ECML'13

Multi-core Structural SVM Training

Kai-Wei Chang, Vivek Srikumar, and Dan Roth

Dept. of Computer Science,
University of Illinois, Urbana-Champaign, IL, USA.
{kchang10,vsrikum2,danr}@illinois.edu

Abstract. Many problems in natural language processing and computer vision can be framed as structured prediction problems. Structural support vector machines (SVM) is a popular approach for training structured predictors, where learning is framed as an optimization problem. Most structural SVM solvers alternate between a model update phase and an inference phase (which predicts structures for all training examples). As structures become more complex, inference becomes a bottleneck and thus slows down learning considerably. In this paper, we propose a new learning algorithm for structural SVMs called DEMI-DCD that extends the dual coordinate descent approach by decoupling the model update and inference phases into different threads. We take advantage of multi-core hardware to parallelize learning with minimal synchronization between the model update and the inference phases. We prove that our algorithm not only converges but also fully utilizes all available processors to speed up learning, and validate our approach on two real-world NLP problems: part-of-speech tagging and relation extraction. In both cases, we show that our algorithm utilizes all available processors to speed up learning and achieves competitive performance. For example, it achieves a relative duality gap of 1% on a POS tagging problem in 192 seconds using 16 threads, while a standard implementation of a multi-threaded dual coordinate descent algorithm with the same number of threads requires more than 600 seconds to reach a solution of the same quality.

1 Introduction

Many prediction problems in natural language processing, computer vision and other fields are structured prediction problems, where decision making involves assigning values to interdependent variables. The output structure can represent sequences, clusters, trees or arbitrary graphs over the decision variables. The *structural support vector machine* (structural SVM) [23] is a widely used approach for training the parameters of structured models. However, training a structural SVM is computationally expensive and this often places limits on the size of the training sets that can be used or limits the expressivity of the structures considered among the interdependent variables. Designing efficient learning algorithms for structural prediction models is therefore an important research question.

Various approaches have been proposed in the literature to learn with the structural SVM algorithm – both exact [22, 12, 3, 4, 13] and approximate [19, 8]. However, these algorithms are inherently single-threaded, and extending them to a multi-core environment is not trivial. Therefore, these algorithms cannot take advantage of the multiple cores available in most modern workstations.

Existing parallel algorithms for training structural SVMs (such as [3]) use a sequence of two phases: an inference phase, where loss-augmented inference is performed using the current model, and a model update phase. Each phase is separated from the other by a barrier that prevents model update until the inference is complete and vice versa. A similar barrier exists in map-reduce implementations of the binary SVM [5] and the Perceptron [17] algorithms. Such barrier-based approaches prevent existing parallel algorithms from fully utilizing the available processors.

In this paper, we propose the DEMI-DCD algorithm (DEcoupled Model-update and Inference with Dual Coordinate Descent), a new barrier-free parallel algorithm based on the dual coordinate descent (DCD) method for the L2-loss structural SVM. DCD has been shown competitive with other optimization techniques such as the cutting plane method [23] and the Frank-Wolfe method [13]. DEMI-DCD removes the need for a barrier between the model update and the inference phases allowing us to distribute these two steps across multiple cores. We show that our approach has the following advantages:

1. DEMI-DCD requires little synchronization between threads. Therefore, it fully utilizes the computational power of multiple cores to reduce training time.
2. As in the standard dual coordinate descent approach, DEMI-DCD can make multiple updates on the structures discovered by the loss-augmented inference, thus fully utilizing the available information. Furthermore, our approach retains the convergence properties of dual coordinate descent.

We evaluate our method on two NLP applications – part-of-speech tagging and entity-relation extraction from text. In both cases, we demonstrate that not only does DEMI-DCD converge faster than existing methods to better performing solutions (according to both primal objective value and test set performance), it also fully takes advantage of all available processors unlike the other methods. For the part-of-speech tagging task, we show that with 16 threads, our approache reaches a relative duality gap of 1% in 192 seconds, while a standard multi-threaded implementation of the dual coordinate descent algorithm with 16 threads takes more than 600 seconds to reach an equivalent solution. Similarly, for the entity-relations task, our approach reaches within 1% of the optimal within 86 seconds, compared to 275 seconds for the baseline.

The rest of this paper is organized as follows. We review the structural SVM model and the DCD method in Section 2. The proposed algorithm is described in Section 3. We survey related methods in Section 4. Empirical results are demonstrated in Section 5. Section 6 provides concluding remarks and discussion.

2 Background: Structural SVM

We are given a set of training examples $\mathcal{D} = \{\mathbf{x}_i, \boldsymbol{y}_i\}_{i=1}^{l}$, where instances $\mathbf{x}_i \in \mathcal{X}$ are annotated with structures $\boldsymbol{y}_i \in \mathcal{Y}_i$. Here the set $\mathcal{Y}_i$ is a set of feasible structures for the i^{th} instance. Training a structural SVMs (SSVM) [23] is framed as the problem of learning a real-valued weight vector $\boldsymbol{w}$ by solving the following optimization problem:

$$\begin{aligned} \min_{\boldsymbol{w},\boldsymbol{\xi}} \quad & \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + C\sum_i \ell(\xi_i) \\ s.t. \quad & \boldsymbol{w}^T\Phi(\boldsymbol{x}_i,\boldsymbol{y}_i) - \boldsymbol{w}^T\Phi(\boldsymbol{x}_i,\boldsymbol{y}) \geq \Delta(\boldsymbol{y}_i,\boldsymbol{y}) - \xi_i, \quad \forall i, \boldsymbol{y} \in \mathcal{Y}_i. \end{aligned} \tag{1}$$

where $\Phi(\boldsymbol{x},\boldsymbol{y})$ is the feature vector extracted from input the $\boldsymbol{x}$ and output $\boldsymbol{y}$ and $\ell(\xi)$ is the loss that needs to be minimized. The constraints in (1) indicate that for all training examples and all possible output structures, the score for the correct output structure $\boldsymbol{y}_i$ is greater than the score for other output structures $\boldsymbol{y}$ by at least $\Delta(\boldsymbol{y}_i,\boldsymbol{y})$. The slack variable $\xi_i \geq 0$ penalizes the violation. The loss ℓ is an increasing function of the slack that is minimized as part of the objective: when $\ell(\xi) = \xi$, we refer to (1) as an L1-loss structural SVM, while when $\ell(\xi) = \xi^2$, we call it an L2-loss structural SVM.[1] For mathematical simplicity, in this paper, we only consider the linear L2-loss structural SVM model, although our method can potentially be extended to other variants of the structural SVM.

