Large-scale discriminative language model reranking for voice-search

Preethi Jyothi
The Ohio State University
Columbus, OH
jyothi@cse.ohio-state.edu

Leif Johnson
UT Austin
Austin, TX
leif@cs.utexas.edu

Ciprian Chelba and **Brian Strope**
Google
Mountain View, CA
{ciprianchelba,bps}@google.com

Abstract

We present a distributed framework for large-scale discriminative language models that can be integrated within a large vocabulary continuous speech recognition (LVCSR) system using lattice rescoring. We intentionally use a weakened acoustic model in a baseline LVCSR system to generate candidate hypotheses for voice-search data; this allows us to utilize large amounts of unsupervised data to train our models. We propose an efficient and scalable MapReduce framework that uses a perceptron-style distributed training strategy to handle these large amounts of data. We report small but significant improvements in recognition accuracies on a standard voice-search data set using our discriminative reranking model. We also provide an analysis of the various parameters of our models including model size, types of features, size of partitions in the MapReduce framework with the help of supporting experiments.

1 Introduction

The language model is a critical component of an automatic speech recognition (ASR) system that assigns probabilities or scores to word sequences. It is typically derived from a large corpus of text via maximum likelihood estimation in conjunction with some smoothing constraints. N-gram models have become the most dominant form of LMs in most ASR systems. Although these models are robust, scalable and easy to build, we illustrate a limitation with the following example from voice-search. We expect a low probability for an ungrammatical or implausible word sequence. However, for a trigram like “a navigate to”, a backoff trigram LM gives 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) attempt to directly optimize error rate by rewarding features that appear in low error hypotheses and penalizing features in misrecognized hypotheses. The trigram “a navigate to” receives a fairly large negative weight of -6.5 thus decreasing its chances of appearing as an ASR output. There have been numerous approaches towards estimating DLMs for large vocabulary continuous speech recognition (LVCSR) (Roark et al., 2004; Gao et al., 2005; Zhou et al., 2006).

There are two central issues that we discuss regarding DLMs. 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. This data could be simulated using 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. 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 unsupervised parallel data (referred to as the *strongAM*). This provides us with enough errors to derive large numbers of potentially useful word features; this 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 (Schlüter et al., 1999). We conduct experiments to verify whether these *weakAM*-trained models will provide performance gains on rescoring lattices from a standard test set generated using *strongAM* (discussed in Section 3.3).

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 (Mann et al., 2009; McDonald et al., 2010; Hall et al., 2010) and supporting parallel computing infrastructure like MapReduce (Ghemawat and Dean, 2004) has made it increasingly feasible to use large amounts of parallel data to train DLMs. We implement a distributed training strategy for the perceptron algorithm (introduced by McDonald et al. (2010) using the MapReduce framework. Our design choices for the MapReduce implementation are specified in Section 2.2 along with its modular nature thus enabling us to experiment with different variants of the distributed structured perceptron algorithm. Some of the descriptions in this paper have been adapted from previous work (Jyothi et al., 2012).

2 The distributed DLM framework: Training and Implementation details

2.1 Learning algorithm

We aim to allow the estimation of large scale distributed models, similar in size to the ones in Brants et al. (2007). To this end, we make use of a distributed training strategy for the structured perceptron to train our DLMs (McDonald et al., 2010). Our model consists of a high-dimensional feature vector function $\mathbf{\Phi}$ that maps an (utterance, hypothesis) pair (x, y) to a vector in $\mathbb{R}^d$, and a vector of model parameters, $\mathbf{w} \in \mathbb{R}^d$. Our goal is to find model parameters such that given x, and a set of candidate hypotheses $\mathcal{Y}$ (typically, as a word lattice or an N-best list that is obtained from a first pass recognizer), $\mathrm{argmax}_{y \in \mathcal{Y}} \mathbf{w} \cdot \mathbf{\Phi}(x, y)$ would be the $y \in \mathcal{Y}$ that minimizes the error rate between y and the correct hypothesis for x. For our experiments, the feature vector $\mathbf{\Phi}(x, y)$ consists of AM and LM costs for y from the lattice $\mathcal{Y}$ for x), as well as "word features" which count the number of times different N-grams (of order up to 5 in our experiments) occur in y.

