# DISTRIBUTED DISCRIMINATIVE LANGUAGE MODELS FOR GOOGLE VOICE-SEARCH

*Preethi Jyothi*[⋆†] *Leif Johnson*[⋆] *Ciprian Chelba*[†] *Brian Strope*[†]

[⋆†] The Ohio State University
[⋆] The University of Texas at Austin
[†] Google Inc.

## ABSTRACT

This paper considers large-scale linear discriminative language models trained using a distributed perceptron algorithm. The algorithm is implemented efficiently using a MapReduce/SSTable framework.

This work also introduces the use of large amounts of unsupervised data (confidence filtered Google voice-search logs) in conjunction with a novel training procedure that regenerates word lattices for the given data with a weaker acoustic model than the one used to generate the unsupervised transcriptions for the logged data. We observe small but statistically significant improvements in recognition performance after reranking N-best lists of a standard Google voice-search data set.

***Index Terms*—** Discriminative language models, Distributed Perceptron, MapReduce

## 1. INTRODUCTION

Ngram language models are ubiquitous in automatic speech recognition (ASR). They are optimized to maximize the likelihood of a training set subject to smoothing constraints. Though these models are robust, scalable and easy to build, here is an instance where they are lacking. A backoff trigram language model (LM) gives "a navigate to" (which is grammatically incorrect) a fairly large LM log probability of -0.266 because both "a" and "navigate to" are popular words in voice-search! Discriminative language models (DLMs) aim at directly optimizing error rate by rewarding features that appear in low error hypotheses and penalizing features in misrecognized hypotheses. A DLM gives "a navigate to" a negative weight of -6.5, thus decreasing the chances of this trigram appearing as an ASR output. There have been numerous approaches towards estimating DLMs for large vocabulary continuous speech recognition (LVCSR) [1, 2, 3].

There are two primary concerns with DLMs that we need to address. Firstly, DLM training requires large amounts of parallel data (in the form of correct transcripts and candidate hypotheses output by an ASR system) to be able to effectively compete with n-gram LMs trained on large amounts of text. In our experiments, such data is comprised of Google voice-search logs that are confidence-filtered from a baseline ASR system to obtain reference transcripts. However, this data is perfectly discriminated by first pass features and leaves little room for learning. Thus, we propose a novel training strategy of using lattices generated with a weaker acoustic model (henceforth referred to as *weakAM*) than the one used to generate reference transcripts for the logged data (*strongAM*). This provides us with enough word errors to derive large numbers of potentially useful word n-gram features, and it is akin to using a weak LM in discriminative acoustic modeling to give more room for diversity in the word lattices resulting in better generalization [4]. Section 4 details experiments on whether improvements on the *weakAM* data translate to data generated with the *strongAM*.

The second issue is that discriminative estimation of LMs is computationally more intensive than regular N-gram LM estimation. The advent of distributed learning algorithms [5, 6, 7] and supporting parallel computing infrastructure like MapReduce [8] has made it increasingly feasible to use large amounts of parallel data to train DLMs. We make use of these techniques by implementing a distributed training strategy for the perceptron algorithm [6] (more details in Section 2) using the MapReduce framework (implementation details are specified in Section 3).

## 2. LEARNING ALGORITHM

We aim to allow the estimation of large scale distributed models, similar in size to the ones in [9]. To this end, we make use of a distributed training strategy for the structured perceptron to train our DLMs [6].

