THE UNIVERSITY *of* EDINBURGH

Iterative Parameter Mixing for Distributed Large-Margin Training of Structured Predictors for Natural Language Processing

Greg Coppola

Doctor of Philosophy

Institute for Language, Cognition and Computation
School of Informatics
University of Edinburgh

2014

Abstract

The development of distributed training strategies for statistical prediction functions is important for applications of machine learning, generally, and the development of distributed structured prediction training strategies is important for **natural language processing** (NLP), in particular. With ever-growing data sets this is, first, because, it is easier to increase computational capacity by adding more processor nodes than it is to increase the power of individual processor nodes, and, second, because data sets are often collected and stored in different locations.

Iterative parameter mixing (IPM) is a distributed training strategy in which each node in a network of processors optimizes a regularized average loss objective on its own subset of the total available training data, making *stochastic* (per-example) updates to its own estimate of the optimal weight vector, and communicating with the other nodes by periodically averaging estimates of the optimal vector across the network. This algorithm has been contrasted with a close relative, called here the **single-mixture** optimization algorithm, in which each node stochastically optimizes an average loss objective on its own subset of the training data, operating in isolation until convergence, at which point the average of the independently created estimates is returned. Recent empirical results have suggested that this IPM strategy produces better models than the single-mixture algorithm, and the results of this thesis add to this picture.

The contributions of this thesis are as follows.

The first contribution is to produce and analyze an algorithm for decentralized stochastic optimization of regularized average loss objective functions. This algorithm, which we call the **distributed regularized dual averaging** algorithm, improves over prior work on distributed dual averaging by providing a simpler algorithm (used in the rest of the thesis), better convergence bounds for the case of regularized average loss functions, and certain technical results that are used in the sequel.

The central contribution of this thesis is to give an optimization-theoretic justification for the IPM algorithm. While past work has focused primarily on its empirical test-time performance, we give a novel perspective on this algorithm by showing that, in the context of the distributed dual averaging algorithm, IPM constitutes a convergent optimization algorithm for arbitrary convex functions, while the single-mixture distribution algorithm is not. Experiments indeed confirm that the superior test-time performance of models trained using IPM, compared to single-mixture, correlates with

better optimization of the objective value on the training set, a fact not previously reported. Furthermore, our analysis of general non-smooth functions justifies the use of distributed large-margin (**support vector machine** [SVM]) training of structured predictors, which we show yields better test performance than the IPM perceptron algorithm, the only version of the IPM to have previously been given a theoretical justification. Our results confirm that IPM training can reach the same level of test performance as a sequentially trained model and can reach better accuracies when one has a fixed budget of training time.

Finally, we use the reduction in training time that distributed training allows to experiment with adding **higher-order dependency features** to a state-of-the-art **phrase-structure** parsing model. We demonstrate that adding these features improves **out-of-domain** parsing results of even the strongest phrase-structure parsing models, yielding a new state-of-the-art for the popular train-test pairs considered. In addition, we show that a **feature-bagging** strategy, in which component models are trained separately and later combined, is sometimes necessary to avoid feature under-training and get the best performance out of large feature sets.

Acknowledgements

Equally well-versed in jazz, children's language, or robots, Mark Steedman is the coolest. I have learned learned as much as I ever hoped to in a lifetime in this time here under Mark's supervision and in the environment of this department which to a great extent reflects his influence. I'd also like to thank the people of Edinburgh, who have one of the world's most charming cities that, with its stormy and perenially overcast skies, makes a great home for thinkers. I'd like to thank Charles Sutton for always letting me ask him questions, Shay Cohen for managing the examination of my thesis, and Sebastian Riedel for examining. I'd like to thank Alex Lascarides for her advice in my first year. Also, I'd like to thank the Scottish Informatics and Computer Science Alliance, the School of Informatics and the European Research Council for financial support.

I'd like to thank my family, for helping me so much over all these years, espcially my mother and father, my sister, and my Zia. I'd like to thank grandmother my Adelina, who passed away several years ago, and to thank my grandfather Verino and great uncle Mario, two great, stylish, and resilient men who passed away while I was in Edinburgh.

I'd like to thank the friends I've had in Edinburgh, especially, in order of appearance: Mike Ferguson ("Chomsky?" "That's not much time for pleasure."), Hannah Douglas ("Three out of four people, and two out of two Hannah's…" "You roots, girl." "The Welsh have hearts as big as the Mexicans." "#*wearethe*99."), Emily Rose Doucet ("We're out of cream, would you like it with no milk instead?" "In Canada, some women don't even have them." "I call it Raul.") and Sarah Rose Carlton (Somehow, I can't remember many funny things she said, but she always looked amazing.), Jose Camunez ("Why did you think?"), Belen Santervas ("It has bells!"), Steve Calvert ("I like you… but you're crazy."), Bastow, Ally and Nigel ("Do you want to go to the hospital?" "No. I'll just have a cup of tea."), Sabine Urban ("*E.T. nach hause telefonieren.*"), Susie Doucet ("The Black Ties"), Linda Tym ("Ohhhhhh…"), Maria Ore ("Every time you say 'computational linguistics,' …"), Mary Coote ("Twenty-four, boar.") and Andrew Johnson, Alan "Best hair in Edinburgh" Mackenzie, Kieran Curran ("DANCE!"), Calen Walse ("You gonna grind?" "So, are you a rated player?" "Face control." "No blunders." "He's the Godfather."), Dani Walshe ("There's no such thing a pig climber."), Devon Walshe ("Greg's big fat Italian foot." "Unmitigated."), Bruno Panara ("greg.hat"), Olly Treadway ("Tokens to heaven."), Gianluca Trombetta ("OK. You can put chillies."), Rohit Marwah ("That's why she bought heart pillows and put them in window."), Lila Matsumoto ("Oh god." "We can put that on a t-shirt

for you.") and Victor Hernandez-Urbina ("*Nunca mames*!" "*La polizia de azucar.*"). I'd also like to thank Walky Inzunza Romero ("*La fuente de la vida.*" "*Palenque, Palenque*."), who I talked to as much as anyone else during this PhD, even though she was kind of far away.

I'd like to thank all the ilcc-party'ers, especially: Mark Granroth-Wilding (and Hanna) (Efficiency improvements, generalization, and gold-plating.) Aciel Eshky (and Ben) ("Quit busting my chops."), the real Dave Matthews (Best parties in ILCC), Stella Frank, Des Elliot, Bharat Ambati ("The B-Man." "Bollywood." Your Indian magic on the night of my viva.), Mike "The Lewiser" Lewis (and Cat) ("Dumb dumb."), Christos "Loopy" C..., semantics master Siva Reddy (and Spandana) ("The S-Man."), Alireza Pourranjbar, Arm Boonkwan, Diego Frasinelli, Tom Kwiatkowski, Teju Deoskar, Moreno Coco, Carina Silberer, Lea Frerman, Francesco Sartorio ("This is the true meaning of the type raising.") and Omri Abend ("Well...").

Declaration

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

(*Greg Coppola*)

Table of Contents

Notation

Sequences and Sums Where α_k is any indexed expression, and $S = \left(i_1, i_2, \ldots, i_{|S|}\right)$ is any index sequence, we use $(\alpha_i)_{i \in S}$ to denote the sequence $\left(\alpha_{i_1}, \alpha_{i_2}, \ldots, \alpha_{i_{|S|}}\right)$. Similarly, $\sum_{i \in S} \alpha_i$ is the sum $\alpha_{i_1} + \alpha_{i_2} + \cdots + \alpha_{i_{|S|}}$. For any $K \in \mathbb{N}$, we use $[K]$ to denote the index sequence $(1, 2, \ldots, K)$.

Conventional Symbols We have endeavoured to use the following symbols as consistently as possible throughout this thesis:

D	The dimension of a given feature space
$\mathbf{d}$	A dual representation of a weight vector (see §4)
$[M]$	The indices of the elements in a set of training data
m	The index of a given element of a set of training data
$[N]$	The indices of the nodes in a network of processors
n	The index of a node in a network of processors
$\mathbb{R}$	The set of real numbers
$[T]$	The indices of the time steps an optimization procedure is run for
t	The index of a given time step of an optimization procedure
$\mathbf{w}$	A weight vector in $\mathbb{R}^D$
$\mathcal{X}$	A set of input objects
$\mathcal{Y}$	A set of output objects, or training examples
Φ	A feature embedding function, $\Phi : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}^D$
Ψ	An arbitrary regularization function, $\Psi : \mathbb{R}^D \to \mathbb{R}$

Due to its popularity in other contexts, the symbol n is one symbol that will often appear temporarily with other meanings, e.g., to denote the length of a sentence.

Chapter 1

Introduction

1.1 Overview

The focus of this thesis is on distributed training of structured predictors, taking a distributed optimization perspective, with a special interest in statistical parsing. As we will see, interest in distributed training is important for progress in *natural language processing* (NLP), because the cheapest feasible way to grow computational power is to distribute processing across a network of computing cores.

Training as Optimization Creating and training a structured prediction function for an NLP application requires a series of choices—one must choose a decoding algorithm, feature set, model family, and loss function, among other things—but, in almost all cases, the final step in the training process is some kind of numerical optimization procedure. This is almost universally true for supervised learning (§2). It is also true, for example, for the M-step of the expectation maximization algorithm for unsupervised or semi-supervised learning, and is used in each training step in a self-training semi-supervised learning strategy (§6.1.2). In the standard case, a model is parameterized as a vector $\mathbf{w} \in \mathbb{R}^D$, often called a **weight vector** (because it assigns positive and negative "weight" to various kinds of features), and one tries to minimize regularized average loss on some set of training data. That is, suppose that $\Psi : \mathbb{R}^D \to \mathbb{R}$ is a convex regularization function, introduced to prevent over-fitting and to keep weights from tending towards infinity, that $\mathcal{Y}$ is a space of training examples and that $\ell : \mathbb{R}^D \times \mathcal{Y} \to \mathbb{R}$ is a function that measures the *loss*, or inappropriateness, of vector $\mathbf{w}$ for each example $z \in \mathcal{Y}$. And, suppose we have access to a set $S = (z_m)_{m \in [M]}$ of training examples. Then,

the **minimization of regularized average loss** problem is to solve:

$$\arg\min_{\mathbf{w}} \Psi(\mathbf{w}) + \frac{1}{|S|} \sum_{z \in S} \ell(\mathbf{w}, z) \tag{1.1}$$

The numerical solution, say $\mathbf{w}^*$, to (1.1) serves as the set of parameter values for the prediction model that will be tested or used in some application. The minimization of loss on a fixed data set is called the minimization of **empirical loss**, and has connections with the minimization of **expected risk**, or loss to be expected on the entire population of examples of interest (cf. §s 2, 3).

Distributed Data Sets Modern data sets can be too large to fit on a single computer's hard drive. Some very large organizations have needed to distribute their file systems across a networks of computers, e.g., Google's distributed file system (Ghemawat et al., 2003). The distributed nature of the storage of this data means that optimizing a function like (1.1) is not as straightforward as if all of the data was on a single machine. Instead, as we will see, a distributed form for (1.1), such as (1.5) can be more appropriate. This distributes the optimization of the training objective to the entire network.

The Desire for Multi-Core Processing *Moore's law* (Moore et al., 1965) observed that the density of computer circuits on computer chips doubled roughly every 18 months. For many years, this meant a correspondingly exponential growth in computing power, on which developers came to depend in order to make their programs run faster (Sutter, 2005). Although it remains possible to continue doubling the circuit density, the power consumption required to run such circuits grows at such a rate as to be prohibitive, because circuit complexity grows faster than performance when additional circuits are added to a single computing core (Gelsinger, 2001). Thus, a more efficient strategy to continue increasing computational power, and the one adopted recently in practice, is to keep the performance of a single computing core fixed, but to add more computing cores (Chu et al., 2007). This has lead to a wide interest in all branches of computer science towards scaling up computing power using strategies that can incorporate various *multi-core* strategies. As data sets of interest grow, optimization time on a single core will, if the data is varied enough, take longer, and it is not a reasonable strategy to continue carrying out the computation (1.1) on a single core.

Types of Multi-Core Processing The addition of more processor cores can take two principal forms: **shared memory parallel architectures** (or **symmetric multi processing systems**) and **distributed architectures**. Shared memory architectures are

characterized by a number of processing cores which have access to the same short-term and long-term memory stores (i.e., random access memory and hard disk, respectively). Distributed architectures, in contrast, involve networks of separate machines, in which each processing node has its own short-term and long-term memory stores. For small projects, a shared memory parallel architecture is often preferable. It is easier to write and reason about shared memory parallel programs because the computer code running on each core can hold references to the same data. But, in many ways, a more sought-after goal is the ability to understand and improve distributed computation. This is because a network of essentially unbounded size can be created out of a number of ordinary computers, or "commodity machines" (Chang et al., 2008; Dean & Ghemawat, 2008), and at less cost to install and replace. For example, the *Blacklight* computer system, which comprises 4096 computing cores, was the world's largest shared memory computing system as of 2013 (Corp., 2011). Supported by a $3 million grant, it is a highly specialized and expensive piece of equipment. A network of commodity machines would be cheaper to build (with 8-core processors currently costing under $300), and easier to repair in part. Note that, in practice, a hybrid system is also possible, in the sense that one might have a distributed network of computing nodes, each of which contains a small number of computing cores locally sharing the same memory. In that case, communication between the small number of cores that are co-located behaves as a shared-memory system, but communication between nodes must take place over the network.

The Consensus Optimization Problem A natural way to conceive of the problem of optimization over a distributed network is to frame the following *consensus optimization problem* (cf. §3.5). We suppose that each node $n \in [N]$ in a network has *access to* its own local function $f_n : \mathbb{R}^D \to \mathbb{R}$, and the goal of the network is to collaborate to find a single vector in some convex set $\mathcal{W}$ that is optimal for the average of those functions:

$$\arg\min_{\mathbf{w}\in\mathcal{W}} \frac{1}{N} \sum_{n\in[N]} f_n(\mathbf{w}) \tag{1.2}$$

By saying that node n has "access to" f_n, we usually mean that only node n can query for gradients of the function f_n. In general, whatever information interface is provided to the functions f_n, we assume that only node n can use that interface, and that all other nodes must learn about f_n by communicating with node n.

The consensus optimization problem is perfectly suited to the optimization of reg-

	Shared Memory Parallel System	Distributed System
Hard Disk	Each core accesses same hard disk	Each node has separate hard disk
RAM	Each core accesses same RAM	Each node has separate RAM
Communication	Communication has essentially no cost, since an object loaded to RAM for one thread is visible to all	Communication incurs a network penalty. In order to communicate an object, one node must serialize and send it across the network.
Scaling	Adding more cores is relatively difficult—machines a machine with thousands of shared cores is very rare.	Scaling to thousands of cores is possible with relatively inexpensive hardware.
Concurrent Access to Mutable State	Errors due to concurrent access to mutable objects are possible, and must be co-ordinated (e.g., using locks) between threads running on same machine.	Errors due to concurrent access between nodes are not possible, as each node has access to its own memory.

Table 1.1: Comparison of shared memory and distributed systems. Each "node" in a distributed system might actually be a shared memory system with a processor containing a small number (e.g., 2, 4, or 8) processing cores.

ularized average loss over a data set that is spread across a network of N nodes. Suppose we split the training set S of (1.1) into shards $(S_n)_{n\in[N]}$, and each node $n \in [N]$ has access only the examples in S_n. These examples can, in general, be different. For simplicity, we will assume each shard S_n has the same size as any other shard.[1] Then, in order to minimize average regularized loss across all examples, the optimization problem we need to solve is:

$$\arg\min_{\mathbf{w}} \Psi(\mathbf{w}) + \frac{1}{|S|}\sum_{z\in S}\ell(\mathbf{w},z) \tag{1.3}$$

$$= \arg\min_{\mathbf{w}} \Psi(\mathbf{w}) + \frac{1}{|S|}\sum_{n\in[N]}\sum_{z\in S_n}\ell(\mathbf{w},z) \tag{1.4}$$

$$= \arg\min_{\mathbf{w}} \frac{1}{N}\sum_{n\in[N]}\left[\Psi(\mathbf{w}) + \frac{1}{|S_n|}\sum_{z\in S_n}\ell(\mathbf{w},z)\right] \tag{1.5}$$

Then, the f_n of (1.2) is the $\Psi(\mathbf{w}) + \frac{1}{|S_n|}\sum_{z\in S_n}\ell(\mathbf{w},z)$ of (1.5). It is important to note that, since each shard S_n is different, each node has access to a potentially very different function.

Thus, it is the distributed optimization problem of the form (1.2) that we offer a novel solution to and perspective of in this thesis. The central characteristics of the strategy we present are:

1. The data are not shared between nodes. Each node has access only to its own local store of (different) training examples.

2. Nodes must communicate in order to find the globally optimal weight vector, with communication occurring relatively infrequently (the costs of network communication will be considered experimentally in §5.8.7).

Large-Margin Learning **Perceptron training** for binary classification was one of the early forms of classifier training, and has more recently been successfully extended to do structured prediction training (see §2.3). In the cases of both binary prediction and structured prediction, it is known that **large-margin training**, using the **support vector machine** (SVM) objective, will often train models that perform better than does the perceptron (see §2). The SVM objective can be cast as a problem of minimizing regularized average loss, where the loss function chosen encourages a significant

[1] If this were not true, it would be possible to weight the various parts of the objective function according to the number of examples in each, if desired.

prediction margin between correct and incorrect outputs. We will below review a distribution strategy for perceptron training called *iterative parameter mixing*. A central contribution of this thesis is to extend the analysis of iterative parameter mixing, from the case of perceptron training, to which it was first applied, to cover the case of SVM training also. Our experiments in §5 confirm that distributed SVM training outperforms distributed perceptron training, and thus constitutes an improvement over past work in this area.

Structured Prediction and Stochastic Optimization Prediction problems for NLP applications are typically *structured prediction* problems (§2), which involve the use of inference functions that are more complicated and time-consuming than predictors for binary classification. Thus, it has become virtually universal to train complex structured predictors, like parsers (Collins, 2002; McDonald et al., 2005) and translation systems with many features (Chiang, 2012) using stochastic training, in which estimates of the optimal weight vector are updated after each example, rather than use batch training systems (stochastic and batch training are reviewed in §3). Thus, an important desideratum for any algorithm that solves the distributed optimization problem (1.2) is that it uses stochastic training. The experiments of (5.8.6) corroborate this finding, and show that, in the distributed setting tested, distributed stochastic methods out-perform distributed batch ones for the training of structured predictors.

Iterative Parameter Mixing for Distributed Optimization McDonald et al. (2010) study a perceptron-based (see §2.3) solution to the distributed training problem called **iterative parameter mixing**, shown as Algorithm 1. Training data is partitioned into separate shards $(S_n)_{n\in[N]}$, where each part S_n resides on node $n \in [N]$. The algorithm takes place in rounds. On each round, each processing node makes a single perceptron pass over its own training data. After each such pass, the nodes communicate with a central server, which averages each node's estimates of the optimal weight vector. The central server then broadcasts the last round's average to each node, to begin the next pass over the data. Following the usual perceptron-style analysis (Block, 1962; Novikoff, 1962; Collins, 2002), this algorithm is not conceived of or analyzed by McDonald et al. (2010) as solving an optimization problem, *per se*. Rather, it is thought of as the search for a separating hyper-plane, and McDonald et al. (2010) give a convergence proof bounding convergence time for training assuming the training data are perfectly separable by a certain margin. McDonald et al. (2010) contrast the IPM training algorithm with another strategy that we will call the **single-mixture perceptron**, shown as

Algorithm 2. In the single-mixture algorithm, each node repeatedly iterates through its own data set, doing perceptron training as usual. Nodes do not communicate until the final stage, at which time the estimates of the different nodes in the network are averaged. The IPM and single-mixture algorithms are almost entirely alike, except for the communication (or lack thereof) after each epoch through the data. The single-mixture algorithm is arguably optimal from the point of communication cost, since it involves only one round of communication. However McDonald et al. (2010) show that IPM can return models as accurate as those trained sequentially, but that the performance of models trained using the single-mixture strategy is significantly worse than models trained with a single-core perceptron algorithm. Note that Algorithms 1 and 2 return output simply the final weight vector from training, but can be modified to instead output the *average* weight vector over all rounds on all nodes (§5.6).

Algorithm 1 Iterative Parameter Mixing Perceptron

procedure IPM-PERCEPTRON(Data set shards: $(S_n)_{n\in[N]}$)
  $\bar{\mathbf{w}}_0 = 0$
  for $e = 1, \cdots, E$ **do** ▷ Do E epochs of training
    $\mathbf{w}_{e,n} = OneEpochPerceptron(S_n, \bar{\mathbf{w}}_{e-1})$ ▷ N threads run in parallel
    $\bar{\mathbf{w}}_e = \frac{1}{N}\sum_{n\in[N]} \mathbf{w}_{e,n}$ ▷ averaging step
  return $\bar{\mathbf{w}}_E$

procedure ONEEPOCHPERCEPTRON($S, \mathbf{w}$)
  $\mathbf{w}_0 = \mathbf{w}$
  for $m \in 1, \cdots, |S|$ **do**
    draw example $(\mathbf{x}_m, \mathbf{y}_m)$ from S
    $\hat{y}_m = \arg\max_{y\in\mathcal{C}(\mathbf{x}_m)} \langle \mathbf{w}_{m-1}, \Phi(y) \rangle$
    if $\hat{y}_m \neq \mathbf{y}_m$ **then**
      $\mathbf{w}_m \leftarrow \mathbf{w}_{m-1} + [\Phi(\mathbf{y}_m) - \Phi(\hat{y})]$
    else
      $\mathbf{w}_m \leftarrow \mathbf{w}_{m-1}$
  return $\mathbf{w}_{|S|}$

As we use the term, defining characteristic of the IPM training strategy, is the mix of rounds of local stochastic training on each node in the network, interspersed with communication between nodes by averaging of estimates of the sought-after weight vector across the network. This general form for IPM is depicted in Algorithm 3, where each **s** is an **optimizer state**, representing an estimate of the optimal weight

Algorithm 2 Single-Mixture Perceptron

procedure SM-PERCEPTRON(Data set shards: $(S_n)_{n\in[N]}$)
  for $n = 1, \cdots N$ **do** ▷ Each $n \in [N]$ nodes runs in parallel
    $\mathbf{w}_{0,n} = 0$ for $n \in [N]$
    for $e = 1, \cdots, E$ **do** ▷ Do E epochs of training
      $\mathbf{w}_{e,n} = OneEpochPerceptron(S_n, \mathbf{w}_{e-1,n})$
  return $\frac{1}{N}\sum_{n\in[N]} \mathbf{w}_{E,n}$

vector according to some representation that supports averaging. For example, the optimizer state might simply be a vector $\mathbf{w} \in \mathbb{R}^D$ representing the estimate of the optimal vector directly. Or, it might be a *dual average* representation, which can be quickly mapped to some $\mathbf{w} \in \mathbb{R}^D$ according to a determinstic rule (this is the approach taken in §4). The defining characteristic of single-mixture optimization is that local rounds of stochastic training are done, until some convergence criterion is satisfied, at which point the output of the algorithm is the average of the estimates across the network. Communication does not occur here until the end of training. The generic form of the single-mixture algorithm is shown as Algorithm 4, in which, again $\mathbf{s}$ is a representation of the optimal weight vector in either the primal or dual space. We analyze the IPM and single-mixture strategies from the perspective of the consensus optimization problem, and show that the IPM strategy does yield an optimization procedure that will converge to the optimal objective value, while the single-mixture strategy, in general, does not.

Algorithm 3 Generic Iterative Parameter Mixing

procedure IPM-GENERIC(Data set shards: $(S_n)_{n\in[N]}$)
  $\mathbf{s}_0 = 0$
  for $e = 1, \cdots, E$ **do** ▷ Do E epochs of training
    $\mathbf{s}_{e,n} = OneEpochStochastic(S_n, \bar{\mathbf{s}}_{e-1})$ ▷ N threads run in parallel
    $\bar{\mathbf{s}}_e = \frac{1}{N}\sum_{n\in[N]} \mathbf{s}_{e,n}$ ▷ averaging step
  deterministically map average state $\bar{\mathbf{s}}_E$ to primal estimate $\hat{\mathbf{w}}$
  return $\hat{\mathbf{w}}$

Our analysis is an application of the Duchi et al. (2012) analysis of their **distributed dual averaging** algorithm. This algorithm allows optimization by a network of processors in a truly decentralized setting. That is, the algorithm can function even in absence of a central server, and in which messages sent by one given node only reach their immediate neighbours in the network. Duchi et al.'s (2012) analysis is

Algorithm 4 Single-Mixture Generic

procedure SM-GENERIC(Data set shards: $(S_n)_{n\in[N]}$)
  for $n = 1, \cdots N$ **do** ▷ Each $n \in [N]$ nodes runs in parallel
    $\mathbf{s}_{0,n} = 0$ for $n \in [N]$
    for $e = 1, \cdots, E$ **do** ▷ Do E epochs of training
      $\mathbf{s}_{e,n} = OneEpochStochastic(S_n, \mathbf{s}_{e-1,n})$
  deterministically map average state $\frac{1}{N}\sum_{n\in[N]} \mathbf{s}_{E,n}$ to primal estimate $\hat{\mathbf{w}}$
  return $\hat{\mathbf{w}}$

based on a connection to the mixing of Markov chains (see §5). Compared to the work of McDonald et al. (2010), our work is a generalization. Since we take an optimization perspective, and perceptron training can be cast as optimization, our includes the case of perceptron training. However, our analysis can also apply to any (potentially non-smooth) convex objective function, including the large-margin linear SVM objective, which, as we said above, can often lead to more accurate models. Compared to the work of Duchi et al. (2012), our work is an application of their framework. Duchi et al. (2012) focus on a very general set of network architectures, but do not analyze the IPM, or the single-mixture strategy, or their relation to each other, nor do they experiment with either. Our contribution is to relate the iterative parameter mixing strategy to the single-mixture strategy using the distributed dual averaging framework, and to compare the theoretical predictions to empirical observation. Our theoretical prediction is that IPM optimization will allow the network to reach a better objective value than does single-mixture optimization. This is borne out by experiments, in which IPM training leads to both better training objective values and better performance than single-mixture training. The fact that the superiority in test-set performance of IPM correlates with the ability to reach a better training objective provides a novel insight on the reason for the test-set performance difference between IPM and single-mixture perceptron training. We also show that distributed SVM training outperforms perceptron training in this distributed setting, as it often does in the sequential training setting. This motivates the desire to generalize the IPM analysis. In line with past work, we also find that IPM stochastic training out-performs the simple distribution of batch training, which like past work, suggests that distributed training strategies should include a stochastic component.

Regularization and Distributed Dual Averaging The distributed dual averaging algorithm (Duchi et al., 2012) was initially intended for the decentralized optimization

of arbitrary convex functions. We want to use this algorithm to solve the distributed *regularized* average loss problem (1.5). The use of this regularizer allows tighter convergence bounds and a simpler algorithm, which we present in §4. Also, our analysis in §4 bounds an important error quantity (in Lemma 2) which we will need in §5.

Higher-Order Features in Phrase-Structure Parsing: An Application of Iterative Parameter Mixing Our experiments with IPM training, recounted in §5 show that, for certain network sizes and certain problems, IPM training leverages the use of multiple processors to provide a speed-up in training time. As §5 shows, this speed-up is most evident when one only can or wants to wait for a single iteration of sequential training. We exploited the reduction in training time that IPM training allows in order to experiment with parsing models with **higher-order dependency features**. Higher-order dependency features are those that refer to 3- and 4- gram relationships between the word-tag pairs in the dependency parse (see §6). We show that the use of higher-order dependency features improve the out-of-domain dependency recovery of a state-of-the-art phrase-structure parsing model. To our knowledge, this is a novel contribution. We use a cube decoding parsing algorithm, which, by allowing a mix of generative and discriminative features, gives state-of-the-art performance at the expense of slower training and decoding speed. Aside from demonstrating the out-of-domain impact of higher-order dependency features, we show that a feature-bagging strategy, in which component models are trained separately and later combined, is sometimes necessary to avoid under-training and get the best performance out of large feature sets.

1.2 Contributions of this Thesis

Our contributions are as follows:

1. We provide a novel decentralized optimization algorithm, the *distributed regularized dual averaging* algorithm, which is used in subsequent chapters, and which improves over published work (Duchi et al., 2012) in three ways:

   (a) It offers improved bounds for regularized objective functions.

   (b) It offers a simpler algorithm, that does not require the user to set any step-size parameters.

   (c) It bounds the regret and cost of weight sequences needed in §5 that are not bounded in Duchi et al. (2012) (see Lemma 2 in §4 for specifics).

2. We provide a theoretical justification for the *iterative parameter mixing* distribution strategy for optimization of potentially non-smooth objective functions, like the large-margin linear SVM training objective, based on the Markov-chain mixing perspective of Duchi et al. (2012):

   (a) We show that IPM dual averaging is guaranteed to converge to the optimal objective value, while single-mixture dual averaging will not always do so.

   (b) This provides a novel explanation for the superior test-set performance of IPM, compared to single-mixture: that training set objective is better optimized using IPM.

   (c) Our analysis applies to all non-smooth objective functions, including the SVM objective, whereas previous theoretical results on IPM only applied to the perceptron.

3. We conduct extensive experiments to investigate the relationship between iterative parameter mixing and single-mixture distribution under a range of experimental settings:

   (a) First of all, our results replicate the perceptron experiments of McDonald et al. (2010), showing again that IPM out-performs single-mixture training in terms of test performance for the perceptron. However, we do so on more data sets and more tasks than the original study, and with larger variation in the number of network nodes.

   (b) We then show that IPM trained models also out-perform those trained using single-mixture in terms of test performance in the case of the SVM objective.

   (c) We show that, among models trained in a distributed fashion with IPM, those trained with the large-margin SVM objective out-perform those trained with the perceptron, thus improving upon the past perceptron-based work in this area.

   (d) Our results confirm that the training value objective is indeed better for IPM than for single-mixture optimization. This supports the hypothesis that superior test-set performance of IPM is due to better optimization of the training objective.

   (e) We show that, when only a limited amount of training time is available, IPM distributed training can yield a better model in the time given than

sequential training, confirming that multi-core optimization can be useful for training models more quickly.

4. We apply the faster training times to investigate the effect of adding higher-order dependency features to a phrase-structure parsing model:

   (a) We show that higher-order dependencies improve dependency recovery on out-of-domain tests, even for the state-of-the-art phrase-structure parsing models, a previously undocumented result.

   (b) This yields a new state-of-the-art performance, for the popular train-test set pairs compared.

   (c) For models with very many features, we show that a strategy of "feature bagging" can be necessary to avoid under-training achieve the best performance in some cases.

Chapter 2

Training of Structured Predictors

In this section we will give an overview of training methods for structured predictors. §2.1 reviews the basic setting for statistical structured prediction tasks, as well as the formalisms that we try to predict in this thesis. §s 2.2, 2.3 and 2.4 review three important supervised training methods often used in NLP: *probabilistic*, *perceptron*, and *large-margin* training. Probabilistic training does not play a major role in this thesis but is simply reviewed due to its importance, and so that it can be contrasted with the margin-based approaches presented later. Perceptron training and large-margin training will both be used extensively in the sequel.

2.1 Structured Prediction

2.1.1 The Concept of Prediction

NLP tasks are normally cast as *prediction problems* (Magerman, 1994; Charniak, 1997; Collins, 1999). In the context of machine learning and its applications, a prediction function is one which maps observed **input objects** to corresponding unobserved **output objects** (Bishop, 2006). The predicted output is meant to be compared against a **gold standard** output. Put informally, the goal of the system designer is to create a prediction function which will perform well on randomly drawn test inputs, predicting as many correct outputs as possible.

Throughout this text, whatever the specific prediction context, we will denote the space of inputs by $\mathcal{X}$ and the space of outputs by $\mathcal{Y}$. Thus, the prediction function is a function $h : \mathcal{X} \to \mathcal{Y}$. We can define a cost function $\rho : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$, which gives the cost, or loss, suffered for a misprediction. That is, $\rho(\mathbf{y}, \hat{y})$ is the cost of predicting

$h(\mathbf{x}) = \hat{y}$ when the correct answer is $\mathbf{y}$. We assume that input-output pairs are drawn according to some fixed distribution $\mathcal{D}$. We can define the **expected risk** of a given hypothesis as:

$$\mathbf{err}_{\mathcal{D}}(h) = \mathbb{E}_{(\mathbf{x},\mathbf{y})}\ell(\mathbf{y}, h(\mathbf{x})) \tag{2.1}$$

That is the expected risk is the expected loss suffered on a pair $(\mathbf{x}, \mathbf{y})$ drawn randomly according to $\mathcal{D}$. This is a central quantity of interest, since, for any application, we will be drawing some examples, for which we will want predictions, from the world, according to some $\mathcal{D}$, and we will want our hypothesis to perform as well as possible on these. So, the goal of the system designer, more formally stated, is to find a hypothesis h with as low an expected risk as possible.

The simplest example of a prediction function is **binary prediction**, or **binary classification**. In this task, whatever the input space, the output space is binary, e.g., $\mathcal{Y} = \{-1, 1\}$. The cost of a prediction $\rho(\mathbf{y}, \hat{y})$ is 0 if $\hat{y} = \mathbf{y}$, and 1 otherwise. Beyond this, we can speak of **multi-class prediction** problems. Here, for a given $\mathbf{x} \in \mathcal{X}$, the goal is to predict one outcome among a finite set of alternatives, i.e., a set isomorphic to $[K]$. For example, predicting whether a sports team will win, lose, or draw is a multi-class prediction problem, which goes beyond the binary prediction framework. NLP tasks—such as part-of-speech tagging, named entity recognition, parsing, word alignment (Manning & Schütze, 1999)—are not binary prediction problems, but instead a certain kind of multi-class prediction problem referred to as **structured prediction**, which we will discuss in detail now.

2.1.2 The Role and Nature of Training

The central philosophy of machine learning is that a prediction function should not be written directly by the designer. Instead, the designer should specify how the prediction function can be *learned* from training data. In practice this leads to much better hypotheses, and does so in a way which is more easily ported to new data sets (Bishop, 2006).

Concretely, we begin by specifying a set of hypotheses $\mathcal{H}$, from which a training procedure will select a single $h \in \mathcal{H}$. This process can be effected by *parameterizing* the prediction function, where the parameters are a vector of objects of some kind, often simply real numbers. That is, we choose some parameter space Θ, such that the choice of a particular parameter setting $\theta \in \Theta$ specifies a particular $h \in \mathcal{H}$. The job of the system designer, then, is twofold. First, he must specify the hypothesis class $\mathcal{H}$,

parameterized by the space Θ. Then, he must specify a **training procedure** that can map a set of training data, T, to a set of parameters, θ.

In the case of **supervised learning**, the training set consists of a set of **labelled examples**, or example input-output pairs, drawn from the distribution $\mathcal{D}$. That is the training set is a set of pairs $\{(\mathbf{x}_m, \mathbf{y}_m)\}_{m\in[M]}$. This training set can be used to estimate the expected risk, by a function called the **empirical risk**, which is the average loss of a hypothesis over the M examples in the training set:

$$\mathbf{err}_{\hat{\mathcal{D}}}(h) = \frac{1}{M} \sum_{m\in[M]} \ell(\mathbf{y}_m, h(\mathbf{x}_m)) \tag{2.2}$$

Thus, supervised training often consists of finding the hypothesis that minimizes empirical risk. Various guarantees exist that, in important cases, a hypothesis that minimizes empirical risk will converge to one that minimizes expected risk as the training set size grows (Vapnik, 2000; Cristianini & Shawe-Taylor, 2000; Cesa-Bianchi et al., 2004). Supervised training methods often predominate in NLP, especially for practical applications. Another possibility is **unsupervised learning**, in which case one attempts to train a prediction function, or perhaps only to estimate density functions, from only **unlabelled examples**, i.e. a set of inputs $\{\mathbf{x}_m\}_{m\in[M]}$ (Bishop, 2006). When used as a means to train prediction functions, unsupervised methods typically yield accuracies which remain well short of supervised methods for the NLP tasks listed in §2.1.3. Other possibilities, such as **semi-supervised learning**, mixes these two paradigms, attempting to learn from a mix of labelled and unlabelled data (for more on semi-supervised parser training, see §6.1.2.2).

2.1.3 The Structured Outputs Used in this Thesis

The set of structured objects predicted in this thesis are canonical examples in structured prediction. We review them here for completeness.

Part-of-Speech Tagging The *part-of-speech tagging problem*, with an example shown in Figure 2.1, is an example of the much-studied *sequence labelling task* (Manning & Schütze, 1999). Here, given a sequence of inputs taken from a finite alphabet $(\mathbf{x}_i)_{i\in[n]}$, the goal is to predict an associated sequence of labels $(\mathbf{y}_i)_{i\in[n]}$. The number of potential labellings is exponential in the length of the input, and potential interdependencies between labels mean that it is often best to predict some of the $\mathbf{y}_i$ together, rather than predicting each individually (Rabiner, 1989; McCallum et al., 2000; Lafferty et al., 2001).

EX VBD DT NN IN DT NN .
There was a cat on the mat .

Figure 2.1: A sentence tagged with its *part-of-speech*.

Phrase-Structure Parse Trees Phrase-structure parse trees correspond to left-most derivations with a **context-free grammar** (Aho & Ullman, 1972). Phrase-structure trees, with an example shown in Figure 2.2, formed the basis of a lot of early work in parsing. Here, the number of sentences for a given parse and a given context-free grammar is finite, but grows exponentially in the length of the sentence, and so efficient algorithms are necessary to search the space of possible outputs.

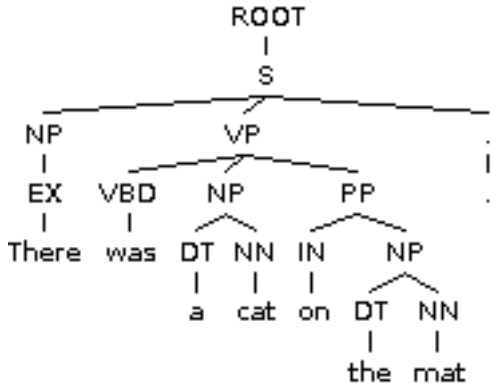


Figure 2.2: A CFG parse tree.

From a psychological point of view, phrase-structure parsing is arguably of little interest. However, general techniques for their improvement can be motivated on the grounds that techniques used in these formalisms invariably have direct analogues in the case of the better-motivated linguistic formalisms (e.g., Sarkar (2000); Hockenmaier & Steedman (2002); Riezler et al. (2002); Clark & Curran (2007); Miyao & Tsujii (2005)).

Dependency Grammar Parse Trees The **dependency grammar** approach, illustrated by an example **dependency parse** in Figure 2.3, eschews the use of phrasal nodes and phrasal categories, and instead represents directly the syntactic or semantic dependencies between the words of a sentence. It is a style of analysis with old roots (Tesnière & Fourquet, 1959), but which has seen a resurgence of interest due to fast parsing times (Nivre & Scholz, 2004; Nivre et al., 2007), allowed by the formalism, and the ability to construct dependency treebanks for lower cost than phrase-structure treebanks (Kübler et al., 2009). Once again, the number of parses for a sentence grows exponentially in the sentence length.

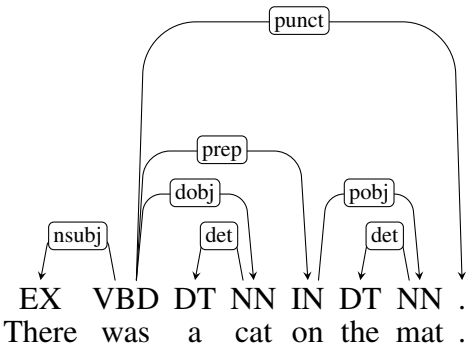


Figure 2.3: A dependency grammar parse tree.

2.1.4 Managing a Structured Search Space

In general, **structured prediction** problems, like those seen in §2.1.3, are multi-class prediction problems, in which the output space for a given input is finite, but generally too large to tractably enumerate. Usually, the number of candidates grows at least exponentially as a function of the size of the input. In the structured prediction setting, given an input space $\mathcal{X}$ and output space $\mathcal{Y}$, we assume a candidate function, $\mathcal{C} : \mathcal{X} \rightarrow P(\mathcal{Y})$, where $P(\mathcal{Y})$ is the power set of $\mathcal{Y}$. That is, $\mathcal{C}$ maps a given input to a subset of the space of all possible outputs that are applicable to the input $\mathcal{X}$. For example, in the dependency parsing problem, $\mathcal{C}$ maps a sentence to the set of all possible dependency parses for that sentence. If a sentence $\mathbf{x}$ has n words, then, for example, parses for sentences with $n' \neq n$ words are not possible parses for $\mathbf{x}$, and are filtered by the function $\mathcal{C}$.

In a structured prediction problem, the fact that the output space is too large to feasibly enumerate means that dynamic programming, pruning, or both must be used to search the output space in tractable time. Sequence labelling can be efficiently performed using the well-known *Viterbi algorithm* (Viterbi, 1967), which is a dynamic programming algorithm, that corresponds to finding the best-scoring path through a kind of directed acyclic graph (actually, a lattice), from a start state to a goal state. The original Viterbi algorithm of Viterbi (1967) is not directly applicable to the prediction of tree structures. However, tree prediction, as well as other NLP tasks, such as machine translation, can be seen as instances of the **generalized Viterbi algorithm** (Klein & Manning, 2001; Eisner et al., 2005; Huang, 2008a). The switch from the Viterbi algorithm to the generalized Viterbi algorithm corresponds to the switch from finding a path through an acyclic graph to finding a *hyper*-path through a *hyper*-graph. The key difference between the two is that a **hyper-graph** has *hyper*-edges, rather than edges.

A **hyper-edge**, $e = (h, (t_1, \cdots, t_N))$ is a pair of a head node $h \in \mathcal{V}$ and a sequence of tail nodes $t_n \in N$ for $n \in [N]$. In the case of phrase-structure parsing, for example, a hyper-edge corresponds to a rule production. For the formal details, we refer the reader to, e.g., Klein & Manning (2001). However, intuitively, a hyper-path proceeds from a set of **initial nodes** (e.g., the words in a sentence), to a single **goal node**. These paths can be given scores, and prediction amounts to selecting the highest-scoring hyper-path from the initial nodes to the goal state.

The **generalized Viterbi algorithm**, which is an exact dynamic programming algorithm, requires that each edge can be scored in isolation. Since each edge can be scored in isolation, the score of a path is simply the sums of the scores of the edges. The lack of interaction between edges means that the best-scoring path to a given node v will be the best scoring sub-path in any larger path that passes through v. Well-known parsing algorithms such the CKY algorithm for phrase-structure parsing (Kasami, 1965; Younger, 1967; Cocke & Schwartz, 1969) and *Eisner's algorithm* for dependency parsing (Eisner, 1996) are instances of the generalized Viterbi algorithm. In §6, we examine the use of the *cube decoding* algorithm, which is useful when the scores of the edges are *not* independent.

2.1.5 Ranking with Linear Hypotheses

We adopt the framework of global linear models for structured prediction. Here, the possible outputs, $\mathcal{C}(\mathbf{x})$, for input $\mathbf{x}$ are ranked according to an inner product with a real-valued parameter vector. For this, we use a feature embedding function $\Phi : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}^D$, along with a **weight vector**, $\mathbf{w} \in \mathbb{R}^D$:

$$\hat{y} = \arg\max_{y \in \mathcal{C}(\mathbf{x})} \langle \mathbf{w}, \Phi(\mathbf{x}, y) \rangle \tag{2.3}$$

In the sequel, to simplify presentation, we will assume that each output $y \in \mathcal{Y}$ can correspond to a single input $\mathbf{x} \in \mathcal{X}$.[1] Adopting this convention, writing $\Phi(\mathbf{x}, y)$ is redundant, so (2.3) can be rewritten as:

$$\hat{y} = \arg\max_{y \in \mathcal{C}(\mathbf{x})} \langle \mathbf{w}, \Phi(y) \rangle \tag{2.4}$$

For multi-class prediction problems with a small number of classes, it is possible to predict the class label using a collection of binary classifiers using a *one-versus-all* or a *one-versus-one* strategy. For k classes, the one-versus-all approach results

[1] That is, we assume a function $\text{INPUT} : \mathcal{Y} \to \mathcal{X}$ and that, for any $\mathcal{Y}$ under consideration $\text{INPUT}(y)$ is defined for each $y \in \mathcal{Y}$.

in the training of k classifiers, while the one-versus-one approach results in training $\frac{k(k-1)}{2}$ classifiers. Hsu & Lin (2002) and Rifkin & Klautau (2004) have compared these methods to those that work on the basis of a single objective formulation. The authors find that the binary methods work as well as the multi-class objective (2.13) but, at that time, trained faster. However, training methods for the multi-class objective have improved, and, for structured prediction, the number of classes is intractably large. Thus, a ranking formulation such as (2.4) is necessary.

2.2 Probabilistic Training

Much early work on parsing focused on models parameterized to maximize the **joint likelihood** of the training data. We can specify the probability of an input-output pair as a function of the parameters. In this case, the goal is to find parameters θ in some parameter space Θ that maximize the log-probability of the training set:

$$\arg\max_{\theta \in \Theta} \sum_{m \in [M]} \log p((\mathbf{x}_m, \mathbf{y}_m); \theta) \tag{2.5}$$

Early models worked on this basis (Charniak, 1997; Collins, 1997; Hockenmaier & Steedman, 2002). In such models, the probability of an input-output pair is modelled as the product of a number of sub-events of the pair. For example, in a *probabilistic context-free grammar (*PCFG*)*, each rule production is associated with a probability, and the probability of a parse is the product of the probabilities of the rule productions that appear in the parse (Jelinek et al., 1992). Typically, each sub-event is associated with a multinomial draw, whose probability is given by some element of θ (Collins, 1999).

One of the problems with maximizing joint likelihood is that it requires modelling the inputs $\mathbf{x}$. A model of the likelihood of the inputs is unnecessary for prediction, because the inputs are observed as part of the prediction task, by definition. In a sense, one can suppose that modelling conditional likelihood directly allows all variability allowed within a parameterization to focus on the discriminative aspect of the task (Minka, 2005). This concern prompted a movement in NLP to training using conditional likelihood (Collins, 2000; Johnson, 2001; Charniak & Johnson, 2005; Clark & Curran, 2007; Finkel et al., 2008). In such a training regime, one adjusts the parameters only to maximize the regularized log-probability of the outputs in the training data *given* the inputs. Where $\Psi : \Theta \to \mathbb{R}$ is a regularization function, the regularized

conditional likelihood objective is:

$$\arg\max_{\theta\in\Theta} \frac{1}{M} \sum_{m\in[M]} \log p(\mathbf{y}_m \mid \mathbf{x}_m; \theta) - \Psi(\theta) \tag{2.6}$$

Modelling of conditional likelihood with powerful, possibly overlapping features can be accomplished by the use of **conditional random fields** (CRFs), a generalization of logistic regression to graphical models (Lafferty et al., 2001; Sutton & McCallum, 2012). Such techniques allow conditional likelihood training of structured predictors. A CRF model, is parameterized by a real-valued vector, i.e. $\theta \in \mathbb{R}^D$. The conditional probability of an output, given an input, is defined as:

$$p(y \mid \mathbf{x}; \theta) = \frac{\exp\left(\langle \Phi(y), \theta \rangle\right)}{\sum_{y' \in \mathcal{C}(\mathbf{x})} \exp\left(\langle \Phi(y'), \theta \rangle\right)} \tag{2.7}$$

The normalization term in (2.7) involves, if looked at naively, a sum over $\left|\mathcal{C}(\mathbf{x})\right|$ candidates. In the case of a structured prediction task, this is, of course, intractable to compute naively. Thus, dynamic programming techniques are required to compute this sum. Such techniques are based on the forward-backward (Levinson et al., 1983) and inside-outside algorithms (Baker, 1979), which are closely related to the Viterbi algorithms.

One advantage of probabilistic training, compared to the distribution-free methods of perceptron training (§2.3) and maximum margin training (§2.4) is that a probabilistic interpretation can sometimes be useful. For example, one can straightforwardly use latent variables in an effort to maximize conditional likelihood (Quattoni et al., 2004; Koo & Collins, 2005; Petrov & Klein, 2007a). Also, certain semi-supervised learning strategies allow for a discriminative conditional likelihood model to be trained on unlabelled data using exogenous constraints (e.g., that each sentence has at least one verb, to help with part-of-speech tagging) (Bellare et al., 2009; Druck & McCallum, 2010; Ganchev et al., 2010). This family of techniques requires a probabilistic interpretation.

One drawback of probabilistic training can be the requirement that the normalization factor must be computed. Although dynamic programming algorithms can often make the computation of the normalization term tractable, there are some important cases in which the normalization factor cannot be computed. For example, the CKY algorithm requires that the feature functions used factor according to the rule productions of the parse. CRF parsers have been built that obey this constraint (Clark & Curran, 2007; Finkel et al., 2008). However, using the *cube decoding* approximate inference algorithm of Huang & Chiang (2007), Huang (2008b) has shown that better phrase-structure parsers can be made using non-local feature functions, that do not so factor

according to the rule productions of the parse. Following Huang (2008b), we will, in §6, want to make use of non-local features. In Huang & Chiang's (2007) cube decoding framework, whether for parsing or for any other structured task, it is not known how to efficiently compute the partition function (though, some work exists, Gimpel & Smith (2009); Eisner et al. (2005)), and so maximum conditional likelihood (much less maximum joint likelihood) is not currently an option when using this decoding algorithm. As we will see, training methods like the structured perceptron and structured SVM allow training to take place, even without being able to compute such a normalization factor.

2.3 Perceptron Training

Collins's (2002) **structured perceptron** algorithm generalizes the well-known binary perceptron algorithm (Rosenblatt, 1958; Block, 1962; Novikoff, 1962) to the case of **structured prediction**. This has been a very influential algorithm in NLP, continues to be arguably the most-used training algorithm, and will form an important comparison algorithm in §5. The algorithm is easy to implement, as, aside from some trivial calculations, it only requires the ability to make a prediction. This is the minimum architecture that must be built in any case for any prediction function. The structured perceptron algorithm is illustrated as Algorithm 5.

Algorithm 5 Structured Perceptron Algorithm

procedure PERCEPTRON($S = \{(\mathbf{x}_m, \mathbf{y}_m)\}_{m\in[M]}$)
  $\mathbf{w}_0 \leftarrow 0 \in \mathbb{R}^D$
  for $t \in [T]$ **do**
    draw example $(\mathbf{x}_t, \mathbf{y}_t)$ from S
    $\hat{y}_t = \arg\max_{y\in\mathcal{C}(\mathbf{x}_t)} \langle \mathbf{w}_{t-1}, \Phi(y) \rangle$
    if $\hat{y}_t \neq \mathbf{y}_t$ **then**
      $\mathbf{w}_t \leftarrow \mathbf{w}_{t-1} + [\Phi(\mathbf{y}_t) - \Phi(\hat{y}_t)]$
  return $\mathbf{w}_T$, or $\bar{\mathbf{w}}_T = \frac{1}{T}\sum_{t\in[T]} \mathbf{w}_t$

The input to the algorithm is a training set $S = \{(\mathbf{x}_m, \mathbf{y}_m)\}_{m\in[M]}$. The algorithm involves repeated prediction with a sequence of estimates, $(\mathbf{w}_t)_{t\in[T]}$ of a weight vector that separates (see below), or comes close to separating, the training data. From this training set, on each round $t \in [T]$, we draw some training example $(\mathbf{x}_t, \mathbf{y}_t)$, and attempt to predict $\mathbf{y}_t$ on the basis of $\mathbf{x}_t$. Whenever a misprediction is made, the algorithm

updates the current estimate of the weight vector by $[\Phi(\mathbf{y}_t) - \Phi(\hat{y})]$ (line 7). When training is finished, one can either return the final estimate of the weight vector $\mathbf{w}_T$, or else the average over all rounds $\bar{\mathbf{w}}_T = \frac{1}{T}\sum_{t\in[T]} \mathbf{w}_t$. Collins (2002) gives experimental evidence that it can be better to return the average of the parameter vectors over the T rounds. This is also supported by theoretical arguments about generalization guarantees (Freund & Schapire, 1999; Cesa-Bianchi et al., 2004).

Theoretical Justification As theoretical justification for this algorithm, Collins (2002) shows that convergence bounds, analogous to those of Block (1962), Novikoff (1962) and Freund & Schapire (1999) for the binary perceptron, hold for this structured training algorithm. First, we define the notion of separation by a margin:

Definition 1. *We say that a vector* $\mathbf{w} \in \mathbb{R}^D$ **separates** *the training set* S_M *by margin* γ *if for all* $m \in [M]$, $y \in \mathcal{C}(\mathbf{x}_m)$, $\langle \mathbf{w}, \Phi(\mathbf{y}_m)\rangle - \langle \mathbf{w}, \Phi(y)\rangle \geqslant \gamma$

Then, we can give the following bound on the convergence of the structured perceptron algorithm on a separable data set:

Theorem 1. *Suppose* R *bounds the norms* $\|\Phi(y) - \Phi(\mathbf{y}_m)\|$ *for each* $m \in [M]$, $y \in \mathcal{C}(\mathbf{x}_m)$, *in the training set* S, *and there exists a* $\mathbf{w}_{opt}$, $\left\|\mathbf{w}_{opt}\right\| = 1$, *that separates* S_M *by a margin of* γ. *Then, the number of mistakes made by the perceptron algorithm on repeated passes through* S_M *is at most:*

$$\left(\frac{R}{\gamma}\right)^2$$

This bound will play a role in the discussion of the distributed perceptron algorithm in §5.2.2. A variant of this bound for non-separable data sets also exists (see Collins (2002)). In practice, though the perceptron is perhaps the most widely-used training algorithm, it is not always as accurate as the large-margin alternative, to which we turn now.

Objective Value The perceptron algorithm can be viewed a sub-gradient optimization of *unregularized* average loss using the following per-example loss function (Martins et al., 2010):

$$\ell_{perceptron}(\mathbf{w}; (\mathbf{x}, \mathbf{y})) \stackrel{\Delta}{=} \max_{y\in\mathcal{C}(\mathbf{x})} \{\langle \mathbf{w}, \Phi(y)\rangle\} - \langle \mathbf{w}, \Phi(\mathbf{y})\rangle \tag{2.8}$$

A problem with this interpretation is that $\mathbf{w} = 0$ is a solution, as it achieves 0 loss on each example. But, $\mathbf{w} = 0$ is not useful as a prediction vector. And, with regularization towards 0, 0 becomes the *only* solution. Since interpretation of the perceptron

algorithm as loss minimization is somewhat deficient, we will usually speak of the perceptron in algorithmic terms, rather than as minimization of a loss function.

2.4 Maximum-Margin Training

Maximum-margin training, or *support vector machine* training, of structured predictors is of particular interest in this thesis. This method of training is shown to outperform the perceptron for distributed training in §5. And, this is the training strategy used to train all models in §6. Practical experience has shown that, for many NLP tasks, especially parsing, maximum-margin training will outperform perceptron training (McDonald et al., 2005; Tsochantaridis et al., 2005; Koo et al., 2007).

2.4.1 Motivation for Maximum-Margin Parsing

For a separable data set, there is not a single separating hyperplane, but in fact an infinite number of them. Thus, the problem of returning a separating hyperplane is ill-posed. This non-determinism is a problem because some hyperplanes are "better" than others. The generalization performance of a unit length weight vector is known to be a function of the margin, as defined in Definition 1, by which it separates the data (Valiant, 1984). This reasoning suggests that, for a separable data set, the goal should be to return the *optimal hyperplane* that correctly classifies the data by the largest possible margin (Vapnik, 2000). In the case of non-separable data, the analysis is more complex, but, intuition and practical experience suggests that better generalization performance can be gained by specifying and achieving a single hyper-plane, with the largest margin possible. Similar results to those obtained for binary prediction have been shown in the structured prediction case by Collins (2005).

Also, in a multi-class prediction setting, one answer (or, some subset of the answers) will be correct, and the rest will be wrong. However, unlike in the binary prediction setting, since there are now multiple wrong answers, it might be that certain wrong answers are better than others. In fact, in many tasks, like parsing, an output that is, e.g., 99% correct is satisfactory, even if it is "wrong" in the sense that it is not the perfectly correct answer. Thus, we will want to incorporate the cost function $\rho : \mathcal{Y} \times \mathcal{Y} \rightarrow \mathbb{R}$ into our training procedure, and use it to discourage worse answers even more strongly.

2.4.2 Objective Formulations

We now review three ways of formulating the maximum-margin training problem for structured predictors. The method discussed in §2.4.2.3 is the one that will be used in §5 and §6.

2.4.2.1 Constrained Optimization Formulation

Separable Data In the structured prediction setting, for input $\mathbf{x}$, a prediction will be chosen from candidate set $\mathcal{C}(\mathbf{x})$. Our goal will be to find the smallest weight vector that separates correct from incorrect candidates for each example by a sufficient margin.[2] Formally, this is encoded as a constrained optimization problem. For each example $(\mathbf{x}_m, \mathbf{y}_m)$, we will introduce one margin constraint per $y \in \mathcal{C}(\mathbf{x}_m), y \neq \mathbf{y}_m$ of the form:

$$\langle \mathbf{w}, \Phi(\mathbf{y}_m) - \Phi(y) \rangle \geqslant \rho(\mathbf{y}_m, y) \tag{2.9}$$

That is, for example $\mathbf{x}_m$, we require that the weight vector, $\mathbf{w}$, separates the correct prediction $\mathbf{y}_m$ from the incorrect alternative y by a margin that is at least equal to the cost, $\rho(\mathbf{y}_m, y)$, of predicting y when the gold answer is $\mathbf{y}_m$. In order to simply future presentation, we introduce the following shorthand:

$$\delta(y_1, y_2) = \Phi(y_1) - \Phi(y_2) \tag{2.10}$$

The resulting constrained optimization problem, incorporating these constraints, is then (Weston & Watkins, 1998; Crammer & Singer, 2002):

$$\arg\min_{\mathbf{w}} \frac{1}{2} \|\mathbf{w}\|_2^2 \tag{2.11}$$

$$\text{subject to } \langle \mathbf{w}, \delta(\mathbf{y}_m, y) \rangle \geqslant \rho(\mathbf{y}_m, y), \forall m \in [M], y \neq \mathbf{y}_m \in \mathcal{C}(\mathbf{x}_m) \tag{2.12}$$

If such an optimal, separating $\mathbf{w}$ exists, then the generalization results of Collins (2005) apply. However, perfect separation of the training set is not possible in practice, and so a solution to (2.12) cannot be found on most data sets. However, this formulation does serve as a useful starting point for the following, more realistic, large-margin training formulation.

[2] We want the *smallest* weight vector separating the data set by the required margin because theoretical guarantees on the generalization performance of a weight vector get better as the separating margin grows, and better as the weight vector *shrinks*.

