Large Language Models in Machine Translation

Thorsten Brants **Ashok C. Popat** **Peng Xu** **Franz J. Och** **Jeffrey Dean**

Google, Inc.
1600 Amphitheatre Parkway
Mountain View, CA 94303, USA
{brants,popat,xp,och,jeff}@google.com

Abstract

This paper reports on the benefits of large-scale statistical language modeling in machine translation. A distributed infrastructure is proposed which we use to train on up to 2 trillion tokens, resulting in language models having up to 300 billion n-grams. It is capable of providing smoothed probabilities for fast, single-pass decoding. We introduce a new smoothing method, dubbed *Stupid Backoff*, that is inexpensive to train on large data sets and approaches the quality of Kneser-Ney Smoothing as the amount of training data increases.

1 Introduction

Given a source-language (e.g., French) sentence $\mathbf{f}$, the problem of *machine translation* is to automatically produce a target-language (e.g., English) translation $\hat{\mathbf{e}}$. The mathematics of the problem were formalized by (Brown et al., 1993), and re-formulated by (Och and Ney, 2004) in terms of the optimization

$$\hat{\mathbf{e}} = \arg\max_{\mathbf{e}} \sum_{m=1}^{M} \lambda_m h_m(\mathbf{e}, \mathbf{f}) \tag{1}$$

where $\{h_m(\mathbf{e}, \mathbf{f})\}$ is a set of M *feature functions* and $\{\lambda_m\}$ a set of *weights*. One or more feature functions may be of the form $h(\mathbf{e}, \mathbf{f}) = h(\mathbf{e})$, in which case it is referred to as a *language model*.

We focus on n-gram language models, which are trained on unlabeled monolingual text. As a general rule, more data tends to yield better language models. Questions that arise in this context include: (1) How might one build a language model that allows scaling to very large amounts of training data? (2) How much does translation performance improve as the size of the language model increases? (3) Is there a point of diminishing returns in performance as a function of language model size?

This paper proposes one possible answer to the first question, explores the second by providing learning curves in the context of a particular statistical machine translation system, and hints that the third may yet be some time in answering. In particular, it proposes a *distributed* language model training and deployment infrastructure, which allows direct and efficient integration into the hypothesis-search algorithm rather than a follow-on re-scoring phase. While it is generally recognized that two-pass decoding can be very effective in practice, single-pass decoding remains conceptually attractive because it eliminates a source of potential information loss.

2 N-gram Language Models

Traditionally, statistical language models have been designed to assign probabilities to strings of words (or tokens, which may include punctuation, etc.). Let $w_1^L = (w_1, \ldots, w_L)$ denote a string of L tokens over a fixed vocabulary. An *n-gram language model* assigns a probability to w_1^L according to

$$P(w_1^L) = \prod_{i=1}^{L} P(w_i | w_1^{i-1}) \approx \prod_{i=1}^{L} \hat{P}(w_i | w_{i-n+1}^{i-1}) \tag{2}$$

where the approximation reflects a Markov assumption that only the most recent $n-1$ tokens are relevant when predicting the next word.

Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, pp. 858–867, Prague, June 2007.

For any substring w_i^j of w_1^L, let $f(w_i^j)$ denote the frequency of occurrence of that substring in another given, fixed, usually very long target-language string called the *training data*. The maximum-likelihood (ML) probability estimates for the n-grams are given by their relative frequencies

$$r(w_i|w_{i-n+1}^{i-1}) = \frac{f(w_{i-n+1}^{i})}{f(w_{i-n+1}^{i-1})}. \tag{3}$$

While intuitively appealing, Eq. (3) is problematic because the denominator and / or numerator might be zero, leading to inaccurate or undefined probability estimates. This is termed the *sparse data* problem. For this reason, the ML estimate must be modified for use in practice; see (Goodman, 2001) for a discussion of n-gram models and smoothing.

In principle, the predictive accuracy of the language model can be improved by increasing the order of the n-gram. However, doing so further exacerbates the sparse data problem. The present work addresses the challenges of processing an amount of training data sufficient for higher-order n-gram models and of storing and managing the resulting values for efficient use by the decoder.

3 Related Work on Distributed Language Models

The topic of large, distributed language models is relatively new. Recently a two-pass approach has been proposed (Zhang et al., 2006), wherein a lower-order n-gram is used in a hypothesis-generation phase, then later the K-best of these hypotheses are re-scored using a large-scale distributed language model. The resulting translation performance was shown to improve appreciably over the hypothesis deemed best by the first-stage system. The amount of data used was 3 billion words.

More recently, a large-scale distributed language model has been proposed in the contexts of speech recognition and machine translation (Emami et al., 2007). The underlying architecture is similar to (Zhang et al., 2006). The difference is that they integrate the distributed language model into their machine translation decoder. However, they don't report details of the integration or the efficiency of the approach. The largest amount of data used in the experiments is 4 billion words.

