Vol.2014-NL-217 No.8
2014/7/4

# Word2vecの並列実行時の学習速度の改善

岡崎 直観[1,2,a)]　乾 健太郎[1,3,b)]

**概要**：単語の意味ベクトルを大規模コーパスから学習するためのツールとして，Mikolov らの手法 [14] を実装した word2vec が注目を浴びている．本論文は，word2vec を複数のプロセッサで並列で動作させた時に学習速度が低下する原因を説明し，これを改善するアルゴリズムを提案する．提案手法は学習で得られる単語ベクトルの質を落とすこと無く，複数のプロセッサを効率よく利用できることを実験的に示す．

## 1. はじめに

大規模なテキストコーパスから単語のベクトル表現を学習する研究が盛んに行われている [5], [6], [14], [15], [16], [17]．特に，Mikolov ら [14] の手法を実装した word2vec[*1] と呼ばれるツールが，自然言語処理の研究コミュニティで急速に注目を集めている [1], [2], [9], [10], [19], [20]．このツールの人気の理由はいくつか考えられる．例えば，オープンソースとしてコードが公開されていること，大規模なデータに対してもスケールすること，確立された理論に基づくオンライン学習（確率的勾配降下法や誤差逆伝搬法），単語ベクトルの加算による単語の意味の構成性などが挙げられる．さらに，この実装は複数のプロセッサを利用した並列処理に対応しており，大量のテキストコーパスを高速に処理することで，単語ベクトルの質の向上させることができる．

word2vec は，複数のプロセッサから共有されるメモリ上で，**データ並列処理**に基づく**非同期**パラメータ更新 [4] を採用している．この方式では，学習データをいくつかのサブセットに分割し，各サブセットを複数のプロセッサ（またはコア）が並列に処理できるように，複数のスレッドを作成する．分類問題に対する確率的勾配降下法と同様に，それぞれのスレッドは担当しているサブセットから学習インスタンスを取り出し，現在のパラメータ（単語ベクトル）に対する勾配を計算し，その勾配に基づいてパラメータを更新する．並列化されたオンライン学習では，パラメータをスレッド内で個別に保持し，適当なタイミングで全スレッドのパラメータの値を平均化することが多い [11], [12], [21]．しかし，非同期方式では複数のスレッドから共有されているメモリ上にパラメータを保持し，各スレッドはパラメータの値を読み書きする際に，他のスレッドがそのパラメータを更新しようとしているかどうかに関わらず，パラメータ更新を行う．非同期パラメータ更新では，あるプロセッサによる更新内容が他のプロセッサの上書きにより失われる可能性があるため，正確にパラメータが更新される保証がない．一方，面倒で処理時間を浪費するスレッド間の同期を考慮しなくてよいので，非同期パラメータ更新はシンプルかつ高速に行える．実用上は，非同期パラメータ更新によるエラーは無視できるほど小さいため，この方式で学習しても実用上問題ない（5 節の表 1 参照）．

非同期パラメータ更新は合理的に思えるが，word2vec を複数のスレッドで動作させると，想定を大きく下回る速度

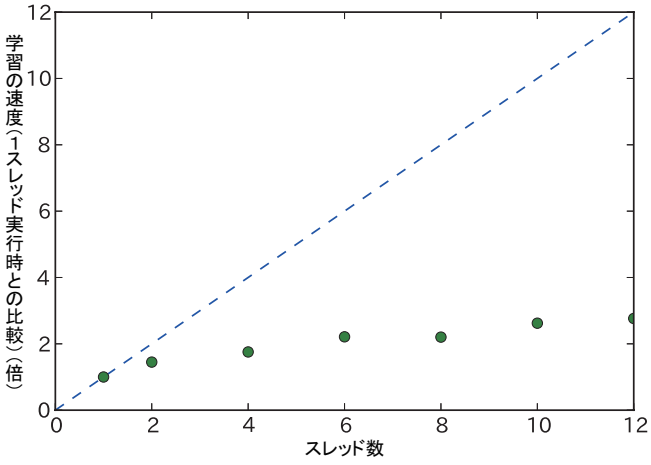


**図 1**　オリジナルの word2vec は複数のプロセッサを用いても速くならない（階層的ソフトマックス使用時）．実験設定は 5 節参照．

1　東北大学大学院情報科学研究科
Graduate School of Information Sciences, Tohoku University
2　科学技術振興機構・戦略的創造研究推進事業「さきがけ」
PRESTO, Japan Sciencce and Technology Agency (JST)
3　科学技術振興機構・戦略的創造研究推進事業「CREST」
CREST, Japan Sciencce and Technology Agency (JST)
a)　okazaki(at)ecei.tohoku.ac.jp
b)　inui(at)ecei.tohoku.ac.jp
*1　https://code.google.com/p/word2vec/