Instead of directly solving (1), several optimization algorithms for SSVM consider its dual form [23, 12, 4] by introducing dual variables $\alpha_{i,\boldsymbol{y}}$ for each output structure $\boldsymbol{y}$ and each example $\boldsymbol{x}_i$. If $\boldsymbol{\alpha}$ is the set of all dual variables, the dual problem can be stated as

$$\begin{aligned} &\min_{\boldsymbol{\alpha}>0} \quad D(\boldsymbol{\alpha}), \text{ and} \\ &D(\boldsymbol{\alpha}) \equiv \frac{1}{2}\left\|\sum_{\alpha_{i,\boldsymbol{y}}} \alpha_{i,\boldsymbol{y}}\phi(\boldsymbol{y},\boldsymbol{y}_i,\boldsymbol{x}_i)\right\|^2 + \frac{1}{4C}\sum_i\left(\sum_{\boldsymbol{y}}\alpha_{i,\boldsymbol{y}}\right)^2 - \sum_{i,\boldsymbol{y}}\Delta(\boldsymbol{y},\boldsymbol{y}_i)\alpha_{i,\boldsymbol{y}}, \end{aligned} \tag{2}$$

where $\phi(\boldsymbol{y},\boldsymbol{y}_i,\boldsymbol{x}_i) = \phi(\boldsymbol{y}_i,\boldsymbol{x}_i) - \phi(\boldsymbol{y},\boldsymbol{x}_i)$. The constraint $\boldsymbol{\alpha} \geq 0$ restricts all the dual variables to be non-negative (i.e., $\alpha_{i,\boldsymbol{y}} \geq 0 \forall i, \boldsymbol{y}$).

For the optimal values, the relationship between the primal optimal $\boldsymbol{w}^*$ (that is, the solution of (1)), and the dual optimal $\boldsymbol{\alpha}^*$ (that is, the solution of (2)) is

$$\boldsymbol{w}^* = \sum_{i,\boldsymbol{y}} \alpha^*_{i,\boldsymbol{y}}\phi(\boldsymbol{y},\boldsymbol{y}_i,\boldsymbol{x}_i).$$

Although this relationship only holds for the solutions, in a linear model, one can maintain a temporary vector

$$\boldsymbol{w} \equiv \sum_{i,\boldsymbol{y}} \alpha_{i,\boldsymbol{y}}\phi(\boldsymbol{y},\boldsymbol{y}_i,\boldsymbol{x}_i) \tag{3}$$

to assist the computations [10].

[1] In L2-loss structural SVM formulation, one may replace $\Delta(\boldsymbol{y}_i,\boldsymbol{y})$ by $\sqrt{\Delta(\boldsymbol{y}_i,\boldsymbol{y})}$ to obtain an upper bound on the empirical risk [23]. However, we keep using $\Delta(\boldsymbol{y}_i,\boldsymbol{y})$ for computational and notational convenience. Thus, Eq. (1) minimizes the mean square loss with a regularization term.

In practice, for most definitions of structures, the set of feasible structures for a given instance (that is, $\mathcal{Y}_i$) is exponentially large, leading to an exponentially large number of dual variables. Therefore, existing dual methods [4, 23] maintain an active set of dual variables $\mathcal{A}$ (also called the working set in the literature). During training, only the dual variables in $\mathcal{A}$ are considered for an update and the rest $\alpha_{i,\boldsymbol{y}} \notin \mathcal{A}$ are fixed to 0. We denote the active set associated with the instance $\boldsymbol{x}_i$ as $\mathcal{A}_i$ and $\mathcal{A} = \bigcup_i \mathcal{A}_i$. The following theorem justifies the use of an active set.

Theorem 1. *Let $\boldsymbol{\alpha}^*$ be the optimal solution of* (2) *and $\mathcal{A}^* = \{\alpha^*_{i,\boldsymbol{y}} \mid \alpha^*_{i,\boldsymbol{y}} > 0\}$. Then any optimal solution of*

$$\min_{\boldsymbol{\alpha}\geq 0} \quad D(\alpha) \quad \textit{s.t. } \alpha_{i,\boldsymbol{y}} = 0, \quad \forall \alpha_{i,\boldsymbol{y}} \notin \mathcal{A}^* \tag{4}$$

is an optimal solution of (2).

This suggests that we can reduce the size of the optimization problem by carefully identifying nonzero α's. This property of the dual is widely used for training SVMs, for example, with the cutting-plane method [23, 12], with a dual coordinate descent method [3, 4], and has also been used for solving binary SVM [11, 10, 2].

We observe that across all these methods, in a single-thread implementation, training consists of two phases:

1. Updating the values $\alpha_{i,\boldsymbol{y}} \in \mathcal{A}$ (learning step), and
2. Selecting and maintaining the active set $\mathcal{A}$ (active set selection step).

The learning step usually updates each dual variable $\alpha_{i,\boldsymbol{y}} \in \mathcal{A}$ several times until convergence for the current active set. The active set selection step involves solving the following loss-augmented inference problem for each example $\boldsymbol{x}_i$:

$$\max_{\boldsymbol{y}\in\mathcal{Y}_i} \quad \boldsymbol{w}^T\phi(\boldsymbol{x}_i, \boldsymbol{y}) + \varDelta(\boldsymbol{y}_i, \boldsymbol{y}) \tag{5}$$

Solving loss-augmented inference is usually computationally more expensive than the time for updating the model. In the traditional sequential implementations, these two steps block each other. Even if inference for each example is performed in parallel on a multi-core machine, the model update cannot be done until inference is solved for all training examples. Similarly, inference cannot start until the model update is complete. Balancing the time spent on these two parts is a crucial aspect of algorithm design. In the next section, we will show that, on a multi-core machine, we can indeed fully utilize the available computational power without the barrier between the two training steps.

