shortlist $|\mathcal{V}_h|$ and tail clusters containing less frequent words (Grave et al., 2016).

that the speed up obtained by differentiated softmax applies not only during training but also at test and evaluation time.

Chen et al. (2015a) carried out an empirical comparison of different softmax methods and reported that differentiated softmax achieved the lowest perplexity on the Gigaword corpus (Parker et al., 2011) and the second lowest perplexity on the one billion word benchmark (Chelba et al., 2013). A closer inspection revealed that high frequency words were predicted with the highest accuracy compared to other methods whereas perplexity of words in the smallest partition was the worst.

Grave et al. (2016) suggested a modification that instead of partitioning the neural network output $\phi(H)$ uses projection matrices to reduce the number of the parameters. In this approach h_i is computed by projecting the full neural network output by a matrix W_{P_i}. Grave et al. (2016) argues that this approach allows to use the full neural network layer capacity in the score computation of each block.

2.2.2.4 Adaptive softmax

Grave et al. (2016) proposed a different variant of class-based factorization that was optimized for execution on a GPU. The overall structure is based on class-based factorization (Mikolov et al., 2011) but includes a shortlist of the most frequent words directly in the root cluster (see Figure 2.1). Thus, it is equivalent to SOUL with just 2 layers.

In more detail, adaptive softmax divides the vocabulary $\mathcal{V}$ into a head cluster $\mathcal{V}_h$ and T tail clusters $\mathcal{V}_t$. The probability of a word within $\mathcal{V}_h$ can be computed directly using the softmax of the head cluster logits s_h. For words assigned to one

of the tail clusters the probability is broken down as the product of two factors: the probability of the tail cluster itself $P(c_t|H)$ and the probability of the word w_i within the respective tail cluster $P(w_i|c_t, H)$. To be able to compute $P(c_t|H)$ the head cluster contains additional T logits, one for each tail cluster $\mathcal{V}_t$. The following equation summarizes the process of computing the word probability $P(w|H)$ from the cluster logits $s_h, s_1, ..., s_T$.

$$P(w|H) = \begin{cases} P(w|H), & \text{if } w \in \mathcal{V}_h \\ P(w|c_t, H)P(c_t|H), & \text{if } w \in \mathcal{V}_t \end{cases}$$
$$= \begin{cases} \dfrac{e^{s_{h,\theta}(w,H)}}{\sum\limits_{w' \in \mathcal{V}_h} e^{s_{h,\theta}(w',H)} + \sum\limits_{t=1}^{T} e^{s_{h,\theta}(c_t,H)}} \\ \dfrac{e^{s_{h,\theta}(c_t,H)}}{\sum\limits_{w' \in \mathcal{V}_h} e^{s_{h,\theta}(w',H)} + \sum\limits_{t=1}^{T} e^{s_{h,\theta}(c_t,H)}} \dfrac{e^{s_{t,\theta}(w,H)}}{\sum\limits_{w' \in \mathcal{V}_t} e^{s_{t,\theta}(w',H)}} \end{cases} \tag{2.16}$$

Grave et al. (2016) observed that the cost of matrix-matrix multiplication on a GPU does not scale linearly for small matrices. They showed that for matrices with sizes $|\mathcal{B}| \times h$ and $h \times k$ the cost g grows linearly with k but is a constant overhead o if k is below a certain inflection point k_0.

$$g(k) = o + \max(0, \lambda(k - k_0)) \tag{2.17}$$

The constants λ, o and k_0 can be determined experimentally for a specific GPU. Thin weight matrices $\mathcal{R}^{d \times \mathcal{V}_k}$ where $|\mathcal{V}_k| < k_0$ often occur in frequency binning schemes for clusters containing the highest ranked words. This suggests that frequency binning is not an efficient partitioning scheme when training on GPUs (Grave et al., 2016).

The probability of a random word w belonging to the tail cluster $\mathcal{V}_t$ is $p_t = \sum_{w \in \mathcal{V}_t} P_u(w)$ where P_u is the unigram probability. The average cost C of adaptive softmax over a training dataset can be calculated as

$$C = g(|\mathcal{V}_h| + T) + \sum_{t=1}^{T} p_t g(|\mathcal{V}_t|). \tag{2.18}$$

Assuming all cluster sizes are greater than k_0, the above cost can be simplified using a linear version of $g(k) = o + \lambda k$ to

$$C = 2o + \lambda(|\mathcal{V}_h| + T + \sum_{t=1}^{T} p_t |\mathcal{V}_t|). \tag{2.19}$$

Assuming two tail clusters $\mathcal{V}_i$ and $\mathcal{V}_j$ have fixed sizes and $|\mathcal{V}_i| > |\mathcal{V}_j|$, the cost C depends only on

$$p_i|\mathcal{V}_i| + p_j|\mathcal{V}_j| = p_i|\mathcal{V}_i| + (p_{i+j} - p_i)|\mathcal{V}_j| = p_i(|\mathcal{V}_i| - |\mathcal{V}_j|) + p_{i+j}|\mathcal{V}_j| \tag{2.20}$$

where $p_{i+j} = p_i + p_j$. The only variable quantity in the equation above is the probability of the larger tail cluster p_i when taking the fixed size constraint into account. The cost can be minimized by reducing p_i which is equivalent to assigning the least frequent words to $\mathcal{V}_i$. Grave et al. (2016) suggests that the optimal cluster sizes can be determined using a dynamic programming approach or estimated empirically.

A batched version of adaptive softmax can be implemented by creating distinct subsets $\bigcup_{t=1}^{T} \mathcal{B}_t = \mathcal{B} - \mathcal{B}_h$ for each tail cluster. The expected size of $\mathcal{B}_t$ is $\mathbb{E}_{P_u}[|\mathcal{B}_t|] = p_t|\mathcal{B}|$. This can also be seen as an example of conditional computation. The full batch $\mathcal{B}$ is always used in the logit computation of the head cluster where for words belonging to a tail cluster $w \in \mathcal{V}_t$ the embedding $\psi(c_t)$ of the respective cluster is used.

2.2.3 Sampling

Based on prior work on contrastive divergence (Hinton, 2002), Bengio et al. (2003b) showed that the gradient of the NPLM negative log-likelihood can be approximated using sampling methods. The gradient can be decomposed into two terms as follows:

$$\begin{aligned}
\nabla_\theta NLL(\theta) &= \nabla_\theta[-\log P_\theta(w_t|H)] \\
&= \nabla_\theta[-\log \frac{e^{s_\theta(w_t,H)}}{\sum\limits_{v\in\mathcal{V}} e^{s_\theta(v,H)}}] \\
&= \nabla_\theta[-s_\theta(w_t,H) + \log \sum_{v\in\mathcal{V}} e^{s_\theta(v,H)}] \\
&= -\nabla_\theta(s_\theta(w_t,H)) + \frac{1}{\sum\limits_{v\in\mathcal{V}} e^{s_\theta(v,H)}} \sum_{v\in\mathcal{V}} e^{s_\theta(v,H)} \nabla_\theta(s_\theta(v,H)) \\
&= -\nabla_\theta(s_\theta(w_t,H)) + \sum_{v\in\mathcal{V}} P_\theta(v|H)\nabla_\theta(s_\theta(v,H)) \\
&= -\nabla_\theta(s_\theta(w_t,H)) + \mathbb{E}_{v\sim P_\theta}[\nabla_\theta(s_\theta(v,H))].
\end{aligned} \tag{2.21}$$

The first term moves the network output $\phi(H)$ and the target word embedding $\psi(w_t)$ closer together and is hence referred to as *positive reinforcement*. The

second term is a *negative reinforcement* which pushes away all other word embeddings. It can also be interpreted as the expected gradient $\nabla_\theta(s_\theta(v, H))$ over the whole vocabulary. During training the full expectation needs to be evaluated for each target word and has a cost proportional to $\mathcal{O}(\mathcal{V})$. The following sections describe various techniques that try to approximate the negative reinforcement term.

2.2.3.1 Importance sampling

Bengio et al. (2003b) argued that it is not required to calculate the exact expectation of the negative reinforcement term in Equation 2.21 as training with stochastic gradient descent (SGD) is already an approximation of the full gradient over the training set $\mathcal{D}$. Subsequently, Bengio et al. (2003b) investigated several Monte-Carlo based sampling methods to estimate the expectation $\mathbb{E}_{v\sim P_\theta}[\nabla_\theta(s_\theta(v, H))]$. Simply using the classic Monte-Carlo approximation

$$\mathbb{E}_{v\sim P_\theta}[\nabla_\theta(s_\theta(v, H))] \approx \frac{1}{|\mathcal{S}|}\sum_{v\in\mathcal{S}} \nabla_\theta(s_\theta(v, H)) \tag{2.22}$$

where $\mathcal{S}$ is a set of samples drawn from P_θ is not feasible as sampling from P_θ still requires the computation of the expensive partition function $Z(H) = \sum_{v\in V} e^{s_\theta(v,H)}$. A Monte-Carlo Markov Chain (MCMC) based approach was reported to give poor results. Instead, Bengio et al. (2003b) proposed the application of importance sampling.

Importance sampling is a statistical method that provides an unbiased estimator for the expectation $\mathbb{E}_{v\sim P_\theta}$ by sampling from a more tractable proposal distribution Q. In the domain of language modeling alternative cheap proposal distributions including the uniform and unigram distributions. More formally, the expectation in Equation 2.21 can be rewritten introducing Q as follows

$$\begin{aligned}
\mathbb{E}_{v\sim P_\theta}[\nabla_\theta(s_\theta(v, H))] &= \sum_{v\in\mathcal{V}} P_\theta(v|H)\nabla_\theta(s_\theta(v, H)) \\
&= \sum_{v\in\mathcal{V}} Q(v|H)\frac{P_\theta(v|H)}{Q(v|H)}\nabla_\theta(s_\theta(v, H)) \\
&= \mathbb{E}_{v\sim Q}[\omega(v, H)\nabla_\theta(s_\theta(v, H))] \\
&\approx \frac{1}{|\mathcal{S}|}\sum_{v\in\mathcal{S}} \omega(v, H)\nabla_\theta(s_\theta(v, H))
\end{aligned} \tag{2.23}$$

where $\omega(v, H) = \frac{P_\theta(v|H)}{Q(v|H)}$ is a correction factor referred to as the importance weight. The Monte-Carlo approximation in Equation 2.23 still references the empirical

distribution P_θ and therefore cannot be used directly. As P_θ is defined as the softmax over the individual word scores, it can be written as $P_\theta(w|H) = \frac{P_\theta(w|H)}{Z(H)}$ where $Z(H)$ is the partition function ensuring the distribution is properly normalized. In this case the importance sampling trick can be applied a second time to approximate $Z(H)$.

$$\begin{aligned} Z(H) &= \sum_{v \in \mathcal{V}} P_\theta^0(v, H) \\ &= \sum_{v \in \mathcal{V}} Q(v|H) \frac{P_\theta^0(v, h)}{Q(v|H)} \\ &= \sum_{v \in \mathcal{V}} Q(v|H) \omega^0(v, H) \\ &\approx \frac{1}{|\mathcal{S}|} \sum_{v \in \mathcal{S}} \omega^0(v, H) \end{aligned} \tag{2.24}$$

Substituting $\omega(v, H) = \frac{P_\theta^0(v|H)}{Q(v|H)Z(H)} = \frac{\omega^0(v,H)}{Z(H)}$ into Equation 2.23 yields

$$\begin{aligned} \mathbb{E}_{v \sim P_\theta}[\nabla_\theta(s_\theta(v, H))] &\approx \frac{1}{|\mathcal{S}|} \sum_{v \in \mathcal{S}} \omega(v, H) \nabla_\theta(s_\theta(v, H)) \\ &= \frac{1}{|\mathcal{S}|} \sum_{v \in \mathcal{S}} \frac{\omega^0(v, H)}{\frac{1}{|\mathcal{S}|} \sum\limits_{v' \in \mathcal{S}} \omega^0(v', H)} \nabla_\theta(s_\theta(v, H)) \\ &= \frac{1}{\sum\limits_{v' \in \mathcal{S}} \omega^0(v', H)} \sum_{v \in \mathcal{S}} \omega^0(v, H) \nabla_\theta(s_\theta(v, H)) \end{aligned} \tag{2.25}$$

where $\omega^0(v, H) = \frac{P_\theta^0(v|H)}{Q(v|H)}$. This alternative form is known as self-normalized importance sampling and can be applied when only the unnormalized distribution $P_\theta^0(v|H) = e^{s_\theta(w,H)} \propto P_\theta(v|H)$ is available. Self-normalized importance sampling is a biased estimator but it can be shown that it's bias decreases towards 0 in the limit of an infinite number of samples (Bengio et al., 2003b).

Self-normalized importance sampling is equivalent to an objective function using a weighted softmax where the normalization term is computed only over words in $\mathcal{S}$.

$$NLL_{IS}(\theta|\mathcal{D}) = -\sum_{w,H \in \mathcal{D}} \log \frac{\frac{e^{s_\theta(w,H)}}{Q(w|H)}}{\sum\limits_{v \in \mathcal{S}} \frac{e^{s_\theta(v,H)}}{Q(v|H)}} \tag{2.26}$$

This can be shown by taking the gradient of the negative log-likelihood in Equa-

tion 2.26.

$$\begin{aligned}\nabla_\theta NLL_{IS}(\theta|\mathcal{D}) &= \sum_{w,H\in\mathcal{D}} \nabla_\theta[-\log \frac{\frac{e^{s_\theta(w,H)}}{Q(w|H)}}{\sum_{v\in\mathcal{S}} \frac{e^{s_\theta(v,H)}}{Q(v|H)}}] \\ &= \sum_{w,H\in\mathcal{D}} \nabla_\theta[-s_\theta(w,H) + \log Q(w|H) + \log \sum_{v\in\mathcal{S}} \frac{e^{s_\theta(v,H)}}{Q(v|H)}] \\ &= \sum_{w,H\in\mathcal{D}} -\nabla_\theta(s_\theta(w,H)) + \frac{1}{\sum_{v'\in\mathcal{S}} \frac{e^{s_\theta(v',H)}}{Q(v'|H)}} \sum_{v\in\mathcal{S}} \frac{e^{s_\theta(v,H)}}{Q(v|H)} \nabla_\theta(s_\theta(v,H)).\end{aligned} \tag{2.27}$$

Jean et al. (2014) pointed out that when the proposal distribution Q is uniform, the gradient in Equation 2.27 simplifies to

$$\nabla_\theta NLL_{IS}(\theta|\mathcal{D}) = \sum_{w,H\in\mathcal{D}} -\nabla_\theta(s_\theta(w,H)) + \frac{1}{\sum_{v\in\mathcal{S}} e^{s_\theta(v,H)}} \sum_{v\in\mathcal{S}} e^{s_\theta(v,H)} \nabla_\theta(s_\theta(v,H)). \tag{2.28}$$

A more common implementation is to always include the target word in the denominator of the softmax. As shown by Ji et al. (2015) this is equivalent to using the following negative log-likelihood function

$$NLL_{IS}(\theta|\mathcal{D}) = -\sum_{w,H\in\mathcal{D}} \log \frac{\frac{e^{s_\theta(w,H)}}{Q(w|H)}}{\sum_{v\in\{w\}\cup\mathcal{S}} \frac{e^{s_\theta(v,H)}}{Q(v|H)}}. \tag{2.29}$$

Such implementations are featured in Tensorflow (Abadi et al., 2016) and BlackOut (Ji et al., 2015). They are equivalent to the above importance sampling objective when the sample $\mathcal{S}$ always includes the target word w.

Bengio et al. (2003b) postulated that the number of samples needs to be increased as training progresses. A suggested explanation is that during training the learned distribution P_θ diverges more and more from the chosen proposal distribution Q. One way to measure the required number of samples is to use the effective sample size (ESS) (Kong, 1992)

$$ESS = \frac{(\sum_{v\in\mathcal{S}} \omega^0(v,H))^2}{\sum_{v\in\mathcal{S}} \omega^0(v,H)^2}. \tag{2.30}$$

Bengio et al. (2003b) suggested that the sample size should be adjusted dynamically by monitoring the current ESS during training. Subsequent work (Bengio

and Senécal, 2008) described a modification called adaptive importance sampling which additionally uses a dynamic proposal distribution that closely tracks P_θ. This adaptive distribution is a back-off n-gram model that is constantly updated during training with the currently computed $P_\theta(w|H)$ probabilities.

Unfortunately, using a context dependent proposal distribution is not practical for modern GPU mini-batch implementations. This would require computing a different set of samples for each training example in the mini-batch which does not take advantage of dense matrix-matrix multiplications. A popular workaround is to use a context independent proposal distribution and share the same set of samples across the mini-batch (Jozefowicz et al., 2016; Zoph et al., 2016).

Another issue that can occur in GPU implementations is the lack of GPU memory to hold the entire output embedding matrix of size $\mathcal{O}(d \times \mathcal{V})$. Jean et al. (2014) described an alternative method that can alleviate such problems by partitioning the training corpus into multiple chunks, each with a smaller vocabulary $\mathcal{V}_n < \mathcal{V}$. During training the samples are chosen only from the smaller vocabulary $\mathcal{V}_n$ of the active chunk. This is equivalent to modifying the proposal distribution Q according to the currently active chunk.

$$Q_n(w) = \begin{cases} \dfrac{1}{|\mathcal{V}_n|} & \texttt{if } w \in \mathcal{V}_n \\ 0 \end{cases} \tag{2.31}$$

One simple mechanism for splitting the training corpus is to continue appending words or sentences to a chunk until the vocabulary $\mathcal{V}_n$ exceeds a set threshold. This strategy was later referred to as target sampling (Chen et al., 2015a).

Importance sampling training was successfully applied in various domains. Jean et al. (2014) used importance sampling to train neural MT models. Jozefowicz et al. (2016) reported state-of-the-art perplexity on the one billion word benchmark (Chelba et al., 2013) using importance sampling.