The conventional structured perceptron [10] is an online learning algorithm where training instances are processed one at a time over multiple training epochs. Given a training utterance $\{x_i, y_i\}$ ($y_i \in \mathcal{Y}$ has the lowest error rate with respect to the reference transcription for $x_i$ , $\mathcal{Y}$ is either a word lattice or an N-best list that is generated as output by a first pass recognizer), let $y^* \in \mathcal{Y}$ be such that it maximizes the inner product of a high-dimensional feature vector $\mathbf{\Phi}(x_i, y_i) \in \mathbb{R}^d$, and the model parameters, $\mathbf{w} \in \mathbb{R}^d$. $\mathbf{\Phi}(x_i, y_i)$ includes AM and LM costs from the lattice of $x_i$; the rest of the "word features" extracted from $y_i$ are count functions for N-grams of varying order (we use up to order 3 for our experiments). If the prediction is incorrect , $\mathbf{w}$ is updated to increase the weights corresponding to features in $y_i$ and decrease the weights of features in $y^*$. Averaging parameters over the total number of

**Algorithm 1** Distributed Perceptron [6]

**Require:** Training samples $\mathcal{T} = \{x_i, y_i\}_{i=1}^{\mathcal{N}}$
1: $\mathbf{w} := [0, \ldots, 0]$ /* $\mathbf{w} \in \mathbb{R}^d$ is the output */
2: Partition $\mathcal{T}$ into $\mathcal{C}$ parts, $\mathcal{T}_1, \ldots, \mathcal{T}_{\mathcal{C}}$
3: $[\mu_1, \ldots, \mu_{\mathcal{C}}] := [\frac{1}{\mathcal{C}}, \ldots, \frac{1}{\mathcal{C}}]$ /* uniform mixing */
4: **for** $t := 1$ to $T$ **do** /* for each training epoch */
5: **for** $c := 1$ to $\mathcal{C}$ **do** /* for each partition $\mathcal{T}_i$ */
6: $\mathbf{w}_c^t := \mathbf{w}$, $\mathbf{\Delta}_c^t := [0, \ldots, 0]$
7: **for** $j := 1$ to $|\mathcal{T}_c|$ **do** /* for each sample in $\mathcal{T}_c$ */
8: $y^* := \mathrm{argmax}_y \, \mathbf{w}_c^t \cdot \mathbf{\Phi}(x_{c,j}, y)$
9: $\mathbf{\Delta}_c^t := \mathbf{\Delta}_c^t + \mathbf{\Phi}(x_{c,j}, y_{c,j}) - \mathbf{\Phi}(x_{c,j}, y^*)$
10: $\mathbf{w}_c^t := \mathbf{w} + \mathbf{\Delta}_c^t$
11: **end for**
12: **end for**
13: $\mathbf{w} := \mathbf{w} + \sum_{c=1}^{\mathcal{C}} \mu_c \mathbf{\Delta}_c^t$
14: **end for**
15: **return** $\mathbf{w}$

utterances and number of training epochs was shown to give substantial improvements in previous work [10, 1].

We make use of a distributed training strategy for the structured perceptron that was first introduced in [6]. The iterative parameter mixing strategy used in this paradigm can be explained as follows: partition the training data $\mathcal{T} = \{x_i, y_i\}_{i=1}^{\mathcal{N}}$ arbitrarily into $\mathcal{C}$ disjoint sets $\mathcal{T}_1, \ldots, \mathcal{T}_{\mathcal{C}}$. Train a structured perceptron model on each data set in parallel. After one training epoch, the parameters from each set are mixed using mixture coefficients $\mu_i$ and re-sent to each perceptron model for the next training epoch where the parameter vector is initialized with these new mixed weights. This is formally described in Algorithm 1; we call it "Distributed Perceptron". We also experiment with two other variants of distributed perceptron training, "Naive Distributed Perceptron" and "Averaged Distributed Perceptron". These models easily lend themselves to be implemented using the distributed infrastructure provided by the MapReduce [8] framework. All the variants perform online updates within a partition (also referred to as "Map chunk"). For clarity, examine two limit cases: a) using a single Map chunk for the entire training data is equivalent to the conventional structured perceptron where on-line updates happen after each utterance, and b) using a single training instance per Map chunk is equivalent to batch training. The following section describes how these three slightly different variants of distributed training for the perceptron algorithm can be represented in *Map*/*Reduce* form.