Slack Variables To deal with inevitable cases of imperfect separability, we introduce non-negative slack variables $(\varepsilon_m)_{m\in[M]}$. These allow the margin constraints to be violated, at the cost of paying a penalty in the objective value:

$$\arg\min_{\mathbf{w}} \frac{\lambda}{2}\|\mathbf{w}\|_2^2 + \frac{1}{M}\sum_{m\in[M]}\varepsilon_m \tag{2.13}$$

$$\text{subject to } \langle \mathbf{w}, \delta(\mathbf{y}_m, y)\rangle \geqslant \rho(\mathbf{y}_m, y) - \varepsilon_m, \forall m \in [M], y \neq \mathbf{y}_m \in \mathcal{C}(\mathbf{x}_m) \tag{2.14}$$

$$\varepsilon_m \geqslant 0, \forall m \in [M] \tag{2.15}$$

In contrast to (2.12), a solution to (2.13) *can* always be found, since the slack variables can be made arbitrarily large, to allow for any sort of margin violation. However, a price must be paid in the objective value for such margin violations. The positively valued regularization parameter λ allows the user to trade off between penalizing large weight vectors relatively more than constraint violations (large λ), and penalizing constraint violations relatively more than large weight vectors (small λ).

We have said that a characteristic property of structured prediction problems is that $\mathcal{C}(\mathbf{x})$ is finite but intractable to enumerate. Thus, the requirement (2.14) actually involves an intractable number of constraints. Taskar et al. (2004) deal with this problem by reformulating the exponential number of constraints as a polynomial number of constraints. Another, perhaps more useful, strategy for dealing with the problem of satisfying an intractable number of constraints is to replace the intractably many constraints in (2.14) with a single constraint. This can be done by rewriting (2.14) as follows:

$$\langle \mathbf{w}, \delta(\mathbf{y}_m, y)\rangle \geqslant \rho(\mathbf{y}_m, y) - \varepsilon_m \tag{2.16}$$

$$\langle \mathbf{w}, \Phi(\mathbf{y}_m) - \Phi(y)\rangle \geqslant \rho(\mathbf{y}_m, y) - \varepsilon_m \tag{2.17}$$

$$\langle \mathbf{w}, \Phi(\mathbf{y}_m)\rangle - \langle \mathbf{w}, \Phi(y)\rangle \geqslant \rho(\mathbf{y}_m, y) - \varepsilon_m \tag{2.18}$$

$$\{\langle \mathbf{w}, \Phi(y)\rangle + \rho(\mathbf{y}_m, y)\} - \langle \mathbf{w}, \Phi(\mathbf{y}_m)\rangle \leqslant \varepsilon_m \tag{2.19}$$

Let:

$$\ell_{\text{SVM}}(\mathbf{w}; (\mathbf{x}, \mathbf{y})) \triangleq \max_{y\in\mathcal{C}(\mathbf{x})} \{\langle \mathbf{w}, \Phi(y)\rangle + \rho(\mathbf{y}, y)\} - \langle \mathbf{w}, \Phi(\mathbf{y})\rangle \tag{2.20}$$

Clearly, (2.19) is satisfied for all $y \in \mathcal{C}(\mathbf{x}_m)$ if and only if $\ell_{\text{SVM}}((\mathbf{x}_m, \mathbf{y}_m); \mathbf{w}) \leqslant \varepsilon_m$. Thus, we can replace the $|\mathcal{C}(\mathbf{x}_m)|$ constraints for each $\mathbf{x}_m$ with a single constraint per $\mathbf{x}_m$ as

follows:

$$\arg\min_{\mathbf{w}} \frac{\lambda}{2}\|\mathbf{w}\|_2^2 + \frac{1}{M}\sum_{m\in[M]} \varepsilon_m \tag{2.21}$$

$$\text{subject to } \ell_{\text{SVM}}(\mathbf{w};(\mathbf{x}_m,\mathbf{y}_m)) \leqslant \varepsilon_m, \forall m \in [M] \tag{2.22}$$

$$\varepsilon_m \geqslant 0, \forall m \in [M] \tag{2.23}$$

This single-constraint formulation can be much easier to work with.

2.4.2.2 Dual Formulation

Though we will not make use of this formulation directly, it is perhaps worth noting that, just as the binary SVM problem is often solved using a dual program, the structured SVM has also often been approached using a dual program. Crammer & Singer (2002); Tsochantaridis et al. (2005); Bordes et al. (2007) derive the following dual program for (2.13). First, the dual function is:

$$D(\alpha) = \sum_{m,y\neq\mathbf{y}_m} \rho(\mathbf{y}_m,y)\alpha_m^y - \frac{1}{2}\sum_{\substack{m,y\neq\mathbf{y}_m\\ m',y'\neq\mathbf{y}_{m'}}} \alpha_m^y\alpha_{m'}^{y'}\langle\delta(\mathbf{y}_m,y),\delta(\mathbf{y}_{m'},y')\rangle \tag{2.24}$$

The dual program is then:

$$\max_{\alpha} D(\alpha) \tag{2.25}$$

$$\text{subject to } \alpha_m^y \geqslant 0, \forall m \in [M], y \neq \mathbf{y}_m \in \mathcal{C}(\mathbf{x}_m) \tag{2.26}$$

$$\sum_{y\neq\mathbf{y}_m\in\mathcal{C}(\mathbf{x}_m)} \alpha_m^y = \frac{1}{\lambda} \tag{2.27}$$

One drawback, compared to the approach of unconstrained optimization presented in the next section, of optimizing in the dual space is that a lot of bookkeeping is required.

2.4.2.3 Unconstrained Optimization Formulation

It is possible to reformulate the constrained optimization problem (2.21) as an equivalent unconstrained optimization problem. Note that, at the optimal solution for $\mathbf{w}$ and $(\varepsilon_m)_{m\in[M]}$, we must have that $\ell_{\text{SVM}}(\mathbf{w};(\mathbf{x}_m,\mathbf{y}_m)) = \varepsilon_m$, otherwise a better objective value could be reached by lowering some ε_m, without violating the constraints. Thus, we can remove the ε_m altogether, to obtain the following unconstrained objective (Shalev-Shwartz et al., 2007):

$$\arg\min_{\mathbf{w}} \frac{\lambda}{2}\|\mathbf{w}\|_2^2 + \frac{1}{M}\sum_{m\in[M]} \ell_{\text{SVM}}(\mathbf{w};(\mathbf{x}_m,\mathbf{y}_m)) \tag{2.28}$$

One benefit of using this unconstrained formulation is that we can now use a broad range of convex optimization techniques (Ratliff et al. (2006); Shalev-Shwartz et al. (2011), §3). This is the version of the structured large-margin objective we will optimize using distributed optimization in §5. Like the hinge-loss function for binary predictors (Shalev-Shwartz et al., 2011), this function is non-smooth, meaning that two-sided derivatives do not exist at all points.

2.4.3 Solution Methods

A variety of ways have been explored to do optimization for structured maximum-margin training formulations. Ratliff et al. (2006) take the sub-gradient descent approach to training structured predictors. Here, analogously to the *Pegasos* algorithm for binary prediction (Shalev-Shwartz et al., 2011), the unconstrained primal objective (2.28) is optimized using sub-gradient descent. Using sub-gradient descent with the step-size schedule of Hazan et al. (2006), they can converge on an ε-accurate primal solution in time $O(\frac{1}{\lambda\varepsilon})$, ignoring logarithmic factors. The sub-gradient descent optimization strategy is discussed further in §3.3. It is interesting to note that, in contrast to some methods proposed below, the convergence of this algorithm is not a function of the training set size M.

Apart from the primal approach to structured SVM optimization, numerous dual optimization approaches also exist. In this case, since SVM objectives (2.13) and (2.25) introduce one constraint per potential output $y \in \mathcal{C}(\mathbf{x})$ per input $\mathbf{x}$, the major obstacle to optimization of a structured SVM objective is the intractably large number of constraints that are involved per input. Methods that explicitly represent the k categories, such as Crammer & Singer (2002) and Keerthi et al. (2008) are not feasible for structured prediction.

Taskar et al. (2004) devise a factorization that expresses the original exponential number of constraints as a polynomial number of marginal constraints. The number of variables per sentence in this factorization is cubic in the length of the sentence, and the number of constraints is quadratic. Although a polynomial number of constraints is, in coarse terms, considered tractable, this is still more computation than other methods have shown to be necessary, and Taskar et al. (2004) were only able to run their model on sentences of length $\leqslant 15$.

Tsochantaridis et al. (2005) approach the structured SVM problem using a cutting planes approach. Their algorithm repeatedly cycles through the examples and, for

each one, selects a maximally violated constraint according to (2.20). This is then added to a constraint set. A quadratic optimizer is repeatedly run after each addition to the constraint set. Tsochantaridis et al. (2005) show that, only a limited number of constraints need be added and that, the satisfaction of this limited number of constraints can result in a ε-accurate solution for all the (intractably many) constraints specified in the problem. Specifically, to obtain a ε-accurate solution for a training set with M examples, they show that the number of constraints added to the working set is $O(\max\left\{\frac{M}{\varepsilon}, \frac{1}{\lambda\varepsilon^2}\right\})$, which, crucially, does not depend on $\max_m |\mathcal{C}(\mathbf{x}_m)|$, nor does it introduce the polynomial number of constraints of Taskar et al. (2004). Smola et al. (2007) use bundle methods (Hiriart-Urruty & Lemaréchal, 1996) to tackle the training of a structured SVM. This involves approximating the true objective function with a sequence of approximate functions based on the pointwise supremum of a set of hyperplanes (together with a fixed regularizer). They show that the time to reach an ε-accurate solution is bounded by $O(\frac{1}{\varepsilon\lambda})$.

Bordes et al. (2007) take an online approach. Optimization is done in the dual using an SMO-like approach (Platt, 1998). On each round, two components of the dual are adjusted. An interesting feature of their approach is that, following Bordes & Bottou (2005), it alternates between fresh examples, and previously seen examples that contributed support vectors. Bordes & Bottou (2005) had earlier reported that an analogous strategy for the binary SVM optimization case worked better than using only fresh examples. This parallels the strategy in batch solutions that emphasizes the importance of focussing on those examples that contribute support vectors. They show that convergence is bounded by $O(\frac{M}{\lambda\varepsilon})$. Collins et al. (2008) explore the use of exponentiated gradient descent to solve the dual problem (2.25). The online involves block co-ordinate descent, in which the components $\alpha_m \stackrel{\Delta}{=} \left(\alpha_m^y\right)_{y\in\mathcal{C}(\mathbf{x}_m)}$ are updated at the same time. As this is an exponential number of components, the effect of this update cannot be done naively, but instead makes use of a factorization that updates only a polynomial number of components per example m, which implicitly amounts to updating α_m. This factorization requires that the predicted object factor according to a set of parts, exactly as required by the generalized Viterbi algorithm. This algorithm reaches an ε-accurate solution of the dual problem in time $O(\frac{M}{\varepsilon})$, times a factor that penalizes the distance of the ultimate solution from the starting point.

Chapter 3

Distributed Gradient-Based Optimization

The focus of §4 and §5 is on distributed optimization. In this chapter, we will give an overview of the notion of convex optimization, in general, and then discuss a variety of distributed and parallel optimization strategies.

3.1 Mathematical Background

A convex set, $C \subseteq \mathbb{R}^D$, is a set such that if $\mathbf{u}, \mathbf{v} \in C$, then the entire *line segment* joining $\mathbf{u}$ and $\mathbf{v}$ is also in C. A convex function, $f : \mathbb{R}^D \to \mathbb{R}$, is a function such that for every $\mathbf{u}, \mathbf{v} \in \mathbb{R}^D$, $0 \leqslant \alpha \leqslant 1$,

$$f(\alpha\mathbf{u} + (1-\alpha)\mathbf{v}) \leqslant \alpha f(\mathbf{u}) + (1-\alpha) f(\mathbf{v})$$

That is, the line segment from $(\mathbf{u}, f(\mathbf{u}))$ to $(\mathbf{v}, f(\mathbf{v}))$ passes never below the curve f itself. In a convex function, all local minima are global minima, and so convex functions make convenient objective functions for statistical learning. We assume a norm of interest $\|.\|$. A function g is called **λ-strongly convex** with respect to $\|.\|$ if $\lambda > 0$ and $g(v) - \frac{\lambda}{2}\|v\|^2$ is convex, where the higher λ, the more sharply curved the function. Stronger curvature usually corresponds greater regularization of an objective function, and leads to faster optimization times (§3.3). A strongly convex function will always have a unique local minimum (Rockafellar, 1970).

A vector $\mathbf{x}^*$ is called a *sub-gradient* of f at $\mathbf{x} \in \mathbb{R}^D$ if

$$f(\mathbf{y}) \geqslant f(\mathbf{x}) + \langle \mathbf{y} - \mathbf{x}, \mathbf{x}^* \rangle, \quad \forall \mathbf{y} \in \mathbb{R}^D \tag{3.1}$$

The set of all sub-gradients at $\mathbf{x}$ is called the *sub-differential* of f at $\mathbf{x}$ and is denoted $\partial f(\mathbf{x})$. A convex function is sub-differentiable everywhere.

3.2 The Convex Optimization Problem

We now briefly review the goal of convex optimization, which is to solve some form of convex optimization problem. As we said in §1, the reason for our interest in solving the convex optimization problem is to solve the regularized average loss problem, which is reiterated in §3.2.3.

3.2.1 Basic Form

Let $f : \mathbb{R}^D \to \mathbb{R}$ be a convex function and let $\mathcal{W}$ be a convex set. If f is an arbitrary convex function, then $\arg\min_{\mathbf{w}\in\mathcal{W}} f(\mathbf{w})$ will, in general, be a set of points. If f is strictly convex, then $\arg\min_{\mathbf{w}\in\mathcal{W}} f(\mathbf{w})$ will be a single point. We often use $\mathbf{w}^*$ to denote a (or, the) optimal value for any function under discussion. Then, the goal of the convex optimization task is to return a point $\hat{\mathbf{w}}$ such that $f(\hat{\mathbf{w}})$ is as close as possible to $f(\mathbf{w}^*)$. Optimization schemes are generally iterative. An iterative optimization scheme will produce a series of estimates $(\mathbf{w}_0, \mathbf{w}_1, \cdots, \mathbf{w}_T)$. The quality of a solution can be measured in terms of the *optimization cost*:

$$\varepsilon_f(\mathbf{w}_T) \stackrel{\Delta}{=} f(\mathbf{w}_T) - f(\mathbf{w}^*) \tag{3.2}$$

To be of any use, an optimization scheme must come with some guarantee that $\varepsilon_f(\mathbf{w}_T) \to 0$ as $T \to \infty$, at least with high probability or in expectation.

3.2.2 Distributed Form

When interested in optimization of a function by a network of N processors, the following form of the optimization problem is very useful. Suppose that we have a network of N nodes. Each node $n \in [N]$ has access (usually by the ability to query sub-gradients) to a local function $f_n : \mathbb{R}^D \to \mathbb{R}$. As we said in §1, the goal of distributed optimization can be cast as the solution of the **consensus optimization problem** (Nedic & Ozdaglar, 2009; Duchi et al., 2012; Boyd et al., 2011):

$$\arg\min_{\mathbf{w}\in\mathcal{W}} \frac{1}{N} \sum_{n\in[N]} f_n(\mathbf{w}) \tag{1.2}$$

That is, the goal is to find a single vector that achieves optimal average performance over the function defined over the network. This is, of course, non-trivial because the weight vector which is optimal on one node might be very different from the weight vector which is optimal on another node.

3.2.3 Regularized Average Loss

For training schemes in machine learning, the function that we want to optimize is almost always the **regularized average loss function**. A regularized average loss function is defined in terms of a convex regularization function $\Psi : \mathbb{R}^D \to \mathbb{R}$, introduced to prevent over-fitting and to keep weights from tending towards infinity, and a per-example loss function $\ell : \mathbb{R}^D \times \mathcal{Y} \to \mathbb{R}$, which measures the *loss*, or inappropriateness of vector $\mathbf{w}$ for example z in the space of examples $\mathcal{Y}$. Suppose our store of training examples is the set S. Then, the minimization of regularized average loss problem is to solve:

$$\arg\min_{\mathbf{w}} \Psi(\mathbf{w}) + \frac{1}{|S|} \sum_{z \in S} \ell(\mathbf{w}, z) \tag{3.3}$$

Many training regimes can be cast in this form, including the probabilistic training method of §2.2, and the unconstrained large-margin training method of §2.4.2.3. A popular choice of regularizer is the ℓ_2 regularization function $\Psi(\mathbf{w}) = \frac{\lambda}{2} \|\mathbf{w}\|_2^2$, which will be used in the experiments of §5 and §6.

In the distributed setting, we have a network of N nodes, and each node $n \in [N]$ has access to its own store of training examples S_n, such that the entire store of examples S is equal to the union of the disjoint sets $(S_n)_{n \in [N]}$. For simplicity, we will assume each shard S_n has the same size as any other shard.[1] Then, in order to minimize average regularized loss across all examples, the optimization problem we need to solve is:

$$\arg\min_{\mathbf{w}} \Psi(\mathbf{w}) + \frac{1}{|S|} \sum_{z \in S} \ell(\mathbf{w}, z) \tag{3.4}$$

$$= \arg\min_{\mathbf{w}} \Psi(\mathbf{w}) + \frac{1}{|S|} \sum_{n \in [N]} \sum_{z \in S_n} \ell(\mathbf{w}, z) \tag{3.5}$$

$$= \arg\min_{\mathbf{w}} \frac{1}{N} \sum_{n \in [N]} \left[\Psi(\mathbf{w}) + \frac{1}{|S_n|} \sum_{z \in S_n} \ell(\mathbf{w}, z) \right] \tag{3.6}$$

[1] If this were not true, it would be possible to weight the various parts of the objective function according to the number of examples in each, if desired.

Thus, the f_n of (1.2) is the $\Psi(\mathbf{w}) + \frac{1}{|S_n|}\sum_{z\in S_n}\ell(\mathbf{w},z)$ of (1.5). It is important to note that, since each shard S_n is different, each node has access to a potentially very different function. In this work, we will not make any assumptions about similarities between the optimal vectors for each f_n, though this would be an interesting direction for future work.

3.3 Sub-Gradient Optimization Algorithms

A standard approach to the single-core convex optimization problem is gradient descent. This basic scheme comes in many different varieties. The SVM objective of §2.4.2.3 is non-smooth, and thus gradient-descent (Bertsekas, 1999) and its variants like LBFGS (Liu & Nocedal, 1989) are not applicable. However, algorithms using *sub*-gradients are applicable. Most optimization algorithms for non-smooth functions to be discussed make use of *first-order sub-gradient oracles*. Suppose that $f : \mathbb{R}^D \to \mathbb{R}$ is a convex function. Then, we can define an associated **first-order sub-gradient oracle**, $\Omega_f : \mathbb{R}^D \to \mathbb{R}^D$, which takes as input a test point $\mathbf{w} \in \mathbb{R}^D$ and returns a sub-gradient of f at $\mathbf{w}$. In this section, we will look at two canonical sub-gradient oracle optimization schemes, but the use of such an oracle is typical, and is used in the dual averaging algorithm explored in §s 4 and 5.

3.3.1 Batch Sub-Gradient Descent

The most basic algorithm for optimizing a potentially non-smooth function is the **sub-gradient descent algorithm**, shown as Algorithm 6. The algorithm takes as input a sub-gradient oracle, Ω_f, for the function f of interest, and a step-size schedule $(\eta_t)_{t\in[T]}$. Much like the *gradient descent* algorithm for differentiable functions (Bertsekas, 1999), this algorithm operates by repeatedly querying for sub-gradients of f and then taking a step in the opposite direction according to the schedule (η_t). For an arbitrary convex function, we can choose the step-size schedule $\eta_t = \Theta(1/\sqrt{t})$ to obtain an optimization cost that shrinks at the rate $O(1/\sqrt{T})$ (Goffin, 1977; Nesterov, 2005). For a λ-strongly convex function, we can choose $\eta_t = \Theta(1/\lambda t)$ to obtain an optimization cost that shrinks at the rate $O(\ln T/\lambda T)$ (Shalev-Shwartz et al., 2007). It should be noted that, if the function is known to be differentiable (as are the probabilistic training methods of §2.2, much faster rates are possible using second-order gradient-descent methods (Bertsekas, 1999), but we focus on algorithms for non-differentiable

functions due to our interest in the non-smooth SVM function (§2.4).

Algorithm 6 Sub-Gradient Descent

procedure SUB-GRADIENT-DESCEND($\Omega_f, (\eta_t)_{t\in[T]}$)
  $\mathbf{w}_0 = 0$
  for $t = 1, \cdots, T$ **do**
    receive $g_t \in \partial f(\mathbf{w}_{t-1})$ by querying $\Omega_f(\mathbf{w}_{t-1})$
    $\mathbf{w}_t = \mathbf{w}_{t-1} - \eta_t g_t$
  return $\mathbf{w}_T$ or $\sum_{t\in[T]} \mathbf{w}_t$

3.3.2 Stochastic Sub-Gradient Descent

The sub-gradient descent algorithm is very simple, but the time required for each call to Ω_f is a problem in practice when the function being minimized is regularized average loss (3.3). This is because each computation of the gradient requires an entire pass through the data set. This produces an accurate gradient update, but takes time $\Omega(M)$ in the size of the data set, which is more costly as the data set grows. This cost turns out to be prohibitive (see §3.3.3). Fortunately, it turns out to be sufficient to work only with unbiased random vectors whose *expected values* are sub-gradient of f. Suppose that, we have some index set $\mathcal{Z}$ and some indexed function $f_{\mathcal{Z}} : \mathbb{R}^D \times \mathcal{Z} \to \mathbb{R}$, and associated oracle function $\Omega_{f_{\mathcal{Z}}} : \mathbb{R}^D \times \mathcal{Z} \to \mathbb{R}^D$. Suppose we can draw indices $z \in \mathcal{Z}$ according to some distribution Z, and $\mathbb{E}_z\left[\Omega_{f_{\mathcal{Z}}}(\mathbf{w}, z)\right] \in \partial f(\mathbf{w})$ for any $\mathbf{w}$, and the $\Omega_{f_{\mathcal{Z}}}(\mathbf{w}, z)$ have bounded size. Then, with an appropriate step-size schedule, a sub-gradient descent algorithm using only the ability to query $\Omega_{f_{\mathcal{Z}}}$ will converge to the optimal value of f. The interesting thing to note is that we do not need to query sub-gradients of f directly. A useful example of this scheme is when the function f we want to minimize is average loss (3.3), and we select an example at random, and compute the sub-gradient for the loss on that example alone. Here, the indices for the function are simply the examples in the training set. This is an unbiased estimate of the sub-gradient of the average loss on the entire data set, but can be computed quickly, in time that is $O(1)$ with respect to the data set size.

One such convergent stochastic algorithm is the **stochastic sub-gradient descent** algorithm in Algorithm 7. This function minimizes a function f, and its input consists of an oracle $\Omega_{f_{\mathcal{Z}}}$ for the indexed function $f_{\mathcal{Z}}$, a step-size schedule $(\eta_t)_{t\in[T]}$, and a distribution Z. For an arbitrary convex function, we can choose the step-size schedule $\eta_t = \Theta(1/\sqrt{t})$ to obtain an optimization cost that shrinks at the rate $O(1/\sqrt{T})$ (Zinke-

vich, 2003). For a λ-strongly convex function, we can choose $\eta_t = \Theta(1/\lambda t)$ to obtain an optimization cost that shrinks at the rate $O(\ln T/\lambda T)$ Hazan et al. (2006). A variety of schemes have been suggested to set the step-size automatically, such as the *passive aggressive* algorithm of Crammer et al. (2006).

Algorithm 7 Stochastic Sub-Gradient Descent

procedure STOCHASTIC-SUB-GRADIENT-DESCEND($\Omega_{f_Z}, (\eta_t)_{t\in[T]}, Z$)
  $\mathbf{w}_0 = 0$
  for $t = 1, \cdots, T$ **do**
    draw an index z according to distribution Z
    receive $g_t \in \partial f_Z(\mathbf{w}_{t-1}, z)$ by querying $\Omega_{f_Z}(\mathbf{w}_{t-1}, z)$
    $\mathbf{w}_t = \mathbf{w}_{t-1} - \eta_t g_t$
  return $\mathbf{w}_T$ or $\sum_{t\in[T]} \mathbf{w}_t$

3.3.3 Stochastic Optimization for Large Data Sets

It is interesting to note that, with respect to sub-gradient descent variants, the number of calls required to be made to a *stochastic* oracle are, asymptotically, of the same number of calls that must be made to a *batch* oracle. However, as noted, for any regularized average loss objective, while the time taken by the full oracle Ω_f is $\Theta(M)$, the stochastic oracle only needs to look at a single example, and therefore runs in $\frac{1}{M}$ the time of the full oracle. Thus, it would seem that the stochastic approach should lead to much faster training, in the case of a non-smooth function. In fact, the advantage grows as the data set size grows. The situation is slightly different for smooth objective functions, but, still, several theoretical and empirical analyses in the general field of machine learning show that stochastic algorithms are theoretical expected to be, and empirically in fact, faster learners than their batch counterparts, even for smooth functions (Bottou, 2004; Bottou & Le Cun, 2005; Bottou & Bousquet, 2008; Shalev-Shwartz & Srebro, 2008). Stochastic algorithms have also come to dominate for many NLP applications (Collins, 2002; McDonald et al., 2005), and in a distributed setting (Hall et al., 2010). For this reason, there is a focus on stochastic algorithms in §4 and §5.

3.4 Parallel Optimization Schemes

In this section, we review some schemes for multi-core optimization on a shared memory system. The focus of this thesis is on optimization for distributed network archi-

tectures, for which past work is reviewed in the next section. But, we include this information about shared memory parallel schemes for completeness.

Synchronous Methods Synchronous shared memory methods are characterized by the need for all threads to co-ordinate after one batch of work, before beginning a new batch. One natural synchronous parallel memory method is to do optimization in mini-batches. Mini-batch optimization sits between the two poles of stochastic and batch optimization, and involves drawing k examples, with $1 < k \ll M$, at a time from which to compute gradient updates. Mini-batch updates should be more accurate than single-example stochastic updates, since each gradient update is based on a greater number of examples, but the updates happen less frequently, and can be parallelized easily on a shared-memory architecture using a master-workers architecture. With N processor cores, and a mini-batch of size k, each core can be sent $\frac{k}{N}$ examples. The workers then communicate their gradients back to the master, who incorporates these into the weight vector. Such an architecture requires that each worker finish one batch before the next batch can begin. The synchronous mini-batch approach has been used in NLP by Finkel et al. (2008) in the context of training a CRF parsing model, which otherwise would have been impractical to train. And, mini-batches have been used by Zhao & Huang (2013) in the context of the perceptron. Takác et al. (2013) study mini-batches for SSGD in the case of the binary SVM optimization problem. They show that speed-ups can be achieved, but that such improvements are a function of the differences among the data points: greater efficiency improvements are found when the training examples in the training set (and thus in each randomly drawn batch) are more different, because the batches are less redundant.

Asynchronous Methods The synchronous mini-batch strategy requires that each worker finishes a given round before any worker can begin the next round. In practice, if the time required to process the examples sent to the various workers is unevenly balanced, this can mean that certain worker will often be idle, in the time between when it finishes its mini-batch, and when the last worker does. Asynchronous strategies avoid this problem by eschewing the requirement that workers wait for others to finish their batch. One approach to the problem uses locking (Nedić et al., 2001; Langford et al., 2009). Here, a worker can obtain a mini-batch of examples, compute the gradient over those examples, and then acquire a lock from the central server to update the weight vector, without co-ordinating with other workers. In this set-up, each worker operates with a weight vector is stale. That is, on round t worker $n \in [N]$ has access to a work-

ing weight vector that only incorporates the first $t - N$ examples. Nedić et al. (2001) show that such an algorithm will eventually converge to the true solution. And, where α is a number that quantifies the correlation of the gradients, Langford et al. (2009) show that the use of stale vectors can be shown to require no more than αN number of examples. Thus, if the gradients have low correlation, the use of stale gradients has less harm. Gimpel et al. (2010) adapt this strategy to allow asynchronous training with mini-batches. Here, each worker draws a mini-batch of k examples, computes the gradient over these, and then communicates with the central server. This communication requires the acquisition of a lock, at which time the worker updates the central weight vector, and makes a copy of the latest version of the weight vector. Gimpel et al. (2010) show empirically that asynchronous updates out-perform synchronous updates, and scale better with the number of cores. It should also be noted that, for many applications, the gradients computed for any given example are sparse. That is, compared to the number of components of the weight vector, the gradient will have only a very small number of non-zero components. Niu et al.'s (2011) *HogWild* algorithm exploits this fact to show that, if there is suitable sparsity among the gradients, it can be practical to avoid locking the weight vector at all. Here, N worker cores can update the same weight vector, without any locking or thread-safe mechanisms, allowing for near linear speed-ups in both theory and practice, if the gradient updates are suitably sparse.

3.5 Distributed Optimization Schemes

A main focus of this thesis is on the distinction between the iterative parameter mixing and single-mixture approaches to the distributed consensus optimization problem (1.2). Here, we review some alternative approaches to solving the distributed optimization problem.

Sub-Sampling A trivial solution to the problem of having lots of data spread across many computers is simply to discard some of it. In other words,rather than optimize $\frac{1}{N}\sum_{n\in[N]} f_n(\mathbf{w})$, we can simply choose some $i \in [N]$ and minimize $f_i(\mathbf{w})$ on a single core. This amounts to a local optimization problem, for which many algorithms exist. Such an approach would seem reasonable if there were great redundancy in the functions f_n, perhaps because the data are all very similar. It would also be reasonable if there are very few parameters to tune. However, practical experience has shown that, for many tasks, simply ignoring data is not a useful solution. McDonald et al. (2010)

and Agarwal et al. (2011) both report that using only a single part of the data consistently performs worse using all of it. In terms of parsing, Steedman et al. (2003b) show that the performance of the Collins (1999) parser on the WSJ levels off after seeing about 60% of the WSJ training data. Petrov (2009) shows that the performance of his parser on the WSJ levels off after seeing about 70% of the WSJ training data. Thus, limiting one's attention a smaller fraction of the data than this would lead to under-performing models.

Distributed Gradient Computation Using a batch optimization strategy, like the sub-gradient descent of Algorithm 6, computation of the gradient can be parallelized without changing the computation. As we said, computing a sub-gradient of the average loss function for any test point takes time $\Omega(M)$, linear in the number of examples. The same gradient can be computed in roughly $\frac{1}{N}$ the time using N computer cores. And, since the exact same result is computed, the same number of iterations is required as with a sequential algorithm, leading to perfectly linear speed-up compared to the original batch algorithm. Indeed, Chu et al. (2007) do exactly this. However, batch algorithms are usually slower than stochastic algorithms (see §3.3.2) to reach a level of test-set performance, so a linear improvement over batch training does not constitute a linear improvement over the state-of-the-art. Distributed gradient approaches, when tested, have been found to lag behind those that use distributed stochastic training. Hall et al. (2010) have investigated the use of distributed gradient computation and report that it performs worse and takes longer than both single-core SGD and SGD with iterative parameter mixing. However, hybrid approaches,like Agarwal et al. (2011), discussed in later in this section, that mix stochastic and batch training are interesting.

Majority Vote Method We now mention a distributed training strategy which is not, strictly speaking, an optimization algorithm. Nevertheless, since we are interested in optimization, here, primarily as a means of training a predictor, we should mention the majority vote method. The idea here is to avoid the problem of distributed optimization, and simply train N predictors, each on the local set of examples S_n. Each predictor can be trained in parallel on $\frac{1}{N}$ of the data, perhaps in less time than would be needed to train on all the data. Then, at inference time, we can use a vote amongst the N predictors to arrive at a prediction. This strategy is reminiscent of the voted perceptron strategy of Freund & Schapire (1999), who advocate voting on noisy or unseparable data. One problem with this strategy is that it requires N decoding steps at test-time, which will be expensive for structured prediction models. Indeed, it is non-trivial to

even combine the output of structured prediction models. In any event, Mann et al. (2009) have reported that voting was not a high-performing distribution strategy.

Single-Mixture Distribution Several facts are known about what we, in §1, called the *single-mixture* communication method. This strategy comes with performance guarantees in the case of maximum entropy objectives. Suppose we have N nodes with M examples each. Mann et al. (2009) compare a single node training on NM examples, to N nodes training on M examples each. In their analysis, in both cases the weight vector will converge to the optimal weight vector at the speed $O(1/(\lambda\sqrt{NM})$ in both cases. Zinkevich et al. (2010) study the post-hoc mixture method for SGD. Assuming each machine has access to all of the data, they show empirically that a speed-up can be achieved with multiple cores. From a statistical learning perspective, Zhang et al. (2012) show that, if the objective function being optimized is suitably smooth, then, the empirical risk of a post-hoc mixture of properly optimized sub-problems will approach the expected risk, or population risk, at a rate $O(\frac{1}{NM} + \frac{1}{N^2})$. If the number of machines, N, is less than the number of examples per machine, M, then this rate of convergence to the population risk is optimal, i.e. as fast as could be achieved with all examples on a single machine.

Alternating Directions Method of Multipliers Another interesting distributed optimization approach, which has gained popularity lately, is *alternating directions method of multipliers* (Boyd et al., 2011), is to rewrite (1.2) as the following constrained optimization problem:

$$\arg\min_{\mathbf{w}_1,\ldots,\mathbf{w}_N,\mathbf{z}} \frac{1}{N} \sum_{n\in[N]} f_n(\mathbf{w}_n) \tag{3.7}$$

$$\text{such that } \mathbf{w}_n = \mathbf{z}, \forall n \in [N] \tag{3.8}$$

Here, we have introduced the auxiliary variable $\mathbf{z} \in \mathbb{R}^D$, which technically allows the problem to be elegantly stated, and which effectively represents the desired solution to the problem. Using an *augmented Lagrangian* approach (Boyd et al., 2011), this problem can be solved by the following distributed scheme. Let $\bar{\mathbf{w}}^t = \frac{1}{N}\sum_{n\in[N]} \mathbf{w}_n^t$, $\rho > 0$ be a constant set by tuning, and:

$$\mathbf{w}_n^{t+1} = \arg\min_{\mathbf{w}_n} \left[f_n(\mathbf{w}_n) + \mathbf{y}_n^t \left(\mathbf{w}_n - \bar{\mathbf{w}}^t\right) + \frac{\rho}{2} \left\|\mathbf{w}_n - \bar{\mathbf{w}}^t\right\|_2^2 \right] \tag{3.9}$$

$$\mathbf{y}_n^{t+1} = \mathbf{y}_n^t + \rho \left(\mathbf{w}_n^{t+1} - \bar{\mathbf{w}}^{t+1}\right) \tag{3.10}$$

Here, each (3.9) step is solved in parallel by the N worker cores. The resulting weight

vectors must be averaged by a central server, which can also handle the very low-cost computation (3.10), and the vectors $\mathbf{y}_n^t$ and $\bar{\mathbf{w}}$ must be communicated to each worker at the beginning of each round. The interesting feature of this approach is that the dual and auxiliary vectors serve to change the optimization problem solved by each worker to increasingly bias it towards consensus with the rest of the group. Preliminary experiments with this strategy conducted by the author, however, did not show significant difference in performance to the IPM optimization algorithm discussed in §5.

Hybrid Batch and Stochastic Agarwal et al. (2011) use a combination of stochastic and batch learning. Each node has its own local store of data. The first pass over the data uses SGD, while remaining passes implement LBFGS. Communication takes place using the *All-Reduce* algorithm, discussed in §3.6, the importance of which was first stressed in this article. Empirically, they show superior performance using this mix of stochastic and batch learning to either fully stochastic or fully batch distributed optimization.

Distributed Mini-Batch Dekel et al. (2012) look at the case of streaming data. That is, each node in the network can always draw a fresh example from a given underlying distribution. Suppose that we have M examples split over N machines. For arbitrary convex functions, the optimal *regret* (difference between loss suffered during the run of a training algorithm and that loss which would have been suffered by the optimal test vector, see §4.2.2.1) suffered by a single processor processing the M examples in sequence is $\Omega(\sqrt{M})$ (Nemirovski & Yudin, 1983; Cesa-Bianchi & Lugosi, 2006; Abernethy et al., 2009). Dekel et al. (2012) demonstrate that, using a distributed mini-batch approach, they can achieve $O(\sqrt{M})$ regret even though the processing is distributed over N nodes, so that each node only sees $\frac{M}{N}$ of the examples. This work is notable in that it clearly demonstrates a benefit to using multiple processors. However, this algorithm relies on the ability to always select new examples from the target distribution. This holds in practice on infinite streams of identical data, and also when all data fits on a single hard drive. The assumption that all new examples on each machine are drawn from the same machine does not hold when repeated passes are made over data held on a distributed network of machines, since each machine has access only to its own set of examples.

Asynchronous SGD Dean et al. (2012) explore the use of asynchronous stochastic

methods in a distributed network for the training of deep belief networks, using the *GraphLab* distributed training framework (Low et al., 2010). They do not give theoretical guarantees. However, in contrast the results of Hall et al. (2010), they find that asynchronous methods worked well on their network, and allow them to reach a higher accuracy than single-core SGD in the time allowed.

3.6 Distributed Network Interfaces

While interfaces for shared memory architectures are rather standard—we assume a number of processing cores, with a single memory, to which access can be regulated using concurrency constructs such as locks (mutual exclusion mechanisms)—systems for co-ordinating distributed networks can display a wider range of variability, in terms of the software interfaces available to the programmer to co-ordinate distributed computation. We briefly review some of these here to give an idea of how the IPM and single-mixture optimization algorithms of §1 could be implemented on a distributed system in practice.

MPI, which stands for *message passing interface*, is a set of protocols for communicating between nodes in a distributed environment (Gropp et al., 1999). This framework is very general and powerful, but, since its inception, more higher-level constructs have become popular, since large classes of distributed programs follow specific patterns, which are not all reflected in this rather low-level interface.

Map-Reduce (Dean & Ghemawat, 2008) was an early popular framework for distributed computation that provides the user with two functions from a common functional programming interface. Simplifying somewhat, the *map* function maps objects of type A to those of type B, and the reduce function collates objects of type B. A data set of very many objects of type A can be split into N smaller shards of objects of type A, so that each *map* task can run in parallel. Then, the results created by these map tasks can be reduced in parallel, to output a single object of type B. A great many programs have such a structure. For example, one might want to count the number of times a word w occurs in a corpus. Here, the *map* task can map each shard S_n of the corpus to the number of times w occurs in S_n, and the *reduce* task can simply sum up the counts produced by the first task. *Hadoop* (White, 2009) is a popular open-source implementation of Map-Reduce. One problem with Map-Reduce for machine learning applications is that many machine learning problems involve repeated passes through the data, while the design of Map-Reduce (Dean & Ghemawat, 2008) seems primarily

focused towards a single pass. Input to a *map* job must be read from disk. Each input to a *reduce* job must be written to and then read from disk. And, a lot of overhead is invested in providing fault-tolerance. In the special case in which the *reduce* job is simply a summing or an averaging of vectors, as is the case for the IPM optimization algorithms of §1, it is possible to greatly speed-up the process, as shown by Agarwal et al. (2011). They provide an *All-Reduce* framework, in which the network of processors is connected using a spanning tree. Messages work their way from the leaves of the tree to the root, which sums up the vector in memory, and result is sent back to each node through the tree. This avoids any stage involving writing writing vectors to disk. Agarwal et al. (2011) show that this can lead to speed-ups for iterative gradient-based algorithms of between 3 and 20 times. At the moment, it seems All-Reduce can be the recommended way of implementing the IPM algorithms in a distributed network. In addition, further similar alternatives exist, such as the *Spark* framework (Zaharia et al., 2012), which allows memory read from disk to persist in main memory for faster re-use, and the *GraphLab* framework (Low et al., 2010), which allows better for asynchronous jobs.

Chapter 4

Distributed Regularized Dual Averaging

This chapter describes a novel distributed optimization algorithm called the *distributed regularized dual averaging* (DRDA) algorithm. This algorithm will be used in the analysis of the *iterative parameter mixing* and *single-mixture* distribution strategies of §5. This new algorithm is a novel extension of the *distributed dual averaging* algorithm of Duchi et al. (2012). The sequential dual averaging algorithm (Nesterov, 2009; Xiao, 2010) is a stochastic optimization algorithm that has been shown to perform well when using $L1$ regularization, and has formed the basis for the popular *AdaGrad* algorithm (Duchi et al., 2011a), which sets different step-sizes for different features, depending on the number of prior updates that have been made for that feature. The distributed dual averaging algorithm (Duchi et al., 2012) applies dual averaging in a decentralized network setting.

Like the distributed dual averaging algorithm of Duchi et al. (2012), the DRDA algorithm presented here provides a provably convergent stochastic optimization framework for a stochastic version of the distributed consensus optimization problem (1.2). This version incorporates results from Xiao (2010) to allow better better handling of a strongly convex regularization term, which is useful for the online, stochastic optimization of a regularized average loss problem (3.6). This work improves over that of Duchi et al. (2011b, 2012) in three ways. First, it gives a stronger regret bound in the case of strongly regularized stochastic functions. Second, it provides a simplified algorithm, that avoids the need to set step-size parameters. Third, we show how to bound the regret of *local sequences* of parameter estimates, that are not discussed in past work with composite objectives (Duchi et al., 2011b). These improvements are

reiterated in §4.3.3, where they can be better explained when the terminology of this chapter has been introduced.

4.1 Stochastic and Online Optimization

In this section, we define the paradigms used for sequential and distributed optimization later in the chapter.

4.1.1 Sequential Online Optimization

Assume we are given some indexed convex loss function $\ell : \mathbb{R}^n \times \mathcal{Z} \to \mathbb{R}$, such that $\ell(.,z) : \mathbb{R}^n \to \mathbb{R}$ is a convex function for each index $z \in \mathcal{Z}$. Assume we are also given a regularization function $\Psi : \mathbb{R}^n \to \mathbb{R}$, which is λ-strongly convex on $\mathcal{W}$ with respect to some norm of interest $\|.\|$. Finally, assume we are able to draw indices z according to some fixed but unknown distribution Z. From these components, we create, $\theta_{\ell,Z,\Psi} : \mathbb{R}^n \to \mathbb{R}$, a regularized stochastic convex function defined as:

$$\theta_{\ell,Z,\Psi}(\mathbf{w}) \stackrel{\Delta}{=} \mathbb{E}_z\left[\ell(\mathbf{w};z)\right] + \Psi(\mathbf{w}) \tag{4.1}$$

That is, $\theta_{\ell,Z,\Psi}(\mathbf{w})$ is the regularized expected loss for weight $\mathbf{w}$, where the expectation is taken with respect to drawing indices z according to the distribution Z. Note that, by choosing the uniform distribution over a set of M examples, we simply recover the regularized average loss problem (3.3). We sometimes omit the subscripts on θ, in which case some sub-scripted arguments should be clear from the context. The associated minimization task is:

$$\arg\min_{\mathbf{w}\in\mathcal{W}} \left\{\theta_{\ell,Z,\Psi}(\mathbf{w})\right\} \tag{4.2}$$

To minimize a function of the form (4.1), we adopt the approach of sampling. That is, we will sample a series of indices $\mathbf{z}_T \stackrel{\Delta}{=} (z_t)_{t\in[T]}$ according to Z, creating a function

$$\sum_{t\in[T]} \left[\ell(\mathbf{w},z_t) + \Psi(\mathbf{w})\right], \tag{4.3}$$

which will be optimized in an online fashion, meaning that, for each round $t \in [T]$, we produce a test weight $\mathbf{w}_{t+1}$, which is a function only of the indices $(z_\tau)_{\tau\in[t]}$.

4.1.2 Decentralized Online Optimization

In the sequential online optimization scenario, all examples are available to a single processing node. But, as we said (§1), to leverage multiple cores and train on data stored in a distributed fashion, we want to look at distributed optimization, and the consensus optimization problem (1.2). The regularized stochastic version of the distributed optimization problem is similar to (1.2). Assume we are given an indexed convex loss function $\ell : \mathbb{R}^n \times \mathcal{Z} \to \mathbb{R}$, a fixed λ-strongly convex regularizer Ψ and N distributions over indices $(Z_n)_{n\in[N]}$. As in the sequential case above, by letting Z_n, for each $n \in [N]$ be the uniform distribution over the set S_n of examples, we recover the regularized average loss problem (3.6) that we are interested in. From these, we can create N regularized stochastic objectives $\theta_{\ell,Z_n,\Psi}(\mathbf{w})$, for $n \in [N]$, each of which represents the part of the problem seen by the n'th node of the network.[1] From this, we can create a composite function:

$$\Theta_{\ell,(Z_n)_{n\in[N]},\Psi}(\mathbf{w}) = \frac{1}{N}\sum_{n\in[N]} \theta_{\ell,Z_n,\Psi}(\mathbf{w}) \tag{4.4}$$

Again, the subscripts of Θ will sometimes be suppressed where possible. The associated optimization task is then:

$$\arg\min_{\mathbf{w}\in\mathcal{W}} \left\{\Theta_{\ell,(Z_n)_{n\in[N]},\Psi}(\mathbf{w})\right\} \tag{4.5}$$

We again adopt the strategy of sampling to approach the distributed stochastic problem (4.5). This time, each of the N nodes will sample T examples. That is, we will sample a sequence of sequences of indices $\mathbf{z}_T^N \triangleq \left((z_t^n)_{t\in[T]}\right)_{n\in[N]}$, creating the function

$$\frac{1}{N}\sum_{t\in[T]}\sum_{n\in[N]} \left[\ell(\mathbf{w}, z_t^n) + \Psi(\mathbf{w})\right], \tag{4.6}$$

which will be optimized in an online fashion, so that on each round $t \in [T]$, we produce a test weight $\mathbf{w}_{t+1}$, which is a function only of the indices $\left((z_\tau)_{\tau\in[t]}\right)_{n\in[N]}$. We underline that node n has access directly only to the indices $(z_t^n)_{t\in[T]}$ drawn from Z_n. In the machine learning setting, this corresponds to the fact that node n has access only to its own supply of examples.

[1] We could let the loss function ℓ vary according to n as well, although such a presentation seems less natural to us. In any event, the restriction to a single ℓ is with loss of generality, since, apart from the requirement that $\ell(.,z)$ be convex, ℓ is an arbitrary function z anyway.

4.1.3 Assumptions

That Ψ is a λ-strongly convex function is stated in the problem definition. We assume that $\arg\min_{\mathbf{w}\in\mathcal{W}}\Psi(\mathbf{w}) \in \mathcal{W}$. Then, we can assume that $\min_{\mathbf{w}\in\mathcal{W}}\Psi(\mathbf{w}) = 0$ and that $\arg\min_{\mathbf{w}\in\mathcal{W}}\Psi(\mathbf{w}) = 0$ without loss of generality, since we can simply translate the coordinates of the problem so that this is so. We also assume that $\ell(.,z)$ is G-Lipschitz continuous on $\mathcal{W}$ for each $z \in \mathcal{Z}$. That is, there exists some $G \in \mathbb{R}$ such that:

$$|\ell(\mathbf{w}_2,z) - \ell(\mathbf{w}_1,z)| \leqslant G\|\mathbf{w}_2 - \mathbf{w}_1\|, \forall \mathbf{w}_1,\mathbf{w}_2 \in \mathcal{W}, z \in \mathcal{Z} \tag{4.7}$$

A norm $\|.\|$ induces a dual norm, $\|\mathbf{g}\|_* = \max_{\|\mathbf{w}\|=1} |\langle \mathbf{w},\mathbf{g}\rangle|$. The dual norm can be used to measure the sizes of sub-gradients. (4.7) implies that sub-gradients are bounded over the domain of interest (Duchi et al., 2012). That is, that there exists some $G \in \mathbb{R}$ such that:

$$\mathbf{g} \in \partial\ell(\mathbf{w},z) \rightarrow \|\mathbf{g}\|_* \leqslant G, \forall \mathbf{w} \in \mathcal{W}, z \in \mathcal{Z} \tag{4.8}$$

This assumption holds, for example, on an unbounded domain if $\ell(.,z)$ is polyhedral for each z, and also holds for arbitrary convex functions if $\mathcal{W}$ is a compact set (Duchi et al., 2012).

4.2 Sequential Regularized Dual Averaging

We begin by reviewing the sequential regularized dual averaging algorithm and quote some results about it (Nesterov, 2009; Xiao, 2010).

4.2.1 Sequential Dual Averaging Algorithm

The regularized dual averaging algorithm (Nesterov, 2009; Xiao, 2010) is depicted in Algorithm 8. The goal of the this algorithm is to solve a problem of the form (4.2). Corresponding to the definition of $\theta_{\ell,Z,\Psi}$, the input to the optimization algorithm is threefold. The first input is an *oracle function* $\Omega_{\partial\ell} : \mathbb{R}^n \times \mathcal{Z} \rightarrow \mathbb{R}^n$, of the kind discussed in §3.3. The second input is $\Psi : \mathbb{R}^n \rightarrow \mathbb{R}$, a λ-strongly convex regularization function. And, third input is a concrete sequence of examples $\mathbf{z}_T = (z_t)_{t\in[T]}$, drawn according to distribution Z, with each $z_t \in \mathcal{Z}$.

The state maintained between rounds is the called *dual average*. The dual average on round $t \in [T]$ is denoted $\mathbf{d}_t$, and constitutes a sum (implicitly, an *average*) of the

Algorithm 8 Regularized Stochastic Dual Averaging

1: **procedure** RSDA-MINIMIZE($\Omega_{\partial\ell}, \Psi, \mathbf{z}_T$)
2: $\mathbf{d}_0 \leftarrow 0 \in \mathbb{R}^n$
3: $\mathbf{w}_1 \leftarrow \arg\min_{\mathbf{w}\in\mathcal{W}} \Psi(\mathbf{w})$
4: **for** $t \in [T]$ **do**
5: receive $\mathbf{g}_t \in \partial\ell(\mathbf{w}_t, z_t)$ by calling $\Omega_{\partial\ell}(\mathbf{w}_t, z_t)$
6: $\mathbf{d}_t \leftarrow \mathbf{d}_{t-1} + \mathbf{g}_t$
7: $\mathbf{w}_{t+1} \leftarrow \arg\min_{\mathbf{w}\in\mathcal{W}} \{\langle \mathbf{d}_t, \mathbf{w}\rangle + t\Psi(\mathbf{w})\}$
8: **return** $\bar{\mathbf{w}} = \frac{1}{T}\sum_{t\in[T]} \mathbf{w}_t$

sub-gradients, $\mathbf{g}_\tau$ for $\tau \in [t]$. That is, for $t \geqslant 1$:

$$\mathbf{d}_t = \mathbf{d}_{t-1} + \mathbf{g}_t = \sum_{\tau\in[t]} \mathbf{g}_\tau \tag{4.9}$$

The test points $\mathbf{w}_t$ are not related directly to one another as in stochastic gradient descent, but instead are obtained by projecting the dual averages into the primal space. The projection step takes place on line 7 of the algorithm. For future reference, let:

$$\Pi_{\Psi,t}(\mathbf{d}) \triangleq \arg\min_{\mathbf{w}\in\mathcal{W}} \{\langle \mathbf{d}, \mathbf{w}\rangle + t\Psi(\mathbf{w})\} \tag{4.10}$$

Using this notation, the projection steps amount to:

$$\mathbf{w}_{t+1} \leftarrow \Pi_{\Psi,t}(\mathbf{d}_t), t \geqslant 1 \tag{4.11}$$

$$\mathbf{w}_1 \leftarrow \Pi_{\Psi,1}(\mathbf{d}_0) \tag{4.12}$$

The test points are elements of the *primal space*, the domain of the original problem. The "dual average" is so called because sub-gradients, conceptually, are elements of the *dual* space (i.e., the space of linear mappings into the primal space). The output of the program given indices $\mathbf{z}_T$ is $\bar{\mathbf{w}}_T = \frac{1}{T}\sum_{t\in[T]} \mathbf{w}_t$, the average test point over all iterations.

4.2.2 Sequential Dual Averaging Analysis

4.2.2.1 Sequential Regret and Cost

The *regret* of a sequence of test vectors $(\mathbf{v}_t)_{t\in[T]}$, with respect to a stochastic function being optimized θ and a sequence of indices $\mathbf{z}_T$, is denoted $R_{\mathbf{z}_T,\theta}((\mathbf{v}_t)_{t\in[T]})$, and is

defined as:

$$R_{\mathbf{z}_T,\theta}((\mathbf{v}_t)_{t\in[T]}) \stackrel{\Delta}{=} \left[\sum_{t\in[T]} \left[\ell(\mathbf{v}_t, z_t) + \Psi(\mathbf{v}_t)\right]\right] - \left[\sum_{t\in[T]} \left[\ell(\mathbf{w}^*, z_t) + \Psi(\mathbf{w}^*)\right]\right] \quad (4.13)$$

That is, the regret is the difference between the online loss that was suffered by the sequence $(\mathbf{v}_t)_{t\in[T]}$, and that which would be suffered by the optimal vector $\mathbf{w}^* = \arg\min_{\mathbf{w}\in\mathcal{W}}\{\theta(\mathbf{w})\}$, on the observed sequence of examples $\mathbf{z}_T$. We are primarily primarily interested in $R_{\mathbf{z}_T,\theta}((\mathbf{w}_t)_{t\in[T]})$, the regret of the sequence $(\mathbf{w}_t)_{t\in[T]}$ of test points produced by Algorithm 8 when run on the sequence $\mathbf{z}_T$.

The quality of a proposed solution $\mathbf{v}$ to the problem (4.1) can be quantified using its *cost*, $\theta(\mathbf{v}) - \theta(\mathbf{w}^*)$ (3.2). The output of Algorithm 8 is $\bar{\mathbf{w}}_T = \frac{1}{T}\sum_{t\in[T]} \mathbf{w}_t$, the average of the test points. When the output of the algorithm is the average test point, in this way, then regret and expected cost of the output vector are related by the following standard theorem (Xiao, 2010):

Theorem 2. *Suppose some stochastic function $\theta_{\ell,Z,\Psi}$ is given. Suppose that, when run on indices $\mathbf{z}_T$, drawn according to Z, some algorithm produces test points $(\mathbf{w}_t)_{t\in[T]}$. And, suppose we have some $\Delta \in \mathbb{R}$ such that for all sequences $\mathbf{z}_T$, $R_{\mathbf{z}_T,\theta}((\mathbf{w}_t)_{t\in[T]}) \leqslant \Delta$. Then:*

$$\mathbb{E}_{\mathbf{z}_T}\left[\theta(\bar{\mathbf{w}}_T)\right] - \theta(\mathbf{w}^*) \leqslant \frac{1}{T}\Delta$$

Here the expectation is taken over the probability of drawing the sequence $\mathbf{z}_T$, according to Z.

4.2.2.2 Bounding Regret

We will now quote a theorem of Xiao (2010), bounding the regret of Algorithm 8. We include only a part of the proof, that which will be used in the next section:

Theorem 3. *(Xiao, 2010) Suppose we are given some indexed convex loss function ℓ, λ-strongly convex regularizer Ψ, and some distribution Z over examples. From these, we create the regularized stochastic objective $\theta_{\ell,\Psi,Z}$. Let $\mathbf{z}_T$ be an arbitrary sequence of examples. Suppose that the test points $(\mathbf{w}_t)_{t\in[T]}$ are generated according to Algorithm 8 on input $\mathbf{z}_T$. Suppose that there exists a $G \in \mathbb{R}$, such that for all $t \in [T]$, $\|\mathbf{g}_t\|_* \leqslant G$. Then, the regret is bounded by:*

$$R_{\mathbf{z}_T,\theta}((\mathbf{w}_t)_{t\in[T]}) \leqslant \frac{G^2}{2\lambda}(6 + \log T)$$

Proof. Suppose that a sequence of weights $(\mathbf{w}_t)_{t\in[T]}$ is generated according to $\mathbf{w}_1 = \Pi_{\Psi,1}(0)$, and for $t \geqslant 1$, $\mathbf{w}_{t+1} = \Pi_{\Psi,t}(\sum_{\tau\in[t]} \mathbf{g}_\tau)$, and define the following *gap function*:

$$\delta^*((\mathbf{g}_t)_{t\in[T]}, \Psi) \triangleq \max_{\mathbf{w}\in\mathcal{W}} \left\{ \left[\sum_{t\in[T]} \langle \mathbf{g}_t, \mathbf{w}_t - \mathbf{w} \rangle + \Psi(\mathbf{w}_t) \right] - T\Psi(\mathbf{w}) \right\} \tag{4.14}$$

The $\mathbf{w}_t$ are not passed explicitly as an argument to δ^*, but are determined on the basis of the gradients according to Algorithm 8. δ^* is a bound on the regret for any $\mathbf{w}^* \in \mathbb{R}^n$ (Nesterov, 2009; Xiao, 2010), and Xiao (2010) shows:

$$\delta^*((\mathbf{g}_t)_{t\in[T]}, \Psi) \leqslant \frac{5\|\mathbf{g}_1\|_*^2}{2\lambda} + \sum_{t=2}^{T} \frac{\|\mathbf{g}_t\|_*^2}{2\lambda(t-1)} \tag{4.15}$$

Recall that $G \in \mathbb{R}$ is a constant such that $\|\mathbf{g}_t\|_* \leqslant G$ for all $t \in [T]$. Then, we have:

$$R_{\mathbf{z}_T,\theta}\left((\mathbf{w}_t)_{t\in[T]}\right) \leqslant \delta^*((\mathbf{g}_t)_{t\in[T]}, \Psi) \leqslant \frac{5G^2}{2\lambda} + \sum_{t=2}^{T} \frac{G^2}{2\lambda(t-1)} \leqslant \frac{G^2}{2\lambda}(6 + \log T) \tag{4.16}$$

□

This implies (by Theorem 2) that the expected error of the output is bounded by:

$$\mathbb{E}_{\mathbf{z}_T}[\theta(\bar{\mathbf{w}}_T)] - \theta(\mathbf{w}^*) \leqslant O\left(\frac{G^2 \log T}{\lambda T}\right) \tag{4.17}$$

4.2.3 Comparison with Other Work

We here discuss the RDA algorithm in the context of other sequential stochastic optimizers, in order to better understand how the distributed variant relates to other optimization strategies. The canonical stochastic optimization approach to the task $\arg\min_{\mathbf{w}\in\mathcal{W}} F(\mathbf{w})$ is to adopt the random first-order oracle model of optimization (§3.3). Suppose F is λ-strongly convex, and the norms of the vectors returned by the oracle are bounded by H. Hazan & Kale (2011) have shown that, under these conditions, the optimal regret of any algorithm constrained to the random first-order oracle model is $\Omega(\frac{H^2 \log T}{\lambda})$. Similarly, Agarwal et al. (2012) have shown that the optimal convergence rate for any random first-order oracle model algorithm is $\Omega(\frac{H^2}{\lambda T})$. Hazan & Kale (2011) and Juditsky & Nesterov (2014) achieve the optimal convergence rate, both using a strategy of exponentially decreasing step sizes, with decreases coming after exponentially growing intervals. Table 4.1 compares Xiao's (2010) regularized dual averaging to important reference algorithms.

Algorithm	Regret	Convergence Rate
Xiao's (2010) RDA	$O(\frac{G^2 \log T}{\lambda})$	$O(\frac{G^2 \log T}{\lambda T})$
Hazan et al. (2007)	$O(\frac{H^2 \log T}{\lambda})$	$O(\frac{H^2 \log T}{\lambda T})$
Hazan & Kale (2011)	$O(\frac{H^2 \log T}{\lambda})$	$O(\frac{H^2}{\lambda T})$

Table 4.1: A comparison of regret bounds and stochastic optimization convergence rate bounds for three sequential algorithms. H bounds regularized gradients, G bounds unregularized gradients, λ is the convexity parameter, and T counts the number of iterations.