3 Parallel Strategies for Structured Learning

In this section, we describe the parallel learning algorithm DEMI-DCD which decouples the model update steps from the inference steps during learning, and hence fully utilizes the computational power of multi-core machines.

Let p be the number of threads allocated for learning. DEMI-DCD first splits the training data $\mathcal{D}$ into $p-1$ disjoint parts: $\{B_j\}_{j=1}^{p-1}$ with each $B_j \subset \mathcal{D}$. Then, it generates two types of threads, a learning thread and $p-1$ active set selection threads. The j^{th} active set selection thread is responsible for maintaining an active set $\mathcal{A}_i$ for each example i in the part B_j. It does so by searching for candidate structures for each instance in B_j using the current model $\boldsymbol{w}$ which is maintained by the learning thread.

The learning thread loops over all the examples and updates the model $\boldsymbol{w}$ using $\alpha_{i,\boldsymbol{y}} \in \mathcal{A}_i$ for the i^{th} example. The $p-1$ active set selection threads are independent of each other, and they share $\mathcal{A}_i$ and $\boldsymbol{w}$ with the learning thread using shared memory buffers. We will now discuss our model in detail. The algorithms executing the learning and the active set selection threads are listed as Algorithm 1 and 2 respectively.

Learning thread The learning thread performs a two-level iterative procedure until the stopping conditions are satisfied. It first initializes $\boldsymbol{\alpha}^0$ to a zero vector, then it generates a sequence of solutions $\{\boldsymbol{\alpha}^0, \boldsymbol{\alpha}^1, \ldots\}$. We refer to the step from $\boldsymbol{\alpha}^t$ to $\boldsymbol{\alpha}^{t+1}$ as the outer iteration. Within this iteration, the learning thread sequentially visits each instance $\boldsymbol{x}_i$ in the data set and updates $\alpha_{i,\boldsymbol{y}} \in \mathcal{A}_i$ while all the other dual variables are kept fixed. To update $\alpha_{i,\boldsymbol{y}}$, it solves the following one variable sub-problem that uses the definition of the dual objective function from Eq. (2):

$$\begin{aligned} \bar{d}_{i,\boldsymbol{y}} &= \arg\min_d D(\boldsymbol{\alpha} + \mathbf{e}_{i,\boldsymbol{y}} d) \quad \text{s.t.} \quad d\alpha_{i,\boldsymbol{y}} + d \geq 0 \\ &= \arg\min_d \frac{1}{2}\|\boldsymbol{w} + \phi(\boldsymbol{y}, \boldsymbol{y}_i, \boldsymbol{x}_i) d\|^2 + \frac{1}{4C}\left(d + \sum_{\boldsymbol{y}\in\mathcal{A}_i} \alpha_{i,\boldsymbol{y}}\right)^2 - d\Delta(y_i, \boldsymbol{y}) \\ &\quad \text{s.t.} \quad \alpha_{i,\boldsymbol{y}} + d \geq 0. \end{aligned} \tag{6}$$

Here, $\boldsymbol{w}$ is defined in Eq. (3) and $\mathbf{e}_{i,\boldsymbol{y}}$ is a vector where only the element corresponding to $(i, \boldsymbol{y})$ is one and the rest are zero. Eq. (6) is a quadratic optimization problem with one variable and has an analytic solution. Therefore, the update rule of $\alpha_{i,\boldsymbol{y}}$ can be written as:

$$\begin{aligned} \bar{d}_{i,\boldsymbol{y}} &\leftarrow \frac{\Delta(\boldsymbol{y}, \boldsymbol{y}_i) - \boldsymbol{w}^T\phi(\boldsymbol{y}, \boldsymbol{y}_i, \boldsymbol{x}_i) - \frac{1}{(2C)}\sum_{\boldsymbol{y}'}\alpha_{i,\boldsymbol{y}'}}{\|\phi(\boldsymbol{y}, \boldsymbol{y}_i, \boldsymbol{x}_i)\|^2 + \frac{1}{(2C)}}, \\ \alpha_{i,\boldsymbol{y}} &\leftarrow \max(\alpha_{i,\boldsymbol{y}} + \bar{d}_{i,\boldsymbol{y}}, 0). \end{aligned} \tag{7}$$

To maintain the relation between $\boldsymbol{\alpha}$ and $\boldsymbol{w}$ specified in Eq. (3), $\boldsymbol{w}$ is updated accordingly:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + d_{i,\boldsymbol{y}}\phi(\boldsymbol{y}, \boldsymbol{y}_i, \boldsymbol{x}_i). \tag{8}$$

We will now discuss two implementation issues to improve DEMI-DCD. First, during the learning, $\boldsymbol{w}$ is shared between the learning and the active set selection threads. Therefore, we would like to maintain $\boldsymbol{w}$ in a shared buffer. However, the update rules in Steps (7)-(8) can be done in $O(\bar{n})$, where $\bar{n}$ is number of average active features of $\phi(\boldsymbol{x}, \boldsymbol{y})$. Thus, when the number of active features is small, the updates are quick. Hence, we require to maintain a lock to prevent the active set selection threads from accessing $\boldsymbol{w}$ when it is being updated. To reduce the cost of synchronization, we maintain a local copy of $\boldsymbol{w}$ and copy $\boldsymbol{w}$ to a shared buffer (denoted by $\bar{\mathrm{w}}$) after every ρ updates. This reduces the overhead of synchronization.