Both approaches differ from ours in that they store corpora in suffix arrays, one sub-corpus per worker, and serve raw counts. This implies that all workers need to be contacted for each n-gram request. In our approach, smoothed probabilities are stored and served, resulting in exactly one worker being contacted per n-gram for simple smoothing techniques, and in exactly two workers for smoothing techniques that require context-dependent backoff. Furthermore, suffix arrays require on the order of 8 bytes per token. Directly storing 5-grams is more efficient (see Section 7.2) and allows applying count cutoffs, further reducing the size of the model.

4 Stupid Backoff

State-of-the-art smoothing uses variations of context-dependent backoff with the following scheme:

$$P(w_i|w_{i-k+1}^{i-1}) =$$
$$\begin{cases} \rho(w_{i-k+1}^{i}) & \text{if } (w_{i-k+1}^{i}) \text{ is found} \\ \lambda(w_{i-k+1}^{i-1})P(w_{i-k+2}^{i}) & \text{otherwise} \end{cases} \tag{4}$$

where $\rho(\cdot)$ are pre-computed and stored probabilities, and $\lambda(\cdot)$ are back-off weights. As examples, Kneser-Ney Smoothing (Kneser and Ney, 1995), Katz Backoff (Katz, 1987) and linear interpolation (Jelinek and Mercer, 1980) can be expressed in this scheme (Chen and Goodman, 1998). The recursion ends at either unigrams or at the uniform distribution for zero-grams.

We introduce a similar but simpler scheme, named *Stupid Backoff*[1], that does not generate normalized probabilities. The main difference is that we don't apply any discounting and instead directly use the relative frequencies (S is used instead of P to emphasize that these are not probabilities but scores):

$$S(w_i|w_{i-k+1}^{i-1}) =$$
$$\begin{cases} \dfrac{f(w_{i-k+1}^{i})}{f(w_{i-k+1}^{i-1})} & \text{if } f(w_{i-k+1}^{i}) > 0 \\ \alpha S(w_i|w_{i-k+2}^{i-1}) & \text{otherwise} \end{cases} \tag{5}$$

[1] The name originated at a time when we thought that such a simple scheme cannot possibly be good. Our view of the scheme changed, but the name stuck.

In general, the backoff factor α may be made to depend on k. Here, a single value is used and heuristically set to $\alpha = 0.4$ in all our experiments[2]. The recursion ends at unigrams:

$$S(w_i) = \frac{f(w_i)}{N} \tag{6}$$

with N being the size of the training corpus.

Stupid Backoff is inexpensive to calculate in a distributed environment while approaching the quality of Kneser-Ney smoothing for large amounts of data. The lack of normalization in Eq. (5) does not affect the functioning of the language model in the present setting, as Eq. (1) depends on relative rather than absolute feature-function values.

5 Distributed Training

We use the *MapReduce* programming model (Dean and Ghemawat, 2004) to train on terabytes of data and to generate terabytes of language models. In this programming model, a user-specified *map* function processes an input key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function aggregates all intermediate values associated with the same key. Typically, multiple *map* tasks operate independently on different machines and on different parts of the input data. Similarly, multiple *reduce* tasks operate independently on a fraction of the intermediate data, which is partitioned according to the intermediate keys to ensure that the same reducer sees all values for a given key. For additional details, such as communication among machines, data structures and application examples, the reader is referred to (Dean and Ghemawat, 2004).

Our system generates language models in three main steps, as described in the following sections.

5.1 Vocabulary Generation

Vocabulary generation determines a mapping of terms to integer IDs, so n-grams can be stored using IDs. This allows better compression than the original terms. We assign IDs according to term frequency, with frequent terms receiving small IDs for efficient variable-length encoding. All words that occur less often than a pre-determined threshold are mapped to a special id marking the unknown word.

The vocabulary generation *map* function reads training text as input. Keys are irrelevant; values are text. It emits intermediate data where keys are terms and values are their counts in the current section of the text. A *sharding* function determines which shard (chunk of data in the *MapReduce* framework) the pair is sent to. This ensures that all pairs with the same key are sent to the same shard. The *reduce* function receives all pairs that share the same key and sums up the counts. Simplified, the *map*, *sharding* and *reduce* functions do the following:

```
Map(string key, string value) {
  // key=docid, ignored; value=document
  array words = Tokenize(value);
  hash_map<string, int> histo;
  for i = 1 .. #words
    histo[words[i]]++;
  for iter in histo
    Emit(iter.first, iter.second);
}

int ShardForKey(string key, int nshards) {
  return Hash(key) % nshards;
}

Reduce(string key, iterator values) {
  // key=term; values=counts
  int sum = 0;
  for each v in values
    sum += ParseInt(v);
  Emit(AsString(sum));
}
```

Note that the *Reduce* function emits only the aggregated value. The output key is the same as the intermediate key and automatically written by *MapReduce*. The computation of counts in the *map* function is a minor optimization over the alternative of simply emitting a count of one for each tokenized word in the array. Figure 1 shows an example for 3 input documents and 2 reduce shards. Which reducer a particular term is sent to is determined by a hash function, indicated by text color. The exact partitioning of the keys is irrelevant; important is that all pairs with the same key are sent to the same reducer.