2.2.3.2 Noise contrastive estimation

A simple idea is to make the partition function $Z(H)$ an explicit parameter of the model that is learned during training. More precisely, $P_\theta(w|H)$ can be re-expressed as

$$P_\theta(w|H) = \frac{P^0_\theta(w|C)}{Z(C)} = P^0_\theta(w|C)e^{z_H} \tag{2.32}$$

where $z_H = -\log Z(H)$. Instead of computing z_H as a function of the context H it is directly estimated by the training procedure. But Gutmann and Hyvärinen (2010, 2012) noticed that the standard MLE approach cannot be used directly as the optimizer can maximize any likelihood by setting the parameter z_H as large as possible.

$$\begin{aligned} \arg\max_{\theta,\mathbf{z}} NLL(\theta,\mathbf{z}|\mathcal{D}) &= -\sum_{(w,H)\in\mathcal{D}} \log P_\theta(w|H) \\ &= -\sum_{(w,H)\in\mathcal{D}} \log P_\theta^0(w|H) + z_H \end{aligned} \tag{2.33}$$

The reason is that this is a constrained optimization problem where only certain values z_H^* successfully normalize $P_\theta^0(w|H)$. Gutmann and Hyvärinen (2010, 2012) proposed a novel statistical method called noise contrastive estimation (NCE) that transforms the problem above into an unconstrained optimization problem. The main idea is to indirectly estimate $P_\theta^0(w|H)$ by comparing it to a known noise distribution $P_n(w|H)$. This is achieved by sampling a set of words $\mathcal{S}$ from $P_n(w|H)$ and training a logistic classifier to distinguish between the target word w_t and the noise samples $v \sim P_n(H)$. If for every target word there are k additional noise samples then the probability of a data point belonging to the original data set $\mathcal{D}$ can be calculated using Bayes theorem.

$$\begin{aligned} P(Y = true|w,H) &= \frac{P(Y = true)P(w|Y = true,H)}{P(w|H)} \\ &= \frac{\frac{1}{k+1}P_\theta(w|H)}{\frac{k}{k+1}P_n(w|H) + \frac{1}{k+1}P_\theta(w|H)} \\ &= \frac{P_\theta(w|H)}{kP_n(w|H) + P_\theta(w|H)} \end{aligned} \tag{2.34}$$

where $P(w|Y = true,H) = P_\theta(w|H)$, $P(w|Y = false,H) = P_n(w|H)$ and $P(Y = true) = \frac{1}{k+1}$. Analogous, the probability of the data point being part of the noise sample $\mathcal{S}$ is

$$\begin{aligned} P(Y = false|w,H) &= 1 - P(Y = true|w,H) \\ &= \frac{kP_n(w|H)}{kP_n(w|H) + P_\theta(w|H)}. \end{aligned} \tag{2.35}$$

Equation 2.34 can also be expressed in terms of the sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$ and the ratio of the probabilities $\frac{P_\theta(w|H)}{kP_n(w|H)}$.

$$\begin{aligned} P(Y = true|w, H) &= \sigma(\log \frac{P_\theta(w|H)}{\log kP_n(w|H))} \\ &= \sigma(\log P_\theta(w|H) - \log kP_n(w|H)) \\ &= \sigma(s_\theta(w, H) + z_C - \log kP_n(w|H)) \end{aligned} \tag{2.36}$$

Thus, NCE can be interpreted as a logistic classifier which learns the log probability ratio to distinguish between the true target words and the noise samples. The model can be trained using standard MLE in a supervised fashion.

$$\begin{aligned} NLL_{NCE}(\theta|\mathcal{D}) &= -\sum_{(w,H)\in\mathcal{D}} [\log P(Y = true|w, H) + \sum_{v\in\mathcal{S}} \log P(Y = false|v, H)] \\ &= -\sum_{(w,H)\in\mathcal{D}} [\log \sigma(\log P_\theta(w|H) - \log kP_n(w|H)) + \\ &\quad \sum_{v\in\mathcal{S}} \log(1 - \sigma(P_\theta(v|H) - \log kP_n(v|C)))] \\ &= -\sum_{(w,H)\in\mathcal{D}} [\log \sigma(s_\theta(w, H) + z_H - \log kP_n(w|H)) + \\ &\quad \sum_{v\in\mathcal{S}} \log \sigma(-s_\theta(v, H) - z_H + \log kP_n(v|H))] \end{aligned} \tag{2.37}$$

Gutmann and Hyvärinen (2010, 2012) proved several theorems about the NCE classifier. Given an unlimited amount of data the NCE objective function has a single global optimum when P_θ equals the true data distribution. Further, in the limit of infinite noise samples the model is guaranteed to converge towards this solution. The proposal distribution Q should ideally be close to P but must be at least non-zero when P is non-zero.

The original NCE algorithm (Gutmann and Hyvärinen, 2010, 2012) was formulated in terms of continuous probability density estimation. Mnih and Teh (2012) adapted and applied NCE to speed up training of NPLMs. Initially, a hash table was used to store the additional z_H parameters. But Mnih and Teh (2012) noticed that setting $z_H = 1$ for all contexts H did not effect the performance and avoids storing up to $\mathcal{V}^{n-1}$ extra parameters when H is an n-gram. It should be noted that in the case of RNN language models the context H is the sequence of all words from the start of the sentence and maintaining a separate z_H for each context is not feasible.

Mnih and Teh (2012) stated that the theoretical speed up achieved by NCE training is $\frac{Cost_{NN}+|\mathcal{V}|}{Cost_{NN}+|\mathcal{S}|}$ where $Cost_{NN}$ represents the cost of the neural network layers. Subsequent work applied NCE to accelerate the training of word embeddings (Mnih and Kavukcuoglu, 2013), ASR rescoring (Chen et al., 2015b; Williams et al., 2015; Chen et al., 2016; He et al., 2016) and MT models (Vaswani et al., 2013; Baltescu and Blunsom, 2014; Zoph et al., 2016). Zoph et al. (2016) proposed to share the noise samples $\mathcal{S}$ between all examples in a mini-batch to enable fast GPU implementations leveraging dense matrix-matrix multiplications. Baltescu and Blunsom (2014) combined NCE with a class-based factorization method to train a neural MT with a $100k$ output vocabulary.

Most work used a context independent unigram distribution as the noise distribution. (Mikolov et al., 2013a) suggested that redistributing probability mass to rare words improves the performance. A simple trick is to distort the unigram distribution by exponentiating each word count $count(w)^{\alpha}$ where $\alpha < 1$. Recently, Labeau and Allauzen (2017) studied the effect of bigram, unigram and uniform noise distributions including distortion. They tracked and plotted each individual part of the NCE loss including the log normalization term z_H and the estimated probabilities $P(Y = true|w, H)$ and $P(Y = false|w, H)$ for data and noise words,respectively. The results showed that the mean probability $P(Y = true|w, H)$ for actual target words was close to 0 when training with a unigram distribution. Switching to a bigram distribution raised the mean noticeably and caused the log normalization term z_H to converge to 0.

experimental analysis of the effect of

2.2.3.3 Negative sampling

Mnih and Kavukcuoglu (2013) successfully applied NCE to speed up training of word embedding models. Mikolov et al. (2013a) noticed that when the task is learning word embeddings and not output probabilities in a language model an exact approximation of the softmax output is not necessary. Thus, Mikolov et al. (2013a) proposed an simplified version of NCE called negative sampling. As in NCE a logistic regression model is trained that classifies samples into real data points and noise samples. But unlike NCE the classifier computes this probability

directly and not through the log probability ratio.

$$\begin{aligned} NLL_{NS}(\theta|\mathcal{D}) &= -\sum_{(w,H)\in\mathcal{D}} [\log P(Y = true|w,H) + \sum_{v\in\mathcal{S}} \log(1 - P(Y = false|v,H))] \\ &= -\sum_{(w,H)\in\mathcal{D}} [\log \sigma(s_\theta(w,H)) + \sum_{v\in\mathcal{S}} \log \sigma(-s_\theta(v,H))] \end{aligned} \tag{2.38}$$

Dyer (2014) showed that negative sampling is equivalent to NCE in the case of a uniform distribution with $|\mathcal{V}|$ noise samples.

$$\begin{aligned} P(Y = true|w,H) &= \sigma(s_\theta(w,H)) \\ &= \sigma(\log \frac{e^{s_\theta(w,H)}}{1}) \\ &= \sigma(\log \frac{e^{s_\theta(w,H)}}{|\mathcal{V}|\frac{1}{\mathcal{V}}}) \\ &= \sigma(\log \frac{P_\theta(w|H)}{|\mathcal{V}|P_u(w|H)}) \end{aligned} \tag{2.39}$$

Dyer (2014) concluded that apart from this special case negative sampling is not an appropriate method for training of neural language models. Recently, Melamud and Goldberger (2017) analyzed negative sampling from an information theoretical viewpoint and showed that skip-gram models trained using negative sampling optimize a measure based on the Jensen-Shannon divergence between the network output and word embeddings.

2.2.3.4 Blackout

Ji et al. (2015) proposed to combine both importance sampling and NCE into a single discriminative objective function. The BlackOut loss is specified as

$$\begin{aligned} NLL_{BlackOut}(\theta|\mathcal{D}) = -\sum_{(w_t,H)\in\mathcal{D}} & [\log P(Y = w_t|H,\mathcal{S}) + \\ & \sum_{w_s\in\mathcal{S}} \log(1 - P(Y = w_s|H,\mathcal{S}))] \end{aligned} \tag{2.40}$$

where $P(Y = w_t|H,\mathcal{S})$ represents the probability that word w_t is the correct target word and not an element of the noise set $\mathcal{S}$. In BlackOut this probability is computed as a multinomial classification problem where the target word and noise samples form the possible classes. Thus, $P(Y = w_t|H,\mathcal{S})$ can be computed

as a weighted softmax over the logits $\{w_i \in w_t \cup \mathcal{S} \,|s_\theta(w_i, H)\}$.

$$P(Y = w_t|H, \mathcal{S}) = P(Y = 1|w_t, H, S) = \frac{\frac{e^{s_\theta(w_t,H)}}{Q(w_t)}}{\frac{e^{s_\theta(w_t,H)}}{Q(w_t)} + \sum_{w_s \in \mathcal{S}} \frac{e^{s_\theta(w_s,H)}}{Q(w_s)}} \tag{2.41}$$

Thus, the first part $\log P(Y = w_t|H, \mathcal{S})$ in the BlackOut loss is simply the importance sampling loss in Equation 2.29. The second part mirrors the NCE objective but using a multinomial instead of a binary classifier. Ji et al. (2015) showed that there is an explicit link between the proposal distribution Q and the noise distribution P_n. The above probability $P(Y = w_t|H, \mathcal{S})$ can be derived from the NCE probability $P(Y = 1|H)$ (Equation 2.34) by setting P_n to

$$P_n(w_t|H) = \frac{1}{|\mathcal{S}|} \sum_{w_s \in \mathcal{S}} Q(w_t) \frac{P_\theta(w_s|H)}{Q(w_s)} \tag{2.42}$$

when the noise samples $\mathcal{S}$ were already sampled using Q.

Similarly, Jozefowicz et al. (2016) stated explicitly that importance sampling and NCE are almost identical methods with the only difference being the classification algorithm. Importance sampling can be expressed as multinomial classification $P(Y = w_t|H) = softmax(s_\theta(w_t, H) - \log Q(w_t))$ whereas NCE uses binary classification $P(Y = true|H) = \sigma(s_\theta(w_t, H) - \log Q(w_t))$.

2.2.3.5 TAPAS

A novel important sampling variation proposed by Yu et al. (2017) sorts the sampled set $\mathcal{S}$ and uses only a small subset of highly relevant samples. The sort criteria is simply the magnitude of the output scores $s_\theta(w, H)$. The idea is that large scores also result in large probabilities. Yu et al. (2017) reported that this modification improved accuracy of rank loss objectives but decreased performance when measured using a full softmax loss.

2.2.3.6 Self-normalization

If the output layer could be trained to directly produce normalized scores then no expensive softmax normalization would be required during evaluation. In such a case, the output scores or logits already represent the probability $P_\theta(w|H)$ and can be used directly in subsequent operations yielding a speed up of $|\mathcal{V}|$ compared to the full softmax normalization.

Baltescu and Blunsom (2014) pointed out that even though NCE avoids computing the normalization factor during training, full normalization of the softmax is still required at test time. Training with NCE estimates the softmax logits $s_\theta(w, H)$ relative to the noise distribution P_n and there is no guarantee that for all contexts H

$$z_H = \sum_{w \in \mathcal{V}} \log P_\theta^0(w|H) = \sum_{w \in \mathcal{V}} s_\theta(w, H)). \tag{2.43}$$

In contrast, Chen et al. (2015b) empirically adjusted z_H to be close to the log of the mean partition function $Z(H)$. During testing the probability $P_\theta(w|H) = e^{s_\theta(w,H)} e^{z_H}$ was computed only from the target word logit.

It should be noted that there is a relationship between the normalization constant z_H and the amount of noise samples k. In more detail, Equation 2.34 can be re-expressed as

$$\begin{aligned} P(Y = true|w, H) &= \frac{P_\theta^0(w|H)e^{z_H}}{kP_n(w|H) + P_\theta^0(w|H)e^{z_H}} \\ &= \frac{P_\theta^0(w|H)}{\frac{k}{e^{z_H}} P_n(w|H) + P_\theta^0(w|H)} \end{aligned} \tag{2.44}$$

Devlin et al. (2014) proposed a different way of enforcing this behavior by adding an additional regularization term to the cross entropy loss function. It forces the neural network to produce logits such that the normalization term $Z(H)$ is close to 1 and therefore can be ignored.

$$\begin{aligned} NLL(\theta|\mathcal{D}) &= -\sum_{(w,H) \in \mathcal{D}} \log \frac{P_\theta^0(w|H)}{Z(H)} + \alpha(\log Z(H) - 0)^2 \\ &= -\sum_{(w,H) \in \mathcal{D}} \log \frac{P_\theta^0(w|H)}{Z(H)} + \alpha \log^2 Z(H) \end{aligned} \tag{2.45}$$

One advantage of this scheme is that the trade off between accuracy and self-normalization can be controlled by the hyper-parameter α. But during training the expensive normalization factor $Z(H)$ still needs to be computed for every data point. Andreas and Klein (2015); Andreas et al. (2015) conducted a formal analysis of self-normalized models and the self-normalization properties of NCE. They concluded that models will exhibit self-normalization as long as a certain number of training examples are explicitly regularized. This finding motivated a modified loss function

$$NLL(\theta|\mathcal{D}) = -\sum_{(w,H) \in \mathcal{D}} s_\theta(w, H) - \alpha \frac{|\mathcal{D}|}{|\mathcal{S}|} \sum_{(w,H) \in \mathcal{S}} \log^2 Z(H) \tag{2.46}$$

where $\mathcal{S}$ is a small subset of all training examples $\mathcal{D}$. Chen et al. (2015a) named this approach infrequent normalization and also experimented with a variation where the normalization penalty term is not squared.

$$NLL(\theta|\mathcal{D}) = -\sum_{(w,H)\in\mathcal{D}} s_\theta(w,H) - \alpha\frac{|\mathcal{D}|}{|\mathcal{S}|}\sum_{(w,H)\in\mathcal{S}} \log Z(H) \tag{2.47}$$

Chen et al. (2015a) noted that Equation 2.47 becomes an unbiased estimator of the MLE likelihood when setting $\alpha = 1$. Furthermore, they observed in empirical tries that despite of the self-normalization penalty the variance of $Z(H)$ was still high and the logits needed to be clipped when interpreted as probabilities.

Self-normalization techniques remove the need to compute the normalization constant and thus the logits over the entire vocabulary. This speed up can only be attained when the target word is known, e.g. at test time. If the full probability distribution is required the expensive $W \times h$ matrix-vector multiplication is still required to compute the score for each word.

2.2.4 Other methods

2.2.4.1 CNN softmax

The use of continuous word embeddings allows the neural language models to capture similarities between related words. But a sufficient amount of data is required to learn these embeddings. Additionally, handling of previously unseen OOV words still poses a major problem. But sometimes the meaning of a word can be guessed from its morphology, e.g. the prefix `in-` often negates the original meaning.

Ling et al. (2015) presented a method that computes word embeddings based on sequences of characters. This was achieved by introducing an additional bi-directional LSTM layer that was presented one character at a time. Each character was first converted into a continuous character embedding vector. After processing the whole string the last LSTM hidden state was used as the new word embedding.

Kim et al. (2015) devised an alternative mechanism to compute word embeddings from character-level information based on convolutions. The new embeddings are constructed in multiple stages. First, the character embeddings for each character in the word are concatenated together into a matrix $Q \in \mathcal{R}^{d\times l}$ where d is the size of a character embedding and l is the number of characters in word w.

Next, a convolutional neural network (CNN) layer (LeCun et al., 1990) computes various features f^k on Q. Multiple convolution kernels of varying widths m are applied to Q. The kernels $K \in \mathcal{R}^{d \times m}$ always match the full size of the character embeddings d and are only moved along the time dimension (which represents the sequence of characters). For each kernel max-over-time pooling selects a single value that represents the maximum score computed at any position. Thus, each filter can be seen as a distinct feature detector. To take advantage of interactions between the various convolutional filters an additional neural network layer is added on top. Kim et al. (2015) discovered that highway networks (Srivastava et al., 2015) resulted in good performance.

A side effect of using character-level embeddings is a substantial reduction of model parameters due to the small number of characters compared to the word vocabulary size $\mathcal{V}$. Jozefowicz et al. (2016) explored the idea of character-level embeddings further and applied it to the output embedding layer. In full softmax word scores are computed by the dot product of the neural network output $\phi(H)$ and the output word embedding $\psi(w)$. In CNN softmax Jozefowicz et al. (2016) replaced the direct lookup $\psi(w)$ with a new function $CNN(.)$ that computes word embeddings from character sequences as described in Kim et al. (2015).

$$s_\theta(w, H) = \phi(H)^T \psi(w) = \phi(H)^T (CNN(chars(w)) + corr(w)) \tag{2.48}$$

Based on empirical evidence an additional correction factor per word $corr(w)$ is necessary as Jozefowicz et al. (2016) reports that words with similar spellings could not be sufficiently differentiated. The correction can be stored as a low-dimensional vector and a projection matrix is applied to up-scale it to match the $\mathcal{R}^d$ dimensional RNN output. At test time the resulting word embeddings can be precomputed and cached to avoid any overhead compared to the full softmax.