しか出ない．図 1 に ukWaC [3] をテキストコーパスとして，複数のスレッドで単語ベクトルを学習させたときの速度を示す．複数のプロセッサを用いても学習が速くならず [*2]，我々の期待（破線）との乖離が大きい．本稿では，複数プロセッサを用いた時に word2vec の学習が速くならない理由を明らかにし，その問題を解消するためのアルゴリズムを提案する．提案したアルゴリズムを用いると，学習後に得られる単語ベクトルの質を落とすことなく，複数のプロセッサを効率よく利用できることを実験的に示す．本研究の実装は，最新の word2vec のソースコードに対するパッチとして公開している [*3]．

## 2. Skip-gram モデル

なぜ word2vec は複数のプロセッサを効率よく活用することができないのか？ この理由を考察するため，本節では **Skip-gram モデル** [14] を簡単に復習する．学習データ中の 1 文（単語列 $w_1, ..., w_T$ から構成されるとする）が与えられると，Skip-gram モデルは以下の対数尤度を最大化するような単語ベクトルを求めようとする．

$$\sum_{t=1}^{T} \sum_{-c \le j \le c, j \ne 0} \log p(w_{t+j}|w_t) \qquad (1)$$

ここで，$c$ は中心の単語 $w_t$ に対して，どのくらいの距離の単語までを文脈とするかを設定する定数である．word2vec は確率的勾配降下法によるオンライン学習で対数尤度を最大化していく．

確率 $p(w_{t+j}|w_t)$ は単語 $w_t$ から周辺の単語 $w_{t+j}$ を予測するもので，対数双線形モデル（log-bilinear model） [18] として定式化される．対数双線型モデルの正確な計算は困難であるため，Mikolov ら [14] は**階層的ソフトマックス（hierarchical softmax）**と**負例サンプリング（negative sampling）**という 2 つの手法を提案している．これらの手法では，確率 $p(w_{t+j}|w_t)$ を次式で近似する．

$$p(w_{t+j}|w_t) = \prod_{(x_i, y_i) \in \mathrm{Gen}(w_{t+j})} \sigma(y_i \boldsymbol{u}_{x_i}^\top \boldsymbol{v}_{w_t}). \qquad (2)$$

ここで，$\mathrm{Gen}(w_{t+j})$ は文脈単語 $w_{t+j}$ に対して（擬似的な）サンプル $x_i$ とそのラベル $y_i \in \{+1, -1\}$ のタプル $(x_i, y_i)$ のリストを返す関数，$\boldsymbol{v}_{w_t}$ は単語 $w_t$ のベクトル表現，$\boldsymbol{u}_{x_i}$ はサンプル $x_i$ に対する隠れベクトル，$\sigma(a)$ はシグモイド関数である．

階層的ソフトマックスでは，$\mathrm{Gen}(w)$ は学習データ中の単語の出現分布から作られたハフマン木上のノード番号 $\{x_1, ..., x_{d_w - 1}\}$ と，そのラベルを返す関数として設計される．ここで，$x_i$ はハフマン木の根から文脈単語 $w$（ハフマン木では葉）に至るパス上で，深さ $i$ にあるノードの ID 番号 [*4] である．ラベル $y_i$ は $x_{i+1}$ がノード $x_i$ から見て右側の子供であれば $+1$，そうでなければ $-1$ と定義する．$n$ 個のサンプルを用いた負例サンプリングでは，関数 $\mathrm{Gen}(w)$ は単語 $\{x_1, ..., x_{n+1}\}$ を返す．ここで，$x_1 = w$ とし，$x_2, ..., x_{n+1}$ は学習データ中に出現する単語のユニグラム分布にもとづいてランダムにサンプリングした単語とする．ラベル $y_1$ は $+1$（正例）とし，$y_2, ..., y_{n+1}$ はすべて $-1$（負例）とする．

