Northern European Journal of Language Technology, 2018, Vol. 5, Article 1, pp 1–15
DOI 10.3384/nejlt.2000-1533.1851

Part of Speech Tagging: Shallow or Deep Learning?

Robert Östling

Stockholm University
Department of Linguistics
robert@ling.su.se

Abstract

Deep neural networks have advanced the state of the art in numerous fields, but they generally suffer from low computational efficiency and the level of improvement compared to more efficient machine learning models is not always significant. We perform a thorough PoS tagging evaluation on the Universal Dependencies treebanks, pitting a state-of-the-art neural network approach against UDPipe and our sparse structured perceptron-based tagger, efselab. In terms of computational efficiency, efselab is three orders of magnitude faster than the neural network model, while being more accurate than either of the other systems on 47 of 65 treebanks.

1 Introduction

There is an ongoing revolution in machine learning, brought on by recent advances in deep neural networks. This has led to breakthroughs in a variety of fields, from image recognition (He et al., 2016) and generation (Goodfellow et al., 2014), through games like the strategic board game Go (Silver et al., 2016) and the first-person shooter Doom (Lample and Chaplot, 2017), to long-standing tasks in natural language processing such as machine translation (Bahdanau et al., 2014; Wu et al., 2016) and parsing (Chen and Manning, 2014).

Neural networks have been applied to nearly every area of natural language processing, as a quick glance at the proceedings of any recent conference in the field will reveal. The level of success varies, however. For sequence processing tasks with long-range dependencies, recurrent neural network models have lead to impressive improvements. Other areas, including sequence labeling tasks with local dependencies, such as part-of-speech (PoS) tagging and Named Entity Recognition (NER), have not seen the overwhelming improvements observed in other fields. Since neural network methods typically suffer from low computational efficiency, the central question asked in this work is this: *are neural networks methods worth the cost for sequence labeling tasks with mainly local dependencies?*

To investigate this, we compare bilty, a state-of-the-art system based on recurrent neural networks (Plank et al., 2016) to two systems based on the structured perceptron (Collins, 2002): the popular implementation in UDPipe/MorphoDiTa (Straka et al.,

Northern European Journal of Language Technology, 2018, Vol. 5, Article 1, pp 1–15
DOI 10.3384/nejlt.2000-1533.1851

2016; Straková et al., 2014) and our EFSELAB system. The primary goal of EFSELAB is computational efficiency, and in this sense it represents the opposite accuracy/efficiency tradeoff compared to BILTY.

The core of EFSELAB is a compiler from feature templates to a corresponding sequence labeling computer program, using the automatic feature hash refactoring procedure described in Section 2.3 to improve computational efficiency. The source code is available as Free Software, along with the necessary scripts to reproduce the experiments in this article.[1] Besides the generic model used for the evaluations presented here, there is also a Swedish annotation pipeline with additional resources, which combines EFSELAB and MaltParser (Nivre et al., 2007), and is able to process raw Swedish text into a full Universal Dependencies (version 2) analysis.[2] In this pipeline, EFSELAB is performing both of the sequence labeling tasks (PoS tagging and Named Entity Recognition).

2 Efficient Sequence Labeling: Methods

Sequence labeling is the task of finding a sequence of labels y_i given a corresponding sequence of inputs x_i. This has been used in natural language processing for a wide range of tasks, including PoS tagging, Named Entity Recognition and shallow parsing. This section describes our implementation of an efficient sequence labeling system, EFSELAB. As mentioned above, its primary goal is computational efficiency, which is reflected by our choice of using:

- Sparse features from a manually specified template.
- A simple linear classifier (structured perceptron).
- Feature hashing with automatic refactoring.
- Feature extraction compiled to native code.

Several of these decisions can be modified in order to gain accuracy at the cost of more computation, but in these cases we have generally decided in favor of computational efficiency.

