Minibatch and Parallelization for Online Large Margin Structured Learning

Kai Zhao[1]
[1]Computer Science Program, Graduate Center
City University of New York
kzhao@gc.cuny.edu

Liang Huang[2,1]
[2]Computer Science Dept, Queens College
City University of New York
huang@cs.qc.cuny.edu

Abstract

Online learning algorithms such as perceptron and MIRA have become popular for many NLP tasks thanks to their simpler architecture and faster convergence over batch learning methods. However, while batch learning such as CRF is easily parallelizable, online learning is much harder to parallelize: previous efforts often witness a decrease in the converged accuracy, and the speedup is typically very small ($\sim$3) even with many (10+) processors. We instead present a much simpler architecture based on "mini-batches", which is trivially parallelizable. We show that, unlike previous methods, minibatch learning (in serial mode) actually *improves* the converged accuracy for both perceptron and MIRA learning, and when combined with simple parallelization, minibatch leads to very significant speedups (up to 9x on 12 processors) on state-of-the-art parsing and tagging systems.

1 Introduction

Online structured learning algorithms such as the structured perceptron (Collins, 2002) and k-best MIRA (McDonald et al., 2005) have become more and more popular for many NLP tasks such as dependency parsing and part-of-speech tagging. This is because, compared to their batch learning counterparts, online learning methods offer faster convergence rates and better scalability to large datasets, while using much less memory and a much simpler architecture which only needs 1-best or k-best decoding. However, online learning for NLP typically involves expensive inference on each example for 10 or more passes over millions of examples, which often makes training too slow in practice; for example systems such as the popular (2nd-order) MST parser (McDonald and Pereira, 2006) usually require the order of days to train on the Treebank on a commodity machine (McDonald et al., 2010).

There are mainly two ways to address this scalability problem. On one hand, researchers have been developing modified learning algorithms that allow inexact search (Collins and Roark, 2004; Huang et al., 2012). However, the learner still needs to loop over the whole training data (on the order of millions of sentences) many times. For example the best-performing method in Huang et al. (2012) still requires 5-6 hours to train a very fast parser.

On the other hand, with the increasing popularity of multicore and cluster computers, there is a growing interest in speeding up training via parallelization. While batch learning such as CRF (Lafferty et al., 2001) is often trivially parallelizable (Chu et al., 2007) since each update is a batch-aggregate of the update from each (independent) example, online learning is much harder to parallelize due to the dependency between examples, i.e., the update on the first example should in principle influence the decoding of all remaining examples. Thus if we decode and update the first and the 1000th examples in parallel, we lose their interactions which is one of the reasons for online learners' fast convergence. This explains why previous work such as the iterative parameter mixing (IPM) method of McDonald et al. (2010) witnesses a decrease in the accuracies of parallelly-learned models, and the speedup is typically very small (about 3 in their experiments) even with 10+ processors.

We instead explore the idea of "minibatch" for online large-margin structured learning such as perceptron and MIRA. We argue that minibatch is advantageous in *both* serial and parallel settings.

First, for minibatch perceptron in the serial set-

Proceedings of NAACL-HLT 2013, pages 370–379,
Atlanta, Georgia, 9–14 June 2013.

ting, our intuition is that, although decoding is done independently within one minibatch, updates are done by averaging update vectors in batch, providing a "mixing effect" similar to "averaged parameters" of Collins (2002) which is also found in IPM (McDonald et al., 2010), and online EM (Liang and Klein, 2009).

Secondly, minibatch MIRA in the serial setting has an advantage that, different from previous methods such as SGD which simply sum up the updates from all examples in a minibatch, a minibatch MIRA update tries to simultaneously satisfy an aggregated set of constraints that are collected from multiple examples in the minibatch. Thus each minibatch MIRA update involves an optimization over many more constraints than in pure online MIRA, which could potentially lead to a better margin. In other words we can view MIRA as an online version or stepwise approximation of SVM, and minibatch MIRA can be seen as a better approximation as well as a middleground between pure MIRA and SVM.[1]

More interestingly, the minibatch architecture is trivially parallelizable since the examples within each minibatch could be decoded in parallel on multiple processors (while the update is still done in serial). This is known as "synchronous minibatch" and has been explored by many researchers (Gimpel et al., 2010; Finkel et al., 2008), but all previous works focus on probabilistic models along with SGD or EM learning methods while our work is the first effort on large-margin methods.

We make the following contributions:

- Theoretically, we present a serial minibatch framework (Section 3) for online large-margin learning and prove the convergence theorems for minibatch perceptron and minibatch MIRA.
- Empirically, we show that serial minibatch could speed up convergence and improve the converged accuracy for both MIRA and perceptron on state-of-the-art dependency parsing and part-of-speech tagging systems.
- In addition, when combined with simple (synchronous) parallelization, minibatch MIRA leads to very significant speedups (up to 9x on 12 processors) that are much higher than that of IPM (McDonald et al., 2010) on state-of-the-art parsing and tagging systems.

[1] This is similar to Pegasos (Shalev-Shwartz et al., 2007) that applies subgradient descent over a minibatch. Pegasos becomes pure online when the minibatch size is 1.

Algorithm 1 Generic Online Learning.

Input: data $D = \{(x^{(t)}, y^{(t)})\}_{t=1}^{n}$ and feature map $\mathbf{\Phi}$
Output: weight vector $\mathbf{w}$

repeat
  for each example (x, y) **in** D **do**
    $C \leftarrow$ FINDCONSTRAINTS$(x, y, \mathbf{w})$ ▷ decoding
    if $C \neq \emptyset$ **then** UPDATE$(\mathbf{w}, C)$
until converged

2 Online Learning: Perceptron and MIRA

We first present a unified framework for online large-margin learning, where perceptron and MIRA are two special cases. Shown in Algorithm 1, the online learner considers each input example (x, y) sequentially and performs two steps:

1. find the set C of violating constraints, and
2. update the weight vector $\mathbf{w}$ according to C.