## 3. キャッシュの偽共有が速度低下をもたらす

図 2 に 4 つのプロセッサが階層的ソフトマックスを用いて単語ベクトルを学習する様子を示した．単語 $w_t$ のベクトルを更新するため，各プロセッサ#1, ..., #4 はそれぞれ，*Paris*，*Lufthansa*，*KLM*，*London* という文脈単語 $(w_{t+j})$ に着目している．プロセッサ#1 が文脈単語 *Paris* に関してパラメータ更新を行うため，ハフマン木で祖先にあたるノード①，②，④，⑧，⑰，…に対応する隠れベクトル $\boldsymbol{u}_1, \boldsymbol{u}_2, \boldsymbol{u}_4, \boldsymbol{u}_8, \boldsymbol{u}_{17}, ...$（青色で示す）を更新したばかりという状況を考える．共有メモリにパラメータを格納した非同期パラメータ更新では，プロセッサ#1 は隠れベクトル $\boldsymbol{u}_{x_i}$ をロックしないため，別のプロセッサは $\boldsymbol{u}_1$ や $\boldsymbol{u}_2$ 等にいつでもアクセス（読み書き）できる．

この非同期パラメータ更新は（精密な学習を犠牲にして）効率が良いように見えるが．プロセッサ間で深刻な**偽共有（false sharing）**を引き起こす．偽共有 [8][*5] とは，あるプロセッサの**キャッシュライン** [*6] が別のプロセッサがそのキャッシュライン上のメモリの内容を更新することにより，無効化されてしまう状況のことを指す．図 2 では，プロセッサ#1 のパラメータ更新により，隠れベクトル $\boldsymbol{u}_1$ と $\boldsymbol{u}_2$ に対するプロセッサ#2, #3, #4 のキャッシュラインが無効化されている．キャッシュの一貫性を保つため，プロセッサ#2, #3, #4 はキャッシュの更新を強制されることになり，深刻なパフォーマンス低下をもたらす．

階層的ソフトマックスで用いるハフマン木では，ほとんどの葉ノードがごく少数の根に近いノードに繋がっている．これらのノードにおいて複数のプロセッサが頻繁なパラメータ更新を行うため，深刻な偽共有が発生する．同様の状況は，少数の頻出語が繰り返しサンプルされやすい負例サンプリングでも発生すると考えられる．

この問題を解決するための単純な方策は，隠れパラメータベクトル $\boldsymbol{u}$ を各スレッドで個別に管理し，プロセッサ間でキャッシュラインの干渉が起こらないようにすることである．しかしながら，この方策はスレッド数の増加に比例してメモリを消費してしまうので，大規模なデータに対し

[*2] 各プロセッサはほぼ 100%の負荷で動作しており，学習データに関する I/O 等の影響は小さいことが確認できている．
[*3] `http://www.chokkan.org/software/word2vec-multi`
[*4] $x_1$ は根のノード，$x_{d_w}$ は文脈単語 $w$ に対応する葉，$d_w$ はその葉ノードの深さを表す．
[*5] `http://www.intel.com/software/threading-guide`
[*6] CPU にキャッシュされるメモリの単位で，例えば 64 バイトの連続したメモリブロックがキャッシュされる．