2.1 Sparse Features with the Structured Perceptron

Feature-rich models for sequence labeling have been popular for the last couple of decades. They are based on defining a vector-valued feature function $\vec{\phi}(x, y, i)$, which computes a sparse vector representation of the sequence x and label sequence y at sequence position i. For instance, we might have the kth element of such a function defined as:

$$\phi_k(x, y, i) = \left\{ \begin{array}{ll} 1 & \text{if } x_i = \text{cat} \wedge y_{i-1} = \text{DET} \wedge y_i = \text{NOUN} \\ 0 & \text{otherwise} \end{array} \right\}$$

[1] `https://github.com/robertostling/efselab`

[2] The Swedish annotation pipeline is joint work with Aaron Smith, Jesper Näsman, Joakim Nivre, Filip Salomonsson and Emil Stenström.

Northern European Journal of Language Technology, 2018, Vol. 5, Article 1, pp 1–15
DOI 10.3384/nejlt.2000-1533.1851

For convenience, we write the sum of a feature function over a sequence as

$$\vec{\phi}(x, y) = \sum_i \vec{\phi}(x, y, i)$$

The task of the classifier is to find a weight vector $\vec{w}$ such that $\vec{\phi}(x, y)^T \cdot \vec{w}$ is high when y is a correct labeling of x, and low otherwise.

Collins (2002) demonstrated that the perceptron algorithm (Rosenblatt, 1957) can yield excellent results for sequence labeling tasks, by applying a search algorithm to whole sequences. Later work further demonstrated the strength of this method, for instance by refining the search method (Shen et al., 2007) or the weight updating scheme (Daumé and Marcu, 2005).

Algorithm 1 Structured Perceptron Training.

▷ *Start at time $t = 0$ with a zero weight vector*
$\vec{w}^{(0)} \leftarrow \vec{0}$
$t \leftarrow 0$
repeat
  ▷ *Repeatedly iterate over the training data D*
  for all $x, y \in D$ **do**
    $t \leftarrow t + 1$
    ▷ *Estimate the labels $\hat{y}$ using old weights*
    $\hat{y} \leftarrow \arg\max_{y'} \vec{\phi}(x, y') \cdot \vec{w}^{(t)}$
    ▷ *Update the weights*
    $\vec{w}^{(t+1)} \leftarrow \vec{w}^{(t)} + \vec{\phi}(x, y) - \vec{\phi}(x, \hat{y})$
  end for
until convergence
▷ *Return the mean weight vector over time*
return $\frac{1}{t} \sum_{i=0}^{t} \vec{w}^{(i)}$

Algorithm 1 describes the structured perceptron learning algorithm. The arg max operation can be performed by any search algorithm, either exactly with e.g. Viterbi algorithm (Viterbi, 1967), or using an approximate method such as beam search. We use beam search, since it offers an easy tradeoff between speed and accuracy by varying the beam size. An important point here is that the *mean* weight vector during training is used, which serves as a way of reducing bias towards recent observations and empirically leads to higher classification accuracy (Collins, 2002).

2.2 Feature Hashing

The value of the feature function $\vec{\phi}(x, y)$ typically has a high dimensionality but is very sparse, with only some dozen out of millions of dimensions being non-zero. This makes it computationally much more efficient to store only the few non-zero elements. Furthermore, given some hash function $h(\cdot)$ which maps features to indexes in a weight vector $\vec{u}$, we have

$$\vec{\phi}(x, y)^T \cdot \vec{w} \approx \sum_{k|\phi_k(x,y)\neq 0} u_{h(k)}$$

Northern European Journal of Language Technology, 2018, Vol. 5, Article 1, pp 1–15
DOI 10.3384/nejlt.2000-1533.1851

if all ϕ_k are binary-valued. The computational advantage is that this amounts to adding a small number of elements from known locations in the weight vector u. Note that u is indexed by the (arbitrary) range of the $h(\cdot)$ function, unlike w which is of the same dimensionality as $\vec{\phi}(x, y)$.

For a function $h(\cdot)$ which is collision-free over the feature set, the equivalence is exact. In practice we want u to be as small as possible to save computational resources. Decreasing the size of u makes collisions more common and the approximation error grows, but the method is often robust despite fairly high collision rates (Ganchev and Dredze, 2008).