Algorithm 1 Learning Thread

```
Input: Dataset $\mathcal{D}$ and the number of iterations before updating the shared buffer, $\rho$.
Output: The learned model $\boldsymbol{w}$
 1: $\boldsymbol{w} \leftarrow \mathbf{0}, \boldsymbol{\alpha} \leftarrow \mathbf{0}$, #updates $\leftarrow 0$.
 2: while stopping conditions are not satisfied do
 3:   for $i = 1 \rightarrow l$ (loop over each instance) do
 4:     for all $\boldsymbol{y}$ in $\mathcal{A}_i$ do
 5:       if Eq. (9) is satisfied then
 6:         $\mathcal{A}_i \leftarrow \mathcal{A}_i \setminus \{\alpha_{i,\boldsymbol{y}}\}$.
 7:       else
 8:         update corresponding $\alpha_{i,\boldsymbol{y}}$ by Eq. (7)-Eq.(8),
 9:         #updates $\leftarrow$ #updates + 1.
10:       end if
11:       if #updates mod $\rho$ = 0 then
12:         Copy $\boldsymbol{w}$ to $\bar{\mathrm{w}}$ in a shared buffer.
13:       end if
14:     end for
15:   end for
16: end while
```

Second, as the active set selection thread keeps adding dual variables into $\mathcal{A}$, the size of $\mathcal{A}$ grows quickly. To avoid the learning thread from wasting time on the bounded dual variables, we implement a shrinking strategy inspired by [11][2]. Specifically, if $\alpha_{i,\bar{\boldsymbol{y}}}$ equals to zero and

$$-\nabla(\boldsymbol{\alpha})_{i,\bar{\boldsymbol{y}}} = \Delta(\bar{\boldsymbol{y}}, \boldsymbol{y}_i) - \boldsymbol{w}^T \phi(\bar{\boldsymbol{y}}, \boldsymbol{y}_i, \boldsymbol{x}_i) - \frac{1}{2} \sum_{\boldsymbol{y} \in \mathcal{A}_i} \alpha_{i,\boldsymbol{y}} < \delta \tag{9}$$

then DEMI-DCD removes $\alpha_{i,\bar{\boldsymbol{y}}}$ from $\mathcal{A}_i$. Notice that the shrinking strategy is more aggressive if δ is large. For binary classification, a negative δ is usually used. This is because, in the binary classification case, the size of the data is usually large (typically millions of examples); therefore incorrectly removing an instance from the active set requires a lengthy process that iterates over all the examples to add it back. However, in our case, aggressive shrinking strategy is safe because the active set selection threads can easily add the candidate structures back. Therefore, in our implementation, we set δ to 0.01.

Active set selection threads As mentioned, the j^{th} active set selection thread iterates over all instances $\boldsymbol{x}_i \in B_j$ and selects candidate active variables based on solving an augmented inference problem (line 4 in Algorithm 2), that is, eq. (5). Just like the learning thread, each active set selection thread maintains a local copy of $\boldsymbol{w}$. This setting aims to prevent $\boldsymbol{w}$ from being changed while the loss augmented inference is being solved, thus avoiding a possibly suboptimal solution. The local copy of $\boldsymbol{w}$ will be updated from $\bar{\mathrm{w}}$ in the shared buffer after each ρ iterations.

[2] This shrinking strategy is also related to the condition used by cutting plane methods for adding constraints into the working set.

Algorithm 2 The j^{th} Active Set Selection Thread

```
Input: Part of dataset for this thread $\mathcal{B}_j$, and the number of iterations before updating the shared
    buffer, $\rho$.
 1: $\boldsymbol{w} \leftarrow \mathbf{0}$ (a local copy), #Inference $\leftarrow 0$,
 2: while Learning thread is not stopped do
 3:     for all $(\boldsymbol{x}_i, \boldsymbol{y}_i) \in \mathcal{B}_j$ do
 4:         $\bar{\boldsymbol{y}} \leftarrow f_{\text{AugInf}}(\boldsymbol{w}, \boldsymbol{x}_i, \boldsymbol{y}_i)$.
 5:         if $\bar{\boldsymbol{y}} \notin \mathcal{A}_i$ then
 6:             $\mathcal{A}_i \leftarrow \{\mathcal{A}_i \cup \bar{\boldsymbol{y}}\}$.
 7:         end if
 8:         #Inference $\leftarrow$ #Inference + 1
 9:         if #Inference mod $\rho$ = 0 then
10:             Copy from the model $\bar{\mathrm{w}}$ in shared buffer to $\boldsymbol{w}$.
11:         end if
12:     end for
13: end while
```

Synchronization Our algorithm requires little synchronization between threads. In fact, only the learning thread can write to $\bar{\mathrm{w}}$ and only the j^{th} active set selection thread can modify $\mathcal{A}_i$ for any $i \in B_j$. It is very unlikely, but possible, that the learning thread and the inference threads will be reading/writing $\bar{\mathrm{w}}$ or $\mathcal{A}_i$ concurrently. To avoid this, one can use a mutex lock to ensure that the copy operation is atomic. However, in practice, we found that this synchronization is unnecessary.

3.1 Analysis

In this section, we analyze the proposed multi-core algorithm. We observed that α's can be added to the $\mathcal{A}$ by the active set selection thread, but might be removed by the learning thread with the shrinking strategy. Therefore, we define $\bar{\mathcal{A}}$ to be a subset of $\mathcal{A}$ that contains $\alpha_{i,\boldsymbol{y}}$ which has been visited by the learning thread at least once and remains in the $\mathcal{A}$.

Theorem 2. *The number of variables which have been added to $\bar{\mathcal{A}}$ during the entire training process is bounded by $O(1/\delta^2)$.*

The proof follows from [4, Theorem 1]. This theorem says that the size of $\bar{\mathcal{A}}$ is bounded as a function of δ.

Theorem 3. *If $\bar{\mathcal{A}} \neq \emptyset$ is not expanded, then the proposed algorithm converges to an ϵ-optimal solution of*

$$\min_{\boldsymbol{\alpha} \geq 0} \quad D(\boldsymbol{\alpha}) \quad \textit{s.t. } \alpha_{i,\boldsymbol{y}} = 0, \forall \boldsymbol{y} \notin \mathcal{A}_i \tag{10}$$

in $O(log(1/\epsilon))$ steps.

If $\bar{\mathcal{A}}$ is fixed, our learning thread performs standard dual coordinate descent, as in [10]. Hence, this theorem follows from the analysis of dual coordinate descent. The global convergence rate of the method can be inferred from [24] which generalizes the proof in [15].

Theorem 2 shows that the size of $\bar{\mathcal{A}}$ will eventually stop growing. Theorem 3 shows that when $\bar{\mathcal{A}}$ is fixed, the weight vector $\boldsymbol{w}$ converges to the optimum of (10). Hence, the local copies in the learning thread and the active set selection threads can be arbitrary close. Following the analysis in [4], the convergence of DEMI-DCD then follows.

4 Related Work

Several related works have a resemblance to the method proposed in this paper. In the following, we briefly review the literature and discuss the connections.