5.2 Generation of n-Grams

The process of n-gram generation is similar to vocabulary generation. The main differences are that now words are converted to IDs, and we emit n-grams up to some maximum order instead of single

[2] The value of 0.4 was chosen empirically based on good results in earlier experiments. Using multiple values depending on the n-gram order slightly improves results.

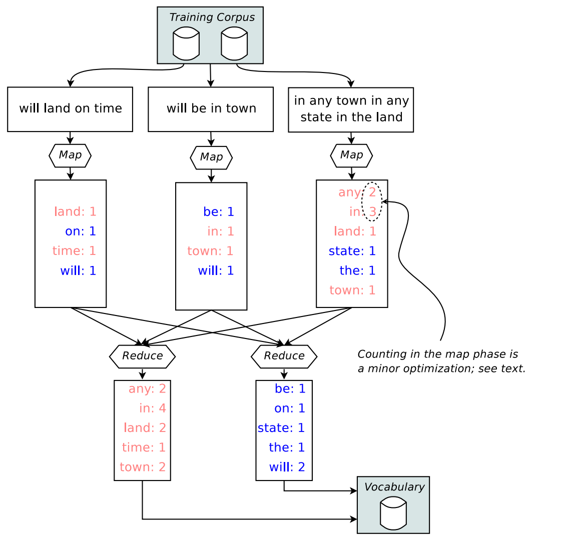

Figure 1: Distributed vocabulary generation.

words. A simplified *map* function does the following:

```
Map(string key, string value) {
  // key=docid, ignored; value=document
  array ids = ToIds(Tokenize(value));
  for i = 1 .. #ids
    for j = 0 .. maxorder-1
      Emit(ids[i-j .. i], "1");
}
```

Again, one may optimize the *Map* function by first aggregating counts over some section of the data and then emit the aggregated counts instead of emitting "1" each time an n-gram is encountered.

The reduce function is the same as for vocabulary generation. The subsequent step of language model generation will calculate relative frequencies $r(w_i|w_{i-k+1}^{i-1})$ (see Eq. 3). In order to make that step efficient we use a sharding function that places the values needed for the numerator and denominator into the same shard.

Computing a hash function on just the first words of n-grams achieves this goal. The required n-grams w_{i-n+1}^{i} and w_{i-n+1}^{i-1} always share the same first word w_{i-n+1}, except for unigrams. For that we need to communicate the total count N to all shards.

Unfortunately, sharding based on the first word only may make the shards very imbalanced. Some terms can be found at the beginning of a huge number of n-grams, e.g. stopwords, some punctuation marks, or the beginning-of-sentence marker. As an example, the shard receiving n-grams starting with the beginning-of-sentence marker tends to be several times the average size. Making the shards evenly sized is desirable because the total runtime of the process is determined by the largest shard.

The shards are made more balanced by hashing based on the first *two* words:

```
int ShardForKey(string key, int nshards) {
  string prefix = FirstTwoWords(key);
  return Hash(prefix) % nshards;
}
```

This requires redundantly storing unigram counts in all shards in order to be able to calculate relative frequencies within shards. That is a relatively small amount of information (a few million entries, compared to up to hundreds of billions of n-grams).

5.3 Language Model Generation

The input to the language model generation step is the output of the n-gram generation step: n-grams and their counts. All information necessary to calculate relative frequencies is available within individual shards because of the sharding function. That is everything we need to generate models with Stupid Backoff. More complex smoothing methods require additional steps (see below).

Backoff operations are needed when the full n-gram is not found. If $r(w_i|w_{i-n+1}^{i-1})$ is not found, then we will successively look for $r(w_i|w_{i-n+2}^{i-1})$, $r(w_i|w_{i-n+3}^{i-1})$, etc. The language model generation step shards n-grams on their last two words (with unigrams duplicated), so all backoff operations can be done within the same shard (note that the required n-grams all share the same last word w_i).