Due to the short-comings of the online-to-batch method of convergence proof (Hazan & Kale, 2011), Xiao's (2010) bound does not attain the asymptotically optimal convergence rate as a function of T. However, the results of §5.8.1 show that regularized dual averaging is as good an optimizer in practice as Hazan & Kale (2011). In theoretical terms, one interesting difference between the dual averaging bound and the others is the dependence on the constant term bounding sub-gradient norms for a regularized loss function of the form $\theta_{\ell,Z,\Psi}$ (cf. equation 4.1). H, in our notation, bounds the sub-gradients of the *regularized* function (i.e. elements of $\partial\left[\ell(\mathbf{w},z)+\Psi(\mathbf{w})\right]$). In contrast, G, as it has been used in this paper, bounds the sub-gradients of the *unregularized* function (i.e. elements of $\partial\left[\ell(\mathbf{w},z)\right]$). It is unclear, however, that this will lead to a meaningful difference in practice. As for modifying the dual averaging algorithm to attain the optimal convergence rate, it should be noted that the algorithms of Hazan & Kale (2011) and Juditsky & Nesterov (2014) each use an unfamiliar step-size regime, and subsequent authors have looked at ways to modify familiar algorithms to achieve the optimal rate. Shamir (2011) and Lacoste-Julien et al. (2012) have shown that simply modifying the averaging step of stochastic gradient descent algorithms can lead to the asymptotically optimal rate. And, Chen et al. (2012) have shown that a change to the proximal projection step of regularized dual averaging can produce a regularized dual averaging algorithm with the asymptotically optimal rate. We leave the question of modifying the algorithm to obtain optimal rates to future work, and use the simpler online-to-batch analysis here, which will be enough to yield an improvement over past work in the distributed dual averaging case and provide a useful algorithm for the work of §5.

4.3 Distributed Regularized Dual Averaging

We now present the distributed regularized dual averaging algorithm. This algorithm and these bounds broadly follow Duchi et al. (2011b, 2012), but offer improved bounds and a simplified algorithm for the strongly regularized case by incorporating techniques adapted from Xiao (2010). Also, Lemma 2 is novel.

4.3.1 Distributed Dual Averaging Algorithm

In the distributed version of the algorithm, we have N worker nodes, each of which has access to its own T examples. From this sequence of sequences of samples, we iteratively approximate the optimizer for the Θ in an online fashion. Communication between nodes on any round $t \in [T]$ can be modelled using graph, $G = (V_N, E_t)$. An edge between two nodes n_i and n_j in E_t specifies that n_i and n_j communicate on round t. The communication and aggregation of opinion at round t is described by a doubly stochastic matrix $P(t)$. T rounds of the algorithm will require a sequence of communication matrices, $(P(t))_{t\in[T]}$, of length T. Let $P(t)_{i,j}$ be the entry in the i'th row and j'th column of the square matrix $P(t)$. To say that an $N \times N$ matrix P is doubly stochastic is to say that $P_{i,j} \geqslant 0$ for each $i, j \in [N]$ and that:

$$\sum_{i\in[N]} P_{i,j} = 1 \text{ and } \sum_{j\in[N]} P_{i,j} = 1 \tag{4.18}$$

The relevance of the choice of a doubly stochastic matrix is that it ensures that information will eventually propagate throughout the network, and will be discussed in greater detail in §5. The fact that $P(t)$ can change as a function of the round index t means that there can be different communication patterns at different rounds. And, $P(t)_{i,j}$ may only be greater than 0 if n_i and n_j are connected in E.

As shown in Algorithm 9, each node $n \in [N]$ will maintain its own copy of a dual average at round t, called $\mathbf{d}_t^n$. But, now, instead of line 6 of Algorithm 8, we have the following update rule for node n at time t:

$$\mathbf{d}_t^n \leftarrow \sum_{i\in[N]} P(t)_{n,i}\mathbf{d}_{t-1}^n + \mathbf{g}_t^n \tag{4.19}$$

In other words, the update at node n, at time t, involves the gradient $\mathbf{g}_t^n$, along with a pooled estimate of the dual averages from all neighbouring nodes, according to $P(t)$. From this we can use the same projection step (4.10) as before to obtain a primal variable $\mathbf{w}_t^n$. The input to the distributed algorithm is oracle $\Omega_{\partial\ell}$ and regularizer Ψ, as

before, along with a *sequence of sequences* of indices $\mathbf{z}_T^N \triangleq \left((z_t^n)_{t\in[T]}\right)_{n\in[N]}$, instead of just a sequence $\mathbf{z}_T$, and a sequence of communication matrices $(P(t))_{t\in[T]}$. The output of the algorithm, depending on one's application environment, can either be $\bar{\mathbf{w}}_T = \frac{1}{TN}\sum_{t\in[T]}\sum_{n\in[N]}\mathbf{w}_t^n$, the average of the average of the test points over all rounds and nodes, or else $\bar{\mathbf{w}}_T^n = \frac{1}{T}\sum_{t\in[T]}\mathbf{w}_t^n$, the average of the test points for the single node $n \in [N]$.

Algorithm 9 Distributed Regularized Stochastic Dual Averaging

procedure DISTRIBUTED-RSDA-MINIMIZE($\Omega_{\partial\ell}, \Psi, \mathbf{z}_T^N, (P(t))_{t\in[T]}$)
  for $n \in [N]$ **do in parallel**
    $\mathbf{d}_0^n \leftarrow 0 \in \mathbb{R}^n$
    $\mathbf{w}_1^n \leftarrow \arg\min_{\mathbf{w}\in\mathcal{W}} \Psi(\mathbf{w})$
    for $t \in [T]$ **do**
      receive $\mathbf{g}_t^n \in \partial\ell(\mathbf{w}_t^n, z_t^n)$ by calling $\Omega_{\partial\ell}(\mathbf{w}_t^n, z_t^n)$
      $\mathbf{d}_t^n \leftarrow \sum_{i\in[N]} P(t)_{n,i}\mathbf{d}_{t-1}^n + \mathbf{g}_t^n$
      $\mathbf{w}_{t+1}^n \leftarrow \arg\min_{\mathbf{w}\in\mathcal{W}} \{\langle \mathbf{d}_t^n, \mathbf{w}\rangle + t\Psi(\mathbf{w})\}$
  global: **return** $\bar{\mathbf{w}}_T = \frac{1}{TN}\sum_{t\in[T]}\sum_{n\in[N]}\mathbf{w}_t^n$
  local n: **return** $\bar{\mathbf{w}}_T^n = \frac{1}{T}\sum_{t\in[T]}\mathbf{w}_t^n$

Examples There are two types of sequences $(P(t))_{t\in[T]}$ that we will focus on in this thesis, which are the two types of sequence that can be used to model the iterative parameter mixing distribution optimization algorithms (Algorithms 1 and 3) and the single-mixture distributed optimization algorithms (Algorithms 2 and 4). As §5.5 describes in more detail, the single-mixture distribution algorithm, in which nodes do not communicate during the optimization process, can be modelled as a process that sets the communication matrix $P(t)$ equal to the identity matrix on each round $t \in [T]$. The intuition is that information is shared between nodes i and j on round t requires that $P(t)_{i,j} > 0$. In the case of the identity matrix, we have $P(t)_{i,j} > 0$ if and only if $i = j$. And, as §5.5 also describes, we can model the iterative parameter mixing algorithm using a mix of uniform and identity matrices. The $N \times N$ uniform matrix has all entries equal to $\frac{1}{N}$ (see §5.1). The uniform matrix models a round of full communication. That is, each node is connected to each other node, and weights are combined equally. Suppose that communication takes place every M rounds. Then, we can model iterative parameter mixing as an algorithm that communicates according to the uniform matrix every round m such that $m \mod M = 1$, and uses the identity matrix on each of the

other and the other $M-1$ out of each M rounds using the identity matrix.

4.3.2 Distributed Dual Averaging Analysis

We now want to consider the convergence of the distributed version of the regularized dual averaging algorithm. Our main quantity of interest is the expected cost of each local vector $n \in [N]$:

$$\mathbb{E}_{\mathbf{z}_T^N}\left[\Theta(\bar{\mathbf{w}}_T^n)\right] - \Theta(\mathbf{w}^*) \tag{4.20}$$

This will allow us to bound the error of the average of these local vectors,

$$\mathbb{E}_{\mathbf{z}_T^N}\left[\Theta(\frac{1}{N}\sum_{n\in[N]}\bar{\mathbf{w}}_T^n)\right] - \Theta(\mathbf{w}^*), \tag{4.21}$$

which will be used in §5 (see, 5.67). This will involve bounding the *distributed regret* of these sequence of vectors, defined in the next section. Along the way, we will also need to bound the regret of an auxiliary centralized primal sequence. The organization of this section is as follows. §4.3.2.1 defines the notion of distributed regret and explains the relation to expected cost. §4.3.2.2 gives an important lemma to allow a bound of the regret of a useful auxiliary central sequence. §4.3.2.3 gives an important lemma to allow a bound of the regret of the local sequences $(\mathbf{w}_t^n)_{t\in[T]}$. Finally, §4.3.2.4 uses these lemmas to bound expected cost in concrete cases.

4.3.2.1 Distributed Regret and Cost

We want to define a distributed notion of regret. We begin by introducing two auxiliary sequences of variables that will be central to the analysis:

$$\mathbf{e}_t \triangleq \frac{1}{N}\sum_{n\in[N]}\mathbf{d}_t^n \tag{4.22}$$

$$\mathbf{u}_{t+1} \triangleq \Pi_{\Psi,t}(\mathbf{e}_t), t \geqslant 1 \tag{4.23}$$

$$\mathbf{u}_1 \triangleq \Pi_{\Psi,1}(\mathbf{e}_0) = \Pi_{\Psi,1}(0) \tag{4.24}$$

$\mathbf{e}_t$ is the average of dual averages $\mathbf{d}_t^n$ across all nodes n. $\mathbf{u}_t$ is the projection of this central dual average into the primal space by the same rules as (4.11), (4.12).

Distributed regret is defined for an arbitrary sequence test points $(\mathbf{v}_t)_{t\in[T]}$, with respect to a distributed stochastic objective Θ being optimized and a sequence of se-

quence of indices $\mathbf{z}_T^N$:

$$R_{\mathbf{z}_T^N,\Theta}\left(\left((\mathbf{v}_t^n)_{t\in[T]}\right)_{n\in[N]}\right) \triangleq \left[\sum_{t\in[T]}\frac{1}{N}\sum_{n\in[N]}\left[\ell(\mathbf{v}_t,z_t^n)+\Psi(\mathbf{v}_t)\right]\right] - \left[\sum_{t\in[T]}\frac{1}{N}\sum_{n\in[N]}\left[\ell(\mathbf{w}^*,z_t^n)+\Psi(\mathbf{w}^*)\right]\right] \tag{4.25}$$

Distributed regret compares the performance of the sequence $(\mathbf{v}_t)_{t\in[T]}$ to that of the solution $\mathbf{w}^*$ on the observed sequence of sequences of examples $\mathbf{z}_T^N$. By arguments nearly identical to those that proved Theorem 2, we have the following relationship between regret and expected cost in the distributed case:

Theorem 4. *Suppose some distributed stochastic function $\Theta_{\ell,(Z_n)_{n\in[N]},\Psi}$ is given. Let $(\mathbf{v}_t)_{t\in[T]}$ be the test points emitted by a given optimization algorithm when run on the indices $\mathbf{z}_T^N$ (the test points are a function of the indices). Suppose the $\mathbf{z}_T^N$ are drawn according to the $(Z_n)_{n\in[N]}$. And, suppose we have some $\Delta\in\mathbb{R}$ such that for all $\mathbf{z}_T^N$, $R_{\mathbf{z}_T^N,\Theta}((\mathbf{v}_t)_{t\in[T]})\leqslant\Delta$. Then, where $\bar{\mathbf{v}}_T \triangleq \frac{1}{T}\sum_{t\in[T]}\mathbf{v}_t$, we have:*

$$\mathbb{E}_{\mathbf{z}_T^N}\left[\Theta(\bar{\mathbf{v}}_T)\right]-\Theta(\mathbf{w}^*)\leqslant\frac{1}{T}\Delta$$

Here the expectation is taken over the probabilities of drawing the sequence of sequences $(z_t^n)_{t\in[T]}$, according to the Z_n, for $n\in[N]$.

4.3.2.2 A Lemma About the Central Primal Sequence

In this section, we will show how to bound the regret of the central sequence $(\mathbf{u}_t)_{t\in[T]}$. This result will be used in the next section to give regret bounds for the sequences $(\mathbf{w}_t^n)_{t\in[T]}$. Before this, let us make an observation about the evolution of the sequence

$(\mathbf{e}_t)_{t\in[T]}$. By the definition of $\mathbf{e}_t$ (4.22), we have:[2]

$$\mathbf{e}_{t+1} = \frac{1}{N}\sum_{n\in[N]} \mathbf{d}_{t+1}^n \tag{4.28}$$

$$= \mathbf{e}_t + \frac{1}{N}\sum_{n\in[N]} \mathbf{g}_t^n \tag{4.29}$$

The following Lemma follows Duchi et al. (2012) closely, but, uses the assumption of strong convexity to use a stronger bound on the gap function (4.49) using Xiao (2010):

Lemma 1. *Suppose we are given some loss function ℓ, λ-strongly convex regularizer Ψ, and N distributions over examples $(Z_n)_{n\in[N]}$. From these, we create the distributed stochastic objective $\Theta_{\ell,(Z_n)_{n\in[N]},\Psi}$. Let $\mathbf{z}_T^N$ be an arbitrary sequence of sequences of examples drawn according to the $(Z_n)_{n\in[N]}$. Suppose that the central sequence $(\mathbf{u}_t)_{t\in[T]}$ are generated according to Algorithm 9, along with equations 4.22 4.23 and 4.24. Suppose that, there exists a $G\in\mathbb{R}$, such that for all $t\in[T], n\in[N]$, $\|\mathbf{g}_t^n\|_* \leqslant G$. And, suppose there exists a $D_{PTG}\in\mathbb{R}$ such that $\|\mathbf{e}_t - \mathbf{d}_t^n\|_* \leqslant D_{PTG}$ given G and $(P(t))_{t\in[T]}$. Then, the regret of the central sequence $(\mathbf{u}_t)_{t\in[T]}$ is bounded by:*

$$R_{\mathbf{z}_T^N,\Theta}((\mathbf{u}_t)_{t\in[T]}) \leqslant \frac{(6+\log T)}{\lambda}\left(\frac{G^2}{2} + 2GD_{PTG}\right)$$

Proof. We begin by noting that:

$$\sum_{t\in[T]}\sum_{n\in[N]}\left[\ell(\mathbf{u}_t, z_t^n) - \ell(\mathbf{w}^*, z_t^n)\right] \tag{4.30}$$

$$= \sum_{t\in[T]}\sum_{n\in[N]}\left[\ell(\mathbf{w}_t^n, z_t^n) - \ell(\mathbf{w}^*, z_t^n)\right] + \sum_{t\in[T]}\sum_{n\in[N]}\left[\ell(\mathbf{u}_t, z_t^n) - \ell(\mathbf{w}_t^n, z_t^n)\right] \tag{4.31}$$

$$\leqslant \sum_{t\in[T]}\sum_{n\in[N]}\left[\ell(\mathbf{w}_t^n, z_t^n) - \ell(\mathbf{w}^*, z_t^n)\right] + \sum_{t\in[T]}\sum_{n\in[N]} G\|\mathbf{u}_t - \mathbf{w}_t^n\| \tag{4.32}$$

[2]This is because:

$$\mathbf{e}_{t+1} = \frac{1}{N}\sum_{n\in[N]}\mathbf{d}_{t+1}^n = \frac{1}{N}\sum_{n\in[N]}\sum_{j\in[N]}\left[P(t)_{n,j}\mathbf{d}_t^n + \mathbf{g}_t^n\right] \tag{4.26}$$

And:

$$\frac{1}{N}\left[\sum_{n\in[N]}\sum_{j\in[N]} P(t)_{n,j}\mathbf{d}_t^n\right] = \frac{1}{N}\left[\sum_{j\in[N]}\mathbf{d}_t^n\sum_{n\in[N]}P(t)_{n,j}\right] = \frac{1}{N}\left[\sum_{j\in[N]}\mathbf{d}_t^n\right] = \mathbf{e}_t \tag{4.27}$$

In the last step we have used the G-Lipschitz continuity of $\ell(.,z_t^n)$.

Now, suppose that $\mathbf{g}_t^n \in \partial\ell(\mathbf{w}_t^n, z_t^n)$ is a sub-gradient of ℓ at $(\mathbf{w}_t^n, z_t^n)$. Then, by the definition of a sub-gradient and the convexity of $\ell(.,z_t^n)$ we have that:

$$\sum_{t\in[T]}\sum_{n\in[N]} \ell(\mathbf{w}_t^n, z_t^n) - \ell(\mathbf{w}^*, z_t^n) \leqslant \sum_{t\in[T]}\sum_{n\in[N]} \langle \mathbf{g}_t^n, \mathbf{w}_t^n - \mathbf{w}^* \rangle \tag{4.33}$$

Then:

$$\sum_{t\in[T]}\sum_{n\in[N]} \langle \mathbf{g}_t^n, \mathbf{w}_t^n - \mathbf{w}^* \rangle \tag{4.34}$$

$$= \sum_{t\in[T]}\sum_{n\in[N]} \langle \mathbf{g}_t^n, \mathbf{u}_t - \mathbf{w}^* \rangle + \sum_{t\in[T]}\sum_{n\in[N]} \langle \mathbf{g}_t^n, \mathbf{w}_t^n - \mathbf{u}_t \rangle \tag{4.35}$$

$$\leqslant \sum_{t\in[T]}\sum_{n\in[N]} \langle \mathbf{g}_t^n, \mathbf{u}_t - \mathbf{w}^* \rangle + \sum_{t\in[T]}\sum_{n\in[N]} G\|\mathbf{w}_t^n - \mathbf{u}_t\| \tag{4.36}$$

Thus:

$$\begin{aligned}\sum_{t\in[T]}\sum_{n\in[N]} &[\ell(\mathbf{u}_t, z_t^n) - \ell(\mathbf{w}^*, z_t^n)] \\ &\leqslant \sum_{t\in[T]}\sum_{n\in[N]} \langle \mathbf{g}_t^n, \mathbf{u}_t - \mathbf{w}^* \rangle + 2\sum_{t\in[T]}\sum_{n\in[N]} G\|\mathbf{w}_t^n - \mathbf{u}_t\|\end{aligned} \tag{4.37}$$

Clearly, then, the key is to bound $\|\mathbf{u}_t - \mathbf{w}_t^n\|$, for arbitrary t, n. We use the following well-known result, based on the Lipschitz smoothness of the conjugate of a strongly convex function (Hiriart-Urruty & Lemaréchal, 1996; Nesterov, 2009; Xiao, 2010; Duchi et al., 2012):

$$\left\|\Pi_{\Psi,t}(\mathbf{d}_1) - \Pi_{\Psi,t}(\mathbf{d}_2)\right\| \leqslant \frac{1}{\lambda t}\|\mathbf{d}_1 - \mathbf{d}_2\|_* \tag{4.38}$$

For $t \geqslant 2$, $\mathbf{u}_t = \Pi_{\Psi,t-1}(\mathbf{e}_{t-1})$ and $\mathbf{w}_t = \Pi_{\Psi,t-1}(\mathbf{d}_{t-1}^n)$. Thus, we have, for $t \geqslant 2$:

$$\|\mathbf{u}_t - \mathbf{w}_t^n\| \leqslant \frac{1}{\lambda(t-1)}\left\|\mathbf{e}_{t-1} - \mathbf{d}_{t-1}^n\right\|_* \tag{4.39}$$

For $t = 1$ we have $\mathbf{w}_1^n = \arg\min_{\mathbf{w}\in\mathcal{W}} \Psi(\mathbf{w})$ for all n. Thus $\mathbf{u}_1 = \mathbf{w}_1^n$ for all n so $\left\|\mathbf{u}_1 - \mathbf{w}_1^n\right\| = 0$ for all n.

We will delay the question of how to bound $\|\mathbf{e}_t - \mathbf{d}_t^n\|_*$ until after this lemma. But, recall the assumption that $D_{PTG} \in \mathbb{R}$ is such that $\|\mathbf{e}_t - \mathbf{d}_t^n\|_* \leqslant D_{PTG}$ for all $t \in [T]$ and

$n \in [N]$. Thus:

$$\sum_{t\in[T]}\sum_{n\in[N]} \|\mathbf{w}_t^n - \mathbf{u}_t\| \leqslant \sum_{n\in[N]} \|\mathbf{w}_1^n - \mathbf{u}_1\| + \sum_{t=2}^{T}\sum_{n\in[N]} \|\mathbf{w}_t^n - \mathbf{u}_t\| \tag{4.40}$$

$$\leqslant 0 + \sum_{t=2}^{T}\sum_{n\in[N]} \frac{1}{\lambda(t-1)} \|\mathbf{d}_t^n - \mathbf{e}_t\|_* \tag{4.41}$$

$$\leqslant \sum_{t=2}^{T}\sum_{n\in[N]} \frac{D_{PTG}}{\lambda(t-1)} \tag{4.42}$$

$$\leqslant \sum_{t\in[T]}\sum_{n\in[N]} \frac{D_{PTG}}{\lambda t} \tag{4.43}$$

Thus, we have that:

$$R_{\mathbf{z}_T^N,\Theta}((\mathbf{u}_t)_{t\in[T]}) \tag{4.44}$$

$$= \sum_{t\in[T]} \frac{1}{N} \left[\sum_{n\in[N]} [\ell(\mathbf{u}_t, z_t^n) - \ell(\mathbf{w}^*, z_t^n)] + [\Psi(\mathbf{u}_t) - \Psi(\mathbf{w}^*)] \right] \tag{4.45}$$

$$\leqslant \sum_{t\in[T]} \frac{1}{N} \left[\sum_{n\in[N]} \langle \mathbf{g}_t^n, \mathbf{u}_t - \mathbf{w}^* \rangle + [\Psi(\mathbf{u}_t) - \Psi(\mathbf{w}^*)] \right] + \frac{2}{N} \sum_{t\in[T]}\sum_{n\in[N]} G\|\mathbf{u}_t - \mathbf{w}_t^n\| \tag{4.46}$$

$$\leqslant \sum_{t\in[T]} \frac{1}{N} \left[\sum_{n\in[N]} \langle \mathbf{g}_t^n, \mathbf{u}_t - \mathbf{w}^* \rangle + [\Psi(\mathbf{u}_t) - \Psi(\mathbf{w}^*)] \right] + \frac{2}{N} \sum_{t\in[T]}\sum_{n\in[N]} \frac{GD_{PTG}}{\lambda t} \tag{4.47}$$

For the first inequality, we used (4.37). In the second, we used (4.43).

We now seek to bound the first term in (4.47). Note that:

$$\sum_{t\in[T]} \frac{1}{N} \sum_{n\in[N]} \langle \mathbf{g}_t^n, \mathbf{u}_t - \mathbf{w}^* \rangle = \sum_{t\in[T]} \langle \frac{1}{N} \sum_{n\in[N]} \mathbf{g}_t^n, \mathbf{u}_t - \mathbf{w}^* \rangle \tag{4.48}$$

We quoted Xiao's (2010) result (4.15) about the gap function δ, which said that, if $(\mathbf{f}_t)_{t\in[T]}$ is an arbitrary sequence of vectors, and if, $\mathbf{v}_1 = \Pi_{\Psi,1}(0)$, and for $t \geqslant 1$, $\mathbf{v}_{t+1} = \Pi_{\Psi,t}(\sum_{\tau\in[t]} \mathbf{f}_\tau)$, then:

$$\sum_{t\in[T]} [\langle \mathbf{f}_t, \mathbf{v}_t - \mathbf{w}^* \rangle + [\Psi(\mathbf{v}_t) - \Psi(\mathbf{w}^*)]] \leqslant \sum_{t\in[T]} \frac{5\|\mathbf{f}_t\|_*}{2\lambda} + \frac{1}{2t}\|\mathbf{f}_t\|_*^2 \tag{4.49}$$

This is true whether $\mathbf{f}_\tau$ takes on the value $\mathbf{g}_\tau$, or else $\frac{1}{N}\sum_{n\in[N]} \mathbf{g}_\tau^n$. Therefore, given the evolution of the $\mathbf{e}_t$, as described by (4.29), and the fact that the $\mathbf{u}_t$, described by (4.23)

and (4.24), meet the requirements for (4.49) to hold, we have:

$$\left[\sum_{t\in[T]}\frac{1}{N}\left[\sum_{n\in[N]}\langle \mathbf{g}_t^n, \mathbf{u}_t-\mathbf{w}^*\rangle + [\Psi(\mathbf{u}_t)-\Psi(\mathbf{w}^*)]\right]\right] \tag{4.50}$$

$$=\sum_{t\in[T]}\left[\langle\frac{1}{N}\sum_{n\in[N]}\mathbf{g}_t^n, \mathbf{u}_t-\mathbf{w}^*\rangle + \frac{1}{N}\left[\sum_{n\in[N]}[\Psi(\mathbf{u}_t)-\Psi(\mathbf{w}^*)]\right]\right] \tag{4.51}$$

$$=\sum_{t\in[T]}\left[\langle\frac{1}{N}\sum_{n\in[N]}\mathbf{g}_t^n, \mathbf{u}_t-\mathbf{w}^*\rangle + [\Psi(\mathbf{u}_t)-\Psi(\mathbf{w}^*)]\right] \tag{4.52}$$

$$\leqslant\frac{5\left\|\frac{1}{N}\sum_{n\in[N]}\mathbf{g}_1^n\right\|_*}{2\lambda}+\sum_{t=2}^{T}\frac{1}{2\lambda t}\left\|\frac{1}{N}\sum_{n\in[N]}\mathbf{g}_t^n\right\|_*^2 \tag{4.53}$$

$$\leqslant\frac{5G^2}{2\lambda}+\sum_{t\in[T]}\frac{1}{2\lambda t}G^2 \tag{4.54}$$

$$\leqslant\frac{G^2}{2\lambda}(6+\log T) \tag{4.55}$$

The first inequality uses (4.49).

We can now assemble the above results to complete the lemma. Starting from line (4.47):

$$R_{\mathbf{z}_T^N,\Theta}((\mathbf{u}_t)_{t\in[T]}) \tag{4.56}$$

$$\leqslant\sum_{t\in[T]}\frac{1}{N}\left[\sum_{n\in[N]}\langle \mathbf{g}_t^n, \mathbf{u}_t-\mathbf{w}^*\rangle + [\Psi(\mathbf{u}_t)-\Psi(\mathbf{w}^*)]\right]+\frac{2}{N}\sum_{t\in[T]}\sum_{n\in[N]}\frac{GD_{PTG}}{\lambda t} \tag{4.57}$$

$$\leqslant\frac{G^2}{2\lambda}(6+\log T)+2\sum_{t\in[T]}\frac{GD_{PTG}}{\lambda t} \tag{4.58}$$

$$\leqslant\frac{G^2}{2\lambda}(6+\log T)+\frac{2GD_{PTG}}{\lambda}(1+\log T) \tag{4.59}$$

$$\leqslant\frac{(6+\log T)}{\lambda}(\frac{G^2}{2}+2GD_{PTG}) \tag{4.60}$$

□

4.3.2.3 A Lemma About the Local Sequences

The next lemma will show how to bound the regret of each local sequence $(\mathbf{w}_t^n)_{t\in[T]}$, for $n\in[N]$. This section departs the furthest from past work. While Duchi et al. (2011b) contains something close to Lemma 1, it does not contain anything similar to Lemma 2.

First, for future use, we want to show that:

$$\|\mathbf{d}_t^n\|_* \leqslant Gt, \forall t \in [T], n \in [N] \tag{4.61}$$

We prove this by induction. For $t = 0$, we have $\mathbf{d}_t^n = 0$ for all $n \in [N]$, and $\|0\|_* = 0 \leqslant G \cdot 0$. For $t \geqslant 1$, assume the induction hypothesis holds for $t-1$. Then, we have, for any $n \in [N]$:

$$\left\| \sum_{i \in [N]} P(t)_{n,i} \mathbf{d}_{t-1}^i \right\|_* \leqslant \sum_{i \in [N]} P(t)_{n,i} \left\| \mathbf{d}_{t-1}^i \right\|_* \leqslant \sum_{i \in [N]} [P(t)_{n,i} G(t-1)] = G(t-1) \tag{4.62}$$

The first step uses the triangle inequality. The second step uses the induction hypothesis. The third step uses the fact that $\sum_{i \in [N]} P(t)_{n,i} = 1$. Using the triangle inequality once more:

$$\left\| \sum_{i \in [N]} P(t)_{n,i} \mathbf{d}_{t-1}^n + \mathbf{g}_t^n \right\|_* \leqslant \left\| \sum_{i \in [N]} P(t)_{n,i} \mathbf{d}_{t-1}^n \right\|_* + \|\mathbf{g}_t^n\|_* \leqslant G(t-1) + G = Gt \tag{4.63}$$

Thus, by induction, (4.61) holds for all $t \in [T]$.

The following Lemma is novel:

Lemma 2. *Suppose the conditions of Lemma 1 hold. Suppose that $(\mathbf{w}_t^n)_{t \in [T]}$ are the test points for the n'th node when Algorithm 9 is run on the sequence $\mathbf{z}_T^N$. Then, the regret for the local sequence $(\mathbf{w}_t^n)_{t \in [T]}$ is bounded by:*

$$R_{\mathbf{z}_T^N, \Theta}((\mathbf{w}_t^n)_{t \in [T]}) \leqslant \frac{(6 + \log T)}{\lambda} \left(\frac{G^2}{2} + 4GD_{PTG} \right)$$

Proof. For brevity, let $\ell_\Psi(\mathbf{w}, z) \stackrel{\Delta}{=} \ell(\mathbf{w}, z) + \Psi(\mathbf{w})$. Then:

$$R_{\mathbf{z}_T^N, \Theta}((\mathbf{w}_t^n)_{t \in [T]}) \tag{4.64}$$

$$= \frac{1}{N} \sum_{t \in [T]} \sum_{n \in [N]} [\ell_\Psi(\mathbf{w}_t^n, z_t^n) - \ell_\Psi(\mathbf{w}^*, z_t^n)] \tag{4.65}$$

$$= \frac{1}{N} \sum_{t \in [T]} \sum_{n \in [N]} [\ell_\Psi(\mathbf{u}_t, z_t^n) - \ell_\Psi(\mathbf{w}^*, z_t^n)] + \frac{1}{N} \sum_{t \in [T]} \sum_{n \in [N]} [\ell_\Psi(\mathbf{w}_t^n, z_t^n) - \ell_\Psi(\mathbf{u}_t, z_t^n)] \tag{4.66}$$

The term on the left is just $R_{\mathbf{z}_T^N, \Theta}((\mathbf{u}_t)_{t \in [T]})$, which we bounded in Lemma 1. The term on the right, which we need to bound now, is equal to $\frac{1}{N}$ times:

$$\sum_{t \in [T]} \sum_{n \in [N]} [\ell(\mathbf{w}_t^n, z_t^n) - \ell(\mathbf{u}_t, z_t^n)] + \sum_{t \in [T]} \sum_{n \in [N]} [\Psi(\mathbf{w}_t^n) - \Psi(\mathbf{u}_t)] \tag{4.67}$$

We bound $\ell(\mathbf{w}_t^n, z_t^n) - \ell(\mathbf{u}_t, z_t^n)$ as before. By the G-Lipschitz continuity of ℓ for any index z_t^n, we have:

$$\ell(\mathbf{w}_t^n, z_t^n) - \ell(\mathbf{u}_t, z_t^n) \leqslant G \left\| \mathbf{w}_t^n - \mathbf{u}_t \right\| \tag{4.68}$$

Now, we want to bound $\Psi(\mathbf{w}_t^n) - \Psi(\mathbf{u}_t)$. We have

$$\mathbf{w}_1^n = \mathbf{u}_1 = \arg\min_{\mathbf{w} \in \mathcal{W}} \Psi(\mathbf{w}) = 0$$

so $\Psi(\mathbf{w}_1^n) - \Psi(\mathbf{u}_1) = 0$. Then, for $t \geqslant 2$,

$$\mathbf{w}_t^n = \arg\min_{\mathbf{w} \in \mathcal{W}} \left\{ \langle \mathbf{d}_{t-1}^n, \mathbf{w} \rangle + (t-1)\Psi(\mathbf{w}) \right\}$$

This implies that, for $t \geqslant 2$:

$$\langle \mathbf{d}_{t-1}^n, \mathbf{w}_t^n \rangle + (t-1)\Psi(\mathbf{w}_t^n) \leqslant \langle \mathbf{d}_{t-1}^n, \mathbf{u}_t \rangle + (t-1)\Psi(\mathbf{u}_t) \tag{4.69}$$

$$(t-1)\Psi(\mathbf{w}_t^n) - (t-1)\Psi(\mathbf{u}_t) \leqslant \langle \mathbf{d}_{t-1}^n, \mathbf{u}_t - \mathbf{w}_t^n \rangle \tag{4.70}$$

$$(t-1)\Psi(\mathbf{w}_t^n) - (t-1)\Psi(\mathbf{u}_t) \leqslant \left\| \mathbf{d}_{t-1}^n \right\|_* \left\| \mathbf{u}_t - \mathbf{w}_t^n \right\| \tag{4.71}$$

$$\Psi(\mathbf{w}_t^n) - \Psi(\mathbf{u}_t) \leqslant \frac{\left\| \mathbf{d}_{t-1}^n \right\|_*}{(t-1)} \left\| \mathbf{u}_t - \mathbf{w}_t^n \right\| \tag{4.72}$$

$$\Psi(\mathbf{w}_t^n) - \Psi(\mathbf{u}_t) \leqslant \frac{G(t-1)}{(t-1)} \left\| \mathbf{u}_t - \mathbf{w}_t^n \right\| \tag{4.73}$$

$$\Psi(\mathbf{w}_t^n) - \Psi(\mathbf{u}_t) \leqslant G \left\| \mathbf{u}_t - \mathbf{w}_t^n \right\| \tag{4.74}$$

On line (4.73), we have used the fact, shown in (4.61), that $\left\| \mathbf{d}_{t-1}^n \right\|_* \leqslant G(t-1)$ for all $t \geqslant 2$, $n \in [N]$.

Thus:

$$\frac{1}{N} \left[\sum_{t \in [T]} \sum_{n \in [N]} \left[\ell(\mathbf{w}_t^n, \mathbf{z}_t^n) - \ell(\mathbf{u}_t, \mathbf{z}_t^n) \right] + \sum_{t \in [T]} \sum_{n \in [N]} \left[\Psi(\mathbf{w}_t^n) - \Psi(\mathbf{u}_t) \right] \right] \tag{4.75}$$

$$\leqslant \frac{2G}{N} \sum_{t \in [T]} \sum_{n \in [N]} \left\| \mathbf{u}_t - \mathbf{w}_t^n \right\| \tag{4.76}$$

$$\leqslant \frac{2G}{N} \sum_{t \in [T]} \sum_{n \in [N]} \frac{D_{PTG}}{\lambda t} \tag{4.77}$$

$$\leqslant 2GD_{PTG}(1 + \log T) \tag{4.78}$$

On line (4.77), we used (4.43). Putting (4.66) together with (4.78) and Lemma 1 completes the lemma.

□

4.3.2.4 Bounding Regret and Expected Cost

To move from Lemmas 1 and 2 to concrete bounds on run-time we need to bound the quantity D_{PTG}, for gradient bound G, and communication sequence $(P(t))_{t\in[T]}$. We will consider two possibilities for the communication pattern. The first is a fixed communication pattern, in which the communication matrix is the same for each iteration. The second is a stochastic communication pattern, in which the communication matrix can vary over iterations, being drawn from a fixed distribution.

4.3.2.4.1 Fixed Communication Pattern We first consider the case in which the communication matrix is fixed. Suppose we have some fixed matrix $N \times N$ matrix P such that, $P = P(t)$ for all $t \in [T]$. And, suppose that the $\mathbf{d}_t^n$ are updated according to (4.19). In this case, Duchi et al. (2012) show that:

$$\|\mathbf{e}_t - \mathbf{d}_t^n\|_* \leqslant 2G\frac{\log(T\sqrt{N})}{1-\sigma_2(P)} + 3G \tag{4.79}$$

We will review the arguments for this bound in §5. This leads immediately to the following theorem bounding regret in the case of a fixed communication pattern.

Theorem 5. *Suppose that the conditions of Lemma 1 hold. And, suppose that there exists a doubly stochastic $N \times N$ matrix P such that $P = P(t)$ for each $t \in [T]$. Then, the regret of the central sequence $(\mathbf{u}_t)_{t\in[T]}$ produced by Algorithm 9, when run on the index sequence $\mathbf{z}_T^N$, is bounded by:*

$$R_{\mathbf{z}_T^N,\Theta}((\mathbf{u}_t)_{t\in[T]}) \leqslant O\left(\frac{G^2\log T\log(T\sqrt{N})}{1-\sigma_2(P)}\right)$$

The expected cost of the average central vector $\bar{\mathbf{u}}_T$ is bounded by:

$$\mathbb{E}_{\mathbf{z}_T^N}[\Theta(\bar{\mathbf{u}}_T)] - \Theta(\mathbf{w}^*) \leqslant O\left(\frac{G^2\log T\log(T\sqrt{N})}{T(1-\sigma_2(P))}\right)$$

The regret of the local sequence $(\mathbf{w}_t^n)_{t\in[T]}$ for any $n \in [N]$ is bounded by:

$$R_{\mathbf{z}_T^N,\Theta}((\mathbf{w}_t^n)_{t\in[T]}) \leqslant O\left(\frac{G^2\log T\log(T\sqrt{N})}{1-\sigma_2(P)}\right)$$

The expected cost of the average local vector $\bar{\mathbf{w}}_T^n$ for any $n \in [N]$ is bounded by:

$$\mathbb{E}_{\mathbf{z}_T^N}[\Theta(\bar{\mathbf{w}}_T^n)] - \Theta(\mathbf{w}^*) \leqslant O\left(\frac{G^2\log T\log(T\sqrt{N})}{T(1-\sigma_2(P))}\right)$$

4.3.2.4.2 Stochastic Communication Pattern A fixed communication pattern may be undesirable or impossible. We now allow a sequence of communication matrices $(P(t))_{t\in[T]}$, possibly each distinct. Suppose that the communication matrices $P(t)$ are treated as draws of a random matrix, according to a fixed distribution, such that each $P(t)$ being doubly stochastic and $N \times N$. Note that, in this case, $\mathbb{E}_{P(t)}[P(t)]$ is also doubly stochastic. Suppose that the $\mathbf{d}_t^n$ are updated according to (4.19). In this case, Duchi et al. (2012) show that, with probability $1 - \frac{1}{T}$, we have:

$$\max_{t\in[T]} \max_{n\in[N]} \|\mathbf{e}_t - \mathbf{d}_t^n\|_* \leqslant \frac{6G\log(T^2N)}{1-\lambda_2(\mathbb{E}_C[(P(t))^T P(t)])} + \frac{G}{T\sqrt{N}} + 2G \tag{4.80}$$

This leads immediately to the following theorem bounding regret in the case of a stochastic communication pattern.

Theorem 6. *Suppose that the conditions of Lemma 1 hold. And, suppose that for each $t \in [T]$, P_t is a doubly stochastic $N \times N$ matrix. Then, with probability $1 - \frac{1}{T}$, the following hold. The regret of the central sequence $(\mathbf{u}_t)_{t\in[T]}$ produced by Algorithm 9, when run on the index sequence $\mathbf{z}_T^N$, is bounded by:*

$$R_{\mathbf{z}_T^N,\Theta}((\mathbf{u}_t)_{t\in[T]}) \leqslant O\left(\frac{G^2\log T\log(T^2N)}{1-\lambda_2(\mathbb{E}_{P(t)}[(P(t))^T P(t)])}\right)$$

The expected cost of average central vector $\bar{\mathbf{u}}_T$ is bounded by:

$$\mathbb{E}_{\mathbf{z}_T^N}[\Theta(\bar{\mathbf{u}}_T)] - \Theta(\mathbf{w}^*) \leqslant O\left(\frac{G^2\log T\log(T^2N)}{T(1-\lambda_2(\mathbb{E}_{P(t)}[(P(t))^T P(t)]))}\right)$$

The regret of the local sequence $(\mathbf{w}_t^n)_{t\in[T]}$, for any $n \in [N]$, is bounded by:

$$R_{\mathbf{z}_T^N,\Theta}((\mathbf{w}_t^n)_{t\in[T]}) \leqslant O\left(\frac{G^2\log T\log(T^2N)}{1-\lambda_2(\mathbb{E}_{P(t)}[(P(t))^T P(t)])}\right)$$

The expected cost of average local vector $\bar{\mathbf{w}}_T^n$ for any $n \in [N]$, is bounded by:

$$\mathbb{E}_{\mathbf{z}_T^N}[\Theta(\bar{\mathbf{w}}_T^n)] - \Theta(\mathbf{w}^*) \leqslant O\left(\frac{G^2\log T\log(T^2N)}{T(1-\lambda_2(\mathbb{E}_{P(t)}[(P(t))^T P(t)]))}\right)$$

4.3.3 Comparison with Past Work

Duchi et al. (2012) Duchi et al. (2011b, 2012) discuss several optimization scenarios, each of which deals with optimization of general convex functions, and none of which

can exploit the special structure of strongly regularized functions. We will discuss the variant of the algorithm (from Duchi et al. (2011b)) most similar to Algorithm 9, where they present an algorithm to optimize a stochastic function regularized by a convex regularization function. This task is exactly like (4.5), except that the regularizer is not assumed to be strongly convex.

Let $\Upsilon : \mathbb{R}^n \to \mathbb{R}$ be a known closed, convex function. Now, we can use Υ as a (not necessarily strongly convex) regularizer, to create the following stochastic convex objective, in the form of (4.4):

$$\Theta_{\ell,(Z_n)_{n\in[N]},\Upsilon}(\mathbf{w}) = \frac{1}{N}\sum_{n\in[N]} \theta_{\ell,Z_n,\Upsilon}(\mathbf{w}) \tag{4.81}$$

$$= \Upsilon(\mathbf{w}) + \frac{1}{N}\sum_{n\in[N]} \mathbb{E}_z\left[\ell(\mathbf{w};z)\right] \tag{4.82}$$

The associated optimization task is:

$$\arg\min_{\mathbf{w}\in\mathcal{W}} \left\{\Theta_{\ell,(Z_n)_{n\in[N]},\Upsilon}(\mathbf{w})\right\} \tag{4.83}$$

As a solution to (4.83), Duchi et al. (2011b) propose an algorithm just like Algorithm 9, except that it uses the following projection rule instead of (4.10). Let $\psi(\mathbf{w}) : \mathbb{R}^n \to \mathbb{R}$ be a 1-strongly convex function with respect to some norm $\|.\|$, and assume $\arg\min_{\mathbf{w}\in\mathcal{W}} \psi(\mathbf{w}) \in \arg\min_{\mathbf{w}\in\mathcal{W}} \Upsilon(\mathbf{w})$. Then:

$$\mathbf{w}_{t+1} = \arg\min_{\mathbf{w}\in\mathcal{W}} \{\langle \mathbf{d}_t, \mathbf{w}\rangle + t\Upsilon(\mathbf{w}) + \beta_t\psi(\mathbf{w})\} \tag{4.84}$$

Here, $(\beta_t)_{t\in[T]}$ is a non-decreasing sequence of positive real values, specified as input to the algorithm. In this case, Duchi et al. (2011b) show that the centralized sequence $(\mathbf{u}_t)_{t\in[T]}$, as defined in (4.23), (4.24), has regret bounded by:

$$R_{\mathbf{z}_T^N,\Theta}((\mathbf{u}_t)_{t\in[T]}) \leqslant \beta_T\psi(\mathbf{w}^*) + \frac{G^2}{2}\sum_{t\in[T]}\frac{1}{\beta_{t-1}} + 2GD_{PTG}\sum_{t\in[T]}\frac{1}{\beta_t} \tag{4.85}$$

Here, G and D_{PTG} have the same meaning as in Lemma 1. Duchi et al. (2011b) do not discuss how to bound the regret of the local sequences $(\mathbf{w}_t^n)_{t\in[T]}$.

Compared to this work, our work offers three improvements:

1. The optimal rate for (4.85) is achieved using $\beta_t = \gamma\sqrt{t}$ for $t \in [T]$ and fixed $\gamma > 0$. In this case, we have convergence in expectation of the central sequence $(\mathbf{u}_t)_{t\in[T]}$ bounded as $\mathbb{E}\left[\Theta(\bar{\mathbf{u}}_T)\right] - \Theta(\mathbf{w}^*) \leqslant O(\frac{GD_{PTG}}{\sqrt{T}})$. This is in contrast to our bound on the regret of the central sequence, which is asymptotically lower at $\mathbb{E}\left[\Theta(\bar{\mathbf{u}}_T)\right] - \Theta(\mathbf{w}^*) \leqslant O(\frac{GD_{PTG}\log T}{T})$.

Algorithm	Convergence Rate
Duchi et al. (2012)	$O(\frac{HD_{PTG}}{\lambda\sqrt{T}})$
This work	$O(\frac{GD_{PTG}\log T}{\lambda T})$
Tsianos & Rabbat (2012)	$O(\frac{HD_{PTG}}{\lambda T})$

Table 4.2: A comparison of regret bounds and stochastic optimization convergence rate bounds for three distributed algorithms. H bounds regularized gradients, G bounds unregularized gradients, D_{PTG} bounds the contribution of the network to the error, λ is the convexity parameter, and T counts the number of iterations.

2. The projection rule (4.84) requires the setting of the parameters $(\beta_t)_{t\in[T]}$. This should be done with knowledge of the function being optimized (Duchi et al., 2011b), which might have to be guessed. In contrast, our projection rule (4.10) does not require the setting of any parameter values, and so is much more user-friendly in practice.

3. Duchi et al. (2011b) do not discuss, in the context of composite functions, how to bound the convergence of each local sequence $(\mathbf{w}_t^n)_{t\in[T]}$. They only bound the convergence of the central sequence (which we have called $(\mathbf{u}_t)_{t\in[T]}$). We analyzed the convergence of each local sequence $(\mathbf{w}_t^n)_{t\in[T]}$ in Lemma 2, to show that convergence in expectation is bounded by $\mathbb{E}\left[\Theta(\bar{\mathbf{w}}_T^n)\right] - \Theta(\mathbf{w}^*) \leqslant O(\frac{GD_{PTG}\log T}{T})$. The proof techniques of Lemma 2 are significantly different from any of those used in Duchi et al. (2011b).

Tsianos & Rabbat (2012) Tsianos & Rabbat (2012) give a Hazan & Kale (2011)-style algorithm (see §4.2.3) for distributed optimization of strongly convex functions, which also uses ideas of Duchi et al. (2012) to bound the contribution of the network to the error. Table 4.2 compares the convergence rates of Tsianos & Rabbat (2012), Duchi et al. (2012) and the present work. The difference between the algorithm of Tsianos & Rabbat (2012) and ours is directly analogous to the difference between the algorithm of Hazan & Kale (2011) and that of Xiao (2010). Practically, the Hazan & Kale (2011) and Xiao (2010) algorithms perform roughly the same in our experiments (§5.8.1). The online-to-batch conversion in our convergence proof results in the non-optimal factor of $\log T$. On the other hand, our regularized dual averaging algorithm has dependence on G to bound sub-gradients, rather than H. As noted in §4.2.3, suggestions have been made that point the way towards modifying known algorithms to achieve the optimal

$O(1/T)$-like rate (Shamir, 2011; Lacoste-Julien et al., 2012; Chen et al., 2012). We leave the question of modifying our algorithm to achieve the optimal rate to future work. An important reason that we chose to work within the framework of Duchi et al. (2012) instead of Tsianos & Rabbat (2012) is that the former makes clear the effect of using stochastic communication matrices, which are central for the analysis of §5, while the latter, as far as we are aware, does not.

4.4 Conclusion

We have have presented a novel variant of the distributed dual averaging algorithm for optimization of strongly regularized stochastic functions. This algorithm improves over the previous state-of-the-art in distributed dual averaging algorithms (Duchi et al., 2011b, 2012) for the case of optimizing distributed stochastic functions with strongly convex regularizers in three ways. First, the regret bound is tighter. Second, we obviate the need to give the step-size schedule $(\beta_t)_{t\in[T]}$ required to optimize a composite objective in the previous work (Duchi et al., 2011b). Finally, we show how to bound the regret and cost for each local sequence $(\mathbf{w}_t^n)_{t\in[T]}$, which is not shown in Duchi et al. (2011b), and which is used in §5.

Chapter 5

A Markov Chain Mixing Approach to Understanding Iterative Parameter Mixing

This chapter presents new theoretical and empirical results on the distinction between the *iterative parameter mixing* and single-mixture strategies of distributed optimization discussed in §1. Based on the Markov Chain-inspired work of Duchi et al. (2012), we show that the IPM version of the distributed dual averaging algorithm of the last chapter constitutes a convergent solution to the distributed optimization problem, (1.2), repeated here:

$$\arg\min_{\mathbf{w}\in\mathcal{W}} \frac{1}{N} \sum_{n\in[N]} f_n(\mathbf{w}) \tag{1.2}$$

We demonstrate the convergence of this method, and compare it to the single-mixture method, for which, on the basis of work by Duchi et al. (2012), it can be shown that there exist functions for which single-mixture dual averaging will not converge.

This theoretical perspective is novel because, to our knowledge, past work on the distinction between IPM and single-mixture has focused only on reporting test-time results. Our analysis suggests suggests that the root of the distinction in test-time performance may be in the ability to optimize the training set. We present experimental results which agree with this analysis, showing that for an average regularized SVM-loss objective, IPM results in a better training objective value than does single-mixture optimization.

Our novel perspective on IPM training supports the use of the non-smooth linear

structured SVM training objective (2.28), repeated here:

$$\arg\min_{\mathbf{w}} \frac{\lambda}{2} \|\mathbf{w}\|_2^2 + \frac{1}{M} \sum_{m \in [M]} \ell_{\text{SVM}}(\mathbf{w}; (\mathbf{x}_m, \mathbf{y}_m)) \tag{2.28}$$

To our knowledge this is the first theoretical justification for the IPM, as opposed to the single-mixture distribution algorithm, beyond the perceptron-based work of McDonald et al. (2010), and is more general in that it is a general result about convex optimization, which includes the perceptron as a special case. We show the benefit of our more general analysis by demonstrating that distributively SVM-trained structured-prediction models for NLP outperform those trained using the perceptron, thereby improving the state-of-the-art in this area. Also, testing three structured-prediction tasks, on three data sets, and on a wide range of network sizes, the research of this chapter investigates the presence of a network speed-up due to multi-core optimization (showing a speed-up in some cases), as well as the effect of communication frequency, and the usefulness of iterate averaging (Collins, 2002) in the distributed setting.

5.1 Notation and Relevant Linear Algebra

Before beginning with the chapter, we briefly review some concepts from linear algebra and introduce some notation that will be used in the sequel.

Where A is any $N \times M$ rectangular matrix, $A_{i,j}$, with $i \in [N]$ and $j \in [M]$ is the entry of A in the i'th row, j'th column. Also, we use $\mathbf{1}_N$ to denote the column vector whose entries are all equal to 1. We use $\mathbf{I}_N$ to denote the $N \times N$ **identity matrix** (i.e., that matrix with 1's on the main diagonal and 0's elsewhere):

$$\mathbf{I}_N = \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ & & \vdots & & \\ 0 & 0 & \cdots & 1 & 0 \\ 0 & 0 & \cdots & 0 & 1 \end{bmatrix} \tag{5.1}$$

We use $\mathbf{U}_N$ to denote the **uniform matrix**, which is a $N \times N$ matrix in which each entry

is $\frac{1}{N}$, i.e., $\frac{1}{N}\mathbf{1}^T\mathbf{1}$:

$$\mathbf{U}_N = \begin{bmatrix} \frac{1}{N} & \frac{1}{N} & \cdots & \frac{1}{N} & \frac{1}{N} \\ \frac{1}{N} & \frac{1}{N} & \cdots & \frac{1}{N} & \frac{1}{N} \\ & & \vdots & & \\ \frac{1}{N} & \frac{1}{N} & \cdots & \frac{1}{N} & \frac{1}{N} \\ \frac{1}{N} & \frac{1}{N} & \cdots & \frac{1}{N} & \frac{1}{N} \end{bmatrix} \tag{5.2}$$

Also, let Δ_N be the N-dimensional simplex, which is the set of all N-dimensional column vectors whose elements form a probability distribution. The sub-script on any of these objects might be omitted if the dimensions are clear from context.

For a square matrix A, an **eigenvalue** and an **eigenvector** are a scalar $\lambda \in \mathbb{R}$ and a non-zero vector $\mathbf{v} \in \mathbb{R}^D$ such that:

$$A\mathbf{v} = \lambda\mathbf{v} \tag{5.3}$$

The eigenvectors of A are precisely those vectors $\mathbf{v}$ whose images under A are parallel to their original forms. The eigenvectors of A are orthogonal if and only if A is symmetric. For any real-valued $m \times n$ rectangular matrix B, the **singular values** of B are the non-negative square roots of the square matrix B^TB. Singular values have many interesting properties, which are beyond the scope here. However, we do mention that the singular values play a central role in the well-known singular value decomposition of an arbitrary rectangular matrix B, and this decomposition plays an important role in many branches of statistics and machine learning.

We can order the eigenvalues of a matrix from largest to smallest, writing $\lambda_1(A) \geqslant \cdots \geqslant \lambda_N(A)$. Similarly, we can order the singular values, writing $\sigma_1(A) \geqslant \cdots \geqslant \sigma_N(A)$. Let $\rho(A) = \max\{|\lambda| : \lambda \text{ is an eigenvalue of } A\}$. For a real-valued matrix A, it is possible to show that:

$$\max_{\|\mathbf{x}\|_2=1} \|A\mathbf{x}\|_2 = \sqrt{\rho(A^TA)} \tag{5.4}$$

A **column stochastic** matrix is a square $N \times N$ matrix with only non-negative entries, where the entries in each column sum to 1 (i.e., constitute a probability distribution):

$$\sum_{i \in [N]} P_{i,j} = 1, \forall j \in [N] \tag{5.5}$$

And, we recall from §4 that a doubly stochastic matrix P is a matrix with only non-negative entries in which the entries of each column and each row sum 1:

$$\sum_{i \in [N]} P_{i,j} = 1, \forall j \in [N] \text{ and } \sum_{j \in [N]} P_{i,j} = 1, \forall i \in [N] \tag{5.6}$$

The largest eigenvalue of a doubly stochastic matrix is always 1 (Horn & Johnson, 1985). And, the square of a doubly stochastic matrix is also doubly stochastic (Horn & Johnson, 1985).

5.2 The IPM and Single-Mixture Optimization Algorithms

5.2.1 Forms of IPM and Single-Mixture Algorithms

5.2.1.1 The Perceptron

The iterative parameter mixing strategy for the perceptron was reviewed as Algorithm 1. As we said in §1, this algorithm involves repeated rounds of perceptron training on each node, interspersed with communication at the end of each epoch between the nodes, in which the estimates of the optimal weight vector are averaged throughout the network. This algorithm is to be contrasted with the single-mixture perceptron, already shown as 2. As we said, the definition of the single-mixture perceptron algorithm is the combination of local perceptron training, parameters being averaged at the end.

5.2.1.2 General Form

We considered a general form for the IPM strategy in Algorithm 3 (page 14). We also considered a general form for the single-mixture strategy in Algorithm 4. The defining characteristics of the IPM strategy in general are:

- Each node performs local stochastic optimization.
- Nodes communicate after each pass through the training data by averaging estimates of the optimal weight vector (either using a primal or dual representation).

On the other hand, the defining characteristics of the single-mixture strategy are:

- Each node performs local stochastic optimization.
- Nodes communicate only at the end, and do not communicate during training.

Algorithm 10 Iterative Parameter Mixing for Dual Averaging

procedure IPM-DA(Data set shards: $(S_n)_{n\in[N]}, \Psi$)
  $\bar{\mathbf{d}}_0 = 0$, $\tau_{0,n} = 0$ for $n \in [N]$
  for $e = 1, \cdots, E$ **do** ▷ Do E epochs of training
    $\mathbf{d}_{n,e}, \tau_{n,e}, \bar{\mathbf{w}}_{n,e} = \mathit{OneEpochDA}_{n,e}(S_n, \bar{\mathbf{d}}_{e-1}, \tau_{e-1,n})$ ▷ N threads run in parallel
    $\bar{\mathbf{d}}_e = \frac{1}{N}\sum_{n\in[N]} \mathbf{s}_{n,e}$ ▷ communication by averaging
  return $\frac{1}{E}\frac{1}{N}\sum_{e\in[E]}\sum_{n\in[N]} \bar{\mathbf{w}}_{n,e}$ ▷ return average over iterates

procedure ONEEPOCHDA$_{n,e}(S, \mathbf{d}_0, \tau)$
  $\mathbf{w}_1 \leftarrow \arg\min_{\mathbf{w}\in\mathcal{W}} \left\{\langle \mathbf{d}_0^{n,e}, \mathbf{w}\rangle + \tau\Psi(\mathbf{w})\right\}$
  $M \leftarrow |S|$
  for $m = 1, \cdots, M$ **do**
    receive $\mathbf{g}_m^{n,e} \in \partial\ell(\mathbf{w}_t, z_m)$
    $\mathbf{d}_m^{n,e} \leftarrow \mathbf{d}_{m-1}^{n,e} + \mathbf{g}_m^{n,e}$
    $\mathbf{w}_{m+1} \leftarrow \arg\min_{\mathbf{w}\in\mathcal{W}} \left\{\langle \mathbf{d}_m^{n,e}, \mathbf{w}\rangle + (\tau + m)\Psi(\mathbf{w})\right\}$
  $\bar{\mathbf{w}} = \frac{1}{M}\sum_{m\in[M]} \mathbf{w}_m$
  return $\mathbf{d}_m^{n,e}, \tau + m, \bar{\mathbf{w}}$

Algorithm 11 Single-Mixture Dual Averaging

procedure SM-DA(Data set shards: $(S_n)_{n\in[N]}, \Psi$)
  for $n = 1, \cdots, N$ **do** ▷ Each $n \in [N]$ nodes runs in parallel
    $\mathbf{d}_{0,n} = 0$, $\tau_{0,n} = 0$
    for $e = 1, \cdots, E$ **do** ▷ Do E epochs of training
      $\mathbf{d}_{n,e}, \tau_{n,e}, \bar{\mathbf{w}}_{n,e} = \mathit{OneEpochDA}(S_n, \mathbf{d}_{e-1,n}, \tau_{e-1,n})$
  return $\frac{1}{E}\frac{1}{N}\sum_{e\in[E]}\sum_{n\in[N]} \bar{\mathbf{w}}_{n,e}$ ▷ return average over iterates

5.2.1.3 Dual Averaging

Our analysis will focus on the distinction between IPM and single-mixture communication in the context of the *regularized distributed dual averaging* algorithm presented in §4. An iterative parameter mixing dual averaging algorithm is given in Algorithms 10, and a single-mixture dual averaging algorithm is shown in Algorithm 11. Both algorithms have been written to take as input N shards of data $(S_n)_{n\in[N]}$, and a regularization function Ψ.

