Normalized Log-Linear Interpolation of Backoff Language Models is Efficient

Citation for published version:

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.
Normalized Log-Linear Interpolation of Backoff Language Models is Efficient

Kenneth Heafield
University of Edinburgh
10 Crichton Street
Edinburgh EH8 9AB
United Kingdom
kheafiel@inf.ed.ac.uk

Chase Geigle
University of Illinois at Urbana-Champaign
707 S. Mathews Ave.
Urbana, IL 61801
United States
{geigle1,massung1,lanes}@illinois.edu

Sean Massung
Lane Schwartz

Abstract

We prove that log-linearly interpolated backoff language models can be efficiently and exactly collapsed into a single normalized backoff model, contradicting Hsu (2007). While prior work reported that log-linear interpolation yields lower perplexity than linear interpolation, normalizing at query time was impractical. We normalize the model offline in advance, which is efficient due to a recurrence relationship between the normalizing factors. To tune interpolation weights, we apply Newton’s method to this convex problem and show that the derivatives can be computed efficiently in a batch process. These findings are combined in new open-source interpolation tool, which is distributed with KenLM: https://kheafield.com/code/kenlm/.

Linear interpolation (Jelinek and Mercer, 1980), combines several language models $p_i$ into a single model $p_L$:

$$p_L(w_n \mid w_{n-1}^1) = \sum_i \lambda_i p_i(w_n \mid w_{n-1}^1)$$

where $\lambda_i$ are weights and $w_i^1$ are words. Because each component model $p_i$ is a probability distribution and the non-negative weights $\lambda_i$ sum to 1, the interpolated model $p_L$ is also a probability distribution. This presumes that the models have the same vocabulary, an issue we discuss in §3.1.

A log-linearly interpolated model $p_{LL}$ uses the weights $\lambda_i$, as powers (Klakow, 1998).

$$p_{LL}(w_n \mid w_{n-1}^1) \propto \prod_i p_i(w_n \mid w_{n-1}^1)^{\lambda_i}$$

The weights $\lambda_i$ are unconstrained real numbers, allowing parameters to soften or sharpen distributions. Negative weights can be used to divide a mixed-domain model by an out-of-domain model. To form a probability distribution, the product is normalized

$$p_{LL}(w_n \mid w_{n-1}^1) = \prod_i p_i(w_n \mid w_{n-1}^1)^{\lambda_i} Z(w_{n-1}^1)$$

where normalizing factor $Z$ is given by

$$Z(w_{n-1}^1) = \sum_x \prod_i p_i(x \mid w_{n-1}^1)^{\lambda_i}$$

The sum is taken over all words $x$ in the combined vocabulary of the underlying models, which can number in the millions or even billions. Computing $Z$ efficiently is a key contribution in this work.

1 Introduction

Log-linearly interpolated backoff language models yielded better perplexity than linearly interpolated models (Klakow, 1998; Gutkin, 2000), but experiments and adoption were limited due the impractically high cost of querying. This cost is due to normalizing to form a probability distribution by brute-force summing over the entire vocabulary for each query. Instead, we prove that the log-linearly interpolated model can be normalized offline in advance and exactly expressed as an ordinary backoff language model. This contradicts Hsu (2007), who claimed that log-linearly interpolated models “cannot be efficiently represented as a backoff $n$-gram model.”

We show that offline normalization is efficient due to a recurrence relationship between the normalizing factors (Whittaker and Klakow, 2002). This forms the basis for our open-source implementation, which is part of KenLM: https://kheafield.com/code/kenlm/.
Our proofs assume the component models $p_i$ are backoff language models (Katz, 1987) that memorize probability for seen $n$-grams and charge a backoff penalty $b_i$ for unseen $n$-grams.

$$p_i(w_n \mid w_{n-1}^{n-1}) = \begin{cases} p_i(w_n \mid w_{n-1}^{n-1}) & \text{if } w_n \text{ is seen} \\ p_i(w_n \mid w_{n-1}^{n-1})b_i(w_{n-1}^{n-1}) & \text{o.w.} \end{cases}$$

While linearly or log-linearly interpolated models can be queried online by querying the component models (Stolcke, 2002; Federico et al., 2008), doing so costs RAM to store duplicated $n$-grams and CPU time to perform lookups. Log-linear interpolation is particularly slow due to normalizing over the entire vocabulary. Instead, it is preferable to combine the models offline into a single backoff model containing the union of $n$-grams. Doing so is impossible for linear interpolation (§3.2); SRILM (Stolcke, 2002) and MITLM (Hsu and Glass, 2008) implement an approximation. In contrast, we prove that offline log-linear interpolation requires no such approximation.