5.4 Other Smoothing Methods

State-of-the-art techniques like Kneser-Ney Smoothing or Katz Backoff require additional, more expensive steps. At runtime, the client needs to additionally request up to 4 backoff factors for each 5-gram requested from the servers, thereby multiplying network traffic. We are not aware of a method that always stores the history backoff factors on the same shard as the longer n-gram without duplicating a large fraction of the entries. This means one needs to contact two shards per n-gram instead of just one for Stupid Backoff. Training requires additional iterations over the data.

	Step 0 context counting	Step 1 unsmoothed probs and interpol. weights	Step 2 interpolated probabilities
Input key	w_{i-n+1}^{i}	(same as Step 0 output)	(same as Step 1 output)
Input value	$f(w_{i-n+1}^{i})$	(same as Step 0 output)	(same as Step 1 output)
Intermediate key	w_{i-n+1}^{i}	w_{i-n+1}^{i-1}	w_{i}^{i-n+1}
Sharding	w_{i-n+1}^{i}	w_{i-n+1}^{i-1}	w_{i-n+1}^{i-n+2}, unigrams duplicated
Intermediate value	$f_{KN}(w_{i-n+1}^{i})$	w_i,$f_{KN}(w_{i-n+1}^{i})$	$\frac{f_{KN}(w_{i-n+1}^{i})-D}{f_{KN}(w_{i-n+1}^{i-1})}$,$\lambda(w_{i-n+1}^{i-1})$
Output value	$f_{KN}(w_{i-n+1}^{i})$	w_i,$\frac{f_{KN}(w_{i-n+1}^{i})-D}{f_{KN}(w_{i-n+1}^{i-1})}$,$\lambda(w_{i-n+1}^{i-1})$	$P_{KN}(w_i \mid w_{i-n+1}^{i-1})$, $\lambda(w_{i-n+1}^{i-1})$

Table 1: Extra steps needed for training Interpolated Kneser-Ney Smoothing

Kneser-Ney Smoothing counts lower-order n-grams differently. Instead of the frequency of the $(n-1)$-gram, it uses the number of unique single word contexts the $(n-1)$-gram appears in. We use $f_{KN}(\cdot)$ to jointly denote original frequencies for the highest order and context counts for lower orders. After the n-gram counting step, we process the n-grams again to produce these quantities. This can be done similarly to the n-gram counting using a *MapReduce* (Step 0 in Table 1).

The most commonly used variant of Kneser-Ney smoothing is interpolated Kneser-Ney smoothing, defined recursively as (Chen and Goodman, 1998):

$$P_{KN}(w_i|w_{i-n+1}^{i-1}) = \frac{\max(f_{KN}(w_{i-n+1}^{i}) - D, 0)}{f_{KN}(w_{i-n+1}^{i-1})} + \lambda(w_{i-n+1}^{i-1}) P_{KN}(w_i|w_{i-n+2}^{i-1}),$$

where D is a discount constant and $\{\lambda(w_{i-n+1}^{i-1})\}$ are interpolation weights that ensure probabilities sum to one. Two additional major MapReduces are required to compute these values efficiently. Table 1 describes their input, intermediate and output keys and values. Note that output keys are always the same as intermediate keys.

The *map* function of MapReduce 1 emits n-gram histories as intermediate keys, so the *reduce* function gets all n-grams with the same history at the same time, generating unsmoothed probabilities and interpolation weights. MapReduce 2 computes the interpolation. Its *map* function emits reversed n-grams as intermediate keys (hence we use w_i^{i-n+1} in the table). All unigrams are duplicated in every reduce shard. Because the *reducer* function receives intermediate keys in sorted order it can compute smoothed probabilities for all n-gram orders with simple book-keeping.

Katz Backoff requires similar additional steps. The largest models reported here with Kneser-Ney Smoothing were trained on 31 billion tokens. For Stupid Backoff, we were able to use more than 60 times of that amount.

6 Distributed Application

Our goal is to use distributed language models integrated into the first pass of a decoder. This may yield better results than n-best list or lattice rescoring (Ney and Ortmanns, 1999). Doing that for language models that reside in the same machine as the decoder is straight-forward. The decoder accesses n-grams whenever necessary. This is inefficient in a distributed system because network latency causes a constant overhead on the order of milliseconds. Onboard memory is around 10,000 times faster.

We therefore implemented a new decoder architecture. The decoder first queues some number of requests, e.g. 1,000 or 10,000 n-grams, and then sends them together to the servers, thereby exploiting the fact that network requests with large numbers of n-grams take roughly the same time to complete as requests with single n-grams.