We can see that Algorithms 10 and 11 can be seen as instances of the generic IPM and single-mixture template Algorithms 3 and 4, respectively. Here, the generic *OneEpochStochastic* specifically takes the form of the function *OneEpochDA*. The vector $\mathbf{s}_{n,e}$ of Algorithms 3 and 4, which constitutes node n's optimizer state takes the form of the pair $(\mathbf{d}_{t,n}, \tau_{t,n})$. This constitutes a dual representation of the optimal weight vector. In §5.5.1 we will verify that Algorithms 10 and 11 are indeed forms of the distributed dual averaging algorithm of Algorithm 9.

5.2.2 Past Work with the IPM Algorithm

5.2.2.1 Empirical

Distributed Training Studies McDonald et al. (2010) investigate the training of a non-projective dependency parser for Czech (with about 70,000 training examples, Hajic et al. (2004)) and a named-entity recognizer (with about 14,000 training examples, Tjong Kim Sang & De Meulder (2003)). They find that distributed training allows for faster training times than sequential training, but do not report the factor of the speed-up. And, they find that only IPM, but not single-mixture or sub-sampling, can reach the same level of accuracy as the sequential solution. They also report, for named-entity recognition experiments, that performance peaks at about a network size of 25-30 nodes. Hall et al. (2010) investigate the training of a web-site click-through model, on training set sizes of about 370 million examples, in one experiment, and 1.6 billion in another. They compare sequential SGD, IPM SGD, and distributed gradient optimization for training with the maximum entropy objective, and compare this to IPM for the perceptron. Hall et al. (2010) find very large improvements for distributed training using IPM SGD compared to single-core SGD, reporting a speed-up factor of about 75. They also show that IPM SGD training returns better models much more quickly than the batch distributed gradient method. Simianer et al. (2012) look at SGD training of a

synchronous CFG translation system (Chiang, 2007) with a perceptron-like loss. They measure test-set performance and find that IPM-based optimization strategies allow optimal test-set performance, while single-mixture SGD does not. Simianer et al. (2012) do not, however, report training times.

Shared Memory Studies At least three additional studies have compared IPM training to shared-memory parallel strategies. (It is important to reiterate, here, that shared-memory parallel programming is more permissive, for a given number of cores, because memory is shared between cores, and so better performance is to be expected.) Chiang (2012) advocates an asynchronous margin-based training algorithm, showing it out-performs IPM training. Zhao & Huang (2013) study and advocate a parallel mini-batch form of structured perceptron training. Here, a collection of nodes on a shared memory machine process each mini-batch in parallel (see §3.4). Zhao & Huang (2013) compare this parallelization technique to IPM, in the training of a shift-reduce dependency parser (Huang & Sagae, 2010), and show that the speed-up of IPM is limited to about 3 times faster, even as the number of cores is increased to 12, whereas they can achieve a speed-up of about 6 times using the parallel mini-batch approach. Chang et al. (2013) compare IPM for the perceptron to a shared-memory parallel SVM training algorithm for part-of-speech tagging and relation extraction, and find that their SVM implementation usually reaches better accuracies in less time in 16 core experiments. It is unclear, however, whether these differences are due to IPM training or the SVM-vs-perceptron difference. Chang et al. (2013) do not make clear how the performance of IPM training scales with the number of processors, but they do show that, for their shared-memory method, it is difficult to increase performance by adding more cores beyond about 8.

5.2.2.2 Theoretical

5.2.2.2.1 McDonald et al. (2010) *Their Analysis* McDonald et al. (2010) analyze the IPM perceptron training strategy and compare it to the single-mixture perceptron. They first prove the following negative result about the single-mixture distributed perceptron:

Theorem 7. *There exists a separable data set S, and a partition of S into* $(S_n)_{n\in[N]}$ *such that, though S is separable,* SM-PERCEPTRON *(the single-mixture perceptron algorithm of Algorithm 2) will not return a separating hyper-plane.*

This theorem is proven using an example of such a set, with 4 training instances and 2 shards. In contrast to the possibility that the single-mixture perceptron will find an existing separating hyper-plane, McDonald et al. (2010) also prove the following theorem:

Theorem 8. *Suppose R bounds the norms* $\|\Phi(y) - \Phi(\mathbf{y}_m)\|$ *for each* $(\mathbf{x}_m, \mathbf{y}_m) \in S$*, each* $y \in \mathcal{C}(\mathbf{x}_m)$*, and that there exists a* $\mathbf{w}_{opt}$*,* $\left\|\mathbf{w}_{opt}\right\| = 1$*, that separates S by a margin of* γ*. Let* $(S_n)_{n\in[N]}$ *be any partition of S into N shards. Let* $k_{n,t}$ *be the number of mistakes made by the n'th node on the t'th epoch of training. Then, the number of mistakes made by the* IPM-PERCEPTRON *algorithm will be bounded as:*

$$\frac{1}{N}\sum_{n\in[N]}\sum_{t\in[T]} k_{n,t} \leqslant \left(\frac{R}{\gamma}\right)^2$$

Recall from Theorem 1 that the number of mistakes for the ordinary sequential perceptron algorithm is bounded by $\left(\frac{R}{\gamma}\right)^2$. Thus, the bound on the number of mistakes made by the sequential perceptron algorithm and the average number of mistakes made by each node in the IPM-PERCEPTRON algorithm is the same. McDonald et al. (2010) show that, if one uses a more sophisticated averaging step instead of line 5 of Algorithm 1, where, instead of the uniform average $\bar{\mathbf{w}}_t = \frac{1}{N}\sum_{n\in[N]} \mathbf{w}_{t,n}$, components are weighted according to the number of mistakes made on each shard in the past round, one can show that the number of rounds of distributed training is also bounded, in the worse cast, by $\left(\frac{R}{\gamma}\right)^2$. Thus, in the worst case, the number of rounds of training required for sequential perceptron training, and the number of rounds required for distributed perceptron training, are predicted to be the same. But, the distributed training processes each epoch N time faster, ignoring communication cost. Thus, distributed perceptron training is predicted to provide a speed-up of a factor of around N, ignoring communication.

Discussion One problem with this style of analysis is that it relies on the assumption that the training data is separable. This is not a realistic assumption for real-life NLP data sets. In virtually all applications, the best that can hope for is that the loss on the training set will reach a fairly low value. A related problem is that what is analyzed is time to convergence, but what is measured, by McDonald et al. (2010) is test-set performance. Thus, what is analyzed and what is measured are different.

5.2.2.2.2 Duchi et al. (2012) Duchi et al. (2012) note that the IPM perceptron algorithm McDonald et al. (2010) algorithm is a special case of their distributed dual

averaging algorithm. They do not elaborate on this point. They do not analyze the IPM algorithm in particular, nor discuss predictions for IPM versus single-mixture communication, and do not experiment with either strategy. The analysis of this algorithm within the framework provided by Duchi et al. (2012), and the attending experiments, are the contribution of this thesis.

5.3 Markov Chains and Mixing Times

5.3.1 Markov Chains

Sampling from a distribution is a central problem for applications of statistical science. The goal of sampling is to draw elements from an output space Ω according to a probability distribution π, and **Monte Carlo** methods, in general, are computational techniques that make use of the ability to computationally generate uniformly distributed pseudo-random numbers in order to sample from, or simulate draws from, an arbitrary target distribution π (MacKay, 2003).[1]

Sometimes it is difficult to sample from π directly. There are various reasons why this might be so. It might be that π is not known. For example, one might know π only up to a multiplicative constant (MacKay, 2003).[2] Or, it might be that Ω is so large that is is not possible to even enumerate the space in order to compute the probability of each (MacKay, 2003; Levin et al., 2009). And, sometimes, both situations may obtain at once. In some cases in which it is not possible to sample from a distribution directly, it is still possible to specify the distribution indirectly, using a *Markov chain* (Guruswami, 2000; Levin et al., 2009). Such a situation obtains, for example, in the case of the *Ising Model* from statistical physics (MacKay, 2003).

A **Markov chain** over state space Ω is a sequence of random variables $(X_0, X_1, \cdots)$, each $X_t \in \Omega$, in which the value of X_{t+1} is independent of $(X_0, \cdots, X_{t-1})$ given X_t (Neal, 1993). That is:

$$Pr(X_{t+1} = \mathbf{y} \mid X_0 = \mathbf{x}_0, X_1 = \mathbf{x}_1, \cdots, X_{t-1} = \mathbf{x}_{t-1}, X_t = \mathbf{x}) = Pr(X_{t+1} = \mathbf{y} \mid X_t = \mathbf{x}) \tag{5.7}$$

The indices t are often viewed as representing "times" (Neal, 1993). A Markov chain can be specified using an initial distribution over states $\mu_0(\mathbf{x})$, and a transition distribu-

[1] From the ability to sample in this way, it is also straightforward to compute the expectation of some function $f : \Omega \to \mathbb{R}$ given π (MacKay, 2003).

[2] That is, we might have a function $f : \Omega \to \mathbb{R}$ such that, for some scalar Z, $f(\mathbf{x})/Z = \pi(\mathbf{x})$ for all $\mathbf{x} \in \Omega$, but Z is not known.

tion $Q_t(\mathbf{x}, \cdot)$ for each t and each $\mathbf{x}$. $Q_{t+1}(\mathbf{x}, \mathbf{y})$ is the probability that the $t+1$'th state is $\mathbf{y}$ if the t'th state is $\mathbf{x}$. A sequence generated in this way obeys the Markov property if and only if Q_t is chosen independently of $(\mathbf{x}_0, \cdots, \mathbf{x}_{t+1})$, e.g., before hand or at independently random. A (time-)**homogeneous Markov Chain** is one on which Q_t does not vary as a function of t. That is, for the given chain there exists some matrix Q such that $Q_t = Q$ for all rounds t, in which case:

$$Pr(X_{t+1} = \mathbf{y} \mid X_t = \mathbf{x}) = Q(\mathbf{x}, \mathbf{y}) \tag{5.8}$$

A homogeneous Markov chain can be conveniently expressed in matrix form. We can enumerate the finite set Ω, so that each $\mathbf{x} \in \Omega$ can be identified with an integer in $[1, \cdots, |\Omega|]$. Then, we can construct a matrix P such that the $\mathbf{x}$'th column of P contains the distribution $Q(\mathbf{x}, \cdot)$. That is, the $\mathbf{y}$'th row of the $\mathbf{x}$'th column contains $Q(\mathbf{x}, \mathbf{y})$. And, we can let $\mu_0 \in \Delta_{|\Omega|}$, the $|\Omega|$ entry column vector, specify an initial distribution over states.

Suppose we start a homogeneous Markov chain with initial distribution μ_0. That is, $Pr(X_0 = \mathbf{x}) = \mu_0(\mathbf{x})$. Now, suppose we want to know the probability distribution over states that the system will be in at time $t > 0$. That is, we want to know the distribution $Pr(X_t = \mathbf{y} \mid X_0 \sim \mu_0)$. Standard reasoning shows that μ_t, for $t > 0$, is equal to:

$$\mu_t = P^t \mu_0 \tag{5.9}$$

In certain cases, if P is chosen properly, the limiting distribution will be π itself. Thus, if π cannot be sampled from directly, one can sometimes sample from this distribution by instead sampling from $P^t \mu_0$.

In the interesting applications, $|\Omega|$ is very large, so the operation of raising the matrix P to the power of t is not feasible. Despite this apparent difficulty, the simulation of a Markov chain can be conducted at low computational cost, using a **random walk** (Levin et al., 2009). We simply sample state $X_0 = \mathbf{x}_0$ according to μ_0. Then, we for any $t > 0$, we sample state $\mathbf{x}_t$ according to $Pr(X_t \mid X_{t-1} = \mathbf{x}_{t-1})$ using the distribution $Q(\mathbf{x}_{t-1}, \cdot)$. Often, such a distribution is easy to compute.

5.3.2 Mixing and Mixing Times

We saw that, in a properly specified homogeneous Markov chain, after t iterations, $P^t \mu_0$ is equal to the distribution π that one hopes to sample from. A central question when sampling using a Markov chain concerns how long one must run the random

walk for until one is sampling from π. Though the random walk representation does not explicitly make use of the matrix representation of the Markov chain, the equivalence between the two means that this question can be studied from a matrix theoretic perspective.

In order to state some formal properties of Markov chains, we introduce the following definitions. A matrix P is called **irreducible** if, for any pair of states $\mathbf{x}, \mathbf{y} \in \Omega$, there exists an integer t such that $P^t_{\mathbf{x},\mathbf{y}} > 0$. This means that it is eventually possible to get from any state $\mathbf{x}$ to any other state $\mathbf{y}$, by taking enough steps, even if it is not possible in a single step. The matrix P is called **aperiodic** if $\gcd\{t : P^t(x,) > 0\} = 1$ for all $x \in \Omega$. The **total variation distance** between two probability distributions (over finite event spaces) is defined as:

$$\|\mu - \nu\|_{TV} = \max_{A \subseteq \Omega} |\mu(A) - \nu(A)| \tag{5.10}$$

Then, the following convergence theorems give the conditions under which we can guarantee the convergence of a homogeneous Markov chain to its stationary distribution:

Theorem 9. *Let P be the transition matrix of an irreducible Markov chain. Then there exists a probability distribution π on Ω such that $P\pi = \pi$.*

Theorem 10. *Suppose that P is irreducible and aperiodic, with stationary distribution π. Then, there exists an α, $0 < \alpha < 1$ such that:*

$$\max_{\mu_0 \in \Delta_{|\Omega|}} \left\| P^t \mu_0 - \pi \right\|_{TV} \leqslant \alpha^t$$

Given that convergence of an appropriately specified Markov to its stationary distribution is guaranteed, we can speak of the amount of time it takes for this to happen. The **mixing time** for a homogeneous Markov chain, and for a given tolerance ε, as:

$$\text{mixing time}(P, \varepsilon) = \min \left\{ t : \max_{\mu_0 \in \Delta_{|\Omega|}} \left\| P^t \mu_0 - \pi \right\|_{TV} \right\} \tag{5.11}$$

5.3.3 Stationary Distribution for a Doubly Stochastic Matrix

Recall that, in §4, we chose the communication matrix to be doubly stochastic, where, as noted, a doubly stochastic matrix is a matrix P with non-negative entries such that the entries in each row and in each column sum to 1:

$$\sum_{i \in [N]} P_{i,j} = 1 \text{ and } \sum_{j \in [N]} P_{i,j} = 1 \tag{5.12}$$

The relevance of choosing a doubly stochastic matrix is that the limiting distribution for any $N \times N$ doubly stochastic matrix is the uniform vector (Horn & Johnson, 1985):

$$\mathbf{1}/N = \begin{bmatrix} 1/N \\ \vdots \\ 1/N \end{bmatrix} \tag{5.13}$$

It is easy to verify that this is indeed the stationary distribution for any doubly stochastic matrix.[3]

And, how long does it take a Markov chain specified by doubly stochastic matrix P to mix? Duchi et al. (2012) show the following:

Lemma 3. *Suppose P is a doubly stochastic $N \times N$ matrix. Then:*

$$\max_{\mu_0 \in \Delta_N} \left\| P^t \mu_0 - \mathbf{1}/N \right\|_{TV} = \frac{1}{2} \max_{\mu_0 \in \Delta_N} \left\| P^t \mu_0 - \mathbf{1}/N \right\|_1 \leqslant \frac{1}{2} \sigma_2(P)^t \sqrt{N} \tag{5.14}$$

Here, as per §5.1, $\sigma_2(P)$ is the second largest singular value of P. Thus, we see that the mixing time of a chain specified by doubly stochastic matrix P chain is closely related to the singular value spectrum of P. The relationship between mixing time and spectral properties holds more generally (Guruswami, 2000; Levin et al., 2009). In §5.4 we will see that the convergence of distributed dual averaging algorithms is closely related to the mixing of an implied Markov chain, and thus the spectral properties of the matrix describing communication in the network is the crucial property for determining convergence of the network of optimizers.

5.4 Convergence Rates for Decentralized Networks

In this section, we will show the connection between the mixing of Markov chains discussed in §5.3, and the convergence results of Duchi et al. (2012), reviewed in (4). We review these results to set up the analysis of §5.5, but is important to note that, in this section, we are only reviewing the work of Duchi et al. (2012), and are not providing any novel analysis of our own.

[3] Assume that P is doubly stochastic. Since the limiting distribution is unique, we need only show that $P\mathbf{1}/N = \mathbf{1}/N$. This is so, as the i'th entry of $P\mathbf{1}/N$ is:

$$\sum_{j \in [N]} \frac{1}{N} P_{i,j} = \frac{1}{N} \sum_{j \in [N]} P_{i,j} = \frac{1}{N} 1 = \frac{1}{N}$$

5.4.1 Fixed Communication Pattern

We saw in Lemma 1 that the bound on the centralized regret of the regularized distributed dual averaging algorithm can be bounded as a function of G, which bounds the sub-gradients received by the nodes, T, which is the number of rounds the algorithm has been run for, λ, which is the convexity parameter of the optimization function, and D_{PTG}, which is a bound on the quantity $\|\mathbf{e}_t - \mathbf{d}_t^n\|_*$ for all $t \in [T]$, with associated $(P(t))_{t\in[T]}$, and all $n \in [N]$. Combining Lemma 1 with Theorem 4, we have the following bound on the expected distance between $\bar{\mathbf{u}}_T$, the auxiliary central sequence that we saw allowed us to bound all quantities of interest, and $\mathbf{w}^*$, the optimal weight vector sought:

$$\mathbb{E}_{\mathbf{z}_T^N}\left[\Phi(\bar{\mathbf{u}}_T)\right] - \Phi(\mathbf{w}^*) \leqslant \frac{(6+\log T)}{\lambda T}\left(\frac{G^2}{2} + 2GD_{PTG}\right) \tag{5.15}$$

The only quantity here that depends on the network is D_{PTG}. As we saw, bounding $\|\mathbf{e}_t - \mathbf{d}_t^n\|_*$ is they key to bounding the error of the solution returned by the network, because the regret of the central sequence $\mathbf{u}_t$ is a function of $\|\mathbf{u}_t - \mathbf{w}_t^n\|$, and this is, in turn, a function of $\|\mathbf{e}_t - \mathbf{d}_t^n\|_*$, as shown in (4.39), repeated here:

$$\|\mathbf{u}_t - \mathbf{w}_t^n\| \leqslant \frac{1}{\lambda(t-1)}\left\|\mathbf{e}_{t-1} - \mathbf{d}_{t-1}^n\right\|_* \tag{4.39}$$

So, how far apart can $\mathbf{e}_t$ and $\mathbf{d}_t^n$, for any $n \in [N]$ get? Well, recall that the nodes are communicating by combining their estimates of the optimal dual vector, using the following update rule:

$$\mathbf{d}_t^n \leftarrow \sum_{i\in[N]} P_{n,i}\mathbf{d}_{t-1}^i + \mathbf{g}_t^n \tag{5.16}$$

$$\tag{5.17}$$

Intuitively, gradient information received by some node $i \in [N]$ will eventually reach any other node $j \in [N]$ via this communication. Even if $P_{i,j}$ is 0, meaning i and j are not directly connected, if P is irreducible, all gradient information received by i will eventually reach j, and vice versa. This can be more clearly seen by expressing

the value of each $\mathbf{d}_t^n$ in a non-recursive form:

$$\mathbf{d}_t^n \leftarrow \sum_{i\in[N]} P_{n,i}\mathbf{d}_{t-1}^i + \mathbf{g}_t^n \tag{5.18}$$

$$= \sum_{i\in[N]} P_{n,i}\left[\sum_{j\in[N]} P_{i,j}\mathbf{d}_{t-2}^j + \mathbf{g}_{t-1}^i\right] + \mathbf{g}_t^n \tag{5.19}$$

$$= \sum_{j\in[N]}\left[\sum_{i\in[N]} P_{n,i}P_{i,j}\right]\mathbf{d}_{t-2}^j + \sum_{i\in[N]} P_{n,i}\mathbf{g}_{t-1}^i + \mathbf{g}_t^n \tag{5.20}$$

$$= \sum_{j\in[N]} P_{n,j}^2\mathbf{d}_{t-2}^j + \sum_{i\in[N]} P_{n,i}\mathbf{g}_{t-1}^i + \mathbf{g}_t^n \tag{5.21}$$

$$= \sum_{i\in[N]} P_{n,i}^2\mathbf{d}_{t-2}^i + \sum_{i\in[N]} P_{n,i}\mathbf{g}_{t-1}^i + \mathbf{g}_t^n \tag{5.22}$$

$$= \sum_{i\in[N]} P_{n,i}^3\mathbf{d}_{t-3}^i + \sum_{i\in[N]} P_{n,i}^2\mathbf{g}_{t-2}^i + \sum_{i\in[N]} P_{n,i}\mathbf{g}_{t-1}^i + \mathbf{g}_t^n \tag{5.23}$$

(5.21) follows by the definition of matrix multiplication,[4] and (5.23) shows the effect of repeating the expansion process just demonstrated.

Let $\Pi(t,s) = P^{t-s}$. That is, $\Pi(t,s)$ represents P raised to the power $t-s$. From what we have just seen, an inductive argument can be used to show that $\mathbf{d}_t^n$ can be written without recursion as:

$$\mathbf{d}_t^n = \left[\sum_{s\in[t-1]}\sum_{i\in[N]} \Pi(t,s)_{n,i}\mathbf{g}_s^i\right] + \mathbf{g}_t^n \tag{5.24}$$

Since P is doubly stochastic, where $\mathbf{U}$ is the uniform matrix, $P^t \to \mathbf{U}$ as $t \to \infty$. Thus, $\Pi(t,0) \to \mathbf{U}$, as $t \to \infty$, and $\Pi(t,t-1) = P$. Thus, intuitively, if t is much bigger than s, then $\Pi(t,s)$ is closer to $\mathbf{U}$. If t and s are close together, then $\Pi(t,s)$ is closer to P itself.

On the other hand, as for expressing $\mathbf{e}_t$, this can be written non-recursively as:

$$\mathbf{e}_t = \frac{1}{N}\sum_{n\in[N]} \mathbf{d}_t^n \tag{5.25}$$

$$= \frac{1}{N}\sum_{n\in[N]}\sum_{j\in[N]}\left[P_{n,j}^{t-1}\mathbf{d}_{t-1}^n + \mathbf{g}_t^n\right] \tag{5.26}$$

$$= \mathbf{e}_{t-1} + \frac{1}{N}\sum_{n\in[N]} \mathbf{g}_t^n \tag{5.27}$$

$$= \frac{1}{N}\sum_{s\in[t]}\sum_{n\in[N]} \mathbf{g}_s^n \tag{5.28}$$

[4] That is, if A is an $m \times n$ matrix, and B is an $n \times o$ matrix, $(AB)_{i,j} = \sum_{k\in[n]} A_{i,k}B_{k,j}$.

So, the difference between $\mathbf{e}_t$ and $\mathbf{d}_t^n$ can be written in non-recursive form as:

$$\mathbf{e}_t - \mathbf{d}_t^n \tag{5.29}$$

$$= \left[\frac{1}{N}\sum_{s\in[t]}\sum_{i\in[N]}\mathbf{g}_s^i\right] - \left[\left[\sum_{s\in[t-1]}\sum_{i\in[N]}\Pi(t,s)_{n,i}\mathbf{g}_s^i\right] + \mathbf{g}_t^n\right] \tag{5.30}$$

$$= \left[\sum_{s\in[t-1]}\sum_{i\in[N]}\left(\frac{1}{N}\mathbf{g}_s^i - \Pi(t,s)_{n,i}\mathbf{g}_s^i\right)\right] - \left[\frac{1}{N}\sum_{i\in[N]}\left(\mathbf{g}_t^i - \mathbf{g}_t^n\right)\right] \tag{5.31}$$

$$= \left[\sum_{s\in[t-1]}\sum_{i\in[N]}\mathbf{g}_s^i\left(\frac{1}{N} - \Pi(t,s)_{n,i}\right)\right] - \left[\frac{1}{N}\sum_{i\in[N]}\left(\mathbf{g}_t^i - \mathbf{g}_t^n\right)\right] \tag{5.32}$$

Let $\Pi(t,s)_i$ denote the i'th column of $\Pi(t,s)$. Then, given the above and assuming $\|\mathbf{g}_t^n\|_* \leqslant G$ for all $t \in [T]$, $n \in [N]$:

$$\|\mathbf{e}_t - \mathbf{d}_t^n\|_* \tag{5.33}$$

$$\leqslant \left[\sum_{s\in[t-1]}\sum_{i\in[N]}\left\|\mathbf{g}_s^i\right\|_*\left(\frac{1}{N} - \Pi(t,s)_{n,i}\right)\right] - \left[\frac{1}{N}\sum_{i\in[N]}\left\|\mathbf{g}_t^i - \mathbf{g}_t^n\right\|_*\right] \tag{5.34}$$

$$\leqslant \sum_{s\in[t-1]} G\left\|\Pi(t,s)_i - \mathbf{1}/N\right\|_1 + 2G \tag{5.35}$$

Here, we see the utility of the perspective of the mixing of Markov chains. For those pairs of t and s that are far apart, $\Pi(t,s)_i$ is close to the uniform vector, meaning a gradient that was received by node i at time s, will have been fully communicated to most other nodes in the network. Thus, the number of iterations that we have to wait for $\|\Pi(t,s)_i - \mathbf{1}/N\|_1$ to be below some threshold $\varepsilon > 0$ is the same as the number of iterations that we have to wait for a Markov chain specified by matrix P to mix to an accuracy of ε.

Using this basic insight, some technical steps can be used to bound $\|\mathbf{e}_t - \mathbf{d}_t^n\|_*$. Lemma 3 implies that $\|\Pi(t,s)_i - \mathbf{1}/N\|_1 \leqslant \sqrt{N}\left[\sigma_2(P)\right]^{t-s}$. Suppose that T is the total number of rounds of training that we will do. Of course, $t \leqslant T$ for all round $t \in [T]$. If $t - s \geqslant \frac{\log(T\sqrt{N})}{\log\left[\sigma_2(P)^{-1}\right]}$, then:

$$\|\Pi(t,s)_i - \mathbf{1}/N\|_1 \leqslant \frac{1}{T} \tag{5.36}$$

Let $\hat{t} = \left\lceil \frac{\log(T\sqrt{N})}{\log\left[\sigma_2(P)^{-1}\right]} \right\rceil$. Thus, for all $s \leqslant t - \hat{t}$, $\sum_{s\in\hat{t}}\|\Pi(t,s)_i - \mathbf{1}/N\|_1 \leqslant 1$. And the number of s such that $s > t - \hat{t}$ is just $\hat{t} = \left\lceil \frac{\log(T\sqrt{N})}{\log\left[\sigma_2(P)^{-1}\right]} \right\rceil$. That is, for many values of s

(those less than or equal to $t-\hat{t}$), $\|\Pi(t,s)_i - \mathbf{1}/N\|_1$ is so small it contributes very little to the error. And, the number of s that do contribute a non-negligible amount to the error is bounded. These considerations can be combined to show (Duchi et al., 2012) that:

$$\|\mathbf{e}_t - \mathbf{d}_t^n\|_* \leqslant 2L \left\lceil \frac{\log(T\sqrt{N})}{\log\left[\sigma_2(P)^{-1}\right]} \right\rceil + 3L \tag{5.37}$$

$$\leqslant 2L \left\lceil \frac{\log(T\sqrt{N})}{1-\sigma_2(P)} \right\rceil + 3L \tag{5.38}$$

The last line uses the basic fact that $\log\left[\sigma_2(P)^{-1}\right] \geqslant 1-\sigma_2(P)$ (Duchi et al., 2012).

The important point here is that the spectral properties of the matrix determine the number of steps that it takes for information to reach the entire network. If the spectral gap is smaller, it will take information longer to propagate through the network, and there will be more past iterations for which $\|\Pi(t,s)_i - \mathbf{1}/N\|_1$ is significant.

5.4.2 Varying Communication Matrix

When the communication matrix can vary in between rounds, the analysis is somewhat changed, but carries on in the same spirit (Duchi et al., 2012). The principal difference is that, in the case of a stochastic communication matrix, one must speak about the spectral properties of the *expected* communication matrix, rather than a fixed communication matrix as in (5.38). Duchi et al.'s (2012) result here is probabilistic. With probability $1-\frac{1}{T}$:

$$\max_{t\in[T]} \max_{n\in[N]} \|\mathbf{e}_t - \mathbf{d}_t^n\| \leqslant \frac{6G\log(T^2N)}{1-\lambda_2(\mathbb{E}_{P(t)}\left[P(t)^T P(t)\right])} + \frac{G}{T\sqrt{N}} + 2G \tag{5.39}$$

In comparison with (5.38), this bound holds with high probability, rather than with certainty, and the effect of the network is characterized by $1-\lambda_2(\mathbb{E}\left[P^T P\right])$, which references the second *eigenvalue* (rather than singular value) of the *expected matrix*, $\mathbb{E}\left[P(t)^T P(t)\right]$.

5.5 Analyzing IPM and Single-Mixture Optimization

In this section, we cast the IPM and single-mixture optimization algorithms as instances of the DRDA algorithm, and then analyze the convergence of both. This gives a central result whose implications will be investigated in the experiments.

5.5.1 IPM and Single-Mixture as Distributed Dual Averaging

Let us now consider how the IPM and single-mixture versions of the dual averaging algorithm can be cast as instances of the DRDA algorithm.

Conversion of Indices Algorithms 10 and 11 proceed in epochs, by calling the function $OneEpochDA_{n,e}$, and the indexing on the dual averages and primal weights reflect this. For each node $n \in [N]$ and epoch $e \in [E]$, these algorithms produce a sequence of dual averages indexed as follows:

$$\mathbf{d}_1^{n,1}, \cdots, \mathbf{d}_M^{n,1}, \cdots, \mathbf{d}_1^{n,e}, \cdots, \mathbf{d}_M^{n,e}, \cdots, \mathbf{d}_1^{n,E}, \cdots, \mathbf{d}_M^{n,E} \tag{5.40}$$

And, it produces for each $n \in [N]$ a sequence of primal weights $\left(\left(\mathbf{w}_m^{n,e}\right)_{m\in[M]}\right)_{e\in[E]}$ with the same indices. The DRDA Algorithm 9 uses the indexing, for each $n \in [N]$:

$$\mathbf{d}_1^n, \mathbf{d}_2^n, \cdots, \mathbf{d}_T^n \tag{5.41}$$

Similarly, we have primal weights $(\mathbf{w}_t^n)_{t\in[T]}$.

To align the two algorithms, we will use the following, natural conversion of indices:

$$t_{e,m,M} = (e-1) * M + m \tag{5.42}$$

Using this conversion, we see that IPM and single-mixture dual averaging algorithms produce a sequence of dual averages and primal weights

$$\left((\mathbf{d}_t^n)_{t\in[T]}\right)_{n\in[N]} \text{ and } \left((\mathbf{w}_t^n)_{t\in[T]}\right)_{n\in[N]}$$

and we will show that this is within the framework allowed by the DRDA algorithm (Algorithm 9).

Return Value We briefly note that the return values of Algorithms 10 and 11 are:

$$\frac{1}{E}\frac{1}{N}\sum_{e\in[E]}\sum_{n\in[N]}\bar{\mathbf{w}}_{n,e} \tag{5.43}$$

Some simple algebraic manipulation shows that this is equivalent to the global return value of Algorithm 9:

$$\frac{1}{T}\frac{1}{N}\sum_{t\in[T]}\sum_{n\in[N]}\mathbf{w}_t^n \tag{5.44}$$

Thus, assuming that the dual and primal sequences produced by Algorithms 10 and 11 conform to those that would be generated by some instance of DRDA, then the output values will as well.

Epochs Through the Data versus Random Selection of Examples The IPM and single-mixture dual averaging algorithms both proceed in repeated epochs through the available training data. However, the DRDA algorithm of §4 is analyzed in terms of examples randomly selected on each iteration. This gap between theory and practice is a matter of choice. This gap would be closed if we were to simply select M examples at random from the available data on each call to *OneEpochDA*. However, although analyses of stochastic optimization algorithms usually proceed on the assumption that examples are randomly selected (e.g., Robbins & Monro, 1951; Polyak & Juditsky, 1992; Hazan et al., 2007; Hazan & Kale, 2011), it has been found to be better for test-set performance in practice to make repeated full passes through the training, perhaps with a random ordering (Shalev-Shwartz et al., 2011). Thus, we take the approach of repeated epochs through the data here.

Communication Patterns Behind IPM *and Single-Mixture* Communication for each round of the DRDA algorithm (Algorithm 9) is characterized by, for each round $t \in [T]$, a communication matrix $P(t)$. We now look at which what $P(t)$ must be set to in order to yield the IPM and single-mixture algorithms.

The common part of both the IPM and the single-mixture dual averaging algorithms is $\mathit{OneEpochDA}_{n,e}$, which is called by both. For each $n \in [N]$, $e \in [E]$, and all rounds $m \geqslant 2$, the update rule on line 6 is:

$$\mathbf{d}_m^{n,e} \leftarrow \mathbf{d}_{m-1}^{n,e} + \mathbf{g}_m^{n,e} \tag{5.45}$$

$$= \sum_{i \in [N]} \mathbf{I}_{n,i} \mathbf{d}_{m-1}^{n,e} + \mathbf{g}_m^{n,e} \tag{5.46}$$

Thus, to have N cores running $\mathit{OneEpochDA}_{n,e}$ over $M-1$ examples in the range $[2, \cdots, M]$ corresponds to running the DRDA algorithm with N nodes for $M-1$ rounds using $\mathbf{I}$ as the communication matrix.

The difference between the two algorithms, of course, is in the value of $\mathbf{d}_0^{n,e}$ used to begin each round of $\mathit{OneEpochDA}_{n,e}$. In the case of IPM dual averaging, the $\mathbf{d}_0^{n,e}$ given as input to each node n on epoch e is actually the average dual average over all nodes

from the previous epoch $e-1$. This means that, for each node n, and each $e \geqslant 1$:

$$\mathbf{d}_1^{n,e} \leftarrow \mathbf{d}_0^{n,e} + \mathbf{g}_1^{n,e} \tag{5.47}$$

$$= \sum_{i \in [N]} \frac{1}{N} \mathbf{d}_M^{i,e-1} + \mathbf{g}_1^{n,e} \tag{5.48}$$

$$= \sum_{i \in [N]} \mathbf{U}_{n,i} \mathbf{d}_0^{n,e} + \mathbf{g}_1^{n,e} \tag{5.49}$$

Thus, for each node $n \in [N]$ and each epoch $e \in [E]$, the first iteration $m = 1$ corresponds to the DRDA algorithm with communication matrix $P(t_{e,m,M}) = \mathbf{U}$, the uniform matrix.

In contrast, in the case of the single-mixture dual averaging algorithm, the first round for each node $n \in [N]$ and each $e \in [E]$ is:

$$\mathbf{d}_1^{n,e} \leftarrow \mathbf{d}_0^{n,e} + \mathbf{g}_1^{n,e} \tag{5.50}$$

$$= \mathbf{d}_M^{i,e-1} + \mathbf{g}_1^{n,e} \tag{5.51}$$

$$= \sum_{i \in [N]} \mathbf{I}_{n,i} \mathbf{d}_M^{i,e-1} + \mathbf{g}_1^{n,e} \tag{5.52}$$

Thus, for each node $n \in [N]$ and each epoch $e \in [E]$, the first iteration $m = 1$ corresponds to the DRDA algorithm with communication matrix $P(t_{e,1}) = \mathbf{I}$, the identity matrix.

So, the dual updates in both algorithms correspond to instances of the DRDA algorithm using a mixture of the communication matrices $\mathbf{I}$ and $\mathbf{U}$.

5.5.2 Spectral Analysis

We will now look at the spectra of eigenvalues and singular values for the identity matrix $\mathbf{I}_N$, the uniform matrix $\mathbf{U}_N$ and the matrix describing communication for the IPM distribution strategy. These results will be referred to in §5.5.3, in order to make predictions about the relationship between IPM and single-mixture dual averaging. Recall, from §5.1 we order the eigenvalues of a matrix from largest to smallest, writing $\lambda_1(A) \geqslant \cdots \geqslant \lambda_N(A)$, and do the same for the singular values, writing $\sigma_1(A) \geqslant \cdots \geqslant \sigma_N(A)$.

The Identity Matrix The eigenspectrum of the identity matrix $\mathbf{I}_N$ can be characterized as follows:

Theorem 11. *The eigenvalues of the $N \times N$ identity matrix $\mathbf{I}_N$ are:*

$$\lambda_1(\mathbf{I}_N) = 1, \lambda_2(\mathbf{I}_N) = 1, \cdots, \lambda_N(\mathbf{I}_N) = 1$$

The singular values of this matrix are:

$$\sigma_1(\mathbf{I}_N) = 1, \sigma_2(\mathbf{I}_N) = 1, \cdots, \sigma_N(\mathbf{I}_N) = 1$$

Proof. Self-evident. □

The Uniform Matrix The eigenspectrum of the identity matrix $\mathbf{I}_N$ can be characterized as follows:

Theorem 12. *The eigenvalues of the $N \times N$ uniform matrix $\mathbf{U}_N$ are:*

$$\lambda_1(\mathbf{I}_N) = 1, \lambda_2(\mathbf{I}_N) = 0, \cdots, \lambda_N(\mathbf{I}_N) = 0$$

The singular values of this matrix are:

$$\sigma_1(\mathbf{I}_N) = 1, \sigma_2(\mathbf{I}_N) = 0, \cdots, \sigma_N(\mathbf{I}_N) = 0$$

Proof. To verify this, let $\mathbf{u} = U\mathbf{v}$, where $\mathbf{v} \in \mathbb{R}^N$ is arbitrary. Let u_i be the i'th component of $\mathbf{u}$, and v_j the j'th component of $\mathbf{v}$. Then:

$$u_i = \sum_{j \in [N]} U_{i,j} v_j = \sum_{j \in [N]} \frac{1}{N} v_j \tag{5.53}$$

So, all components of $\mathbf{u} = U\mathbf{v}$ are identical, meaning $U\mathbf{v}$ is a multiple of $\mathbf{1}$, the all one's vector. Thus, we can only have $U\mathbf{v} = \lambda\mathbf{v}$ if $\mathbf{v}$ is parallel to $\mathbf{1}$. There are no other non-zero eigenvectors, and thus no other non-zero eigenvalues. Also, $\mathbf{U}^T\mathbf{U} = \mathbf{U}$, so the singular value spectrum is the same. □

Iterative Parameter Mixing Iterative parameter mixing, as we saw in §5.5.1, can be modelled as a mix of using the uniform matrix and using the identity matrix. We can view $P(t)$ as a random matrix, drawn according to the following probability distribution:

$$P(t) = \begin{cases} \mathbf{I}_N : \text{ with probability } \frac{M-1}{M} \\ \mathbf{U}_N : \text{ with probability } \frac{1}{M} \end{cases} \tag{5.54}$$

Of course, the value of $P(t)$ for any $t \in [T]$ is not actually random, since it is determined according to a deterministic and predictable schedule. However,this seems to us to be a good approximation since, whether communication rounds are chosen randomly or according to this schedule, the expected number of communication rounds in M DRDA rounds is 1.

Now, for the analysis of the next section, we will want to compute $\lambda_2(\mathbb{E}_{P(t)}\left[P(t)^T P(t)\right])$ according to this distribution. Let $\gamma_I = \frac{M-1}{M}$ and $\gamma_U = \frac{1}{M}$. From (5.54), we have that:

$$\mathbb{E}_{P(t)}\left[P(t)\right] = \gamma_I \mathbf{I}_N + \gamma_U \mathbf{U}_N \tag{5.55}$$

Since $\mathbf{I}$ and $\mathbf{U}$ are both symmetric, we have:

$$(\gamma_I \mathbf{I}_N + \gamma_U \mathbf{U}_N)^T = \gamma_I \mathbf{I}_N + \gamma_U \mathbf{U}_N \tag{5.56}$$

So, since $\mathbf{I}^2 = \mathbf{I}$ and $\mathbf{U}^2 = \mathbf{I}\mathbf{U} = \mathbf{U}$:

$$\begin{aligned}
&\mathbb{E}_{P(t)}\left[P(t)^T P(t)\right] && (5.57)\\
=&(\gamma_I \mathbf{I}_N + \gamma_U \mathbf{U}_N)^T (\gamma_I \mathbf{I}_N + \gamma_U \mathbf{U}_N) && (5.58)\\
=&(\gamma_I \mathbf{I}_N + \gamma_U \mathbf{U}_N)^2 && (5.59)\\
=&\gamma_I^2 \mathbf{I}_N^2 + 2\gamma_I\gamma_U \mathbf{I}_N\mathbf{U}_N + \gamma_U^2 \mathbf{U}_N^2 && (5.60)\\
=&\gamma_I^2 \mathbf{I}_N + (2\gamma_I\gamma_U + \gamma_U^2)\mathbf{U}_N && (5.61)
\end{aligned}$$

Theorem 13. *The eigenspectrum of* $Z_{\gamma_I,\gamma_U} \stackrel{\Delta}{=} \left[\gamma_I^2 \mathbf{I}_N + (2\gamma_I\gamma_U + \gamma_U^2)\mathbf{U}_N\right]$ *is:*

$$\lambda_1(Z_{\gamma_I,\gamma_U}) = 1$$

and

$$\lambda_n(Z_{\gamma_I,\gamma_U}) = \gamma_I^2, \forall n, 2 \leqslant n \leqslant N$$

That is, the largest eigenvalue is 1*, and all others are* $\gamma_I^2 = \left(\frac{M-1}{M}\right)^2$.

Proof. That the largest eigenvalue of Z_{γ_I,γ_U} is 1 follows from the fact that it is doubly stochastic (§5.1). To determine the other eigenvalues of Z_{γ_I,γ_U}, we can use Weyl's theorem (Horn & Johnson, 1985), used here as a lemma:

Lemma 4. *Suppose that A and B are two* $N \times N$ *symmetric matrices. As in the text, the eigenvalues of matrix A be written as* $\lambda_1(A) \geqslant \cdots \geqslant \lambda_N(A)$ *and likewise for B. Then, for each* $n \in [N]$*:*

$$\lambda_n(A) + \lambda_N(B) \leqslant \lambda_n(A+B) \leqslant \lambda_n(A) + \lambda_1(B)$$

Suppose $n \geqslant 2$. Let the A of Weyl's theorem be $(2\gamma_I\gamma_U + \gamma_U^2)U_N$ and let the B be $\gamma_I^2 I_N$. Then, we have $\lambda_2((2\gamma_I\gamma_U + \gamma_U^2)U_N^2) = 0$, and $\lambda_N(\gamma_I I_N) = \lambda_1(\gamma_I I) = \gamma_I^2$. Thus:

$$\begin{aligned}
\lambda_n(Z_{\gamma_I,\gamma_U}) &\geqslant \lambda_n((2\gamma_I\gamma_U + \gamma_U^2)U_N) + \lambda_N(\gamma_I^2 I_N) && (5.62)\\
&\geqslant 0 + \gamma_I^2 && (5.63)
\end{aligned}$$

And:

$$\lambda_n(Z_{\gamma_I,\gamma_U}) \leqslant \lambda_n(\gamma_U U_N) + \lambda_1(\gamma_I I_N) \tag{5.64}$$

$$\leqslant 0 + \gamma_I^2 \tag{5.65}$$

Thus, for all $n \geqslant 2$, $\lambda_n(Z_{\gamma_I,\gamma_U}) = \gamma_I^2$. □

5.5.3 Implications

Iterative Parameter Mixing The first conclusion we can draw from mathematical facts noted so far is that IPM dual averaging can indeed be proven to be convergent, in the sense that the expected cost of the output can be made arbitrarily close to zero, with arbitrarily high probability, by running the algorithm for long enough. We saw in Theorem 6 that, if communication in a network is characterized by the stochastic communication matrices $(P(t))_{t\in[T]}$, drawn according to a fixed distribution, then the expected cost of average local vector $\bar{\mathbf{w}}_T^n$ for some T, and any $n \in [N]$, when optimizing distributed stochastic objective $\Theta_{\ell,(Z_n)_{n\in[N]},\Psi}$, is bounded with probability $1 - \frac{1}{T}$ by:

$$\mathbb{E}_{\mathbf{z}_T^N}[\Theta(\bar{\mathbf{w}}_T^n)] - \Theta(\mathbf{w}^*) \leqslant O\left(\frac{\log T \log(T^2 N)}{T(1 - \lambda_2(\mathbb{E}_{P(t)}[(P(t))^T P(t)]))}\right) \tag{5.66}$$

Also, by the convexity of Θ,

$$\Theta\left(\frac{1}{N}\sum_{n\in[N]} \bar{\mathbf{w}}_T^n\right) \leqslant \frac{1}{N}\sum_{n\in[N]} \Theta(\bar{\mathbf{w}}_T^n), \tag{5.67}$$

(5.66) is also a bound on the error of the program output $\frac{1}{N}\sum_{n\in[N]} \bar{\mathbf{w}}_T^n$.

We saw in §5.5.2 that $\lambda_2(\mathbb{E}[P(t)^T P(t)]) = \left(\frac{M-1}{M}\right)^2$. And,

$$1 - \left(\frac{M-1}{M}\right)^2 \tag{5.68}$$

$$= 1 - \frac{M^2 - 2M + 1}{M^2} \tag{5.69}$$

$$= \frac{M^2 - M^2 + 2M - 1}{M^2} \tag{5.70}$$

$$= \frac{2M-1}{M^2} \tag{5.71}$$

Thus, $\frac{1}{1-\gamma_I^2} = O(M)$.

Thus, when running the IPM dual averaging algorithm over shards of size M, we have:

$$\mathbb{E}_{\mathbf{z}_T^N}[\Theta(\bar{\mathbf{w}}_T^n)] - \Theta(\mathbf{w}^*) \leqslant O\left(\frac{M \log T \log(T^2 N)}{T}\right), \tag{5.72}$$

with probability $1 - \frac{1}{T}$. In other words, the IPM algorithm is convergent.

Single-Mixture We saw in §4.3.2.4 that, if communication in a network is characterized by the fixed communication matrix P, then the expected cost of the average local vector $\bar{\mathbf{w}}_T^n$ for any $n \in [N]$, when optimizing $\Theta_{\ell,(Z_n)_{n\in[N]},\Psi}$, is bounded by:

$$\mathbb{E}_{\mathbf{z}_T^N}\left[\Theta(\bar{\mathbf{w}}_T^n)\right] - \Theta(\mathbf{w}^*) \leqslant O\left(\frac{\ln T \ln(T\sqrt{N})}{T(1-\sigma_2(P))}\right) \tag{5.73}$$

We have seen, as we said in §5.5.1, that single-mixture dual averaging corresponds to running the DRDA algorithm with communication on *every* round described by the identity matrix, $\mathbf{I}_N$. And, we saw that:

$$\sigma_2(\mathbf{I}_N) = 1 \tag{5.74}$$

$$1 - \sigma_2(\mathbf{I}_N) = 0 \tag{5.75}$$

In other words, no bound at all can be placed on convergence based on (5.73) using the $\mathbf{I}_N$ as the communication matrix, since the fraction $\frac{1}{1-\sigma_2(P)}$ diverges as $\sigma_2(P) \to 1$. From a Markov Chain perspective, the single-mixture algorithm corresponds to the specification of a Markov chain by a matrix that is not irreducible, for which the associated Markov chain will never mix.

In fact, a stronger statement than this can be made, because Duchi et al. (2012) show that the dependence on the $\frac{1}{1-\sigma_2(P)}$ is tight. That is, for any graph structure in which communication is characterized by matrix P, there exists a set of functions $(f_n)_{n\in[N]}$ such that convergence is *lower*-bounded by:

$$\Omega\left(\frac{1}{1-\sigma_2(P)}\right) \tag{5.76}$$

So, the single-mixture dual averaging algorithm, it is possible to construct an objective function such that the algorithm will never converge. This is analogous to the finding of McDonald et al. (2010) that there exist separable data sets that the single-mixture perceptron cannot separate.

Summary Thus, the points we have shown are:

- Convergence will eventually occur (with high probability) using IPM dual averaging.
- Convergence cannot be guaranteed for single-mixture dual averaging.

- Moreover, it is possible to constructive an objective function such that single-mixture dual averaging will never converge.

This gives us a novel perspective on the difference between the IPM and single-mixture distribution strategies. Namely, one is a convergent optimization algorithm, while the other is not. Past work has mostly focused on the difference in test-time performance between models trained using the two methods (see §5.2.2). However, we have shown that this difference might be explained by a difference in the ability of the two algorithms to optimize their training objective. This is something we will test for in §5.7. Indeed, we will see that test set performance of SVM-trained models is correlated with training objective value reach.

Discussion This bound is interesting and novel in that it demonstrates that IPM is a convergent optimization algorithm, while single-mixture, in general, is not. However, the bound (5.72), unfortunately, does not predict a speed-up for IPM due to parallelization. In fact, an $O(M)$ penalty is incurred, where M is the frequency of averaging between vectors. According to the usual interpretation of the IPM algorithm (McDonald et al., 2010; Hall et al., 2010; Simianer et al., 2012), we have averaged parameters across the network after each pass through the data. It is of course possible to let the frequency of averaging be different from the size of each data shard, averaging more or less frequently than after each pass through the data. If the frequency of averaging is less than N, the number of optimization cores, we would predict a speed-up. This is not the normal case, however.

Also, the bound is not sensitive to the similarity between the optimal vectors for each shard. Progress can be made more quickly if each shard is assumed to be identical, which is effectively what is assumed by Dekel et al. (2012), when the assume that N cores each repeatedly draw new examples from the exact same distribution. So, in this sense, our bound must be considered pessimistic, since it must account for *all* kinds of functions spread across N computers. Similarly, it is interesting that our IPM bound is not a function of the network size N. Suppose we split a fixed-size (but, perhaps large) data set into N shards. As N grows, each shard becomes more idiosyncratic and we might expect the distribution strategy to become less efficient. In §5.8, we see that, indeed, the efficiency of training on the fixed-size data sets investigated seems to decrease for larger network sizes, like 64, compared to network sizes like 8 and 16. Experiments in §5.8.5 show the effect of varying the averaging frequency, and shows that, as one might expect, more frequent averaging leads to better objective values in

less time.

5.5.4 Distributed Dual Averaging Compared to Other Frameworks for this Analysis

The framework in which we have conducted this analysis initially appeared in Duchi et al. (2011b, 2012). Clearly, this framework does provide a way to prove that the IPM algorithm is convergent. We might then ask whether there would be any better way. Tsianos & Rabbat (2012) give an algorithm for distributed optimization of a strongly convex function which, as we said in Table 4.2 of §4.3.3 has a slightly convergence bound (lower by a factor of $\log T$, where T is the number of iterations). This would certainly be an alternative direction for research, although, since they do not discuss the case of varying communication matrices, it was a choice between extending their paradigm or extending that of Duchi et al. (2012). As the experiments of this chapter show, the dual averaging algorithm performs as well as the Hazan & Kale (2011) algorithm on which Tsianos & Rabbat (2012) is based, despite having different theoretical analyses. Perhaps more refined dual averaging analysis will show that it too achieves $O(\frac{1}{\lambda T})$-like bounds. Also, the dual averaging framework is interesting because important ongoing work, such as the *AdaGrad* algorithm (Duchi et al., 2011a), which has been popular for using different step-sizes for different features depending on the number of times that feature has been seen, is happening in this framework.

5.6 Efficient Dual Average Updates and Averaging

The goal of optimization is to find an optimal weight $\mathbf{w}^* \in \mathbb{R}^D$. Suppose the number of non-zero components (otherwise known as the ℓ_0 size) of the gradients received by the perceptron or dual averaging algorithm is bounded by some integer d. Usually, D is much larger than d (i.e., $D \gg d$). For example, in the case of the parsers used below, D is on the order of millions, while d is on the order of hundreds or thousands. If gradient updates took time $\Theta(D)$, training would be far more inefficient than if these gradient updates were $\Theta(d)$. (In fact, training might even be impractical in such a case.)

In the case of the dual averaging algorithm, the update of the dual average,

$$\mathbf{d}_{t+1} = \mathbf{d}_t + \mathbf{g}_t,$$

is trivial to compute sparsely. And, the projection of the dual average into the primal

space,

$$\text{i.e., the solving of } \mathbf{w}_{t+1} \leftarrow \arg\min_{\mathbf{w}\in\mathcal{W}} \{\langle \mathbf{d}_t, \mathbf{w}\rangle + t\Psi(\mathbf{w})\},$$

is known to be sparse for many regularizers (Xiao, 2010), because (as discussed in §5.6.1) one only needs to compute those components of $\mathbf{w}_{t+1}$ if and when they are needed to compute the next gradient (see §5.6.1). However, we are not aware of any previously published or existing algorithm for efficiently (i.e. sparsely) computing the average primal vector over all iterates,

$$\frac{1}{T}\sum_{t\in[T]} \mathbf{w}_t,$$

during a run of the dual averaging algorithm. One difficulty, or at least novel feature of the problem, is that the sparsity requirement means that the primal weight $\mathbf{w}_t$ is never computed in full for any t.

We require such an efficient algorithm for computing the primal average in order to test the effects of averaging iterates in the experiment section. So, in §5.6.2, we explain our novel, efficient algorithm for averaging primal weights in the dual averaging algorithm. First, §5.6.1 reviews why (Nesterov, 2009; Xiao, 2010) updates are sparse in the ordinary dual averaging algorithm.

5.6.1 Efficient Gradient Updates

Ordinary gradient updates, like (5.77) or (5.78), are both $O(d)$ since both only involve updating the d components of $\mathbf{g}_t$ of either the primal weight $\mathbf{w}$ or dual average $\mathbf{d}$:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \mathbf{g}_t \tag{5.77}$$

$$\mathbf{d}_{t+1} = \mathbf{d}_t + \mathbf{g}_t \tag{5.78}$$

The non-trivial part for the dual averaging algorithm is the projection step that projects the dual average into the primal space. The projection step is:

$$\mathbf{w}_{t+1} \leftarrow \arg\min_{\mathbf{w}\in\mathcal{W}} \{\langle \mathbf{d}_t, \mathbf{w}\rangle + t\Psi(\mathbf{w})\} \tag{5.79}$$

With ℓ_2 regularization ($\Psi(\mathbf{w}) = \frac{\lambda}{2}\|\mathbf{w}\|_2^2$), this arg min has a closed form solution Xiao (2010):

$$\mathbf{w}_{t+1} = \frac{1}{\lambda t}\mathbf{d}_t \tag{5.80}$$

This can be computed on the fly. Where, $\mathbf{w}_t(i)$ is the i'th component of $\mathbf{w}_t$, when the gradient computation routine $\partial\ell$ requests $\mathbf{w}_{t+1}(i)$, it can be computed in time $O(1)$ as $\frac{1}{\lambda t}\mathbf{d}_t(i)$. We only compute those indices needed, and the number needed will be much smaller than D.

5.6.2 Efficient Averaging

The problem of computing the average over primal iterates $\frac{1}{T}\sum_{t\in[T]}\mathbf{w}_t$ is the problem of computing the sum $\sum_{t\in[T]}\mathbf{w}_t$, since the $\frac{1}{T}$ factor is trivial to introduce at any time. We begin by reviewing an efficient algorithm for summing the primal weights $\mathbf{w}_t$ for the perceptron algorithm, which seems to be implied in Collins (2002). Then, we discuss how this algorithm can be modified to support summing of primal weights $\mathbf{w}_t$ on the basis of sparse updates to the dual average $\mathbf{d}_t$.

Perceptron If operating directly on the primal weights, as in the perceptron algorithm, efficient averaging can be achieved with $O(d)$ updates by using the following data structure. For each index i of the weight vector, let $\mathbf{w}_t(i)$ again be the value of the i'th component of $\mathbf{w}_t$. Our goal is to compute $\sum_{t\in[T]}\mathbf{w}_t(i)$ efficiently for each $i\in[D]$. As state, we maintain the following triple:

$$(v_i, t_{\sigma_i}, \sigma_i)$$

v_i is value of $\mathbf{w}_t(i)$ when it was last changed, t_{σ_i} stores the round inex at which v_i was last changed, and σ_i stores the sum $\sum_{s\in[t_{\sigma_i}]}\mathbf{w}_s(i)$. At some future round index $\hat{t}$, in order to adjust the i'th component by some amount $\Delta\in\mathbb{R}$, we use Algorithm 12. This

Algorithm 12 Efficient Sum Updates for Averaging Primal Weights

1: **procedure** UPDATE-FROM-SUM-PRIMAL($i,\hat{t},\Delta$)
2: $\quad\sigma_i \leftarrow (\hat{t}-t_{\sigma_i})\cdot v_i$
3: $\quad t_{\sigma_i} \leftarrow \hat{t}$
4: $\quad v_i \leftarrow v_i+\Delta$

algorithm can be proved correct by induction on the number of updates. If σ_i stores $\sum_{s\in[t_{\sigma_i}]}\mathbf{w}_s(i)$ before an update at time $\hat{t}$, it will store $\sum_{s\in[\hat{t}]}\mathbf{w}_s(i)$ afterwards. (Note that $\sum_{s\in[\hat{t}]}\mathbf{w}_s(i)$ is not a function of Δ. The first iterate that Δ affects is $\mathbf{w}_{\hat{t}+1}$.)

Dual Averaging In the case of the regularized dual averaging algorithm, the task is to sum the primal weights $\mathbf{w}_t$, even though these primal weights are never computed

explicitly. The correctness of Algorithm 12 relies on the fact that the components of the primal weight $\mathbf{w}_t$ only change if they are part of the gradient update $\mathbf{g}_t$. This is not the case for ℓ_2 regularized dual averaging, in which case each component of the primal weight *does* change on every iteration, shrinking due to regularization. Specifically, the sum that we need to compute now is:

$$\sum_{t\in[T]} \mathbf{w}_t = \sum_{t\in[T]} \frac{\mathbf{d}_t}{\lambda t} \tag{5.81}$$

To do so, we can maintain the following state:

$$(d_i, t_{\sigma_i}, \sigma_i)$$

And, let $\Sigma_\lambda : \mathbb{N} \to \mathbb{R}$ such that:

$$\Sigma_\lambda(t) = \sum_{s\in[t]} \frac{1}{\lambda t} \tag{5.82}$$

Then, efficient summation can be accomplished using 13.