## 2 Related Work

Instead of building separate models then weighting, Zhang and Chiang (2014) show how to train Kneser-Ney models (Kneser and Ney, 1995) on weighted data. Their work relied on prescriptive weights from domain adaptation techniques rather than tuning weights, as we do here.

Our exact normalization approach relies on the backoff structure of component models. Several approximations support general models: ignoring normalization (Chen et al., 1998), noise-contrastive estimation (Vaswani et al., 2013), and self-normalization (Andreas and Klein, 2015). In future work, we plan to exploit the structure of other features in high-quality unnormalized log-linear language models (Sethy et al., 2014).

Ignoring normalization is particularly common in speech recognition and machine translation. This is one of our baselines. Unnormalized models can also be compiled into a single model by multiplying the weighted probabilities and backoffs.\(^1\) Many use unnormalized models because weights can be jointly tuned along with other feature weights. However, Haddow (2013) showed that linear interpolation weights can be jointly tuned by pairwise ranked optimization (Hopkins and May, 2011). In theory, normalized log-linear interpolation weights can be jointly tuned in the same way.

Dynamic interpolation weights (Weintraub et al., 1996) give more weight to models familiar with a given query. Typically the weights are a function of the contexts that appear in the combined language model, which is compatible with our approach. However, normalizing factors would need to be calculated in each context.

## 3 Linear Interpolation

To motivate log-linear interpolation, we examine two issues with linear interpolation: normalization when component models have different vocabularies and offline interpolation.

### 3.1 Vocabulary Differences

Language models are normalized with respect to their vocabulary, including the unknown word.

$$\sum_{x \in \text{vocab}(p_1)} p_1(x) = 1$$

If two models have different vocabularies, then the combined vocabulary is larger and the sum is taken over more words. Component models assign their unknown word probability to these new words, leading to an interpolated model that sums to more than one. An example is shown in Table 1.

<table>
<thead>
<tr>
<th></th>
<th>$p_1$</th>
<th>$p_2$</th>
<th>$p_L$</th>
<th>Zero</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;unk&gt;</td>
<td>0.4</td>
<td>0.2</td>
<td>0.3</td>
<td>0.3</td>
</tr>
<tr>
<td>A</td>
<td>0.6</td>
<td></td>
<td>0.4</td>
<td>0.3</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>0.8</td>
<td>0.6</td>
<td>0.4</td>
</tr>
<tr>
<td>Sum</td>
<td>1</td>
<td>1</td>
<td>1.3</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 1: Linearly interpolating two models $p_1$ and $p_2$ with equal weight yields an unnormalized model $p_L$. If gaps are filled with zeros instead, the model is normalized.

To work around this problem, SRILM (Stolcke, 2002) uses zero probability instead of the unknown word probability for new words, producing a model that sums to one, but differs from what users might expect.

IRSTLM (Federico et al., 2008) asks the user to specify a common large vocabulary size. The unknown word probability is downweighted so that all models sum to one over the large vocabulary.

A component model can also be renormalized with respect to a larger vocabulary. For unigrams,
the extra mass is the number of new words times the unknown word probability. For longer contexts, if we assume the typical case where the unknown word appears only as a unigram, then queries for new words will back off to unigrams. The total mass in context \( w_{1}^{n-1} \) is

\[
1 + |\text{new}| p(\text{<unk>}) \prod_{i=1}^{n-1} b(w_{i}^{n-1})
\]

where \( \text{new} \) is the set of new words. This is efficient to compute online or offline. While there are tools to renormalize models, we are not aware of a tool that does this for linear interpolation.

Log-linear interpolation is normalized by construction. Nonetheless, in our experiments we extend IRSTLM’s approach by training models with a common vocabulary size, rather than retrofitting it at query time.

### 3.2 Offline Linear Interpolation

Given an interpolated model, offline interpolation seeks a combined model meeting three criteria: (i) encoding the same probability distribution, (ii) being a backoff model, and (iii) containing the union of \( n \)-grams from component models.