Here a triple $\langle x, y, z \rangle$ is said to be a "violating constraint" with respect to model $\mathbf{w}$ if the incorrect label z scores higher than (or equal to) the correct label y in $\mathbf{w}$, i.e., $\mathbf{w} \cdot \Delta\mathbf{\Phi}(\langle x, y, z \rangle) \leq 0$, where $\Delta\mathbf{\Phi}(\langle x, y, z \rangle)$ is a short-hand notation for the update vector $\mathbf{\Phi}(x, y) - \mathbf{\Phi}(x, z)$ and $\mathbf{\Phi}$ is the feature map (see Huang et al. (2012) for details). The subroutines FINDCONSTRAINTS and UPDATE are analogous to "APIs", to be specified by specific instances of this online learning framework. For example, the structured perceptron algorithm of Collins (2002) is implemented in Algorithm 2 where FINDCONSTRAINTS returns a singleton constraint if the 1-best decoding result z (the highest scoring label according to the current model) is different from the true label y. Note that in the UPDATE function, C is always a singleton constraint for the perceptron, but we make it more general (as a set) to handle the batch update in the minibatch version in Section 3.

On the other hand, Algorith 3 presents the k-best MIRA Algorithm of McDonald et al. (2005) which generalizes multiclass MIRA (Crammer and Singer, 2003) for structured prediction. The decoder now

Algorithm 2 Perceptron (Collins, 2002).

function FINDCONSTRAINTS($x, y, \mathbf{w}$)
  $z \leftarrow \mathbf{argmax}_{s \in \mathcal{Y}(x)} \mathbf{w} \cdot \mathbf{\Phi}(x, s)$ ▷ decoding
  if $z \neq y$ **then return** $\{\langle x, y, z \rangle\}$
  else return $\emptyset$
procedure UPDATE($\mathbf{w}, C$)
  $\mathbf{w} \leftarrow \mathbf{w} + \frac{1}{|C|} \sum_{c \in C} \Delta\mathbf{\Phi}(c)$ ▷ (batch) update

Algorithm 3 k-best MIRA (McDonald et al., 2005).

function FINDCONSTRAINTS($x, y, \mathbf{w}$)
  $Z \leftarrow k\text{-best}_{z \in \mathcal{Y}(x)} \mathbf{w} \cdot \mathbf{\Phi}(x, z)$
  $Z \leftarrow \{z \in Z \mid z \neq y, \mathbf{w} \cdot \Delta\mathbf{\Phi}(\langle x, y, z \rangle) \leq 0\}$
  return $\{(\langle x, y, z \rangle, \ell(y, z)) \mid z \in Z\}$
procedure UPDATE($\mathbf{w}, C$)
  $\mathbf{w} \leftarrow \mathop{\mathbf{argmin}}_{\mathbf{w}': \forall (c,\ell) \in C,\ \mathbf{w}' \cdot \Delta\mathbf{\Phi}(c) \geq \ell} \|\mathbf{w}' - \mathbf{w}\|^2$

finds the k-best solutions Z first, and returns a set of violating constraints in Z, The update in MIRA is more interesting: it searches for the new model $\mathbf{w}'$ with minimum change from the current model $\mathbf{w}$ so that $\mathbf{w}'$ corrects each violating constraint by a margin at least as large as the loss $\ell(y, z)$ of the incorrect label z.

Although not mentioned in the pseudocode, we also employ "averaged parameters" (Collins, 2002) for both perceptron and MIRA in all experiments.

3 Serial Minibatch

The idea of serial minibatch learning is extremely simple: divide the data into $\lceil n/m \rceil$ minibatches of size m, and do batch updates after decoding each minibatch (see Algorithm 4). The FINDCONSTRAINTS and UPDATE subroutines remain unchanged for both perceptron and MIRA, although it is important to note that a perceptron batch update uses the average of update vectors, not the sum, which simplifies the proof. This architecture is often called "synchronous minibatch" in the literature (Gimpel et al., 2010; Liang and Klein, 2009; Finkel et al., 2008). It could be viewed as a middleground between pure online learning and batch learning.

3.1 Convergence of Minibatch Perceptron

We denote $C(D)$ to be the set of all possible violating constraints in data D (cf. Huang et al. (2012)):

$$C(D) = \{\langle x, y, z \rangle \mid (x, y) \in D, z \in \mathcal{Y}(x) - \{y\}\}.$$

Algorithm 4 Serial Minibatch Online Learning.

Input: data D, feature map $\mathbf{\Phi}$, and minibatch size m
Output: weight vector $\mathbf{w}$
Split D into $\lceil n/m \rceil$ minibatches $D_1 \ldots D_{\lceil n/m \rceil}$
repeat
  for $i \leftarrow 1 \ldots \lceil n/m \rceil$ **do** ▷ for each minibatch
    $C \leftarrow \cup_{(x,y) \in D_i}$ FINDCONSTRAINTS$(x, y, \mathbf{w})$
    if $C \neq \emptyset$ **then** UPDATE$(\mathbf{w}, C)$ ▷ batch update
until converged

A training set D is **separable** by feature map $\mathbf{\Phi}$ with **margin** $\delta > 0$ if there exists a unit oracle vector $\mathbf{u}$ with $\|\mathbf{u}\| = 1$ such that $\mathbf{u} \cdot \Delta\mathbf{\Phi}(\langle x, y, z \rangle) \geq \delta$, for all $\langle x, y, z \rangle \in C(D)$. Furthermore, let radius $R \geq \|\Delta\mathbf{\Phi}(\langle x, y, z \rangle)\|$ for all $\langle x, y, z \rangle \in C(D)$.