Algorithm 13 Efficient Sum Updates for Averaging Primal Weights

1: **procedure** UPDATE-SUM-DUAL($i, \hat{t}, \Delta$)
2: $\quad\sigma_i \leftarrow [\Sigma_\lambda(\hat{t}) - \Sigma_\lambda(t_{\sigma_i})] \cdot d_i$
3: $\quad t_{\sigma_i} \leftarrow \hat{t}$
4: $\quad v_i \leftarrow v_i + \Delta$

Again, the correctness of this algorithm can be proven by induction on the number of updates. If σ_i stores $\sum_{s\in[t_{\sigma_i}]} \frac{\mathbf{d}_s(i)}{\lambda t}$ before an update at time $\hat{t}$, it will store $\sum_{s\in[\hat{t}]} \frac{\mathbf{d}_s(i)}{\lambda t}$ afterwards. The value of $\Sigma_\lambda(t)$ can of course be computed on the basis of $\Sigma_\lambda(t-1)$ in time $O(1)$. A number of past values must be cached, since we need to compute $\Sigma(\hat{t}) - \Sigma(t_{\sigma_i})$. However, we only need values as old as $\min_{i\in[D]} t_{\sigma_i}$. If storing many values of Σ is a problem, one can simply update a all σ_i at some time $\hat{t}$, and then discard values for Σ older than $\hat{t}$.

5.7 Experimental Methods

5.7.1 Questions Addressed

The goal of the experiments below is to provide evidence towards answering the following questions:

1. Does the IPM distributed optimization strategy result in more accurate models than the single-mixture strategy? If so, does greater test-time accuracy co-occur with a lower objective value reached on the training set?

2. Does distributed training allow for models which are as accurate as those trained sequentially?

3. Does SVM training improve over perceptron training in the context of distributed optimization?

4. Does the IPM distribution strategy result in more quickly trained models than sequential training? That is, does the addition of processing nodes allow a speed-up?

5. How does distributed batch sub-gradient descent compare to IPM?

6. Does any potential speed-up vary with the data set or prediction algorithm?

7. Is the averaging of iterates during optimization, important for sequential stochastic training, still important under the IPM and no-communication strategies?

All of our distributed experiments use either: i) the distributed dual averaging algorithm to train a linear SVM objective, or ii) the perceptron training algorithm. In order to place the distributed dual averaging method of conducting linear SVM training, we first conduct, in §5.8.1, a small number of sequential training experiments, which compare the dual averaging method to forms of stochastic gradient descent and the well-known passive aggressive training algorithm.

5.7.2 Experiment Details

Prediction Tasks Tested The below experiments report results with the following three structured prediction tasks:

1. An *n*-best parser re-ranker.
   This 50-best parser, based on the features of Collins (2000); Charniak & Johnson (2005), re-ranks the output of a first-stage generative parser. The particular features used are those in the set Φ_{phrase} in §6.2.1.

2. A chart-based dependency parser.
   This is the sibling-factored MST projective dependency parsing model of McDonald & Pereira (2006).

3. A trigram part-of-speech tagger.
   This a standard trigram tagger, included as part of the BUBS NLP library (Dunlop et al., 2011), which recovers part-of-speech tags from the *Penn treebank* (Marcus et al., 1994) tag set.

For efficiency, the tagger uses greedy decoding, rather than Viterbi decoding.[5]

Training Methods We compare sequential and distributed perceptron training to sequential and distributed SVM training. The sequential, IPM and single-mixture perceptron algorithms were given as Algorithms 5, 1 and 2. SVM training consists of the regularized SVM training objective (2.28), optimized by the sequential, IPM and single-mixture versions of the dual averaging algorithm, which were given as Algorithms 8, 10 and 11.

Cost Functions The structured hinge-loss function was defined in (2.20), and is repeated here:

$$\ell_{\text{SVM}}(\mathbf{w};(\mathbf{x},\mathbf{y})) \stackrel{\Delta}{=} \max_{y\in\mathcal{C}(\mathbf{x})} \{\langle \mathbf{w},\Phi(y)\rangle + \rho(\mathbf{y},y)\} - \langle \mathbf{w},\Phi(\mathbf{y})\rangle \tag{2.20}$$

Apart from the vector $\mathbf{w}$ being optimized over, loss function per example is thus a function of the feature embedding Φ, the candidate function $\mathcal{C}$, and the cost function ρ. The feature embedding and candidate functions, Φ and $\mathcal{C}$, for each of the three tasks are those that are described in the papers or that are included by default with the software. The cost functions used are:

1. For the n-best parser re-ranker: $\rho(\mathbf{y},y)$ is the sum of the number of labelled and unlabelled dependency attachment errors in parse y compared to gold $\mathbf{y}$.

2. For the chart-based dependency parser: $\rho(\mathbf{y},y)$ is the sum of the number of labelled and unlabelled dependency attachment errors in parse y compared to gold $\mathbf{y}$.

[5] Viterbi decoding finds the globally optimal tagging for a word. But, for a tag vocabulary of size K, and a sentence of length n, Viterbi for a trigram tagger runs in time $O(K^3n)$, which is very slow. Greedy decoding runs in time $O(Kn)$. When tagging the sentence $w_1,\cdots,w_n$, determines tags for the words in the order $i = 1,\cdots,n$. When considering possible tags for the j'th word, all previously predicted tags are considered fixed, and the suitability for tags for indices $i > j$ are not considered.

3. For the trigram part-of-speech tagger: $\rho(\mathbf{y}, y)$ is 0 if tag y equals the gold tag $\mathbf{y}$, and 1 otherwise.

According to the original architectures of the models used, the *n*-best re-ranker and the MST parser use array-based weight vectors, and the trigram tagger uses a map-backed weight vector. This distinction is discussed further on page 5.7.2, and §5.8.7.3.

Test Metrics We use one metric only for each task to measure test accuracies. The use of multiple correlated metrics would make the trends too difficult to interpret. With respect to the dependency parses, and dependencies extracted from phrase-structure parses, the two main metrics typically reported are: 1) **unlabelled attachment accuracy**, UAS, which measures the percentage of unlabelled dependency arcs that are guessed correctly in a predicted parse, compared to the gold, and 2) **labelled attachment accuracy**, LAS, which measures the percentage of labelled arcs that are guessed correctly in a parse. Rather than choose between the two, for the two parsers, the *n*-best and the dependency parser, we aggregate the two, by reporting the arithmetic average between UAS and LAS, i.e., $\frac{\text{UAS}+\text{LAS}}{2}$. For the part-of-speech tagger, accuracy is measured as the percentage of tags guessed correctly in the entire data set.

Data Sets Used We investigate the performance of these optimization algorithms on three data sets of varying sizes created from the *Penn treebank* (Marcus et al., 1994). These are: i) the BROWN part of the *Penn Treebank*, ii) the WSJ part of the *Penn Treebank*, and iii) a BIG data set, which is the conjunction of the BROWN and WSJ data sets. The WSJ data is taken from 1980's newswire text. The BROWN corpus is taken from an assorted set of American literature from the 1960's (Francis & Kucera, 1979), and the division into 10 sections (with the first indexed as 0 below) are as introduced by Gildea (2001). All experiments done in this chapter report results from a development test set: we do not report results from the traditional test sets for these data sets, because with repeated testing of models, we thought that would lead to corruption of the traditional test sets. The data set sizes are shown in Table 5.7.2. The BIG training set is a concatenation of the WSJ and BROWN training sets, with a random permutation of the examples. The test set combines an equal number of examples from each of the WSJ and BROWN test sets used.

Data Set	Train Sections	Train Size	Dev. Test Section	Dev. Test Size
WSJ	2–19	36,149	20	2012
BROWN	0–7	19,740	first $\frac{1}{2}$ of 8	1039
BIG	union of above	55,889	union of above	2078

Table 5.1: Data set sizes for experiments in this chapter.

Creating Shards For each training set, for each network of N processors, the training set was split into N equally-sized contiguous shards. The first $\frac{M}{N}$ examples when to the first shard, the next $\frac{M}{N}$ examples went to the next shard, and so on. We chose to leaven contiguous examples together in order to better simulate data drawn from different distributions.

Averaging Theoretical analysis assumes averaging of iterates (Xiao, 2010; Duchi et al., 2012), i.e., that program output is $\frac{1}{T}\sum_{t\in[T]}\mathbf{w}_t$ instead of simply $\mathbf{w}_T$. And, past empirical work (Collins, 2002) has stressed the averaging of iterates. But, past work using IPM distribution has shown it to be less important in the distributed setting for the perceptron algorithm. Averaging increases the run-time by a constant factor (§5.6.2), so we would like to know whether the averaging step can be avoided. We seek to replicate the results of perceptron training, and to investigate how important iterate averaging is for dual averaging optimization of the SVM objective. Recent work has suggested that averaging only relatively recent iterates actually works better than averaging all iterates (Rakhlin et al., 2011). So, we compare the following two conditions:

1. No averaging of iterates. Simply output the weight vector from the final round.

2. Return an average of all iterates from the last epoch through the training data.

Determining Regularization Parameter The optimal value of λ for each prediction task, and for each data set, was determined using a grid search using sequential dual averaging for 10 iterations over the entire data set. We tried values

$$\lambda \in [10, 1, .1, .01, .001, .0001, .00001]$$

and picked the value for each task and set that led to best performance on the development test set. The values chosen are shown in Table 5.7.2.

	WSJ	BROWN	BIG
the *n*-best re-ranker	10^{-3}	10^{-3}	10^{-3}
the MST parser	10^{-3}	10^{-2}	10^{-3}
the trigram tagger	10^{-5}	10^{-4}	10^{-5}

Table 5.2: Regularization values used for distributed experiments.

Network Sizes We simulate networks of nodes of sizes 1, 8, 16, 32, and 64. This should give a rather complete picture of how performance scales. Some past work has suggested that, actually, the performance improvement starts to degrade as many cores are added (McDonald et al., 2010; Zinkevich et al., 2010). Thus, we wanted to use a broad range of node numbers, in order to better capture the point at which this degradation occurs.

Computing Architecture The *n*-best ranking experiments were each conducted on a shared memory multi-core computer using 8 out of its 16 1.8 Ghz AMD Opteron processing cores. The amount of RAM allocated for the WSJ, BROWN, and BIG training sets was 40G, 40G and 60G respectively. Examples were loaded into memory before training, so hard disk access time does not play a role in the numbers to be reported. Access to computing cores and resources is not explicitly scheduled on this machine, so system load is a possible source of variance in observed results, but we observed that the experiments were run at times in which no other large processing was taking place. The MST parser and trigram tagger experiments were each conducted using 8 cores of a shared-memory multi-core computer of 2.4 Ghz Intel Xeon processors with hyper-threading. The MST parsing experiments were alloted 16 G of RAM and the trigram tagging experiments were alloted 40G. The MST parser training involves caching of features to disk, so disk read times on the cluster are a potential source of variability in training time. The trigram tagger extracts features on the fly based on a set of lexicons stored in memory, and so disk read times are not a source of fluctuation in that case. Access to processor cores and RAM is explicitly controlled by the scheduler on the Intel Xeon cluster used for the MST and tagging experiments. Jobs are given exclusive access to the resources they reserve for the duration of each experiment, and so system load is not a source of variability for the MST parser and trigram tagger experiments.

Simulating Large Networks with Available Architecture While network sizes of

$$\{1, 8, 16, 32, 64\}$$

are simulated, all such simulations are actually done using only 8 cores at a time. Given this, in order to report the wall-clock time that *would have* passed on a network of size $n > 8$ required a calculation based on times observed on the 8 core network. Also, this network is a shared memory network, meaning that that there is no actual communication cost, the communication cost of a truly distributed network must be simulated. Time spent in an epoch is computed as a sum of four quantities:

1. Time spent processing examples.
2. Time spent on network communication.
3. Time spent to compute the averaging primal weights (perceptron) or dual averages (dual averaging) between rounds.
4. Time spent by each node initializing their weight vector for the next iteration.

Since the nodes are meant to run in parallel, the time spent processing examples by the network (1) is calculated by taking the maximum time spent processing the examples of the epoch of any individual node in the network. Time spent in network communication (2) is, in our actual shared memory implementation, 0. However, in §5.8.7, we also experiment with a simulated network delay based on the *All-Reduce* network delays reported in Agarwal et al. (2011). We find that these delays make little qualitative difference. Time spent averaging weights (3) and time spent by nodes initializing weight vectors for the next iteration (4) depend on the representation of the weight vector. Weight vectors can either be represented by an array data structure, or by a map data structure. This component of training time required is looked at in §5.8.7.3.

Output Data In the distributed experiments, after each epoch e through the training data, we output:

1. Total wall-clock time taken to train to end of epoch e.
2. Training set objective value using output vector from end of epoch e.
3. Test-set performance on the development test set using output vector from end of epoch e.

By **wall-clock time**, we mean the amount of time for the network to reach the end of epoch. For IPM algorithms, this constitutes the time until the network reaches the end of line (5) of Algorithm 3, which pools (averages) the estimates of the network.

For single-mixture algorithms, this constitutes the time until all nodes reach the end of the given epoch. Wall-clock time is, of course, system dependent. For each task, all experiments for that task were run on the same type of hardware.

Total Number of Epochs If the cost of communicating between the nodes were negligible, we would expect a network of size $n \geqslant 1$ to run n times as many iterations as a 1 core network in a given time frame. So, in theory, it would be ideal to run n times the number of iterations on an network of size n, in order to compare performance up to a given wall-clock time. However, as we said above, networks of size $\geqslant 8$ are actually simulated using an 8 core network. Thus, the amount of time needed for us to run an optimization experiment up to E epochs with, e.g., 64 nodes is no faster than with 8 cores. Thus, it was not practical to increase the number of epochs linearly with the number of cores. And, there appeared to be diminishing returns on later iterations. So, we chose to increase the number of training epochs as a function of network size as shown in Table 5.3. Furthermore, the MST and tagging experiments were run on a cluster in which each computing job was limited to 48 hours. Simulations of longer than 48 hours could have been by writing program state to disk, and re-starting. However, due to limited access to this cluster and the length of time needed to be spent queueing for each job, we chose to simply work within the 48 hour execution limit. We simply perform as many epochs as are possible as were possible in 48 hours, where the number actually completed may be less than the number attempted as listed in Table 5.3.

Network Size (in cores)	Number of training epochs attempted
1	15
8	60
16	75
32	90
64	105

Table 5.3: Number of training epochs attempted by network size.

Epochs Versus Randomly Drawn Examples As discussed in §5.5.1, on page 88, we make repeated passes through the training data in epochs. Although the theory discussed in §4 prescribes each new example trained on be drawn randomly, we make passes through the data in a fixed order. This is done to avoid the potential confound-

ing affect of different example selection orders, i.e., if one condition seems to be better or worse simply due to its drawing a better or worse example selection order for training on.

5.7.3 Graphing of Results

Linear Interpolation Test vectors are output and evaluated at the end of each epoch of training. Line graphs show either test-time performance or training set objective value as a function of training time. In between two data points, the graphing software we have used uses linear interpolation. That is, a simple line is drawn between any two data points on a graph of, e.g., performance versus time. Test-time performance versus time is plotted using a linear scale for the vertical axis, whereas objective values are plotted using a log-scale. Test performance before training has started is assumed to be 0. This is perhaps somewhat inaccurate because one might suppose that, especially within the first epoch, learning happens faster at the beginning of the epoch than at the end. However, this seems to be standard practice, and we do not feel that this distorts the image relayed by the data. The first objective value plotted is the value after the first iteration of training.[6]

Averaging over Curves In this chapter, we want to summarize the data obtained from experiments involving a large Cartesian product of conditions. In order to highlight key trends, we will want to average over sets of curves. For example, we will often want present curves of performance versus time for a single task. Looking at such curves for each of the three data sets, for each task, might make it difficult to identify the dominant trend. So, we will often average over the curves of the three data sets. Alternatively, we might want to better understand the performance of distributed networks by averaging over the curves for all network sizes > 1. The averaging of curves is performed in the obvious way. Given the set of curves $\left(\left((x_t^k, y_t^k) \right)_{t \in [T_K]} \right)_{k \in [K]}$, the average curve is, with

[6]The starting weight for each run of dual averaging is $\mathbf{w}_1 = 0$. The regularized loss for this vector can be computed. However, due to the mechanics of the dual averaging algorithm, the weight vectors in the first iterations have very large norms. This can mean a large regularization penalty and also large per-example loss. Thus, there may be several output weights that actually have loss greater than the original weight $\mathbf{w}_1 = 0$. We found it confusing to plot this value, since that would mean that the objective value is non-monotonic.

$T = \min_{k \in [K]} T_k$:

$$\left((\frac{1}{K} \sum_{k \in [K]} x_t^k, \frac{1}{K} \sum_{k \in [K]} y_t^k) \right)_{t \in [T]} \tag{5.83}$$

5.7.4 Limits of Our Experimental Design

The purpose of these experiments is to provide evidence to address the questions listed in §5.7.1. We were limited in the amount of time and computational resources with which to run these experiments. As such, only a subset of the relevant experimental settings were investigated. We do believe that through judicious choices and use of resources, we were able to provide meaningful evidence towards the questions of interest. Nevertheless, we would like to give here some deliberate consideration to the ways in which the (necessarily) limited number of experimental conditions might be argued to constrain our ability to draw the intended conclusions. For a given task and data set, there are a number of experimental parameters to fix (or, in the case of a concern like, e.g., *computer load*, which are fixed to some extent by external factors). For each varying parameter, we can imagine that we are actually interested in results given a distribution over values for this parameter, and view a single experiment as a sample from this distribution. This might be a sample from a distribution with variance, in which we are unsure whether our sample has a value above or below the mean. Or, it might be that our choice of setting for the parameter in question introduces a systematic bias. So, we should consider how variance and bias may play a role in the results reported.

Shard Selection The method by which examples are distributed to the various shards is a variable that can affect the comparison between multi-core and sequential training. Shard selection determines the distribution of examples given to each node in the network, and so the similarity or dissimilarity of the local optimization problems on the various nodes. One might conjecture that distributions that vary more extremely present a problem to the distributed optimizer, since the various shards may come to work at cross purposes. As described in §5.7.2, we allocate contiguous blocks of the original training sets to each of the shards. For a data set in which examples are more similar to their neighbouring examples than to examples further away, the use of contiguous blocks compared to a random shuffle, or to the assignment of the $kN + n$'th example (where k is a non-negative integer) to shard n, should make the distributions in each shard more different. Ultimately, the chosen selection method may constitute

a biasing factor, and a serious limitation of this study is that it does not investigate the extent to which similarity or difference of shard distributions affects the usefulness of the IPM and single-mixture distributed algorithms. We leave such an investigation to future work.

Stochastic Example Selection Algorithm As discussed in §5.5.1, the theory of stochastic optimization is predicated on the assumption that the order through which the examples are traversed by the stochastic algorithm is random. Thus, to measure the results using a single order through the data is to take a single sample from the population of all randomly chosen orders through the data, and we can expect there to be variance of the statistics measured. The set of experiments done was all that could be afforded in the time available and so it was not possible to vary order through the data in order to estimate this variance or to average parameter estimates across runs. However, our personal experience, together with the fact that many researchers have actually trained their models using a fixed order through the data (e.g., McDonald & Pereira, 2006), suggests that the typical variation in test performance due to example selection order is quite small—probably, within one or two tenths of a percentage point—and such noise is small compared to the magnitude and trend of the key results reported in §5.7.

Regularization We chose, for each task and data set, the value of the regularization parameter that worked best for sequential training for that task and data set. We have noted several times (§3, §4) that convergence rate for optimization is inversely proportional to the regularization parameter. So, it may be that, with high regularization, sequential training converges very quickly and thus compares very well to multi-core training. However, Table 5.7.2 shows that low levels of regularization worked better in practice, so, though we could not afford to explore all settings of regularization parameters, and may be introducing a bias into the results by choosing only small values for the regularization parameter, the low values investigated are presumably the values that are most relevant for producing NLP models in practice.

Computer Load We reviewed the computing architecture used in the experiments in §5.7.2. For each task, the computing architecture on which the distributed experiments were run was held fixed. However, computer system load due to other jobs running on the machine at the same time as our experiments is a *conceivable* source of variance in our results—one that operates, to an extent, outside of our control. It might be that, during one experimental run the system was quite quiet, while during another

it was quite busy. This would lead to one condition spuriously seeming faster than another. For the MST parsing and trigram tagging experiments, the scheduler on the system that these experiments were run on guarantees each job dedicated access to the resources requested. So, for these experiments, system load is not an expected source of variability. For the *n*-best parsing experiments, dedicated access to resources was not guaranteed by a scheduler, but, we observed that jobs typically ran without other major competing resources. Thus, we do not suppose that system load was an important source of variance in these experiments.

5.8 Experimental Results

5.8.1 Sequential Experiments

In the next sections, all experiments will either involve perceptron training or SVM training using the dual averaging algorithm. Dual averaging is new and less widely used than stochastic sub-gradient descent, and its variants. We have used the dual averaging algorithm primarily because it allows us to apply the analysis of Duchi et al. (2012), which, to our knowledge, does not have an analogous result in the case of stochastic gradient descent. In order to get an understanding of how the dual averaging method of optimizing the SVM objective compares to better-known methods, we begin with some comparisons in the sequential setting.

Objective Value We begin by comparing three methods for optimizing the regularized SVM primal objective. The first is stochastic sub-gradient descent, called *Sgd* in the graphs, for optimization of the SVM objective (see §3.3.2). Following Bottou (2004) we use the step size sequence $(\eta_t)_{t\in[T]}$ given by $\eta_t = \frac{\eta_0}{1+\lambda\eta_0 t}$. The definition of this step-size involves a line search for the parameter η_0 on a sub-set of the data of size $M_s = 400$. The number of iterations depends on the initial point used for the line search. We did not include this as part of the training time, in an effort to simplify the results, and on the basis that such a parameter might only need be tuned once for any given application. However, this choice is of course charitable to the SGD algorithm. Second, we look at "optimal" variant of sub-gradient descent, due to Hazan & Kale (2011), that decreases the step-size on an exponential schedule. This algorithm, shown as *HazanSgd* in the graphs, involves a similar search for an initial step-size, which is not depicted as part of the training time. This was the first SGD-based algorithm to be shown to converge at the rate $O(\frac{1}{\lambda T})$ for a λ-strongly convex function. Finally, we

look at the dual averaging algorithm, shown as *DA* in the graphs. We do not perform averaging of the iterates in these experiments.

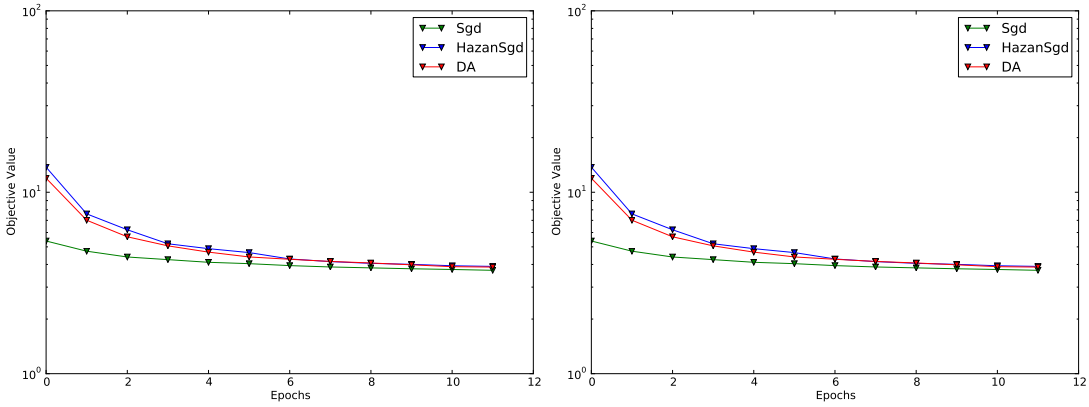


Figure 5.1: Sequential optimization performance of three methods of optimizing the SVM objective for training of the n-best re-ranker. Left figure shows results for BROWN corpus, right figure shows WSJ.

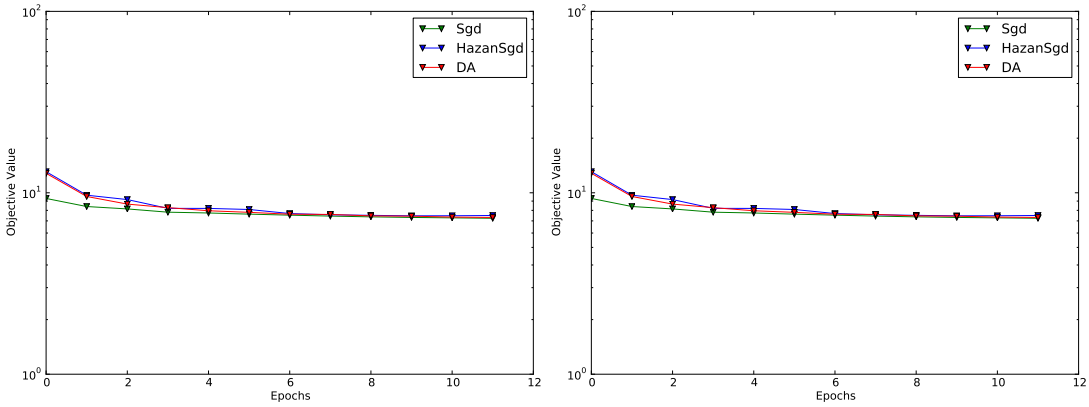


Figure 5.2: Sequential optimization performance of three methods of optimizing the SVM objective for training of the MST parser. Left figure shows results for BROWN corpus, right figure shows WSJ.

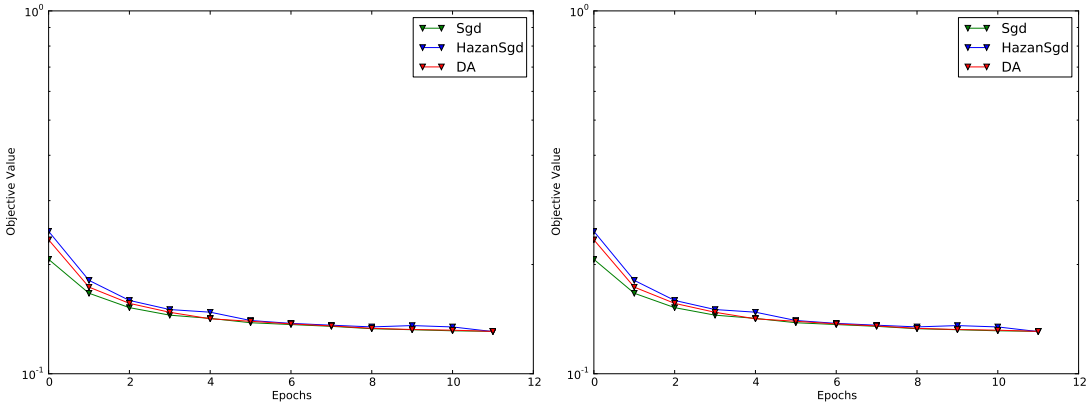


Figure 5.3: Sequential optimization performance of three methods of optimizing the SVM objective for training of the trigram tagger. Left figure shows results for BROWN corpus, right figure shows WSJ.

The objective values reached as a function of iteration are shown in Figures 5.1–5.3. All algorithms find similar objective values eventually. Compared to SGD, the dual averaging algorithm tends to produce weight vectors of larger magnitudes than are appropriate in the first few iterations, but settles down as t, the number of examples seen, grows. Note that the grid search for a suitable η_0 undertaken for the SGD algorithm is done precisely for the purpose of ensuring that initial vector norms are appropriate. Curiously, we find that the HAZANSGD variant of SGD, also needs a few iterations for the objective value to settle down, even though it benefits from a search for suitable starting η_0. It seems the HAZANSGD algorithm performs less well than SGD in practice, here, despite its strong asymptotic guarantees.

Accuracy We now look at the accuracy levels reached for the three optimization methods just discussed. And, we compare these accuracies to accuracies reached by two other training methods. The first is the perceptron algorithm, which we have discussed at length. The second is the max-loss variant of the passive-aggressive algorithm (Crammer et al., 2006) of SVM training. This method can be seen as doing co-ordinate ascent in the dual of an SVM objective. As in the last set of experiments, we do not perform averaging of the iterates in these experiments.

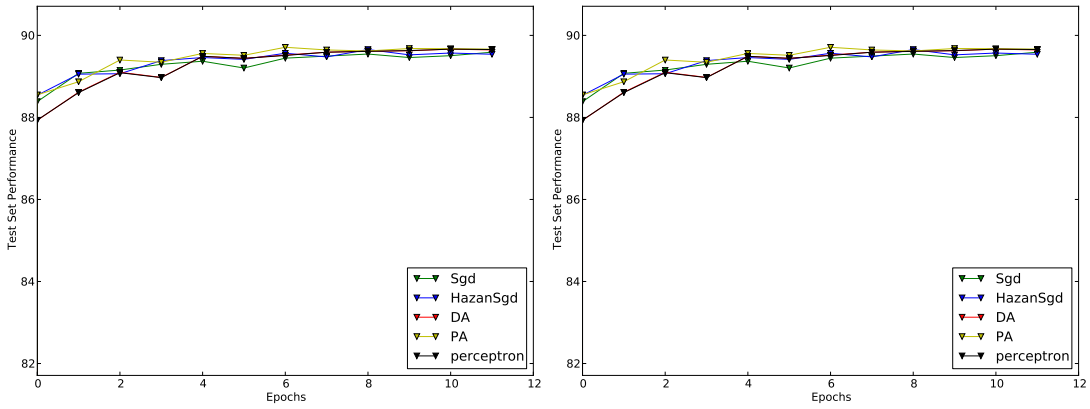


Figure 5.4: Performance of trained models for five methods of sequentially training the n-best re-ranker. Left figure shows results for BROWN corpus, right figure shows WSJ.

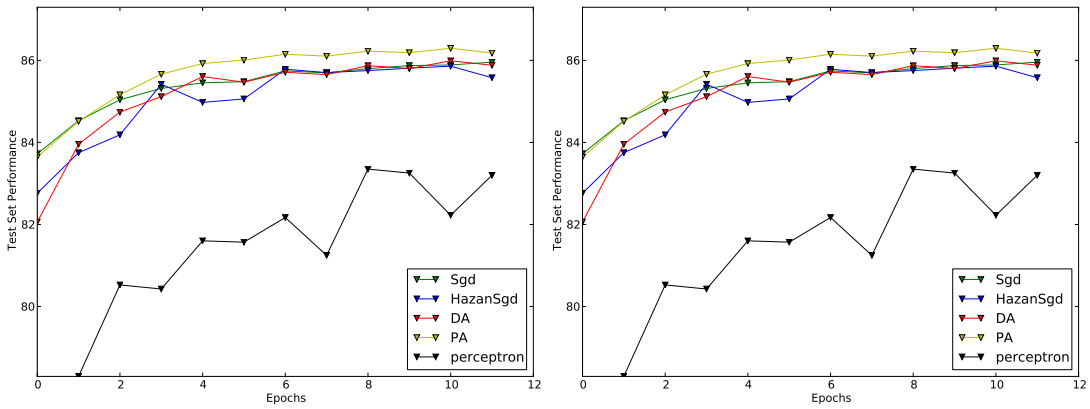


Figure 5.5: Performance of trained models for five methods of sequentially training the MST parser. Left figure shows results for BROWN corpus, right figure shows WSJ.

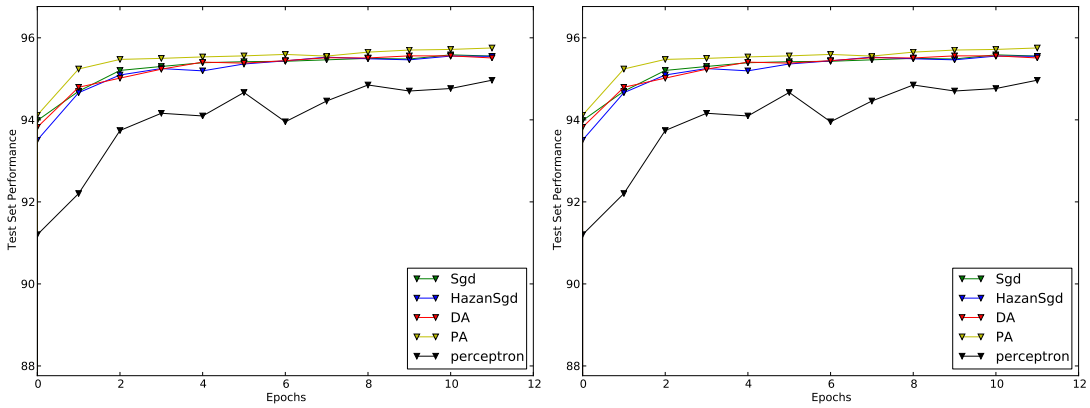


Figure 5.6: Performance of trained models for five methods of sequentially training the trigram tagger. Left figure shows results for BROWN corpus, right figure shows WSJ.

We find that the four SVM-based methods produce similar results. The passive-aggressive algorithm is perhaps the most reliable, but dual averaging fares at least as well, if not better than the SGD variants. The perceptron seems to lag further behind the alternatives. Its inferiority is probably exaggerated by the fact that averaging of iterates is not used. As demonstrated in §5.8.3, averaging of iterates has a bigger impact on perceptron training than on SVM training with the dual averaging algorithm.

5.8.2 IPM versus Single-Mixture

We begin our series of distributed comparisons with the central comparison of this chapter: the difference in objective values and test accuracies reached by optimizing with the IPM and single-mixture distributed optimization and training strategies. We do so by comparing IPM and single-mixture training of the n-best ranker and MST parser

using both dual averaging for the SVM objective, and the perceptron algorithm. In order to condense the results reported, all graphs in this section constitute an average over the three data sets BROWN, WSJ, and BIG.

5.8.2.1 Accuracy

We begin by looking at accuracy. Past work such as McDonald et al. (2010); Hall et al. (2010); Simianer et al. (2012) has found that iterative parameter mixing outperforms single-mixture strategies. Figures 5.7–5.10 compare IPM to single-mixture training for SVM for the n-best re-ranker and the MST parser. In each figure, the graph on the left averages over data set. The graph on the right averages over both data set and network size, for networks of size greater than 1, which gives a different, and perhaps clearer, perspective on the trend for distributed training. The results in this section show a consistent pattern for both dual averaging SVM training and for perceptron training. The perceptron results replicate those of McDonald et al. (2010), with different tasks, and a wider range of network sizes. The SVM results are novel. Models trained using IPM can reach the same accuracy as sequentially trained models, while models trained using the single-mixture algorithm do not reach this level of accuracy.

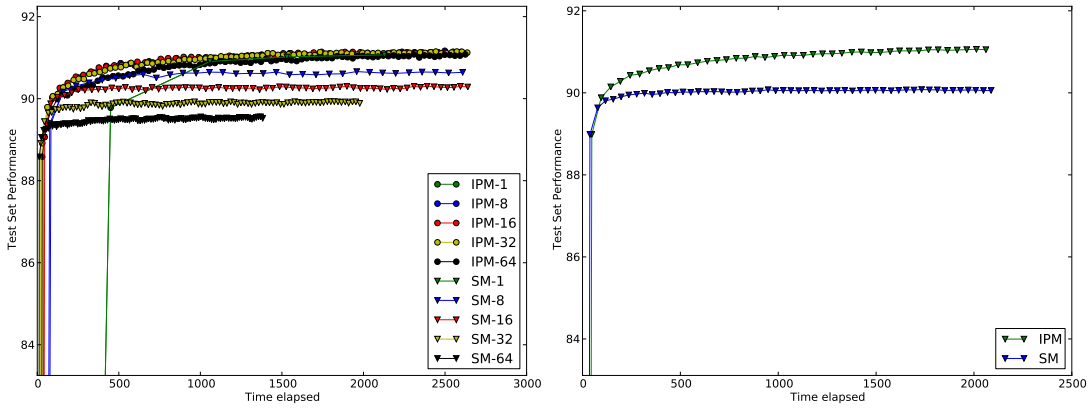


Figure 5.7: IPM versus no communication for SVM training of the n-best re-ranker. Curves on the right represent an average over both data sets, and network sizes greater than 1.

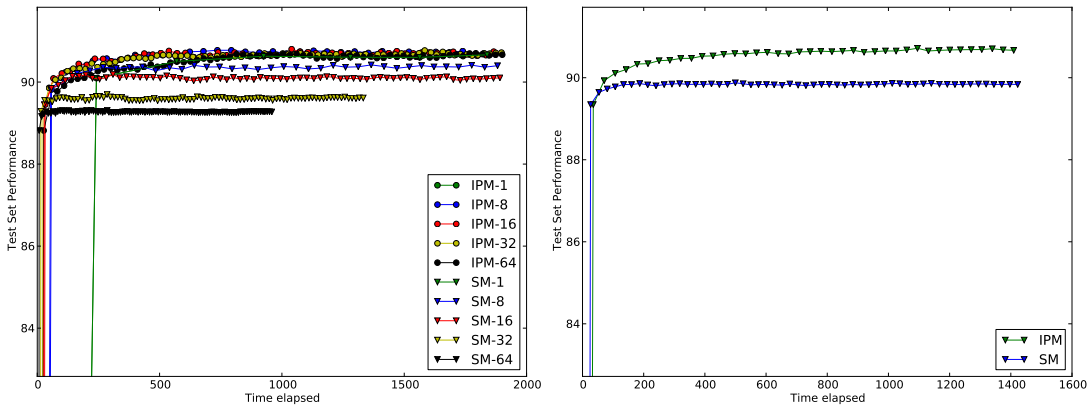


Figure 5.8: IPM versus no communication for perceptron training of the n-best re-ranker. Curves on the right represent an average over both data sets, and network sizes greater than 1.

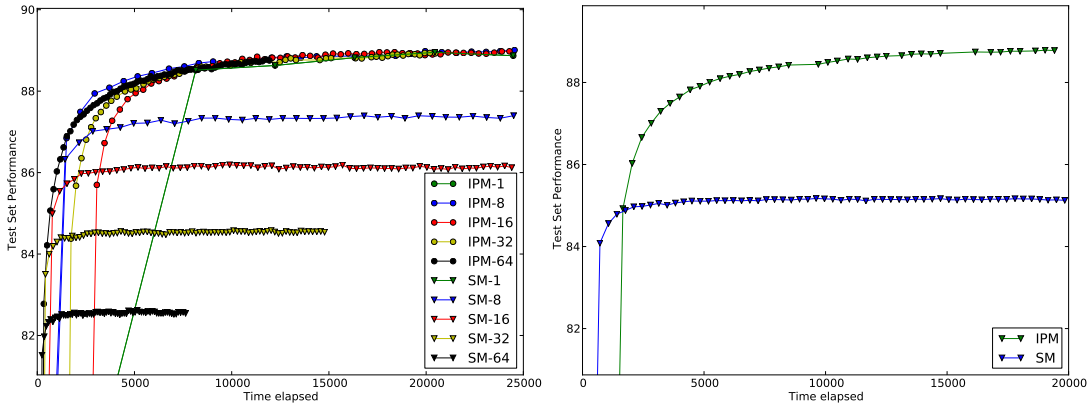


Figure 5.9: IPM versus no communication for SVM training of the MST parser. Curves on the right represent an average over both data sets, and network sizes greater than 1.

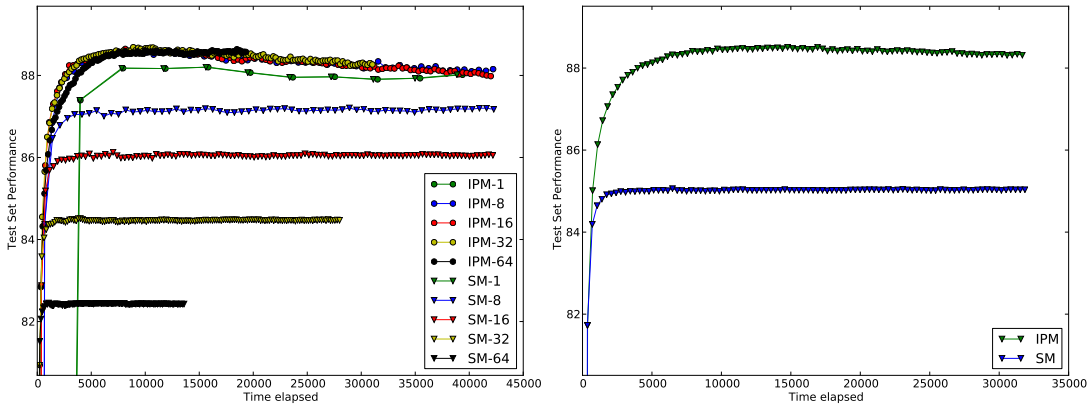


Figure 5.10: IPM versus no communication for perceptron training of the MST parser. Curves on the right represent an average over both data sets, and network sizes greater than 1.

5.8.2.2 Objective Value

5.8.2.2.1 Results The theoretical considerations produced in this chapter do not deal directly with test-time performance. Instead, they deal with objective value over the *training set*. Thus, in this section, we want to investigate whether the objective value reached also parallels the findings for test-time performance. Since the perceptron does not have a meaningful interpretation as regularized average loss, we look only at the SVM objective value reached for both the n-best ranker and the MST parser. This objective value is averaged regularized loss over the entire training set, i.e., over all shards of the data. Figures 5.11 and 5.13 show the objective value reached by each network as a function of time for each of the two tasks investigated. In case it is hard to see the final objective value reached by each network in the graphs with many curves, Figures 5.12 and 5.14 show bar graphs depicting just the final objective values reached by each network.

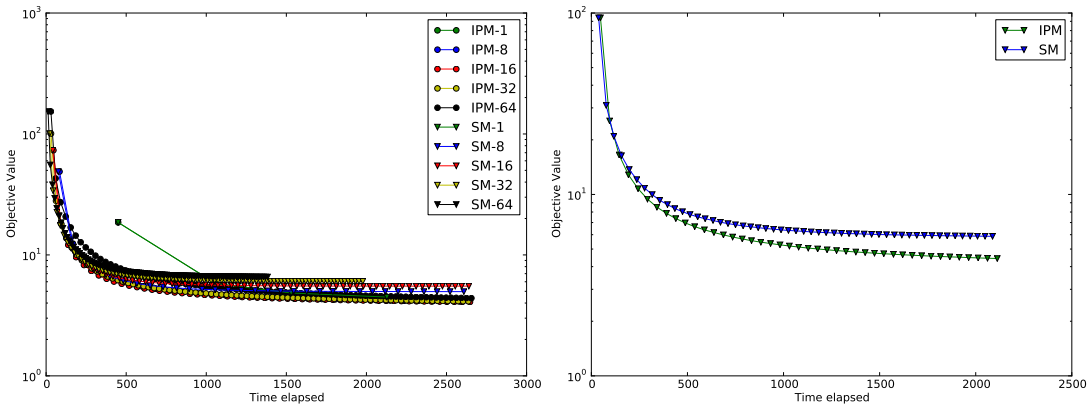


Figure 5.11: IPM versus no communication for SVM training of the n-best re-ranker. Curves on the right represent an average over both data sets, and network sizes greater than 1.

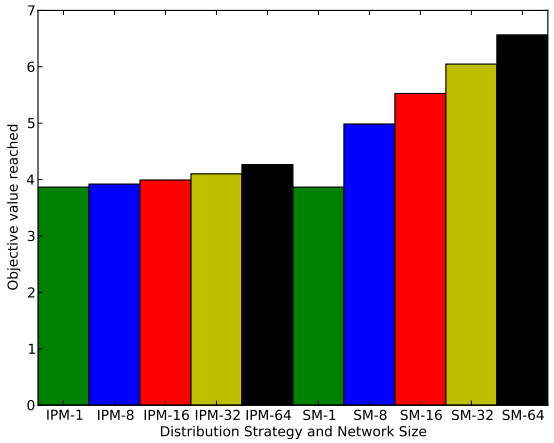


Figure 5.12: This chart shows the final objective values reached for each network size and strategy for the SVM objective in training the n-best re-ranker.

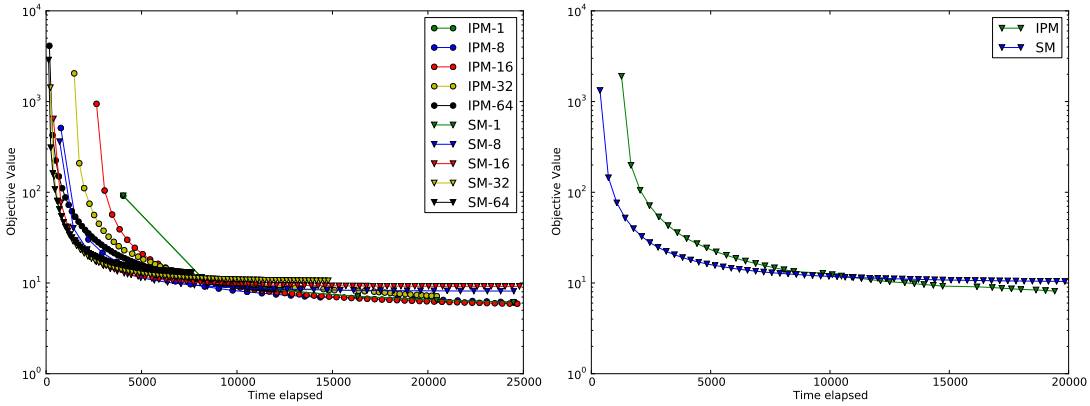


Figure 5.13: IPM versus no communication for SVM training of the MST parser. Curves on the right represent an average over both data sets, and network sizes greater than 1.

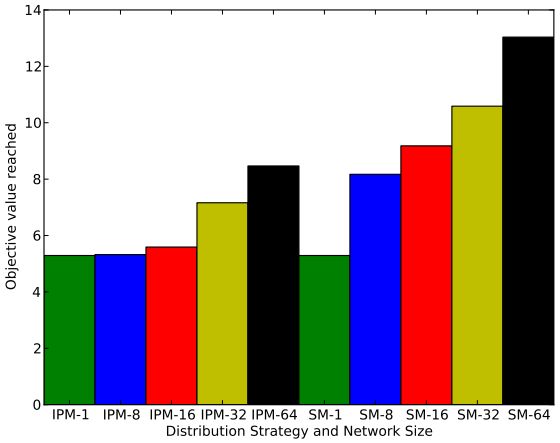


Figure 5.14: This chart shows the final objective values reached for each network size and strategy for the SVM objective in training the MST parser.

IPM *is the Better Optimizer* The clear trend is that IPM is the better optimizer, as the theory predicts. While the performance of the single-mixture models decays as the network size grows, the performance of the IPM networks either doesn't decay (as seems to be the case for the n-best ranker) or seems to decay slowly (as seems to be the case for the MST parser). However, we now more closely investigate the decay in objective value reached for both types of network, and find that the decay of the IPM models may be an artifact of the fact that these models did not have time to run to convergence.

On the Degradation of Objective Value as Network Size Grows We see in Figure 5.12 that the *n*-best ranking objective value for the single-mixture networks decay quite sharply as network sizes grows. And, in Figure 5.14, we see that the MST objective value decays for both IPM and single-mixture conditions, with the decay being more pronounced for single-mixture than for IPM. One question raised as to why objective value decays more for the IPM MST training than it does for IPM *n*-best ranking. We believe this may be an artifact of the way that the experiment was conducted. We said on page 103 of §5.7.2 that networks larger than 8 cores were always simulated with actual networks of 8 cores. And, all experiments had to be run in a roughly fixed period of time, due to constraints on the availability of resources. In a fixed period of time, if we can make E epochs through the data with an 8 core network, we can still only make E actual epochs though the data for any network size simulated in the same time, since each simulation was actually conducted with 8 cores. But, the amount of time simulated on the larger network would have been smaller. E.g., a 64 node network moves roughly 8 times faster than an 8 node network, so the amount of wall-clock time *simulated* in the larger network is roughly $\frac{1}{8}$ the time spent computing by the 8 core network. (In other words, a 64 node network could have done 8 times more computation than what was simulated.) In the case of the MST parsing experiments, there was a hard limit of 48 hours per job.[7] In the case of the *n*-best parsing experiments, there was not a hard limit but rather a soft limit imposted by the need to share the machine amicably with other researchers. This set-up is unfair to the larger networks, since the fair thing would be to run them for the same amount of (simulated) time. And, a closer look reveals that, in some cases, the larger networks were sometimes before the optimization procedure had converged. To give a sense of how much time each network was run for, Figure 5.15 shows just the final point in the objective value versus time curve for each network structure, on both tasks.

[7]This limit could have been worked around by caching results to disk, and restarting jobs. However, we judged this to require considerable bookkeeping and increase in the chance of an error, given that so many conditions were being run in limited time.

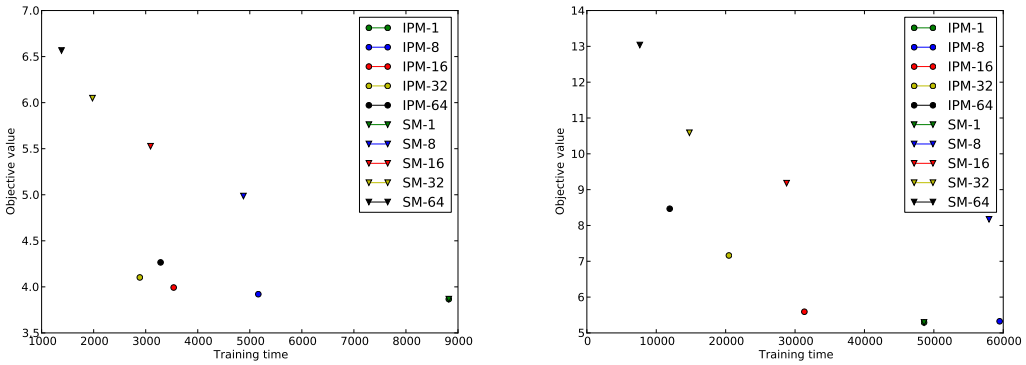


Figure 5.15: Final objective value reached and amount of time simulated for each network in the SVM training of the n-best re-ranker (left) and the MST parser (right). Time-value curves are averaged as discussed in §5.7.3.

In order to get a better idea of the extent to which each condition converged, we calculated and graphed the slopes between the last two points of each objective value versus time curve for each data set. That is, let (x_T, y_T) be the final pair of time (x_T) and objective value (y_T), and let (x_{T-1}, y_{T-1}) be the second to last time-value pair. Then, the slope

$$\text{SLOPE}(x_T, y_T, x_{T-1}, y_{T-1}) = \frac{y_T - y_{T-1}}{x_T - x_{T-1}}$$

characterizes the amount of progress in the objective value that was being made by the optimizer when it was stopped due to computing time limits. The results are shown in Figure 5.16. Convergence would correspond to a value of 0 or greater. Note that, since the single-mixture algorithm is not a convergent optimization algorithm, it is not guaranteed that more time spent optimizing will lead to better solutions, and it may be that, while each node is locally getting a better estimate of its local optimal parameter vector, the overall objective of the network is actually worsening. Thus, in many cases the single-mixture curves end with positive slope. We can see that, for the WSJ and BIG data sets, the IPM conditions did not converge for larger network sizes.

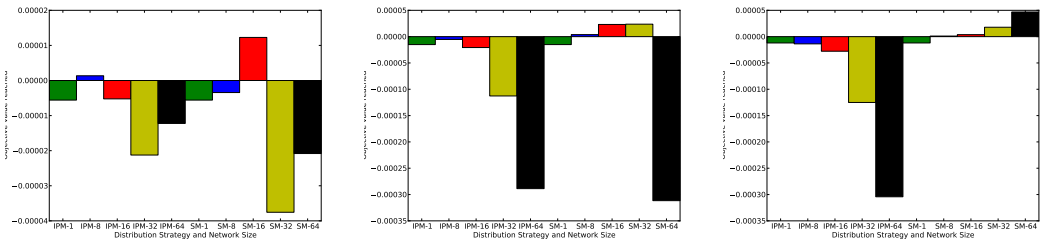


Figure 5.16: Final slope of curves in objective value versus times for the SVM training of the MST parser. Left figure shows results for BROWN, middle figures shows WSJ, and right figure shows BIG.

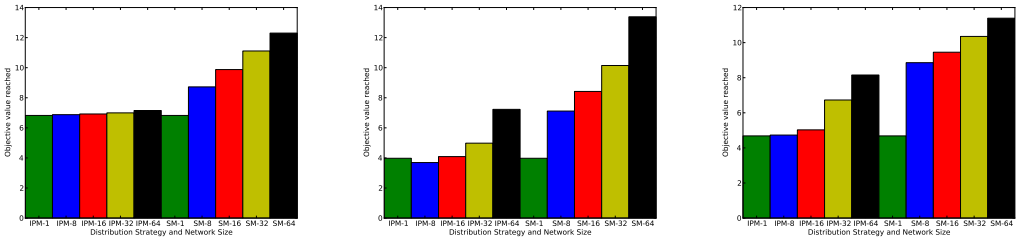


Figure 5.17: Final objective value reached for the SVM training of the MST parser. Left figure shows results for BROWN, middle figures shows WSJ, and right figure shows BIG.

The data set for MST parsing on which convergence was overall the best was the BROWN data set. This makes sense, since this is the smallest data set, and so more passes through it can be made in a given period of time. The final objective values reached for each data set are shown in 5.17. In the case of the best-converged condition, BROWN, we see that the objective values reached for the IPM conditions do not degrade much as network size grows. Indeed, this trend resembles the one seen in Figure 5.12. Thus, we conclude that, while degradation of objective value as network size grows is quite pronounced for single-mixture distributed optimization, degradation in IPM objective value reached would likely have been non-existent if available resources had allowed each such network to be run to convergence.

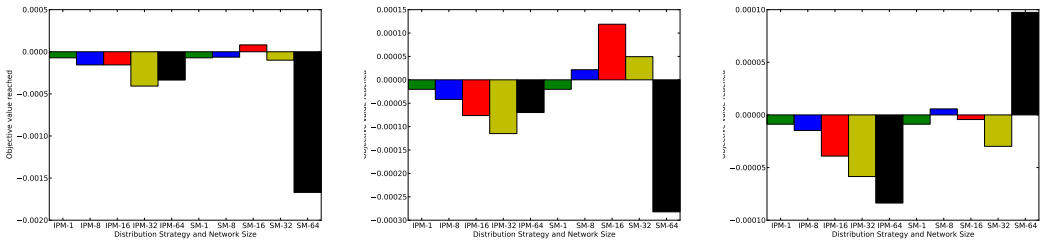


Figure 5.18: Final slope of curves in objective value versus times for the SVM training of the n-best re-ranker. Left figure shows results for BROWN, middle figures shows WSJ, and right figure shows BIG.

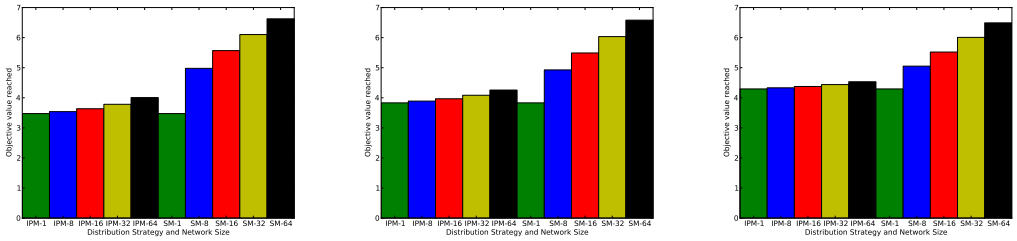


Figure 5.19: Final objective value reached for the SVM training of the n-best re-ranker. Left figure shows results for BROWN, middle figures shows WSJ, and right figure shows BIG.

Finally, for completeness, in Figures 5.18 and 5.19 we show the same breakdown for the n-best ranker. In general, the trend is similar. The IPM models were usually stopped while still converging. In the cases of the BROWN and WSJ data sets, it seems that the SM-64 was actually also still converging, though the general trend for all single-mixture models that had converged suggests each SM-64 model would have converged to an objective value worse than SM-32 if it had been allowed to run to convergence.

5.8.2.2.2 Discussion IPM *Converges while Single-Mixture Does Not* We have concluded that IPM networks always either reach the optimal objective value (or, at least the same one reached by a sequential optimizer), or else would have reached it if available resources had allowed the network to be run to convergence. In contrast, single-mixture networks do not converge to the correct objective value and the distance between the solution of single-mixture networks and the correct objective degrades as the network size increases. Both facts accord with the predictions of §5.5.

Better Optimization Performance Correlates with Better Performance The IPM networks all reach essentially the same test-time performance of the sequentially trained model, and also reach essentially the same objective value on the training set as the sequential model. In contrast, the test-time performance of the single-mixture models degrades as the network size increases, as does the objective value reached on the training set. Thus, we can say that ultimate optimization performance and test-time performance are correlated.

Degradation in Objective Reached as Cores Added The IPM algorithm is predicted to be a convergent optimization algorithm and so we do expect that, regardless of the network size, the true optimal objective value will be reached. In fact, the theory does not

mention network size as a factor in convergence rate at all for the IPM model. Empirical results do seem to corroborate this as each IPM network either reached the objective value, or else was still improving in objective value when stopped. The degradation in objective value reached in the case of the single-mixture optimization algorithm is perfectly consistent with, and indeed arguably to be expected given, the theory that we have presented. We have said that the single-mixture algorithm is not convergent. Indeed, the theory showed that there exist objectives for which the single-mixture algorithm will never converge. In other words, objectives can be found for which the error is arbitrarily large after even an infinite number of iterations. Thus, we should expect that real-world cases with very large error are possible. However, on the other hand, there is also the possibility for extremely benign cases. Simply imagine a single optimization problem replicated on N nodes. Since each node is solving the same problem, they will be able to reach the optimal solution at least as fast as a single node working on a single copy of that problem. The general bound (in which convergence may not occur at all) is too pessimistic for this benign case. We expect that, as the number of shards into which a finite data set is divided grows, the examples in the shards become more idiosyncratic, and so local optimization problems become more dissimilar. So, when the number of nodes is small, the optimization problems become more similar (i.e., more like the benign case of a single optimization problem replicated on N nodes). As the problem gets more benign, the worst-case bound becomes overly pessimistic. Conversely, as we increase the number of nodes, the dissimilarity of the local optimization problems increases, and we approach a situation in which the pessimistic bound (which says that the single-mixture algorithm may not converge at all) becomes more relevant. Thus, it does make perfect sense that the objective value should degrade as a function of network size given the theory presented.

A Remaining Discrepancy Between Empirical and Expected Risk Though we can conclude that final objective values reached and test-time performance do correlate, there were cases in which IPM networks (which had not yet converged) reached worse objective values but produced better test-time models than single-mixture trained models. For example, in the case of the MST parsing results shown in Figures 5.9 and 5.14, the IPM-8 network never reaches (in the time-frame of the experiment) as good of an objective as the SM-64 network. But, the test-set performance of the IPM-8 model is better. It may be that, for whatever reason, the convergence of the IPM algorithm to the optimum of the *expected* risk (performance on new examples sampled from the target

distribution), is even more rapid, compare to single-mixture, than its convergence to the *empirical* risk (performance on the training set). It is unclear why this might be, but the issue presents an interesting question for further research.

5.8.3 Averaging of Iterates

This section investigates whether it helps to average the iterates during stochastic training. Like most work on analysis of stochastic optimization (e.g., Nesterov (2009); Xiao (2010); Duchi et al. (2012)), the analysis of §4 analyzes the convergence of the *average* test vector over all iterations of the algorithm. Collins (2002) showed iterate averaging to be important for perceptron training, but McDonald et al. (2010) showed that the effect of averaging each iterate was less important using the IPM strategy, as averaging across nodes accomplishes the same thing. Averaging can be accomplished efficiently, with $O(d)$ updates where d is the size of the gradient. However, the constant factor penalty in both time and space used means that one would want to avoid the averaging step if it is not necessary. In this section, we compare both SVM and perceptron training for the n-best ranker and the MST parser, and results presented average over the training set.