In principle, such a model can be trained using the conventional structured perceptron algorithm (Collins, 2002). This is an online learning algorithm which continually updates $\mathbf{w}$ as it processes the training instances one at a time, over multiple training epochs. Given a training utterance $\{x_i, y_i\}$ ($y_i \in \mathcal{Y}_i$ has the lowest error rate with respect to the reference transcription for x_i, among all hypotheses in the lattice $\mathcal{Y}_i$ for x_i), if $\tilde{y}_i^* := \mathrm{argmax}_{y \in \mathcal{Y}_i} \mathbf{w} \cdot \mathbf{\Phi}(x_i, y)$ is not y_i, $\mathbf{w}$ is updated to increase the weights corresponding to features in y_i and decrease the weights of features in $\tilde{y}_i^*$. During evaluation, we use parameters averaged over all utterances and over all training epochs. This was shown to give substantial improvements in previous work (Collins, 2002; Roark et al., 2004).

Unfortunately, the conventional perceptron algorithm takes impractically long for the amount of training examples we have. We make use of a distributed training strategy for the structured perceptron that was first introduced in McDonald et al. (2010). The iterative parameter mixing strategy used in this paradigm can be explained as follows: the training data $\mathcal{T} = \{x_i, y_i\}_{i=1}^{\mathcal{N}}$ is suitably partitioned into $\mathcal{C}$ disjoint sets $\mathcal{T}_1, \ldots, \mathcal{T}_\mathcal{C}$. Then, a structured perceptron model is trained on each data set in parallel. After one training epoch, the parameters in the $\mathcal{C}$ sets are mixed together (using a "mixture coefficient" μ_i for each set $\mathcal{T}_i$) and returned 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 framework. The following section describes this infrastructure in greater detail.

2.2 MapReduce implementation details

We propose a distributed infrastructure using MapReduce (Ghemawat and Dean, 2004) to train our large-scale DLMs on terabytes of data. The MapReduce (Ghemawat and Dean, 2004) paradigm, adapted from a specialized functional programming construct, is specialized for use over clusters with

Algorithm 1 Distributed Perceptron (McDonald et al., 2010)

```
Require: Training samples $\mathcal{T} = \{x_i, y_i\}_{i=1}^{\mathcal{N}}$
$\mathbf{w}^0 := [0, \ldots, 0]$
Partition $\mathcal{T}$ into $\mathcal{C}$ parts, $\mathcal{T}_1, \ldots, \mathcal{T}_{\mathcal{C}}$
$[\mu_1, \ldots, \mu_{\mathcal{C}}] := [\frac{1}{\mathcal{C}}, \ldots, \frac{1}{\mathcal{C}}]$
for $t := 1$ to $T$ do
    for $c := 1$ to $\mathcal{C}$ do
        $\mathbf{w} := \mathbf{w}^{t-1}$
        for $j := 1$ to $|\mathcal{T}_c|$ do
            $\tilde{y}_{c,j}^t := \mathrm{argmax}_y \, \mathbf{w} \cdot \mathbf{\Phi}(x_{c,j}, y)$
            $\boldsymbol{\delta} := \mathbf{\Phi}(x_{c,j}, y_{c,j}) - \mathbf{\Phi}(x_{c,j}, \tilde{y}_{c,j}^t)$
            $\mathbf{w} := \mathbf{w} + \boldsymbol{\delta}$
        end for
        $\mathbf{w}_c^t := \mathbf{w}$
    end for
    $\mathbf{w}^t := \sum_{c=1}^{\mathcal{C}} \mu_c \mathbf{w}_c^t$
end for
return $\mathbf{w}^T$
```

a large number of nodes. Chu et al. (2007) have demonstrated that many standard machine learning algorithms can be phrased as MapReduce tasks, thus illuminating the versatility of this framework. In relation to language models, Brants et al. (2007) recently proposed a distributed MapReduce infrastructure to build Ngram language models having up to 300 billion *n*-grams. We take inspiration from this evidence of being able to build very large models and use the MapReduce infrastructure for our DLMs. Also, the MapReduce paradigm allows us to easily fit different variants of our learning algorithm in a modular fashion by only making small changes to the MapReduce functions.

In the MapReduce framework, any computation is expressed as two user-defined functions: *Map* and *Reduce*. The *Map* function takes as input a key/value pair and processes it using user-defined functions to generate a set of intermediate key/value pairs. The *Reduce* function receives all intermediate pairs that are associated with the same key value. The distributed nature of this framework comes from the ability to invoke the *Map* function on different parts of the input data simultaneously. Since the framework assures that all the values corresponding to a given key will be accummulated at the end of all the *Map* invocations on the input data, different machines can simultaneously execute the *Reduce* to operate on different parts of the intermediate data.