At test time using CNN softmax still requires computing the logits over the entire vocabulary. To speed up computation Jozefowicz et al. (2016) further suggested to use character outputs as in character-level language models. After training the word-level RNN language model the output layer is replaced with a new LSTM layer that outputs characters conditioned on the word-level RNN output $\phi(H)$. In practice, empirical tests could not match the accuracy of full and CNN softmax.

Finally, it should also be noted that character-level language models allow training on multiple corpora as the model is not dependent on the actual training

vocabulary $\mathcal{V}$ anymore.

2.2.4.2 Maximum inner product search

Vijayanarasimhan et al. (2014) investigated the usage of locality sensitive hashing (LSH) to find the most similar embedding vectors to the neural network output $\phi(H)$. The goal is to efficiently find the few words with high probabilities without computing scores for the entire vocabulary. As scores are computed using the dot product, word embeddings close to the RNN output $\phi(H)$ yield the highest scores and thus also highest probabilities. The proposed method is based on winner-takes-all (WTA) hashing (Yagnik et al., 2011) which given a query vector returns the k most similar items in sub-linear time. The method is used to speed up queries at test time after the model has been trained. In a preprocessing step a hash table is built using the WTA algorithm. During evalution inputs are propagated through the network up until the softmax output layer. There, the k most likely words are retrieved from the hash table using $\phi(H)$ as the key. The output probabilities are computed using only the scores of the k most likely words. Given a large vocabulary the majority of words will have a probabilities close to zero.

Vijayanarasimhan et al. (2014) used a variant of the above method also for training but frequent re-hashing between batches cancels any speed up obtained through the WTA lookup (Chen et al., 2015a).

The idea of quickly finding the most similar vectors which maximize the dot product is also known as maximum inner product search (MIPS) (Ram and Gray, 2012). Various solutions have been proposed for solving MIPS including tree search (Ram and Gray, 2012), hashing (Shrivastava and Li, 2014; Vijayanarasimhan et al., 2014) and k-means clustering (Auvolat et al., 2015).

2.2.4.3 Alternative loss functions

At the time of writing, standard softmax is by far the most prevalent method. This can be attributed to its probabilistic interpretation and natural fit to MLE which leads to the cross entropy loss. But standard softmax is costly to compute due to its normalization term. Vincent et al. (2015); de Brébisson and Vincent (2015); Oland et al. (2017) pointed out some possible alternatives.

Ollivier (2015) analysed several softmax alternatives including spherical softmax.

$$f(x_i) = \frac{x_i^2}{\sum_j x_j^2} \tag{2.49}$$

Vincent et al. (2015) showed that for the family of spherical loss functions which includes the squared error loss the exact output and gradient updates can be computed independently from the vocabulary size. In more detail, gradient updates for the log-spherical softmax loss can be computed in $\mathcal{O}(d^2)$ instead of $\mathcal{O}(d \times \mathcal{V})$. In a subsequent work further analysis was carried out on the log-Taylor softmax (de Brébisson and Vincent, 2015).

$$f(x_i) = \frac{1 + x_i + \frac{1}{2}x_i^2}{\sum_j 1 + x_j + \frac{1}{2}x_j^2} \tag{2.50}$$

This version was inspired by the Taylor approximation of e^x around the origin. Empirical tests showed that these alternative loss functions yield good results on classification tasks in computer vision but fail to be competitive on language modeling benchmarks.

More recently, Oland et al. (2017) claims that the standard softmax is not a natural choice for gradient descent optimization as the saturation leads to vanishing or exploding gradients. They reason that the main benefit of softmax stems from the exponentiation of the logits and not from normalization. In applications when only the relative ranking of the output classes is important (e.g. one-hot image classification) Oland et al. (2017) suggested to use an unnormalized cubic polynomial activation together with a squared loss function.

$$f(x_i) = \alpha x_i^3 + \beta \tag{2.51}$$

The argument is based on an analysis of second order derivatives of the log-softmax loss objective where the normalization term causes the error surface to become more non-convex. This can hinder the convergence of SGD training.

Chapter 3

Tensorflow implementation

All experiments in this project were implemented using Tensorflow (Abadi et al., 2016) version 1.2. Tensorflow is a popular toolkit for implementing deep learning networks and features reverse-mode automatic differentiation. It is maintained as an open source project by Google. At the beginning of this project the source code was manually compiled using specific configuration options suitable for the available hardware.

The code used to construct the LSTM neural language models was partly based on the Tensorflow RNN tutorial[1]. It was initially used as the underlying framework but later extended to support a more efficient mechanism for feeding corpus data, additional hyper-parameters and various kinds of output layers. Further, facilities for scoring and generating example sentences were added.

The following sections describe relevant details of the alternative softmax and LSTM layer implementations used in this work.

3.1 Full softmax

The classic full softmax output layer is implemented in two steps. First, logits are computed in a single dense matrix-matrix multiplication. A specialized Tensorflow function computes the softmax and cross entropy loss.

```
softmax_w = tf.get_variable("softmax_w", [output_size, vocab_size], dtype=tf.float32)
softmax_b = tf.get_variable("softmax_b", [vocab_size], dtype=tf.float32)
logits = tf.nn.bias_add(tf.matmul(output, softmax_w), softmax_b)
loss = tf.nn.sparse_softmax_cross_entropy_with_logits(logits=logits, labels=tf.reshape(
    targets, [-1]))
```

[1] https://www.tensorflow.org/tutorials/recurrent

3.2 Importance sampling

Tensorflow includes a built-in function `tf.nn.sampled_softmax_loss` but the official documentation does not describe the actual method used. Instead, only a reference to Jean et al. (2014) is provided. This motivated the next section which performs a thorough analysis of the implementation. A full understanding of the actual loss is critical for an informed comparison of different softmax methods. The inspection revealed that `tf.nn.sampled_softmax_loss` implements a variant of importance sampling loss where the target word is always part of the sample set $\mathcal{S}$.

The main part of the program logic resides within a shared internal function `_compute_sampled_logits` that is also called in `tf.nn.nce_loss`. In more detail, `_compute_sampled_logits` receives a batch $\mathcal{B}$ of LSTM network outputs Φ along with corresponding target words $T = (w_{t_1} ... w_{t_{|\mathcal{B}|}})$ where Φ is a

$$\Phi = \begin{bmatrix} \phi(H_1)^T \\ \vdots \\ \phi(H_{|\mathcal{B}|})^T \end{bmatrix} \in \mathcal{R}^{|\mathcal{B}| \times d}$$

matrix. Additionally, a set of samples $\mathcal{S}$ is drawn from a proposal distribution Q and shared between all examples in the batch $\mathcal{B}$. In a first step the embedding vectors ψ are retrieved for the batch targets T and the set of sampled words $\mathcal{S}$. The result are two embedding matrices

$$\Psi_T = \begin{bmatrix} \psi(w_{t_1})^T \\ \vdots \\ \psi(w_{t_{|\mathcal{B}|}})^T \end{bmatrix} \in \mathcal{R}^{|\mathcal{B}| \times d}, \Psi_S = \begin{bmatrix} \psi(w_{s_1})^T \\ \vdots \\ \psi(w_{s_{|\mathcal{S}|}})^T \end{bmatrix} \in \mathcal{R}^{|\mathcal{S}| \times d}$$

Next, a single logit is computed for each training example using the corresponding target word embedding. An efficient vectorized implementation is achieved by the element-wise multiplication $\Phi \odot \Psi_T$ and subsequent row-wise summation. The resulting column vector is

$$scores_T = \begin{bmatrix} s_\theta(w_{t_1}, H_1) \\ \vdots \\ s_\theta(w_{t_{|\mathcal{B}|}}, H_{|\mathcal{B}|}) \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{d} \phi(H_1)_i \cdot \psi(w_{t_1})_i \\ \vdots \\ \sum_{i=1}^{d} \phi(H_{|\mathcal{B}|})_i \cdot \psi(w_{t_{|\mathcal{B}|}})_i \end{bmatrix} \in \mathcal{R}^{|\mathcal{B}|}$$

As samples are shared across the batch, a separate logit for each combination of training example and sampled word needs to be computed. This operation is

name	proposal distribution
`tf.nn.uniform_candidate_sampler`	$Q(rank) = \frac{1}{\lvert\mathcal{V}\rvert}$
`tf.nn.log_uniform_candidate_sampler`	$Q(rank) = \frac{\log(rank+2)-\log(rank+1)}{\log(\lvert\mathcal{V}\rvert+1)}$
`tf.nn.fixed_unigram_candidate_sampler`	unigram distribution provided as a list of counts
`tf.nn.learned_unigram_candidate_sampler`	unigram distribution learned over time from target values

Table 3.1: Candidate sampler functions in Tensorflow

simply the matrix-matrix multiplication $\Phi {\Psi_S}^T$

$$scores_S = \begin{bmatrix} s_\theta(w_{s_1}, H_1) & s_\theta(w_{s_2}, H_1) & \dots & s_\theta(w_{s_{|\mathcal{B}|}}, H_1) \\ \vdots & \vdots & \vdots & \vdots \\ s_\theta(w_{s_1}, H_{|\mathcal{B}|}) & s_\theta(w_{s_2}, H_{|\mathcal{B}|}) & \dots & s_\theta(w_{s_{|\mathcal{B}|}}, H_{|\mathcal{B}|}) \end{bmatrix} \in \mathcal{R}^{|\mathcal{B}|\times|\mathcal{S}|}$$

where each row corresponds to the logits over the sampled words $\mathcal{S}$ for a given training example. Before concatenating the target and sample logits into a single matrix, $-\log Q(.)$ is added to each logit. The resulting matrix $\mathcal{R}^{|\mathcal{B}|\times(1+|\mathcal{S}|)}$ returned by `_compute_sampled_logits` can be interpreted as

$$\begin{bmatrix} \log \frac{e^{s_\theta(w_{t_1}, H_1)}}{Q(w_{t_1})} & \log \frac{e^{s_\theta(w_{s_1}, H_1)}}{Q(w_{s_1})} & \dots & \log \frac{e^{s_\theta(w_{s_k}, H_1)}}{Q(w_{s_k})} \\ \vdots & \vdots & \vdots & \vdots \\ \log \frac{e^{s_\theta(w_{t_{|\mathcal{B}|}}, H_{|\mathcal{B}|})}}{Q(w_{t_{|\mathcal{B}|}})} & \log \frac{e^{s_\theta(w_{s_1}, H_{|\mathcal{B}|})}}{Q(w_{s_1})} & \dots & \log \frac{e^{s_\theta(w_{s_k}, H_{|\mathcal{B}|})}}{Q(w_{s_k})} \end{bmatrix} \tag{3.1}$$

Finally, the importance sampling objective is obtained by taking the negative logarithm of the softmax function applied to each row. The exponent in the softmax cancels out the logarithm and leads to the following loss function

$$NLL(\theta|\mathcal{B}) = -\sum_{i=1}^{|\mathcal{B}|} \log\left(\frac{\frac{e^{s\theta(w_{t_i}, H_i)}}{Q(w_{t_i})}}{\sum\limits_{s\in w_{t_i}\cup\mathcal{S}} \frac{e^{s_\theta(w_s, H_i)}}{Q(w_s)}}\right). \tag{3.2}$$

The above loss function is equivalent to Equation 2.29. Hence, `tf.nn.sampled_softmax_loss` is an implementation of importance sampling where the target word is always included in the sampled set $\mathcal{S}$.

Tensorflow provides several functions for drawing samples from different proposal distributions Q. Table 3.1 lists the available options. Each function returns

a tuple of 3 tensors including the sampled words and the expected counts of the target and sampled words. An important point to note is the effect of the `unique` parameter. This parameter controls the sampling process and switches between sampling with and without replacement. But it also has an effect on the returned expected counts. Enabling sampling without replacement changes the expected count from $|\mathcal{S}| \cdot Q(.)$ to $1-(1-Q(.))^{tries}$ where $tries$ is the amount of draws that were necessary to obtain $|\mathcal{S}|$ unique values. Only the first definition yields the original importance sampling loss as $|\mathcal{S}|$ is a constant that cancels itself out when plugged into Equation 3.2. In the case of sampling without replacement (the default value) the returned expected counts do not form a proper probability distribution.

`tf.nn.sampled_softmax_loss` is only called during the training phase in conjunction with one of the candidate sampler functions. Below is the code that computes the training loss where `sampled_values` contains the return value of a candidate sampler function.

```
softmax_w = tf.get_variable("softmax_w", [vocab_size, output_size], dtype=data_type())
softmax_b = tf.get_variable("softmax_b", [vocab_size], dtype=data_type())
loss = tf.nn.sampled_softmax_loss(
        softmax_w, softmax_b,
        targets, output,
        sample_count, vocab_size,
        sampled_values=sampled_values,
        partition_strategy="div")
```

During evaluation the exact loss must be computed using the full softmax.

3.3 Noise contrastive estimation

Tensorflow provides a built-in implementation of the NCE loss `tf.nn.nce_loss` which shares some of the code with `tf.nn.sampled_softmax_loss` in an internal function `_compute_sampled_logits`. The return value of `_compute_sampled_logits` was described earlier in Equation 3.1. The NCE loss is obtained by applying the standard logistic regression loss implemented in `tf.nn.sigmoid_cross_entropy_with_logits` where the target words $w_{t_1}, w_{t_2}, ..., w_{t_{|\mathcal{B}|}}$ and the set of sampled words $\mathcal{S}$ constitute positive and negative

examples, respectively.

$$NLL(\theta|\mathcal{B}) = -\sum_{i=1}^{|\mathcal{B}|}[\log\sigma(e^{s_\theta(w_{t_i},H_i)} - \log Q(w_{t_i}) + \sum_{s\in\mathcal{S}}\log(1-\sigma(e^{s_\theta(w_s,H_i)} - \log Q(w_s))] \tag{3.3}$$

Comparing the above loss and the actual NCE loss derived in Equation 2.37 reveals that $\log k \cdot Q(.)$ has been replaced with $\log Q(.)$ where k was the number of noise samples. As Tensorflow substitutes Q with the return value of the candidate sampler functions, the proper NCE loss is only obtained when setting `unique` to false. In this case the returned expected count is $|\mathcal{S}| \cdot Q(.)$ which matches $k \cdot Q(.)$. As pointed out in Equation 2.44, the log of the normalization constant z_H can be fixed to different values. The built-in Tensorflow implementation `tf.nn.nce_loss` assumes that z_H has been fixed to 0. To be able to run experiments with other constants, the original Tensorflow source code was cloned and an additional argument for the normalization constant was added.

The following code was used during training to calculated the NCE loss where `nce_loss` is the modified version of the original `tf.nn.nce_loss` implementation which takes an additional parameter specifying the normalization constant.

```
softmax_w = tf.get_variable("softmax_w", [vocab_size, output_size], dtype=data_type())
softmax_b = tf.get_variable("softmax_b", [vocab_size], dtype=data_type())
loss = nce_loss(
        softmax_w, softmax_b,
        targets, output,
        Z, sample_count, vocab_size,
        sampled_values=sampled_values,
        partition_strategy="div")
```

The exact loss at evaluation time is calculated using the full softmax analogous to the importance sampling code.

3.4 Adaptive softmax

The authors of the adaptive softmax method (Grave et al., 2016) provided an open-source implementation[2] in Lua using the Torch toolkit. At the time of writing a second open-source implementation written for Tensorflow was also available[3]. This implementation was used as the starting point and updated to run with Tensorflow `1.2`. Additional modifications include support to skip the tail cluster projection matrices and input tensors with a dynamic batch size.

The original implementation returned the loss for each cluster separately as a list of tensors. This makes the interface incompatible with other softmax loss implementations including `tf.nn.sparse_softmax_cross_entropy_with_logits`, `tf.nn.sampled_softmax_loss` or `tf.nn.nce_loss`. The loss tensors cannot be easily combined due to the different cluster allocation of each batch example. The original code expanded each tensor to the full batch size using `tf.sparse_tensor_to_dense`. Unfortunately, this stops gradient propagation in Tensorflow `1.2`. A possible workaround is to use `tf.scatter_nd` instead[4].

Additionally, a novel method for computing logits over the entire vocabulary was derived. The original implementation computes logits only for the cluster containing the target word but the full probability distribution over the entire vocabulary is required in evaluation and generation tasks. It can also help with integration of the adaptive softmax implementation into frameworks that use other softmax methods.