5.8.3.1 Test Performance

We begin by looking at test-time performance. Differences due to averaging are rather slight. When the number of optimization nodes and the averaging method are shown together, the results can be difficult to interpret. This difficulty can be seen, for example, in Figure 5.20:

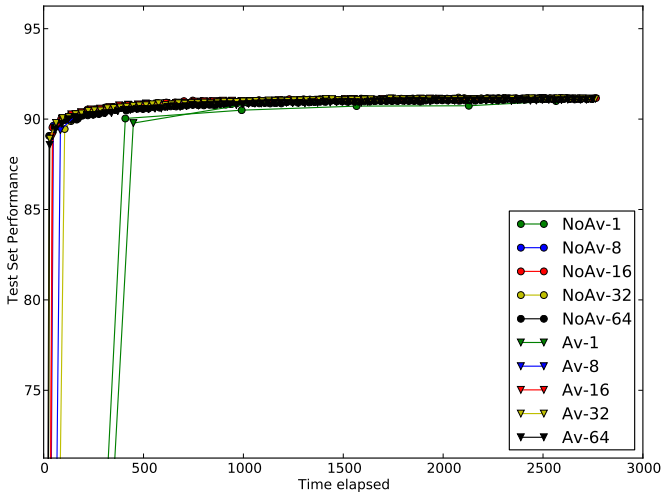


Figure 5.20: Effect on accuracy of averaging of iterates versus no averaging of iterates for SVM training of the the n-best re-ranker. Curves represent an average over three data sets.

Thus, in Figures 5.21–5.24, we depict two graphs each. The graph on the left shows the results for sequential training, and the graph on the right averages over all distributed networks of size greater than 1. Of all cases tested, averaging only seems to be important for the sequentially trained models, and mainly then only in the case of the MST parser. We conclude that averaging of iterates does not seem to be necessary for the test performance distributed training algorithms on the tasks and data sets investigated.

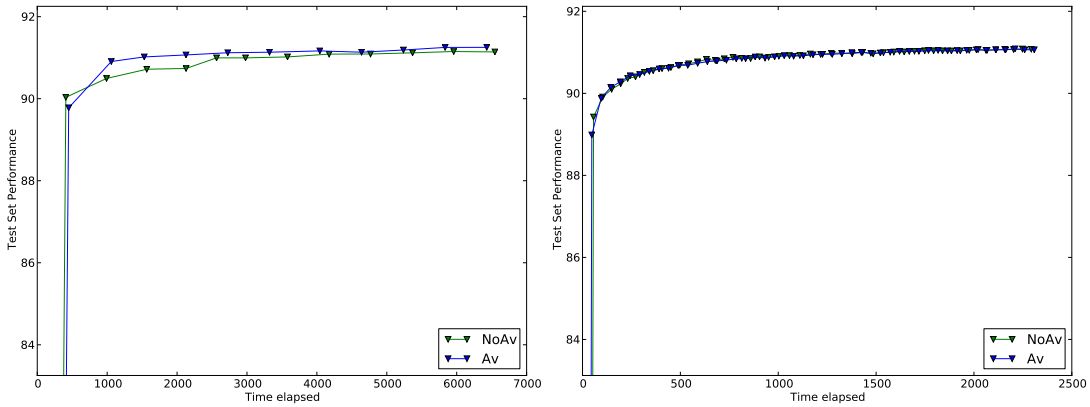


Figure 5.21: Test performance as a function of time, for averaging versus no averaging of iterates in the SVM training of the n-best re-ranker with IPM. Left figure shows sequential training (single node), and right figure averages over numbers of nodes greater than 1.

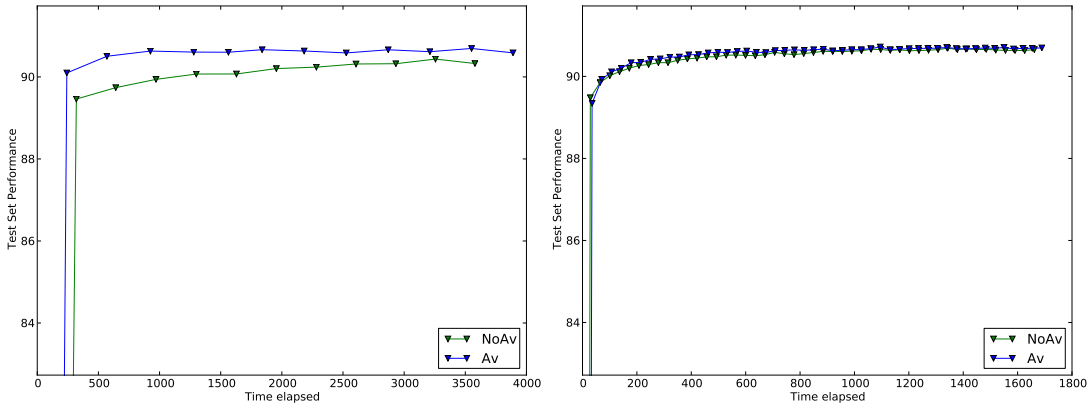


Figure 5.22: Test performance as a function of time, for averaging versus no averaging of iterates in the perceptron training of the n-best re-ranker with IPM. Left figure shows sequential training (single node), and right figure averages over numbers of nodes greater than 1.

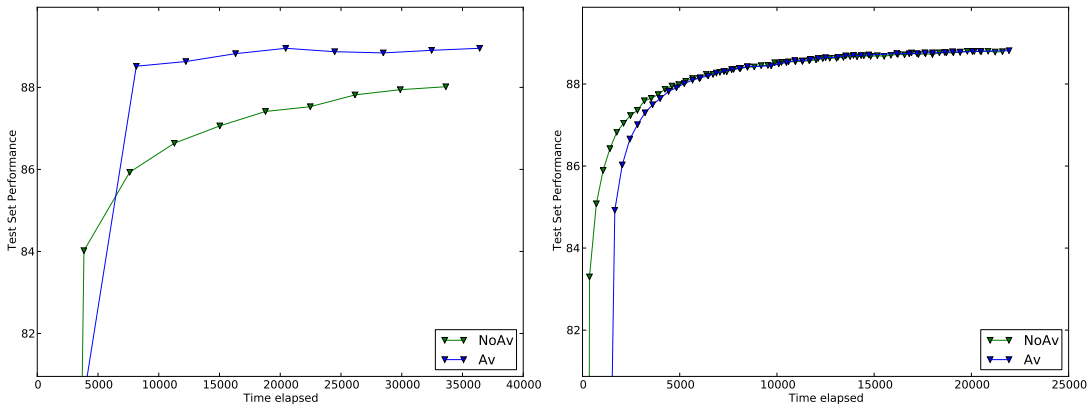


Figure 5.23: Test performance as a function of time, for averaging versus no averaging of iterates in the SVM training of the MST parser with IPM. Left figure shows sequential training (single node), and right figure averages over numbers of nodes greater than 1.

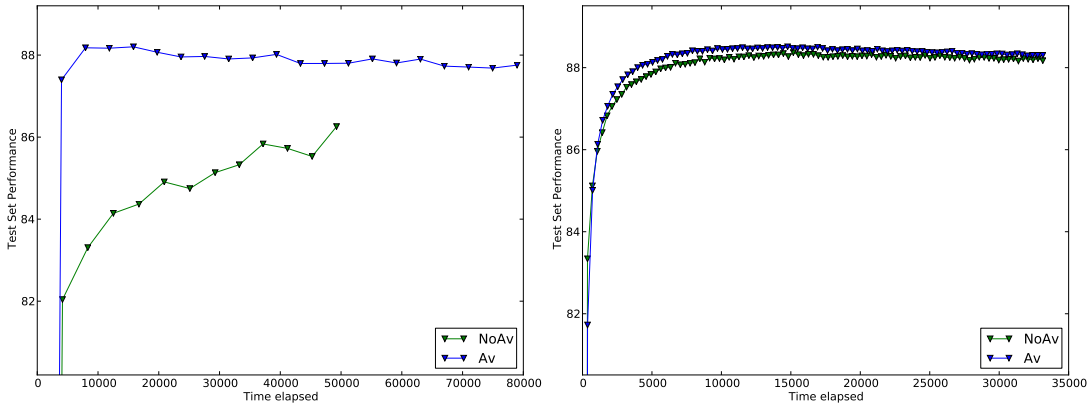


Figure 5.24: Test performance as a function of time, for averaging versus no averaging of iterates in the SVM training of the MST parser with IPM. Left figure shows sequential training (single node), and right figure averages over numbers of nodes greater than 1.

5.8.3.2 Objective Value

The story for the objective value, shown in Figures 5.25 and 5.26, largely parallels that for test-time performance. There is actually appears to be a slight advantage to not averaging in the case of distributed training of the MST parser.

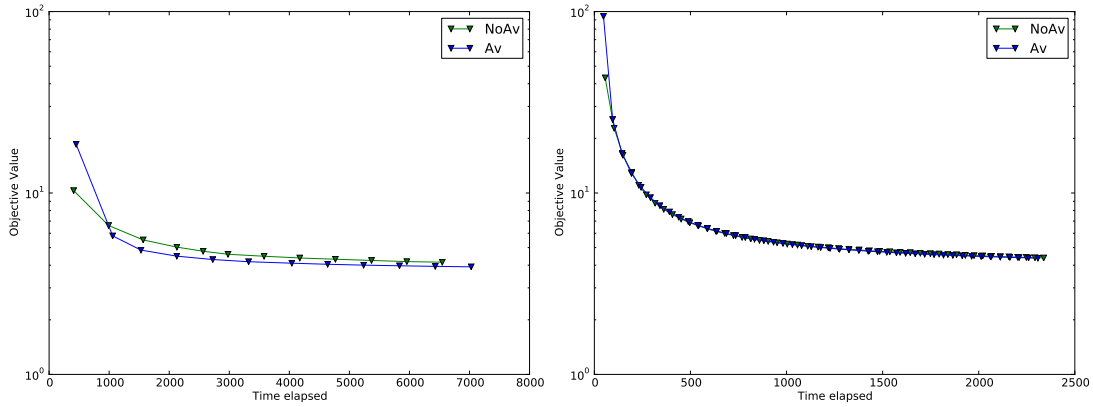


Figure 5.25: Objective value as a function of time, for averaging versus no averaging of iterates in the SVM training of the n-best re-ranker with IPM. Left figure shows sequential training (single node), and right figure averages over numbers of nodes greater than 1.

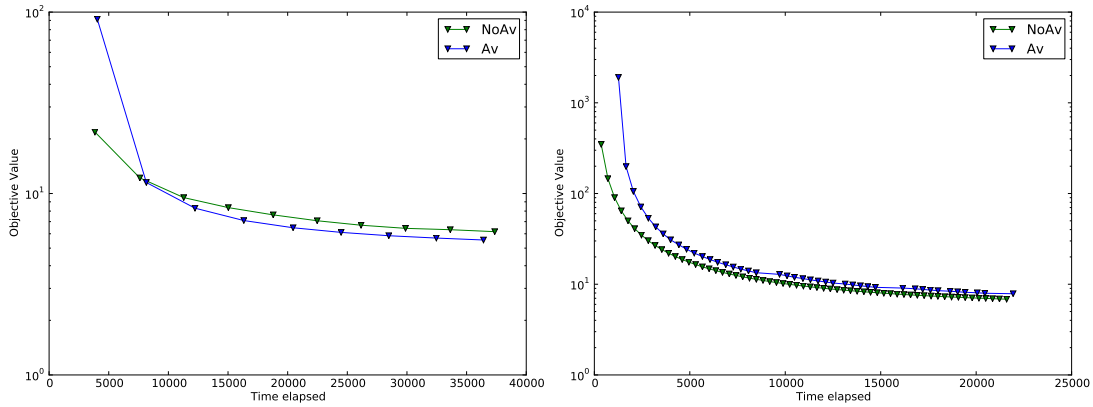


Figure 5.26: Objective value as a function of time, for averaging versus no averaging of iterates in the SVM training of the MST parser with IPM. Left figure shows sequential training (single node), and right figure averages over numbers of nodes greater than 1.

On the Use of Averaging in the Following Experiments The following experiments all assume averaging of iterates. Though we have seen that this is not necessary for distributed networks, it would be unfair to the sequential paradigm to not use averaging, as the lack of averaging can weaken sequential results, and it seemed fairest not to mix averaging and non-averaging paradigms. On the other hand, the use of average does slow down the distributed models by a constant factor, and, thus, perhaps under-represents the potential speed-up they can bring.

5.8.4 Large-Margin Learning

In this section, we examine the use of large-margin training (SVM) versus perceptron training, for each of the three tasks under investigation. The fact that our IPM analysis supports a justification for distributed SVM training with IPM is one of the benefits of the analysis conducted in this section. Here we present the differences in accuracy level reached between SVM and perceptron training. Figures 5.27, 5.29, and 5.31 show accuracy versus time for distributed networks, and Figures 5.28, 5.30, and 5.32 show results for sequentially trained models.

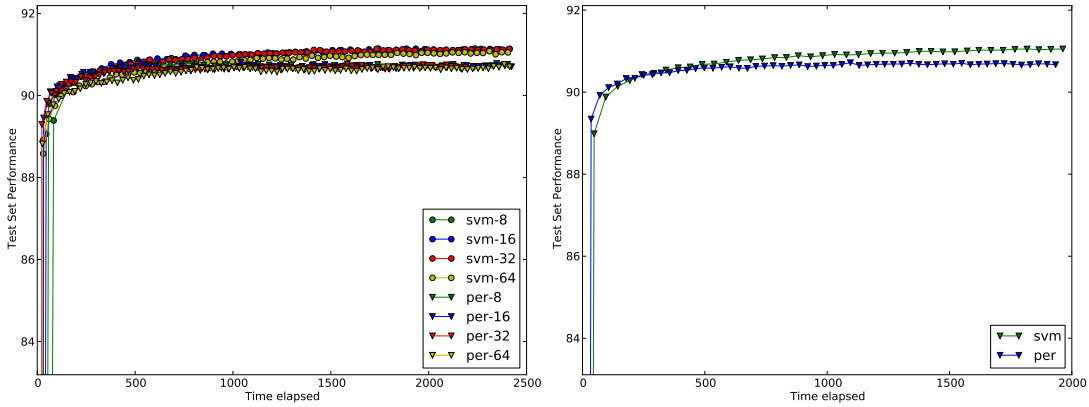


Figure 5.27: SVM training versus perceptron training of the n-best re-ranker using distributed training. Curves are averaged over three data sets. Left figure depicts performance for each network size greater than 1. The right figure averages over all network sizes greater than 1.

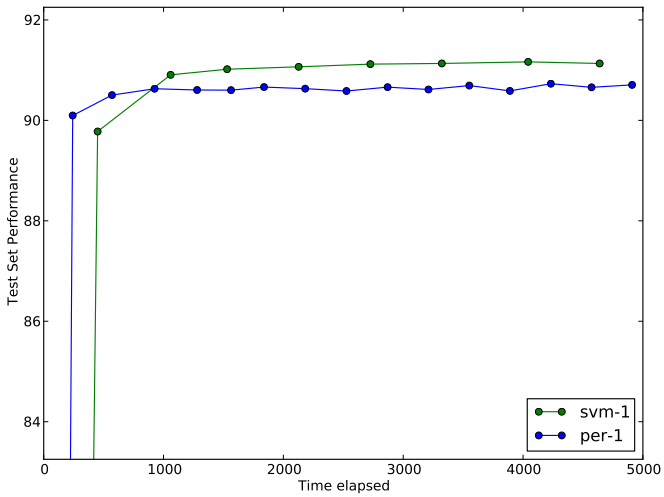


Figure 5.28: SVM training versus perceptron training of the n-best re-ranker using sequential training. Curves are averaged over three data sets.

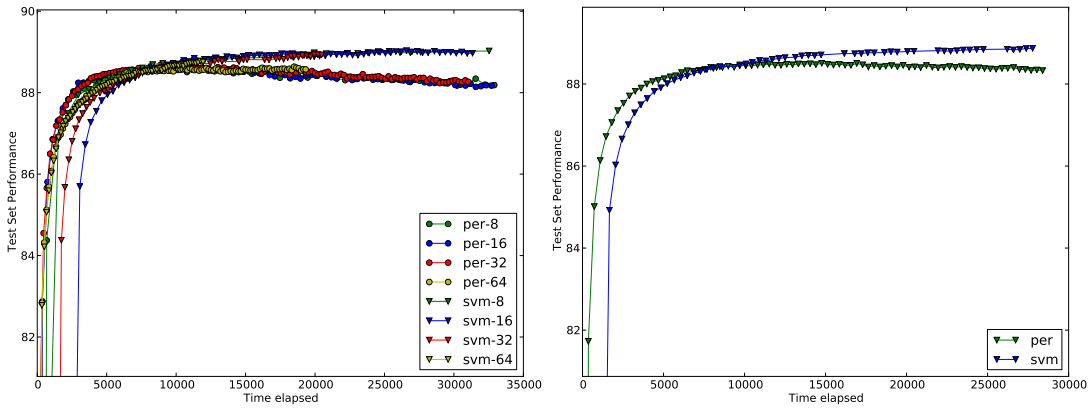


Figure 5.29: SVM training versus perceptron training of the MST parser using distributed training. Curves are averaged over three data sets. Left figure depicts performance for each network size greater than 1. The right figure averages over all network sizes greater than 1.

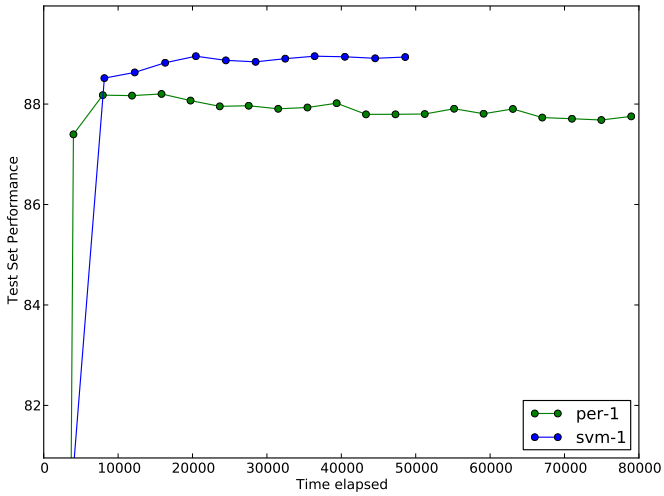


Figure 5.30: SVM training versus perceptron training of the MST parser using sequential training. Curves are averaged over three data sets.

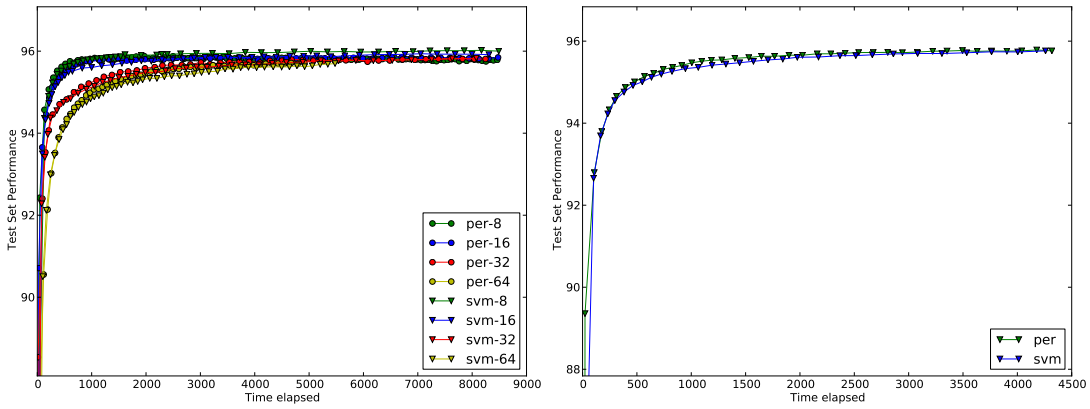


Figure 5.31: SVM training versus perceptron training of a trigram tagger using distributed training. Curves are averaged over three data sets. Left figure depicts performance for each network size greater than 1. The right figure averages over all network sizes greater than 1.

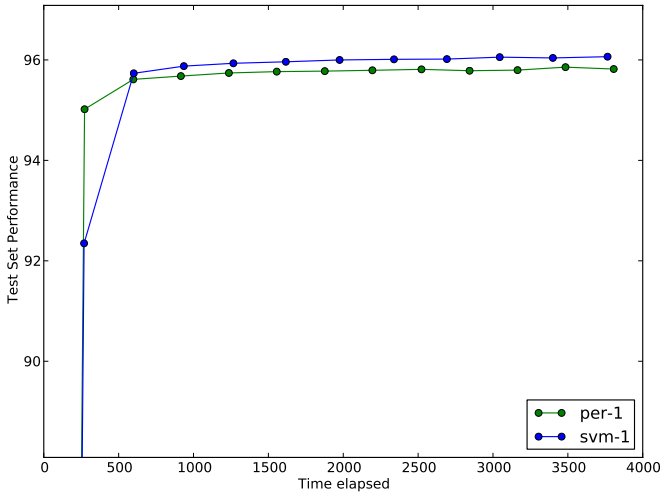


Figure 5.32: SVM training versus perceptron training of the trigram tagger using sequential training. Curves are averaged over three data sets.

We can see that SVM training often allows us to reach a slightly better accuracy than perceptron training. In the case of the MST parser, training with the SVM objective is more stable than perceptron training. The accuracy of perceptron trained models peaks after a few iterations, and then degrades. This does not happen with the SVM models, and suggests that perhaps the SVM training procedure can be run indefinitely in an online setting, if need be. The benefit of SVM training is least pronounced in the case of the tagger. This is perhaps because the cost function in that case is $0/1$, and thus makes less of an impact than truly cost-sensitive loss functions.

5.8.5 Communication Frequency

The analysis of §5.5 suggests that the convergence rate of the network depends on the frequency of communication, rather than the size of the network. We now investigate whether or not more frequent averaging does indeed lead to better convergence. Here, we focus only the n-best ranker. We use a fixed network size of $N = 12$ nodes, and give each the same problem to optimize: the SVM objective for the BIG data set. Unlike the other experiments in this chapter, each node is not working on a shard but on the entire data set, and each of the 12 nodes chooses examples randomly on each iteration.

Nodes either communicate after each 100, 1000 or 10,000 examples. And, we track the test-set performance and objective value reached, as a function of the total number of examples processed. In Figures 5.33 and 5.33, we see that though the difference is slight, there is indeed an advantage to communicating more frequently, especially for objective value reached, as the theory predicts. This is especially visible in terms of the training objective value.

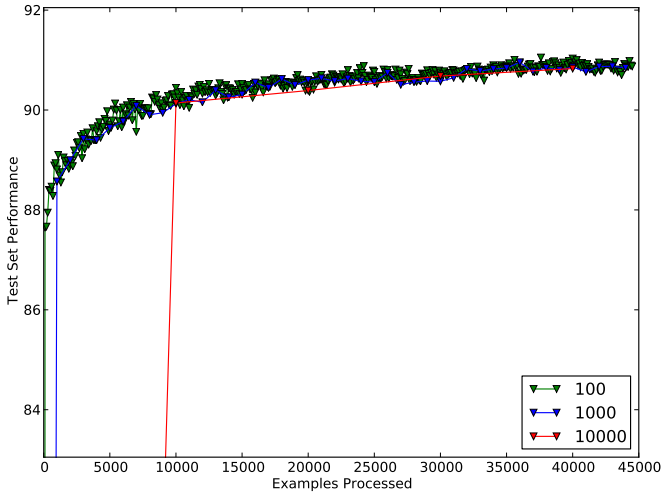


Figure 5.33: Test set performance reached on BIG data set for network of size 12, as frequency of communication varies.

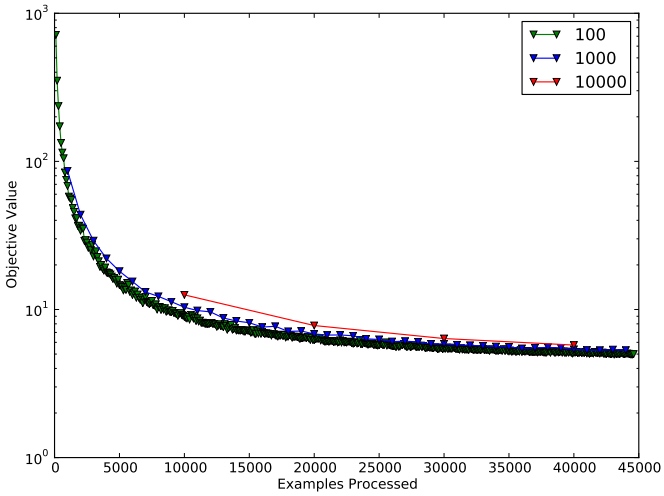


Figure 5.34: Training objective reached on BIG data set for network of size 12, as frequency of communication varies.

5.8.6 Distributed Gradient

We said in §1 that stochastic training is very popular for NLP. We also noted that batch training is very simple to analyze, since the computation is unchanged compared to

sequential batch training, but happens roughly N times faster on an N core network. The problem with batch training for non-smooth functions is that the iteration complexity is the same as for stochastic algorithms (§3.3), even though a batch iteration takes M times longer (where M measures data set size) than a stochastic iteration. Past work has suggested that distributed batch learning does not out-perform distributed stochastic learning (Hall et al., 2010; Agarwal et al., 2011). We conduct a small number of experiments to examine whether this result holds in our scenario. We look at batch sub-gradient descent optimization of the SVM objective. The step-size schedule is chosen as it would be for stochastic sub-gradient descent (§5.8.1). In order to do 150 iterations of batch optimization, we pick the step size schedule that results in the largest objective value decrease after 150 *stochastic* iterations. This schedule may be too slow for batch optimization, but it can be hard to suggest an alternative. Fixing a step-size to optimize progress of 150 *batch* iterations would not be practical, as this would involve repeatedly making 150 iterations over the entire data set which, by assumption, we can only afford to do once. Similarly, the line search required by LBFGS can require multiple passes over the entire data set per gradient step (although, this is apparently done by Agarwal et al., 2011), so an iterative line search to find the optimal step-size also seems impractical. In order to determine whether the step-size schedule was a problem, we experimented with the use of adaptive step-sizes using the passive aggressive scheme of Crammer et al. (2006), but this did not make a difference to progress in accuracy to that reported. The left chart in Figure 5.35 shows the test set performance over time of a distributed batch sub-gradient descent in an 8-core network, compared to IPM optimization with the same number of cores, and the right side shows the progress of the objective value. Although the batch algorithm does get to a decent solution after one iteration, it makes very slow progress beyond that.

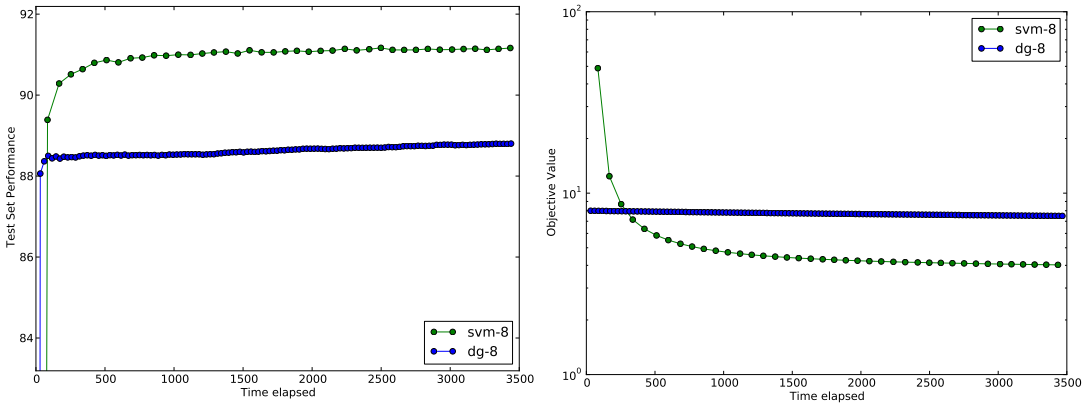


Figure 5.35: SVM training versus perceptron training of the MST parser using distributed training. Curves are averaged over three data sets. Left figure depicts performance for each network size greater than 1. The right figure averages over all network sizes greater than 1.

5.8.7 Speed-Up Due to Multi-Core

Aside from the ability to train on data that is collected in a distributed fashion, the main reasons of interest in distributed training is the ability to speed up the training process. We now investigate whether the IPM training method allows faster training than does sequential training. We consider this from two perspectives. First, we look at low long it takes for distributed networks to reach a given level of accuracy of sequentially trained models. Second, we look at how much training can be gotten done with a fixed wall-clock budget.

Simulating Network Delays Results are presented in two sets. First, we report the wall-clock time with 0 network delay, as was observed on our actual shared memory architecture. Second, we report results with simulated network delay of 5 seconds per iteration, which was the per-iteration delay reported by Agarwal et al. (2011) reported for a 100 node network in their distributed experiments using an *All-Reduce* cluster architecture (see §3.6). Since all of our network sizes are less than this, this seems like a conservative estimate.

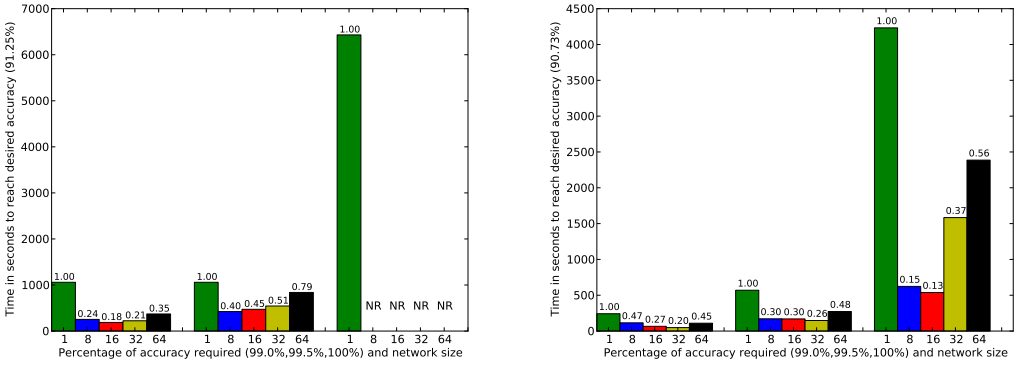
5.8.7.1 Time to Given Accuracy

In looking at the time needed to reach given accuracy, we look at plots that indicate the amount of time needed for a network of size N to reach 99%, 99.5%, and 100% of the best accuracy reached by the sequential model. This information is depicted in bar graphs in Figures 5.36–5.44. The amount of time taken, as a fraction of the sequential optimizer's time to reach the same accuracy, is shown as a fraction above each bar. If the required level of accuracy is not reached by the network size, then the letters *NR* are written in lieu of a bar. Figures 5.36–5.38 show results averaged over data set for both perceptron and SVM training, while Figures 5.42–5.44 show results for SVM training on each data set separately.

Averaging over Data Sets Figures 5.36–5.38 show times needed to reach given accuracy levels with 0 seconds of network delay per iteration, averaging performance

across all data sets. Figures 5.39–5.41 with 5 seconds of delay per iteration, averaging performance across all data sets. The network delay seems to make little difference to the overall trend. For the *n*-best ranker and the MST parser, there is a substantial speed-up to reach the level of 99% test accuracy of the best sequential model. There is overall a speed-up to reach 99.5%. But, 100% of accuracy of the best sequential model, is not always reached. This seems to be the general trend of the data.[8] For the case of the trigram tagger, the results are quite different. In these experiments, multi-core training seems only to slow down training of the tagger. But, the results of §5.8.7.3 show that this is largely due to the fact that the weight vector, in this case, is represented using a map, rather than a vector, which can be avoided using by feature hashing (Weinberger et al., 2009).

First of all, this suggests that if only 99% accuracy is desired, a distributed strategy can be highly preferable to a sequential strategy, provided network communication times are sufficiently low. Also, the ability of distributed stochastic training to reach a decent accuracy level quickly might suggest the benefits of a hybrid approach, that starts with stochastic training and later switches do distributed batch training, as is suggested by Agarwal et al. (2011). Finally, we see that speed-ups appear greater for the perceptron than for the SVM training. This seems to be an artifact of the fact that sequential perceptron training results in weaker models than sequential SVM training, so there is a lower threshold to be reached. To quantify this, the accuracy required to be reached is noted in the label of vertical axis for each bar graph.



[8]The data do exhibit some variance. For example, in the SVM training results of Figure 5.37, we see that, for 8 and 16 cores, a greater savings is achieved to 99% and 100% of the accuracy of the sequential model than is achieved to 99.5% accuracy. This seems to be, at least in some part, because the sequential model reaches 99.5% accuracy in roughly the amount of time that it takes to reach 99% accuracy, but then takes much longer to reach its full (100%) accuracy. This is perhaps just an artifact of a small number of examples from the test set benefiting from some fluctuation in later iterations of sequential training (probably in part a result of the order of training examples, see §5.7.4), and may, in any case, constitute over-fitting of the test set.

Figure 5.36: Time needed to reach given level of accuracy for SVM (left) and perceptron (right) training of the n-best re-ranker with a 0 second between-round network communication delay. Graphs represent an average over three data sets.

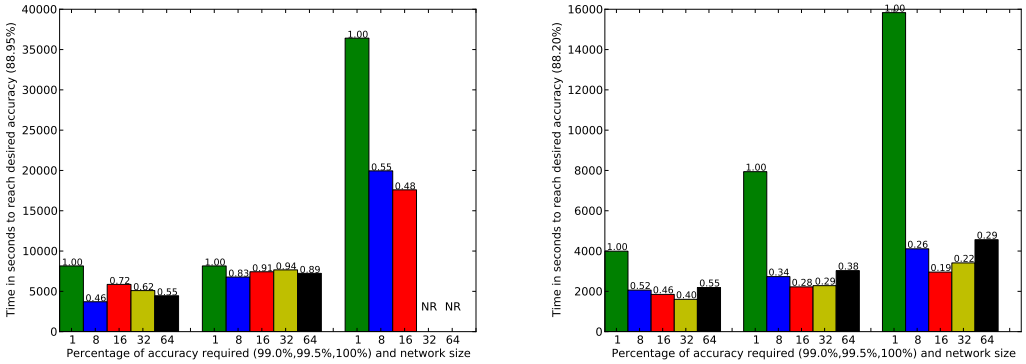


Figure 5.37: Time needed to reach given level of accuracy for SVM (left) and perceptron (right) training of the MST parser with a 0 second between-round network communication delay. Graphs represent an average over three data sets.

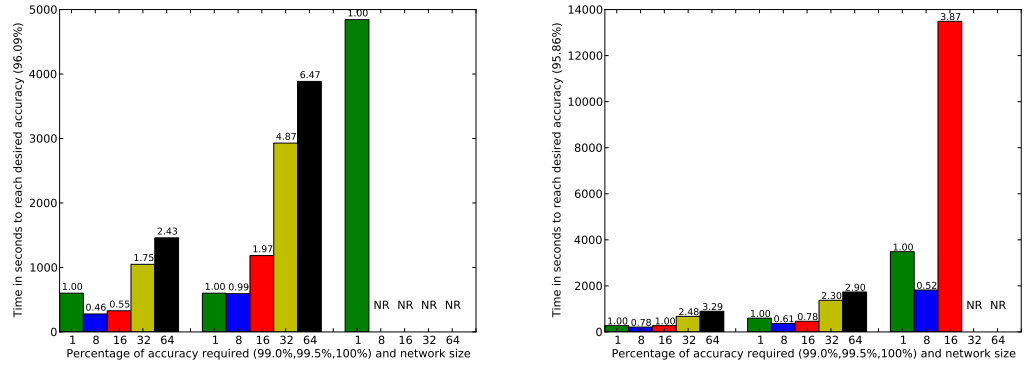


Figure 5.38: Time needed to reach given level of accuracy for SVM (left) and perceptron (right) training of the trigram tagger with a 0 second between-round network communication delay. Graphs represent an average over three data sets.

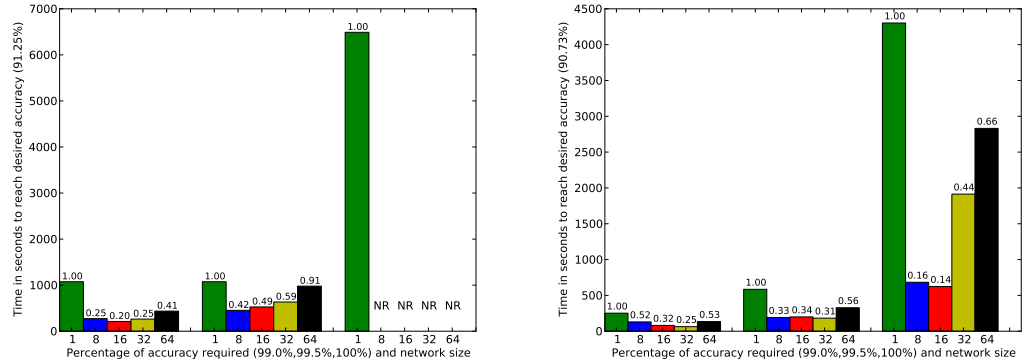


Figure 5.39: Time needed to reach given level of accuracy for SVM (left) and perceptron (right) training of the n-best re-ranker with a 5 second between-round network communication delay. Graphs represent an average over three data sets.

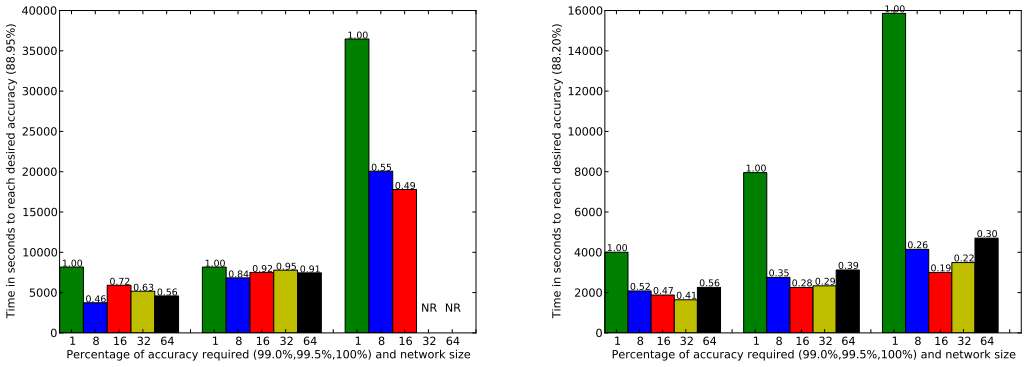


Figure 5.40: Time needed to reach given level of accuracy for SVM (left) and perceptron (right) training of the MST parser with a 5 second between-round network communication delay. Graphs represent an average over three data sets.

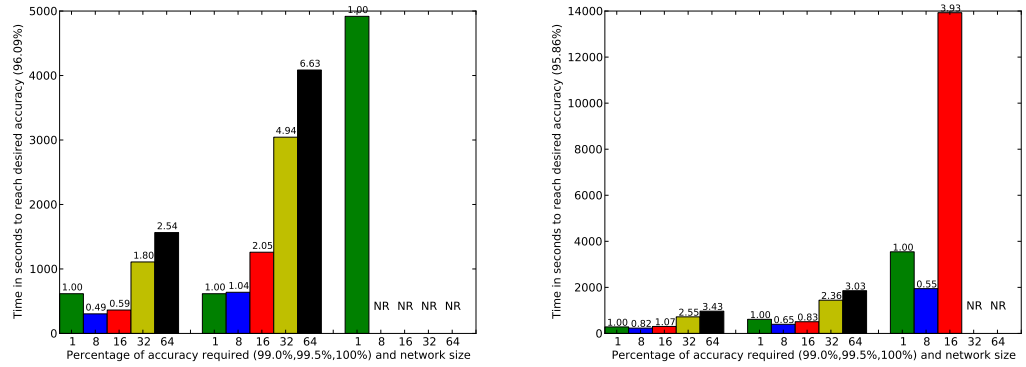


Figure 5.41: Time needed to reach given level of accuracy for SVM (left) and perceptron (right) training of the trigram tagger with a 5 second between-round network communication delay. Graphs represent an average over three data sets.

Results by Data Set We now plot the bar graphs depicting time needed to reach a given accuracy for each data set individually. This allows us to investigate whether the peculiar features of each data set lead to differing behaviours. It does not seem to be the case that there are discernible trends between data sets.

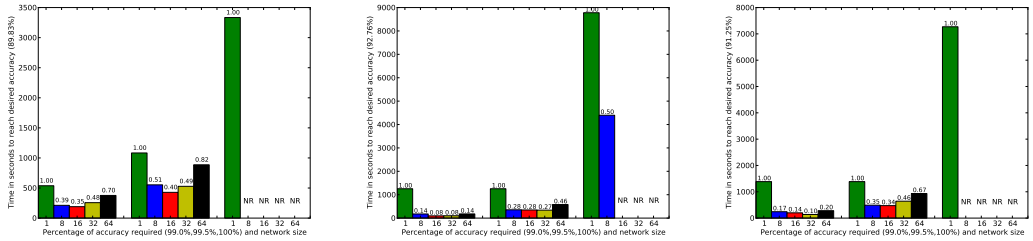


Figure 5.42: Time needed to reach given level of accuracy for SVM (left) and perceptron (right) training of the n-best re-ranker. Left graph depicts performance on BROWN data, middle on WSJ and right on BIG data sets.

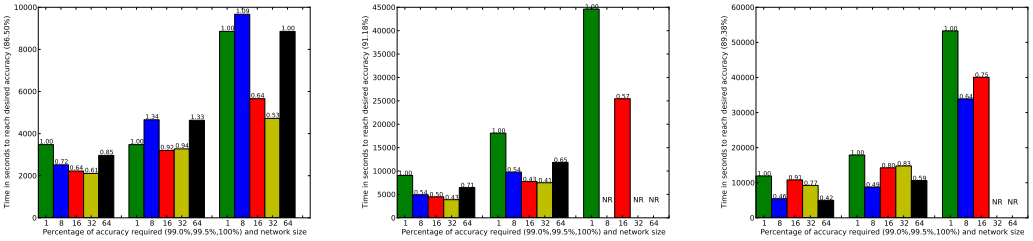


Figure 5.43: Time needed to reach given level of accuracy for SVM (left) and perceptron (right) training of the MST parser. Left graph depicts performance on BROWN data, middle on WSJ and right on BIG data sets.

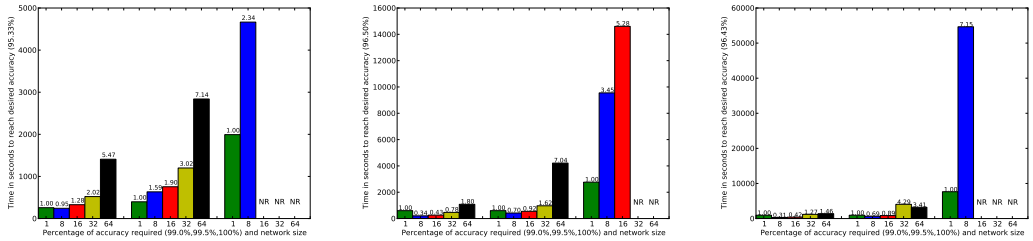


Figure 5.44: Time needed to reach given level of accuracy for SVM (left) and perceptron (right) training of the trigram tagger. Left graph depicts performance on BROWN data, middle on WSJ and right on BIG data sets.

5.8.7.2 Training with a Wall-Clock Budget

In the §5.8.7, we looked at the question of how long it would take for the networks of various sizes to reach comparable levels of accuracy to a sequentially trained model. Usually, there is a speed-up to reach the level of 99% accuracy, but often do not reach the level of 100% accuracy. Clearly, the models trained using distributed training get up to a reasonable accuracy more quickly than the sequential algorithm, but then are

sometimes eventually surpassed by the sequentially trained model. We now look at the question of speed-up in another way, by assuming that there is a fixed budget of training time. Suppose we are only able to or want to wait enough time to run one iteration of sequential training. We now inquire into what level of accuracy can be reached with that amount of time in a distributed network. Figures 5.45–5.47 show the accuracy that can be reached with a budget of 1, 2, and 10 sequential training iterations without any network delay. Clearly, with time for only 1 sequential iteration, one can reach a much better accuracy using IPM distribution. With enough time to make 2 iterations, one can get a better model using distribution, but usually only marginally so. With enough time to make 10 iterations, the accuracy of the sequentially trained model is better. Figures 5.48–5.50 show the accuracy that can be reached with a budget of 1, 2, and 10 sequential training iterations with a network delay of 5 seconds per iteration.

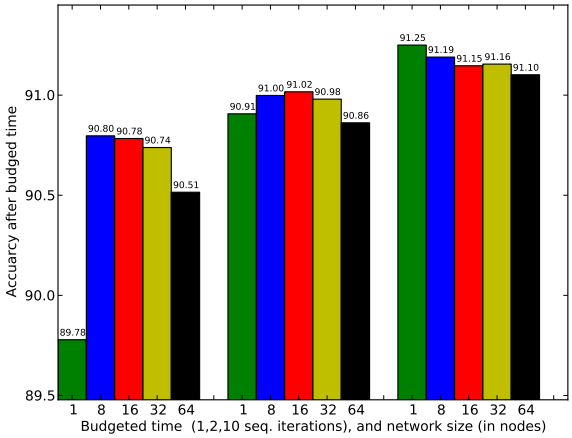


Figure 5.45: Accuracy reachable with given budgets for SVM training of the n-best reranker with a network delay of 0 seconds per iteration.

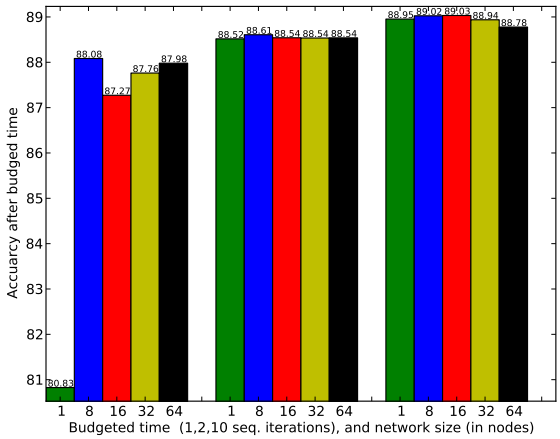


Figure 5.46: Accuracy reachable with given budgets for SVM training of the MST parser with a network delay of 0 seconds per iteration.

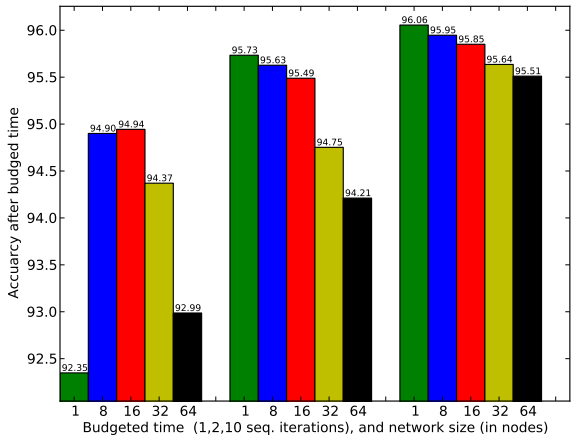


Figure 5.47: Accuracy reachable with given budgets for SVM training of the trigram tagger with a network delay of 0 seconds per iteration.

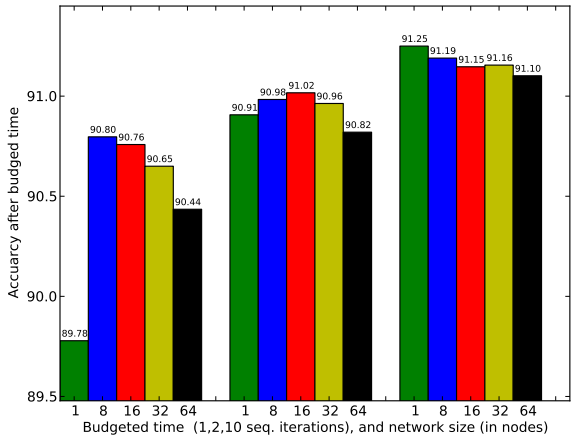


Figure 5.48: Accuracy reachable with given budgets for SVM training of the n-best re-ranker with a network delay of 5 seconds per iteration.

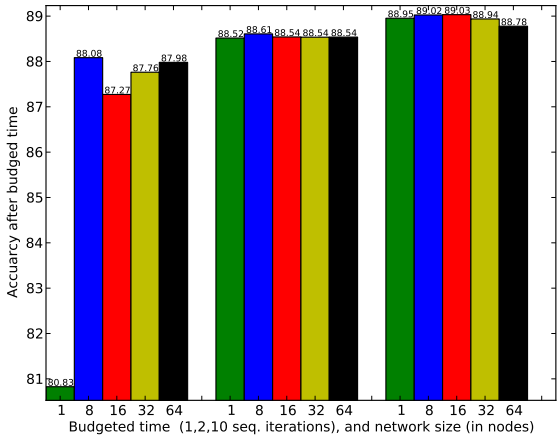


Figure 5.49: Accuracy reachable with given budgets for SVM training of the MST parser with a network delay of 5 seconds per iteration.

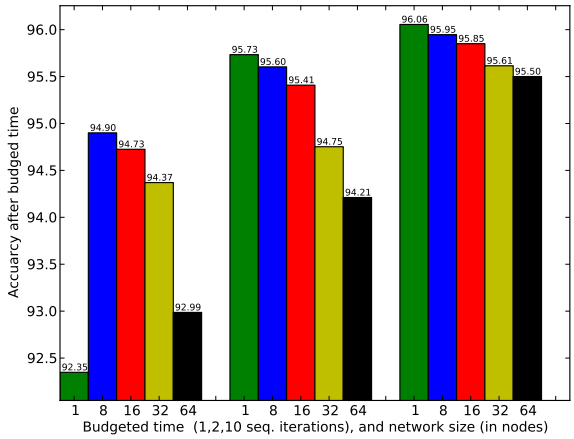


Figure 5.50: Accuracy reachable with given budgets for SVM training of the trigram tagger with a network delay of 5 seconds per iteration.

5.8.7.3 Contribution of Weight Representation to Training Time

The weights for the n-best re-ranker and the MST parser are backed by arrays. The weights for the the trigram tagger are backed by a hash table-based map. Time taken to average any number of arrays up to 64 is negligible. However, the averaging of vectors of several million entries that is backed by a hash map, in our implementation, took considerable time as the network size grew. The time taken to compute the sum of N vectors of dimension D is $O(ND)$ for an array. And, in terms of amortized time, the same asymptotic bound is roughly true for adding N D-dimensional vectors using a hash map (Cormen et al., 2001). However, the constant term is much higher for

hash tables due to the need to the computation of the hash function for each look-up, as well as time spent resizing the hash table when it grows. The fraction of time spent in the parameter averaging step of the IPM algorithm for each network size in SVM training of the the trigram tagger on the BIG data set is given in Figure 5.51. In order to understand whether this time spent averaging vectors was impeding the speed-up due to parallelization, in Figures 5.52, we depict the time needed to reach level of accuracy for both perceptron and SVM training of the trigram tagger on the BIG data set, in which time spent averaging the weight vector was removed. Now, we see that distributed SVM training does actually provide a speed-up to reach 99% of the accuracy of the sequentially trained SVM model. The behaviour for training with a budget is similar to the original results. An alternative to using map-backed weight vectors for large dimension weights is to use feature hashing, in which a fixed array size H is chosen, and all features are hashed to one of H values. Here, two features with the same hash value will share a feature, even if they are different. However, in practice this approach works well (Weinberger et al., 2009). This would mean that, during the averaging step, one is simply adding N arrays.

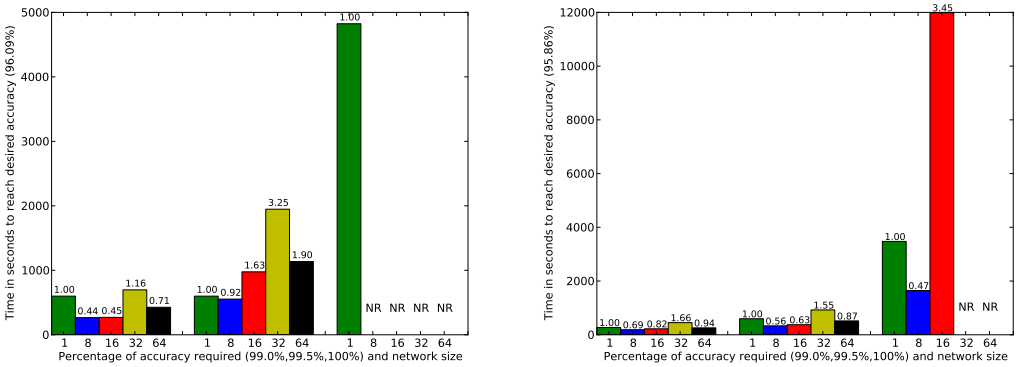


Figure 5.52: Time needed to reach given level of accuracy for SVM (left) and perceptron (right) training of the trigram tagger with a scond between-round network communication delay. Graphs represent an average over three data sets.

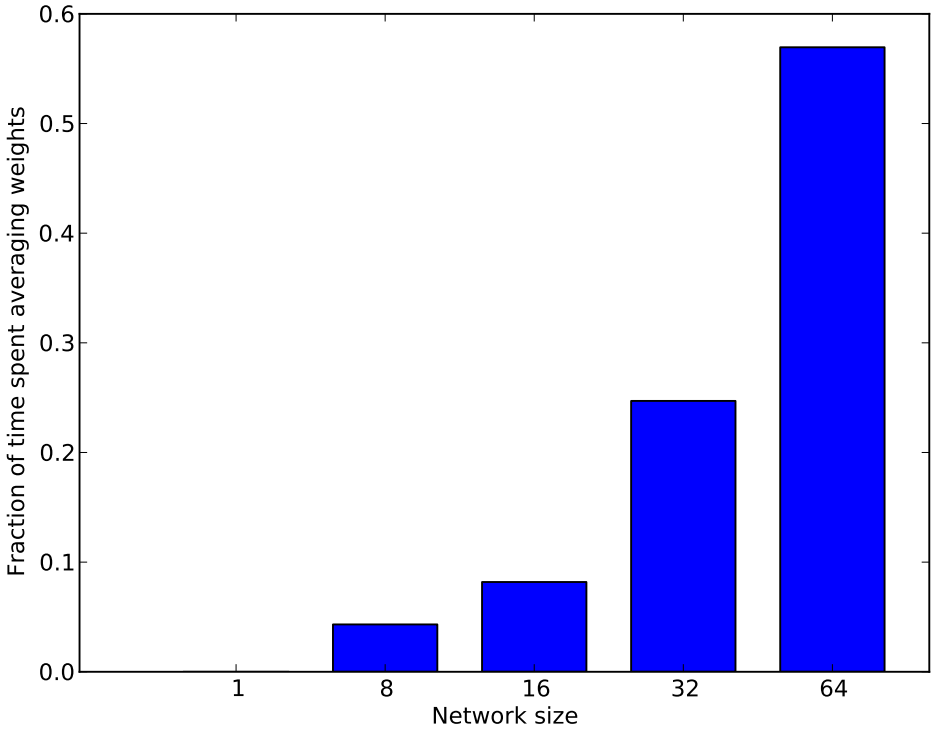


Figure 5.51: Fraction of total training time used for computing the average dual vector, for each network size, for SVM training of the the trigram tagger on the BIG data set.

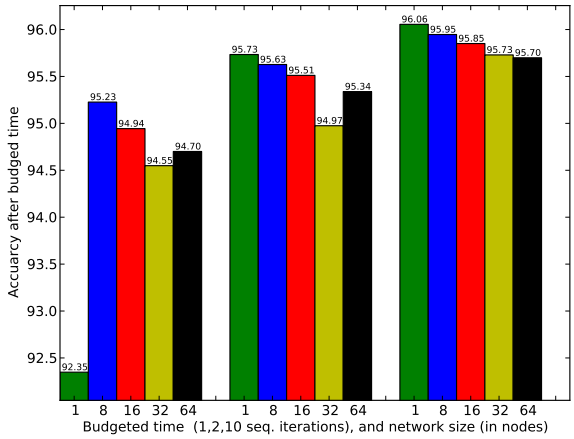


Figure 5.53: Accuracy reachable with given budgets for SVM training of the trigram tagger with a network delay of 0 seconds per iteration, if weight averaging time is excluded.

5.8.8 Summary

Using n-best ranking and MST parser training experiments, the results just discussed have corroborated past results that the IPM distribution strategy outperforms the single-mixture distribution strategy in terms of test performance for the perceptron, and extended these results to training using the SVM objective. Furthermore, we have seen that the superior test-set performance of IPM optimization of the SVM objective correlates with better optimization of the training objective. This corroborates the predictions of the theory discussed in §5.5. The combination of this new theory and new results gives better understanding to the result that has been reported in the past that IPM performs better at test time than single-mixture, with the explanation being a better training objective is reached. Distributed training using the IPM distribution strategy can almost always reach a model that is 99.5% as accurate as a sequentially trained model and often one which is 100% as accurate. In contrast, single-mixture distribution strategies cannot reach this level of accuracy, and distributed batch processing seemed to progress far too slowly to compete with the IPM algorithm. This implies that IPM distributed training is at least a viable option for reaching the true level of accuracy training in a case in which the data set cannot fit on a single computer and distribute training is necessary. We have also seen that SVM training out-performs perceptron training, even in the distributed training case, which justifies the use of the more general version of the IPM algorithm presented in this chapter. In the distributed case, as in the sequential case, this difference seems to be most important when the cost function

has a gradient nature, like the dependency loss function as opposed to the $0/1$ loss of the trigram tagger.

In terms of whether or not there is a speed-up to be derived from IPM distributed training, we have seen mixed results. For models that are backed by arrays, with a number of features on the order of millions, like n-best ranking and the MST parser, if communication cost is negligible, distributed training methods can reach the level of 99% of the accuracy of a sequential method in a small fraction of the time as can sequential training. Also, in such cases, if one only has enough time to make a single sequential pass through the data, one can train a much better model using the IPM distribution strategy. With enough time to make two passes through the data with a sequential algorithm, one can get a better method using IPM distribution, but only marginally so. However, as the distributed algorithms often can only get to 99.5% of the accuracy of the sequentially trained model, they are eventually overtaken by the sequential training method. Again, this may point the way towards hybrid strategies that begin with IPM and later switch to batch training, as suggested by Agarwal et al. (2011). For models backed by a map vector, a speed-up is also achieved if the time spent averaging estimates is removed, suggesting that an approach that uses feature hashing (Weinberger et al., 2009) may yield speed-ups in the same cases as for array-backed vectors.

5.9 Discussion

We believe that the results in this section will help to provide a better understanding of the IPM distribution strategy, which seems to remain a competitive in practice algorithm at the moment in practice for distributed systems (Hall et al., 2010; Simianer et al., 2012). Theoretically, we have shown IPM to be a convergent algorithm, and, empirically, we have shown that, on a wide variety of data sets and for a wide variety of tasks IPM distribution can reach 99.5% of the accuracy of a sequentially trained model. For cases in which data sets must be distributed due to their size, this is important because such models cannot be trained sequentially in any case, and so knowing that the true solution can be reached using distributed training is crucial.