**Theorem 1.** The three offline criteria cannot be satisfied for general linearly interpolated backoff models.

**Proof.** By counterexample. Consider the models given in Table 2 interpolated with equal weight.

<table>
<thead>
<tr>
<th></th>
<th>( p_1 )</th>
<th>( p_2 )</th>
<th>( p_L )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p(A) )</td>
<td>0.4</td>
<td>0.2</td>
<td>0.3</td>
</tr>
<tr>
<td>( p(B) )</td>
<td>0.3</td>
<td>0.3</td>
<td>0.3</td>
</tr>
<tr>
<td>( p(C) )</td>
<td>0.3</td>
<td>0.5</td>
<td>0.4</td>
</tr>
<tr>
<td>( p(C \mid A) )</td>
<td>0.4</td>
<td>0.8</td>
<td>0.6</td>
</tr>
<tr>
<td>( b(A) )</td>
<td>\frac{6}{7} \approx 0.857</td>
<td>0.4</td>
<td>\frac{2}{3} \approx 0.667</td>
</tr>
</tbody>
</table>

Table 2: Counterexample models.

The probabilities shown for \( p_L \) result from encoding the same distribution. Taking the union of \( n \)-grams implies that \( p_L \) only has entries for \( A, B, C \), and \( A \), \( C \). Since the models have the same vocabulary, they are all normalized to one.

\[
p(A \mid A) + p(B \mid A) + p(C \mid A) = 1
\]

Since all models have backoff structure,

\[
p(A)b(A) + p(B)b(A) + p(C \mid A) = 1
\]

which when solved for backoff \( b(A) \) gives the values shown in Table 2. We then query \( p_L(B \mid A) \) online and offline. Online interpolation yields

\[
p_L(B \mid A) = \frac{1}{2} p_1(B \mid A) + \frac{1}{2} p_2(B \mid A)
\]

\[
= \frac{1}{2} p_1(B)b_1(A) + \frac{1}{2} p_2(B)b_2(A) = \frac{33}{175}
\]

Offline interpolation yields

\[
p_L(B \mid A) = p_L(B)L(A) = 0.2 \neq \frac{33}{175} \approx 0.189
\]

The same problem happens with real language models. To understand why, we attempt to solve for the backoff \( b_L(w_{1}^{n-1}) \). Supposing \( w_{1}^{n} \) is not in either model, we query \( p_L(w_n \mid w_{1}^{n-1}) \) offline

\[
p_L(w_n | w_{1}^{n-1}) = p_L(w_n | w_{2}^{n-1}) + \lambda p_2(w_n | w_{2}^{n-1})b_L(w_{1}^{n-1})
\]

while online interpolation yields

\[
p_L(w_n | w_{1}^{n-1}) = \lambda_1 p_1(w_n | w_{1}^{n-1}) + \lambda_2 p_2(w_n | w_{2}^{n-1})
\]

\[
= \lambda_1 p_1(w_n | w_{1}^{n-1}) + \lambda_2 p_2(w_n | w_{2}^{n-1})b_1(w_{1}^{n-1}) + \lambda_1 p_2(w_n | w_{2}^{n-1})b_2(w_{1}^{n-1})
\]

Solving for \( b_L(w_{1}^{n-1}) \) we obtain

\[
\frac{\lambda_1 p_1(w_n | w_{1}^{n-1})b_1(w_{1}^{n-1}) + \lambda_2 p_2(w_n | w_{2}^{n-1})b_2(w_{1}^{n-1})}{\lambda_1 p_1(w_n | w_{1}^{n-1}) + \lambda_2 p_2(w_n | w_{2}^{n-1})}
\]

which is a weighted average of the backoff weights \( b_1(w_{1}^{n-1}) \) and \( b_2(w_{1}^{n-1}) \). The weights depend on \( w_n \), so \( b_L \) is no longer a function of \( w_{1}^{n-1} \).

In the SRILM approximation (Stolcke, 2002), probabilities for \( n \)-grams that exist in the model are computed exactly. The backoff weights are chosen to produce a model that sums to one. However, newer versions of SRILM (Stolcke et al., 2011) interpolate by ingesting one component model at a time. For example, the first two models are approximately interpolated before adding a third model. An \( n \)-gram appearing only in the third model will have an approximate probability. Therefore, the output depends on the order in which users specify models. Moreover, weights were optimized for correct linear interpolation, not the approximation.