A Parallel Algorithm for Dual Coordinate Descent The structural SVM package JLIS [3] implements a parallel algorithm in a Master-Slave architecture to solve Eq. (2).[3], Given p processors, it first splits the training data into p parts. Then the algorithm maintains a model $\boldsymbol{w}$, dual variables $\boldsymbol{\alpha}$, and an active set $\mathcal{A}$ and updates these in an iterative fashion. In each iteration, the master thread sends one part of the data to a slave thread. For each slave thread, it iterates over each assigned example $\boldsymbol{x}_i$, and picks the best structures $\bar{\boldsymbol{y}}$ according to current $\boldsymbol{w}$. Then $(\boldsymbol{x}_i, \bar{\boldsymbol{y}})$ is added into the active set $\mathcal{A}$ if the following condition is satisfied:

$$\Delta(\bar{\boldsymbol{y}}, \boldsymbol{y}_i) - \boldsymbol{w}^T \phi(\bar{\boldsymbol{y}}, \boldsymbol{y}_i, \boldsymbol{x}_i) - \frac{1}{2} \sum_{\boldsymbol{y} \in \mathcal{A}_i} \alpha_{i,\boldsymbol{y}} > \delta. \tag{11}$$

Only after all the slave threads have finished processing all the examples, the master thread performs dual coordinate descent updates to solve the following optimization loosely:

$$\min_{\boldsymbol{\alpha} \geq 0} \quad D(\boldsymbol{\alpha}) \quad \text{s.t. } \alpha_{i,\boldsymbol{y}} = 0, \forall \boldsymbol{y} \notin \mathcal{A}_i.$$

The algorithm stops when a stopping condition is reached. This approach is closely related to the n-slack cutting plane method for solving structural SVM [23]. However, [23] assumes that the sub-problem is solved exactly[4], while this restriction can be relaxed under a dual coordinate descent framework.

We will refer to this approach for parallelizing structural SVM training as MS-DCD (for Master-Slave dual coordinate descent) and compare it experimentally with DEMI-DCD in Section 5.

Structured Perceptron and its parallel version The Structured Perceptron [6] algorithm has been widely used in the literature. At each iteration, it picks an example $\mathbf{x}_i$ that is annotated with $\boldsymbol{y}_i$ and finds its best structured output $\bar{\boldsymbol{y}}$ according to the current model $\boldsymbol{w}$ using an inference algorithm. Then, the model is updated as

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \eta(\phi(\mathbf{x}_i, \boldsymbol{y}_i) - \phi(\mathbf{x}_i, \bar{\boldsymbol{y}})),$$

[3] The implementation of MS-DCD can be downloaded at `http://cogcomp.cs.illinois.edu/page/software_view/JLIS`.

[4] The cutting-plane solver for structural SVM usually sets a tolerance parameter, and stops the sub-problem solver when the inner stopping criteria is satisfied.

where η is a learning rate. Notice that the Structural Perceptron requires an inference step before each update. This makes it different from the dual methods for structured SVMs where the inference step is used to update the active set. One important advantage of the dual methods is that they can perform multiple updates on the elements in the active set without doing inference for each update. As we will show in Section 5, this limits the efficiency of the Perceptron algorithm. Some caching strategies have been developed for structural Perceptron. For example, [7] introduces a caching technique to periodically update the model with examples on which the learner had made mistakes in previous steps. However, this approach is ad-hoc without convergence guarantees.

A parallelization strategy for structured Pereceptron (SP-IPM) has been proposed [18] using the Map-Reduce framework. It calls for first splitting the training data into several parts. In the map phase, it distributes data to each mapper and runs a separate Perceptron learner on each shard in parallel. Then, in the reduce phase, the models are mixed using a linear combination. The mixed model serves as the initial model for the next round. In this paper, we implement this map-reduce framework as a multi-thread program. SP-IPM requires barrier synchronizations between the map and reduce phases, which limits the computational performance. In addition, in the model mixing strategy, each local model is updated using exclusive data blocks. Therefore, the mappers might make updates that are inconsistent with each other. As a result, it requires many iterations to converge.

General Parallel Algorithms for Convex Optimization Some general parallel optimization algorithms have been proposed in the literature. For example, delayed stochastic (sub-)gradient descent methods have been studied with assumptions on smoothness of the problem [1, 14]. However, their applicability to structured SVM has not been explored.

5 Experiments

In this section, we show the effectiveness of the DEMI-DCD algorithm compared to other parallel structured learning algorithms.

5.1 Experimental Setup

We evaluate our algorithm on two natural language processing tasks: part-of-speech tagging (POS-WSJ) and jointly predicting entities and their relations (Entity-Relation). These tasks, which have very different output structures, are described below.

POS tagging (POS-WSJ) POS tagging is the task of labeling each word in a sentence with its part of speech. This task is typically modeled as a sequence labeling task, where each tag is associated with emission features that capture word-tag association and transition features that capture sequential tag-tag association. For this setting, inference can be solved efficiently using the Viterbi algorithm.

We use the standard Penn Treebank Wall Street Journal corpus [16] to train and evaluate our POS tagger using the standard data split for training (sections 2-22, 39832 sentences) and testing (section 23, 2416 sentences). In our experiments, we use indicators for the conjunction between the word and tags as emission features and pairs of tags as transition features.

Entity and Relation Recognition (Entity-Relation) This is the task of assigning entity types to spans of text and identifying relations among them [20]. For example, for the sentence *John Smith resides in London*, we would predict *John Smith* to be a PERSON, LONDON to be a LOCATION and the relation LIVES-IN between them.

As in the original work, we modeled prediction as a 0-1 linear program (ILP) where binary indicator variables capture all possible entity-label decisions and entity pair label decisions (that is, relation labels). Linear constraints force exactly one label to be assigned with each entity and relation. In addition, the constraints also encode background knowledge about the types of entities allowed for each relation label. For example, a LIVES-IN relation can only connect a PERSON to a LOCATION. We refer the reader to [21] for further details. Unlike the original paper, which studied a decomposed learning setting, we jointly train the entity-relation model using an ILP for the loss-augmented inference. We used the state-of-the-art Gurobi ILP solver [9] to solve inference for this problem both during training and test. We report results using the annotated data from [21] consisting of 5925 examples with an 80-20 train-test split.