## 3. IMPLEMENTATION DETAILS

In the MapReduce programming model [8], computations are expressed as two user-defined functions: *Map* and *Reduce*. *Map* processes an input key/value pair and generates a set of intermediate key/value pairs. All intermediate values associated with the same intermediate key are aggregated and received as input by the *Reduce* function. Typically, the *Map* function can be invoked on different parts of the input data

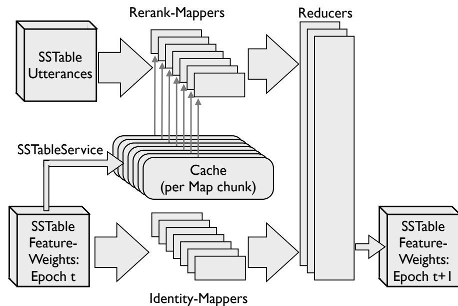


**Fig. 1**: MapReduce implementation of reranking using discriminative language models.

simultaneously. The *Reduce* function can operate independently on a part of the intermediate data where each *Reduce* task is assured to receive all the values corresponding to a given key. The reader is referred to [8] for applications and implementation details of the MapReduce interface. We use Google's SSTable data format ([11], an ordered, immutable map of key/value pairs) for storing inputs/outputs to/from our implementation. A format similar to SSTables has been open-sourced as part of the LevelDB project[1].

We have two kinds of Mappers (as illustrated in Figure 1): one that computes feature updates for the training data received by a Mapper (Map chunk) for one training epoch (*Rerank-Mapper*) and an identity Mapper that provides feature values from the previous training epoch (*Identity-Mapper*). The Reducer combines the outputs from *Rerank-Mapper* and *Identity-Mapper* to produce feature weights for the current training epoch as its output. The output feature weights are stored on disk as an SSTable, and later loaded in memory (one tablet per machine) for serving over the network as an SSTable service for the next training epoch. *Rerank-Mapper* receives training utterances as input and also requests from the SSTable service the feature values computed in the previous training epoch. *Rerank-Mapper* stores the features needed for the current Map chunk in a cache. This allows us to estimate very large distributed models: the bottleneck is no longer the total model size but instead the cache size which is controlled by the Map chunk size.

We experiment with three slightly differing variants of the structured perceptron in a distributed framework, explained using the MapReduce paradigm below. The weights $(w_{NP}^t, w_{DP}^t, w_{AV}^t)$ are specific to a single feature $f$ at training epoch $t$; $\phi(\cdot,\cdot)$ and $\Delta_c^t$ correspond to feature $f$'s value in $\mathbf{\Phi}$ and $\mathbf{\Delta}_c^t$ from Algorithm 1, respectively.

**Naive Distributed Perceptron:** For the utterances in each Map chunk $\mathcal{T}_c$, $c = 1 \ldots \mathcal{C}$, the Mappers compute $\Delta_c^t = \sum_{j=1}^{\mathcal{N}_c} \left(\phi(x_{c,j}, y_{c,j}) - \phi(x_{c,j}, {y^*}_{c,j}^t)\right)$ where $\mathcal{N}_c$ = number of utterances in Map chunk $\mathcal{T}_c$. At the Reducer, we output $w_{NP}^t = w_{NP}^{t-1} + \sum_{c=1}^{\mathcal{C}} \Delta_c^t$.

**Distributed Perceptron:** The Mappers compute $\Delta_c^t$ as above. But, at the Reducer, $\Delta_c^t$ is averaged over the total number of Map chunks (uniformly mixed using weights $\mu_c$

[1] http://code.google.com/p/leveldb/

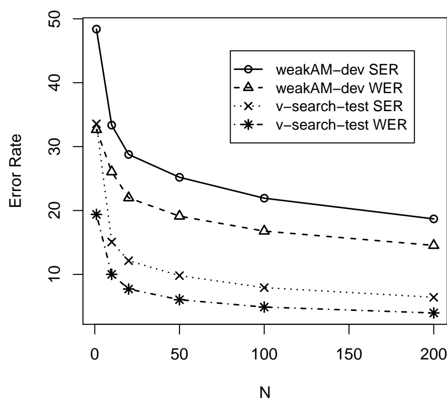


**Fig. 2**: Oracle error rates at word/sentence level for *weakAM-dev* with the weak AM and *v-search-test* with the baseline AM.