Stolcke (2002) find that the approximation actually decreases perplexity, which we also see in the experiments (§6). However, approximation only happens when the model backs off, which is less likely to happen in fluent sentences used for perplexity scoring.
4 Offline Log-Linear Interpolation

Log-linearly interpolated backoff models \( p_i \) can be collapsed into a single offline model \( p_{\text{LL}} \). The combined model takes the union of \( n \)-grams in component models.\(^2\) For those \( n \)-grams, it memorizes correct probability \( p_{\text{LL}} \).

\[
p_{\text{LL}}(w_n | w_1^{n-1}) = \frac{\prod_i p_i(w_n | w_1^{n-1})^{\lambda_i}}{Z(w_1^{n-1})} \tag{1}
\]

When \( w_1^n \) does not appear, the backoff \( b_{\text{LL}}(w_1^{n-1}) \) modifies \( p_{\text{LL}}(w_n | w_2^{n-2}) \) to make an appropriately normalized probability. To do so, it cancels out the shorter query’s normalization term \( Z(w_2^{n-2}) \) then applies the correct term \( Z(w_1^{n-1}) \). It also applies the component backoff terms.

\[
b_{\text{LL}}(w_1^{n-1}) = \frac{Z(w_2^{n-2})}{Z(w_1^{n-1})} \prod_i b_i(w_1^{n-1})^{\lambda_i} \tag{2}
\]

Almost by construction, the model satisfies two of our criteria (§3.2): being a backoff model and containing the union of \( n \)-grams. However, backoff models require that the backoff weight of an unseen \( n \)-gram be implicitly 1.

**Lemma 1.** If \( w_1^{n-1} \) is unseen in the combined model, then the backoff weight \( b_{\text{LL}}(w_1^{n-1}) = 1 \).

**Proof.** Because we have taken the union of entries, \( w_1^{n-1} \) is unseen in component models. These components are backoff models, so implicitly \( b_i(w_1^{n-1}) = 1 \) \( \forall i \). Focusing on the normalization term \( Z(w_1^{n-1}) \),

\[
Z(w_1^{n-1}) = \sum_x \prod_i p_i(x | w_1^{n-1})^{\lambda_i} = \sum_x \prod_i p_i(x | w_2^{n-2})^{\lambda_i} b_i(w_1^{n-1})^{\lambda_i} = \sum_x \prod_i p_i(x | w_2^{n-2})^{\lambda_i} = Z(w_2^{n-2})
\]

All of the models back off because \( w_1^{n-1} \) is unseen, being a superstring of \( w_1^{n-1} \). Relevant backoff weights \( b_i(w_1^{n-1}) = 1 \) as noted earlier. Recalling the definition of \( b_{\text{LL}}(w_1^{n-1}) \),

\[
\frac{Z(w_2^{n-2})}{Z(w_1^{n-1})} \prod_i b_i(w_1^{n-1})^{\lambda_i} = \frac{Z(w_2^{n-2})}{Z(w_1^{n-1})} = 1 \quad \square
\]

\(^2\)We further assume that every substring of a seen \( n \)-gram is also seen. This follows from estimating on text, except in the case of adjusted count pruning by SRILM. In such cases, we add the missing entries to component models, with no additional memory cost in trie data structures.

We now have a backoff model containing the union of \( n \)-grams. It remains to show that the offline model produces correct probabilities.

**Theorem 2.** The proposed offline model agrees with online log-linear interpolation.

**Proof.** By induction on the number of words backed off in offline interpolation. To disambiguate, we will use \( p_{\text{on}} \) to refer to online interpolation and \( p_{\text{off}} \) to refer to offline interpolation.

Base case: the queried \( n \)-gram is in the offline model and we have memorized the online probability by construction.

Inductive case: Let \( p_{\text{off}}(w_n | w_1^{n-1}) \) be a query that backs off. In online interpolation,

\[
p_{\text{on}}(w_n | w_1^{n-1}) = \prod_i p_i(w_n | w_1^{n-1})^{\lambda_i} / Z(w_1^{n-1})
\]

Because \( w_1^n \) is unseen in the offline model and we took the union, it is unseen in every model \( p_i \).

\[
\frac{Z(w_2^{n-2})}{Z(w_1^{n-1})} p_{\text{on}}(w_n | w_2^{n-2}) = \prod_i p_i(w_n | w_2^{n-2})^{\lambda_i} b_i(w_1^{n-1})^{\lambda_i} / Z(w_1^{n-1})
\]

Recognizing the unnormalized probability \( Z(w_2^{n-2})p_{\text{on}}(w_n | w_2^{n-2}) \),

\[
Z(w_2^{n-2})p_{\text{on}}(w_n | w_2^{n-2}) \prod_i b_i(w_1^{n-1})^{\lambda_i} = \frac{Z(w_2^{n-2})}{Z(w_1^{n-1})} \prod_i b_i(w_1^{n-1})^{\lambda_i}
\]