Theorem 1. *For a separable dataset D with margin δ and radius R, the minibatch perceptron algorithm (Algorithms 4 and 2) will terminate after t minibatch updates where $t \leq R^2/\delta^2$.*

Proof. Let $\mathbf{w}^t$ be the weight vector **before** the t^{th} update; $\mathbf{w}^0 = \mathbf{0}$. Suppose the t^{th} update happens on the constraint set $C_t = \{c_1, c_2, \ldots, c_a\}$ where $a = |C_t|$, and each $c_i = \langle x_i, y_i, z_i \rangle$. We convert them to the set of update vectors $\mathbf{v}_i = \Delta\mathbf{\Phi}(c_i) = \Delta\mathbf{\Phi}(\langle x_i, y_i, z_i \rangle)$ for all i. We know that:

1. $\mathbf{u} \cdot \mathbf{v}_i \geq \delta$ (margin on unit oracle vector)
2. $\mathbf{w}^t \cdot \mathbf{v}_i \leq 0$ (violation: z_i dominates y_i)
3. $\|\mathbf{v}_i\|^2 \leq R^2$ (radius)

Now the update looks like

$$\mathbf{w}^{t+1} = \mathbf{w}^t + \frac{1}{|C_t|} \sum_{c \in C_t} \Delta\mathbf{\Phi}(c) = \mathbf{w}^t + \frac{1}{a} \textstyle\sum_i \mathbf{v}_i. \quad (1)$$

We will bound $\|\mathbf{w}^{t+1}\|$ from two directions:

1. Dot product both sides of the update equation (1) with the unit oracle vector $\mathbf{u}$, we have

$$\begin{aligned} \mathbf{u} \cdot \mathbf{w}^{t+1} &= \mathbf{u} \cdot \mathbf{w}^t + \frac{1}{a} \textstyle\sum_i \mathbf{u} \cdot \mathbf{v}_i \\ &\geq \mathbf{u} \cdot \mathbf{w}^t + \frac{1}{a} \textstyle\sum_i \delta && \text{(margin)} \\ &= \mathbf{u} \cdot \mathbf{w}^t + \delta && (\textstyle\sum_i = a) \\ &\geq t\delta && \text{(by induction)} \end{aligned}$$

Since for any two vectors $\mathbf{a}$ and $\mathbf{b}$ we have $\|\mathbf{a}\|\|\mathbf{b}\| \geq \mathbf{a}\cdot\mathbf{b}$, thus $\|\mathbf{u}\|\|\mathbf{w}^{t+1}\| \geq \mathbf{u}\cdot\mathbf{w}^{t+1} \geq t\delta$. As $\mathbf{u}$ is a unit vector, we have $\|\mathbf{w}^{t+1}\| \geq t\delta$.

2. On the other hand, take the norm of both sides of Eq. (1):

$$\begin{aligned}
&\|\mathbf{w}^{t+1}\|^2 = \|\mathbf{w}^t + \frac{1}{a}\textstyle\sum_i \mathbf{v}_i\|^2 \\
=&\|\mathbf{w}^t\|^2 + \|\textstyle\sum_i \frac{1}{a}\mathbf{v}_i\|^2 + \frac{2}{a}\,\mathbf{w}^t \cdot \textstyle\sum_i \mathbf{v}_i \\
\leq&\|\mathbf{w}^t\|^2 + \|\textstyle\sum_i \frac{1}{a}\mathbf{v}_i\|^2 + 0 && \text{(violation)} \\
\leq&\|\mathbf{w}^t\|^2 + \textstyle\sum_i \frac{1}{a}\|\mathbf{v}_i\|^2 && \text{(Jensen's)} \\
\leq&\|\mathbf{w}^t\|^2 + \textstyle\sum_i \frac{1}{a}R^2 && \text{(radius)} \\
=&\|\mathbf{w}^t\|^2 + R^2 && (\textstyle\sum_i = a) \\
\leq&tR^2 && \text{(by induction)}
\end{aligned}$$

Combining the two bounds, we have

$$t^2\delta^2 \leq \|\mathbf{w}^{t+1}\|^2 \leq tR^2$$

thus the number of minibatch updates $t \leq R^2/\delta^2$. □

Note that this bound is identical to that of pure online perceptron (Collins, 2002, Theorem 1) and is irrelevant to minibatch size m. The use of Jensen's inequality is inspired by McDonald et al. (2010).

3.2 Convergence of Minibatch MIRA

We also give a proof of convergence for MIRA with relaxation.[2] We present the optimization problem in the UPDATE function of Algorithm 3 as a quadratic program (QP) with slack variable ξ:

$$\begin{aligned}
\mathbf{w}^{t+1} \leftarrow &\operatorname*{\mathbf{argmin}}_{\mathbf{w}^{t+1}} \|\mathbf{w}^{t+1} - \mathbf{w}^t\|^2 + \xi \\
&\text{s.t. } \mathbf{w}^{t+1}\cdot\mathbf{v}_i \geq \ell_i - \xi, \text{ for all} (c_i, \ell_i) \in C_t
\end{aligned}$$

where $\mathbf{v}_i = \Delta\mathbf{\Phi}(c_i)$ is the update vector for constraint c_i. Consider the Lagrangian:

$$\begin{aligned}
\mathcal{L} =&\|\mathbf{w}^{t+1} - \mathbf{w}^t\|^2 + \xi + \sum_{i=1}^{|C_t|}\eta_i(\ell_i - \mathbf{w}'\cdot\mathbf{v}_i - \xi) \\
&\eta_i \geq 0, \text{ for } 1 \leq i \leq |C_t|.
\end{aligned}$$

[2] Actually this relaxation is *not* necessary for the convergence proof. We employ it here solely to make the proof shorter. It is *not* used in the experiments either.