We based our implementation on the publicly available implementation of MS-DCD, which implements a dual coordinate descent method for solving structural SVM. DCD [4] has been shown competitive comparing to other L2-loss and L1-loss structural SVM solvers such as a 1-slack variable cutting-plane method [12] and a Frank-Wolfe optimization method [13] when using one CPU core.

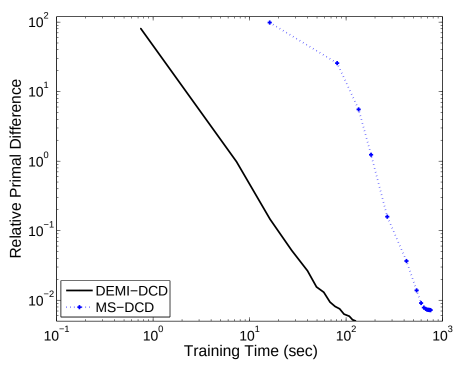
As described in Section 3, our method is an extension of DCD and further improves its performance using multiple cores. All the algorithms are implemented in Java. We conducted our experiments on a 24-core machine with Xeon E5-2440 processors running 64-bit Scientific Linux. Unless otherwise stated, all results use 16 threads. We set the value of C to 0.1 for all experiments.

Our experiments compare the following three methods:

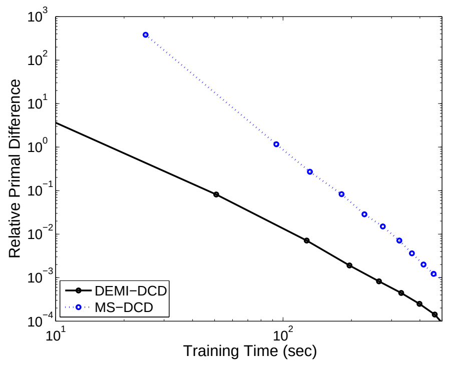
1. DEMI-DCD: the proposed algorithm described in Section 3. We use one thread for learning and the rest for inference (that is, active set selection).
2. MS-DCD: A master-slave style parallel implementation of dual coordinate descent method from JLIS package [3] described in Section 4.
3. SP-IPM: A parallel structural Perceptron algorithm proposed in [18]. We run 10 epochs of Perceptron updates for each shard at each outer iteration.

Note that the first two methods solve an L2-Structural SVM problem (1) and converge to the same minimum.

Our experiments answer the following research question: Can DEMI-DCD make use of the available CPU resources to converge faster to a robust structural prediction model? To answer this, for both our tasks, we first compare the convergence speed of DEMI-DCD and MS-DCD. Second, we compare the performance of the three algorithms on the test sets of the two tasks. Third, we show that DEMI-DCD maximally utilizes all available CPU cores unlike the other two algorithms. Finally, we report the



(a) POS-WSJ



(b) Entity-Relation

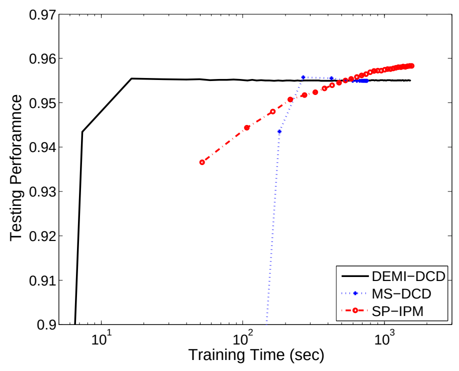
Fig. 1: Relative primal function value difference to the reference model versus wall clock time. See the text for more details.

results of an analysis experiment, where we show the performance of our algorithm with different number of available threads.

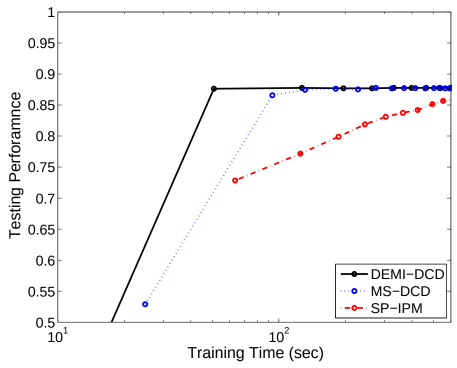
5.2 Empirical Convergence Speed

First, we compare DEMI-DCD to MS-DCD in terms of their speed of convergence in terms of objective function value. We omit SP-IPM in this comparison because it does not solve the SVM optimization problem. Figure 1 shows the relative primal objective function (that is, the value of the objective in Eq. (1)) with respect to a reference model $\boldsymbol{w}^*$ as a function of wall-clock training time. In other words, if the objective function is denoted by f, we plot $(f(\boldsymbol{w}) - f(\boldsymbol{w}^*))\,/f(\boldsymbol{w}^*)$ as $\boldsymbol{w}$ varies with training time for each algorithm. The reference model is the one that achieves the lowest primal value among the compared models. In the figure, both the training time and the relative difference are shown in log-scale. From the results, we see that the proposed method is faster than MS-DCD on both tasks. DEMI-DCD is especially faster than MS-DCD in the early stage of optimization. This is important because usually we can achieve a model with reasonable generative performance before solving (1) exactly. Note that the inference in Entity-Relation is much slower than inference for POS-WSJ. For example, at each iteration on POS-WSJ, MS-DCD takes 0.62 seconds to solve inference on all the training samples using 16 threads and takes 3.87 seconds to update the model using 1 thread, while it takes 10.64 seconds to solve the inference and 8.45 seconds to update the model on Entity-Relation. As a result, the difference between DEMI-DCD and MS-DCD is much higher on POS-WSJ. We will discuss the reason for this in Section 5.3.