The last equality follows from the definition of \( b_{\text{off}} \) and Lemma 1, which extended the domain of \( b_{\text{off}} \) to any \( w_1^{n-1} \). By the inductive hypothesis, \( p_{\text{on}}(w_n | w_2^{n-2}) = p_{\text{off}}(w_n | w_2^{n-2}) \) because it backs off one less time.

\[
p_{\text{off}}(w_n | w_2^{n-2})b_{\text{on}}(w_1^{n-1}) = p_{\text{off}}(w_n | w_1^{n-1})
\]

The offline model \( p_{\text{off}}(w_n | w_1^{n-1}) \) backs off because that is the case we are considering. Combining our chain of equalities,

\[
p_{\text{on}}(w_n | w_1^{n-1}) = p_{\text{off}}(w_n | w_1^{n-1})
\]

By induction, the claim holds for all \( w_1^n \). \( \square \)
4.1 Normalizing Efficiently

In order to build the offline model, the normalization factor $Z$ needs to be computed in every seen context. To do so, we extend the tree-structure method of Whittaker and Klakow (2002), which they used to compute and cache normalization factors on the fly. It exploits the sparsity of language models: when summing over the vocabulary, most queries will back off. Formally, we define $s(w^n_i)$ to be the set of words $x$ where $p_i(x \mid w^n_i)$ does not back off in some model.

$$s(w^n_i) = \{ x : w^n_i x \text{ is seen in any model} \}$$

To exploit this, we use the normalizing factor $Z(w^n_2)$ from a lower order and patch it up by summing over $s(w^n_1)$.

**Theorem 3.** The normalization factors $Z$ obey a recurrence relationship:

$$Z(w^n_1) = \sum_{x \in s(w^n_1)} \prod_i p_i(x \mid w^n_1)^{\lambda_i} + \left( Z(w^n_2) - \sum_{x \in s(w^n_1)} \prod_i p_i(x \mid w^n_2)^{\lambda_i} \right) \prod_i b_i(w^n_1)^{\lambda_i}$$

**Proof.** The first term handles seen $n$-grams while the second term handles unseen $n$-grams. The definition of $Z$

$$Z(w^n_1) = \sum_{x \in \text{vocab}} \prod_i p_i(x \mid w^n_1)^{\lambda_i}$$

can be partitioned by cases.

$$\sum_{x \in s(w^n_1)} \prod_i p_i(x \mid w^n_1)^{\lambda_i} + \sum_{x \not\in s(w^n_1)} \prod_i p_i(x \mid w^n_1)^{\lambda_i}$$

The first term agrees with the claim, so we focus on the case where $x \not\in s(w^n_1)$. By definition of $s$, all models back off.

$$\sum_{x \not\in s(w^n_1)} \prod_i p_i(x \mid w^n_1)^{\lambda_i} = \sum_{x \not\in s(w^n_1)} \prod_i p_i(x \mid w^n_2)^{\lambda_i} b_i(w^n_1)^{\lambda_i}$$

$$= \left( \sum_{x \not\in s(w^n_1)} \prod_i p_i(x \mid w^n_2)^{\lambda_i} \right) \prod_i b_i(w^n_1)^{\lambda_i}$$

$$= \left( Z(w^n_2) - \sum_{x \in s(w^n_1)} \prod_i p_i(x \mid w^n_2)^{\lambda_i} \right) \prod_i b_i(w^n_1)^{\lambda_i}$$

This is the second term of the claim. \(\square\)

![Diagram](image_url)

**Figure 1:** Multi-stage streaming pipeline for offline log-linear interpolation. Bold arrows indicate sorting is performed.

The recurrence structure of the normalization factors suggests a computational strategy: compute $Z(c)$ by summing over the unigrams, $Z(w_n)$ by summing over bigrams $w_n x$, $Z(w^n_{n-1})$ by summing over trigrams $w^n_{n-1} x$, and so on.

4.2 Streaming Computation

Part of the point of offline interpolation is that there may not be enough RAM to fit all the component models. Moreover, with compression techniques that rely on immutable models (Whittaker and Raj, 2001; Talbot and Osborne, 2007), a mutable version of the combined model may not fit in RAM. Instead, we construct the offline model with disk-based streaming algorithms, using the framework we designed for language model estimation (Heafield et al., 2013). Our pipeline (Figure 1) has four conceptual steps: merge probabilities, apply backoffs, normalize, and output. Applying backoffs and normalization are performed in the same pass, so there are three total passes.