Set the partial derivatives to 0 with respect to $\mathbf{w}'$ and ξ we have:

$$\mathbf{w}' = \mathbf{w} + \textstyle\sum_i \eta_i\mathbf{v}_i \quad (2)$$

$$\textstyle\sum_i \eta_i = 1 \quad (3)$$

This result suggests that the weight change can always be represnted by a linear combination of the update vectors (i.e. normal vectors of the constraint hyperplanes), with the linear coefficencies sum to 1.

Theorem 2 (convergence of minibatch MIRA)**.** *For a separable dataset D with margin δ and radius R, the minibatch MIRA algorithm (Algorithm 4 and 3) will make t updates where $t \leq R^2/\delta^2$.*

Proof. 1. Dot product both sides of Equation 2 with unit oracle vector $\mathbf{u}$:

$$\begin{aligned}
&\mathbf{u}\cdot\mathbf{w}^{t+1} = \mathbf{u}\cdot\mathbf{w}^t + \textstyle\sum_i \eta_i\mathbf{u}\cdot\mathbf{v}_i \\
\geq&\mathbf{u}\cdot\mathbf{w}^t + \textstyle\sum_i \eta_i\delta && \text{(margin)} \\
=&\mathbf{u}\cdot\mathbf{w}^t + \delta && \text{(Eq. 3)} \\
=&t\delta && \text{(by induction)}
\end{aligned}$$

2. On the other hand

$$\begin{aligned}
&\|\mathbf{w}^{t+1}\|^2 = \|\mathbf{w}^t + \textstyle\sum_i \eta_i\mathbf{v}_i\|^2 \\
=&\|\mathbf{w}^t\|^2 + \|\textstyle\sum_i \eta_i\mathbf{v}_i\|^2 + 2\,\mathbf{w}^t\cdot\textstyle\sum_i \eta_i\mathbf{v}_i \\
\leq&\|\mathbf{w}^t\|^2 + \|\textstyle\sum_i \eta_i\mathbf{v}_i\|^2 + 0 && \text{(violation)} \\
\leq&\|\mathbf{w}^t\|^2 + \textstyle\sum_i \eta_i\mathbf{v}_i^2 && \text{(Jensen's)} \\
\leq&\|\mathbf{w}^t\|^2 + \textstyle\sum_i \eta_i R^2 && \text{(radius)} \\
=&\|\mathbf{w}^t\|^2 + R^2 && \text{(Eq. 3)} \\
\leq&tR^2 && \text{(by induction)}
\end{aligned}$$

From the two bounds we have:

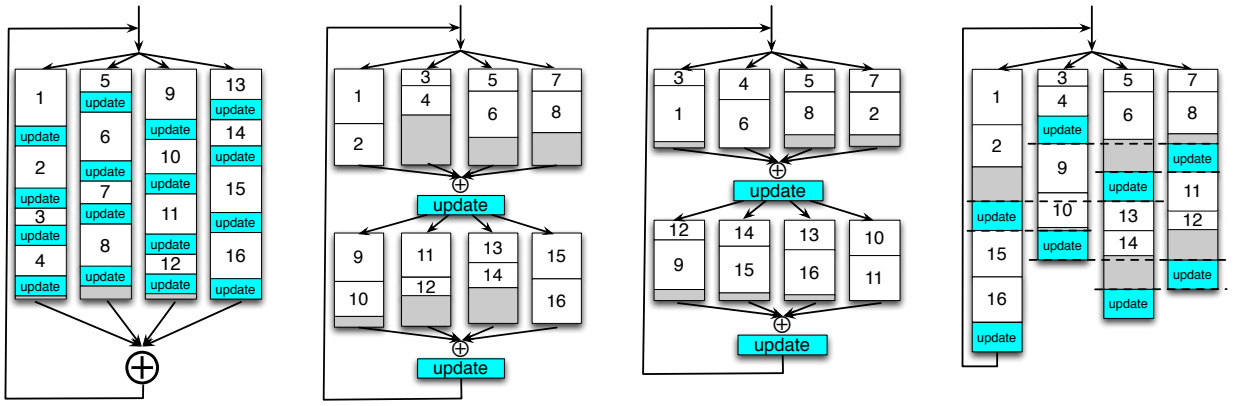
$$t^2\delta^2 \leq \|\mathbf{w}^{t+1}\|^2 \leq tR^2$$

thus within at most $t \leq R^2/\delta^2$ minibatch updates MIRA will converge.

□

4 Parallelized Minibatch

The key insight into parallelization is that the calculation of constraints (i.e. decoding) for each example within a minibatch is completely independent of



(a) IPM (b) unbalanced (c) balanced (d) asynchronous

Figure 1: Comparison of various methods for parallelizing online learning (number of processors $p = 4$). (a) iterative parameter mixing (McDonald et al., 2010). (b) unbalanced minibatch parallelization (minibatch size $m = 8$). (c) minibatch parallelization after load-balancing (within each minibatch). (d) asynchronous minibatch parallelization (Gimpel et al., 2010) (not implemented here). Each numbered box denotes the decoding of one example, and $\oplus$ denotes an aggregate operation, i.e., the merging of constraints after each minibatch or the mixing of weights after each iteration in IPM. Each gray shaded box denotes time wasted due to synchronization in (a)-(c) or blocking in (d). Note that in (d) at most one update can happen concurrently, making it substantially harder to implement than (a)-(c).