The full set of logits over the vocabulary $\mathcal{V}$ can be computed from the logits of each cluster $s_h, s_1, ..., s_T$ by plugging in the adaptive softmax output word probability into the inverse softmax function. Given that the softmax function has one free parameter (the last logit can be always computed from the previous $|\mathcal{V}| - 1$ values), the inverse of the softmax function can be derived by setting an arbitrary logit to zero, e.g.

$$softmax^{-1}(w_i) = \log \frac{P(w_i)}{P(w_1)} \tag{3.4}$$

Logits for words in the head cluster $\mathcal{C}_h$ are computed by plugging the definition

[2] https://github.com/facebookresearch/adaptive-softmax
[3] https://github.com/TencentAILab/tf-adaptive-softmax-lstm-lm
[4] suggested by Daniel Renshaw

from Equation 2.16 into the inverse softmax

$$
\begin{aligned}
s_h(w_i, H) &= \log \frac{P(w_i|H)}{P(w_1|H)} \\
&= \log \frac{\dfrac{e^{s_h(w_i,H)}}{\sum\limits_{w' \in \mathcal{V}_h} e^{s_h(w',H)} + \sum\limits_{j=1}^{|\mathcal{C}|} e^{s_h(c_j,H)}}}{\dfrac{e^{s_h(w_1,H)}}{\sum\limits_{w' \in \mathcal{V}_h} e^{s_h(w',H)} + \sum\limits_{j=1}^{|\mathcal{C}|} e^{s_h(c_j,H)}}} \\
&= \log \frac{e^{s_h(w_i,h)}}{e^{s_h(w_1,H)}} \\
&= s_h(w_i, H) - s_h(w_1, H)
\end{aligned}
\tag{3.5}
$$

where s_h and $s_1, \ldots, s_T$ represent the word logits in the head and tail clusters, respectively and $c_1, \ldots, c_k$ the tail cluster logits in the head. Similarly, logits for words in tail clusters $\mathcal{C}_t$ can be computed with

$$
\begin{aligned}
s_t(w_i, H) &= \log \frac{P(w_i|c_t, H)P(c_t, H)}{P(w_1|H)} \\
&= \log \frac{\dfrac{e^{s_h(c_t,H)}}{\sum\limits_{w' \in \mathcal{V}_h} e^{s_h(w',H)} + \sum\limits_{j=1}^{|\mathcal{C}|} e^{s_h(c_j,H)}} \dfrac{e^{s_t(w_i,H)}}{\sum\limits_{w' \in \mathcal{V}_k} e^{s_t(w',H)}}}{\dfrac{e^{s_h(w_1,H)}}{\sum\limits_{w' \in \mathcal{V}_h} e^{s_h(w',H)} + \sum\limits_{j=1}^{|\mathcal{C}|} e^{s_h(c_j,H)}}} \\
&= \log \frac{e^{s_h(c_k,H)}}{e^{s_h(w_1,H)}} \frac{e^{s_t(w_i,H)}}{\sum\limits_{w' \in \mathcal{V}_k} e^{s_t(w',H)}} \\
&= s_h(c_t, H) - s_h(w_1, H) + \log softmax_{\mathcal{C}_t}(w_i).
\end{aligned}
\tag{3.6}
$$

The logits over the entire vocabulary $\mathcal{V}$ are obtained by concatenating the logits computed by Equations 3.5 and 3.6. This procedure was implemented as an extension to `adaptive_softmax_loss` as follows:

```
full_logits_ = [tf.slice(head_logits, [0,0], [-1, cutoff[0]]) -
                tf.slice(head_logits, [0,0], [-1,1])]
for i in range(cluster_num):
  if project_factor is None:
    tail_logits = math_ops.matmul(inputs, tail_w[i][0])
  else:
    tail_logits = math_ops.matmul(math_ops.matmul(inputs, tail_w[i][0]), tail_w[i][1])
  full_logits_.append(
      tf.nn.log_softmax(tail_logits) +
```

```
        tf.slice(head_logits, [0, cutoff[0]+i], [-1,1]) -
        tf.slice(head_logits, [0,0], [-1,1])
    )
full_logits = tf.concat(full_logits_, axis=1)
```

3.5 cuDNN LSTM

Tensorflow offers several built-in types of RNN cells including `tf.contrib.rnn.BasicLSTMCell` and `tf.contrib.rnn.LSTMCell`. These can be combined with other Tensorflow ops to form the RNN layer. Appleyard et al. (2016) commented that many advanced optimizations cannot be easily incorporated into general purpose toolkits where the operation graph is specified by the user and not known beforehand. In Appleyard et al. (2016) various optimization strategies are described that were employed in the native NVIDIA CUDA® LSTM implementation shipped as part of the NVIDIA's cuDNN library[5]. These include pre-transposing weight matrices, fusing simple point-wise operations to minimize CUDA® kernel launch overheads and using CUDA® streams to increase the number of parallel operations. Within the first LSTM layer all matrix multiplications involving the input signal can be run in parallel as often the full input sequence is known beforehand (e.g. during training). In multi-layer LSTM networks additional parallelism can be introduced by reordering the computation of LSTM cells. Once the output of an LSTM cell is available, the computation for the next time step within the same layer as well as the same time step in the layer above can start simultaneously. Appleyard et al. (2016) compared this scheduling strategy to a diagonal wave sweeping through the network. In benchmarks a total speed up of 11.1× was achieved compared to a basic unoptimized implementation. This implementation can be accessed in Tensorflow since version `1.2` using the `tf.contrib.cudnn_rnn.CudnnLSTM` wrapper. It represents a full multi-layer LSTM network as a single Tensorflow op. Experiments described in section 5.4 contrasted the optimized cuDNN based implementation with the built-in Tensorflow LSTM cell.

[5] https://developer.nvidia.com/cudnn

Chapter 4

Datasets and training framework

This chapter describes in full detail the datasets, experimental setup and procedures used to carry out the experiments.

4.1 Datasets

All experiments were conducted on two English language datasets, Wikipedia Text8 and a large-scale financial domain corpus. The following sections provide a detailed description of each dataset.

4.1.1 Wikipedia Text8 corpus

The Text8 dataset is a small to medium sized English language corpus. Due to its moderate size it is a popular choice for testing and benchmarking new methods. The corpus consists of text extracted from a 2006 snapshot of Wikipedia. The original Wikipedia dump is part of a standard text compression benchmark (Hutter, 2006) and includes 1GB of extensible markup language (XML) data. Text8 is a clean text version that contains the first 100 million characters after all markup and punctuation has been removed. It contains no sentence segmentation information and is a popular character language model benchmark. Mikolov et al. (2014) used the corpus for training word-based language models after replacing all words occurring less than 10 times by a special OOV token. An artificial sentence boundary was added every $1,000$ words. The last 1 million characters were treated as a held-out validation set.

split	tokens	sentences	batches	vocabulary
train	16,835,605	16,836	6,582	44,370
valid	169,603	170	66	14,576

Table 4.1: Main statistics of the Text8 corpus.

split	tokens	sentences	batches	vocabulary
train	1,059,498,116	48,005,141	432,618	459,454
valid	61,382,199	2,848,939	25090	136,680
test	61,699,037	2,862,852	25219	136,989

Table 4.2: Main statistics of the FDT corpus.

split	tokens	sentences	batches	vocabulary
train	105,874,312	4,800,514	43,232	200,606
valid	6,140,877	284,893	2,510	57,755
test	6,182,099	286,285	2,526	57,727

Table 4.3: Main statistics for the first 10% slice of the FDT corpus

min. word occurrence	vocabulary	OOV tokens	OOV percentage
≥ 1	459,454	0	0%
≥ 9	155,024	697,397	0.066%

Table 4.4: FDT corpus vocabulary after replacing rare word with OOV tokens.

In this project the original preprocessing script[1] was applied to recreate the dataset. Table 4.1 summarizes the key statistics.

4.1.2 Large-scale financial domain text corpus

The large-scale financial domain text (FDT) corpus is a closed-sourced, proprietary dataset provided under licenses by an industry sponsor. It contains conversational English speech that was created from transcribed meetings. It is approximately 60 times larger than Text8 and has a maximum vocabulary of $\sim 500,000$ words. A large portion of the vocabulary is comprised of proper nouns and includes many specialized terms from the financial domain. In a preprocessing step words occurring less than 9 times within the training split were replaced by a special OOV token (see Table 4.4).

The dataset was divided into separate training, validation and test sets ac-

[1] https://github.com/facebookarchive/SCRNNs/blob/master/data/makedata-text8.sh

cording to standard practice. Table 4.2 summarizes the main statistics for each split. In addition, a smaller version of the corpus was created using the first 10% of the training, validation and test files (see Table 4.3).

4.2 Preprocessing

A preprocessing stage transformed all text content into an low overhead format suitable for fast loading into the Tensorflow implementation. A first pass scanned the text data and counted each word occurrence. A unique ID was assigned to each word where more frequent words received lower IDs. A second pass translated words into a sequences of IDs where infrequent words below a set threshold were replace with the OOV token. Each new line in the original text was interpreted as a sentence boundary.

Finally, the ID sequences were partitioned into batches and stored in a Tensorflow specific file format[2]. Each batch contains the next 20 time steps of 128 sentence streams. Each sentence was randomly assigned to one of the 128 streams where sentences were separated by the end-of-sentence (EOS) token. No additional padding was inserted. To simplify program logic, the target outputs of the last time step were also included in a batch.

4.3 Training framework

The preprocessing and training procedure follows closely the setup described in Jozefowicz et al. (2016). All experiments were exclusively conducted with a batch size of 128 streams each 20 tokens long. The shuffling was performed only once per dataset and the resulting batches were stored permanently. Hence, the batch sequence stayed constant for each experiment and training epoch. The LSTM hidden state was set to zero at the start of each epoch. After each batch the final LSTM hidden state was transfered to the consecutive batch. Such an arrangement does not allow for learning long dependencies across multiple sentences (Jozefowicz et al., 2016).

The models were implemented using Tensorflow as described in Chapter 3. Training was performed using the built-in Tensorflow mechanism of unrolling the RNN for a predefined number of steps. This is similar to truncated BPTT

[2] as `TFRecords`

Hyper-parameter	Text8	FDT
RNN	1x512 LSTM	1x2048 LSTM
Cell type	`BasicLSTMCell`	`BasicLSTMCell`
Forget bias	0	0
Embed. size	512	512
Optimizer	Adagrad	SDG
Learning rate	0.2	2
Learning rate decay	50%	N/A
Uniform init. range	0.1	0.05
Gradient clipping	0.25	1.0
Dropout	25%	0%

Table 4.5: Hyper-parameters used for training models on the Text8 and FDT corpora.

(Zipser, 1990) but it should be noted that the Tensorflow implementation does not propagate the gradients between batch boundaries (Jozefowicz et al., 2016). Thus, the gradient of the first target word in a batch only flows to the first input and not the inputs in the previous batch. In all experiments the LSTM layer was unrolled for 20 time steps. Different optimizers were used for each corpus. Adagrad (Duchi et al., 2011) was used for training on the small Text8 corpus whereas plain SGD without momentum was used for the FDT corpus. In both cases gradient clipping was applied. The learning rate was reduced every time the validation perplexity did not decrease between training epochs on Text8. Regularization was added using dropout (Srivastava et al., 2014) before, between and after each LSTM layer (Zaremba et al., 2014). Weight matrices were initialized using a uniform distribution. A preliminary exploration phase explored various hyper-parameter settings. Table 4.5 specifies the chosen values used in the subsequent experiments.

At the end of each epoch the perplexity was measured on a separate held-out validation dataset. The models were evaluated once on the test set after training for a predefined number of epochs. In the case of Text8 no test set was available. On the large FDT corpus full epochs were subdivided into smaller units where 10 and 100 pseudo epochs are equivalent to a full epoch of training on the 10% slice and the full FDT corpus, respectively.

All models were trained using a single GPU. The majority of experiments was performed on an NVIDIA GeForce GTX 1080 GPU with 8GB of memory. Long running experiments on the full FDT corpus were put on an NVIDIA Tesla K80 GPU with 12GB of memory.

Chapter 5

Experiments and discussion

The following sections list the main experiments that were carried out during this project. Each section first describes the research objective and experimental method in more detail and is followed by a presentation of the results and further discussion.

5.1 Adaptive softmax

One of the main objectives of this work is to reproduce the results in Grave et al. (2016) and investigate if adaptive softmax is a viable and possibly better alternative to sampling methods. The ideal method must achieve a considerable speed up while having similar accuracy to full softmax. The following experiments aim to establish the effect of various properties of the adaptive softmax method including the number of clusters, shortlist and cluster sizes, projection matrices and the influence of vocabulary size. Two key performance indicators measured the above criteria: training throughput in words per second (WPS) and generalization error estimated by validation and test set perplexity.

5.1.1 Training speed benchmarks

Grave et al. (2016) defined the theoretical cost of adaptive softmax with different cluster sizes. The following experiments were designed to measure the actual training speed observed with a complete Tensorflow implementation.

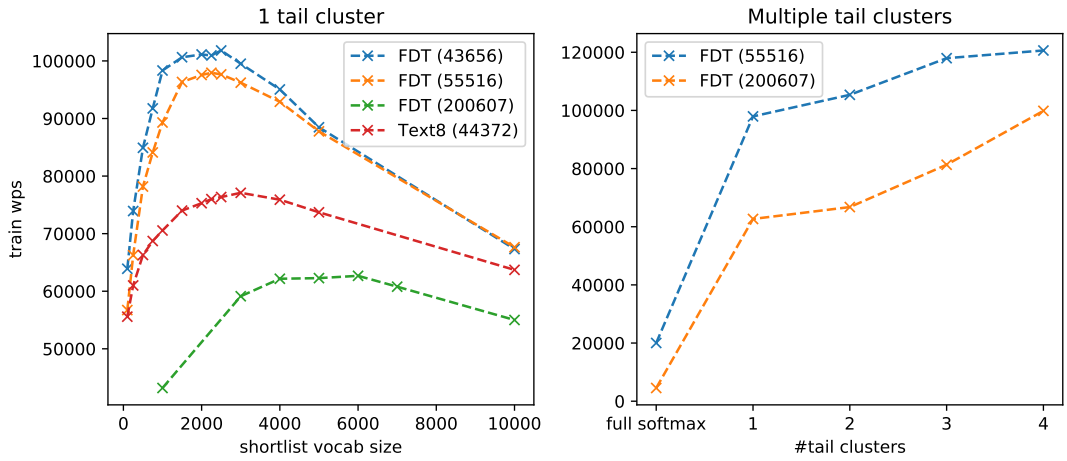


Figure 5.1: Adaptive softmax training speed in relation to shortlist size and number of tail clusters. The cluster dimensions have been tuned only for training speed and not model accuracy. Experiments were repeated with different vocabulary sizes.

5.1.1.1 Method

An extensive grid search covered most combinations of shortlist and tail cluster sizes up to 5 tail clusters. Additional experiments investigated the effect of projection matrices on the throughput. Models were trained with the standard setup on Text8 and variations of FDT with different vocabulary sizes. The achieved training speed was measured by the elapsed wall time required to train each model for a single epoch. In the case of FDT a single pseudo epoch was used.

5.1.1.2 Results and discussion

The left plot in Figure 5.1 shows the training speed of adaptive softmax with a single tail cluster as a function of the shortlist size. The training speed peaks with shortlist sizes around 2500 and 6000 words for the small and large vocabulary corpora, respectively. This corresponds to only a tiny fraction of the respective vocabularies ($\sim 5\%$ and $\sim 3\%$). The overall trend matches the results reported by Grave et al. (2016). Similar plots with multiple tail clusters were omitted for brevity as they showed similar outcomes preferring non-linearly increasing tail cluster sizes. The right plot in Figure 5.1 summarizes the maximum training throughput in WPS that could be achieved as a function of the number of tail clusters. The switch from the full to adaptive softmax yields a substantial speed up whereas the effect of adding additional tail clusters is not as pronounced,

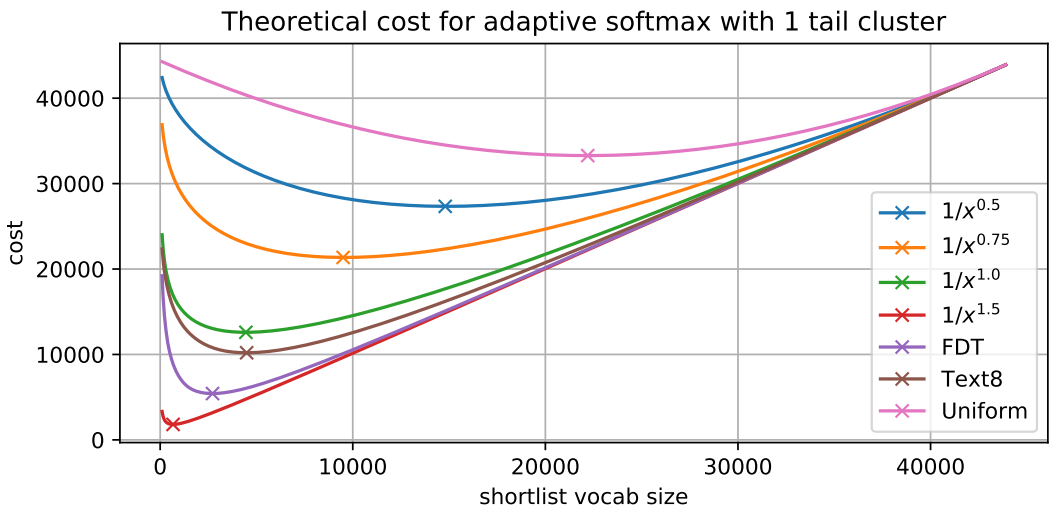


Figure 5.2: Theoretical cost of adaptive softmax with a single tail cluster for different distributions. The crosses mark the optimal shortlist size.

especially for small vocabulary corpora.

The plots in Figure 5.1 indicate that the speed up of adaptive softmax is affected by the vocabulary size. The achieved speed up decreases steadily for FDT versions with larger vocabularies. But comparing the throughput on Text8 and the similarly sized FDT (43656) version suggests that not only the vocabulary size but also the nature of the corpus has an influence on training speed. Further investigation discovered that the performance of adaptive softmax is dependent on the unigram word distribution in the corpus. Corpora with unigram distributions skewed towards the most frequent words achieved higher speed ups as the probability mass of words included in the shortlist is increased. Figure 5.2 plots the theoretical cost (Grave et al., 2016) of adaptive softmax as a function of the shortlist size using various unigram probability distributions. Crosses mark the optimal shortlist size. Zipf's distribution $P(rank) \propto \frac{1}{rank^n}$ with different n was chosen as it allows to easily vary its skewness. The effect of higher n allows to reduce the shortlist size thus reducing the amount of computation. In the extreme case of a uniform distribution the optimal shortlist size contains exactly half of the vocabulary. As the top ranked words become more and more frequent the optimal shortlist size is decreasing. Figure 5.7 also includes the Text8 and FDT corpora. The vocabulary in the FDT corpus is domain specific and includes many proper nouns that occur only a few times. This results in a large flat tail and is a better match for adaptive softmax than Text8. Hence, adaptive softmax

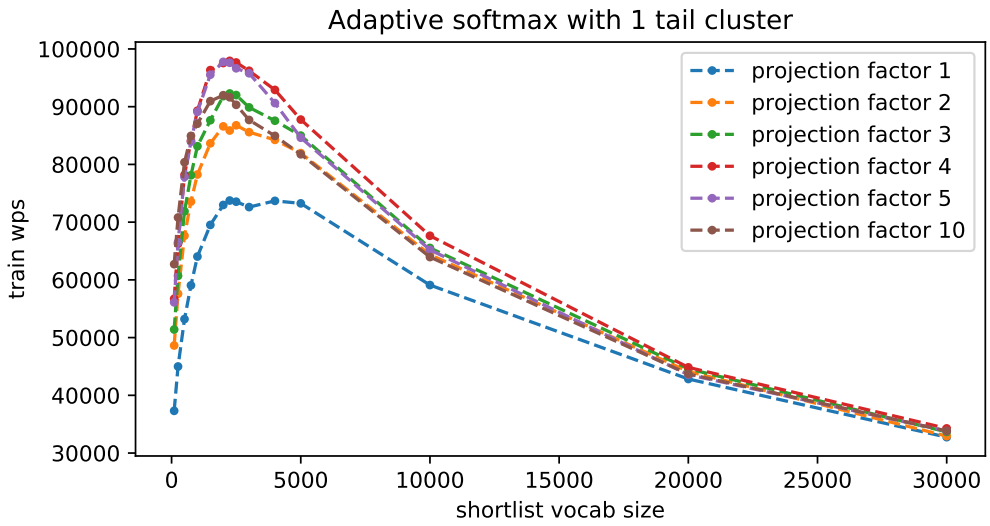


Figure 5.3: Adaptive softmax training speed in relation to the projection factor.

appears well suited for speeding up training of large vocabulary corpora including many infrequent words.