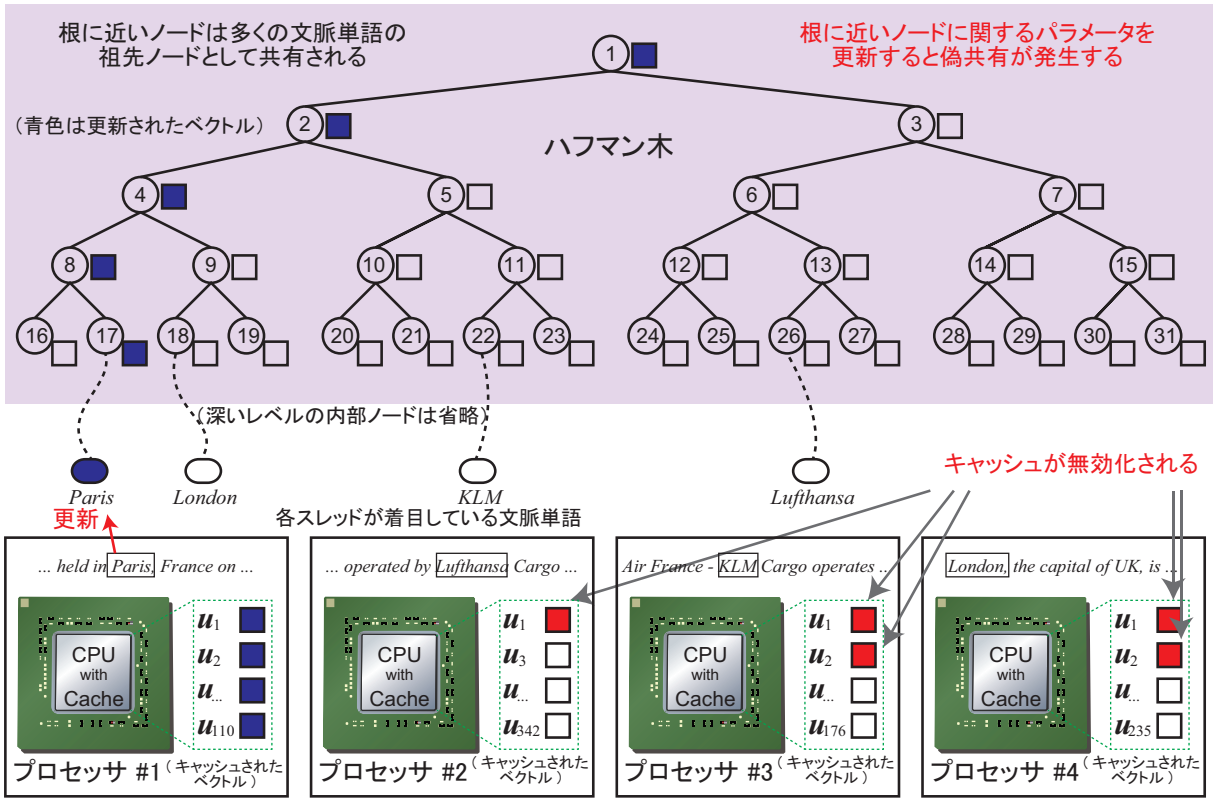


図 2　階層的ソフトマックスによって引き起こされるキャッシュの偽共有

てスケールしない．さらに，Skip-gram モデルで得られる単語ベクトルの意味的な性質（例えば，ある次元が特定の意味を強く示唆するなど）は，学習の過程で得られるものである [*7] ため，Iterative Parameter Mixture [12] のように複数のスレッドで学習したパラメータを平均化することは非合理的である．

## 4. 提案手法

偽共有は複数のスレッドから頻繁にアクセスされるベクトルを更新する際に発生する．偽共有を防ぐには，頻繁にアクセスされるベクトルを各スレッドで個別に管理しつつ，複数のスレッド間でベクトルの値の同期をとり，各スレッドが学習している単語ベクトルの一貫性を保つことが理想的である．本研究では，これを以下のように実装する．

まず，式 2 で使われている隠れ意味ベクトル $\boldsymbol{u}_j$（$j \in \{1,...,m\}$）のインデックス番号を，$\boldsymbol{u}_1$ が最も頻繁に更新されるベクトル，$\boldsymbol{u}_m$ が最も更新されないベクトルになるように並び替える．ここで，$m$ は隠れベクトルの数を表す [*8]．あるパラメータ（定数）$k \ll m$ を設定し，各スレッド毎に 2 種類のベクトル群 $\widetilde{\boldsymbol{u}_1},...,\widetilde{\boldsymbol{u}_k}$ と $\widetilde{\boldsymbol{\delta}_1},...,\widetilde{\boldsymbol{\delta}_k}$ を確保する．これらのベクトルは次のように初期化する: すべての $j \in \{1,...,k\}$ に対し，$\widetilde{\boldsymbol{u}_j} \leftarrow \boldsymbol{u}_j$; $\widetilde{\boldsymbol{\delta}_j} \leftarrow 0$．式 2 の計算などで隠れベクトル $\boldsymbol{u}_j$ の値が必要な場合は，$j > k$ ならば共有メモリ上のベクトル $\boldsymbol{u}_j$ から読み出し，$j \leq k$ ならば各スレッド毎に管理している $\widetilde{\boldsymbol{u}_j}$ から読み出す．