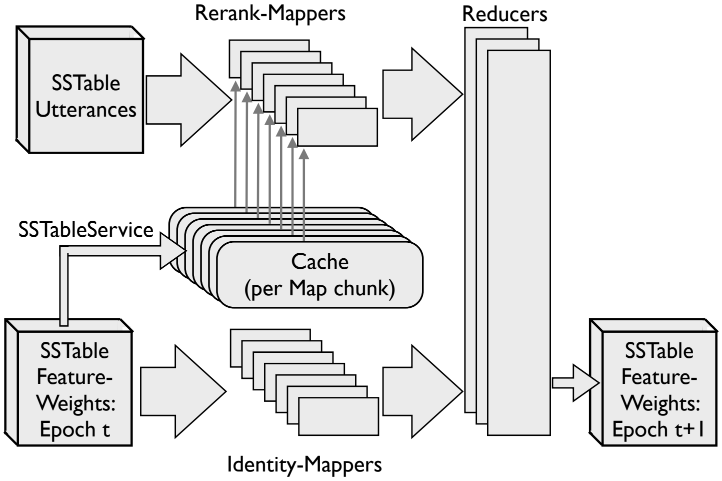
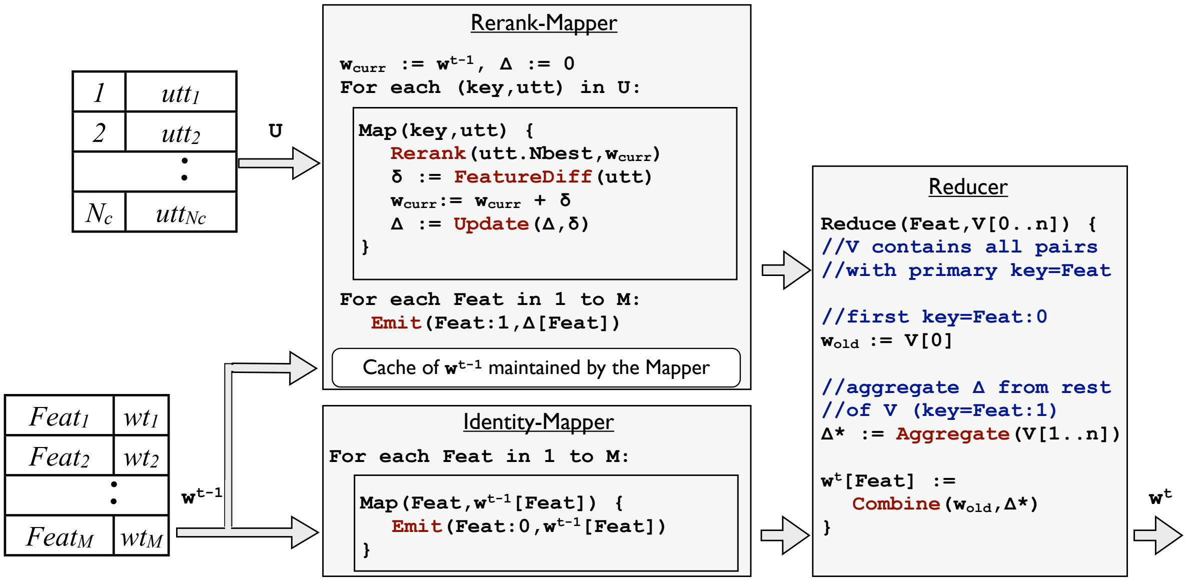


Figure 1: MapReduce implementation of reranking using discriminative language models.

Any MapReduce application typically implements *Mapper*/*Reducer* interfaces to provide the desired *Map*/*Reduce* functionalities. For our models, we use two different Mappers (as illustrated in Figure 1) to compute feature weights for one training epoch. The *Rerank-Mapper* receives as input a set of training utterances and also requests for feature weights computed in the previous training epoch. *Rerank-Mapper* then computes feature updates for the given training data (the subset of the training data received by a single *Rerank-Mapper* instance will be henceforth referred to as a “Map chunk”). We also have a second *Identity-Mapper* that receives feature weights from the previous training epoch and directly maps the inputs to outputs which are provided to the *Reducer*. The *Reducer* combines the outputs from both *Rerank-Mapper* and *Identity-Mapper* and outputs the feature weights for the current training epoch. These output feature weights are persisted on disk in the form of SSTables that are an efficient abstraction to store large numbers of key-value pairs.