Finally, Figure 5.3 shows the effect of tail cluster projection matrices on the training speed. A reduction of parameters in the tail cluster weight matrices results in larger speed ups but shrinking the dimensionality by a factor of more than 4 ($d_t = \frac{d}{4}$) has no beneficial effect on the throughput. This empirical result is in line with the observation that narrow matrix multiplications are not efficient on GPUs (Grave et al., 2016) .

5.1.2 Generalization error

The motivation behind many alternative softmax implementations is the reduction of required computation compared to the full softmax. But any speed up must be carefully weighted against possible loss of accuracy. The following experiments investigate the convergence characteristic of adaptive softmax on the Text8 and large-scale FDT corpora.

5.1.2.1 Method

To measure the impact of hyper-parameters like cluster sizes and tail projection factors on the generalization error the models were now trained for 30 and 2 epochs on the Text8 and FDT corpora, respectively. At this point no substantial improvement in validation perplexity was measurable. As some of the Text8

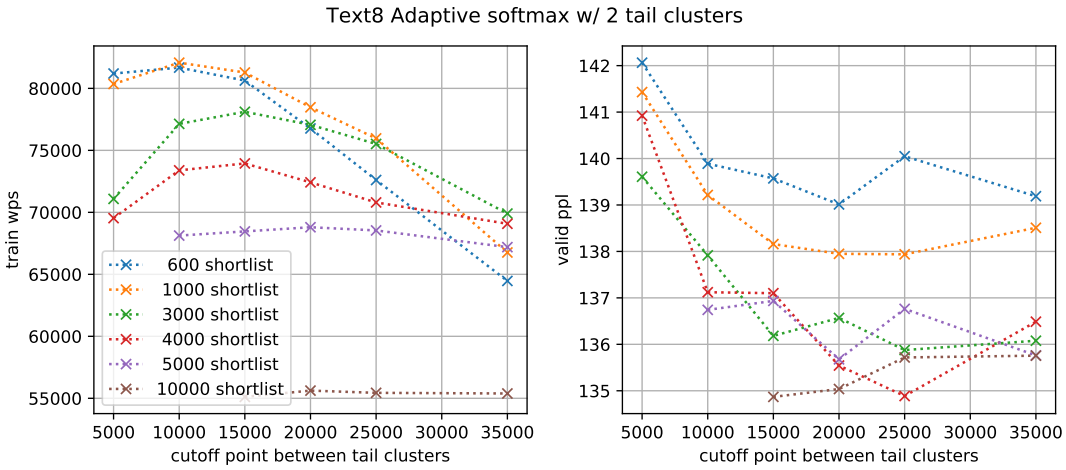


Figure 5.4: Comparison of adaptive softmax with different shortlist sizes on Text8.

results in the literature were reported after 5 epochs of training (Grave et al., 2016; Shen et al., 2017), the initial experiments were also stopped after 5 epochs. But additional experimentation revealed that more than 30 epochs are required to reach convergence on Text8. Furthermore, the results after 5 epochs are not necessarily representative of the ranking between methods after convergence.

Training until full convergence requires substantial resources and an exhaustive grid search was not feasible. In particular, only few experiments on the large-scale FDT corpus were possible within the available time frame. The standard training setup outlined in section 4.3 was used. The WPS training speed was calculated by averaging the training times of each epoch.

5.1.2.2 Results and discussion

Figure 5.4 summarizes the training speed and perplexity of adaptive softmax with 2 tail clusters on the Text8 corpus. Training speed shows a similar pattern as the previous benchmarks. The highest throughput was achieved with a 1000 word shortlist ($\sim 2.2\%$). The optimal split point between the two tail clusters depends on the actual shortlist size but in general the first tail cluster containing more frequent words should have a smaller size. Unfortunately, Figure 5.4 shows that the optimal cluster size with regard to training speed does not yield the optimal perplexity. Here, a shortlist of 4000 words ($\sim 9\%$) obtains substantially lower perplexity while still having a high throughput. Large shortlists also yield good generalization performance but loose a lot of the speed advantage. The accuracy of the model jumps between various cutoff points which might be caused by vari-

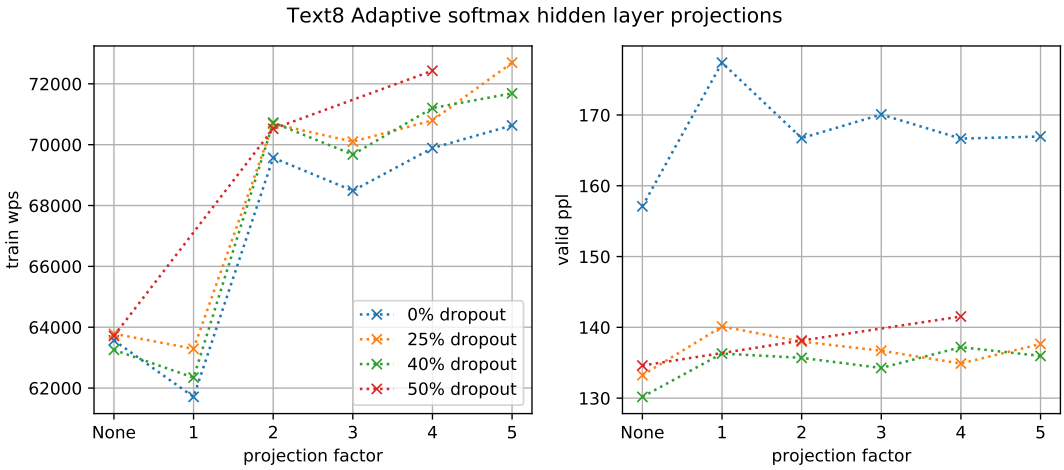


Figure 5.5: Comparison of adaptive softmax with different projection factors on Text8.

ance from random initialization. Another explanation is that the method itself is somewhat sensitive to the exact assignment as it is the case for hierarchical softmax (Morin and Bengio, 2005; Mnih and Hinton, 2009). Thus, the optimal split point needs to be determined experimentally on a case by case basis. Nevertheless, assigning approximately half of the vocabulary to the last cluster seems to work well independent of the shortlist size.

Figure 5.5 shows the impact of applying projection matrices to reduce the dimension d_t of the tail cluster weight matrices. For comparison, multiple models with different dropout rates were trained as the reduction in the number of parameters results in a higher model bias. Surprisingly, only the introduction of the projection matrices (projection factor ≥ 1) has a negative impact on validation perplexity. Halving the number of parameters in the tail clusters through projections does not have a clear effect on accuracy but results in a 10% speed up. The only exception were the models with 50% dropout. This suggests that the model might contain too many parameters. These findings also agree with Chen et al. (2015a). Making less accurate predictions for rare words has a marginal effect on the overall perplexity as the probability mass of words assigned to the tail clusters is small.

Table 5.1 lists the results obtained after training on the large-scale FDT corpus for full 2 epochs. As previously seen on the smaller Text8 corpus, training speed is increased with the number of tail clusters. But unfortunately accuracy also decreases with the number of tail clusters. There is a substantial gap in perplexity between the models with 3 and 4 tail clusters. Models with fewer tail clusters do

Cluster cutoff points	Train WPS[1]	Valid PPL	Test PPL
5000, 155026	6389	36.23	36.029
4000, 40000, 155026	7496	36.257	36.081
2000, 25000, 50000, 155026	8341	36.507	36.346
601, 5000, 15000, 50000, 155026	9158	37.582	37.412

Table 5.1: Effect of the number of tail clusters on the validation and test accuracy on the large-scale FDT corpus. Perplexity values reported after training for 2 full epochs.

not show further perplexity gains but have a larger computational cost.

5.2 Sampling methods

The objective of the following experiments is to establish competitive baselines that will be used subsequently to compare and contrast adaptive softmax. The two sampling methods being investigated are importance sampling and NCE. Reasonable values for common hyper-parameters were established in preliminary experiments. The following experiments investigate the effect of hyper-parameters specific to sampling methods on the two key performance indicators, training speed and validation accuracy. These include, amongst others, the choice of proposal distribution and sample set size. The experiments were limited to the Text8 corpus as training the models until convergence on the large-scale FDT corpus would require too many resources.

5.2.1 NCE normalization constant

Section 3.3 detailing the built-in Tensorflow NCE implementation highlighted the inability to specify the normalization constant Z thus effectively hard-coding it to 1. Results in the literature are not conclusive about the correct value to use for Z (Mnih and Teh, 2012; Chen et al., 2015b). Thus, an important questions is whether the original Tensorflow implementation is justified in hard-coding the normalization constant to 1.

[1]Using NVIDIA Telsa K80 GPU

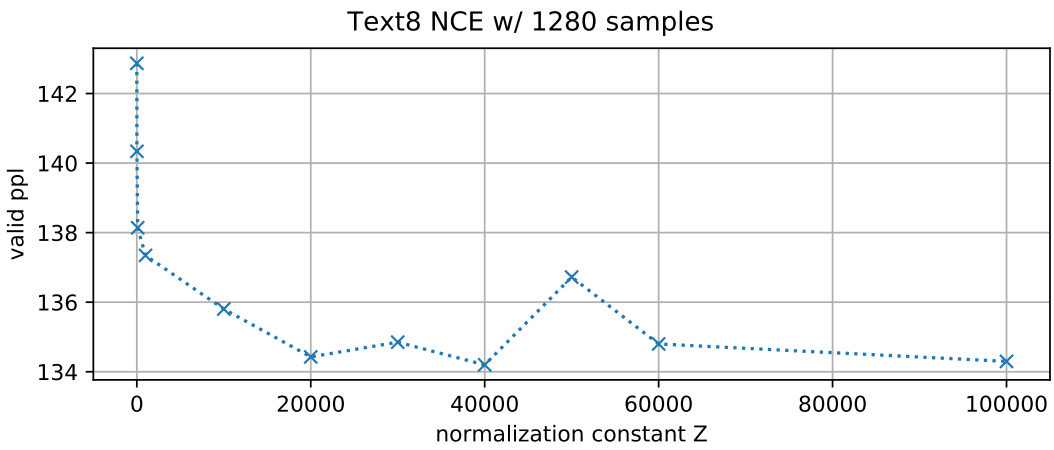


Figure 5.6: Comparison of NCE with different values of the normalization constant Z on Text8 with 25% dropout.

5.2.1.1 Method

This work extended the original NCE implementation with support for different normalization constants Z. A grid search was conducted to find the optimal value for Z with regard to validation set perplexity. Each model was trained for 30 epochs on the Text8 corpus using the standard training setup.

5.2.1.2 Results and discussion

Figure 5.6 shows the validation perplexity after training for 30 epochs as a function of different normalization constants Z. The plot shows that values less than 20000 are suboptimal, especially values around 1. These results are similar to Chen et al. (2015b) where $Z = e^9 \approx 8103$ was reported as a good choice across various corpora. The trend in Figure 5.6 indicates that larger normalization constants result in lower perplexity apart from the outlier around $Z = 50,000$. Further work is required to establish if this is due to initialization noise or if there exists an underlaying cause. In subsequent experiments using the NCE method the normalization constant Z was fixed to $40,000$ if not stated otherwise.

5.2.2 Proposal and noise distributions

Previous research suggested that the choice of the proposal or noise distribution has an impact on convergence (Bengio and Senécal, 2008; Gutmann and Hyvärinen, 2010; Labeau and Allauzen, 2017). Tensorflow offers built-in imple-

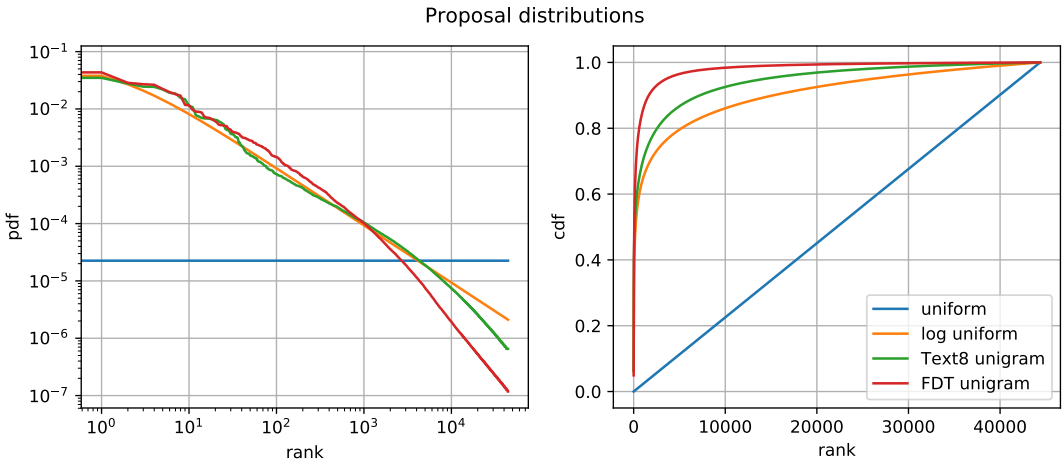


Figure 5.7: Plots of different proposal distributions Q including the unigram Text8 and FDT distributions.

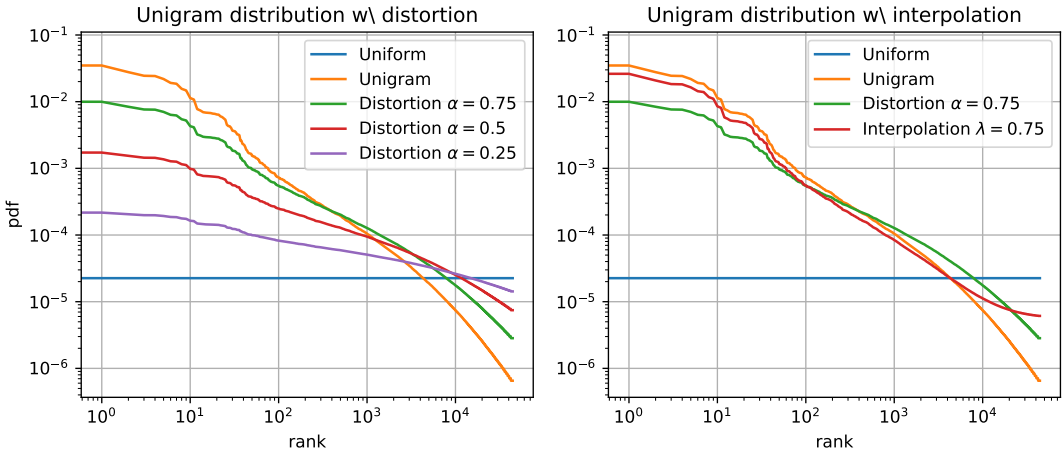


Figure 5.8: Comparison of the distortion and interpolation techniques to boost the tail of the Text8 unigram distribution.

mentations for sampling from unigram, log uniform and uniform distributions (see Figure 5.7 and Table 3.1). Further, a distortion to the unigram distribution $Q(w) \propto P_{Unigram}(w)^{\alpha}$ can be applied (Mikolov et al., 2013a). The left-hand plot in Figure 5.8 illustrates the effect of varying distortions applied to the Text8 unigram distribution. In all cases there is a further choice between sampling with and without replacement. The following experiments attempt to answer the question if there are specific choices that offer an advantage in conjunction with importance sampling and NCE.

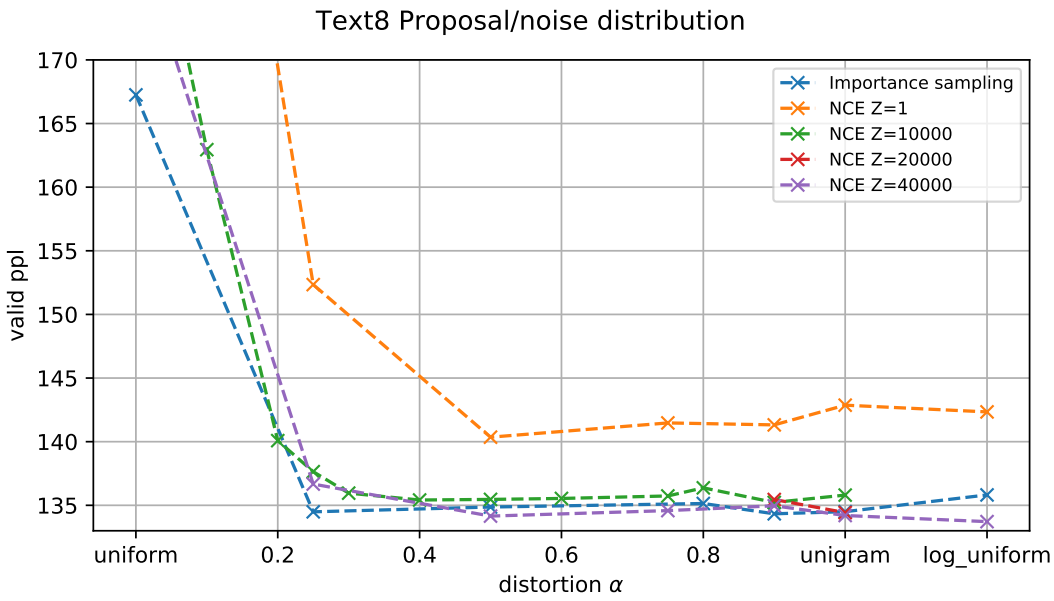


Figure 5.9: Comparison of various distorted unigram noise distributions on Text8 with 25% dropout.