あるスレッドがある学習インスタンスに関する勾配を計算し，ベクトル $\boldsymbol{u}_j$ に必要な更新量 $\boldsymbol{g}$ を計算したとする．もし $j > k$ なら，そのスレッドは共有メモリ上のベクトル $\boldsymbol{u}_j$ の値を更新する（通常の共有メモリ上の非同期パラメータ更新）: $\boldsymbol{u}_j \leftarrow \boldsymbol{u}_j + \boldsymbol{g}$．もし $j \leq k$ なら，そのスレッドはローカルで管理しているベクトル $\widetilde{\boldsymbol{u}_j}$ の値を更新する: $\widetilde{\boldsymbol{u}_j} \leftarrow \widetilde{\boldsymbol{u}_j} + \boldsymbol{g}$; $\widetilde{\boldsymbol{\delta}_j} \leftarrow \widetilde{\boldsymbol{\delta}_j} + \boldsymbol{g}$．つまり，学習時に頻繁にアクセスされる上位 $k$ 個のベクトル $\boldsymbol{u}_1,...,\boldsymbol{u}_k$ に対して，スレッド毎にコピー $\widetilde{\boldsymbol{u}_1},...,\widetilde{\boldsymbol{u}_k}$ を用意しておくことで，偽共有を防止している．

各スレッドは $t$ 個の学習インスタンス（単語）を処理する毎に（$t$ は定数パラメータ），スレッド毎に管理しているベクトルのコピーと共有メモリ上のベクトルとの間で同期をとる: すべての $j \in \{1,...,k\}$ に対し，$\boldsymbol{u}_j \leftarrow \boldsymbol{u}_j + \widetilde{\boldsymbol{\delta}_j}$; $\widetilde{\boldsymbol{u}_j} \leftarrow \boldsymbol{u}_j$; $\widetilde{\boldsymbol{\delta}_j} \leftarrow 0$．本研究では，この同期処理の間も共有メモリ上のベクトル $\boldsymbol{u}_j$ をロックせず，非同期パラメータ更新とする [*9]．

ところで，隠れベクトルをアクセス頻度に応じてソート

*7 重みベクトルの意味が人間が設計した素性によって定義されている教師あり学習において，パラメータの平均化が使える状況とは大きく異なる．

*8 隠れベクトルの数 $m$ は，学習コーパスに含まれる単語の種類数を $|V|$ とすると，階層的ソフトマックスの場合は $|V|-1$，負例サンプリングの場合は $|V|$ である．

*9 頻繁にアクセスされるベクトルはスレッド内のコピーで管理されるようになったため，この処理においてキャッシュの偽共有が発生する可能性は低くなっている．

表 1　アナロジータスクにおける各モデルの正解率．この表では，各手法でスレッドの数を 1, 2, 4, 6, 8, 10, 12 として学習した時のモデルの正解率の平均と分散を示す．

| 手法 | 平均 | 分散 |
|---|---|---|
| オリジナル (HS) | 0.521 | 0.00418 |
| 提案手法 (HS;100) | 0.520 | 0.00276 |
| 提案手法 (HS;1000) | 0.521 | 0.00443 |
| オリジナル (NEG-5) | 0.549 | 0.00244 |
| 提案手法 (NEG-5;100) | 0.551 | 0.00322 |

することは非常に簡単である．階層的ソフトマックスでは，階層木の根から葉に向かって，幅優先で番号を割り当てていけばよく，小さいインデックス番号を根に，大きいインデックス番号を葉に割り当てればよい．負例サンプリングでは，学習データ中の頻度に応じて単語をソートし，頻出語（＝よくサンプルされる語）に対して小さいインデックス番号を割り当てればよい．

## 5. 実験

提案手法の効果を調べるために，ukWaC コーパス [3] から 300 次元の Skip-gram モデル（文脈のウィンドウ幅は 5，$10^{-5}$ のサブサンプリングを用いた）を学習する実験を行った．最新版 *10 の word2vec は学習データ中に含まれる 2,324,509,927 個の単語の中から，1,943,050 種類の単語を認識した．提案手法は，word2vec のソースコードに修正を加えて実装した．すべての実験は，6 コアの AMD Opteron 2435 (2.6GHz) を 2 基，64GB の主記憶を積んだ計算サーバ上で行った．このプロセッサには，6 x 64kB の L1 キャッシュ，6 x 512kB の L2 キャッシュ，6MB の L3 キャッシュ（コア間で共有される）が搭載されている．隠れベクトル $\boldsymbol{u}$ による偽共有を防ぐため，スレッド内でローカルに管理するベクトルの数は，$k = 512$ [kB]/300 [dim]/8 [B/double] $\approx 218$ と設定した．