An important limitation of this study is the rather small sizes of the training sets used. They range from about $20,000$ to about $60,000$. In terms of labelled NLP data sets, these are rather large. It was not practical for us to run any larger experiments on the available hardware. However, it may be that training on a very large data set allows

even more of a speed-up than training on smaller data sets. For example, with a training set of 370 million click-through examples, and 9 million features, Hall et al. (2010) show a speed-up factor of about 75, compared to sequential training, on a network with 240 machines. Larger shard sizes might allow each node to be more productive in each epoch, leading to better training times over all.

5.10 Conclusion

The superior test-time performance of models trained using iterative parameter mixing (IPM) to those trained using single-mixture distributed training methods has been noted in the past (McDonald et al., 2010; Hall et al., 2010; Simianer et al., 2012). We give a novel perspective on this result by showing, using the distributed dual averaging framework of Duchi et al. (2012), that the IPM distributed optimization algorithm is guaranteed to converge, whereas the single-mixture training algorithm is not. This suggests that the test-time difference between these two kinds of training is due to superior optimization of the training objective by the IPM-optimized models. Our experiments have verified that the superior test-time performance of IPM-trained models does indeed correlate with better optimization of the objective value. The optimization-theoretic analysis of this chapter is more general than the perceptron-based analysis of McDonald et al. (2010), and supports optimization of the SVM objective. We exploit this fact to conduct distributed SVM training, which we show to be superior in test-time performance to distributed perceptron training. Corroborating the findings of McDonald et al. (2010), we find that averaging of iterates does not seem to be as important in the distributed settings test, for the perceptron or for SVM training, as it is in the sequential case. We investigate the presence of a speed-up due to distributed training. We find that, in some cases, a speed-up exists. In particular, if one has a budget of 1-2 iterations of sequential training, one can can use distributed training to return a much better model in the same time than one could do with sequential training.

Chapter 6

Higher-Order Dependency Features and Out-of-Domain Performance

The last chapter showed that multi-core training using the iterative parameter mixing algorithm allows us to reach a greater level of accuracy, compared to sequential training, when operating with a limited budget of training time. When developing and prototyping a model, we are often in the situation that we are training with a budget, since we are trying to iterate through the develop, debug and evaluate cycle as quickly as possible. In this chapter, we present work that actually exploited the multi-core speed-up to accelerate the development of a new state-of-the-art parsing model. The parser that we present in this chapter advances the state-of-the-art in parsing accuracy on out-of-domain tests, and isolates the contribution of various kinds of features to both in- and out-of-domain parse accuracy. We estimate that that the savings in time during the prototyping and development phase due to the use of IPM multi-core training were considerable, and give specific calculations in §6.6.

Moving on to the contribution of this chapter, we will demonstrate the impact on out-of-domain parsing performance gained by using *higher-order dependency features* in a phrase-structure parser. Such features (see §6.2) encode dependency relationships beyond the bigram head-modifier relationships used since Collins (1997), and have been shown to improve in-domain parsing performance (McDonald & Pereira, 2006; Carreras, 2007; Koo & Collins, 2010). We demonstrate that such features also show a significant benefit when parsing *out-of-domain* material (see §6.1), even when added to a state-of-the-art phrase structure parser along the lines of Charniak & Johnson (2005); Huang (2008b), and so results in a new state-of-the-art for the popular train/test domain pairs investigated. The documentation of this out-of-domain effect of such

features is, as far as we are aware, a novel contribution. Through feature ablation tests, we characterize exactly which kinds of features contribute most to in-domain versus out-of-domain accuracy. We investigate the presence of a feature under-training by comparing models trained using a single round of SVM training to models trained using feature-bagging approach (Sutton et al., 2005), and show that, in some but not all cases, feature-bagging gives a slight edge in performance. Finally, we conduct performance experiments to demonstrate the parsing times under various parser parameter settings of a Huang (2008b)-style cube decoding parser (see §6.3). This work was published as Coppola & Steedman (2013).[1]

6.1 The Problem of Out-of-Domain Parsing

6.1.1 The Problem

The Concept of Domain A well-documented fact in the field of NLP generally, and for parsing in particular, is that a model trained on data from one "domain" does not perform as well when tested on another "domain." We can think of each **domain**, formally, as distribution over input-output pairs $(\mathbf{x},\mathbf{y})$ (Ben-David et al., 2007). Thus, creating a training or test set from a domain A corresponds to repeatedly drawing examples according to the distribution, $\mathcal{D}_A$, associated with domain A, to create a training set D_A. In a *newswire* domain for sentence-parse pairs, we might be very likely to draw sentences that discuss *stocks*, *mergers*, *vice-presidents*, etc. In a *literature* domain, in contrast, we might be more likely to draw sentence that discuss *plots*, *content*, *authors*, etc. Perhaps the newswire domain is described by distribution p_{news} and the literature domain is described by distribution p_{lit}. Then, for any given sentence-parse pair, $(\mathbf{x},\mathbf{y})$, we might expect to find that $p_{news}((\mathbf{x},\mathbf{y})) \neq p_{lit}((\mathbf{x},\mathbf{y}))$. In practice, we can expect that domains can be refined to an almost unbounded degree. That is, "newswire text" can be refined to distinguish between publications, sections of publications, year of publication, author, etc. But, in practice, identifying a natural level of granularity is not a problem.

Problems Associated with a Change in Domain Suppose we have a distribution $\mathcal{D}$ over input-output pairs, $p_{\mathcal{D}}(\mathbf{x},\mathbf{y})$. This induces a marginal distribution over just the

[1]The numbers reported in this paper come from new runs of the experiments compared to Coppola & Steedman (2013). The results are qualitatively the same, but exhibit slight differences. These are discussed at the end of §6.5.1.

inputs $p_{\mathcal{D}}(\mathbf{x})$, and a conditional distribution over outputs given inputs, $p_{\mathcal{D}}(\mathbf{y} \mid \mathbf{x})$. This can be factored as follows:

$$p_{\mathcal{D}}(\mathbf{x}, \mathbf{y}) = p_{\mathcal{D}}(\mathbf{y} \mid \mathbf{x}) p_{\mathcal{D}}(\mathbf{x}) \tag{6.1}$$

This sort of decomposition is used by Ben-David et al. (2007) in the case of binary classification, and Jiang & Zhai (2007) in the case of multi-class classification for NLP. From this point of view, we can look at domain change as consisting of:

1. A change in $p(\mathbf{x})$ but not $p(\mathbf{y} \mid \mathbf{x})$.

2. A change in $p(\mathbf{y} \mid \mathbf{x})$ (possibly, as well as $p(\mathbf{x})$).

We will consider briefly the effects of a change in $p(\mathbf{x})$ alone versus a situation in which $p(\mathbf{y} \mid \mathbf{x})$ also changes. Let us assume, for the purposes of this discussion, that prediction is done using ranking according a global linear model, in which each candidate output $y \in \mathcal{C}(\mathbf{x})$ for input $\mathbf{x}$ is characterized by $\Phi(y) \in \mathbb{R}^n$, its embedding into a real-valued feature space.

In the case of a change in $p(\mathbf{x})$ but not $p(\mathbf{y} \mid \mathbf{x})$, we can expect that, if a feature has been seen in training, the weight associated with that feature is roughly correctly set. This would be true in the case of probabilistic training §2.2. And, it should also be true in the case of margin-based training, in which we assume the prediction function learned will do well on examples generated by the underlying distribution (Vapnik, 1998; Collins, 2005). The problem that domain shift presents in this case, is that many of the features that distinguish candidate outputs in the target domain will not have been seen in the source domain training data. Thus, the weights associated with unseen features will be 0, and so will not be able to help distinguish correct outputs from incorrect ones in the target domain. In the case of a change in $p(\mathbf{y} \mid \mathbf{x})$, we might expect that feature weights which have been set to profitable values for the source domain are set to the wrong values in the target domain. That is, not only might some necessary feature values be set to 0, but those that are set might have the wrong sign, or the wrong magnitude. One might suppose that the first type of shift is more benign. Unfortunately, as Daumé III (2007) demonstrates in his error analysis on a variety of NLP tasks a shift in conditional distribution is possible. But, to the extent that domain adaptation has been successful (see §6.1.2, it seems to indicate that either the conditional distribution $p(\mathbf{y} \mid \mathbf{x})$ does not always change too drastically, or else that this is not fatal.

6.1.2 Proposed Solutions

6.1.2.1 Labelled Data from Target Domain

A number of approaches exist which attempt to augment a large source domain training set with a small number of examples from the target domain. Compared to semi-supervised learning (see below), which attempts to augment a large labelled training set with unlabelled examples from the target domain, this approach has the advantage that techniques similar to those for learning from labelled data, which are powerful and well-understood, can be used. This approach has received a great deal of attention (Roark & Bacchiani, 2003; Blitzer et al., 2006; Daumé III & Marcu, 2006; Daumé III, 2007; Jiang & Zhai, 2007). The drawback, of course, is that adding labelled data is expensive, and so it is not possible to annotate data for all domains of interest.

6.1.2.2 Semi-Supervised Learning

An important approach to domain adaptation is to develop the technique of *semi-supervised* parser training. This involves training from a mix of labelled and unlabelled data. The rationale for doing so is that unlabelled data is essentially free, whereas labelled data is costly. It is universally considered impractical to pay human annotators to label data from every domain of interest, especially since, for many applications, the distribution over examples continues to change. For example, text on the web displays continually changing vocabulary and even style (Petrov & McDonald, 2012). We review some semi-supervised training approaches here.

Clusters Koo et al. (2008) use unlabelled data to create clusters using the Brown et al. (1992) clustering algorithm. These clusters are then used in place of lexical items as features in a parser trained on labelled data. Though they use only in-domain tests, they demonstrate that they can roughly half the amount of data required to achieve the same performance as a parser trained without the clusters. The use of clusters was also very popular in a recent shared task on semi-supervised learning for parsing the web (Seddah et al., 2012; Bohnet et al., 2012; Hayashi et al., 2012).

Feature Statistics from Unlabelled Data A related approach to the use of clusters is to count the number of times that features occur or co-occur in a large unlabelled corpus. These feature counts can then be incorporated into discriminative features to train a supervised classifier. This technique has its early roots in techniques that collect co-occurrence counts, which, in turn, are used to arrive at a prediction in a deterministic

way. For example, Hindle & Rooth (1993) collect co-occurrence counts from a large unlabelled corpus and use the relative magnitudes of the counts, according to a hand-written formula, to make a prediction about prepositional phrase attachment. The same approach has been applied to noun-compound bracketing (Lauer, 1995). Lapata & Keller (2004), however, conduct an investigation of a variety of NLP tasks and caution that such deterministic use of co-occurrence counts fails to out-perform the state-of-the-art supervised statistical methods. In contrast to the pessimistic result of Lapata & Keller (2004) in the case of deterministic application of co-occurrence counts, statistical approaches that use co-occurrence counts as features have been more successful. Nakov & Hearst (2005) show that using co-occurrence statistics in a supervised classifier can out-perform the deterministic models studied by Lapata & Keller (2004). Pitler et al. (2010) use co-occurrence counts as features for noun-compound bracketing to obtain better accuracies than work without access to such counts (Vadas & Curran, 2007). Bansal & Klein (2011) show that co-occurrence counts and paraphrase features from unlabelled text can be used to improve the performance of dependency and phrase-structure parsers.

Automatic Labelling of Unlabeled Data Given the existence of powerful supervised training methods for parsers, a simple and sometimes effective way to make use of unlabelled data is the **automatic labelling** approach. First, one trains one ore more supervised parsers on the available labelled data. Then, the supervised parser(s) are used to automatically label a large quantity of unlabeled data. Finally, another supervised classifier (or classifiers) can be trained on the automatically labelled corpus. Since the original labelled data (labelled by humans) is presumably of much higher quality than the automatically labelled data, interpolation between the model trained on the original labelled data and that trained on the automatically labelled data can be tuned to maximize performance.

The simplest example of automatic labelling is **self-training**, in which a single parser is used to create an automatically labelled data set, and in which the same (or a similar) parser is then trained on the automatically created output. One obvious drawback of the self-training approach is that, if the automatically labelled data contains mistakes, the resulting parser will incorporate these mistakes, presumably degrading performance. This is a likely problem, since we expect a fair number of errors from any parser, especially when parsing out-of-domain. However, modest-to-strong improvements from self-training in practice suggest that the benefits of additional data

outweigh, to an extent, the accumulation of errors. Early on, Charniak (1997) demonstrated modest improvements to a generative parser by self-training on unlabelled data. McClosky et al. (2006a) show that self-training can improve the in-domain performance of the Charniak (2000) and Charniak & Johnson (2005) parsers, but only if the Charniak & Johnson (2005) re-ranker is used. This result is notable for demonstrating that self-training could significantly improve the performance of a then state-of-the-art parsing model trained on a full-sized corpus. McClosky et al. (2006b) demonstrated that this self-training method could also improve out-of-domain parsing performance of the Charniak (2000) parser even without the re-ranker. Sagae (2010) shows that the re-ranking step is not necessary to obtain out-of-domain improvements. Huang & Harper (2009) and Huang et al. (2010) demonstrate improvements from self-training over the state-of-the-art LA-PCFG parser of Petrov & Klein (2007b). Interestingly, Petrov et al. (2010) show that, when training a fast but relatively less-accurate parser (Nivre et al., 2007), it can be better to use a more accurate, but slower, parser (Petrov & Klein, 2007b), rather than simply training the fast parser on its own output.

Also, there exists variants of self-training that attempt to lessen the chance of bad parses (or outputs, generally) being added to the automatically created training sets. **Example selection** techniques attempt to discern between automatically created outputs. Sarkar (2001) and Steedman et al. (2003a) use confidence, in terms of, as a means to select the correct parses. A related technique is that of **co-training**, **tri-training**, etc., (Blum & Mitchell, 1998) In these approaches, multiple predictors are used to automatically label an unlabelled data set. The hope is that these different predictors will have different "views" of the data, i.e., they will use different kinds of features or search strategies. In some cases, this can have connections with example selection, because agreement between different predictors can be used as a test of output correctness. Sarkar (2001) demonstrates that co-training can improve over a parser trained with only a limited amount of labelled seed data. Steedman et al. (2003b) show that co-training is more effective than self-training, and that it can improve the performance of a parser given access to a small amount of labelled seed data. Clark et al. (2003) found that co-training could improve the performance of taggers trained on a small amount of seed data, but not those trained on full-sized data sets. Søgaard & Rishøj (2010) demonstrate that semi-supervised learning via tri-training improves over both supervised and self-trained systems, and could improve over the accuracy of then state-of-the-art models trained on full data sets.

Expectation Maximization An alternative to automatic labelling is to use some variant of *expectation maximization*. Unlike automatic labelling, expectation maximization allows for the range of possible parses for a sentence, which are weighted according to their posterior distribution. In comparison with automatic labelling approaches, one might hope that the consideration of all possible parses for each sentence will allow patterns to emerge in the posteriors for the unlabelled sentences, even if the patterns come from parses that would not have been the 1-best output from any supervised parser based on the labelled data. The drawback of such methods can be that, in order to maximize joint likelihood over the data, one needs to work with a generative model, but generative models are typically not competitive with discriminative models for parsing (Collins, 2000; Charniak & Johnson, 2005).

Some expectation maximization-like approaches have been successful in parsing. Bacchiani et al. (2006) have reported improvements over a baseline generative parser by incorporating the weighted posteriors of the top 20 parses for each unlabelled sentence as training data for a re-trained generative parser. Suzuki et al. (2009) train generative models over the posterior over parses for unlabelled sentences, and then use the estimated generative model parameters as features in a new discriminative models. Deoskar et al. (2014) demonstrate that repeated Viterbi expectation maximization passes (Spitkovsky et al., 2010) out-performs a single pass of self-training. And, Coppola et al. (2011) show that the opinion of a supervised classifier and a generative model trained with expectation maximization on unlabelled data can outperform a supervised classifier alone, for the problem of prepositional phrase attachment.

In machine learning more generally, a number of approaches have arisen that attempt to allow the user to constrain expectation maximization training by specifying constraints, either from intuition or from a supervised statistical model (Chang et al., 2007; Ganchev et al., 2010; Bellare et al., 2009). Ganchev et al. (2009) have used this training method to train weakly supervised parsing models from unlabelled data. There is, however, no results yet that show that such a method can outperform state-of-the-art supervised models.

6.1.2.3 Better Use of Labelled Data

A final option for better out-of-domain parsing, of course, is simply to make better use of labelled data. The Charniak & Johnson (2005) parser discriminatively re-ranks the n-best output of the Charniak (2000) parser using a variety of structural and lexical features described in §6.2.1. The difference in F_1 between the two parsers when trained

and tested on the *Wall Street Journal* part of the *Penn Treebank* (Marcus et al., 1993) is about 1.6%F_1 (91.3 versus 89.7) (McClosky et al., 2006a). And, McClosky et al. (2006b) reports that, when these parsers are trained on the *Wall Street Journal* part, and tested on the *Brown* part, of the *Penn Treebank*, the difference between the two is about 1.9%F_1 (85.8 versus 83.9). Thus, by making better use of labelled data, we can improve out-of-domain parsing performance. And, in the context of semantic role labelling, Meza-Ruiz & Riedel (2009) show that, by modelling the several stages of semantic role labelling jointly, significant improvement out-of-domain can be achieved compared to a prediction cascade. In this vein, this chapter will adopt the strategy of demonstrating that better performance can be achieved on out-of-domain tests using only the labelled data.

6.2 Feature Sets

We investigate the interaction of three feature sets, in terms of their effect on parsing performance both in- and out-of-domain. The first is a phrase-structure feature set, based on the work of Collins (2000); Charniak & Johnson (2005); Huang (2008b). The second is a set of higher-order dependency features, based on the work of (McDonald & Pereira, 2006; Carreras, 2007; Koo & Collins, 2010). While it is known that such features improve parsing performance in-domain, we are not aware of any work that systematically demonstrates their effect out-of-domain. Finally, the third feature set contains a single feature expressing a score resembling the conditional probability of a given parse according to a strong generative paring model (Petrov & Klein, 2007b).

6.2.1 Phrase-Structure Features

Our phrase structure features are taken from the influential model of Charniak & Johnson (2005), which draws on Collins (2000). This was the feature set used by Huang (2008b). Some of Charniak & Johnson's (2005) features are omitted, with choices made based on the ablation studies of Johnson & Ural (2010). We will use XP to represent any internal non-terminal node. The phrase structure feature set, Φ_{phrase}, then contains:

- **CoPar** The depth (number of levels) of parallelism between adjacent conjoined XPs
- **CoParLen** The difference in length between adjacent conjoined XPs

- **Edges** The words or (part-of-speech) tags on the outside and inside edges of a given XP [2]
- **NGrams** Parent node P of a rule production, along with n-grams of the children of P
- **NGramTree** An n-gram of the input sentence, or the tags, along with the minimal tree containing that n-gram
- **HeadTree** A sub-tree containing the path from a word to its maximal projection, along with all siblings of all nodes in that path
- **Heads** Head-modifier bi-grams between each word and its head
- **Rule** A entire single rule production
- **Tag** The tag of a given word
- **Word** The tag of and first XP above a word
- **WProj** The tag of and maximal projection of a word

Several of the features are sometimes annotated with the parent category of the given phrase. **NGrams** and **Rule** have versions that indicate the head-word of the phrase. For alternative descriptions, see Charniak & Johnson (2005); Huang (2008b).

6.2.2 Dependency Parsing Features

McDonald et al. (2005) showed that chart-based dependency parsing could profitably be approached in a discriminative framework. In this early work, each feature function was restricted to only being able to reference a single head-modifier relationship. Such a feature, indicating a bigram dependency relationship between pos-tag pairs is called a **first-order** dependency feature, and is exemplified by **Child** below. Subsequent work looked at allowing features to access more complex, *higher-order* relationships, such as **Child+Sib** and **Child+GrandKid** below (McDonald & Pereira, 2006; Carreras, 2007; Koo & Collins, 2010). Such **higher-order dependency features** indicate pos-tag relationships beyond the bigram, such as trigram or 4-gram. Then, Zhang & McDonald (2012) went on to show how arbitrarily complex relationships could be incorporated on top of the original Eisner-algorithm backbone using a "cube" approximation similar to the one described in §6.3. Our work can be seen as a phrase-structure analog of this work.

[2] The tags on the outside edges of a given XP are not definitely known when we visit that XP's vertex. For this reason, Huang excludes them. We approximate a tag on the outside edge of an XP using the marginally most likely tag for that word according to the first-stage parser.

The dependency features that we will use are, Φ_{deps}, are:

- **Child** Parent and child
- **Child+Sib** Parent, child c, and c's inner sibling
- **Child+GrandChild** Parent, child, and grand child
- **Child+Sib+GrandChild** Parent, child c, c's inner sibling, and one of c's children
- **Child+GrandChild+GrandSib** Parent, child, grand child g, and g's inner sibling

Note that, here, notions of parent, child, etc., refer to relationships in the dependency tree induced from the CFG tree, rather than relationships between the nodes in the CFG tree itself. For reasons of efficiency, features are insensitive to arc labels (i.e., are *unlabelled*).[3] The dependency features induce m-grams, with m being either 2, 3, or 4, depending on the feature. With each m gram, we lexicalize up to a_m of the words, where $a_2 = 2$, $a_3 = 3$, and $a_4 = 1$. Each of these features has two versions. The *full* version uses the full lexical items and tags, and the *reduced* version uses the stems in place of the lexical items, and reduced tags in place of the tags.[4]

6.2.3 Generative Model Score Feature

Finally, we have a feature set, Φ_{gen}, with only one feature. That is, $\Phi_{\text{gen}}(y)$ will always have exactly one non-zero element, which is the logarithm of the MAX-RULE-PRODUCT score of the LA-PCFG parser. This score has the character of a conditional likelihood for the parse, and is described by Petrov et al. (2006); Petrov & Klein (2007b). The basic idea of the LA-PCFG parser of Petrov et al. (2006) is that each rule production $A \rightarrow B\ C$ in a binarized treebank is assumed to be an instance of a rule $A_x \rightarrow B_y\ C_z$, where x, y, z are latent states, from a finite set of possibilities, that are not observed directly in the data. An expectation-maximization-based procedure is run on the training data to estimate the posterior probability that each nodes is in each given latent state. Given that this parser makes use of automatically posited latent states, an important parameter is the number of *split-merge* iterations conducted during training and inference (Petrov et al., 2006). Typically set to 5 or 6, each split-merge iteration

[3] We are testing against the labelled dependencies of de Marneffe & Manning (2008). We found it impractical to extract these at parse time. Perhaps this could be improved with optimization of the dependency labelling routine, but we felt this would not be worth the time involved.

[4] Following Koo and Collins, the reduced tag is the first two letters of the tag, unless the tag is *PRP* or *PRP*$.

refines the set of latent categories, creating a larger grammar. Each refinement corresponds to a stage of inference, so that a model that is trained using 6 split-merge iterations will require one more iteration of a decoding loop at inference time than a model using 5 split-merge iterations. This distinction will be relevant later, since the LA-PCFG parser is the *first-stage* parser in our cube decoding architecture (see §6.3).

6.3 Cube Decoding

6.3.1 Non-Local Features

Recall the general framework for viewing structured prediction discussed in §2.1.4. Structured prediction is viewed as finding the highest-scoring hyper-path, through a packed forest, from a collection of nodes representing the input, to a designated target node. The generalized Viterbi algorithm is an exact dynamic programming algorithm that requires the assumption that each hyper-edge can be assigned a score independently of all other edges. In the case of CFG parsing, this translates to the requirement that each rule production can be assigned a score independently of any others. Since the input is constant across all proposed outputs, it is possible to refer to any part of the input without breaking the independence assumption. Suppose that $R(y)$ are the set of rule productions in the parse y, and $\mathcal{R}$ is the set of CFG rule productions (including those for pre-terminals). Then, in order for Viterbi-style dynamic programming to work, we require the existence of a function $\Phi' : \mathcal{X} \times \mathcal{R} \rightarrow \mathbb{R}^n$ such that:

$$\Phi(\mathbf{x}, y) = \sum_{r \in R(y)} \Phi'(r, \mathbf{x}) \tag{6.2}$$

Following Huang (2008a), those feature functions that factor according to the hyper-edges are called **local**, and that do not are called **non-local**.

Most of the features referred to in §6.2 are non-local. For example, the **NGramTree** feature of Charniak & Johnson (2005) indicates an n-gram of the input sentence, along with the minimal tree containing that n-gram. This can include phrasal nodes from any number of layers of rule productions. Also, **CoPar** measures the maximum depth at which at which two adjacent trees that are conjoined by a conjunction word (e.g., *and*, *or*) are identical. This again requires inspecting more than one level of rule production.

Crucially, in a phrase-structure parsing framework, any features that refer to the head word (or head part-of-speech) of a phrase are non-local. Consider the algorithm, shown as Algorithm 14, for finding the head word of a phrasal node. $find\text{-}head\text{-}child$

Algorithm 14 Finding the Head Word

```
1: procedure FIND-HEAD-WORD(find-head-child, P, t)
2:     if P is terminal then
3:         return P
4:     else
5:         P_h = find-head-child(P, t)
6:         return Find-Head-Word(find-head-child, P_h, t)
```

takes a pair of a phrasal node P and a tree t and returns the head child P_h of node P in tree t. In our implementation, as is usual, the head child of a CFG node can be found by looking only at the local rule production below P in t (Collins, 1999; Yamada & Matsumoto, 2003). Thus, if a feature were only to refer to the head *child* of a given phrasal node, that feature would still be local. But, as Algorithm 14 makes clear, the recursive nature of determining a head *word* (or *tag*) means that the head word of a phrasal node is, for any interior node, a non-local feature. The non-locality of features mentioning the head word is crucial for this chapter because of the emphasis on the inclusion of higher-order dependency features in a phrase-structure parsing model. Each of these features is non-local. Interestingly, as we will see in §6.3, the solution that allows first-order dependency features like **Child** also allows dependency features of arbitrary order.

6.3.2 The Cube Decoding Algorithm

In order to parse with non-local features, we cannot use the Viterbi algorithm. However, chart-based parsing with non-local features is possible using the **cube decoding** algorithm of Huang & Chiang (2007); Huang (2008b,a). The cube decoding algorithm, shown as Algorithm 15, is an inexact dynamic programming algorithm that relies on beam search to allow tractable parsing time.

Algorithm 15 Cube Decoding

```
1: procedure CUBE(H_x = (V, E), w, Φ_F, k)
2:     Let D be an empty map
3:     for v ∈ V in topological order do
4:         D(v) ← KBEST(v, H, D, k, w, Φ)
5:     return D(TOP)_1          ▷ Return first (highest-scoring) element from D(TOP)
```

The input to the algorithm is: 1) a hyper-graph $H_{\mathbf{x}} = (\mathcal{V}, \mathcal{E})$, which is a packed

forest compactly representing possible outputs for the input $\mathbf{x}$, 2) a weight vector, $\mathbf{w}$, 3) a feature function Φ, and 4) an integer beam width k. The nodes of the acyclic hyper-graph are visited in a topological order that starts with the nodes representing the input, and ends with the node *TOP*. (In the CKY algorithm, this corresponds to using spans of increasing sizes and, for each span, considering each node label for that span.) For each node v, although there are exponentially many possible derivations of (or, hyper-paths to) v, we only maintain the k highest-scoring derivations. These are assembled by the function called KBEST in Algorithm 15. We omit details of this algorithm here (the reader is referred to Huang (2008b,a)). However, the important point to note is that $\text{KBEST}(v, H, D, k\mathbf{w}, \Phi)$ returns the k-best derivations for the node v in the hyper-graph $H_{\mathbf{x}}$, according to weight function $\mathbf{w}$ and feature function Φ, given that the map from nodes to lists of k-best derivations so far is D. Along with each of the k-best derivations, the algorithm also records the score of the feature vector of that derivation. By caching these values, we prevent their needless re-computation later in the run of the algorithm. Only the derivations in D are used, which means that, in creating a derivation for node v, only the top k derivations for those nodes earlier in the traversal are available. Each call to KBEST involves k extractions from a priority queue. Huang (2008b,a) gives the time complexity of this algorithm as $O(|\mathcal{E}| + |\mathcal{V}|Fk\log k)$, where F is a constant that bounds the feature extraction time. This algorithm is not exact, in that it can make search errors. A **search error** is said to occur when there exists a candidate $y \in \mathcal{C}(\mathbf{x})$ that has a higher model score than the one that is returned by the approximate search algorithm, but search errors are not a great problem in practice (Zhang & McDonald, 2012).

Suppose we want to use the feature function $\Phi : \mathcal{Y} \rightarrow \mathbb{R}^n$, such that $\Phi(y)$ is a feature vector for the full parse y. Suppose $S(\mathcal{Y})$ is the set of sub-trees in y. In order to parse efficiently in the cube decoding framework, we must convert Φ into a function $\Phi_F : \mathcal{X} \times S(\mathcal{Y}) \rightarrow \mathbb{R}^n$. Φ_F must be constructed such that:

$$\Phi(y) = \sum_{t \in y} \Phi_F(\mathbf{x}, t) \tag{6.3}$$

Because we cache the feature vector of each derivation of each node v, this computation is not repeated. Thus, $\Phi_F(\mathbf{x}, t)$ only computes those features about the sub-tree t that were not computed yet for any sub-tree of t.

6.3.3 Pruning the Parse Forest

From an asymptotic perspective, the cube decoding algorithm runs in time which is linear in both $|\mathcal{E}|$ and $|\mathcal{V}|$. In the case of standard CKY parsing, the number of nodes, $|\mathcal{V}|$, for a sentence of length n is $O(n^2 g)$, where g bounds the number of non-terminal symbols in the grammar. For a binary branching tree, the number of edges, $|\mathcal{E}|$, is $O(n^3 g^3)$. These numbers imply that the $O(|\mathcal{E}| + |\mathcal{V}| F k \log k)$ run-time of the cube decoding algorithm is *tractable*, in the sense of polynomial, in n. However, of course, simply having a polynomial run-time does not imply that the algorithm will be fast enough for any particular purpose in practice. In order to make the cube decoding algorithm fast enough to be at all practical, it is necessary to prune $|\mathcal{V}|$ to be something like linear in n. (In fact, the cube decoding algorithm remains a very slow algorithm, relative to the shift-reduce alternatives. See §6.3.5 for discussion, and §6.5.2.3 for timing results.) Pruning of the packed forest can be accomplished by first running a Viterbi-style **first-stage parser** over the sentence, for which we use the LA-PCFG parser discussed in §6.2.3, and pruning edges according to a figure-of-merit (Huang, 2008b; Charniak & Johnson, 2005). Since the LA-PCFG parser uses binary trees internally, but our features refer to de-binarized trees. So, we prune before debinarizing.

6.3.4 Margin-Based Training with the Cube Decoding Algorithm

Huang (2008b) trains his cube decoding parser using the perceptron. We will focus on training with the linear SVM objective. In the case of the perceptron, training requires two candidate outputs: the gold candidate and the prediction under the current weight vector. In the case of linear SVM training, we also require two candidate outputs: the gold candidate and the max-loss candidate.

6.3.4.1 The Oracle Candidate

As for the gold candidate, note that this candidate may not be reachable from the pruned parse forest returned by the first stage parser (§6.3.3). It is well-established wisdom in the NLP and machine learning communities that, when using an approximation to the full output space, often better results can be obtained by treating as the gold candidate the best *reachable* candidate, rather than the true gold. This best candidate is customarily called the **oracle** candidate. Huang (2008b) presents an algorithm for constructing an oracle from a packed forest. It exactly returns the candidate from the parse forest with the highest F_1 score. We use this same algorithm. Though we are

primarily interested in dependency loss functions in this chapter, our initial implementation of the oracle construction function was done to return the best tree by F_1, which usually produces a tree with near-perfect dependency accuracy.

6.3.4.2 The Max-Loss Candidate

The max-loss candidate is the maximizing argument of (2.20). That is:

$$\tilde{y} = \arg\max_{y \in \mathcal{C}(\mathbf{x})} \{\{\langle \mathbf{w}, \Phi(y) \rangle + \rho(\mathbf{y}, y)\} - \langle \mathbf{w}, \Phi(\mathbf{y}) \rangle\} \tag{6.4}$$

We saw in (6.5) that, in order to be compatible with the cube decoding algorithm, the feature function must factor according to the sub-trees of the parse. Similarly, given a desired cost function $\rho : \mathcal{Y} \times \mathcal{Y} \rightarrow \mathbb{R}$, in order for this function to be used in a cube decoding framework, we require the existence of a factored cost function $\rho_F : \mathcal{Y} \times S(\mathcal{Y}) \rightarrow \mathbb{R}$ such that:

$$\rho(\mathbf{y}, y) = \sum_{t \in y} \rho_F(\mathbf{y}, t) \tag{6.5}$$

Interestingly, although we use a CFG parsing algorithm, we can use a dependency-based loss function. This would not be possible if the loss function were required to factor according to the rule productions of the parse, as is the case in CKY-based SVM formulations (Taskar et al., 2004; Tsochantaridis et al., 2005).

6.3.4.3 Probabilistic Cube Decoding is Not Yet Possible

Now that the mechanics of perceptron and linear SVM training for a model using a cube decoding architecture have been introduced, we can discuss the contrast between these model types and probabilistic training. It is currently not known how to train a cube decoding model using a maximum entropy conditional likelihood objective. There are two principal problems in this regard. First, as usually construed, the use of conditional maximum entropy requires the computation of the normalization factor (2.7), which, for a sentence $\mathbf{x}$ and a given weight vector, is the sum of the potentials of all parses in $\mathcal{C}(\mathbf{x})$. Second, we require the ability to sum up the contributions to the expected feature vector. Computing this sum in a framework in which a beam is being applied, especially such an extreme beam, is not well understood, although Gimpel & Smith (2009) have suggested a technique for approximating the normalization factor, and shown its connections to weighted dynamic programming (Eisner et al., 2005), but have not given any experiments to demonstrate the effectiveness of their method.

6.3.5 Cube Decoding versus Alternative Parsing Algorithms

Standard CKY-based parsing algorithms, in which each node in the hyper-graph is *constituent* is characterized as a pair of i) *label* and ii) *span* cannot accommodate the use of non-local features, including head-modifier relationships. The basic CKY algorithm for binary trees, as noted, runs in time $O(Gn^3)$. Collins (1997) introduced the idea of modifying the dynamic programming for parsing, so that each node in the graph is actually a triple of i) *label*, ii) *span*, and iii) *head word*. This idea has also been adopted by Hockenmaier & Steedman (2002); Clark & Curran (2007). The inclusion of the head word into the dynamic programming factorization increases the run-time of the algorithm to $O(Gn^5)$, and thus requires heavy pruning. However, while allowing an additional important class of features (including bi-lexical head-modifier dependencies), this kind of factorization does not allow the other non-local features of §6.2.1. The initial means of using non-local features was to re-rank an k-best list of candidates (Collins, 2000; Charniak & Johnson, 2005). Of course, decoding time of the re-ranking stage is linear in k. The problem with re-ranking is that often-times the list has much redundancy, and the best option in the list, especially for longer sentences, can be quite far from the gold parse.

Another option for using non-local features in a phrase-structure parser is to use a **shift-reduce** phrase-structure parser Sagae & Lavie (2005); Wang et al. (2006); Zhang & Clark (2009, 2011); Zhu et al. (2013). Such a system works like a shift-reduce dependency parser (Yamada & Matsumoto, 2003; Nivre & Scholz, 2004), storing the input sentence as a queue, from which words are *shifted* onto a working stack, on which sub-trees are combined (or, *reduced*) to create larger ones. The run-time of such a system in terms of the input size and grammar constant is $O(Gn)$, which is much faster than chart-based parsers. If a beam width of k is used (as opposed to maintaining only a single working hypothesis), then the run-time becomes $O(Gnk \log k)$. In the case of a shift-reduce system, it is possible to look arbitrarily deeply at any aspect of those sub-trees already built. As such, features which would not be allowed in a vanilla CKY-style parsing system can be easily included in a shift-reduce system. For example, it is common to include a feature that references the head words (or tags) of the top two sub-trees on the stack. The use of the head words of the two sub-trees combined in a reduce operation is equivalent to using bi-lexical dependencies. Zhang & Clark (2009, 2011) also include features that reference the head words (or tags) of the top three constituents on the stack. This is very similar, although perhaps not exactly equivalent,

to the use of higher-order dependency features.[5]

Compared to cube decoding, shift-reduce parsing is faster and may be a better choice in practice. However, for experimenting with feature sets, we thought that the chart-based cube decoding framework would be a better choice. First of all, although the algorithm is inexact, search errors are very rare (Zhang & McDonald, 2012). Thus, we can compare features in a more idealized environment. Second, using this framework allows us to examine the well-known feature set of Collins (2000); Charniak & Johnson (2005). The Charniak & Johnson (2005) parser has been the basis for much work in NLP, especially in out-of-domain parsing settings. It is difficult to use this feature set in a shift-reduce setting, because trees in that case must be binarized. Finally, the use of the cube decoding framework allows us to investigate the inclusion of the generative model score feature that has been common in many discriminative parsing model. This allows us to quantify the contribution of the generative model to overall parse accuracy. In other words, it allows us to quantify how much performance *lacks* in a framework that does not include a generative first-stage parser as part of its architecture.

6.4 Feature Bagging and Model Combination

6.4.1 The Problem of Under-Training

We want to examine the combinations of parsing feature sets. Each of these feature sets constitutes a fairly good basis for a model on its own. We thus might run the risk of feature **under-training**, a problem was discussed in the context of conditional random fields by Sutton et al. (2005). Under-training refers to a situation in which feature weights are not set as strongly as they should be, because there are many other features that can also be used to explain the training set. In the case of a conditional likelihood model, this means that making the correct output most likely might only require each of the features to have small magnitudes. In the case of SVM training, it means that the required margin for each example can be achieved with small weight magnitudes. The problem with under-training is that, while many features may be active for the training examples, it might be that on test examples, especially on out-of-domain tests, some features used in training might not be present, and other features that are

[5]The lack equivalence comes from the fact that the relationship of the third sub-tree on the stack from the first two is not yet determined. In contrast, in the features of §6.2.2, the relationship is always included as a part of the feature.

present might have weights with smaller magnitudes than they should have. Sutton et al. (2005) refer to a case discussed by Pomerleau (1995) in which a driving robot trained with a neural network came to rely entirely on a single feature, the location of a ditch by the right side of the road, during training, and was unable to perform during a test in which it was asked to drive in the opposite direction, in which the ditch was no longer to its right. In the case of discriminative parsing models that re-rank generative parsing models, there is, as with Pomerleau's (1995) example, a very strong feature that the re-ranked model might come to rely on. That feature is the model score of the underlying generative parser (Φ_{gen}, here). This single feature actually represents a sophisticated statistical collation of very many features—those on which the generative model's score is based—and so is much more indicative than any boolean feature describing some raw characteristic of the input.

Sutton et al. (2005) suggest, as an antidote to under-training, a strategy of **feature bagging**. Feature bagging involves splitting one's entire feature set into n groups, or *bags*, of features. Then, for CRFs, one trains CRF models containing the n subsets (bags) of features to create component models. Then, finally, one uses the conditional probabilities of these smaller models as features in a new CRF model, which is then the one used for prediction. Sutton et al. (2005) show examples in which models trained using feature bagging outperform those trained without. Smith et al. (2005) show that this architecture can be used to train CRF models without a regularization. This allows "parameter-free" training, as it obviates the need to tune the regularization parameter λ.

6.4.2 Model Combination by Minimum Error-Rate

Since we are not using CRF training, we cannot adopt the exact probabilistic model combination exact that Sutton et al. (2005) used. Instead, we will train component models using the linear SVM training objective (2.28). The scores of these component models could then be used as features to a model combination trained using another linear SVM. However, given that the combined models will only have up to three components, we were concerned that SVM training with such few features might be somewhat unstable. When training a model for small number of features (around 12 or less), an interesting and very accurate option is to simply set the weights directly to minimize test error on the training set according to one's chosen error metric.

6.4.2.1 The Form of the Combination

Suppose we have some **expert** functions, $(h_n)_{n\in[N]}$, $h_n : \mathcal{Y} \to \mathbb{R}$, each of which takes an output and returns a real number representing the quality of that output according to some model.[6] Given **combination weights** $\mu = (\mu_n)_{n\in[N]}$, $\mu_n \in \mathbb{R}$, we can combine the opinions in turn in another linear combination as $\Sigma_i \mu_i h_i(y)$. Our prediction strategy would then be:

$$\hat{y}_{(h_n),(\mu_n),\mathcal{C}}(\mathbf{x}) = \arg\max_{y\in\mathcal{C}(x)} \sum_{n\in[N]} \mu_n h_n(y) \tag{6.6}$$

Here, the component experts h_n will be linear models associated with the feature sets Φ_{phrase}, Φ_{deps}, and Φ_{gen}, trained using maximum-margin training.

We should perhaps take a moment to distinguish this sort of model combination from another sense of the term "model combination" that exists in the parsing literature (Henderson & Brill, 1999; Zeman & Žabokrtskỳ, 2005; Sagae & Lavie, 2006; Fossum & Knight, 2009; Zhang et al., 2009). The works referred to combine the output of diverse parsers using the 1-best or n-best outputs from a range of parsers, with have both heterogeneous models and heterogeneous search strategies. Our method simply involves a combination of model scores, but the same search strategy is used for each model. (We do not claim one method is better than the other, but only note the difference.) Somewhat similarly, Petrov (2010) presents a model combination of generative LA-PCFG parsers, in which the models are combined with equal weights (i.e., $\mu_n = 1$ for all n) and a single search strategy is used.

6.4.2.2 Training by Minimum Error-Rate

The setting of weights to directly minimize an error metric on the training set is called **minimum error-rate training** (Och, 2003). Here, space of tuning parameters is searched to find combination weights that lead to predictions with the lowest loss on the training set. Such an objective is non-convex, and would be very difficult to search naively. However, Och (2003) uses an n-best version of the decoding problem, which allows a piece-wise linear representation of the function to optimize, which allows fast and exact search in any single-dimension search direction. However, as the problem is

[6] More positive values should correspond to better outputs according to the model and more negative values should correspond to worse outputs. In practice, though, the system would work just as well if more negative values correspond to better outputs, and more positive values correspond to worse outputs, so long as the sign of the combination weights are reversed.

non-convex, it is still not always possible to find the globally optimal parameter setting, because a different ordering of search directions will produce different results.

We chose to use simple grid search, as opposed to Och's (2003) algorithm. It is much simpler to implement—especially because it allows us to avoid alternating between n-best and full prediction stages, and because it seemed to work well for the small number of parameters tuned. Parameters are fixed in a greedy fashion. In order to fix $(\mu_1, \cdots, \mu_N)$, we first set $\mu_1 = 1$. Then, for $n > 2$, choose:

$$\mu_n = \arg\min_{\mu} E(\{\mu_1, \ldots, \mu_{i-1}, \mu, 0, \ldots, 0\}) \tag{6.7}$$

In all experiments, the order in which the combination weights are set (if present) are: Φ_{phrase}, Φ_{deps}, and then Φ_{gen}. Tuning is performed on 400 examples taken from section 21. These examples are withheld in all earlier rounds of training, so that neither the LA-PCFG parser nor any of they discriminative models have seen these examples.

6.5 Experiments

6.5.1 Methods

Data Sets The data sets involved were the *Wall Street Journal* (WSJ) and *Brown* (BROWN) portions of the *Penn Treebank* (Marcus et al., 1994). The training/test splits for these results are standard. For the WSJ, we use sections 2-21 for training, 22 for tuning, and 23 for reporting results. For BROWN, we use Gildea's (2001) sections 0-7 for training, 8 for tuning, and 9 for reporting results. The sizes of these data sets is shown in Table 6.5.1.

Data Set	Train Size	Tune Size	Test Size
WSJ	39,832	1,700	2,416
BROWN	19,740	2,078	2,425

Table 6.1: Data set sizes for experiments in this chapter.

MERT *Tuning Set* In order to tune the model weights in the MERT stage, we withheld 400 examples from the end of both the BROWN and WSJ training sets, from both the generative and SVM stages of training. This means that all training conducted was done with data from the training sets, and not from the tuning or test sets. These withheld examples were only used to train the MERT combination weights for combination

models, but not to train weights during the SVM phase. This means that models trained using a MERT combination actually have access to 400 extra examples. This might be argued to bias results towards preferring MERT combination, since they have access to more training data. However, these extra examples are only used to tune 1 or 2 combination weights, so this was judged to be a negligible advantage, and greatly simplified the experimental structure.

Training Set Preparation In order to create first-stage parsers for the training stage of the discriminative parser, we use a jack-knifing procedure (Collins, 2000; Charniak & Johnson, 2005). Each parser was an LA-PCFG parser trained 5 iterations of the split-merge procedure (see Petrov et al. (2006)). During training, decoding is done with a beam width of 12 individual CFG trees per node for phrase-structure feature sets, and 5 individual unlabelled dependency structures per node (cf. §6.3) for the dependency-only feature set Φ_{deps}.

SVM *Training Details* Linear SVM training is done using the objective (2.28), which we have discussed several times. As the cost function, ρ, we use arithmetic average of the number unlabelled dependency mistakes plus the number of labelled dependency mistakes. This objective function is optimized using the IPM distributed dual averaging algorithm (Algorithm 10) discussed in §5. Each case of training uses 12 nodes (computer cores), using 25 epochs through the data, with parameter mixing between nodes after every epoch through the data. These experiments were run on a shared memory multi-core computer containing 16 1.8 Ghz AMD Opteron processors. Models containing Φ_{phrase} or Φ_{deps}, but not both, were trained with 40 GB of RAM and models containing both Φ_{phrase} and Φ_{deps} were trained with 50. The iteration used is the one that performs best on the tuning set. A filter was applied to the features. We kept only features that were seen in candidate parses for at least 3 inputs from the training set. The regularization parameters λ used for each training set in $(\text{BROWN}, \text{WSJ})$ was taken from the optimal values found determined in the *n*-best re-ranking experiments in §3.

MERT *Training Details* For the MERT-trained model combinations, MERT tuning was done on using the 400 held-out examples for the given training set, according to the method described in §6.4.2.

Test Metrics The test loss functions used are shown in Table 6.5.1.

Function name	Description
F_1	Measures harmonic average of precision and recall of the number of labelled brackets in the predicted parse compared to the gold parse.
UAS	The percentage of unlabelled dependencies correct in the dependency tree extracted from the predicted phrase-structure parse.
LAS	The percentage of unlabelled dependencies correct in the dependency tree extracted from the predicted phrase-structure parse.

Table 6.2: Loss functions used to report results of parsing experiments.

Parameter Settings For all accuracy experiments, we use a beam width (i.e., the k of Algorithm 15) of 12. The cube pruning algorithm is set to output a maximum of 6 brackets per word. All training is done using forests output by a 5 split-merge LA-PCFG parser (see §6.2.3).

Significance Tests For significance tests reported in the next section, we used the version of the paired bootstrap (Efron & Tibshirani, 1994) recommended in Berg-Kirkpatrick et al. (2012).

Differences from Coppola & Steedman (2013) The results presented in this chapter are qualitatively the same but do differ slightly from those reported in Coppola & Steedman (2013). This is because these results constitute new runs of the training and testing procedures, with slight changes to certain parameters. For one, we switched to the regularized SVM training objective to the prediction-based MIRA learning algorithm of Crammer et al. (2006). Second, we broke the training data into distinct shards, in accordance with the IPM algorithm analyzed in §5. Third, we made slight changes to the MERT tuning stage. In Coppola & Steedman (2013), tuning was done using n-best lists, whereas below tuning is done directly with the cube decoding algorithm. It would seem that none of these changes should work towards or against any result with respect to the differences in feature sets, but only produce slight fluctuations.

6.5.2 Results

We begin by comparing accuracies of the various feature sets (§6.5.2.1), and then move on to some experiments that document the variation in parsing speed versus accuracy as parameters of the parsing algorithm are varied (§6.5.2.3).

6.5.2.1 Accuracy

Initial Presentation of the Results We begin by comparing accuracies. The test-time prediction results for the final test sets are shown in Tables 6.3 to 6.10. Tables 6.3, 6.5, 6.7 and 6.9 depict the performance results, while Tables 6.4, 6.6, 6.8 and 6.10 depict the results of the significance tests done on the important model comparisons. We begin with numerical charts that convey the results in detail. Some salient points from this data will be illustrated in graphical charts later in this section.

Bearing on the question of the effectiveness of feature bagging (§6.4.2), charts distinguish models trained using a single round of SVM training, denoted *Single*, and those trained using a MERT model combination, denoted MERT. For models containing only a single atomic feature set (i.e., either Φ_{phrase}, Φ_{deps} or Φ_{gen}), the only possibility is to have been trained in a single round of training, and so the entries in MERT category for these models are empty. The model *Type* of each model is shown on the left. Those models which contain only the generative model score feature are given type *G*. Those models using only discriminative features are given type *D*. Those models using a mix of generative and discriminative features are given type *G+D*. The generative/discriminative combinations are the strongest, but we make the distinction between model type in order to highlight the relative contributions of the generative and discriminative components. The best score for each metric overall is indicated by **bold face**, while the best score for a fully discriminative model is indicated by being underlined. In the architecture we have used, the generative model score feature can be used at no additional cost. However, in other architectures, such as shift-reduce parsers Zhang & Clark (2009, 2011); Zhu et al. (2013), the generative model score is not available, and so we want to quantify what is lost by these models. We compare two versions of each generative model. One uses 5 rounds of SM training for the Petrov et al. (2006) parser, and the other uses 6. Since the best setting for each of these data sets should either be 5 or 6, we can be quite sure that the best setting has been tried for each train-test pair, and so the strength of the generative component is not being under-estimated.

		Training Method					
		Single			MERT		
Type	Model	F_1	UAS	LAS	F_1	UAS	LAS
G	gen5	89.7	93.1	90.9	—	—	—
	gen6	90.4	93.6	91.3	—	—	—
D	phrase	90.9	93.7	91.0	—	—	—
	deps	85.6	93.0	88.1	—	—	—
	phrase+deps	91.3	94.3	91.7	91.2	94.2	91.6
G+D	phrase+gen5	91.6	94.4	92.4	91.6	94.3	92.3
	phrase+gen6	91.8	94.6	92.5	92.0	94.6	92.6
	phrase+deps+gen5	91.7	94.7	92.6	91.9	94.7	92.7
	phrase+deps+gen6	92.0	94.8	92.7	**92.4**	**95.0**	**93.0**

Table 6.3: Cube decoding parsing results, for models trained on WSJ, and tested on WSJ.

Significance Tests The significance tests attempt to answer the following two questions:

1. Does the MERT feature-bagging strategy lead to improved models?

2. Does the addition of the higher-order dependency feature sets, Φ_{deps}, improve performance?

Towards this end, the first group of pairs tested in each significance test table compares combined models trained with the MERT combination to those trained in a single round of SVM training. The second group of pairs compares models that include Φ_{deps} to the corresponding models that do not. We write *nb.* a cell of the significance table to indicate that the performance of the model under the category *Better* was *not actually better* than that list under *Worse* on that metric and train-test pair.

Main Trends In the parsing literature, usually an absolute difference of about $.4-.5\%$ on any metric is considered interesting enough to warrant publication (cf. Charniak & Johnson (2005); McDonald & Pereira (2006); Koo et al. (2008)). Using this standard to evaluate the size of a performance increase, we can identify the following four trends in the results in Tables 6.3 to 6.10:

1. Among models using only discriminative features, $\Phi_{\text{phrase+deps}}$ always out-performs

Better	Worse	F_1	UAS	LAS
phrase+deps (MERT)	phrase+deps (Single)	nb.	nb.	nb.
phrase+gen5 (MERT)	phrase+gen5 (Single)	nb.	nb.	nb.
phrase+gen6 (MERT)	phrase+gen6 (Single)	**.011**	.384	.240
phrase+deps+gen5 (MERT)	phrase+deps+gen5 (Single)	.137	.269	**.047**
phrase+deps+gen6 (MERT)	phrase+deps+gen6 (Single)	**<.001**	**.016**	**.001**
phrase+deps (MERT)	phrase	**.018**	**<.001**	**<.001**
phrase+deps+gen5 (MERT)	phrase+gen5 (MERT)	**.001**	**<.001**	**<.001**
phrase+deps+gen6 (MERT)	phrase+gen6 (MERT)	**<.001**	**<.001**	**<.001**

Table 6.4: Cube decoding significance results, for models trained on WSJ, and tested on WSJ.

		Training Method					
		Single			MERT		
Type	Model	F_1	UAS	LAS	F_1	UAS	LAS
G	gen5	85.0	88.6	84.9	—	—	—
	gen6	85.2	88.9	85.1	—	—	—
D	phrase	85.5	89.1	84.8	—	—	—
	deps	81.2	88.8	82.5	—	—	—
	phrase+deps	86.0	89.6	85.5	86.0	89.7	85.5
G+D	phrase+gen5	86.9	90.0	86.6	87.2	90.0	86.5
	phrase+gen6	86.7	90.0	86.5	87.1	90.1	86.7
	phrase+deps+gen5	87.0	90.1	86.7	**87.4**	90.4	86.9
	phrase+deps+gen6	86.8	90.1	86.7	**87.4**	**90.6**	**87.2**

Table 6.5: Cube decoding parsing results, for models trained on WSJ, and tested on BROWN.

Better	Worse	F_1	UAS	LAS
phrase+deps (MERT)	phrase+deps (Single)	.440	.183	.439
phrase+gen5 (MERT)	phrase+gen5 (Single)	**.010**	nb.	nb.
phrase+gen6 (MERT)	phrase+gen6 (Single)	**.002**	.146	.090
phrase+deps+gen5 (MERT)	phrase+deps+gen5 (Single)	**.002**	**.003**	**.019**
phrase+deps+gen6 (MERT)	phrase+deps+gen6 (Single)	**<.001**	**<.001**	**<.001**
phrase+deps (MERT)	phrase	**<.001**	**<.001**	**<.001**
phrase+deps+gen5 (MERT)	phrase+gen5 (MERT)	**.016**	**<.001**	**<.001**
phrase+deps+gen6 (MERT)	phrase+gen6 (MERT)	**.004**	**<.001**	**<.001**

Table 6.6: Cube decoding significance results, for models trained on WSJ, and tested on BROWN.

		Training Method					
		Single			MERT		
Type	Model	F_1	UAS	LAS	F_1	UAS	LAS
G	gen5	87.4	90.3	87.3	—	—	—
	gen6	87.1	90.4	87.4	—	—	—
D	phrase	87.8	91.0	87.4	—	—	—
	deps	—	89.9	—	—	—	—
	phrase+deps	87.9	91.1	87.5	88.3	91.3	87.8
G+D	phrase+gen5	89.0	91.6	88.8	89.1	91.5	88.7
	phrase+gen6	88.7	91.6	88.8	88.6	91.6	88.8
	phrase+deps+gen5	89.1	91.7	88.9	**89.5**	92.0	89.3
	phrase+deps+gen6	88.7	91.7	88.9	89.1	**92.2**	**89.4**

Table 6.7: Cube decoding parsing results, for models trained on BROWN, and tested on BROWN.

Better	Worse	F_1	UAS	LAS
phrase+deps (MERT)	phrase+deps (Single)	**<.001**	**.036**	**.011**
phrase+gen5 (MERT)	phrase+gen5 (Single)	.442	nb.	nb.
phrase+gen6 (MERT)	phrase+gen6 (Single)	nb.	nb.	.270
phrase+deps+gen5 (MERT)	phrase+deps+gen5 (Single)	**.003**	**.001**	**<.001**
phrase+deps+gen6 (MERT)	phrase+deps+gen6 (Single)	**<.001**	**<.001**	**<.001**
phrase+deps (MERT)	phrase	**<.001**	**.017**	**.002**
phrase+deps+gen5 (MERT)	phrase+gen5 (MERT)	**<.001**	**<.001**	**<.001**
phrase+deps+gen6 (MERT)	phrase+gen6 (MERT)	**<.001**	**<.001**	**<.001**

Table 6.8: Cube decoding significance results, for models trained on BROWN, and tested on BROWN.

		Training Method					
		Single			MERT		
Type	Model	F_1	UAS	LAS	F_1	UAS	LAS
G	gen5	81.7	86.4	82.5	—	—	—
	gen6	81.3	86.0	82.0	—	—	—
D	phrase	82.3	87.1	81.9	—	—	—
	deps	79.1	86.7	80.5	—	—	—
	phrase+deps	82.4	87.2	82.0	83.0	87.8	82.6
G+D	phrase+gen5	83.5	87.8	83.9	83.4	87.7	83.8
	phrase+gen6	83.3	87.2	83.3	83.3	87.5	83.6
	phrase+deps+gen5	**83.8**	88.1	84.1	**83.8**	**88.3**	**84.4**
	phrase+deps+gen6	83.6	87.4	83.5	**83.8**	87.9	83.9

Table 6.9: Cube decoding parsing results, for models trained on BROWN, and tested on WSJ.

Better	Worse	F_1	UAS	LAS
phrase+deps (MERT)	phrase+deps (Single)	<**.001**	<**.001**	<**.001**
phrase+gen5 (MERT)	phrase+gen5 (Single)	nb.	nb.	nb.
phrase+gen6 (MERT)	phrase+gen6 (Single)	.258	**.008**	**.007**
phrase+deps+gen5 (MERT)	phrase+deps+gen5 (Single)	nb.	.124	**.036**
phrase+deps+gen6 (MERT)	phrase+deps+gen6 (Single)	.096	<**.001**	<**.001**
phrase+deps (MERT)	phrase	<**.001**	<**.001**	<**.001**
phrase+deps+gen5 (MERT)	phrase+gen5 (MERT)	<**.001**	<**.001**	<**.001**
phrase+deps+gen6 (MERT)	phrase+gen6 (MERT)	<**.001**	<**.001**	<**.001**

Table 6.10: Cube decoding significance results, for models trained on BROWN, and tested on WSJ.

Φ_{phrase} or Φ_{deps} by a significant margin. This trend is at least as strong on out-of-domain tests as on in-domain tests.

2. Among models using a generative/discriminative combination, $\Phi_{\text{phrase+deps+gen}}$ always out-performs its corresponding $\Phi_{\text{phrase+gen}}$ counterpart. The increase is not as dramatic as in the case of $\Phi_{\text{phrase+deps}}$ versus Φ_{phrase}, but still quite noticable. This trend is at least as strong on out-of-domain tests as on in-domain tests.

3. The models trained using the MERT combination are usually better than those trained with a single round of SVM training, but sometimes are worse. In some cases, especially when BROWN is the training set, the advantage to the MERT combination is rather large.

4. Significance tests find significant differences in most cases that differences exist. This may be due to the large sizes of the test sets used.

Graphical Visualization of Key Comparisons We now summarize key comparisons graphically in Figures 6.1 and 6.2. Each of the data points plotted shows a *difference* between scores. For example, the left side of Figure 6.1 shows the scores for $\Phi_{\text{phrase+deps}}$ less those for the Φ_{phrase} model for each train/test pair. The explanation of the other figures is analogous. All models in this section use the version of Φ_{gen} with 5 SM rounds of training for the LA-PCFG parser.