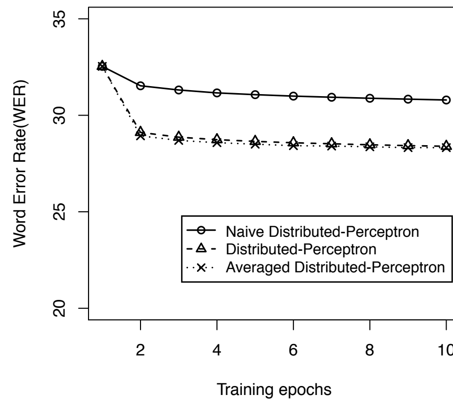


**Fig. 3**: Word error rates on *weakAM-dev* using *Perceptron*, *Distributed Perceptron* and *AveragedPerceptron* models.

as in Algorithm 1): $w^t_{DP} = w^{t-1}_{DP} + \sum_{c=1}^{\mathcal{C}} \mu_c \Delta^t_c$.

**Averaged Distributed Perceptron:** For each utterance in $\mathcal{T}_c$, $c = 1 \ldots \mathcal{C}$, let $\mathrm{sum}\Delta^t_{c,j} = \sum_{k=1}^{j} (\phi(x_{c,k}, y_{c,k}) - \phi(x_{c,k}, {y^*}^t_{c,k}))$; note that the previously defined $\Delta^t_c = \mathrm{sum}\Delta^t_{c,\mathcal{N}_c}$. The Mappers compute $\mathrm{Sum}\Delta^t_c = \sum_{j=1}^{\mathcal{N}_c} \mathrm{sum}\Delta^t_{c,j}$. Output from the Reducer is $w^t_{AV} = \frac{t-1}{t} w^{t-1}_{AV} + \frac{1}{t} w^{t-1}_{DP} + \frac{1}{\mathcal{N}t} \sum_{c=1}^{\mathcal{C}} \mathrm{Sum}\Delta^t_c$, where $\mathcal{N}$ is the total number of training utterances.

## 4. EXPERIMENTAL SETUP AND RESULTS

Our largest models are trained on 87,000 hours of speech, or ∼350 million words (*weakAM-train*) obtained by filtering voice-search logs at 0.8 confidence, and re-decoding the speech with a *weakAM* to generate N-best. To evaluate our learning setup we also use a *weakAM* development/test set (*weakAM-dev*/*weakAM-test*) consisting of 328,460/316,992 utterances, or 1,182,756/1,129,065 words, respectively. We use maximum likelihood (ML) trained single mixture Gaussians for our *weakAM*. We use a baseline LM that is sufficiently small (21 million n-grams) to allow for sub-real time lattice generation on the training data with a small memory footprint, without compromising on its strength: as shown in [12], it takes much larger LMs to get a significant relative gain in WER.

For actual ASR performance on Google voice-search we evaluate on a standard test set (*v-search-test* [13]), consisting of 27,273 utterances, or 87,360 words, and manually transcribed. Our experiments investigate whether improvements on *weakAM-dev/test* translate to *v-search-test* where N-best are generated using the *strongAM*, and scored against *manual* transcripts using fully fledged text normalization instead of the raw string edit distance used in training the DLM. All voice-search data used in the experiments is anonymized.

Figure 2 shows the oracle error rates (at the sentence/word levels) for *weakAM-dev* generated using a weak AM and *v-search-test* generated using the baseline AM. These error rates are obtained by choosing the best out of the top N hypotheses—closest in raw string edit distance to correct transcript. Note there are sufficient word errors in the *weakAM* data to train DLMs and plenty of room for improvement when we threshold N to 100.