Algorithm 5 Parallized Minibatch Online Learning.

Input: D, $\mathbf{\Phi}$, minibatch size m, and # of processors p
Output: weight vector $\mathbf{w}$
Split D into $\lceil n/m \rceil$ minibatches $D_1 \dots D_{\lceil n/m \rceil}$
Split each D_i into m/p groups $D_{i,1} \dots D_{i,m/p}$
repeat
  for $i \leftarrow 1 \dots \lceil n/m \rceil$ **do** ▷ for each minibatch
    for $j \leftarrow 1 \dots m/p$ **in parallel do**
      $C_j \leftarrow \cup_{(x,y) \in D_{i,j}} \textsc{FindConstraints}(x, y, \mathbf{w})$
    $C \leftarrow \cup_j C_j$ ▷ in serial
    if $C \neq \emptyset$ **then** $\textsc{Update}(\mathbf{w}, C)$ ▷ in serial
until converged

other examples in the same batch. Thus we can easily distribute decoding for different examples in the same minibatch to different processors.

Shown in Algorithm 5, for each minibatch D_i, we split D_i into groups of equal size, and assign each group to a processor to decode. After all processors finish, we collect all constraints and do an update based on the union of all constraints. Figure 1 (b) illustrates minibatch parallelization, with comparison to iterative parameter mixing (IPM) of McDonald et al. (2010) (see Figure 1 (a)).

This synchronous parallelization framework should provide significant speedups over the serial mode. However, in each minibatch, inevitably, some processors will end up waiting for others to finish, especially when the lengths of sentences vary substantially (see the shaded area in Figure 1 (b)).

To alleviate this problem, we propose "**per-minibatch load-balancing**", which rearranges the sentences within each minibatch based on their lengths (which correlate with their decoding times) so that the total workload on each processor is balanced (Figure 1c). It is important to note that this shuffling does **not** affect learning at all thanks to the independence of each example within a minibatch. Basically, we put the shortest and longest sentences into the first thread, the second shortest and second longest into the second thread, etc. Although this is not necessary optimal scheduling, it works well in practice. As long as decoding time is linear in the length of sentence (as in incremental parsing or tagging), we expect a much smaller variance in processing time on each processor in one minibatch, which is confirmed in the experiments (see Figure 8).[3]

[3] In IPM, however, the waiting time is negligible, since the workload on each processor is almost balanced, analogous to a huge minibatch (Fig. 1a). Furthermore, shuffling *does* affect learning here since each thread in IPM is a pure online learner. So our IPM implementation does *not* use load-balancing.

5 Experiments

We conduct experiments on two typical structured prediction problems: incremental dependency parsing and part-of-speech tagging; both are done on state-of-the-art baseline. We also compare our parallelized minibatch algorithm with the iterative parameter mixing (IPM) method of McDonald et al. (2010). We perform our experiments on a commodity 64-bit Dell Precision T7600 workstation with two 3.1GHz 8-core CPUs (16 processors in total) and 64GB RAM. We use Python 2.7's `multiprocessing` module in all experiments.[4]

5.1 Dependency Parsing with MIRA

We base our experiments on our dynamic programming incremental dependency parser (Huang and Sagae, 2010).[5] Following Huang et al. (2012), we use max-violation update and beam size $b = 8$. We evaluate on the standard Penn Treebank (PTB) using the standard split: Sections 02-21 for training, and Section 22 as the held-out set (which is indeed the test-set in this setting, following McDonald et al. (2010) and Gimpel et al. (2010)). We then extend it to employ 1-best MIRA learning. As stated in Section 2, MIRA separates the gold label y from the incorrect label z with a margin at least as large as the loss $\ell(y, z)$. Here in incremental dependency parsing we define the loss function between a gold tree y and an incorrect *partial* tree z as the number of incorrect edges in z, plus the number of correct edges in y which are already ruled out by z. This MIRA extension results in slightly higher accuracy of 92.36, which we will use as the pure online learning baseline in the comparisons below.

5.1.1 Serial Minibatch

We first run minibatch in the serial mode with varying minibatch size of 4, 16, 24, 32, and 48 (see Figure 2). We can make the following observations. First, except for the largest minibatch size of 48, minibatch learning generally improves the accuracy

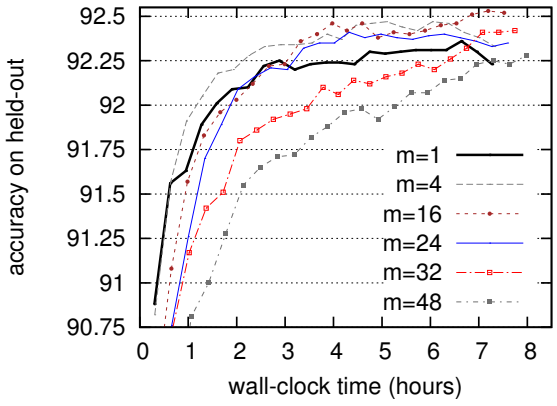


Figure 2: Minibatch with various minibatch sizes ($m = 4, 16, 24, 32, 48$) for parsing with MIRA, compared to pure MIRA ($m = 1$). All curves are on a single CPU.

of the converged model, which is explained by our intuition that optimization with a larger constraint set could improve the margin. In particular, $m = 16$ achieves the highest accuracy of 92.53, which is a 0.27 improvement over the baseline.