The size of u can be empirically determined by evaluating model performance on held-out data, until a suitable balance between computational efficiency and learning ability is found. However, re-training the model every time is costly, and we have found that a simpler method works equally well. First, a conservative (high) value for the dimensionality of u is chosen and the model is trained. Then, we update $u \leftarrow u_{1\ldots N/2} + u_{N/2+1\ldots N}$ and use $h(\cdot) \mod N/2$ as the new hashing function. In other words, we break the vector u in two halves and add them, then wrap the hash function to cover the new u.

For word types that occur in the training data, we restrict the allowed tags to those occurring with the (lowercased) type in the training data. To avoid string comparisons and increase computational efficiency, we consider all strings with the same hash sum equivalent. This means that there will likely be a small number of mistaggings due to the tag lexicon, but in practice this is not a significant problem if the table size is sufficiently large. Additionally, errors tend to affect infrequent types due to Zipf's law, and therefore affect relatively few tokens.

2.3 Refactoring Feature Templates

A naive way to define $h(\cdot)$ would be to construct a string of characters representing the corresponding feature function $\phi_k(x, y, i)$, for instance "`suffix=ed,tag=VERB`", and then use any function for string hashing to map this into an integer. In most cases the feature functions ϕ_k are created from templates that generate a number of very similar functions. For instance, with $x_i = \textit{hinted}$ and $y_{i-1} = \text{NOUN}, y_i = \text{VERB}$ we might have non-zero feature functions with conditions such as these:

$$
\begin{aligned}
&\text{suffix1} = d \quad \wedge \text{tag} = \text{VERB}\\
&\text{suffix2} = ed \quad \wedge \text{tag} = \text{VERB}\\
&\text{suffix3} = ted \quad \wedge \text{tag} = \text{VERB}\\
&\text{form} = \textit{hinted} \wedge \text{tag} = \text{VERB}\\
&\text{suffix1} = d \quad \wedge \text{tag} = \text{VERB} \wedge \text{last-tag} = \text{NOUN}\\
&\text{suffix2} = ed \quad \wedge \text{tag} = \text{VERB} \wedge \text{last-tag} = \text{NOUN}\\
&\text{suffix3} = ted \quad \wedge \text{tag} = \text{VERB} \wedge \text{last-tag} = \text{NOUN}\\
&\text{form} = \textit{hinted} \wedge \text{tag} = \text{VERB} \wedge \text{last-tag} = \text{NOUN}\\
&\ldots
\end{aligned}
$$

A typical hash function over sequences (or trees) of integers works by recursively applying a mixing function $m(a, b)$ that maps integers a and b to another integer in a pseudo-

Northern European Journal of Language Technology, 2018, Vol. 5, Article 1, pp 1–15
DOI 10.3384/nejlt.2000-1533.1851

Table 1: Refactored feature hash computation.

Hash value	**Feature condition**	
$t_1 = d$		
$t_2 = m(e, t_1)$		
$t_3 = m(t, t_2)$		
$t_4 = m(n, t_3)$		
$t_5 = m(i, t_4)$		
$t_6 = m(h, t_5)$		
$t_7 = m(\text{suffix1}, t_1)$	suffix1 = *d*	
$t_8 = m(\text{suffix2}, t_2)$	suffix2 = *ed*	
$t_9 = m(\text{suffix3}, t_3)$	suffix3 = *ted*	
$t_{10} = m(\text{form}, t_5)$	form = *hinted*	
$t_{11} = m(\text{tag}, \text{VERB})$	tag = VERB	
$t_{12} = m(\text{last-tag}, \text{NOUN})$	last-tag = NOUN	
$t_{13} = m(t_{11}, t_{12})$		tag = VERB ∧ last-tag = NOUN
$t_{14} = m(t_7, t_{11})$	suffix1 = *d*	∧tag = VERB
$t_{15} = m(t_8, t_{11})$	suffix2 = *ed*	∧tag = VERB
$t_{16} = m(t_9, t_{11})$	suffix3 = *ted*	∧tag = VERB
$t_{17} = m(t_{10}, t_{11})$	form = *hinted*	∧tag = VERB
$t_{18} = m(t_7, t_{13})$	suffix1 = *d*	∧tag = VERB ∧ last-tag = NOUN
$t_{19} = m(t_8, t_{13})$	suffix2 = *ed*	∧tag = VERB ∧ last-tag = NOUN
$t_{20} = m(t_9, t_{13})$	suffix3 = *ted*	∧tag = VERB ∧ last-tag = NOUN
$t_{21} = m(t_{10}, t_{13})$	form = *hinted*	∧tag = VERB ∧ last-tag = NOUN

random manner.[3] For the second of the examples above, we might therefore compute $m(\text{suffix2}, m(e, m(d, m(\text{tag}, \text{VERB}))))$, assuming that suffix2, *e*, *d*, tag and VERB are all discrete symbols that can be represented by integers.