4.2.1 Merge Probabilities

This step takes the union of $n$-grams and multiplies probabilities from component models. We
assume that the component models are sorted in suffix order (Figure 4), which is true of models produced by lmplz (Heafield et al., 2013) or stored in a reverse trie. Moreover, despite having different word indices, the models are consistently sorted using the string word, or a hash thereof.

Table 3: Merging probabilities processes n-grams in lexicographic order by suffix. Column headings indicate precedence.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4: Sorting orders (Heafield et al., 2013). In suffix order, the last word is primary. In context order, the penultimate word is primary. Column headings indicate precedence.

Each output tuple has the form

\[
\langle w_1^n, m(w_1^n), \prod_i p_i(w_n|w_{m_i}(w_1^n))^{\lambda_i}, \\
\prod_i p_i(w_n|w_{m_i}(w_2^n))^{\lambda_i} \rangle
\]

where \(m(w_1^n)\) is a vector of backoff requests, from which \(m(w_2^n)\) can be computed.

4.2.2 Apply Backoffs

This step fulfills the backoff requests from merging probabilities. The merged probabilities are sorted in context order (Table 4) so that n-grams \(w_1^n\), sharing the same context \(w_1^{n-1}\) are consecutive. Moreover, contexts \(w_1^{n-1}\) appear in suffix order. We use this property to stream through the component models again in their native suffix order, this time reading backoff weights \(b_i(w_1^{n-1}), b_i(w_2^{n-1}), \ldots, b_i(w_{n-1})\). Multiplying the appropriate backoff weights by \(\prod_i p_i(w_n|w_{m_i}(w_1^n))^{\lambda_i}\) yields unnormalized probability

\[
\prod_i p_i(w_n|w_1^{n-1})^{\lambda_i}
\]

The same applies to the lower order.

\[
\prod_i p_i(w_n|w_2^{n-1})^{\lambda_i}
\]

This step also merges backoffs from component models, with output still in context order.

\[
\langle w_1^n, \prod_i b_i(w_1^{n-1})^{\lambda_i}, \prod_i p_i(w_n|w_1^{n-1})^{\lambda_i}, \\
\prod_i p_i(w_n|w_2^{n-1})^{\lambda_i} \rangle
\]

The implementation is combined with normalization, so the tuple is only conceptual.
4.2.3 Normalize

This step computes normalization factor $Z$ for all contexts, which it applies to produce $p_{LL}$ and $b_{LL}$. Recalling §4.1, $Z(w_1^{n-1})$ is efficient to compute in a batch process by processing suffixes $Z(\epsilon), Z(w_n), \ldots Z(w_2^{n-1})$ first. In order to minimize memory consumption, we chose to evaluate the contexts in depth-first order by suffix, so that $Z(A)$ is computed immediately before it is needed to compute $Z(AA)$ and forgotten at $Z(B)$.

Computing $Z(w_1^{n-1})$ by applying Theorem 3 requires the sum

$$\sum_{x \in s(w_1^{n-1})} \prod_i p_i(x \mid w_1^{n-1})^\lambda_i$$

where $s(w_1^{n-1})$ restricts to seen $n$-grams. For this, we stream through the output of the apply backoffs step in context order, which makes the various values of $x$ consecutive. Theorem 3 also requires a sum over the lower-order unnormalized probabilities

$$\sum_{x \in s(w_1^{n-1})} \prod_i p_i(x \mid w_2^{n-1})^\lambda_i$$

We placed these terms in the input tuple for $w_1^{n-1}x$. Otherwise, it would be hard to access these values while streaming in context order.

While we have shown how to compute $Z(w_1^{n-1})$, we still need to normalize the probabilities. Unfortunately, $Z(w_1^{n-1})$ is only known after streaming through all records of the form $w_1^{n-1}x$, which are the very same records to normalize. We therefore buffer up to the vocabulary size for each order in memory to allow rewinding. Processing context $w_1^{n-1}$ thus yields normalized probabilities $p_{LL}(x \mid w_1^{n-1})$ for all seen $w_1^{n-1}x$.

$$\langle w_1^{n}, p_{LL}(x \mid w_1^{n-1}) \rangle$$

These records are generated in context order, the same order as the input.

The normalization step also computes backoffs.

$$b_{LL}(w_1^{n-1}) = \frac{Z(w_2^{n-1})}{Z(w_1^{n-1})} \prod_i b_i(w_1^{n-1})^{\lambda_i}$$

Normalization $Z(w_1^{n-1})$ is computed by this step, numerator $Z(w_2^{n-1})$ is available due to depth-first search, and the backoff terms $\prod_i b_i(w_1^{n-1})$ are present in the input. The backoffs $b_{LL}$ are generated in suffix order, since each context produces a backoff value. These are written to a sidechannel stream as bare values without keys.