The n-best search of our machine translation decoder proceeds as follows. It maintains a graph of the search space up to some point. It then extends each hypothesis by advancing one word position in the source language, resulting in a candidate extension of the hypothesis of zero, one, or more additional target-language words (accounting for the fact that variable-length source-language fragments can correspond to variable-length target-language fragments). In a traditional setting with a local language model, the decoder immediately obtains the necessary probabilities and then (together with scores

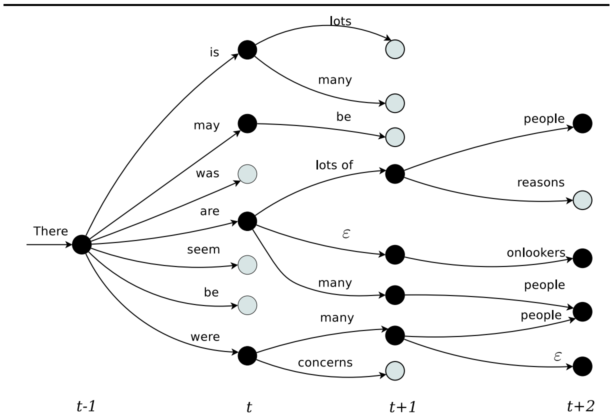

Figure 2: Illustration of decoder graph and batch-querying of the language model.

from other features) decides which hypotheses to keep in the search graph. When using a distributed language model, the decoder first tentatively extends all current hypotheses, taking note of which n-grams are required to score them. These are queued up for transmission as a batch request. When the scores are returned, the decoder re-visits all of these tentative hypotheses, assigns scores, and re-prunes the search graph. It is then ready for the next round of extensions, again involving queuing the n-grams, waiting for the servers, and pruning.

The process is illustrated in Figure 2 assuming a trigram model and a decoder policy of pruning to the four most promising hypotheses. The four active hypotheses (indicated by black disks) at time t are: *There is, There may, There are,* and *There were.* The decoder extends these to form eight new nodes at time $t+1$. Note that one of the arcs is labeled ϵ, indicating that no target-language word was generated when the source-language word was consumed. The n-grams necessary to score these eight hypotheses are *There is lots, There is many, There may be, There are lots, are lots of,* etc. These are queued up and their language-model scores requested in a batch manner. After scoring, the decoder prunes this set as indicated by the four black disks at time $t+1$, then extends these to form five new nodes (one is shared) at time $t+2$. The n-grams necessary to score these hypotheses are *lots of people, lots of reasons, There are onlookers,* etc. Again, these are sent to the server together, and again after scoring the graph is pruned to four active (most promising) hypotheses.

The alternating processes of queuing, waiting and scoring/pruning are done once per word position in a source sentence. The average sentence length in our test data is 22 words (see section 7.1), thus we have 23 rounds[3] per sentence on average. The number of n-grams requested per sentence depends on the decoder settings for beam size, re-ordering window, etc. As an example for larger runs reported in the experiments section, we typically request around 150,000 n-grams per sentence. The average network latency per batch is 35 milliseconds, yielding a total latency of 0.8 seconds caused by the distributed language model for an average sentence of 22 words. If a slight reduction in translation quality is allowed, then the average network latency per batch can be brought down to 7 milliseconds by reducing the number of n-grams requested per sentence to around 10,000. As a result, our system can efficiently use the large distributed language model at decoding time. There is no need for a second pass nor for n-best list rescoring.

We focused on machine translation when describing the queued language model access. However, it is general enough that it may also be applicable to speech decoders and optical character recognition systems.

7 Experiments

We trained 5-gram language models on amounts of text varying from 13 million to 2 trillion tokens. The data is divided into four sets; language models are trained for each set separately[4]. For each training data size, we report the size of the resulting language model, the fraction of 5-grams from the test data that is present in the language model, and the BLEU score (Papineni et al., 2002) obtained by the machine translation system. For smaller training sizes, we have also computed test-set perplexity using Kneser-Ney Smoothing, and report it for comparison.

7.1 Data Sets

We compiled four language model training data sets, listed in order of increasing size:

[3] One additional round for the sentence end marker.

[4] Experience has shown that using multiple, separately trained language models as feature functions in Eq (1) yields better results than using a single model trained on all data.

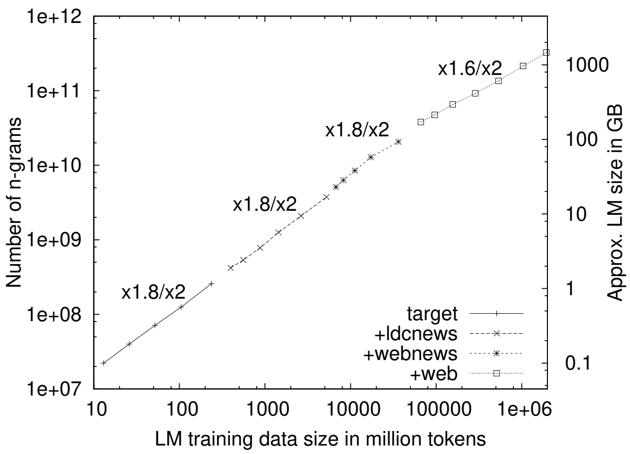

Figure 3: Number of n-grams (sum of unigrams to 5-grams) for varying amounts of training data.

target: The English side of Arabic-English parallel data provided by LDC[5] (237 million tokens).
ldcnews: This is a concatenation of several English news data sets provided by LDC[6] (5 billion tokens).
webnews: Data collected over several years, up to December 2005, from web pages containing predominantly English news articles (31 billion tokens).
web: General web data, which was collected in January 2006 (2 trillion tokens).