Given the redundancy among these feature functions, it is possible to reduce computation (applications of the mixing function) significantly by sharing subtrees between feature hashes. Naively computing the hashes of the features above would require 48 evaluations of the mixing function $m(\cdot)$, but by refactoring the computation graph and sharing intermediate values, it can be computed (as t_{14} through t_{21}) using 20 evaluations as shown in Table 1.

Beyond this refactoring, note that only t_{11} and later depend on the label y, which means that during search the values of t_1 through t_{10} are constant and do not need to be recomputed. In addition, only t_{18} through t_{21} depend on previous tag assignments (t_{i-1} or earlier). During beam search, scoring the first hypothesis requires evaluating $m(\cdot)$ 20 times to compute t_2 through t_{21}, but subsequent hypotheses need only to recompute six values: t_{12}, t_{13} and t_{18} through t_{21}. Assuming a beam size of four, these optimizations

[3]The choice of m is arbitrary, but we adapt it from MurmurHash: `https://github.com/aappleby/smhasher`

Northern European Journal of Language Technology, 2018, Vol. 5, Article 1, pp 1–15
DOI 10.3384/nejlt.2000-1533.1851

together reduce the number of operations from $4 \cdot 48 = 192$ when using the naive way, to $20 + 3 \cdot 6 = 38$ per token.

The EFSELAB system takes as input a set of feature templates and a tagset definition, automatically performs the refactoring described above, and produces code in the C programming language for a sequence labeling tool. This can then be compiled to efficient native code using a standard C compiler.

3 Experimental Setup

The task we evaluate in this work is PoS tagging with version 2 of the Universal Dependencies tagset. This allows us to compare tagging accuracy over many languages with an identical tagset, to maximize comparability across languages. The tagset consists of 17 PoS categories, listed in Table 2. These are intended to represent coarse, cross-linguistically common categories, and their specific delineation is (or should be) decided for each language. Universal Dependencies may have further language-specific categories as well as morphological features, which we do not use.

For this work, we limit ourselves to PoS tagging, even through EFSELAB can of course be applied to other sequence labeling tasks (our Swedish annotation pipeline also uses it for NER). This is because of data availability, with the Universal Dependency treebanks being a unique resource in terms of uniformly annotated text in a large number of languages.

3.1 Data

We use the Universal Dependencies treebanks, version 2.0 (Nivre et al., 2017). This consists of 70 treebanks in 50 languages. Of these we use 65 in 48 languages for our evaluations. Two are excluded (Kazakh and Uyghur) because no training data is available, only small development and test sets. The Italian ParTUT treebank lacks a test set, the Arabic NYUAD treebank requires access to non-free data, and finally the Russian SynTagRus treebank was excluded because the UDPIPE tool (Straka et al., 2016) was unable to process it during training. Since Arabic, Italian and Russian are represented by other Universal Dependencies treebanks, we are able to include them in our experiments.

Note that we use the standard training, development and test data split of the treebanks, with the test data that was released after the CoNLL 2017 shared task.[4]

3.2 Our System

While we have developed a Swedish annotation pipeline based on EFSELAB that is specifically tuned to Swedish (see Section 1), we use exactly the same settings and hyperparameters for all treebanks. The feature templates, described in Section 3.2.1, contain very generic features based on tag and word n-grams as well as affixes. A beam size of 4 is used during training and evaluation, as a compromise between tagging accuracy and computational efficiency. We use development set tagging accuracy as a criterion for

[4]The training and development data is available at `http://hdl.handle.net/11234/1-1983` while the test data is available at `http://hdl.handle.net/11234/1-2184`.

Northern European Journal of Language Technology, 2018, Vol. 5, Article 1, pp 1–15
DOI 10.3384/nejlt.2000-1533.1851

Table 2: The Universal PoS tagset.