4.2.4 Output

Language model toolkits store probability $p_{LL}(w_n \mid w_1^{n-1})$ and backoff $b_{LL}(w_1^{n})$ together as values for the key $w_1^{n}$. To reunify them, we sort $\langle w_1^{n}, p_{LL}(w_n \mid w_1^{n-1}) \rangle$ in suffix order and merge with the backoff sidechannel from normalization, which is already in suffix order. Suffix order is also preferable because toolkits can easily build a reverse trie data structure.

5 Tuning

Weights are tuned to maximize the log probability of held-out data. This is a convex optimization problem (Klakow, 1998). Iterations are expensive due to the need to normalize over the vocabulary at least once. However, the number of weights is small, which makes the Hessian matrix cheap to store and invert. We therefore selected Newton’s method.\(^3\)

The log probability of tuning data $w$ is

$$\log \prod_n p_{LL}(w_n \mid w_1^{n-1})$$

which expands according to the definition of $p_{LL}$

$$\sum_n \left( \sum_i \lambda_i \log p_i(w_n \mid w_1^{n-1}) \right) - \log Z(w_1^{n-1})$$

The gradient with respect to $\lambda_i$ has a compact form

$$\sum_n \log p_i(w_n \mid w_1^{n-1}) + CH(p_{LL}, p_i \mid w_1^{n-1})$$

where $CH$ is cross entropy. However, computing the cross entropy directly would entail a sum over the vocabulary for every word in the tuning data. Instead, we apply Theorem 3 to express $Z(w_1^{n-1})$ in terms of $Z(w_2^{n-1})$ before taking the derivative. This allows us to perform the same depth-first computation as before (§4.2.3), only this time $\frac{\partial}{\partial \lambda_i} Z(w_1^{n-1})$ is computed in terms of $\frac{\partial}{\partial \lambda_i} Z(w_2^{n-1})$.

The same argument applies when taking the Hessian with respect to $\lambda_i$ and $\lambda_j$. Rather than compute it directly in the form

$$\sum_n - \sum_x p_{LL}(x \mid w_1^{n-1}) \log p_i(x \mid w_1^{n-1}) \log p_j(x \mid w_1^{n-1}) + CH(p_{LL}, p_i \mid w_1^{n-1})CH(p_{LL}, p_j \mid w_1^{n-1})$$

we apply Theorem 3 to compute the Hessian for $w_1^{n}$ in terms of the Hessian for $w_2^{n}$.

\(^3\)We also considered minibatches, though grouping tuning data to reduce normalization cost would introduce bias.
6 Experiments

We perform experiments for perplexity, query speed, memory consumption, and effectiveness in a machine translation system.

Individual language models were trained on English corpora from the WMT 2016 news translation shared task (Bojar et al., 2016). This includes the seven newswires (afp, apw, cna, ltw, nyt, wpb, xin) from English Gigaword Fifth Edition (Parker et al., 2011); the 2007–2015 news crawls; News discussion; News commentary v11; English from Europarl v8 (Koehn, 2005); the English side of the French-English parallel corpus (Bojar et al., 2013); and the English side of SETIMES2 (Tiedemann, 2009). We additionally built one language model trained on the concatenation of all of the above corpora. All corpora were preprocessed using the standard Moses (Koehn et al., 2007) scripts to perform normalization, tokenization, and truecasing. To prevent SRILM from running out of RAM, we excluded the large monolingual CommonCrawl data, but included English from the parallel CommonCrawl data.

All language models are 5-gram backoff language models trained with modified Kneser-Ney smoothing (Chen and Goodman, 1998) using lmplz (Heafield et al., 2013). Also to prevent SRILM from running out of RAM, we pruned singleton trigrams and above.

For linear interpolation, we tuned weights using IRSTLM. To work around SRILM’s limitation of ten models, we interpolated the first ten then carried the combined model and added nine more component models, repeating this last step as necessary. Weights were normalized within batches to achieve the correct final weighting. This simply extends the way SRILM internally carries a combined model and adds one model at a time.

6.1 Perplexity experiments

We experiment with two domains: TED talks, which is out of domain, and news, which is in-domain for some corpora. For TED, we tuned on the IWSLT 2010 English dev set and test on the 2010 test set. For news, we tuned on the English side of the WMT 2015 Russian–English evaluation set and test on the WMT 2014 Russian–English evaluation set. To measure generalization, we also evaluated news on models tuned for TED and vice-versa. Results are shown in Table 5.