The features corresponding to a Map chunk at the end of training epoch need to be made available to *Rerank-Mapper* in the subsequent training epoch. Instead of accessing the features on demand from the SSTables that store these feature weights, every *Rerank-Mapper* stores the features needed for the current Map chunk in a cache. Though the number



Rerank-Mapper

```
$w_{curr}$ := $w^{t-1}$, Δ := 0
For each (key,utt) in U:
  Map(key,utt) {
     Rerank(utt.Nbest,$w_{curr}$)
     δ := FeatureDiff(utt)
     $w_{curr}$:= $w_{curr}$ + δ
     Δ := Update(Δ,δ)
  }
For each Feat in 1 to M:
   Emit(Feat:1,Δ[Feat])
```

Cache of w^{t-1} maintained by the Mapper

Identity-Mapper

```
For each Feat in 1 to M:
  Map(Feat,$w^{t-1}$[Feat]) {
     Emit(Feat:0,$w^{t-1}$[Feat])
  }
```

Reducer

```
Reduce(Feat,V[0..n]) {
//V contains all pairs
//with primary key=Feat

//first key=Feat:0
$w_{old}$ := V[0]

//aggregate Δ from rest
//of V (key=Feat:1)
Δ* := Aggregate(V[1..n])

$w^t$[Feat] :=
   Combine($w_{old}$,Δ*)
}
```

Figure 2: Details of the Mapper and Reducer.

Naive Distributed Perceptron:

- Update(Δ, δ) returns $\Delta + \delta$.
- Aggregate$([\Delta_1^t, \ldots, \Delta_{\mathcal{C}}^t])$ returns $\Delta^* = \sum_{c=1}^{\mathcal{C}} \Delta_c^t$.
- Combine(w_{NP}^{t-1}, Δ^*) returns $w_{NP}^{t-1} + \Delta^*$.

Distributed Perceptron:

- Update and Combine are as for the *Naive Distributed Perceptron*.
- Aggregate$([\Delta_1^t, \ldots, \Delta_{\mathcal{C}}^t])$ returns $\Delta^* = \sum_{c=1}^{\mathcal{C}} \mu_c \Delta_c^t$.

Averaged Distributed Perceptron: Here, $w^t = (w_{AV}^t, w_{DP}^t)$, and $\Delta = (\boldsymbol{\beta}, \boldsymbol{\alpha})$ contain pairs of values; $\boldsymbol{\alpha}$ is used to maintain w_{DP}^t and $\boldsymbol{\beta}$, both of which in turn are used to maintain w_{AV}^t ($\boldsymbol{\alpha}_c^t$ plays the role of Δ_c^t in *Distributed Perceptron*). Only w_{AV}^t is used in the final evaluation and only w_{DP}^t is used during training.

- Update$((\boldsymbol{\beta}, \boldsymbol{\alpha}), \delta)$ returns $(\boldsymbol{\beta} + \boldsymbol{\alpha} + \delta, \boldsymbol{\alpha} + \delta)$.
- Aggregate$([\Delta_1^t, \ldots, \Delta_{\mathcal{C}}^t])$ where $\Delta_c^t = (\boldsymbol{\beta}_c^t, \boldsymbol{\alpha}_c^t)$, returns $\Delta^* = (\boldsymbol{\beta}^*, \boldsymbol{\alpha}^*)$ where $\boldsymbol{\beta}^* = \sum_{c=1}^{\mathcal{C}} \boldsymbol{\beta}_c^t$, and $\boldsymbol{\alpha}^* = \sum_{c=1}^{\mathcal{C}} \mu_c \boldsymbol{\alpha}_c^t$.
- Combine$((w_{AV}^{t-1}, w_{DP}^{t-1}), (\boldsymbol{\beta}^*, \boldsymbol{\alpha}^*))$ returns $(\frac{t-1}{t} w_{AV}^{t-1} + \frac{1}{t} w_{DP}^{t-1} + \frac{1}{Nt} \boldsymbol{\beta}^*, w_{DP}^{t-1} + \boldsymbol{\alpha}^*)$.

Figure 3: Update, Aggregate and Combine procedures for the three variants of the Distributed Perceptron algorithm.

of features stored in the SSTables are determined by the total number of training utterances, the number of features that are accessed by a *Rerank-Mapper* instance are only proportional to the chunk size and can be cached locally. This is an important implementation choice because it allows us to estimate very large distributed models: the bottleneck is no longer the total model size but rather the cache size that is in turn controlled by the Map chunk size. Section 3.2 discusses in more detail about different model sizes and the effects of varying Map chunk size on recognition performance.