表 1 に，各実験設定で学習した単語ベクトルの性能を，アナロジー・タスク [13] で評価した結果を載せた．HS は階層的ソフトマックス，NEG-5 は負例サンプリング（1 インスタンスあたり 5 個の負例を用いる），HS/NEG-5 に続く数字はスレッドと共有メモリ間で同期をとる間隔 $t$ の設定値を表す．Mikolov ら [14] の報告の通り，負例サンプリングの方が階層的ソフトマックスよりも若干よい性能を示している．また，分散の値が示すように，スレッドの数を変化させても性能はほとんど変わらなかったことから，非同期パラメータ更新は単語ベクトルの質にほぼ影響しないことが分かった．提案手法では，隠れベクトル $\boldsymbol{u}$ の更新においてスレッド間で情報を交換する頻度を減らしているが，実験結果から学習で得られる単語ベクトルの質に影響を及ぼさないことが分かった．

*10 2014 年 5 月 26 日現在．

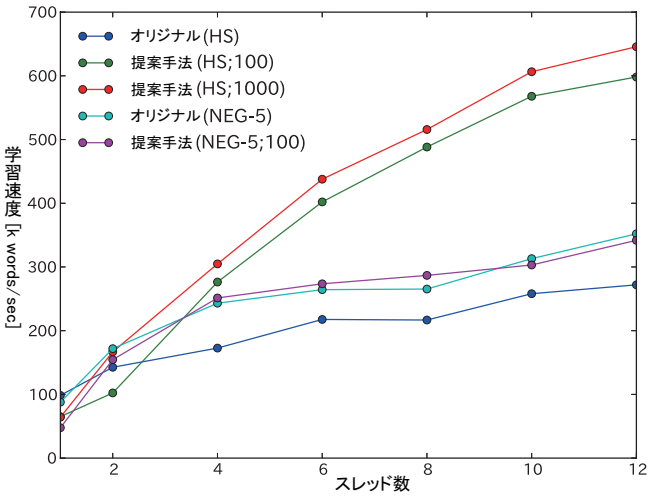


図 3　オリジナルの word2vec と提案手法の学習速度

図 3 に学習のスピード（単位時間あたりに処理できる学習インスタンスの数）を示した．階層的ソフトマックスでスレッド間の同期頻度 $t$ を 1,000 としたとき（提案手法 HS; 1000）の学習が最も速く，12 スレッド利用時は 1 スレッドの時よりも 6.6 倍速くなっている．元々の word2vec の実装では 12 スレッドを利用しても 2.8 倍しか速くなっていなかったことを考えると，提案手法の効果とともに，元々の実装では偽共有によるパフォーマンス低下があったことが伺える．より頻繁に同期を行うようにすると（提案手法 HS; 100），学習速度はやや遅くなるが，それでも元々の実装を上回る速度で学習が行えている．残念ながら，提案手法は負例サンプリングに対しては，学習速度を改善することができなかった．これは，1 インスタンスのパラメータ更新に 6 個のサンプルしか用いておらず，擬似サンプルの数が階層的ソフトマックスよりも少ないこと，負例サンプリングでは隠れベクトルによる偽共有がそれほど起こっていないことが理由として考えられる．

## 6. 結論

本研究では，階層的ソフトマックスを用いた word2vec の学習がマルチスレッドで実行した時に遅くなる主要因として，偽共有が考えられることを説明し，これを実験により確認した．大規模テキストコーパスを用いた実験により，提案手法は単語ベクトルの質を落とすことなく，複数のプロセッサの利用効率を高めることが示された．

本研究で得られた成果は様々な局面で活用できる．本研究のアイディアは，word2vec を拡張するような研究 [9] にも適用可能である．将来的に，ハフマン木ではなく単語の意味を反映したような木を用い，階層的ソフトマックスで学習される単語ベクトルの質を改善するような研究が出てきた場合にも，本研究の成果を利用できる．また，非同期パラメータ更新を行う他の学習アルゴリズム，例えば条件付き確率場 [7] での実験を進めたいと考えている．

**謝辞** 本研究は，JST 戦略的創造研究推進事業「さきがけ」，および JST 戦略的創造研究推進事業「CREST」から部分的な支援を受けて行われた．

## 参考文献

[1] Andreas, J. and Klein, D.: How much do word embeddings encode about syntax?, *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (ACL-2014)*, p. (to appear) (2014).