Open word classes	
ADJ	Adjective
ADV	Adverb
INTJ	Interjection
NOUN	Noun
PROPN	Proper noun
VERB	Verb
Closed word classes	
ADP	Adposition
AUX	Auxiliary
CCONJ	Coordinating conjunction
DET	Determiner
NUM	Numeral
PART	Particle
PRON	Pronoun
SCONJ	Subordinating conjunction
Other	
PUNCT	Punctuation
SYM	Symbol
X	Other

Northern European Journal of Language Technology, 2018, Vol. 5, Article 1, pp 1–15
DOI 10.3384/nejlt.2000-1533.1851

Table 3: Feature templates for EFSELAB used in our evalutaion.

t_{i-1}, t_i	tag bigram
t_{i-2}, t_{i-1}, t_i	tag trigram
t_i, chartype(w_i)	tag and character types of current word
t_i, unique-chartype(w_i)	tag and unique character types of current word
$t_i, l(w_i)$	tag and current word (lowercased)
$t_i, l(w_{i-1})$	tag and previous word (lowercased)
$t_i, l(w_{i+1})$	tag and next word (lowercased)
$t_i, l(w_{i-1}), l(w_i)$	tag and previous word bigram (lowercased)
$t_i, l(w_i), l(w_{i+1})$	tag and word bigram (lowercased)
t_i, prefix$_n(l(w_i))$ for $n \in 1 \dots 5$	tag and (lowercased) prefix of length 1, 2, 3, 4, 5
t_i, suffix$_n(l(w_i))$ for $n \in 1 \dots 5$	tag and (lowercased) suffix of length 1, 2, 3, 4, 5

stopping training, and for the model compression technique described in Section 2.2 we allow at most a 1% increase in development set error rate compared to the uncompressed model.

To improve both efficiency and accuracy, we use a tag lexicon for word types that occur in the training data. Out-of-vocabulary words are assumed to be either adjectives, adverbs, interjections, (proper) nouns, verbs, symbols or belong to the miscellaneous category.

3.2.1 Feature Templates

The feature templates used in our evaluation are listed in Table 3. There are altogether 19 templates, which is also the number of non-zero elements in the feature vector. The types of features used are standard, essentially following Ratnaparkhi (1996). In order to better predict out-of-vocabulary items, we also use the functions *chartype* and *unique-chartype*. These normalize a token so that only the type of each character is preserved, e.g. upper-case Latin character, lower-case Greek character, symbol, numeral, and so on. The *unique-chartype* function furthermore reduces any sequence of one or more such character type to a single item, in order to also abstract away from token length.

One argument commonly heard in favor of neural network models is that they eliminate the need of manual feature engineering. This is true, but we note that engineering the features in Table 3 required very little effort, and very similar feature templates have been well-established for decades.

3.3 Baselines

To ensure a strong baseline, we use the BILTY tagger[5] of Plank et al. (2016), which has claimed state-of-the-art results on PoS tagging for the Universal Dependencies treebanks. They use bidirectional Long Short-Term Memory (LSTM) networks (Hochreiter and Schmidhuber, 1997) on two levels: first on the character level to encode characters into word representations, then on the word level to encode the sequence of word representations into PoS tag predictions. We trained models on the universal PoS tags from the

[5]We used the version in commit `b0cd84a` (published 2017-05-22) in our evaluation.

Northern European Journal of Language Technology, 2018, Vol. 5, Article 1, pp 1–15
DOI 10.3384/nejlt.2000-1533.1851

Table 4: Timing results.

	BILTY	OUR
Kilotokens per second per CPU core	0.66	503
Microseconds per token per CPU core	1509	2.0

Universal Dependencies treebanks, the same data used for EFSELAB. Following the recommendations of the authors, we use 64-dimensional word embeddings, 100-dimensional character embeddings, a 3-layer stacked bidirectional LSTM with 100 dimensions per layer, and 20 epochs of plain stochastic gradient descent for optimization. Both BILTY and EFSELAB are able to incorporate unsupervised learning from unannotated text, through pretrained embeddings and word clusters, respectively. Since this data is only available for a subset of the languages we decided to not use any data beyond the Universal Dependencies treebanks, but it should be noted that the performance of both systems can be somewhat improved by adding such data, if available.