Figure 1 is a schematic diagram of our entire framework; Figure 2 shows a more detailed representation of a single *Rerank-Mapper*, an *Identity-Mapper* and a *Reducer*, with the pseudocode of these interfaces shown inside their respective boxes. *Identity-Mapper* gets feature weights from the previous training epoch as input ($\mathbf{w}^t$) and passes them to the output unchanged. *Rerank-Mapper* calls the function Rerank that takes an N-best list of a training utterance (**utt.Nbest**) and the current feature weights

($\mathbf{w}_{curr}$) as input and reranks the N-best list to obtain the best scoring hypothesis. If this differs from the correct transcript for utt, FeatureDiff computes the difference in feature vectors corresponding to the two hypotheses (we call it δ) and $\mathbf{w}_{curr}$ is incremented with δ. Emit is the output function of a Mapper that outputs a processed key/value pair. For every feature Feat, both *Identity-Mapper* and *Rerank-Mapper* also output a secondary key (0 or 1, respectively); this is denoted as Feat:0 and Feat:1. At the *Reducer*, its inputs arrive sorted according to the secondary key; thus, the feature weight corresponding to Feat from the previous training epoch produced by *Identity-Mapper* will necessarily arrive before Feat's current updates from the *Rerank-Mapper*. This ensures that $\mathbf{w}^{t+1}$ is updated correctly starting with $\mathbf{w}^{t}$. The functions Update, Aggregate and Combine are explained in the context of three variants of the distributed perceptron algorithm in Figure 3.

2.2.1 MapReduce variants of the distributed perceptron algorithm

Our MapReduce setup described in the previous section allows for different variants of the distributed perceptron training algorithm to be implemented easily. We experimented with three slightly differing variants of a distributed training strategy for the structured perceptron, *Naive Distributed Perceptron*, *Distributed Perceptron* and *Averaged Distributed Perceptron*; these are defined in terms of Update, Aggregate and Combine in Figure 3 where each variant can be implemented by plugging in these definitions from Figure 3 into the pseudocode shown in Figure 2. We briefly describe the functionalities of these three variants. The weights at the end of a training epoch t for a single feature f are ($w^t_{NP}, w^t_{DP}, w^t_{AV}$) corresponding to *Naive Distributed Perceptron*, *Distributed Perceptron* and *Averaged Distributed Perceptron*, respectively; $\phi(\cdot,\cdot)$ correspond to feature f's value in $\mathbf{\Phi}$ from Algorithm 1. Below, $\delta^t_{c,j} = \phi(x_{c,j}, y_{c,j}) - \phi(x_{c,j}, \tilde{y}^t_{c,j})$ and $\mathcal{N}_c$ = number of utterances in Map chunk $\mathcal{T}_c$.

Naive Distributed Perceptron: At the end of epoch t, the weight increments in that epoch from all map chunks are added together and added to w^{t-1}_{NP} to obtain w^t_{NP}.

Distributed Perceptron: Here, instead of adding increments from the map chunks, at the end of epoch t, they are averaged together using weights μ_c, $c = 1$ to $\mathcal{C}$, and used to increment w^{t-1}_{DP} to w^t_{DP}.

Averaged Distributed Perceptron: In this variant, firstly, all epochs are carried out as in the Distributed Perceptron algorithm above. But at the end of t epochs, all the weights encountered during the whole process, over all utterances and all chunks, are averaged together to obtain the final weight w^t_{AV}. Formally,

$$w^t_{AV} = \frac{1}{\mathcal{N} \cdot t} \sum_{t'=1}^{t} \sum_{c=1}^{\mathcal{C}} \sum_{j=1}^{\mathcal{N}_c} w^{t'}_{c,j},$$

where $w^t_{c,j}$ refers to the current weight for map chunk c, in the t^{th} epoch after processing j utterances and $\mathcal{N}$ is the total number of utterances. In our implementation, we maintain only the weight w^{t-1}_{DP} from the previous epoch, the cumulative increment $\gamma^t_{c,j} = \sum_{k=1}^{j} \delta^t_{c,k}$ so far in the current epoch, and a running average w^{t-1}_{AV}. Note that, for all c, j, $w^t_{c,j} = w^{t-1}_{DP} + \gamma^t_{c,j}$, and hence

$$\begin{aligned} \mathcal{N}t \cdot w^t_{AV} &= \mathcal{N}(t-1) w^{t-1}_{AV} + \sum_{c=1}^{\mathcal{C}} \sum_{j=1}^{\mathcal{N}_c} w^t_{c,j} \\ &= \mathcal{N}(t-1) w^{t-1}_{AV} + \mathcal{N} w^{t-1}_{DP} + \sum_{c=1}^{\mathcal{C}} \boldsymbol{\beta}^t_c \end{aligned}$$

where $\boldsymbol{\beta}^t_c = \sum_{j=1}^{\mathcal{N}_c} \gamma^t_{c,j}$. Writing $\boldsymbol{\beta}^* = \sum_{c=1}^{\mathcal{C}} \boldsymbol{\beta}^t_c$, we have $w^t_{AV} = \frac{t-1}{t} w^{t-1}_{AV} + \frac{1}{t} w^{t-1}_{DP} + \frac{1}{\mathcal{N}t} \boldsymbol{\beta}^*$.