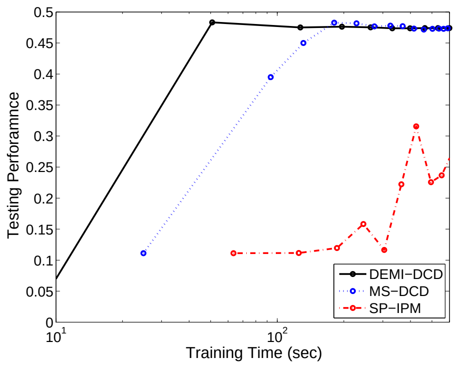
Next, we show that the proposed method can obtain a reasonable and stable solution in a short time. Figure 2 shows the test set performance of the three algorithms as training proceeds. For POS-WSJ, we evaluate the model using token-based accuracy. For Entity-Relation, we evaluate the entity and relation labels separately and report the micro-averaged F1 over the test set. In all cases, DEMI-DCD is the fastest one to achieve a converged performance. As mentioned, DEMI-DCD is more efficient than MS-DCD in the early iterations. As a result, it takes less time to generate a reasonable



(a) POS-WSJ: Token-wise accuracy



(b) Entity-Relation: Entity F1

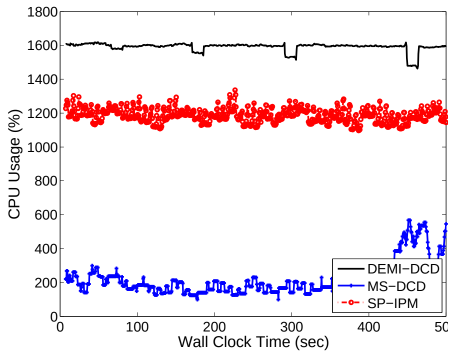


(c) Entity-Relation: Relation F1

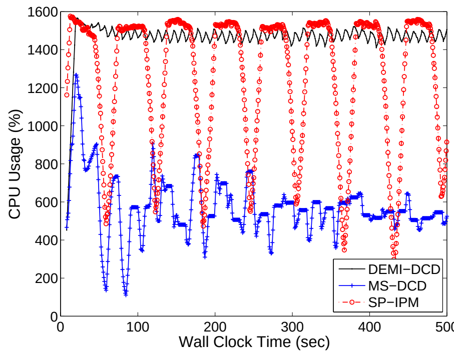
Fig. 2: Performance of POS-WSJ and Entity-Relation plotted as a function of wall clock training time. For Entity-Relation, we report the F1 scores of both entity and relation tasks. Note that the X-axis is in **log scale**. We see that DEMI-DCD converges faster to better performing models.

model. Note that SP-IPM achieves better final performance on the POS-WSJ task. This is so because SP-IPM converges to a different model from DEMI-DCD and MS-DCD. For both entities and relations, SP-IPM converges slowly and, moreover, its performance is unstable for the relations. We observe that the convergence of SP-IPM is slow because the Perceptron algorithm needs to solve an inference problem before each update. When the inference is slow, the number of updates in SP-IPM is significantly smaller than DEMI-DCD and MS-DCD. For example, in the Entity-Relation task, DEMI-DCD makes around 55 million updates within 100 seconds, while SP-IPM only makes 76 thousand updates in total.[5] In other words, DEMI-DCD is faster and better than SP-IPM because it performs many inexpensive updates.

[5] SP-IPM has 16 threads running on different shards of data in parallel. We simply sum up all the updates on different shards.



(a) POS-WSJ (1,318%, 257%, 1,177%)



(b) Entity-Relation (1,462%, 527%, 1248%)

Fig. 3: CPU usage of each method during training. The numbers listed in the caption are the average CPU usage percentage per second for DEMI-DCD, MS-DCD, SP-IPM, respectively. We show moving averages of CPU usage with a window of 2 seconds. Note that we allow all three algorithms to use 16 threads in a 24 core machine. Thus, while all the algorithms can report upto 1600% CPU usage, only DEMI-DCD consistently reports high CPU usage.

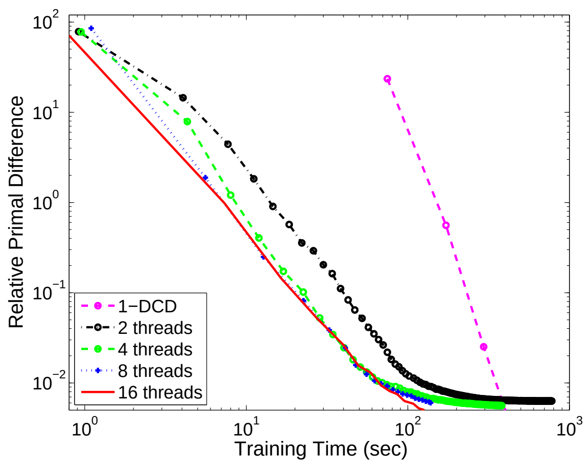
5.3 CPU Utilization

Finally, we investigate the CPU utilization of the three algorithms by plotting the *moving average* of CPU usage in Figure 3 during training, as reported by the Unix command `top`. Since we provide all algorithms with 16 threads, the maximum CPU utilization can be 1600%. The results show that DEMI-DCD almost fully utilizes the available CPU resources.

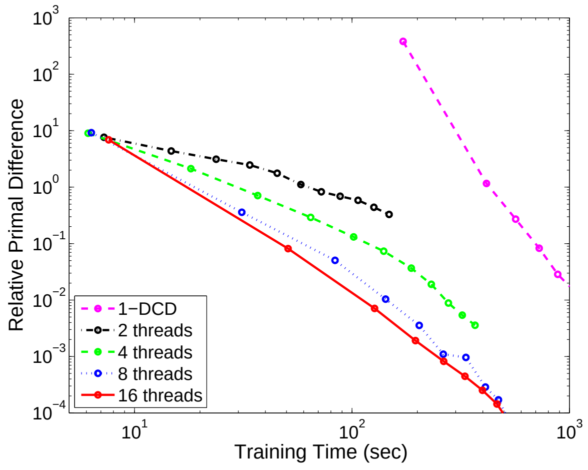
We see that neither baseline manages to use the available resources consistently. The average CPU usage for MS-DCD on POS-WSJ is particularly small because the inference step on this task is relatively easy (i.e. using the Viterbi algorithm). In fact, MS-DCD spends only around 25% of the time on the inference steps, which is solved by the slave threads in parallel. Both MS-DCD and SP-IPM require a barrier to ensure that the subtasks sent to the slave threads are complete. This barrier limits the CPU utilization and hence slows down the learners. Since inference is more expensive (i.e. an ILP call) for the Entity-Relation case, more time is spent on inference in the learning algorithms. Since this step is distributed across the slaves for both MS-DCD and SP-IPM, we see periods of high CPU activity followed by low activity (when only the master thread is active).