For testing we use the "NIST" part of the 2006 Arabic-English NIST MT evaluation set, which is not included in the training data listed above[7]. It consists of 1797 sentences of newswire, broadcast news and newsgroup texts with 4 reference translations each. The test set is used to calculate translation BLEU scores. The English side of the set is also used to calculate perplexities and n-gram coverage.

7.2 Size of the Language Models

We measure the size of language models in total number of n-grams, summed over all orders from 1 to 5. There is no frequency cutoff on the n-grams.

[5]`http://www.nist.gov/speech/tests/mt/doc/LDCLicense-mt06.pdf` contains a list of parallel resources provided by LDC.

[6]The bigger sets included are LDC2005T12 (Gigaword, 2.5B tokens), LDC93T3A (Tipster, 500M tokens) and LDC2002T31 (Acquaint, 400M tokens), plus many smaller sets.

[7]The test data was generated after 1-Feb-2006; all training data was generated before that date.

	target	***webnews***	***web***
# tokens	237M	31G	1.8T
vocab size	200k	5M	16M
# n-grams	257M	21G	300G
LM size (SB)	2G	89G	1.8T
time (SB)	20 min	8 hours	1 day
time (KN)	2.5 hours	2 days	–
# machines	100	400	1500

Table 2: Sizes and approximate training times for 3 language models with Stupid Backoff (SB) and Kneser-Ney Smoothing (KN).

There is, however, a frequency cutoff on the vocabulary. The minimum frequency for a term to be included in the vocabulary is 2 for the *target*, *ldcnews* and *webnews* data sets, and 200 for the *web* data set. All terms below the threshold are mapped to a special term `UNK`, representing the unknown word.

Figure 3 shows the number of n-grams for language models trained on 13 million to 2 trillion tokens. Both axes are on a logarithmic scale. The right scale shows the approximate size of the served language models in gigabytes. The numbers above the lines indicate the relative increase in language model size: x1.8/x2 means that the number of n-grams grows by a factor of 1.8 each time we double the amount of training data. The values are similar across all data sets and data sizes, ranging from 1.6 to 1.8. The plots are very close to straight lines in the log/log space; linear least-squares regression finds $r^2 > 0.99$ for all four data sets.

The *web* data set has the smallest relative increase. This can be at least partially explained by the higher vocabulary cutoff. The largest language model generated contains approx. 300 billion n-grams.

Table 2 shows sizes and approximate training times when training on the full *target*, *webnews*, and *web* data sets. The processes run on standard current hardware with the Linux operating system. Generating models with Kneser-Ney Smoothing takes 6 – 7 times longer than generating models with Stupid Backoff. We deemed generation of Kneser-Ney models on the *web* data as too expensive and therefore excluded it from our experiments. The estimated runtime for that is approximately one week on 1500 machines.

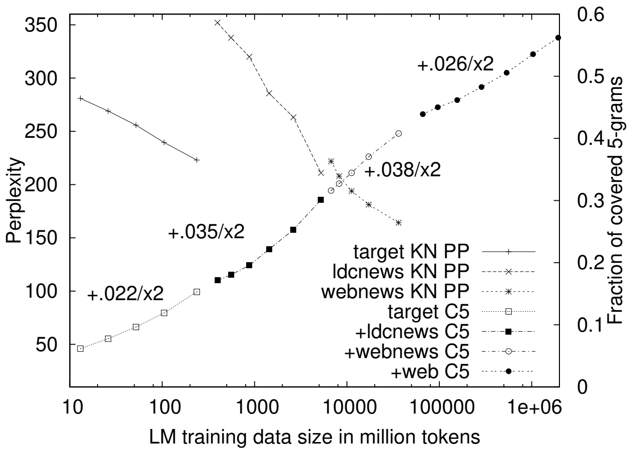

Figure 4: Perplexities with Kneser-Ney Smoothing (KN PP) and fraction of covered 5-grams (C5).