Secondly, minibatch learning can reach high levels of accuracy faster than the baseline can. For example, minibatch of size 4 can reach 92.35 in 3.5 hours, and minibatch of size 24 in 3.7 hours, while the pure online baseline needs 6.9 hours. In other words, just minibatch alone in serial mode can already speed up learning. This is also explained by the intuition of better optimization above, and contributes significantly to the final speedup of parallelized minibatch.

Lastly, larger minibatch sizes slow down the convergence, with $m = 4$ converging the fastest and $m = 48$ the slowest. This can be explained by the trade-off between the relative strengths from online learning and batch update: with larger batch sizes, we lose the dependencies between examples within the same minibatch.

Although larger minibatches slow down convergence, they actually offer better potential for parallelization since the number of processors p has to be smaller than minibatch size m (in fact, p should divide m). For example, $m = 24$ can work with 2, 3, 4, 6, 8, or 12 processors while $m = 4$ can only work with 2 or 4 and the speed up of 12 processors could easily make up for the slightly slower convergence

[4] We **turn off garbage-collection** in worker processes otherwise their running times will be highly unbalanced. We also admit that Python is not the best choice for parallelization, e.g., asychronous minibatch (Gimpel et al., 2010) requires "shared memory" not found in the current Python (see also Sec. 6).

[5] Available at `http://acl.cs.qc.edu/`. The version with minibatch parallelization will be available there soon.

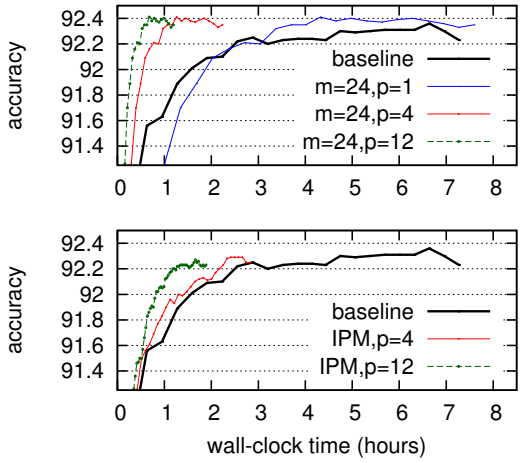


Figure 3: Parallelized minibatch is much faster than iterative parameter mixing. Top: minibatch of size 24 using 4 and 12 processors offers significant speedups over the serial minibatch and pure online baselines. Bottom: IPM with the same processors offers very small speedups.

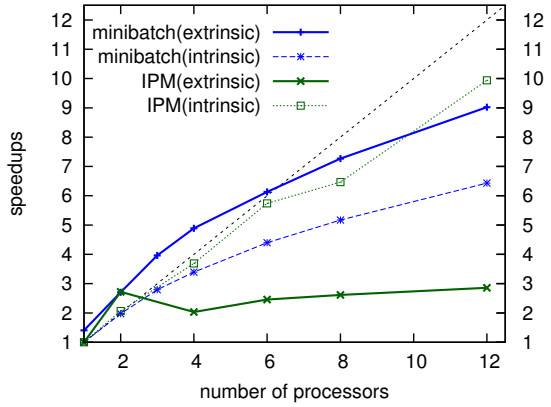


Figure 4: Speedups of minibatch parallelization vs. IPM on 1 to 12 processors (parsing with MIRA). Extrinsic comparisons use "the time to reach an accuracy of 92.27" for speed calculations, 92.27 being the converged accuracy of IPM using 12 processors. Intrinsic comparisons use average time per iteration regardless of accuracy.

rate. So there seems to be a "sweetspot" of minibatch sizes, similar to the tipping point observed in McDonald et al. (2010) when adding more processors starts to hurt convergence.

5.1.2 Parallelized Minibatch vs. IPM

In the following experiments we use minibatch size of $m = 24$ and run it in parallel mode on various numbers of processors ($p = 2 \sim 12$). Figure 3 (top) shows that 4 and 12 processors lead to very significant speedups over the serial minibatch and pure online baselines. For example, it takes the 12 processors only 0.66 hours to reach an accuracy of 92.35, which takes the pure online MIRA 6.9 hours, amounting to an impressive speedup of 10.5.

We compare our minibatch parallelization with the iterative parameter mixing (IPM) of McDonald et al. (2010). Figure 3 (bottom) shows that IPM not only offers much smaller speedups, but also converges lower, and this drop in accuracy worsens with more processors.

Figure 4 gives a detailed analysis of speedups. Here we perform both extrinsic and intrinsic comparisons. In the former, we care about the time to reach a given accuracy; in this plot we use 92.27 which is the converged accuracy of IPM on 12 processors. We choose it since it is the lowest accuracy among all converged models; choosing a higher accuracy would reveal even larger speedups for our methods. This figure shows that our method offers superlinear speedups with small number of processors (1 to 6), and almost linear speedups with large number of processors (8 and 12). Note that even $p = 1$ offers a speedup of 1.5 thanks to serial minibatch's faster convergence; in other words, within the 9 fold speed-up at $p = 12$, parallelization contributes about 6 and minibatch about 1.5. By contrast, IPM only offers an almost constant speedup of around 3, which is consistent with the findings of McDonald et al. (2010) (both of their experiments show a speedup of around 3).