3 Experiments and Results

Our DLMs are evaluated in two ways: 1) we extract a development set (*weakAM-dev*) and a test set (*weakAM-test*) from the speech data that is re-decoded with a *weakAM* to evaluate our learning setup, and 2) we use a standard voice-search test set (*v-search-test*) (Strope et al., 2011) to evaluate actual ASR performance on voice-search. More details regarding our experimental setup along with a discussion of our experiments and results are described in the rest of the section.

3.1 Experimental setup

We generate training lattices using speech data that is re-decoded with a *weakAM* acoustic model and

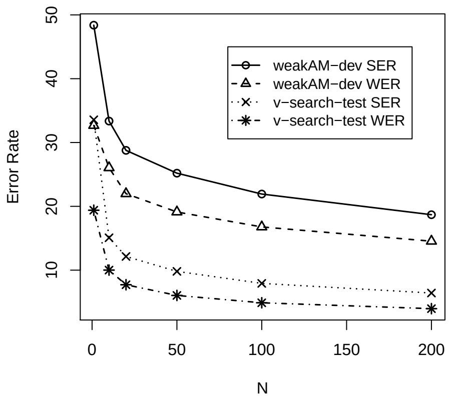


Figure 4: Oracle error rates at word/sentence level for *weakAM-dev* with the weak AM and *v-search-test* with the baseline AM.

a baseline language model. We use maximum likelihood trained single mixture Gaussians for our *weakAM*. And, we use a sufficiently small baseline LM ($\sim$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. Chelba et al. (2010) demonstrate that it takes much larger LMs to get a significant relative gain in WER. Our largest models are trained on 87,000 hours of speech, or $\sim$350 million words (*weakAM-train*) obtained by filtering voice-search logs at 0.8 confidence, and re-decoding the speech data with a *weakAM* to generate N-best lists. We set aside a part of this *weakAM-train* data to create *weakAM-dev* and *weakAM-test*: these data sets consist of 328,460/316,992 utterances, or 1,182,756/1,129,065 words, respectively. We use a manually-transcribed, standard voice-search test set (*v-search-test* (Strope et al., 2011)) consisting of 27,273 utterances, or 87,360 words to evaluate actual ASR performance using our *weakAM*-trained models. All voice-search data used in the experiments is anonymized.

Figure 4 shows oracle error rates, both at the sentence and word level, using N-best lists of utterances in *weakAM-dev* and *v-search-test*. These error rates are obtained by choosing the best of the top N hypotheses that is either an exact match (for sentence error rate) or closest in edit distance (for word error rate) to the correct transcript. The N-best lists for *weakAM-dev* are generated using a weak AM and N-best lists for *v-search-test* are generated using the baseline (strong) AM. Figure 4 shows these error rates plotted against a varying threshold N for the N-best lists. Note there are sufficient word errors in the *weakAM* data to train DLMs; also, we observe that the plot flattens out after N=100, thus informing us that N=100 is a reasonable threshold to use when training our DLMs.

Experiments in Section 3.2 involve evaluating our learning setup using *weakAM-dev/test*. We then 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 string edit distance used in training the DLM. More details about the implications of this text normalization on WER can be found in Section 3.3.

3.2 Evaluating our DLM rescoring framework on *weakAM-dev/test*

Improvements on *weakAM-dev* using different variants of training for the DLMs

We evaluate the performance of all the variants of the distributed perceptron algorithm described in Section 2.2 over ten training epochs using a DLM trained on $\sim$20,000 hours of speech with trigram word features. Figure 5 shows the drop in WER for all the three variants. We observe that the *Naive Distributed Perceptron* gives modest improvements in WER compared to the baseline WER of 32.5%. However, averaging over the number of Map chunks as in the *Distributed Perceptron* or over the total number of utterances and training epochs as in the *Averaged Distributed Perceptron* significantly improves recognition performance; this is in line with the findings reported in Collins (2002) and McDonald et al. (2010) of averaging being an effective way of adding regularization to the perceptron algorithm.