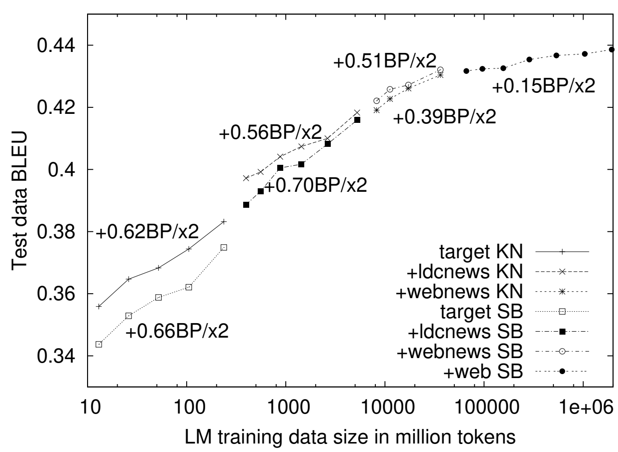

Figure 5: BLEU scores for varying amounts of data using Kneser-Ney (KN) and Stupid Backoff (SB).

7.3 Perplexity and n-Gram Coverage

A standard measure for language model quality is perplexity. It is measured on test data $T = w_1^{|T|}$:

$$PP(T) = e^{-\frac{1}{|T|}\sum_{i=1}^{|T|} \log p(w_i|w_{i-n+1}^{i-1})} \qquad (7)$$

This is the inverse of the average conditional probability of a next word; lower perplexities are better. Figure 4 shows perplexities for models with Kneser-Ney smoothing. Values range from 280.96 for 13 million to 222.98 for 237 million tokens *target* data and drop nearly linearly with data size ($r^2 = 0.998$). Perplexities for *ldcnews* range from 351.97 to 210.93 and are also close to linear ($r^2 = 0.987$), while those for *webnews* data range from 221.85 to 164.15 and flatten out near the end. Perplexities are generally high and may be explained by the mixture of genres in the test data (newswire, broadcast news, newsgroups) while our training data is predominantly written news articles. Other held-out sets consisting predominantly of newswire texts receive lower perplexities by the same language models, e.g., using the full *ldcnews* model we find perplexities of 143.91 for the NIST MT 2005 evaluation set, and 149.95 for the NIST MT 2004 set.

Note that the perplexities of the different language models are not directly comparable because they use different vocabularies. We used a fixed frequency cutoff, which leads to larger vocabularies as the training data grows. Perplexities tend to be higher with larger vocabularies.

Perplexities cannot be calculated for language models with Stupid Backoff because their scores are not normalized probabilities. In order to nevertheless get an indication of potential quality improvements with increased training sizes we looked at the 5-gram coverage instead. This is the fraction of 5-grams in the test data set that can be found in the language model training data. A higher coverage will result in a better language model if (as we hypothesize) estimates for seen events tend to be better than estimates for unseen events. This fraction grows from 0.06 for 13 million tokens to 0.56 for 2 trillion tokens, meaning 56% of all 5-grams in the test data are known to the language model.

Increase in coverage depends on the training data set. Within each set, we observe an almost constant growth (correlation $r^2 \geq 0.989$ for all sets) with each doubling of the training data as indicated by numbers next to the lines. The fastest growth occurs for *webnews* data (+0.038 for each doubling), the slowest growth for *target* data (+0.022/x2).

7.4 Machine Translation Results

We use a state-of-the-art machine translation system for translating from Arabic to English that achieved a competitive BLEU score of 0.4535 on the Arabic-English NIST subset in the 2006 NIST machine translation evaluation[8]. Beam size and re-ordering window were reduced in order to facilitate a large

[8] See `http://www.nist.gov/speech/tests/mt/mt06eval_official_results.html` for more results.

number of experiments. Additionally, our NIST evaluation system used a mixture of 5, 6, and 7-gram models with optimized stupid backoff factors for each order, while the learning curve presented here uses a fixed order of 5 and a single fixed backoff factor. Together, these modifications reduce the BLEU score by 1.49 BLEU points (BP)[9] at the largest training size. We then varied the amount of language model training data from 13 million to 2 trillion tokens. All other parts of the system are kept the same.

Results are shown in Figure 5. The first part of the curve uses *target* data for training the language model. With Kneser-Ney smoothing (KN), the BLEU score improves from 0.3559 for 13 million tokens to 0.3832 for 237 million tokens. At such data sizes, Stupid Backoff (SB) with a constant backoff parameter $\alpha = 0.4$ is around 1 BP worse than KN. On average, one gains 0.62 BP for each doubling of the training data with KN, and 0.66 BP per doubling with SB. Differences of more than 0.51 BP are statistically significant at the 0.05 level using bootstrap resampling (Noreen, 1989; Koehn, 2004).

We then add a second language model using *ldcnews* data. The first point for *ldcnews* shows a large improvement of around 1.4 BP over the last point for *target* for both KN and SB, which is approximately twice the improvement expected from doubling the amount of data. This seems to be caused by adding a new domain and combining two models. After that, we find an improvement of 0.56–0.70 BP for each doubling of the *ldcnews* data. The gap between Kneser-Ney Smoothing and Stupid Backoff narrows, starting with a difference of 0.85 BP and ending with a not significant difference of 0.24 BP.