[2] Bansal, M., Gimpel, K. and Livescu, K.: Tailoring Continuous Word Representations for Dependency Parsing, *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (ACL-2014)*, p. (to appear) (2014).

[3] Baroni, M., Bernardini, S., Ferraresi, A. and Zanchetta, E.: The WaCky wide web: a collection of very large linguistically processed web-crawled corpora, *Language Resources and Evaluation*, Vol. 43, No. 3, pp. 209–226 (2009).

[4] Bengio, Y., Ducharme, R., Vincent, P. and Janvin, C.: A Neural Probabilistic Language Model, *Journal of Machine Learning Research*, Vol. 3, pp. 1137–1155 (2003).

[5] Bullinaria, J. A. and Levy, J. P.: Extracting semantic representations from word co-occurrence statistics: A computational study, *Behavior Research Methods*, Vol. 39, No. 3, pp. 510–526 (2007).

[6] Collobert, R. and Weston, J.: A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning, *Proceedings of the 25th International Conference on Machine Learning (ICML '08)*, pp. 160–167 (2008).

[7] Gimpel, K., Das, D. and Smith, N. A.: Distributed Asynchronous Online Learning for Natural Language Processing, *Proceedings of the Fourteenth Conference on Computational Natural Language Learning (CoNLL 2010)*, pp. 213–222 (2010).

[8] Intel Corporation: *Intel Guide for Developing Multithreaded Applications* (2011).

[9] Levy, O. and Goldberg, Y.: Dependency-Based Word Embeddings, *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (ACL-2014)*, p. (to appear) (2014).

[10] Levy, O. and Goldberg, Y.: Linguistic Regularities in Sparse and Explicit Word Representations, *Proceedings of the Eighteenth Conference on Computational Natural Language Learning (CoNLL-2014)*, p. (to appear) (2014).

[11] McDonald, R., Hall, K. and Mann, G.: Distributed Training Strategies for the Structured Perceptron, *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pp. 456–464 (2010).

[12] McDonald, R., Mohri, M., Silberman, N., Walker, D. and Mann, G. S.: Efficient Large-Scale Distributed Training of Conditional Maximum Entropy Models, *Advances in Neural Information Processing Systems 22 (NIPS 2009)*, pp. 1231–1239 (2009).

[13] Mikolov, T., Chen, K., Corrado, G. and Dean, J.: Efficient estimation of word representations in vector space, *Proceedings of the Workshop at International Conference on Learning Representations (ICLR 2013)* (2013).

[14] Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S. and Dean, J.: Distributed Representations of Words and Phrases and their Compositionality, *Advances in Neural Information Processing Systems 26 (NIPS 2013)*, pp. 3111–3119 (2013).

[15] Mitchell, J. and Lapata, M.: Vector-based Models of Semantic Composition, *Proceedings of ACL-08: HLT*, pp. 236–244 (2008).

[16] Mitchell, J. and Lapata, M.: Composition in distributional models of semantics, *Cognitive Science*, Vol. 34, No. 8, pp. 1388–1429 (2010).

[17] Mnih, A. and Kavukcuoglu, K.: Learning word embeddings efficiently with noise-contrastive estimation, *Advances in Neural Information Processing Systems 26 (NIPS 2013)*, pp. 2265–2273 (2013).

[18] Mnih, A. and Teh, Y. W.: A fast and simple algorithm for training neural probabilistic language models, *Proceedings of the 29th International Conference on Machine Learning*, pp. 1751–1758 (2012).

[19] Tian, R., Miyao, Y. and Matsuzaki, T.: Logical Inference on Dependency-based Compositional Semantics, *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (ACL-2014)*, p. (to appear) (2014).

[20] Yu, M. and Dredze, M.: Improving Lexical Embeddings with Semantic Knowledge, *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (ACL-2014)*, p. (to appear) (2014).

[21] Zhao, K. and Huang, L.: Minibatch and Parallelization for Online Large Margin Structured Learning, *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 370–379 (2013).