Our best-performing *Distributed Perceptron* model gives a 4.7% absolute ($\sim$15% relative) improvement over the baseline WER of 1-best hypotheses in *weakAM-dev*. This, however, could be attributed to a combination of factors: the use of large amounts of additional training data for the DLMs or the discriminative nature of the model. In order to isolate the improvements brought upon mainly by the second factor, we build an ML trained backoff trigram LM (ML-3gram) using the

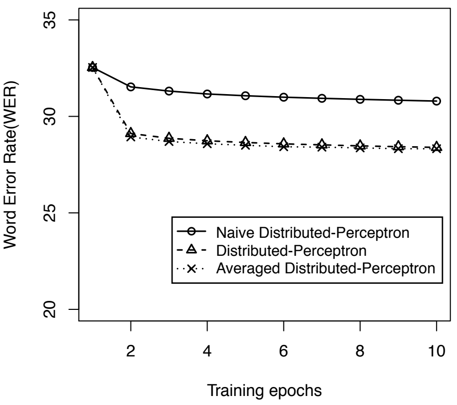


Figure 5: Word error rates on *weakAM-dev* using *Perceptron*, *Distributed Perceptron* and *AveragedPerceptron* models.

reference transcripts of all the utterances used to train the DLMs. The N-best lists in *weakAM-dev* are reranked using ML-3gram probabilities linearly interpolated with the LM probabilities from the lattices. We also experiment with a log-linear interpolation of the models; this performs slightly worse than rescoring with linear interpolation.

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

Impact of varying orders of N-gram features

Table 1 shows that our best performing model (DLM-3gram) gives a significant ∼2% absolute (∼6% relative) improvement over ML-3gram. We also observe that most of the improvements come from the unigram and bigram features. We do not expect higher order N-gram features to significantly help recognition performance; we further confirm this by building DLM-4gram and DLM-5gram that use up to 4-gram and 5-gram word features, respectively. Table 2 gives the progression of WERs for six epochs using DLM-3gram, DLM-4gram and DLM-5gram showing minute improvements as we increase the order of Ngram features from 3 to 5.

Table 2: WERs on *weakAM-dev* using DLM-3gram, DLM-4gram and DLM-5gram of six training epochs.

Iteration	DLM-3gram (%)	DLM-4gram (%)	DLM-5gram (%)
1	32.53	32.53	32.53
2	29.52	29.47	29.46
3	29.26	29.23	29.22
4	29.11	29.08	29.06
5	29.01	28.98	28.96
6	28.95	28.90	28.87

Impact of model size on WER

We experiment with varying amounts of training data to build our DLMs and assess the impact of model size on WER. Table 3 shows each model along with its size (measured in total number of word features), coverage on *weakAM-test* in percent of tokens (number of word features in *weakAM-test* that are in the model) and WER on *weakAM-test*. As expected, coverage increases with increasing model size with a corresponding tiny drop in WER as the model size increases. To give an estimate of the time complexity of our MapReduce, we note that Model1 was trained in ≈1 hour on 200 mappers with a Map chunk size of 2GB. "Larger models", built by increasing the number of training utterances used to train the DLMs, do not yield significant gains in accuracy. We need to find a good way of adjusting the model capacity with increasing amounts of data.

Impact of varying Map chunk sizes

We also experiment with varying Map chunk sizes to determine its effect on WER. Figure 6 shows WERs on *weakAM-dev* using our best *Distributed Perceptron* model with different Map chunk sizes (64MB, 512MB, 2GB). For clarity, we examine two limit cases: a) using a single Map chunk for the entire training data is equivalent to the conventional structured perceptron and b) using a single training in-

Table 3: 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

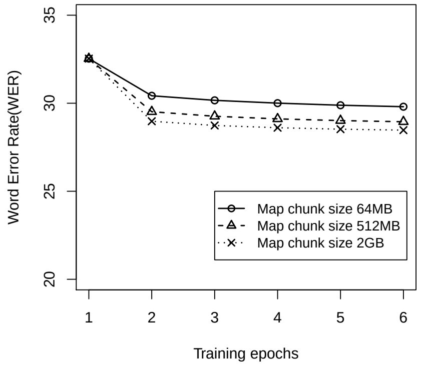


Figure 6: Word error rates on *weakAM-dev* using varying Map chunk sizes of 64MB, 512MB and 2GB.

stance per Map chunk is equivalent to batch training. We observe that moving from 64MB to 512MB significantly improves WER and the rate of improvement in WER decreases when we increase the Map chunk size further to 2GB. We attribute these 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.