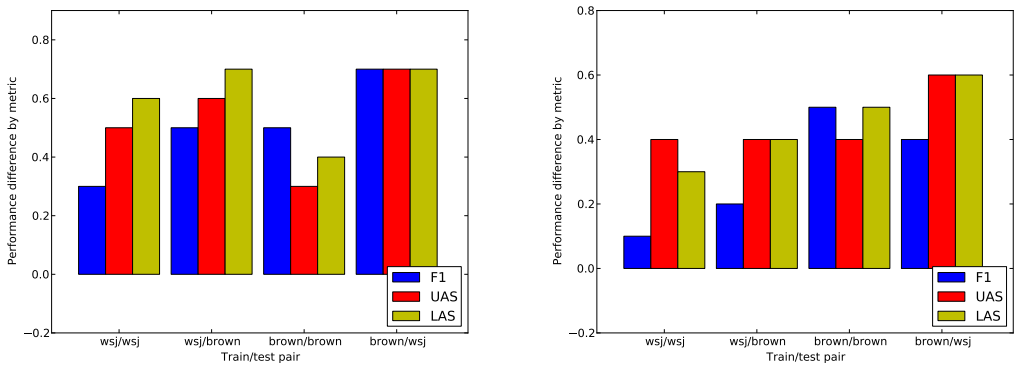


Figure 6.1: Difference made by adding the dependency features, for each train/test pair. The left figure shows scores for $\Phi_{\text{phrase+deps}}$ minus those for Φ_{phrase}. The right figure shows scores for $\Phi_{\text{phrase+deps+gen}}$ minus $\Phi_{\text{phrase+gen}}$. Combined models are trained using the MERT model combination.

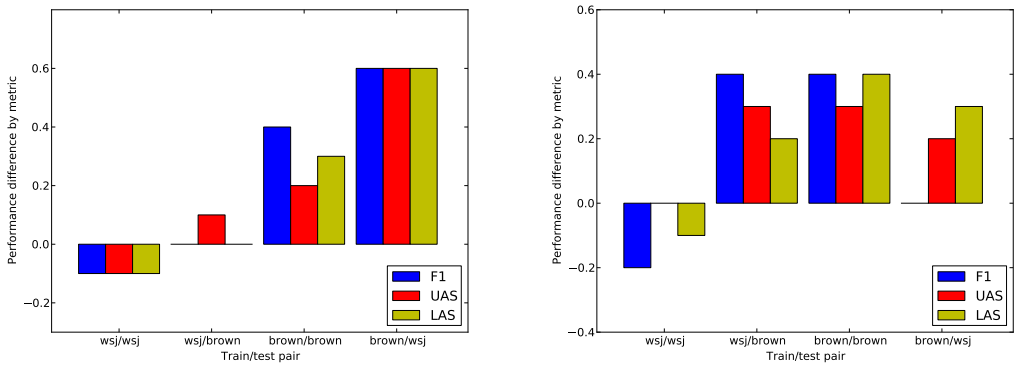


Figure 6.2: Difference made by using a MERT combination for training, for each train/test pair. The left figure shows scores for $\Phi_{\text{phrase+deps}}$ models trained with MERT minus scores for the same model trained with a single SVM round. The right figure shows the same comparison for models using the $\Phi_{\text{phrase+deps+gen}}$ feature set.

In Figure 6.1, we see consistent gains for adding in the Φ_{deps} feature set, especially when testing out of domain and especially when the training set is the BROWN corpus. In 6.2, we see that the difference made by the use of the MERT model combination is often important, and seems, again to be most important for the largest $\Phi_{\text{phrase+deps+gen}}$ feature set, as well as when the training set is BROWN. Perhaps the reason that these additions help more when training on the BROWN corpus is that it is smaller. Though there is not time to run more experiments now, it would be interesting to know whether a smaller version of the WSJ corpus would behave more like the BROWN corpus in this regard.

6.5.2.2 Ablation Tests

We have seen that higher-order dependency features seem to offer some improvement in the parsing of out-of-domain material. This is perhaps surprising because higher-order dependency features involve 3- and 4-grams of words and part-of-speech tags. One might expect that more specific features will be less likely to be of use on out-of-domain test material. Is it possible that a 3-gram dependency relationship between lexical items found in one domain is useful in another domain? In this section, we look more closely at what is happening. Specifically, we conduct ablation tests that look at which kinds of features are most helpful. There are two ways in which we will characterize feature types. First, we will characterize a feature by how many lexical items they refer to. Second, we will characterize it by its feature class.

Ablation According to Lexicalization Level On the number of lexical items referred to by a feature, note that, e.g., a 3-gram feature like **Child+GrandChild** will output several features for the same trigram. Suppose we have the dependency trigram:

$$((drove,VBD),(to,IN),(store,NN))$$

This trigram as input will result the firing of multiple overlapping features, including:

$$((drove,VBD),(to,IN),(store,NN)) \tag{6.8}$$

$$((drove,VBD),(to,IN),(-,NN)) \tag{6.9}$$

$$((drove,VBD),(-,IN),(-,NN)) \tag{6.10}$$

$$((-,VBD),(-,IN),(-,NN)) \tag{6.11}$$

Here, the $-$ masks the word, so that any word-tag pair with the same tag will activate the same feature. We can naturally say that feature (6.8) refers to 3 lexical items, while feature (6.11) refers to 0. The first set of ablation experiments with ablating features according to the number of lexical items referred to. The results are shown in Figure 6.3.

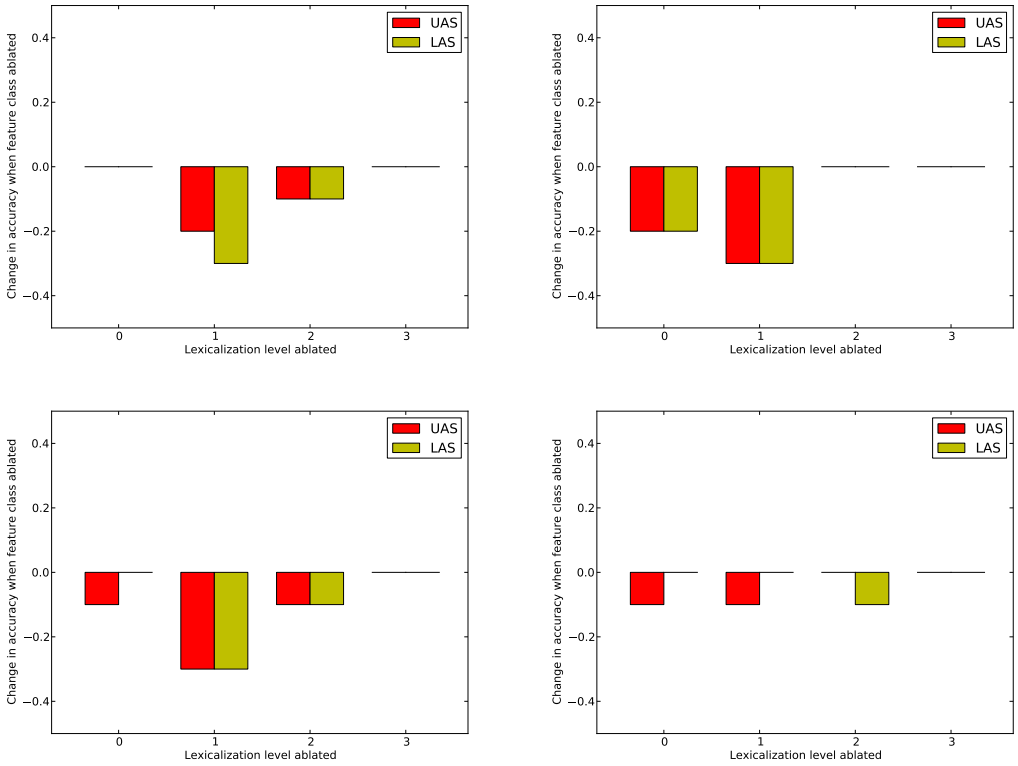


Figure 6.3: Results of ablating features according to the number of lexical items in the feature for each of the train/test set pairs. Top row shows train/test WSJ/WSJ and BROWN/BROWN (in-domain tests). Bottom row shows WSJ/BROWN and BROWN/WSJ (out-of-domain-tests).

It seems that features of lexicalization levels between 0 and 2, inclusive, play a role. Overall, the single most important level of lexicalization, both in- and out-of-domain is 1, i.e., with only a single lexical item mentioned in the feature.

Ablation According to Feature Class Having seen that features with higher levels of lexicalization do not seem to be important, we now ask whether it is still important to use higher-order dependency features. Figure 6.4 shows how performance is affected when each feature class is removed.

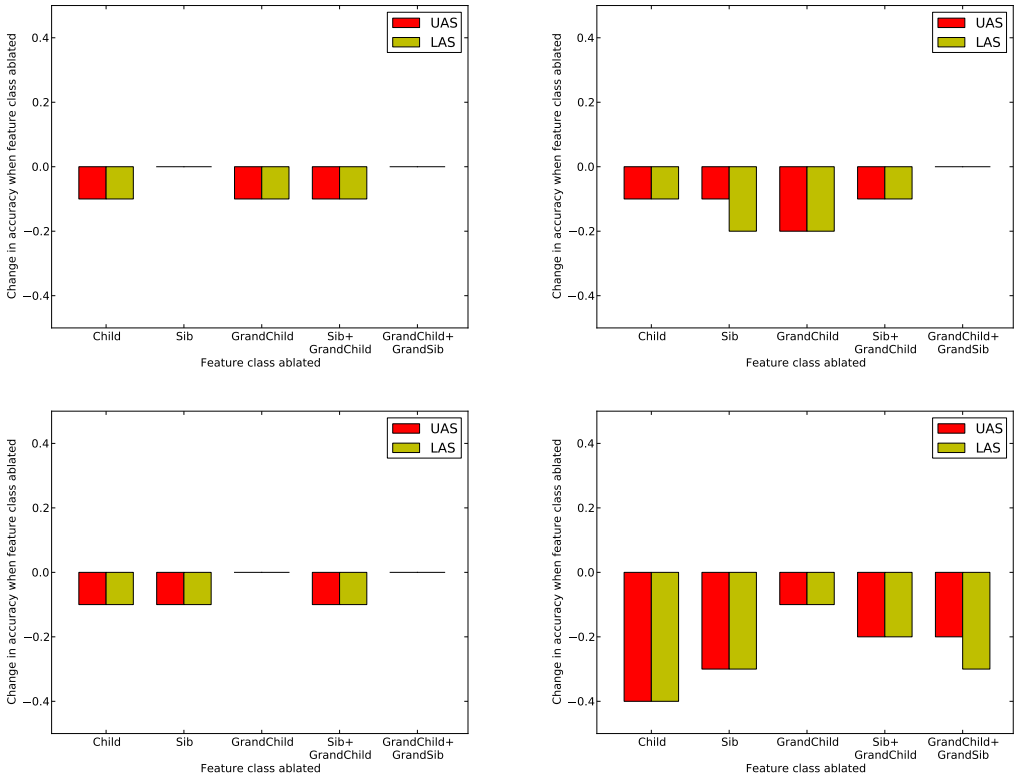


Figure 6.4: Results of ablating features according to feature class for each of the train/test set pairs. Top row shows train/test WSJ/WSJ and BROWN/BROWN (in-domain tests). Bottom row shows WSJ/BROWN and BROWN/WSJ (out-of-domain-tests).

We see that, for most feature classes, the removal of that class does indeed lead to some drop in performance. This suggests that features that mention 3- and 4-grams dependency relationships *are* useful, but, as we saw in the last series of ablation tests, not because they mention 3- and 4-gram relationships *between words*. It must instead be that these features capture relationships between *part-of-speech* tags, thus encoding something about the non-lexical structure of the language (written English, in this case).

6.5.2.3 Parsing Times

Having looked at the accuracies of the feature sets, we now look at how parsing time varies as certain experimental parameters vary. We will see that, although the accuracy of this parser is rather high, parsing times are extremely high and unlikely to be of use in practice. This is as is to be expected from the theoretical run-time complexity of the cube decoding algorithm (discussed on page 6.3.2), and, as we said, one major reason for being interested in this parser is that it allows us to investigate the combination of

generative and discriminative modelling techniques. All experiments listed here look at parsing section 22 of the WSJ data (the development test set), having trained on the WSJ data.

Time to Create Forests First Figure 6.5 at the time taken to create the pruned forests that are input to the discriminative stage of the parsing algorithm, as a function the number of split-merge iterations of the Petrov et al. (2006) LA-PCFG parser (see §6.2.3). It seems that creating forests using the 4SM versus the 5SM parser makes little difference, however, using the 6SM parser adds considerably to the time required to create the forests.

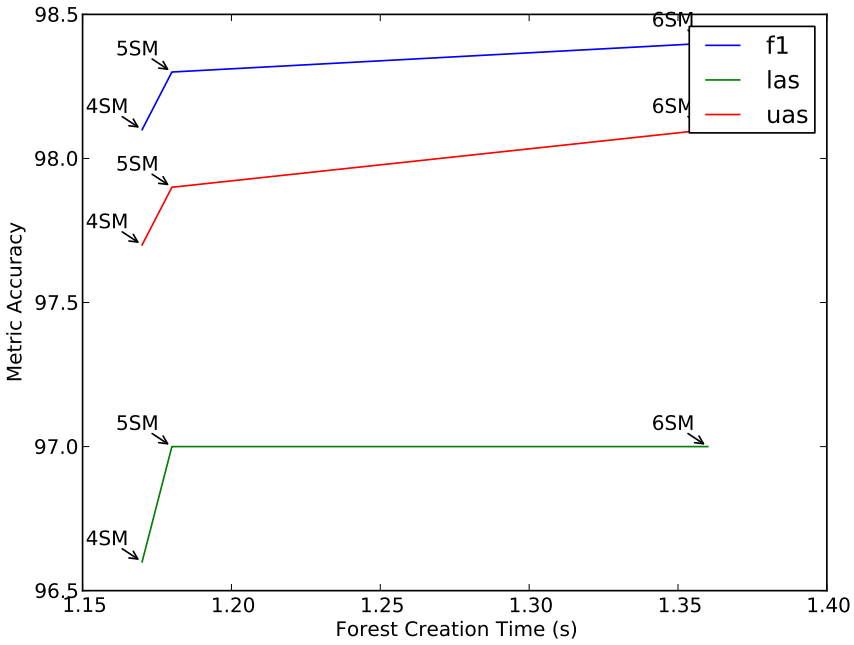


Figure 6.5: Oracle F_1 score of forests versus time to create forest, for different numbers of Split-Merge iterations of the Petrov et al. (2006) parser.

Nodes Per Word Here, we look at how varying the size of the forest, measured in number of nodes per word, from the first-stage parser affects parsing scores and parsing time. Figure 6.6 shows how oracle parse performance and actual cube decoding parser performance, with a beam width of 12, and discriminative-stage parsing time varies as a function of average forest size, which is varied by changing the pruning threshold. It seems that around 3.4 nodes per word provides a good trade-off between speed and accuracy.

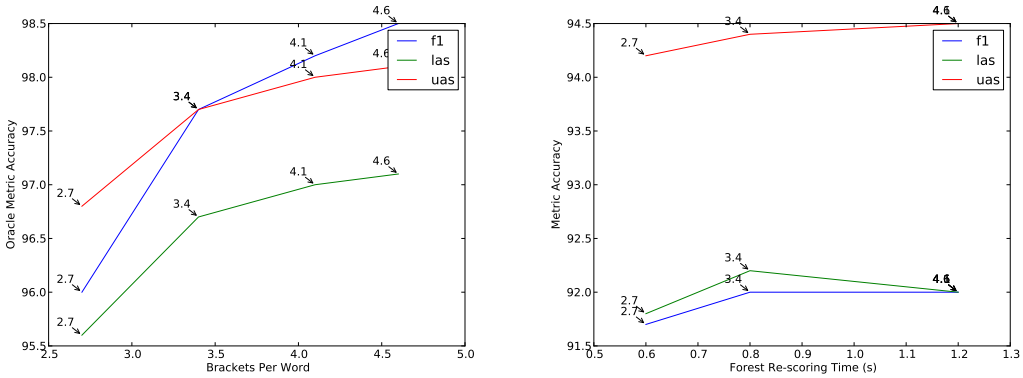


Figure 6.6: Oracle performance and actual parsing performance as a function of nodes per word in the parse forest passed on by the first-stage parser, for parser trained on WSJ, tested on WSJ development set.

Parsing Times by Feature Sets Here, using a beam of width 12 and an average forest size of 4.6 nodes per word, we look at the *total* parsing time for the various features sets tested (i.e., from sentence to parse). Figure 6.7 depicts parsing performance as a function of parsing time per sentence as the feature set varies. These parsing times are very slow compared to shift-reduce parsers (Nivre et al., 2007), and those that use super-tagging (Clark & Curran, 2007), which can parse on the order of a hundred or hundreds of sentences a second (depending on computing hardware). Thus, while our work with feature sets is hopefully of interest, the parser described does not seem to be efficient enough for use in practice.

Type	Model	WSJ
G+D	Huang (2008)	91.7
D	phrase+deps (MERT)	91.2
G+D	phrase+gen6 (MERT)	92.0
G+D	phrase+deps+gen (MERT)	**92.4**

Table 6.11: Comparison of constituency parsing results in the cube decoding framework, on the WSJ test set.

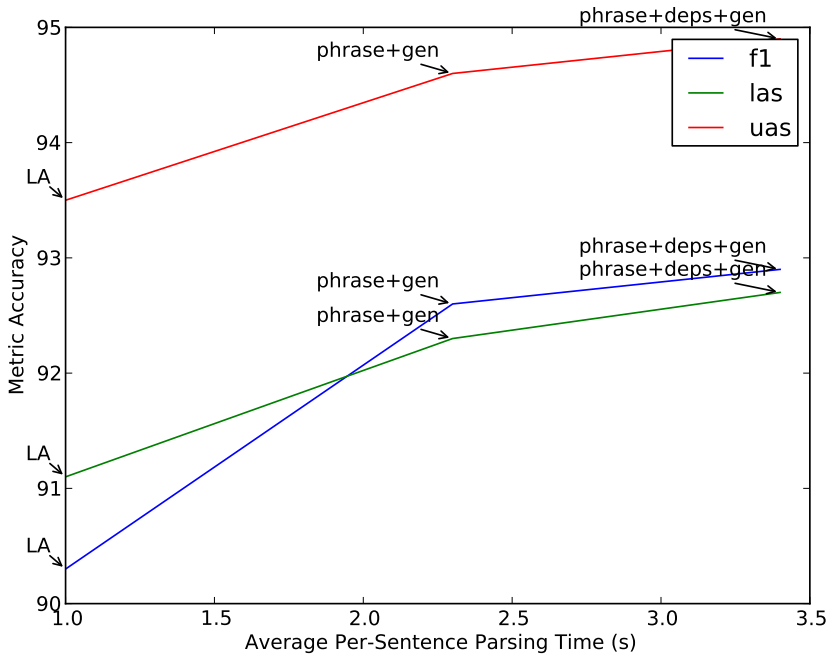


Figure 6.7: Average per-sentence parsing time for the full parser as a function of feature set, for models trained on WSJ, tested on WSJ development.

6.6 Discussion

Contribution to Knowledge about Parsing In terms of the discussion of §6.1, we have demonstrated a means of getting better out-of-domain performance using only supervised training. The accuracies of the models presented are compared to past work with the cube decoding algorithm in Table 6.11, and with past work on out-of-domain parsing in 6.12. We believe that the results presented in this chapter can help to guide out-of-domain parsing efforts in the future. For example, Huang et al. (2010) have

Parser	Training Data	BROWN F_1
CJ	WSJ	85.2
CJ	WSJ+NANC	87.8
CJ	BROWN	**88.4**
Our Best	WSJ	<u>87.4</u>

Table 6.12: Comparison of our best model, $\Phi_{\text{phrase+deps+gen}}$, on BROWN, with the Charniak & Johnson (2005) parser, denoted CJ, as reported in McClosky et al. (2006b). <u>Underline</u> indicates best trained on WSJ, bold face indicates best overall.

	$\Phi_{\text{phrase}}(\Phi_{\text{phrase+gen}})$	$\Phi_{\text{phrase+deps}}(\Phi_{\text{phrase+deps+gen}})$
Per-sentence re-scoring time	1.2 seconds	2.4 seconds
WSJ training set size	40,000 examples	40,000 examples
One training epoch	13.3 hours	26.6 hours
Two training epochs	26.6 hours	53.2 hours

Table 6.13: Statistics describing estimated sequential sub-gradient training time for Φ_{phrase} and $\Phi_{\text{phrase+deps}}$ models. (Note that adding the single feature Φ_{gen} does not add measurably to training time.)

shown that better supervised parsers can lead to better self-trained parsers, because self-training works better when the parser used for automatic labelling is stronger. Thus, improved supervised parsing, especially for out-of-domain results, can hopefully help improve semi-supervised parsing as well. Also, our ablation studies have shown that tri-lexical features are not that helpful, so these can probably be omitted in the future. A major limitation of our parser as it is is clearly its speed. This is not a practical parser for parsing, e.g., the world wide web. But, hopefully this work can inform the building of faster shift-reduce parsers (Zhang & Nivre, 2011; Zhang & Clark, 2011), by demonstrating what is gained by adding each feature set tested (or, what is lost by removing it). Furthermore, our demonstration that feature combinations can sometimes be useful may also be helpful in practical applications that must get the very best performance out of parsers.

Speed-Up in Development Time Using Iterative Parameter Mixing Prototyping and development of these models was done using multi-core training in the form of the iterative parameter mixing algorithm. In this passage, we will try to quantify the extent

to which the use of multi-core training sped up the development process. In actual fact, training times during development of the model and debugging of associated software were not carefully measured, since collection of such statistics are not usually the focus during the model development phase. However, we feel that the following calculations, based on data presented in this chapter and §5 are illustrative of the nature of the savings that were observed in practice.

When training the discriminative re-ranking component of our parser, the parse forests produced by the first-stage parser are computed once and cached. Thus, the significant part of training time is determined entirely by the time taken to discriminatively re-rank the parse forests, for which feature extraction is in turn the dominant factor. The average time needed by the discriminative component to re-rank a single forest (measured on WSJ 22) for the Φ_{phrase} and $\Phi_{\text{phrase+deps}}$ feature sets are shown in Table 6.13. From this average per-sentence parsing time, we can estimate the amount of time needed to make one or more passes over the training data, where we use the WSJ training set as an example. We saw in §5 that, using multi-core training with the IPM distribution strategy, one can get a model of fairly good quality with a budget of a single sequential pass through the data. In contrast, a sequential model needs at least about two epochs to reach a decent quality. During model development, we only need an estimate of the quality of a model, in order to determine whether there are bugs, or whether a new feature set is helping. Thus, during development, we only need to run IPM training for the time of a single epoch of sequential training time. But, using sequential training, we need to run for at least two epochs. So, we can say that the savings in training time, during the development phase, for training a single model is the difference in time between one and two sequential training epochs. From Table 6.13, we can see that, in the case of training a model with the feature set Φ_{phrase} (or $\Phi_{\text{phrase+gen}}$, since Φ_{gen} does not add measurably to training time), this amounts to a savings of about 13.3 hours. In the case of a model with feature set $\Phi_{\text{phrase+deps}}$ (or $\Phi_{\text{phrase+deps+gen}}$), the savings is about 26.6 hours. Clearly, these sorts of savings speed the development process, and show the benefits we derived from multi-core training during model development.

6.7 Conclusion

This chapter has demonstrated that higher-order dependency features benefit out-of-domain parsing, even when added to a state-of-the-art phrase structure parser along

the lines of Charniak & Johnson (2005); Huang (2008b). To our knowledge, our new parser constitutes a new state-of-the-art for the train-test domains studied (see Table 6.12). Through feature ablation tests, we have found that all feature classes seem to be important, and that lexicalization levels between 0 and 2 seem to be most important for out-of-domain accuracy. We have investigated the use of a MERT-based feature-bagging approach to training and shown that feature bagging gives a slight edge in performance. Finally, we have conducted performance experiments to demonstrate the parsing times under various parser parameter settings of a cube decoding parser.

Chapter 7

Conclusion

This thesis has studied, improved, and applied the iterative parameter mixing algorithm for distributed training of structured predictors for NLP.

In §4, we presented a novel algorithm for decentralized optimization of a regularized average loss function, the *distributed regularized dual averaging algorithm*. This algorithm improves over past work (Duchi et al., 2011b, 2012), in terms of convergence bound and algorithmic simplicity. And, it added the analysis of a regret bound (Lemma 2) not present in past work on composite functions (Duchi et al., 2011b), which was needed in §5.

In §5, we presented a novel theoretical and empirical comparison of the *iterative parameter mixing* (IPM) distributed optimization algorithm to the *single-mixture* optimization algorithm. The distinction in test performance between these algorithms has been a theme in recent work (McDonald et al., 2010; Hall et al., 2010; Simianer et al., 2012), but little has been known theoretically about where this difference comes from. We show that the IPM optimization algorithm is a convergent optimization algorithm—i.e., one which will converge on the true optimal objective value given enough time—while the single-mixture optimization algorithm is not. This suggests that the difference in test-time performance might be explained by the difference in optimization of the objective over the training set. The results of our experiments support this hypothesis by showing that better optimization of the training objective does indeed correlate with better test-time performance. We used this convergence analysis to justify distributed SVM training of structured predictors, which we showed led to better test-time performance than the perceptron algorithm, the only version of IPM to have been theoretically justified previously. Finally, we showed that distributed training can lead to better objective values reached if one has a budget of training time of

one or two sequential iterations.

In §6, leveraging the speed-up of distributed training for training with a time budget, we analyzed the use of higher-order dependency features in the context of out-of-domain phrase-structure parsing. We showed that these features lead to significant improvements in dependency recovery, even out-of-domain. This fact was previously unreported and led to a new state-of-art in accuracies for the popular train-test pairs investigated. Our work also showed that, in some cases, it is preferable to train models using a feature-bagging approach for the best test-time accuracy.

In terms of future work, we said that our bound in §5 for the IPM algorithm was pessimistic because was not a function of the similarity or dissimilarity between the functions on the various noes being optimized. Presumably, similarity between function values should lead to an easier optimization problem. Future work on understanding the iterative parameter mixing algorithm could look at exploiting similarity between the objective functions optimized at each node to achieve better bounds on convergence. For high-accuracy phrase-structure parsing, we have shown that both the addition of the generative model and higher-order dependency features both improve parse accuracy. The cube decoding algorithm is one way to combine both a generative model score feature along with non-local higher-order dependency features. Future work should investigate whether there are faster ways to combine these kinds of feature sets. Also, our ablation tests demonstrated that the benefit of higher-order dependency features comes entirely from those with 0, 1 and 2 lexical items mentioned. Future work should look at the speed-up achieved by using only these higher-order features in various parsing architectures.

Bibliography

Abernethy, J., Agarwal, A., Bartlett, P. L., & Rakhlin, A. (2009). A stochastic view of optimal regret through minimax duality. *arXiv*, 0903.5328.

Agarwal, A., Bartlett, P. L., Ravikumar, P., & Wainwright, M. J. (2012). Information-theoretic lower bounds on the oracle complexity of stochastic convex optimization. *Information Theory, IEEE Transactions on*, 58(5), 3235–3249.

Agarwal, A., Chapelle, O., Dudík, M., & Langford, J. (2011). A reliable effective terascale linear learning system. *arXiv*, 1110.4198.

Aho, A. V., & Ullman, J. D. (1972). *The theory of parsing, translation, and compiling*. New Jersey: Prentice-Hall, Inc.

Bacchiani, M., Riley, M., Roark, B., & Sproat, R. (2006). Map adaptation of stochastic grammars. *Computer Speech & Language*, 20(1), 41–68.

Baker, J. K. (1979). Trainable grammars for speech recognition. *The Journal of the Acoustical Society of America*, 65, S132.

Bansal, M., & Klein, D. (2011). Web-scale features for full-scale parsing. In *ACL*, 693–702.

Bellare, K., Druck, G., & McCallum, A. (2009). Alternating projections for learning with expectation constraints. In *UAI*, 43–50.

Ben-David, S., Blitzer, J., Crammer, K., & Pereira, F. (2007). Analysis of representations for domain adaptation. In *NIPS*. Cambridge, MA: MIT Press.

Berg-Kirkpatrick, T., Burkett, D., & Klein, D. (2012). An empirical investigation of statistical significance in NLP. In *EMNLP*, 995–1005.

Bertsekas, D. P. (1999). *Nonlinear Programming*. Belmont, MA: Athena Scientific, 2nd edition.

Bishop, C. M. (2006). *Pattern recognition and machine learning*. Cambridge: Springer.

Blitzer, J., McDonald, R., & Pereira, F. (2006). Domain adaptation with structural correspondence learning. In *EMNLP*, 120–128.

Block, H. D. (1962). The perceptron: A model for brain functioning. *Reviews of Modern Physics*, 34(1), 123.

Blum, A., & Mitchell, T. (1998). Combining labeled and unlabeled data with co-training. In *Proceedings of 11th Annual Conference on Computational Learning Theory*.

Bohnet, B., Farkas, R., & Cetinoglu, O. (2012). SANCL 2012 shared task: The IMS system description. In *Notes of the First Workshop on Syntactic Analysis of Non-Canonical Language, NAACL.*

Bordes, A., & Bottou, L. (2005). The Huller: a simple and efficient online SVM. In *ECML*, 505–512.

Bordes, A., Bottou, L., Gallinari, P., & Weston, J. (2007). Solving multiclass support vector machines with LaRank. In *ICML*, 89–96. ACM.

Bottou, L. (2004). Stochastic learning. In *Advanced lectures on machine learning*, (pp. 146–168). Springer.

Bottou, L., & Bousquet, O. (2008). The tradeoffs of large scale learning. In *NIPS*, 161–168.

Bottou, L., & Le Cun, Y. (2005). On-line learning for very large data sets. *Applied Stochastic Models in Business and Industry*, 21(2), 137–151.

Boyd, S., Parikh, N., Chu, E., Peleato, B., & Eckstein, J. (2011). Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning*, 3(1), 1–122.

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

Carreras, X. (2007). Experiments with a higher-order projective dependency parser. In *Proceedings of the CoNLL Shared Task Session of EMNLP-CoNLL 2007*, 957–961.

Cesa-Bianchi, N., Conconi, A., & Gentile, C. (2004). On the generalization ability of on-line learning algorithms. *Information Theory, IEEE Transactions on*, 50(9), 2050–2057.

Cesa-Bianchi, N., & Lugosi, G. (2006). *Prediction, learning, and games*. Cambridge University Press.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., & Gruber, R. E. (2008). Bigtable: A distributed storage system for structured data. *ACM Transactions on Computing Systems*, 26(2), 4:1–4:26.

Chang, K.-W., Srikumar, V., & Roth, D. (2013). Multi-core structural svm training. In *ECML*.

Chang, M.-W., Ratinov, L.-A., & Roth, D. (2007). Guiding semi-supervision with constraint-driven learning. In *ACL*, 280–287.

Charniak, E. (1997). Statistical parsing with a context-free grammar and word statistics. In *Proceedings of AAAI*, 598–603.

Charniak, E. (2000). A maximum-entropy-inspired parser. In *ANLP*, 132–139.

Charniak, E., & Johnson, M. (2005). Coarse-to-fine n-best parsing and MaxEnt discriminative reranking. In *ACL*, 173–180.

Chen, X., Lin, Q., & Pena, J. (2012). Optimal regularized dual averaging methods for stochastic optimization. In *NIPS*, 404–412.

Chiang, D. (2007). Hierarchical phrase-based translation. *Computational Linguistics*, 33(2), 201–228.

Chiang, D. (2012). Hope and fear for discriminative training of statistical translation models. *JMLR*, 13, 1159–1187.

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

Clark, S., & Curran, J. R. (2007). Wide-coverage efficient statistical parsing with CCG and log-linear models. *Computational Linguistics*, 33(4), 493–552.

Clark, S., Curran, J. R., & Osborne, M. (2003). Bootstrapping pos taggers using unlabelled data. In *CoNLL*, 49–55.

Cocke, J., & Schwartz, J. T. (1969). *Programming languages and their compilers*. Courant Institute of Mathematical Sciences, New York University.

Collins, M. (1997). Three generative, lexicalised models for statistical parsing. In *ACL*, 16–23.

Collins, M. (1999). Head-driven statistical models for natural language parsing. Doctoral Dissertation, University of Pennsylvania.

Collins, M. (2000). Discriminative reranking for natural language parsing. In *ICML*, 175–182.

Collins, M. (2002). Discriminative training methods for Hidden Markov Models: theory and experiments with perceptron algorithms. In *EMNLP*, 1–8.

Collins, M. (2005). Parameter estimation for statistical parsing models: Theory and practice of distribution-free methods. In *New developments in parsing technology*, (pp. 19–55). Springer.

Collins, M., Globerson, A., Koo, T., Carreras, X., & Bartlett, P. L. (2008). Exponentiated gradient algorithms for conditional random fields and max-margin markov networks. *JMLR*, 9, 1775–1822.

Coppola, G. F., & Steedman, M. (2013). The effect of higher-order dependency features in discriminative phrase-structure parsing. In *ACL Short Papers*, 610–616.

Coppola, G. F., Birch, A., Deoskar, T., & Steedman, M. (2011). Simple semi-supervised learning for prepositional phrase attachment. In *IWPT*, 129–139.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C., et al. (2001). *Introduction to algorithms*, volume 2. Cambridge: MIT Press.

Corp., S. G. I. (2011). World's largest coherent shared memory system used for multiple projects. Online at http://arxiv.org/pdf/1005.2012.pdf.

Crammer, K., Dekel, O., Keshet, J., Shalev-Shwartz, S., & Singer, Y. (2006). Online passive-aggressive algorithms. *JMLR*, 7, 551–585.

Crammer, K., & Singer, Y. (2002). On the algorithmic implementation of multiclass kernel-based vector machines. *JMLR*, 2, 265–292.

Cristianini, N., & Shawe-Taylor, J. (2000). *An introduction to support vector machines and other kernel-based learning methods*. Cambridge University Press.

Daumé III, H. (2007). Frustratingly easy domain adaptation. In *ACL*, volume 1785, 1787.

Daumé III, H., & Marcu, D. (2006). Domain adaptation for statistical classifiers. *JAIR*, 26, 101–126.

Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Le, Q. V., Mao, M. Z., Ranzato, M., Senior, A. W., Tucker, P. A., et al. (2012). Large scale distributed deep networks. In *NIPS*, 1232–1240.

Dean, J., & Ghemawat, S. (2008). Mapreduce: Simplified data processing on large clusters. *ACM*, 51(1), 107–113.

Dekel, O., Gilad-Bachrach, R., Shamir, O., & Xiao, L. (2012). Optimal distributed online prediction using mini-batches. *JMLR*, 13, 165–202.

Deoskar, T., Cristodoulopoulos, C., Birch, A., & Steedman, M. (2014). Generalizing a strongly lexicalized parser using unlabeled data. ACL.

Druck, G., & McCallum, A. (2010). High-performance semi-supervised learning using discriminatively constrained generative models. In *ICML*, 319–326.

Duchi, J., Hazan, E., & Singer, Y. (2011a). Adaptive subgradient methods for online learning and stochastic optimization. *JMLR*, 12, 2121–2159.

Duchi, J. C., Agarwal, A., & Wainwright, M. J. (2011b). Dual averaging for distributed optimization: convergence analysis and network scaling. *arXiv*, 1005.2012v3.

Duchi, J. C., Agarwal, A., & Wainwright, M. J. (2012). Dual averaging for distributed optimization: convergence analysis and network scaling. *Automatic Control, IEEE Transactions on*, 57(3), 592–606.

Dunlop, A., Bodenstab, N., & Roark, B. (2011). Efficient matrix-encoded grammars and low latency parallelization strategies for cyk. In *IWPT*, 163–174.

Efron, B., & Tibshirani, R. J. (1994). *An introduction to the bootstrap*, volume 57. CRC press.

Eisner, J. (1996). Three new probabilistic models for dependency parsing: An exploration. In *COLING*, 340–345.

Eisner, J., Goldlust, E., & Smith, N. A. (2005). Compiling comp ling: Weighted dynamic programming and the Dyna language. In *HLT-EMNLP*.

Finkel, J. R., Kleeman, A., & Manning, C. D. (2008). Efficient, feature-based, conditional random field parsing. In *ACL*, 959–967.

Fossum, V., & Knight, K. (2009). Combining constituent parsers. In *Proceedings of NAACL*, 253–256.

Francis, W. N., & Kucera, H. (1979). *Brown corpus manual*. Providence, RI: Brown University.

Freund, Y., & Schapire, R. E. (1999). Large margin classification using the perceptron algorithm. *Machine Learning*, 37(3), 277–296.

Ganchev, K., Gillenwater, J., & Taskar, B. (2009). Dependency grammar induction via bitext projection constraints. In *ACL*, 369–377.

Ganchev, K., Graça, J., Gillenwater, J., & Taskar, B. (2010). Posterior regularization for structured latent variable models. *JMLR*, 11, 2001–2049.

Gelsinger, P. P. (2001). Microprocessors for the new millennium: Challenges, opportunities, and new frontiers. In *Solid-State Circuits Conference, 2001. Digest of Technical Papers. ISSCC. 2001 IEEE International*, 22–25. IEEE.

Ghemawat, S., Gobioff, H., & Leung, S.-T. (2003). The google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, 29–43. ACM.

Gildea, D. (2001). Corpus variation and parser performance. In *EMNLP*, 167–202.

Gimpel, K., Das, D., & Smith, N. A. (2010). Distributed asynchronous online learning for natural language processing. In *CoNLL*, 213–222.

Gimpel, K., & Smith, N. A. (2009). Cube summing, approximate inference with non-local features, and dynamic programming without semirings. In *EACL*, 318–326.

Goffin, J. (1977). On convergence rates of subgradient optimization methods. *Mathematical programming*, 13(1), 329–347.

Gropp, W., Lusk, E., & Skjellum, A. (1999). *Using mpi: portable parallel programming with the message-passing interface*, volume 1. MIT press.

Guruswami, V. (2000). Rapidly mixing markov chains: A comparison of techniques. Online at cs.washington.edu/homes/venkat/pubs/papers.html.

Hajic, J., Smrz, O., Zemánek, P., Šnaidauf, J., & Beška, E. (2004). Prague arabic dependency treebank: Development in data and tools. In *Proc. of the NEMLAR Intern. Conf. on Arabic Language Resources and Tools*, 110–117.

Hall, K. B., Gilpin, S., & Mann, G. (2010). Mapreduce/bigtable for distributed optimization. In *NIPS LCCC Workshop*.

Hayashi, K., Kondo, S., Duh, K., Matsumoto, Y., Sudoh, K., Tsukada, H., Nagata, M., Wu, X., Matsuzaki, T., Tsujii, J., et al. (2012). The NAIST dependency parser for SANCL2012 shared task. In *Notes of the First Workshop on Syntactic Analysis of Non-Canonical Language, NAACL*.

Hazan, E., Agarwal, A., & Kale, S. (2007). Logarithmic regret algorithms for online convex optimization. *Machine Learning*, 69(2-3), 169–192.

Hazan, E., Kalai, A., Kale, S., & Agarwal, A. (2006). Logarithmic regret algorithms for online convex optimization. In *Learning theory*, (pp. 499–513). Springer.

Hazan, E., & Kale, S. (2011). Beyond the regret minimization barrier: an optimal algorithm for stochastic strongly-convex optimization. *JMLR-Proceedings Track*, 19, 421–436.

Henderson, J. C., & Brill, E. (1999). Exploiting diversity in natural language processing: Combining parsers. In *EMNLP*, 187–194.

Hindle, D., & Rooth, M. (1993). Structural ambiguity and lexical relations. *Computational Linguistics*, 19, 103–120.

Hiriart-Urruty, J.-B., & Lemaréchal, C. (1996). *Convex analysis and minimization algorithms: Part 1: Fundamentals*, volume 1. Springer.

Hockenmaier, J., & Steedman, M. (2002). Generative models for statistical parsing with combinatory categorial grammar. In *ACL*, 335–342.

Horn, R. A., & Johnson, C. R. (1985). *Matrix analysis*. Cambridge University Press.

Hsu, C.-W., & Lin, C.-J. (2002). A comparison of methods for multiclass support vector machines. *Neural Networks, IEEE Transactions on*, 13(2), 415–425.

Huang, L. (2008a). Forest-based algorithms in natural language processing. Doctoral Dissertation, The University of Pennsylvania.

Huang, L. (2008b). Forest reranking: Discriminative parsing with non-local features. In *ACL*, 586–594.

Huang, L., & Chiang, D. (2007). Forest rescoring: Faster decoding with integrated language models. In *ACL*.

Huang, L., & Sagae, K. (2010). Dynamic programming for linear-time incremental parsing. In *ACL*, 1077–1086.

Huang, Z., Harper, M., & Petrov, S. (2010). Self-training with products of latent variable grammars. In *EMNLP*, 12–22.

Huang, Z., & Harper, M. P. (2009). Self-training pcfg grammars with latent annotations across languages. In *EMNLP*, 832–841.

Jelinek, F., Lafferty, J., & Mercer, R. (1992). Basic methods of probabilistic context free grammars. In *Speech recognition and understanding*, volume 75, (pp. 345–360). Springer Berlin Heidelberg.

Jiang, J., & Zhai, C. (2007). Instance weighting for domain adaptation in nlp. In *ACL*, volume 2007, 22.

Johnson, M. (2001). Joint and conditional estimation of tagging and parsing models. In *ACL*, 322–329.

Johnson, M., & Ural, A. E. (2010). Reranking the Berkeley and Brown parsers. In *HLT-NAACL*, 665–668.

Juditsky, A., & Nesterov, Y. (2014). Primal-dual subgradient methods for minimizing uniformly convex functions. *arXiv*, 1401.1792v1.

Kasami, T. (1965). An efficient recognition and syntax analysis algorithm for context-free languages. Technical report, Air Force Cambridge Research Lab.

Keerthi, S. S., Sundararajan, S., Chang, K.-W., Hsieh, C.-J., & Lin, C.-J. (2008). A sequential dual method for large scale multi-class linear svms. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '08, 408–416. New York, NY, USA: ACM.

Klein, D., & Manning, C. D. (2001). Parsing and hypergraphs. In *IWPT*.

Koo, T., Carreras, X., & Collins, M. (2008). Simple semi-supervised dependency parsing. In *ACL*, 595–603.

Koo, T., & Collins, M. (2005). Hidden-variable models for discriminative reranking. In *HLT-EMNLP*, 507–514.

Koo, T., & Collins, M. (2010). Efficient third-order dependency parsers. In *ACL*, 1–11.

Koo, T., Globerson, A., Carreras, X., & Collins, M. (2007). Structured prediction models via the matrix-tree theorem. In *EMNLP-CoNLL*, 141–150.

Kübler, S., McDonald, R. T., & Nivre, J. (2009). *Dependency parsing*. Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers.

Lacoste-Julien, S., Schmidt, M., & Bach, F. (2012). A simpler approach to obtaining an o (1/t) convergence rate for projected stochastic subgradient descent. *arXiv*, 1212.2002.

Lafferty, J. D., McCallum, A., & Pereira, F. C. N. (2001). Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *ICML*, 282–289.

Langford, J., Smola, A. J., & Zinkevich, M. (2009). Slow learners are fast. *NIPS*, 22, 2331–2339.

Lapata, M., & Keller, F. (2004). The web as a baseline: Evaluating the performance of unsupervised web-based models for a range of nlp tasks. In *HLT-NAACL*, 121–128.

Lauer, M. (1995). Corpus statistics meet the noun compound: Some empirical results. In *ACL*, ACL '95, 47–54.

Levin, D. A., Peres, Y., & Wilmer, E. L. (2009). *Markov chains and mixing times*. American Mathematical Society.

Levinson, S. E., Rabiner, L. R., & Sondhi, M. M. (1983). An introduction to the application of the theory of probabilistic functions of a markov process to automatic speech recognition. *Bell System Technical Journal*, 62(4), 1035–1074.

Liu, D. C., & Nocedal, J. (1989). On the limited memory BFGS method for large scale optimization. *Math. Programming*, 45(3, (Ser. B)), 503–528.

Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., & Hellerstein, J. M. (2010). Graphlab: A new framework for parallel machine learning. *CoRR*, .

MacKay, D. J. (2003). *Information theory, inference and learning algorithms*. Cambridge university press.

Magerman, D. M. (1994). Natural language parsing as statistical pattern recognition. Doctoral Dissertation, Stanford University.

Mann, G., McDonald, R. T., Mohri, M., Silberman, N., & Walker, D. (2009). Efficient large-scale distributed training of conditional maximum entropy models. In *NIPS*.

Manning, C. D., & Schütze, H. (1999). *Foundations of statistical natural language processing*. MIT press.

Marcus, M., Kim, G., Marcinkiewicz, M. A., MacIntyre, R., Bies, A., Ferguson, M., Katz, K., & Schasberger, B. (1994). The penn treebank: annotating predicate argument structure. In *HLT*, HLT '94, 114–119.

Marcus, M. P., Santorini, B., & Marcinkiewicz, M. A. (1993). Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2), 313–330.

de Marneffe, M.-C., & Manning, C. D. (2008). The Stanford typed dependencies representation. In *Coling 2008: Proceedings of the workshop on Cross-Framework and Cross-Domain Parser Evaluation*, 1–8.

Martins, A. F., Gimpel, K., Smith, N. A., Xing, E. P., Figueiredo, M. A., & Aguiar, P. M. (2010). Learning structured classifiers with dual coordinate ascent. Technical report, DTIC Document.

McCallum, A., Freitag, D., & Pereira, F. C. (2000). Maximum entropy markov models for information extraction and segmentation. In *ICML*, 591–598.

McClosky, D., Charniak, E., & Johnson, M. (2006a). Effective self-training for parsing. In *HLT-NAACL*.

McClosky, D., Charniak, E., & Johnson, M. (2006b). Reranking and self-training for parser adaptation. In *ACL*, 337–344.

McDonald, R., & Pereira, F. (2006). Online learning of approximate dependency parsing algorithms. In *EACL*, 81–88.

McDonald, R. T., Crammer, K., & Pereira, F. C. N. (2005). Online large-margin training of dependency parsers. In *ACL*, 91–98.

McDonald, R. T., Hall, K., & Mann, G. (2010). Distributed training strategies for the structured perceptron. In *HLT-NAACL*, 456–464.

Meza-Ruiz, I., & Riedel, S. (2009). Jointly identifying predicates, arguments and senses using markov logic. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, NAACL '09, 155–163.

Minka, T. (2005). Discriminative models not discriminative training. Technical report.

Miyao, Y., & Tsujii, J. (2005). Probabilistic disambiguation models for wide-coverage hpsg parsing. In *ACL*.

Moore, G. E., et al. (1965). Cramming more components onto integrated circuits. *Electronics Magazine*, 38(8), .

Nakov, P., & Hearst, M. (2005). Search engine statistics beyond the n-gram: Application to noun compound bracketing. In *CoNLL*, 17–24.

Neal, R. M. (1993). Probabilistic inference using markov chain monte carlo methods, .

Nedić, A., Bertsekas, D., & Borkar, V. (2001). Distributed asynchronous incremental subgradient methods. *Studies in Computational Mathematics*, 8, 381–407.

Nedic, A., & Ozdaglar, A. (2009). Distributed subgradient methods for multi-agent optimization. *IEEE Transactions on Automatic Control*, 54(1), 48–61.

Nemirovski, A. S., & Yudin, D. B. (1983). *Problem complexity and method efficiency in optimization*. UK: John Wiley.

Nesterov, Y. (2005). Smooth minimization of non-smooth functions. *Mathematical Programming*, 103(1), 127–152.

Nesterov, Y. (2009). Primal-dual subgradient methods for convex problems. *Mathematical programming*, 120(1), 221–259.

Niu, F., Recht, B., Ré, C., & Wright, S. J. (2011). Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. *arXiv*, 1106.5730.

Nivre, J., Hall, J., Nilsson, J., Chanev, A., Eryigit, G., Kübler, S., Marinov, S., & Marsi, E. (2007). Maltparser: A language-independent system for data-driven dependency parsing. *Natural Language Engineering*, 13(2), 95–135.

Nivre, J., & Scholz, M. (2004). Deterministic dependency parsing of english text. In *Proceedings of the 20th International Conference on Computational Linguistics*, 64.

Novikoff, A. B. J. (1962). On convergence proofs for perceptrons. In *1962 Symposium on the Mathematical Theory of Automata*, 615–620.

Och, F. J. (2003). Minimum error rate training in statistical machine translation. In *ACL*, 160–167.

Petrov, S. (2009). Coarse-to-fine natural language processing. Doctoral Dissertation, University of California at Bekeley, Berkeley, CA, USA.

Petrov, S. (2010). Products of random latent variable grammars. In *HLT-NAACL*, 19–27.

Petrov, S., Barrett, L., Thibaux, R., & Klein, D. (2006). Learning accurate, compact, and interpretable tree annotation. In *ACL*, 433–440.

Petrov, S., Chang, P.-C., Ringgaard, M., & Alshawi, H. (2010). Uptraining for accurate deterministic question parsing. In *EMNLP*, 705–713.

Petrov, S., & Klein, D. (2007a). Discriminative log-linear grammars with latent variables. In *NIPS*.

Petrov, S., & Klein, D. (2007b). Improved inference for unlexicalized parsing. In *HLT-NAACL*, 404–411.

Petrov, S., & McDonald, R. (2012). Overview of the 2012 shared task on parsing the web. In *Notes of the First Workshop on Syntactic Analysis of Non-Canonical Language, NAACL*.

Pitler, E., Bergsma, S., Lin, D., & Church, K. (2010). Using web-scale n-grams to improve base np parsing performance. In *Proceedings of the 23rd International Conference on Computational Linguistics*, 886–894.

Platt, J. C. (1998). Fast training of support vector machines using sequential minimal optimization. In S. B., B. C., & S. A. (Eds.), *Advances in kernel methods–support vector learning*. MIT Press.

Polyak, B. T., & Juditsky, A. B. (1992). Acceleration of stochastic approximation by averaging. *SIAM Journal on Control and Optimization*, 30(4), .

Pomerleau, D. (1995). Neural network vision for robot driving. *The Handbook of Brain Theory and Neural Networks*, 161–181.

Quattoni, A., Collins, M., & Darrell, T. (2004). Conditional random fields for object recognition. In *NIPS*, 1097–1104.

Rabiner, L. (1989). A tutorial on hidden markov models and selected applications in speech recognition. *IEEE*, 77(2), 257–286.

Rakhlin, A., Shamir, O., & Sridharan, K. (2011). Making gradient descent optimal for strongly convex stochastic optimization. *arXiv*, 1109.5647.

Ratliff, N., Bagnell, J. A., & Zinkevich, M. (2006). Subgradient methods for maximum margin structured learning. In *ICML Workshop on Learning in Structured Output Spaces*, volume 46.

Riezler, S., King, T. H., Kaplan, R. M., Crouch, R. S., III, J. T. M., & Johnson, M. (2002). Parsing the wall street journal using a lexical-functional grammar and discriminative estimation techniques. In *ACL*, 271–278.

Rifkin, R., & Klautau, A. (2004). In defense of one-vs-all classification. *JMLR*, 5, 101–141.

Roark, B., & Bacchiani, M. (2003). Supervised and unsupervised pcfg adaptation to novel domains. In *NAACL*, 126–133.

Robbins, H., & Monro, S. (1951). A stochastic approximation method. *The Annals of Mathematical Statistics*, 400–407.

Rockafellar, R. T. (1970). *Convex analysis*. Princeton University Press, Princeton, New Jersey.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6), 386–408.

Sagae, K. (2010). Self-training without reranking for parser domain adaptation and its impact on semantic role labeling. In *Proceedings of the 2010 Workshop on Domain Adaptation for Natural Language Processing*, 37–44.

Sagae, K., & Lavie, A. (2005). A classifier-based parser with linear run-time complexity. In *IWPT*, 125–132.

Sagae, K., & Lavie, A. (2006). Parser combination by reparsing. In *NAACL*, 129–132.

Sarkar, A. (2000). Statistical parsing algorithms for lexicalized tree adjoining grammars. Technical report, University of Pennsylvania.

Sarkar, A. (2001). Applying co-training methods to statistical parsing. In *NAACL*.

Seddah, D., Sagot, B., Candito, M., et al. (2012). The alpage architecture at the SANCL 2012 shared task: Robust pre-processing and lexical bridging for user-generated content parsing. In *Notes of the First Workshop on Syntactic Analysis of Non-Canonical Language, NAACL*.

Shalev-Shwartz, S., Singer, Y., & Srebro, N. (2007). Pegasos: Primal estimated sub-gradient solver for svm. In *ICML*, 807–814.

Shalev-Shwartz, S., Singer, Y., Srebro, N., & Cotter, A. (2011). Pegasos: primal estimated sub-gradient solver for svm. *Mathematical Programming*, 127(1), 3–30.

Shalev-Shwartz, S., & Srebro, N. (2008). Svm optimization: inverse dependence on training set size. In *ICML*, 928–935. ACM.

Shamir, O. (2011). Making gradient descent optimal for strongly convex stochastic optimization. *OPT 2011*, .

Simianer, P., Riezler, S., & Dyer, C. (2012). Joint feature selection in distributed stochastic learning for large-scale discriminative training in smt. In *ACL*, 11–21.

Smith, A., Cohn, T., & Osborne, M. (2005). Logarithmic opinion pools for conditional random fields. In *ACL*.

Smola, A. J., Viswanathan, S. V. N., & Le, Q. V. (2007). Bundle methods for machine learning. In *NIPS*, 1377–1384.

Søgaard, A., & Rishøj, C. (2010). Semi-supervised dependency parsing using generalized tri-training. In *COLING*, 1065–1073.

Spitkovsky, V. I., Alshawi, H., Jurafsky, D., & Manning, C. D. (2010). Viterbi training improves unsupervised dependency parsing. In *CoNLL*, CoNLL '10, 9–17.

Steedman, M., Hwa, R., Clark, S., Osborne, M., Sarkar, A., Hockenmaier, J., Ruhlen, P., Baker, S., & Crim, J. (2003a). Example selection for bootstrapping statistical parsers. In *HLT-NAACL*.

Steedman, M., Osborne, M., Sarkar, A., Clar, S., Hwa, R., Hockenmaier, J., Ruhlen, P., Baker, S., & Crim, J. (2003b). Bootstrapping statistical parsers from small datasets. In *EACL*.

Sutter, H. (2005). The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), 202–210.

Sutton, C., & McCallum, A. (2012). An introduction to conditional random fields. *Foundations and Trends in Machine Learning*, 4(4), 267373.

Sutton, C., Sindelar, M., & McCallum, A. (2005). Feature bagging: Preventing weight undertraining in structured discriminative learning. In *HLT-NAACL*.

Suzuki, J., Isozaki, H., Carreras, X., & Collins, M. (2009). An empirical study of semi-supervised structured conditional models for dependency parsing. In *EMNLP*, 551–560.

Takác, M., Bijral, A., Richtárik, P., & Srebro, N. (2013). Mini-batch primal and dual methods for svms. In *ICML*.

Taskar, B., Klein, D., Collins, M., Koller, D., & Manning, C. D. (2004). Max-margin parsing. In *EMNLP*, 1–8.

Tesnière, L., & Fourquet, J. (1959). *Eléments de syntaxe structurale*, volume 1965. Paris: Klincksieck.

Tjong Kim Sang, E. F., & De Meulder, F. (2003). Introduction to the conll-2003 shared task: Language-independent named entity recognition. In *CoNLL*, 142–147.

Tsianos, K. I., & Rabbat, M. G. (2012). Distributed strongly convex optimization. In *Communication, Control, and Computing (Allerton), 2012 50th Annual Allerton Conference on*, 593–600. IEEE.

Tsochantaridis, I., Joachims, T., Hofmann, T., & Altun, Y. (2005). Large margin methods for structured and interdependent output variables. In *JMLR*, 1453–1484.

Vadas, D., & Curran, J. R. (2007). Large-scale supervised models for noun phrase bracketing. In *Proceedings of the 10th Conference of the Pacific Association for Computational Linguistics*, 104–112.

Valiant, L. G. (1984). A theory of the learnable. *Communications of the ACM*, 27(11), 1134–1142.

Vapnik, V. (1998). *Statistical learning theory*. New York: Wiley.

Vapnik, V. (2000). *The nature of statistical learning theory*. New York: Springer.

Viterbi, A. (1967). Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *Information Theory, IEEE Transactions on*, 13(2), 260–269.

Wang, M., Sagae, K., & Mitamura, T. (2006). A fast, accurate deterministic parser for chinese. In *Proceedings of the 21st International Conference on Computational Linguistics*, 425–432.

Weinberger, K., Dasgupta, A., Langford, J., Smola, A., & Attenberg, J. (2009). Feature hashing for large scale multitask learning. In *ICML*, 1113–1120. ACM.

Weston, J., & Watkins, C. (1998). Multi-class support vector machines. Technical report, Royal Holloway, University of London.

White, T. (2009). *Hadoop: The definitive guide: The definitive guide*. O'Reilly Media.

Xiao, L. (2010). Dual averaging methods for regularized stochastic learning and online optimization. *JMLR*, 11, 2543–2596.

Yamada, H., & Matsumoto, Y. (2003). Statistical dependency analysis with support vector machines. In *IWPT*, 195–206.

Younger, D. H. (1967). Recognition and parsing of context-free languages in time n^3. *Information and control*, 10(2), 189–208.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., & Stoica, I. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, 2–2. USENIX Association.

Zeman, D., & Žabokrtskỳ, Z. (2005). Improving parsing accuracy by combining diverse dependency parsers. In *IWPT*, 171–178.

Zhang, H., & McDonald, R. (2012). Generalized higher-order dependency parsing with cube pruning. In *EMNLP*, 238–242.

Zhang, H., Zhang, M., Tan, C. L., & Li, H. (2009). K-best combination of syntactic parsers. In *EMNLP*, 1552–1560.

Zhang, Y., Duchi, J. C., & Wainwright, M. J. (2012). Communication-efficient algorithms for statistical optimization. In *Decision and Control*, 6792–6792.

Zhang, Y., & Clark, S. (2009). Transition-based parsing of the chinese treebank using a global discriminative model. In *IWPT*, 162–171.

Zhang, Y., & Clark, S. (2011). Shift-reduce CCG parsing. In *ACL*, 683–692.

Zhang, Y., & Nivre, J. (2011). Transition-based dependency parsing with rich non-local features. In *ACL*, 188–293.

Zhao, K., & Huang, L. (2013). Minibatch and parallelization for online large margin structured learning. In *HLT-NAACL*, 370–379.

Zhu, M., Zhang, Y., Chen, W., Zhang, M., & Zhu, J. (2013). Fast and accurate shift-reduce constituent parsing. In *ACL (1)*, 434–443.

Zinkevich, M. (2003). Online convex programming and generalized infinitesimal gradient ascent. In *ICML*.

Zinkevich, M., Weimer, M., Li, L., & Smola, A. J. (2010). Parallelized stochastic gradient descent. In *NIPS*, 2595–2603.