Adding a third language models based on *webnews* data does not show a jump at the start of the curve. We see, however, steady increases of 0.39–0.51 BP per doubling. The gap between Kneser-Ney and Stupid Backoff is gone, all results with Stupid Backoff are actually *better* than Kneser-Ney, but the differences are not significant.

We then add a fourth language model based on *web* data and Stupid Backoff. Generating Kneser-Ney models for these data sizes is extremely expensive and is therefore omitted. The fourth model shows a small but steady increase of 0.15 BP per doubling, surpassing the best Kneser-Ney model (trained on less data) by 0.82 BP at the largest size. Goodman (2001) observed that Kneser-Ney Smoothing dominates other schemes over a broad range of conditions. Our experiments confirm this advantage at smaller language model sizes, but show the advantage disappears at larger data sizes.

The amount of benefit from doubling the training size is partly determined by the domains of the data sets[10]. The improvements are almost linear on the log scale within the sets. Linear least-squares regression shows correlations $r^2 > 0.96$ for all sets and both smoothing methods, thus we expect to see similar improvements when further increasing the sizes.

8 Conclusion

A distributed infrastructure has been described to train and apply large-scale language models to machine translation. Experimental results were presented showing the effect of increasing the amount of training data to up to 2 trillion tokens, resulting in a 5-gram language model size of up to 300 billion n-grams. This represents a gain of about two orders of magnitude in the amount of training data that can be handled over that reported previously in the literature (or three-to-four orders of magnitude, if one considers only single-pass decoding). The infrastructure is capable of scaling to larger amounts of training data and higher n-gram orders.

The technique is made efficient by judicious batching of score requests by the decoder in a server-client architecture. A new, simple smoothing technique well-suited to distributed computation was proposed, and shown to perform as well as more sophisticated methods as the size of the language model increases.

Significantly, we found that translation quality as indicated by BLEU score continues to improve with increasing language model size, at even the largest sizes considered. This finding underscores the value of being able to train and apply very large language models, and suggests that further performance gains may be had by pursuing this direction further.

[9] 1 BP = 0.01 BLEU. We show system scores as BLEU, differences as BP.

[10] There is also an effect of the order in which we add the models. As an example, *web* data yields +0.43 BP/x2 when added as the second model. A discussion of this effect is omitted due to space limitations.

References

Peter F. Brown, Stephen Della Pietra, Vincent J. Della Pietra, and Robert L. Mercer. 1993. The mathematics of statistical machine translation: Parameter estimation. *Computational Linguistics*, 19(2):263–311.

Stanley F. Chen and Joshua Goodman. 1998. An empirical study of smoothing techniques for language modeling. Technical Report TR-10-98, Harvard, Cambridge, MA, USA.

Jeffrey Dean and Sanjay Ghemawat. 2004. Mapreduce: Simplified data processing on large clusters. In *Sixth Symposium on Operating System Design and Implementation (OSDI-04)*, San Francisco, CA, USA.

Ahmad Emami, Kishore Papineni, and Jeffrey Sorensen. 2007. Large-scale distributed language modeling. In *Proceedings of ICASSP-2007*, Honolulu, HI, USA.

Joshua Goodman. 2001. A bit of progress in language modeling. Technical Report MSR-TR-2001-72, Microsoft Research, Redmond, WA, USA.

Frederick Jelinek and Robert L. Mercer. 1980. Interpolated estimation of Markov source parameters from sparse data. In *Pattern Recognition in Practice*, pages 381–397. North Holland.

Slava Katz. 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).

Reinhard Kneser and Hermann Ney. 1995. Improved backing-off for m-gram language modeling. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 181–184.

Philipp Koehn. 2004. Statistical significance tests for machine translation evaluation. In *Proceedings of EMNLP-04*, Barcelona, Spain.

Hermann Ney and Stefan Ortmanns. 1999. Dynamic programming search for continuous speech recognition. *IEEE Signal Processing Magazine*, 16(5):64–83.

Eric W. Noreen. 1989. *Computer-Intensive Methods for Testing Hypotheses*. John Wiley & Sons.

Franz Josef Och and Hermann Ney. 2004. The alignment template approach to statistical machine translation. *Computational Linguistics*, 30(4):417–449.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of ACL-02*, pages 311–318, Philadelphia, PA, USA.

Ying Zhang, Almut Silja Hildebrand, and Stephan Vogel. 2006. Distributed language modeling for n-best list re-ranking. In *Proceedings of EMNLP-2006*, pages 216–223, Sydney, Australia.