We elaborate on our experimental results by examining each of the following aspects of interest:

**1. Improvements on *weakAM-dev* using DLMs:** Figure 3 shows the drop in WER over ten training epochs for all the variants of the perceptron algorithm described in Section 3. These results are obtained after tuning for the best lattice cost weight and Map chunk size on *weakAM-dev*. We observe that averaging (either over the number of Map chunks as in "Distributed Perceptron" or over the total number of utterances and total number of training epochs as in "Averaged Distributed Perceptron") is crucial for significant gains in recognition performance; both "Distributed Perceptron" and "Averaged Distributed Perceptron" perform comparably, with "Distributed Perceptron" doing slightly better.

Our best-performing "Distributed Perceptron" model gives a 4.7% absolute (∼15% relative) improvement over the baseline WER of 1-best hypotheses in *weakAM-dev*. This could be attributed to a combination of factors: the discriminative nature of the model or the use of large amounts of additional training data. We attempt at isolating the improvements brought upon mainly by the first factor by building an ML trained backoff trigram LM (ML-3gram). This is built using the reference transcripts of all the training utterances that were used to train the DLMs. We linearly interpolate the ML-3gram probabilities with the LM probabilites from the lattices to rerank the N-best lists in *weakAM-dev*. Table 1 shows that our best performing model (DLM-3gram) gives a significant ∼2% absolute (∼6% relative) improvement over ML-3gram. We also observe most of the improvements come from unigram/bigram features with DLM-1gram performing comparably to ML-3gram.

**Table 1**: WERs on *weakAM-dev* using the baseline 1-best system, ML-3gram and DLM-1/2/3gram.

| Data set | Baseline (%) | ML-3gram (%) | DLM-1gram (%) | DLM-2gram (%) | DLM-3gram (%) |
|---|---|---|---|---|---|
| *weakAM-dev* | 32.5 | 29.8 | 29.5 | 28.3 | 27.8 |

**2. Impact of model size on WER:** We experiment with vary-

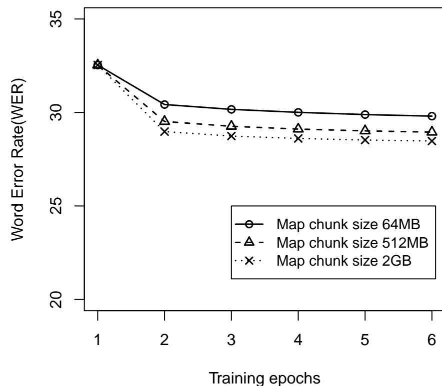


**Fig. 4**: Word error rates on *weakAM-dev* using varying Map chunk sizes of 64MB, 512MB and 2GB.

**Table 2**: WERs on *weakAM-test* using DLMs of varying sizes.

| Model | Size (in millions) | Coverage (%) | WER (%) |
|---|---|---|---|
| Baseline | 21M | - | 39.08 |
| Model1 | 65M | 74.8 | 34.18 |
| Model2 | 135M | 76.9 | 33.83 |
| Model3 | 194M | 77.8 | 33.74 |
| Model4 | 253M | 78.4 | 33.68 |

ing amounts of training data to build our DLMs and assess the impact of model size on WER. These are evaluated on an unseen data set from the *weakAM* data (*weakAM-test*). Table 2 shows each model along with its size (measured in total number of word features), coverage on *weakAM-test* (number of word features in *weakAM-test* that are in the model) and WER on *weakAM-test*. Intuitively enough, the coverage increases as the model size increases and there is a tiny drop in WER with increasing model size.

We also experiment with varying Map chunk sizes to determine its effect on WER. Figure 4 shows WERs on *weakAM-dev* using our best "Distributed Perceptron" model with different Map chunk sizes (64MB, 512MB, 2GB). We attribute the reductions in WER with increasing Map chunk size to on-line parameter updates being done on increasing amounts of training samples in each Map chunk.