We also try to understand where the speedup comes from. For that purpose we study **intrinsic speedup**, which is about the speed regardless of accuracy (see Figure 4). For our minibatch method, intrinsic speedup is the average time per iteration of a parallel run over the serial minibatch baseline. This answers the questions such as "how CPU-efficient is our parallelization" or "how much CPU time is wasted". We can see that with small number of processors (2 to 4), the **efficiency**, defined as S_p/p where S_p is the intrinsic speedup for p processors, is almost 100% (ideal linear speedup), but with more processors it decreases to around 50% with $p = 12$, meaning about half of CPU time is

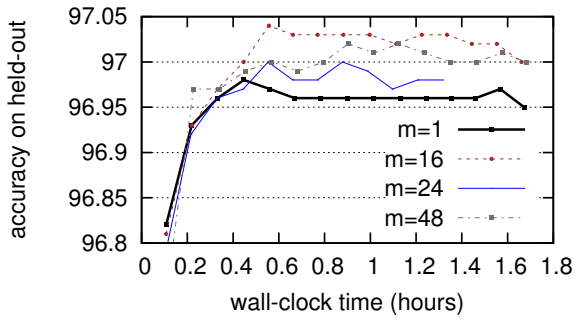


Figure 5: Minibatch learning for tagging with perceptron ($m = 16, 24, 32$) compared with baseline ($m = 1$) for tagging with perceptron. All curves are on single CPU.

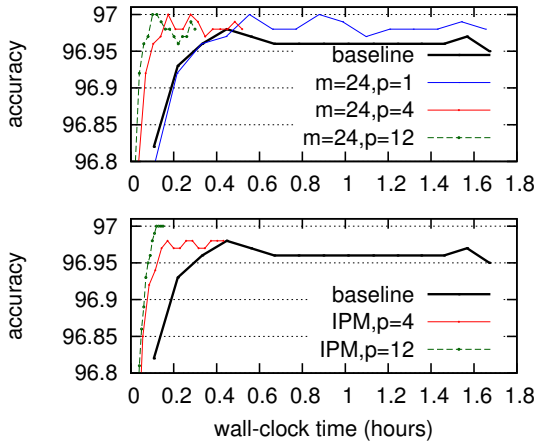


Figure 6: Parallelized minibatch is faster than iterative parameter mixing (on tagging with perceptron). Top: minibatch of size 24 using 4 and 12 processors offers significant speedups over the baselines. Bottom: IPM with the same 4 and 12 processors offers slightly smaller speedups. Note that IPM with 4 processors converges lower than other parallelization curves.

wasted. This wasting is due to two sources: first, the load-balancing problem worsens with more processors, and secondly, the update procedure still runs in serial mode with $p - 1$ processors sleeping.

5.2 Part-of-Speech Tagging with Perceptron

Part-of-speech tagging is usually considered as a simpler task compared to dependency parsing. Here we show that using minibatch can also bring better accuracies and speedups for part-of-speech tagging.

We implement a part-of-speech tagger with averaged perceptron. Following the standard splitting of Penn Treebank (Collins, 2002), we use Sections 00-18 for training and Sections 19-21 as held-out. Our implementation provides an accuracy of 96.98 with beam size 8.

First we run the tagger on a single processor with minibatch sizes 8, 16, 24, and 32. As in Figure 5, we observe similar convergence acceleration and higher accuracies with minibatch. In particular, minibatch of size $m = 16$ provides the highest accuracy of 97.04, giving an improvement of 0.06. This improvement is smaller than what we observe in MIRA learning for dependency parsing experiments, which can be partly explained by the fast convergence of the tagger, and that perceptron does not involve optimization in the updates.

Then we choose minibatch of size 24 to investigate the parallelization performance. As Figure 6 (top) shows, with 12 processors our method takes only 0.10 hours to converge to an accuracy of 97.00, compared to the baseline of 96.98 with 0.45 hours. We also compare our method with IPM as in Figure 6 (bottom). Again, our method converges faster and better than IPM, but this time the differences are much smaller than those in parsing.

Figure 7 uses 96.97 as a criteria to evaluate the extrinsic speedups given by our method and IPM. Again we choose this number because it is the lowest accuracy all learners can reach. As the figure suggests, although our method does not have a higher pure parallelization speedup (intrinsic speedup), it still outperforms IPM.

We are interested in the reason why tagging benefits less from minibatch and parallelization compared to parsing. Further investigation reveals that in tagging the working load of different processors are more unbalanced than in parsing. Figure 8 shows that, when p is small, waiting time is negligible, but when $p = 12$, tagging wastes about 40% of CPU cycles and parser about 30%. By contrast, there is almost no waiting time in IPM and the intrinsic speedup for IPM is almost linear. The communication overhead is not included in this figure, but by comparing it to the speedups (Figures 4 and 7), we conclude that the communication overhead is about 10% for both parsing and tagging at $p = 12$.

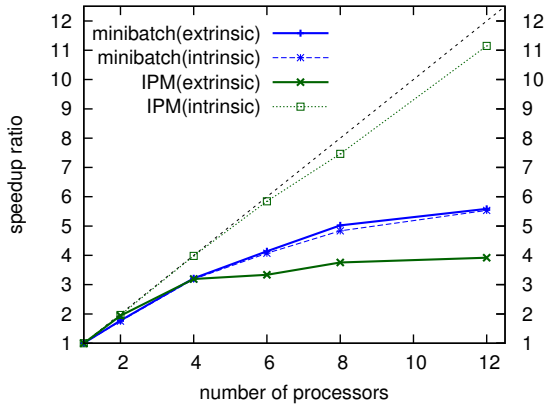


Figure 7: Speedups of minibatch parallelization and IPM on 1 to 12 processors (tagging with perceptron). Extrinsic speedup uses "the time to reach an accuracy of 96.97" as the criterion to measure speed. Intrinsic speedup measures the pure parallelization speedup. IPM has an almost linear intrinsic speedup but a near constant extrinsic speedup of about 3 to 4.