5.4 Performance with Different Number of Threads

In our final set of experiments, we study the performance of DEMI-DCD for different number of threads. Figure 4 shows the change in the primal objective function value difference as a function of training time for different number of threads. Note that the training time and the function values are shown in *log-scale*. For comparison, we also show the performance of a DCD implementation using only one CPU core (1-DCD).



(a) POS-WSJ



(b) Entity-Relation

Fig. 4: Relative primal function value difference along training time using different number of threads. We also show a DCD implementation using one CPU core (1-DCD). Both x-axis and y-axis are in **log scale**.

As can be seen in the figure, the training time is reduced as the number of threads increases. With multiple threads, DEMI-DCD is significantly faster than 1-DCD. However, when the number of threads is more than 8, the difference is small. That is because the inference step (i.e, the active set selection step) is no longer the bottleneck.

6 Discussion and Conclusion

In this paper, we have proposed a new learning algorithm for training structural SVMs, called DEMI-DCD. This algorithm decouples the model update and inference phases of learning allowing us to execute them in parallel, thus allowing us taking advantage of multi-core machines for training. We showed that the DEMI-DCD algorithm converges to the optimum solution of the structural SVM objective. We experimentally evaluated our algorithm on the structured learning tasks of part-of-speech tagging and entity-relation extraction and showed that it outperforms existing strategies for training structured predictors in terms of both convergence time and CPU utilization.

The approach proposed in this paper opens up several directions for future research. Here, we have considered the case of a single learning thread that updates the models using the available active sets. A promising direction for future exploration is to explore the possibility of using multiple learning threads to update the models.

For some structured prediction problems, the output structure may be too complex for inference to be solved in a tractable fashion. For such cases, learning schemes with approximate inference have been proposed (e.g., [8, 19]). Incorporating approximate inference into our method is an interesting topic for future study.

Acknowledgments We thank Ming-Wei Chang for helpful discussions. This material is based on research sponsored by DARPA under agreement number FA8750-13-2-0008. The U. S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U. S. Government. This work is also supported by an ONR Award on "Guiding Learning and Decision Making in the Presence of Multiple Forms of Information."

References

1. Agarwal, A., Duchi, J.: Distributed delayed stochastic optimization. In Shawe-Taylor, J., Zemel, R., Bartlett, P., Pereira, F., Weinberger, K., eds.: NIPS. (2011) 873–881
2. Chang, K., Roth, D.: Selective block minimization for faster convergence of limited memory large-scale linear models. In: KDD. (2011)
3. Chang, M., Srikumar, V., Goldwasser, D., Roth, D.: Structured output learning with indirect supervision. In: ICML. (2010)
4. Chang, M.W., Yih, W.T.: Dual coordinate descent algorithms for efficient large margin structural learning. Transactions of the Association for Computational Linguistics (2013)
5. Chu, C., Kim, S.K., Lin, Y., Yu, Y., Bradski, G., Ng, A.Y., Olukotun, K.: Map-reduce for machine learning on multicore. NIPS **19** (2007) 281
6. Collins, M.: Discriminative training methods for hidden Markov models: Theory and experiments with perceptron algorithms. In: EMNLP. (2002)
7. Collins, M., Roark, B.: Incremental parsing with the perceptron algorithm. In: ACL. (2004)
8. Finley, T., Joachims, T.: Training structural SVMs when exact inference is intractable. In: ICML. (2008) 304–311
9. Gurobi Optimization, Inc.: Gurobi optimizer reference manual (2012)

10. Hsieh, C.J., Chang, K.W., Lin, C.J., Keerthi, S.S., Sundararajan, S.: A dual coordinate descent method for large-scale linear SVM. In: ICML. (2008)
11. Joachims, T.: Making large-scale SVM learning practical. In Schölkopf, B., Burges, C., Smola, A., eds.: Advances in Kernel Methods - Support Vector Learning. (1999)
12. Joachims, T., Finley, T., Yu, C.N.: Cutting-plane training of structural svms. Machine Learning (2009)
13. Lacoste-Julien, S., Jaggi, M., Mark Schmidt, P.P.: Block-coordinate Frank-Wolfe optimization for structural SVMs. In: ICML. (2013)
14. Langford, J., Smola, A.J., Zinkevich, M.: Slow learners are fast. In: NIPS. (2009) 2331–2339
15. Luo, Z.Q., Tseng, P.: Error bounds and convergence analysis of feasible descent methods: a general approach. Annals of Operations Research **46** (1993) 157–178
16. Marcus, M.P., Santorini, B., Marcinkiewicz, M.A.: Building a large annotated corpus of english: The penn treebank. Computational Linguistics
17. McDonald, R., Hall, K., Mann, G.: Distributed training strategies for the structured Perceptron. In: Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics, Los Angeles, California, Association for Computational Linguistics (June 2010) 456–464
18. McDonald, R.T., Hall, K., Mann, G.: Distributed training strategies for the structured perceptron. In: HLT-NAACL. (2010) 456–464
19. Meshi, O., Sontag, D., Jaakkola, T., Globerson, A.: Learning efficiently with approximate inference via dual losses. In: ICML. (2010)
20. Roth, D., Yih, W.: A linear programming formulation for global inference in natural language tasks. In Ng, H.T., Riloff, E., eds.: CoNLL. (2004)
21. Roth, D., Yih, W.: Global inference for entity and relation identification via a linear programming formulation. In Getoor, L., Taskar, B., eds.: Introduction to Statistical Relational Learning. (2007)
22. Taskar, B., Chatalbashev, V., Koller, D., Guestrin, C.: Learning structured prediction models: a large margin approach. In: ICML. (2005)
23. Tsochantaridis, I., Joachims, T., Hofmann, T., Altun, Y.: Large margin methods for structured and interdependent output variables. Journal of Machine Learning Research (2005)
24. Wang, P.W., Lin, C.J.: Iteration complexity of feasible descent methods for convex optimization. Technical Report, National Taiwan University (2013)