5.2.2.1 Method

The standard training setup was used to train models on the Text8 corpus for 30 epochs. Each experiment initialized a different candidate sampler object which was then passed to the importance sampling and NCE loss functions. The investigation was constrained to the previously introduced context independent proposal distributions. Context dependent proposal distributions could not be used due to the constraints of the GPU based batch implementation. Sampling with replacement and a sample set size of 1280 were used in all experiments.

5.2.2.2 Results and discussion

Figure 5.9 shows the validation set perplexity as a function of the proposal distribution for importance sampling and NCE with various choices of the normalization constant Z. An immediate conclusion is that the uniform distribution is a poor choice for both importance sampling and NCE. Otherwise, the exact shape of the unigram distribution has a smaller effect on the final perplexity. Good performance is achieved as soon as the proposal distribution begins to resemble Zipf's law (Zipf, 1949). Only the NCE model with normalization constant $Z = 1$ showed a more pronounced sensitivity to the distortion α. A recent empirical

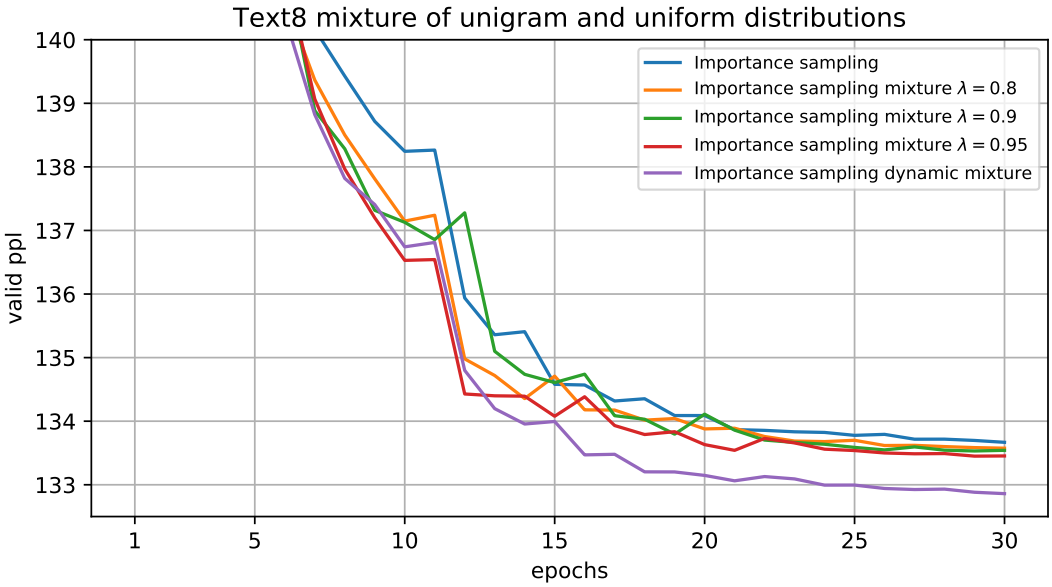


Figure 5.10: Effect of interpolating unigram and uniform proposal distributions on the Text8 corpus with 25% dropout.

study (Labeau and Allauzen, 2017) using the built-in Tensorflow NCE implementation reported that using distortion with low α improved the convergence compared to the unmodified unigram distribution. Labeau and Allauzen (2017) attribute this to boosted probabilities of rare words. In Figure 5.9 this effect can only be weakly observed for $Z = 1$ (which was shown to be used implicitly in `tf.nn.nce_loss`). The results here indicate that the unigram distortion becomes less important with larger normalization constants Z. The suggested hypothesis is that predictions for rare words tend to be inaccurate and noisy and increasing Z in Equation 2.44 helps to smooth out potential spikes while allowing to use the original unigram distribution which is closer to the underlying data. Future work applying the metrics developed in Labeau and Allauzen (2017) to NCE implementations supporting other normalization constants could provide additional insights.

Inspired by the above results, this work proposes an alternative method for boosting the tail of the unigram distributions. A simple linear interpolation of the unigram and uniform distributions raises the probability of the lowest rank words without substantially altering the rest of the distribution. The new probability is obtained by

$$Q(w) = \lambda P_{Unigram}(w) + (1-\lambda) P_{Uniform}(w) \tag{5.1}$$

where λ is the interpolation weight. The right hand plot in Figure 5.8 contrasts the effect of linear interpolation and unigram distortion on the Text8 corpus.

Figure 5.10 shows the results of preliminary tests using importance sampling and different interpolation weights λ. A small perplexity gain can be achieved by a minimal boost of the most rare words using $\lambda = 0.95$. A bigger gain was realized by starting off training with a lower interpolation weight around $\lambda \approx 0.8$ and gradually increasing λ towards the unigram distribution with each training epoch. Other research reported that switching to a bigram distribution did not yield good results (Bengio et al., 2003b). In future work additional experiments should measure the effect of interpolated distributions on large vocabulary corpora and on the NCE method.

5.2.3 Sample set size

The sample set size is a crucial hyper-parameter of the importance sampling and NCE methods. It controls the trade-off between the training and convergence speed. The goal of the following experiments is to quantify this trade-off and determine good baseline values.

5.2.3.1 Method

The standard training setup was used to train models on Text8 for 30 epochs. Additionally, a few experiments using importance sampling were carried out on the 10% slice of the FDT corpus and trained for half an epoch.

5.2.3.2 Results and discussion

Figures 5.11 and 5.12 summarize the results of training importance sampling and NCE models with different sample set sizes. The left-hand side plots shows that the training speed is monotonically decreasing with larger sample set sizes. The right-hand side plots show the achieved validation set perplexity on the Text8 and FDT corpora. For Text8 a sample size around 1000 is a good trade-off between training speed and model accuracy. For the larger vocabulary FDT corpus the sample size needs to be doubled to achieve better accuracy. In general, increasing the sample size always decrease training speed but shows diminishing perplexity gains for large sample sets.

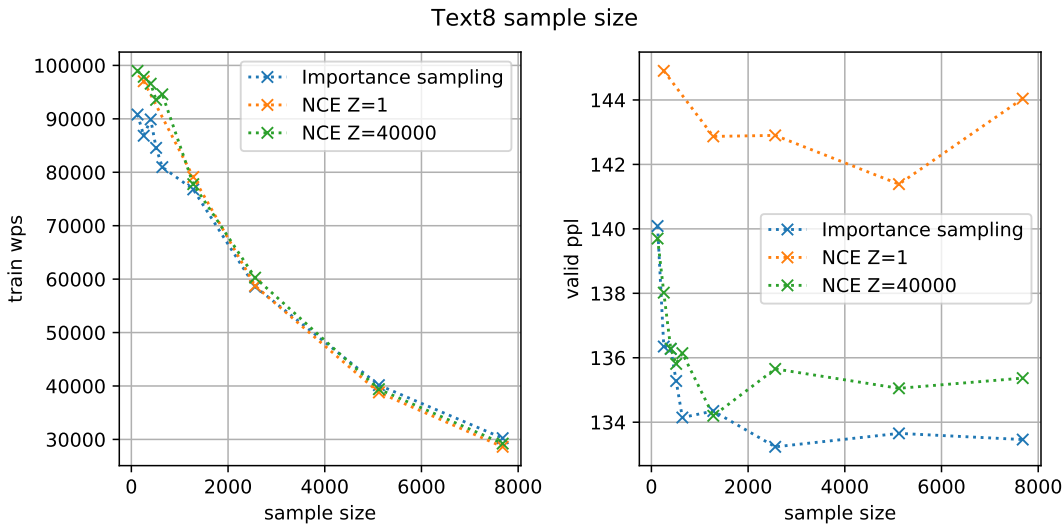


Figure 5.11: Comparison of importance sampling and NCE with different sample set sizes on the Text8 corpus with 25% dropout.

The NCE plots in Figure 5.11 suggest that increasing the sample set size can sometimes decrease the accuracy. This finding contradicts the theory which says that the NCE gradient becomes the maximum likelihood gradient with unlimited noise samples (Mnih and Teh, 2012). For NCE with normalization constant $Z = 40000$ the optimal sample size is 1280 though this size was also used to determine the optimal normalization constant Z.

A closer inspection of the validation perplexity over time uncovered that some of the models did not fully converge after 30 epochs, especially for $Z = 1$. But this does not fully explain the observed relationship between sample set size and normalization constant for $Z = 40000$. Equation 2.44 hints that increasing the number of noise samples can be seen as increasing the normalization constant but in the context of the NCE loss (Equation 2.37) this is the correct behavior. Nevertheless, it highlights a possible problem with using NCE in batch mode training on GPUs where the same noise samples are shared across all examples. In this case there are only $|\mathcal{S}|$ unique samples whereas the term $P(Y = false|v, H)$ is included $|\mathcal{B}| \times |\mathcal{S}|$ times in the NCE loss. In a simplified scenario where the context H is assumed to be equal for all batch examples it leads to

$$NLL(\theta|\mathcal{B}) = -\sum_{i=1}^{|\mathcal{B}|} \log P(Y = true|w_i, H) - |\mathcal{B}| \sum_{v \in \mathcal{S}} P(Y = false|v, H). \tag{5.2}$$

As NCE considers target and sampled words separately, words in the noise sample $v \in \mathcal{S}$ receive a disproportionate amount of attention. On the other hand, the

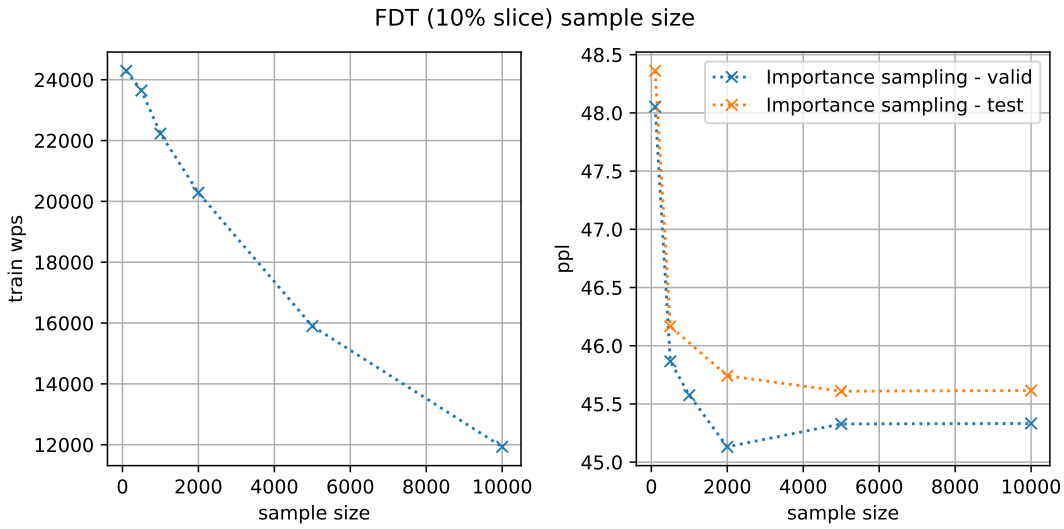


Figure 5.12: Comparison of importance sampling with different number of samples on the 10% slice of the FDT corpus.

importance sampling objective in Equation 2.29 combines the target word and noise samples which could make it less susceptible to sharing of samples. A related argument was made in Jozefowicz et al. (2016) where importance sampling is interpreted as a variant of NCE with multi-nominal classification. Further work is required to substantiate this claim.

5.3 Comparison of alternative softmax methods

The following sections compare and contrast the adaptive softmax method to other full softmax alternatives including importance sampling and NCE. The main criteria are as before training speed and achieved validation set accuracy. Additional considerations are the number of trainable parameters, total memory usage and performance on the extrinsic task of n-best list re-scoring.

5.3.1 Trainable parameters, memory consumption and training speed

The memory available on modern GPU cards is limited and 8 to 12 GB is the current norm for standard consumer hardware. Therefore, an important consideration is the amount of memory required to train a model. If it exceeds the capacity of the GPU card, a large penalty is paid for transferring data from and

to host memory. In other cases the model will fail due to out of memory errors and training needs to be split across multiple cards. The following experiments measure the number of parameters, the amount of memory used and the training speed with different vocabulary sizes.

5.3.1.1 Method

The profiling tool `tfprof` was used to analyze the number of trainable parameters and memory usage in various parts of the model. `tfprof` is included in the Tensorflow source code[2] but has to be compiled manually. It acts on additional profiling meta-data that has to be collected during a Tensor flow session. The meta-data can be enabled using the following code:

```
run_options = tf.RunOptions(trace_level=tf.RunOptions.FULL_TRACE)
run_metadata = tf.RunMetadata()
sess.run(tensors,feed_dict,options=run_options, run_metadata=run_metadata)
```

The experiments were performed on variations of the 10% FDT slice with different vocabulary sizes. The profiling was enabled for an arbitrary batch in the middle of the first pseudo epoch and serialized to disc. Later, various statistics were generated with `tfprof`.

5.3.1.2 Results and discussion

Figure 5.13 compares the number of trainable parameters and memory usage of each method. Importance sampling (as well as NCE) uses the full sized $\mathcal{R}^{d\times|\mathcal{V}|}$ output embedding matrix as only the gradient computation is approximated. Thus, the sample set size has no influence on the number of parameters. In comparison, the projection matrices before each tail cluster in adaptive softmax achieve a huge reduction of trainable parameters. The right-hand side shows the amount of memory required during computation of the output loss. Here, the number of samples linearly increases the cost of the importance sampling loss. The memory usage also increases with vocabulary size due to the growing output embedding matrix. This is also true for adaptive softmax, but projections in the tail clusters reduce the slope substantially.

Figure 5.14 shows a breakup of the memory usage in each layer of the RNN language model as well as the overall total. The embedding layer grows linearly

[2] https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/profiler/README.md

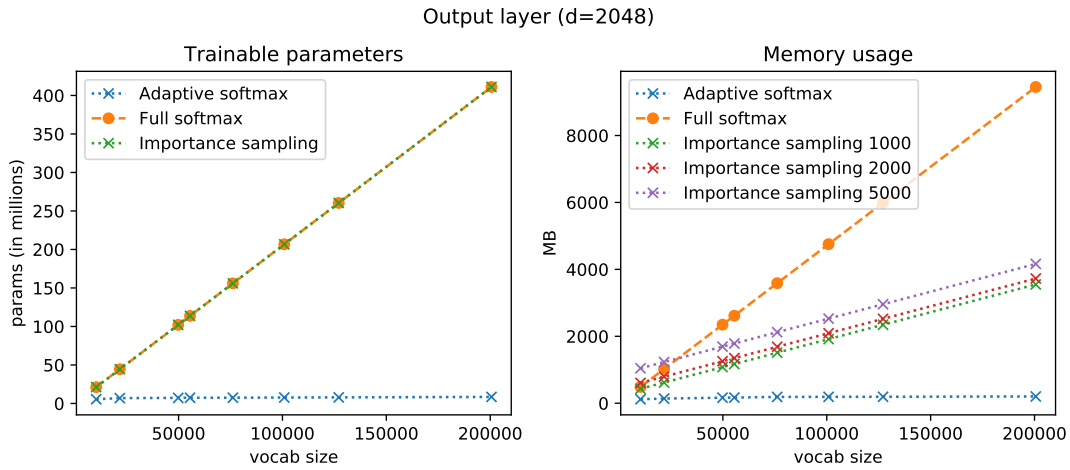


Figure 5.13: Comparison of number of parameters and memory usage in output layer.

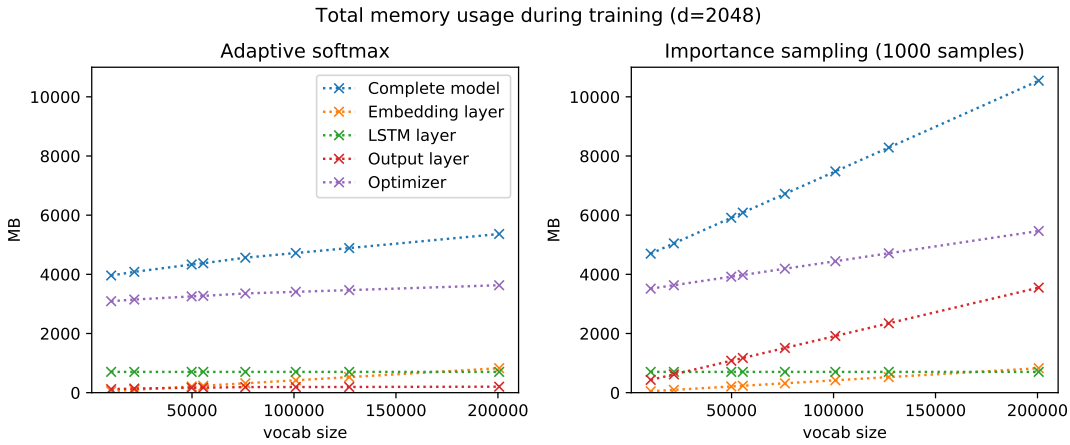


Figure 5.14: Comparison of memory usage of various methods and vocabulary sizes.