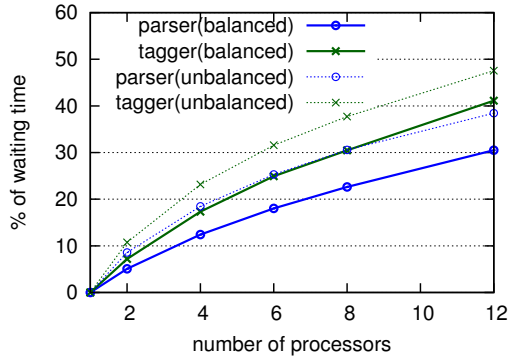


Figure 8: Percentage of time wasted due to synchronization (waiting for other processors to finish) (minibatch $m = 24$), which corresponds to the gray blocks in Figure 1 (b-c). The number of sentences assigned to each processor decreases with more processors, which worsens the unbalance. Our load-balancing strategy (Figure 1 (c)) alleviates this problem effectively. The communication overhead and update time are **not** included.

6 Related Work and Discussions

Besides synchronous minibatch and iterative parameter mixing (IPM) discussed above, there is another method of asychronous minibatch parallelization (Zinkevich et al., 2009; Gimpel et al., 2010; Chiang, 2012), as in Figure 1. The key advantage of asynchronous over synchronous minibatch is that the former allows processors to remain near-constant use, while the latter wastes a significant amount of time when some processors finish earlier than others in a minibatch, as found in our experiments. Gimpel et al. (2010) show significant speedups of asychronous parallelization over synchronous minibatch on SGD and EM methods, and Chiang (2012) finds asynchronous parallelization to be much faster than IPM on MIRA for machine translation. However, asynchronous is significantly more complicated to implement, which involves locking when one processor makes an update (see Fig. 1 (d)), and (in languages like Python) message-passing to other processors after update. Whether this added complexity is worthwhile on large-margin learning is an open question.

7 Conclusions and Future Work

We have presented a simple minibatch parallelization paradigm to speed up large-margin structured learning algorithms such as (averaged) perceptron and MIRA. Minibatch has an advantage in both serial and parallel settings, and our experiments confirmed that a minibatch size of around 16 or 24 leads to a significant speedups over the pure online baseline, and when combined with parallelization, leads to almost linear speedups for MIRA, and very significant speedups for perceptron. These speedups are significantly higher than those of iterative parameter mixing of McDonald et al. (2010) which were almost constant (3$\sim$4) in both our and their own experiments regardless of the number of processors.

One of the limitations of this work is that although decoding is done in parallel, update is still done in serial and in MIRA the quadratic optimization step (Hildreth algorithm (Hildreth, 1957)) scales superlinearly with the number of constraints. This prevents us from using very large minibatches. For future work, we would like to explore parallelized quadratic optimization and larger minibatch sizes, and eventually apply it to machine translation.

Acknowledgement

We thank Ryan McDonald, Yoav Goldberg, and Hal Daumé, III for helpful discussions, and the anonymous reviewers for suggestions. This work was partially supported by DARPA FA8750-13-2-0041 "Deep Exploration and Filtering of Text" (DEFT) Program and by Queens College for equipment.

References

David Chiang. 2012. Hope and fear for discriminative training of statistical translation models. *J. Machine Learning Research (JMLR)*, 13:1159–1187.

C.-T. Chu, S.-K. Kim, Y.-A. Lin, Y.-Y. Yu, G. Bradski, A. Ng, and K. Olukotun. 2007. Map-reduce for machine learning on multicore. In *Advances in Neural Information Processing Systems 19*.

Michael Collins and Brian Roark. 2004. Incremental parsing with the perceptron algorithm. In *Proceedings of ACL*.

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

Koby Crammer and Yoram Singer. 2003. Ultraconservative online algorithms for multiclass problems. *J. Mach. Learn. Res.*, 3:951–991, March.

Jenny Rose Finkel, Alex Kleeman, and Christopher D. Manning. 2008. Efficient, feature-based, conditional random field parsing. In *Proceedings of ACL*.

Kevin Gimpel, Dipanjan Das, and Noah Smith. 2010. Distributed asynchronous online learning for natural language processing. In *Proceedings of CoNLL*.

Clifford Hildreth. 1957. A quadratic programming procedure. *Naval Research Logistics Quarterly*, 4(1):79–85.

Liang Huang and Kenji Sagae. 2010. Dynamic programming for linear-time incremental parsing. In *Proceedings of ACL 2010*.

Liang Huang, Suphan Fayong, and Yang Guo. 2012. Structured perceptron with inexact search. In *Proceedings of NAACL*.

John Lafferty, Andrew McCallum, and Fernando Pereira. 2001. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of ICML*.

Percy Liang and Dan Klein. 2009. Online em for unsupervised models. In *Proceedings of NAACL*.

Ryan McDonald and Fernando Pereira. 2006. Online learning of approximate dependency parsing algorithms. In *Proceedings of EACL*.

Ryan McDonald, Koby Crammer, and Fernando Pereira. 2005. Online large-margin training of dependency parsers. In *Proceedings of the 43rd ACL*.

Ryan McDonald, Keith Hall, and Gideon Mann. 2010. Distributed training strategies for the structured perceptron. In *Proceedings of NAACL*, June.

Shai Shalev-Shwartz, Yoram Singer, and Nathan Srebro. 2007. Pegasos: Primal estimated sub-gradient solver for svm. In *Proceedings of ICML*.

M. Zinkevich, A. J. Smola, and J. Langford. 2009. Slow learners are fast. In *Advances in Neural Information Processing Systems 22*.