**3. Impact of using a weak AM for training DLMs:** We evaluate our DLMs on *v-search-test* lattices generated using a strong AM with the hope that the improvements we saw on *weakAM-dev* translate to similar gains on *v-search-test*. Table 3 shows the WERs on both *weakAM-test* and *v-search-test* using Model 1 (from Table 2). We observe a small but statistically significant ($p < 0.05$) reduction ($\sim$2% relative) in WER on *v-search-test* over reranking with ML-3gram. This is encouraging because we attain this improvement using training lattices that were generated using a considerably weaker AM.

## 5. CONCLUSIONS

In conclusion, using DLMs that have been trained with lattices regenerated using a weak AM results in small but significant gains in recognition performance on a voice-search data

**Table 3**: WERs on *weakAM-test* and *v-search-test*.

| Data set | Baseline (%) | ML-3gram (%) | DLM-3gram (%) |
|---|---|---|---|
| *weakAM-test* | 39.1 | 36.7 | 34.2 |
| *v-search-test* | 14.9 | 14.6 | 14.3 |

set that is generated using a stronger AM. This suggests that we could hope for larger improvements by using a slightly better "weakAM". Also, we have a scalable implementation of DLMs which would be useful if we generate the contrastive set by sampling from text instead of re-decoding logs, [14]. Future work could also include using this reranking framework for longer sentences (YouTube/voicemail) and employing other learning techniques like minimizing an exponential loss function (boosting loss).

## 6. REFERENCES


[1] B. Roark, M. Saraçlar, M. Collins, and M. Johnson, "Discriminative language modeling with conditional random fields and the perceptron algorithm," in *Proc. ACL*, 2004.

[2] J. Gao, H. Yu, W. Yuan, and P. Xu, "Minimum sample risk methods for language modeling," in *Proc. of EMNLP*, 2005.

[3] Z. Zhou, J. Gao, F.K. Soong, and H. Meng, "A comparative study of discriminative methods for reranking LVCSR N-best hypotheses in domain adaptation and generalization," in *Proc. ICASSP*, 2006.

[4] R. Schlüter, B. Müller, F. Wessel, and H. Ney, "Interdependence of language models and discriminative training," in *Proc. ASRU*, 1999.

[5] G. Mann, R. McDonald, M. Mohri, N. Silberman, and D. Walker, "Efficient large-scale distributed training of conditional maximum entropy models," *Proc. NIPS*, 2009.

[6] R. McDonald, K. Hall, and G. Mann, "Distributed training strategies for the structured perceptron," in *Proc. NAACL*, 2010.

[7] K.B. Hall, S. Gilpin, and G. Mann, "MapReduce/Bigtable for distributed optimization," in *NIPS LCCC Workshop*, 2010.

[8] S. Ghemawat and J. Dean, "Mapreduce: Simplified data processing on large clusters," in *Proc. OSDI*, 2004.

[9] T. Brants, A.C. Popat, P. Xu, F.J. Och, and J. Dean, "Large language models in machine translation," in *Proc. EMNLP*, 2007.

[10] M. Collins, "Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms," in *Proc. EMNLP*, 2002.

[11] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R.E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems*, vol. 26, no. 2, pp. 1–26, 2008.

[12] C. Chelba, J. Schalkwyk, T. Brants, V. Ha, B. Harb, W. Neveitt, C. Parada, and P. Xu, "Query language modeling for voice search," in *Proc. of SLT*, 2010.

[13] B. Strope, D. Beeferman, A. Gruenstein, and X. Lei, "Unsupervised testing strategies for ASR," in *Proc. of Interspeech*, 2011.

[14] P. Jyothi and E. Fosler-Lussier, "Discriminative language modeling using simulated ASR errors," in *Proc. of Interspeech*, 2010.