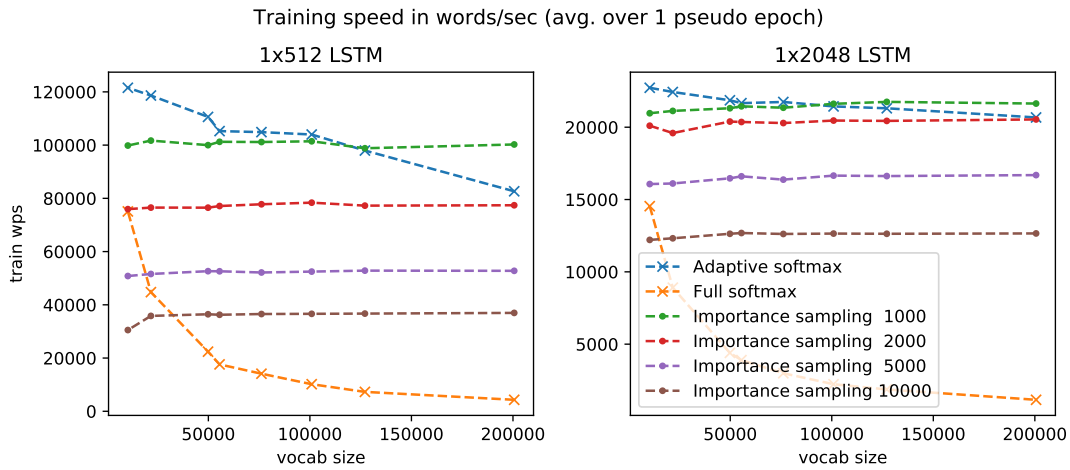
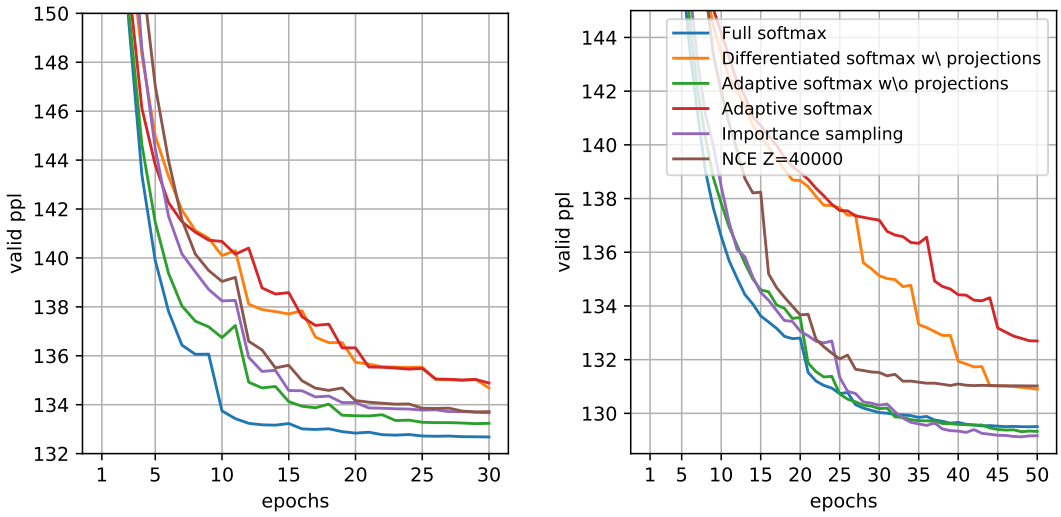


Figure 5.15: Effect of increasing vocabulary sizes on training speed of different softmax methods measured on 10% slice of FDT corpus after one pseudo epoch.



(a) Using 25% dropout rate

(b) Using 40% dropout rate

Figure 5.16: Comparison of alternative softmax methods on the Text8 corpus.

with the vocabulary size whereas the memory usage in the LSTM layer remains constant. The SGD optimizer has the highest memory consumption in both cases. Overall, switching from importance sampling to adaptive softmax in the output layer can half the total memory consumption of the model for large vocabulary sizes.

Figure 5.15 shows the relationship between training speed and vocabulary size. Importance sampling computes just the logits for the target and sampled words. Thus, the training speed is only effected by the sample set size. But larger vocabularies might require a larger sample set size to achieve optimal perplexity (see section 5.2.3.2). On the other hand, adaptive softmax has to compute the logits in at least one cluster. The sum of the cluster sizes must always match the vocabulary size and thus the average amount of logits computed by adaptive softmax will grow with the vocabulary size. For models with large LSTM layers this slow down is masked by the additional overhead. For a large vocabulary of 200,000 words adaptive softmax offers a similar speed up as importance sampling with 2000 samples.

5.3.2 Convergence and perplexity

This experiment compares the convergence speed and achieved perplexity of various alternative softmax methods. The main objective is to determine if the

alternative methods can match the perplexity of full softmax. Other important factors are the convergence speed and sensitivity to hyper-parameters.

5.3.2.1 Method

The standard training setup is used where initial models using the Text8 corpus were trained for 30 epochs with a 25% dropout rate. Later, it was discovered that a dropout rate of 40% yielded lower perplexity and the experiments were repeated and trained for 50 epochs.

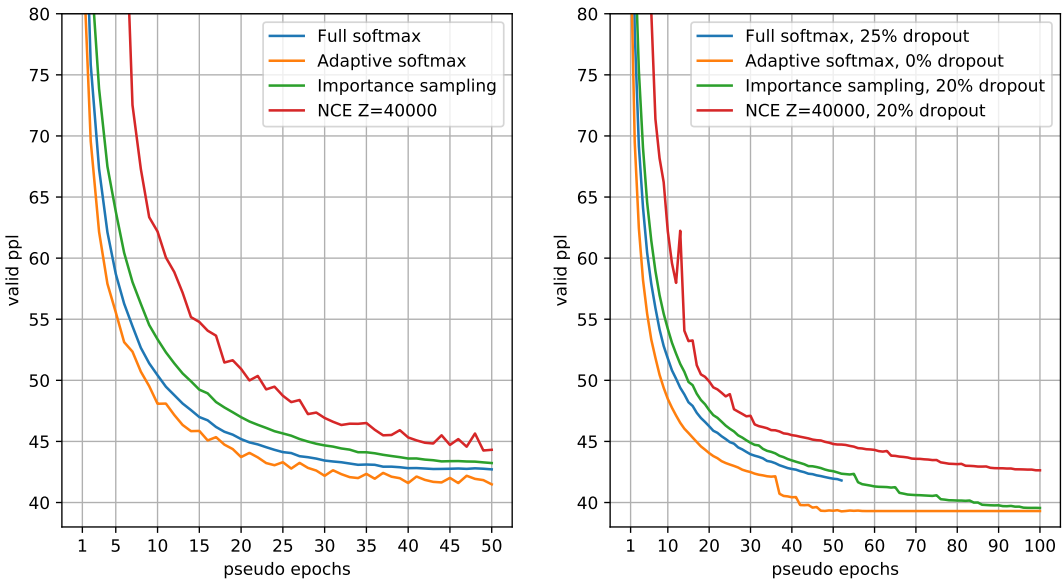
Most FDT models were trained on the 10% slice for 5 or 10 full epochs. The methods were compared against each other once with the same global hyper-parameter settings (SGD with $\eta = 2$ and no dropout) and a second time with hyper-parameters that were adjusted separately for each method [3] The final evaluation used the entire large-scale FDT corpus and was run on an additional machine with an older generation NVIDIA Tesla K80 GPU for 2 full epochs.

5.3.2.2 Results and discussion

Figures 5.16a and 5.16b illustrate the convergence of each method on the Text8 corpus using dropout rates of 25% and 40%, respectively. Table 5.2a reports the validation set perplexity at the end of the training period as well as the averaged training speed in WPS. The main observation is that adaptive softmax (without projection matrices) and importance sampling achieve similar competitive results. The speed ups attained by each method are noticeable with the adaptive softmax version without projection matrices being $\sim 16.5\%$ slower than importance sampling. In the scenario with 40% dropout NCE could not match the perplexity of full softmax.

Figures 5.17a and 5.17b show a similar comparison using models trained on a 10% slice of the FDT corpus. In the scenario with shared hyper-parameters adaptive softmax converged faster and achieved a lower perplexity than the other models. This could be due to the reduced number of trainable parameters and its regularization effect. Thus, in follow up experiments dropout was added to the other methods. Figure 5.17b plots the convergence using dropout and a learning rate schedule. With the additional regularization importance sampling reached the perplexity of adaptive softmax after 10 full epochs. These results confirm

[3] Due to the available computing resources no extensive grid search was possible.



(a) Using same global hyper-parameters (b) Methods optimized separately

Figure 5.17: Comparison of alternative softmax methods on a 10% slice of the FDT corpus. 10 pseudo epochs correspond to one full epoch over the training set.

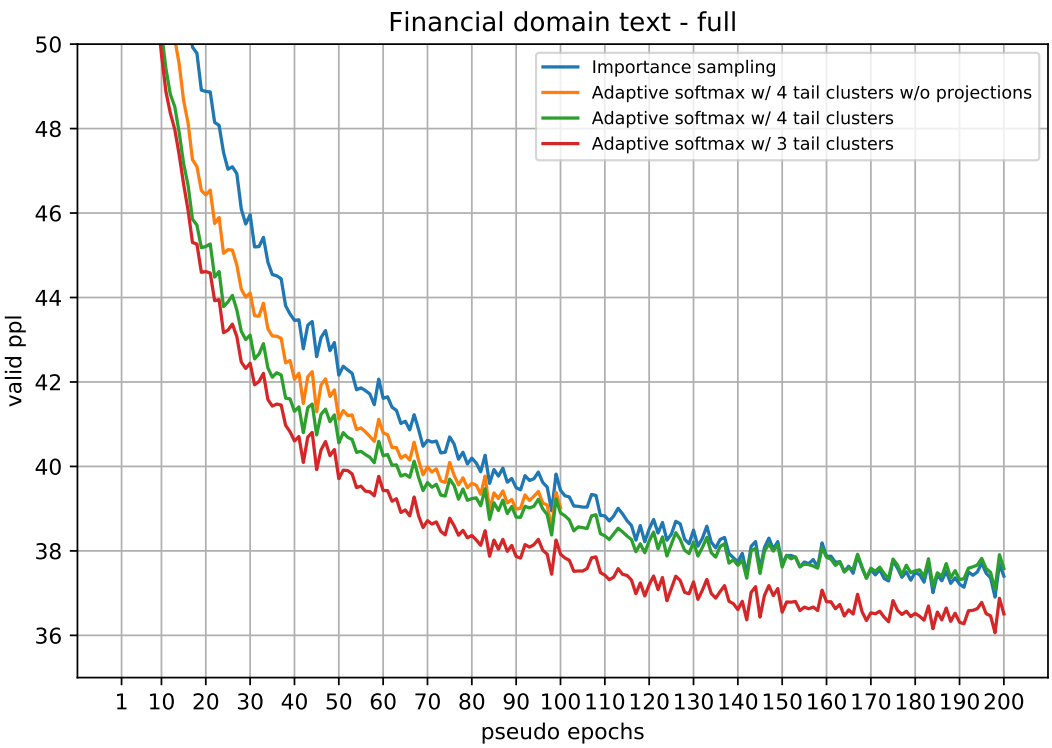


Figure 5.18: Comparison of adaptive softmax and importance sampling on full FDT corpus. 100 pseudo epochs correspond to one full epoch over the training set.

Method	WPS	PPL
Full softmax	22110	129.51
Adaptive softmax w/o projections	63256	129.33
Adaptive softmax	71203	132.69
Importance sampling	75749	129.17
NCE	78182	131.02

(a) Text8 corpus with 40% dropout

Method	WPS
Full softmax	3897
Adaptive softmax	21929
NCE	22188
Importance sampling	23304

(b) FDT corpus

Table 5.2: Comparison of training speed of alternative softmax methods.

that adaptive softmax and importance sampling are both capable of matching the perplexity of the full softmax. Further, Table 5.2b shows that both methods achieve similar training speed improvements. Interestingly, the dropout did not have a clear impact on NCE which still lies 5 perplexity points above adaptive softmax after half an epoch. Figure 5.18 shows the converge over 2 epochs on full FDT corpus.

A hypothesis was stated in section 5.2.3.2 that the standard NCE loss is not well suited to batch mode GPU implementations that share noise samples. This could provide one possible explanation for the overall discrepancy between importance sampling and NCE observed in the experiments. Retrospectively, it required extensive experimentation during this work to achieve reasonable results with NCE. This might also hint at a higher sensitivity to hyper-parameter choices.

A surprising result is that adaptive softmax with projection matrices did not achieve good results on Text8 but was the best method on the large-scale FDT corpus. As argued in section 5.1.1.2 the high probability mass of frequent words in FDT favors the shortlist in adaptive softmax. In the case of importance sampling a constant sample set size will cover each word less often due to the increased vocabulary size. On the other hand, increasing the sample set size will decrease the training throughput. Future work should further investigate the effect of vocabulary size and corpus on the accuracy of importance sampling.

5.3.3 N-best list re-scoring

Measuring validation or test set perplexity is an easy and efficient way to estimate the generalization error. This is an important intrinsic measure but in practice language models are integrated into larger ASR or MT systems to improve their

quality. In ASR the most important metric is the word error rate (WER) indicating the accuracy of output transcriptions. This experiment aims to establish the effect of alternative softmax output methods on this extrinsic measure.

5.3.3.1 Method

For a series of short utterances an n-best list of possible transcriptions has been provided. The n-best lists were extracted from a commercial ASR system trained on a large quantity of data from a variety of financial domains. For each utterance up to 10 alternative decodings were provided. These alternative sentences were fed to the previously trained language models on the FDT corpus and the log probability was computed as the sum over the words in each sentence. This was performed in a batched, continuous stream where sentences were separated by the EOS token. A script selected the best candidate transcription for each utterance according to the log probability. Finally, the output was compared to the reference transcriptions using the tool `sctk` which calculated an overall WER.

It should be noted that it was not expected to beat the original ASR system which was trained on a much larger quantity of data. Additionally, there is some mismatch between the vocabularies in the FDT corpus and the ASR system which causes additional OOV words.

5.3.3.2 Results and discussion

Table 5.3 shows the WER achieved on the down-stream re-scoring task for different models. Both adaptive softmax and importance sampling resulted in similar WERs. It is unclear if the small difference between adaptive softmax and importance sampling is statistically significant. As mentioned above it was not expected for the FDT language models to improve upon the ranking proposed by the ASR system.

5.4 cuDNN LSTM benchmark

Although the output softmax layer is the main computational bottleneck (Bengio et al., 2003a), a large multi-layer LSTM network can also slow down training times. As described in section 3.5 NVIDIA offers a GPU optimized implementation of LSTM networks as part of the NVIDIA cuDNN library. The following

Method	Test PPL	WER
1-best	N/A	21.6
Adaptive softmax	38.4	21.9
Importance sampling	38.2	22.0

Table 5.3: Re-scoring of n-best lists after training for 2 full epochs on FDT

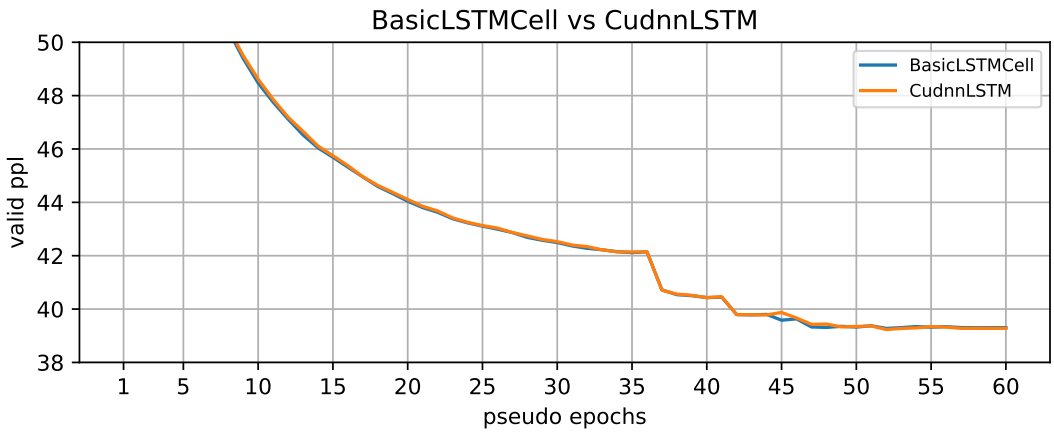


Figure 5.19: Comparison of `BasicLSTMCell` and `CudnnLSTM` convergence.

benchmarks attempt to quantify the speed up that can be obtained by replacing the basic Tensorflow LSTM implementation with the optimized cuDNN library.

5.4.1 Method

To measure the difference in training throughput `BasicLSTMCell` was replaced by `CudnnLSTM` in the Tensorflow implementation that was used to run all previous experiments. As before, the throughput was measured by timing the training of one pseudo epoch. The benchmarks were run using a NVIDIA GeForce GTX 1080 GPU.

5.4.2 Results and discussion

Tables 5.4a and 5.4a compares the training throughput achieved with the `BasicLSTMCell` and `CudnnLSTM` implementations. Switching to the `CudnnLSTM` implementation resulted in an WPS increase of 35% and 45% with the standard Text8 and FDT setup, respectively. In other tests a two-fold speed up of larger LSTM layers was observed. No difference in convergence between the two implementations could be observed (see Figure 5.19).

Implementation	1×512 LSTM Text8 SDG	1×2048 LSTM FDT SGD	1×512 LSTM Adagrad	1×512 LSTM SGD	2×2048 LSTM Adagrad	2×2048 LSTM SGD
Basic LSTM	84,000	22,000	78k	102k	8k	9k
cuDNN LSTM	115,000	32,000	76k	133k	13k	16k

(a) Adaptive softmax training (b) NCE training on Text8

Table 5.4: Comparison of training speed in WPS using the standard `BasicLSTMCell` and the optimized `CudnnLSTM` LSTM implementation.

Unfortunately, there are several caveats regarding the `CudnnLSTM` implementation. It features an incompatible application programming interface (API) compared to the standard RNN cells available in the `tf.contrib.rnn` package. `CudnnLSTM` represents an entire multi-layer LSTM network as a closed unit without any possibility of customization. The model parameters are stored as a single nontransparent tensor. The parameter tensor size is determined at runtime whereas many Tensorflow features require static shape information during construction of the computation graph. These include `tf.get_variable` as well as per-weight adaptive optimizers like Adam and Adagrad. There is some recent debate about the benefit of adaptive optimizers (Wilson et al., 2017) but a workaround[4] exists if training with SGD is not an option. The system architecture dependent tensor shape can be determined in a separate run with SGD and then hard-coded as a constant. Table 5.4b compares the WPSs throughput obtained with Adagrad and SGD on Text8.

Another obstacle making the transition to `CudnnLSTM` possibly more difficult is the requirement of time-major encoded batches. Finally, it has to be reported that using dropout values other than 0.0 resulted in uninformative error messages. `CudnnLSTM` is viable option if the main concern is to maximize the speed of standard models. Within research the above restrictions will limit its applicability.

[4] https://github.com/tensorflow/tensorflow/issues/6620

Chapter 6

Conclusion

This project investigated alternative softmax output methods and LSTM layer implementations in the context of neural language models. A particular emphasis was put on methods that accelerate training with large vocabulary corpora. An extensive literature review summarized the research in this field.