For News Crawl 2014, we used version 2.
<table>
<thead>
<tr>
<th>LM</th>
<th>Tuning</th>
<th>Compiling</th>
<th>Querying</th>
</tr>
</thead>
<tbody>
<tr>
<td>All concatenated</td>
<td>N/A</td>
<td>N/A</td>
<td>0.186µs 46.7G</td>
</tr>
<tr>
<td>Offline linear</td>
<td>0.876m 60.2G</td>
<td>641m 123G</td>
<td>0.186µs 46.8G</td>
</tr>
<tr>
<td>Online linear</td>
<td>0.876m 60.2G</td>
<td>N/A</td>
<td>6.57µs 89.1G</td>
</tr>
<tr>
<td>Log-linear</td>
<td>600m 63.9G</td>
<td>89.8m 63.9G</td>
<td>0.186µs 46.8G</td>
</tr>
</tbody>
</table>

**Table 6: Speed and memory consumption of LM combination methods.** Interpolated models include the concatenated model. Tuning and compiling times are in minutes, memory consumption in gigabytes, and query time in microseconds per query (on 1G of held-out Common Crawl monolingual data).

Log-linear interpolation performs better on TED (72.58 perplexity versus 75.91 for offline linear interpolation). However, it performs worse on news. In future work, we plan to investigate whether log-linear wins when all corpora are out-of-domain since it favors agreement by all models.

Table 6 compares the speed and memory performance of the competing methods. While the log-linear tuning is much slower, its compilation is faster compared to the offline linear model’s long run time. Since the model formats are the same for the concatenation and log-linear, they share the fastest query speeds. Query speed was measured using KenLM’s (Heafield, 2011) faster probing data structure.\(^5\)

### 6.2 MT experiments

We trained a statistical phrase-based machine translation system for Romanian-English on the Romanian-English parallel corpora released as part of the 2016 WMT news translation shared task. We trained three variants of this MT system. The first used a single language model trained on the concatenation of the 21 individual LM training corpora. The second used 22 language models, with each LM presented to Moses as a separate feature. The third used a single language model which is an interpolation of all 22 models. This variant was run with offline linear, online linear, and log-linear interpolation. All MT system variants were optimized using IWSLT 2011 Romanian-English TED test as the development set, and were evaluated using the IWSLT 2012 Romanian-English TED test.

As shown in Table 7, no significant difference in MT quality as measured by BLEU was observed; the best BLEU score came from separate features at 18.40 while log-linear scored 18.15. We leave jointly tuned normalized log-linear interpolation to future work.

**Table 7: Machine translation performance comparison in an end-to-end system.**

<table>
<thead>
<tr>
<th>LM</th>
<th>BLEU</th>
<th>BLEU-c</th>
</tr>
</thead>
<tbody>
<tr>
<td>22 separate LMs</td>
<td>18.40</td>
<td>17.91</td>
</tr>
<tr>
<td>All concatenated</td>
<td>18.02</td>
<td>17.55</td>
</tr>
<tr>
<td>Offline linear</td>
<td>18.00</td>
<td>17.53</td>
</tr>
<tr>
<td>Online linear</td>
<td>18.27</td>
<td>17.82</td>
</tr>
<tr>
<td>Log-linear</td>
<td>18.15</td>
<td>17.70</td>
</tr>
</tbody>
</table>

**7 Conclusion**

Normalized log-linear interpolation is now a tractable alternative to linear interpolation for backoff language models. Contrary to Hsu (2007), we proved that these models can be exactly collapsed into a single backoff language model. This solves the query speed problem. Empirically, compiling the log-linear model is faster than SRILM can collapse its approximate offline linear model. In future work, we plan to improve performance of feature weight tuning and investigate more general features.

**Acknowledgments**

Thanks to João Sedoc, Grant Erdmann, Jeremy Gwinnup, Marcin Junczys-Dowmunt, Chris Dyer, Jon Clark, and MT Marathon attendees for discussions. Partial funding was provided by the Amazon Academic Research Awards program. This material is based upon work supported by the NSF GRFP under Grant Number DGE-1144245.

**References**


Ashish Vaswani, Yinggong Zhao, Victoria Fossum, and David Chiang. 2013. Decoding with large-scale neural language models improves translation. In Proceedings of EMNLP.