3.3 Evaluating ASR performance on *v-search-test* using DLM rescoring

We evaluate our best *Distributed Perceptron* DLM model on *v-search-test* lattices that are generated using a strong AM. We hope that the large relative gains on *weakAM-dev/test* translate to similar gains on this standard voice-search data set as well. Table 4 shows the WERs on both *weakAM-test* and *v-search-test* using Model 1 (from Table 3)[1]. We observe a small but statistically significant ($p < 0.05$) reduction (∼2% relative) in WER on *v-search-test* over reranking with a linearly interpolated ML-3gram. This is encouraging because we attain this improvement using training lattices that were generated using a considerably weaker AM.

Table 4: 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

It is instructive to analyze why the relative gains in performance on *weakAM-dev/test* do not translate to *v-search-test*. Our DLMs are built using N-best outputs from the recognizer that live in the "spoken domain" (SD) and the manually transcribed *v-search-data* transcripts live in the "written domain" (WD). The normalization of training data from WD to SD is as described in Chelba et al. (2010); inverse text normalization (ITN) undoes most of that when moving text from SD to WD, and it is done in a heuristic way. There is ∼2% absolute reduction in WER when we move the N-best from SD to WD via ITN; this is how WER on *v-search-test* is computed by the voice-search evaluation code. Contrary to this, in DLM training we compute WERs using string edit distance between test data transcripts and the N-best hypotheses and thus we ignore the mismatch between domains WD and SD. It is quite likely that part of what the DLM learns is to pick N-best hypotheses that come closer to WD, but may not truly result in WER gains after ITN. This would explain part of the mismatch between the large relative gains on *weakAM-dev/test* compared to the smaller gains on *v-search-test*. We could correct for this by applying ITN to the N-best lists from SD to move to WD before computing the oracle best in the list. An even more desirable solution is to build the LM directly on WD text; text normalization would be employed for pronunciation generation, but ITN is not needed anymore (the LM picks the most likely WD word string for homophone queries at recognition).

4 Conclusions

In this paper, we successfully build large-scale discriminative N-gram language models with lattices regenerated using a weak AM and derive small but significant gains in recognition performance on a voice-search task where the lattices are generated using a stronger AM. We use a very simple weak AM and this suggests that there is room for improvement if we use a slightly better "weak AM". Also, we have a scalable and efficient MapReduce implementation that is amenable to adapting minor changes to the training algorithm easily and allows for us to train large LMs. The latter functionality will be particularly useful if we generate the contrastive set by sampling from text instead of redecoding logs (Jyothi and Fosler-Lussier, 2010).

[1] We also experimented with the larger Model 4 and saw similar improvements on *v-search-test* as with Model 1.

References

Thorsten Brants, Ashok C. Popat, Peng Xu, Franz J. Och, and Jeffrey Dean. 2007. Large language models in machine translation. In *Proc. of EMNLP*, pages 858–867.

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

C.T. Chu, S.K. Kim, Y.A. Lin, Y.Y. Yu, G. Bradski, A.Y. Ng, and K. Olukotun. 2007. Map-reduce for machine learning on multicore. *Proc. NIPS*, 19:281.

M. Collins. 2002. Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms. In *Proc. EMNLP*.

J. Gao, H. Yu, W. Yuan, and P. Xu. 2005. Minimum sample risk methods for language modeling. In *Proc. of EMNLP*.

S. Ghemawat and J. Dean. 2004. Mapreduce: Simplified data processing on large clusters. In *Proc. OSDI*.

K.B. Hall, S. Gilpin, and G. Mann. 2010. MapReduce/Bigtable for distributed optimization. In *NIPS LCCC Workshop*.

P. Jyothi and E. Fosler-Lussier. 2010. Discriminative language modeling using simulated ASR errors. In *Proc. of Interspeech*.

P. Jyothi, L. Johnson, C. Chelba, and B. Strope. 2012. Distributed discriminative language models for Google voice-search. In *Proc. of ICASSP*.

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

R. McDonald, K. Hall, and G. Mann. 2010. Distributed training strategies for the structured perceptron. In *Proc. NAACL*.

B. Roark, M. Saraçlar, M. Collins, and M. Johnson. 2004. Discriminative language modeling with conditional random fields and the perceptron algorithm. In *Proc. ACL*.

R. Schlüter, B. Müller, F. Wessel, and H. Ney. 1999. Interdependence of language models and discriminative training. In *Proc. ASRU*.

B. Strope, D. Beeferman, A. Gruenstein, and X. Lei. 2011. Unsupervised testing strategies for ASR. In *Proc. of Interspeech*.

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