An empirical study compared the recently proposed adaptive softmax method with importance sampling and NCE. The performance of the selected methods was evaluated on two corpora of varying sizes. Text8 is a small to medium corpus and the FDT corpus served as an example of a large corpus.

All 3 methods provided substantial training speed improvements over the full softmax. The main differences between the methods were convergence speed, number of parameters, memory consumption and sensitivity to hyper-parameter choices. Adaptive softmax achieved the lowest perplexity on the large-scale FDT corpus though on the smaller Text8 corpus only a version without projection matrices matched the accuracy of the full softmax. It was shown that skewed frequencies of the top ranked words decreases the theoretical cost of adaptive softmax. When using projection matrices adaptive softmax has vastly fewer parameters than full softmax and the sampling methods. An ASR re-scoring task showed no difference between models trained with adaptive softmax and importance sampling. Profiling showed that importance sampling required twice the amount of memory than adaptive softmax for specific model parameters during training.

Importance sampling could match the accuracy of full softmax in all experiments. NCE showed a high sensitivity to the chosen normalization constant. For the standard setup on Text8 an extensive grid search was able to tune NCE to

achieve validation set perplexity on par with full softmax. But with increased dropout and in other experiments NCE did not match the low perplexity values of the other methods. The sharing of noise samples within a batch was pointed out as a possible cause. Future work is warranted to confirm this effect and investigate other explanations. In other experiments a gradual shift during training from uniform to unigram proposal distribution showed some promising results.

A detailed study of existing Tensorflow implementations revealed potential pitfalls and limitations. The choice between sampling with and without replacement changes the way the proposal distribution is calculated. In Tensorflow `1.2` only sampling without replacement leads to the actual NCE loss in `tf.nn.nce_loss`. Further, the implementation assumes a normalization constant $Z = 1$ which was shown to yield suboptimal results on the studied corpora. An open-source adaptive softmax implementation was augmented and a new method was devised to calculate the logits over the entire vocabulary. Finally, benchmarks with the highly optimized cuDNN LSTM implementation showed considerable speed ups for large LSTM networks but various restrictions can make it difficult to integrate into existing code.

The experimental results highlighted that it is crucial to train models until convergence before claims about the model's final accuracy can be made. The large impact of hyper-parameter choices confirms that extra care needs to be taken before drawing firm conclusions from empirical results Melis et al. (2017).

In future work the computational overhead of adaptive softmax might be decreased further with a native GPU implementation fusing multiple Tensorflow ops. Another potential optimization is to precompute separate batch tensors for each tail cluster in the preprocessing step. This avoids repeated computation of boolean masks during the training. Choosing the adaptive softmax cluster sizes involved a trade-off between training speed and model accuracy. Testing various settings on large corpora is often not feasible and additional work is required to devise good guidelines. Finally, there has been little or no work investigating the effect of adaptive softmax on extrinsic tasks in ASR and MT systems.

Bibliography

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., et al. (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*.

Andreas, J. and Klein, D. (2015). When and why are log-linear models self-normalizing? In *HLT-NAACL*, pages 244–249.

Andreas, J., Rabinovich, M., Jordan, M. I., and Klein, D. (2015). On the accuracy of self-normalized log-linear models. In *Advances in Neural Information Processing Systems*, pages 1783–1791.

Appleyard, J., Kocisky, T., and Blunsom, P. (2016). Optimizing performance of recurrent neural networks on GPUs. *arXiv preprint arXiv:1604.01946*.

Auvolat, A., Chandar, S., Vincent, P., Larochelle, H., and Bengio, Y. (2015). Clustering is efficient for approximate maximum inner product search. *arXiv preprint arXiv:1507.05910*.

Baltescu, P. and Blunsom, P. (2014). Pragmatic neural language modelling in machine translation. *arXiv preprint arXiv:1412.7119*.

Bengio, Y., Boulanger-Lewandowski, N., and Pascanu, R. (2013). Advances in optimizing recurrent networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8624–8628. IEEE.

Bengio, Y., Ducharme, R., Vincent, P., and Jauvin, C. (2003a). A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155.

Bengio, Y. and Senécal, J.-S. (2008). Adaptive importance sampling to accelerate training of a neural probabilistic language model. *IEEE Transactions on Neural Networks*, 19(4):713–722.

Bengio, Y., Senécal, J.-S., et al. (2003b). Quick training of probabilistic neural nets by importance sampling. In *AISTATS*.

Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166.

Bishop, C. M. (1995). *Neural networks for pattern recognition.* Oxford university press.

Brown, P. F., deSouza, P. V., Mercer, R. L., Pietra, V. J. D., and Lai, J. C. (1992). Class-based n-gram models of natural language. *Computational linguistics*, 18(4):467–479.

Chelba, C., Mikolov, T., Schuster, M., Ge, Q., Brants, T., Koehn, P., and Robinson, T. (2013). One Billion word benchmark for measuring progress in statistical language modeling. *arXiv preprint arXiv:1312.3005.*

Chen, S. F. and Goodman, J. (1996). An empirical study of smoothing techniques for language modeling. In *Proceedings of the 34th annual meeting on Association for Computational Linguistics*, pages 310–318. Association for Computational Linguistics.

Chen, W., Grangier, D., and Auli, M. (2015a). Strategies for training large vocabulary neural language models. *arXiv preprint arXiv:1512.04906.*

Chen, X., Liu, X., Gales, M. J., and Woodland, P. C. (2015b). Recurrent neural network language model training with noise contrastive estimation for speech recognition. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, pages 5411–5415. IEEE.

Chen, X., Liu, X., Qian, Y., Gales, M., and Woodland, P. C. (2016). CUED-RNNLM - an open-source toolkit for efficient training and evaluation of recurrent neural network language models. In *Acoustics, Speech and Signal Processing (ICASSP), 2016 IEEE International Conference on*, pages 6000–6004. IEEE.

Cho, K., Van Merriënboer, B., Bahdanau, D., and Bengio, Y. (2014). On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259.*

de Brébisson, A. and Vincent, P. (2015). An exploration of softmax alternatives belonging to the spherical loss family. *arXiv preprint arXiv:1511.05042.*

Dempster, A. P., Laird, N. M., and Rubin, D. B. (1977). Maximum likelihood from incomplete data via the EM algorithm. *Journal of the royal statistical society. Series B (methodological)*, pages 1–38.

Devlin, J., Zbib, R., Huang, Z., Lamar, T., Schwartz, R. M., and Makhoul, J. (2014). Fast and robust neural network joint models for statistical machine translation. In *ACL (1)*, pages 1370–1380. Citeseer.

Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159.

Dyer, C. (2014). Notes on noise contrastive estimation and negative sampling. *arXiv preprint arXiv:1410.8251.*

Elman, J. L. (1990). Finding structure in time. *Cognitive science*, 14(2):179–211.

Gers, F. A. and Schmidhuber, J. (2000). Recurrent nets that time and count. In *Neural Networks, 2000. IJCNN 2000, Proceedings of the IEEE-INNS-ENNS International Joint Conference on*, volume 3, pages 189–194. IEEE.

Gers, F. A., Schmidhuber, J., and Cummins, F. (1999). Learning to forget: Continual prediction with LSTM.

Goodman, J. (2001). Classes for fast maximum entropy training. In *Acoustics, Speech, and Signal Processing, 2001. Proceedings.(ICASSP'01). 2001 IEEE International Conference on*, volume 1, pages 561–564. IEEE.

Grave, E., Joulin, A., Cissé, M., Grangier, D., and Jégou, H. (2016). Efficient softmax approximation for GPUs. *arXiv preprint arXiv:1609.04309*.

Graves, A. (2013). Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*.

Graves, A., Mohamed, A.-r., and Hinton, G. (2013). Speech recognition with deep recurrent neural networks. In *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*, pages 6645–6649. IEEE.

Greff, K., Srivastava, R. K., Koutník, J., Steunebrink, B. R., and Schmidhuber, J. (2016). LSTM: A search space odyssey. *IEEE transactions on neural networks and learning systems*.

Gutmann, M. and Hyvärinen, A. (2010). Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. In *AISTATS*, volume 1, page 6.

Gutmann, M. U. and Hyvärinen, A. (2012). Noise-contrastive estimation of unnormalized statistical models, with applications to natural image statistics. *Journal of Machine Learning Research*, 13(Feb):307–361.

He, T., Zhang, Y., Droppo, J., and Yu, K. (2016). On training bi-directional neural network language model with noise contrastive estimation. *arXiv preprint arXiv:1602.06064*.

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

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

Hutter, M. (2006). The human knowledge compression prize.

Jean, S., Cho, K., Memisevic, R., and Bengio, Y. (2014). On using very large target vocabulary for neural machine translation. In *ACL 2015*.

Jelinek, F. (1980). Interpolated estimation of Markov source parameters from sparse data. In *Proc. Workshop on Pattern Recognition in Practice, 1980*.

Ji, S., Vishwanathan, S., Satish, N., Anderson, M. J., and Dubey, P. (2015). Blackout: Speeding up recurrent neural network language models with very large vocabularies. *arXiv preprint arXiv:1511.06909*.

Jozefowicz, R., Vinyals, O., Schuster, M., Shazeer, N., and Wu, Y. (2016). Exploring the limits of language modeling. *arXiv preprint arXiv:1602.02410*.

Karpathy, A. (2016). The unreasonable effectiveness of recurrent neural networks, 2015. `http://karpathy.github.io/2015/05/21/rnn-effectiveness` [Accessed on 9 Aug 2017].

Katz, S. (1987). Estimation of probabilities from sparse data for the language model component of a speech recognizer. *IEEE transactions on acoustics, speech, and signal processing*, 35(3):400–401.

Kilgarriff, A. (2000). WordNet: An electronic lexical database.

Kim, Y., Jernite, Y., Sontag, D., and Rush, A. M. (2015). Character-aware neural language models. *CoRR*, abs/1508.06615.

Kneser, R. and Ney, H. (1993). Improved clustering techniques for class-based statistical language modelling. In *Eurospeech*, volume 93, pages 973–76.

Kneser, R. and Ney, H. (1995). Improved backing-off for m-gram language modeling. In *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, volume 1, pages 181–184. IEEE.

Kong, A. (1992). A note on importance sampling using standardized weights. *University of Chicago, Dept. of Statistics, Tech. Rep*, 348.

Labeau, M. and Allauzen, A. (2017). An experimental analysis of noise-contrastive estimation: the noise distribution matters. *EACL 2017*, page 15.

Le, H.-S., Allauzen, A., and Yvon, F. (2012). Continuous space translation models with neural networks. In *Proceedings of the 2012 conference of the north american chapter of the association for computational linguistics: Human language technologies*, pages 39–48. Association for Computational Linguistics.

Le, H.-S., Oparin, I., Allauzen, A., Gauvain, J.-L., and Yvon, F. (2011a). Structured output layer neural network language model. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, pages 5524–5527. IEEE.

Le, H.-S., Oparin, I., Allauzen, A., Gauvain, J.-L., and Yvon, F. (2013). Structured output layer neural network language models for speech recognition. *IEEE Transactions on Audio, Speech, and Language Processing*, 21(1):197–206.

Le, H.-S., Oparin, I., Messaoudi, A., Allauzen, A., Gauvain, J.-L., and Yvon, F. (2011b). Large vocabulary SOUL neural network language models. In *Twelfth Annual Conference of the International Speech Communication Association*.

LeCun, Y., Boser, B. E., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W. E., and Jackel, L. D. (1990). Handwritten digit recognition with a back-propagation network. In *Advances in neural information processing systems*, pages 396–404.

Ling, W., Luís, T., Marujo, L., Astudillo, R. F., Amir, S., Dyer, C., Black, A. W., and Trancoso, I. (2015). Finding function in form: Compositional character models for open vocabulary word representation. *arXiv preprint arXiv:1508.02096*.

Melamud, O. and Goldberger, J. (2017). Information-theory interpretation of the skip-gram negative-sampling objective function. In *Proceedings of ACL*.

Melis, G., Dyer, C., and Blunsom, P. (2017). On the state of the art of evaluation in neural language models. *arXiv preprint arXiv:1707.05589*.

Mikolov, T., Joulin, A., Chopra, S., Mathieu, M., and Ranzato, M. (2014). Learning longer memory in recurrent neural networks. *arXiv preprint arXiv:1412.7753*.

Mikolov, T., Karafiát, M., Burget, L., Cernockỳ, J., and Khudanpur, S. (2010). Recurrent neural network based language model. In *Interspeech*, volume 2, page 3.

Mikolov, T., Kombrink, S., Burget, L., Černockỳ, J., and Khudanpur, S. (2011). Extensions of recurrent neural network language model. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, pages 5528–5531. IEEE.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. (2013a). Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119.

Mikolov, T., Yih, W.-t., and Zweig, G. (2013b). Linguistic regularities in continuous space word representations. In *hlt-Naacl*, volume 13, pages 746–751.

Mnih, A. and Hinton, G. E. (2009). A scalable hierarchical distributed language model. In *Advances in neural information processing systems*, pages 1081–1088.

Mnih, A. and Kavukcuoglu, K. (2013). Learning word embeddings efficiently with noise-contrastive estimation. In *Advances in neural information processing systems*, pages 2265–2273.

Mnih, A. and Teh, Y. W. (2012). A fast and simple algorithm for training neural probabilistic language models. *arXiv preprint arXiv:1206.6426*.

Morin, F. and Bengio, Y. (2005). Hierarchical probabilistic neural network language model. In *Aistats*, volume 5, pages 246–252. Citeseer.

Oland, A., Bansal, A., Dannenberg, R. B., and Raj, B. (2017). Be careful what you backpropagate: A case for linear output activations & gradient boosting. *arXiv preprint arXiv:1707.04199*.

Ollivier, Y. (2015). Riemannian metrics for neural networks I: feedforward networks. *Information and Inference: A Journal of the IMA*, 4(2):108–153.

Park, J., Liu, X., Gales, M. J., and Woodland, P. C. (2010). Improved neural network based language modelling and adaptation. In *INTERSPEECH*, volume 10, pages 1041–1044.

Parker, R., Graff, D., Kong, J., Chen, K., and Maeda, K. (2011). English Gigaword fifth edition, Linguistic Data Consortium. Technical report, Technical report, Technical Report. Linguistic Data Consortium, Philadelphia.

Pascanu, R., Mikolov, T., and Bengio, Y. (2012). Understanding the exploding gradient problem. *CoRR, abs/1211.5063*.

Ram, P. and Gray, A. G. (2012). Maximum inner-product search using cone trees. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 931–939. ACM.

Ruder, S. (2016). On word embeddings - Part 2: Approximating the softmax. `http://sebastianruder.com/word-embeddings-softmax` [Accessed on 9 Aug 2017].

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1985). Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science.

Schwenk, H. (2004). Efficient training of large neural networks for language modeling. In *Neural Networks, 2004. Proceedings. 2004 IEEE International Joint Conference on*, volume 4, pages 3059–3064. IEEE.

Schwenk, H. (2007). Continuous space language models. *Computer Speech & Language*, 21(3):492–518.

Schwenk, H. and Gauvain, J.-L. (2004). Neural network language models for conversational speech recognition. In *INTERSPEECH*.

Schwenk, H. and Gauvain, J.-L. (2005). Training neural network language models on very large corpora. In *Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing*, pages 201–208. Association for Computational Linguistics.

Shen, Y., Tan, S., Pal, C., and Courville, A. (2017). Self-organized hierarchical softmax. *arXiv preprint arXiv:1707.08588*.

Shrivastava, A. and Li, P. (2014). Asymmetric LSH (ALSH) for sublinear time maximum inner product search (MIPS). In *Advances in Neural Information Processing Systems*, pages 2321–2329.

Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958.

Srivastava, R. K., Greff, K., and Schmidhuber, J. (2015). Training very deep networks. In *Advances in neural information processing systems*, pages 2377–2385.

Sundermeyer, M., Schlüter, R., and Ney, H. (2012). LSTM neural networks for language modeling. In *Interspeech*, pages 194–197.

Sutskever, I., Martens, J., and Hinton, G. E. (2011). Generating text with recurrent neural networks. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 1017–1024.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112.

Vaswani, A., Zhao, Y., Fossum, V., and Chiang, D. (2013). Decoding with large-scale neural language models improves translation. In *EMNLP*, pages 1387–1392. Citeseer.

Vijayanarasimhan, S., Shlens, J., Monga, R., and Yagnik, J. (2014). Deep networks with large output spaces. *arXiv preprint arXiv:1412.7479*.

Vincent, P., de Brébisson, A., and Bouthillier, X. (2015). Efficient exact gradient update for training deep networks with very large sparse targets. In *Advances in Neural Information Processing Systems*, pages 1108–1116.

Werbos, P. J. (1988). Generalization of backpropagation with application to a recurrent gas market model. *Neural networks*, 1(4):339–356.

Williams, W., Prasad, N., Mrva, D., Ash, T., and Robinson, T. (2015). Scaling recurrent neural network language models. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, pages 5391–5395. IEEE.

Wilson, A. C., Roelofs, R., Stern, M., Srebro, N., and Recht, B. (2017). The marginal value of adaptive gradient methods in machine learning. *arXiv preprint arXiv:1705.08292*.

Yagnik, J., Strelow, D., Ross, D. A., and Lin, R.-s. (2011). The power of comparative reasoning. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 2431–2438. IEEE.

Yu, B., Goldman, S., and Zhang, L. (2017). TAPAS: Two-pass approximate adaptive sampling for softmax. *arXiv preprint arXiv:1707.03073*.

Zaremba, W., Sutskever, I., and Vinyals, O. (2014). Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*.

Zipf, G. (1949). The principle of least effort: An introduction to human ecology.

Zipser, D. (1990). Subgrouping reduces complexity and speeds up learning in recurrent networks. In *Advances in neural information processing systems*, pages 638–641.

Zoph, B., Vaswani, A., May, J., and Knight, K. (2016). Simple, fast noise-contrastive estimation for large RNN vocabularies. In *Proceedings of NAACL-HLT*, pages 1217–1222.

Zweig, G. and Makarychev, K. (2013). Speed regularization and optimality in word classing. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8237–8241. IEEE.