For additional comparison with a popular tool on the same dataset, we also include UDPIPE (Straka et al., 2016) in the comparison. Its PoS tagger module uses the model of Straková et al. (2014), which is also based on a structured perceptron. We used the most recent version of UDPIPE[6], and trained on the full training set of the Universal Dependencies treebanks version 2.0, using hyperparameters from the CoNLL 2017 shared task baseline supplementary data package.[7]

4 Results

4.1 Computational Efficiency

In order to evaluate the computational efficiency of BILTY and EFSELAB, we let both systems tag the Czech[8] development set. Since UDPIPE by default also performs numerous additional tasks, such as morphological analysis and lemmatization, we do not include it in this comparison. While the choice of data is not critical, we chose the Universal Dependencies development set because it contains in-domain data with a realistic proportion of out-of-vocabulary items. In order to obtain reliable timing statistics, we concatenated multiple copies of the development set. All experiments were run on a single core of an Intel Xeon E5645 CPU clocked at 2.4 GHz. The result can be found in Table 4, which shows that EFSELAB is nearly three orders of magnitude faster.

Dense-feature neural networks (like BILTY) are notably easy to parallelize, so the comparison in Table 4 on a single CPU core favors sparse feature models (like EFSELAB). A well-optimized implementation of an LSTM tagger model similar to Plank et al. (2016)

[6]Commit `54b3027` (published 2017-05-18), available at `https://github.com/ufal/udpipe`.

[7]Available at `http://hdl.handle.net/11234/1-1990`. Note that this release also contains pretrained UDPIPE models, but since those are only trained on a subset of the training data (due to restrictions of the CoNLL 2017 shared task), we train our own models but use the same hyperparameters that were used for the shared task baselines.

[8]We use Czech because its development set is the largest among the Universal Dependencies treebanks. It contains 159,284 tokens, and ten concatenated versions (1,592,840 tokens) are used for timing BILTY while 100 copies (15,928,400 tokens) are required for EFSELAB.

Northern European Journal of Language Technology, 2018, Vol. 5, Article 1, pp 1–15
DOI 10.3384/nejlt.2000-1533.1851

running on a modern GPU could be expected to approach 10^{12} operations per second (Appleyard et al., 2016, Figure 1), roughly two orders of magnitude faster than the CPU core used in our experiments, although with considerably higher power consumption. While BILTY is not explicitly optimized for speed, it relies on the DyNet library (Neubig et al., 2017), which in our configuration uses the highly optimized Intel Math Kernel Library (MKL) for matrix operations. Thus we expect that the performance of BILTY in our experiments is close to the maximum achievable for their model on the given hardware. Since sentences are processed independently, all of the models discussed are trivial to parallelize over multiple CPU cores, which on a modern multi-core CPU means that the actual tagging speed is at least an order of magnitude above what is indicated by Table 4 for a single core.

4.2 Tagging Accuracy

In the beginning of this study, we set out to find out how high the price for increased computational efficiency is. Contrary to our expectations, it turns out that not only is there no such price, but that EFSELAB is actually the most accurate tagger for the vast majority of the treebanks: 48 out of 65, including one tie (for Arabic).

The full results are presented in Table 5. Note that these are not directly comparable to the evaluation of BILTY on version 1.2 of the Universal Dependencies treebanks (Plank et al., 2016, Table 2, column $\vec{w}+\vec{c}$), since the treebanks have been extended and the tagset itself has been somewhat modified since then. Still, our evaluation largely agrees with theirs.

As pointed out by Plank et al. (2016), the accuracy of their and other neural network-based models degrades relatively quickly with reduced training data size. To see how much this affects our results, Figure 1 shows the relationship between training data size and tagging accuracy for both BILTY and EFSELAB. As the regression lines indicate, our method is indeed less sensitive to the size of training data: accuracy is proportional to $n^{-0.33}$ for data size n, compared to $n^{-0.44}$ for BILTY. For small corpora, below about 50,000 tokens, the neural network model of BILTY indeed performs poorly in all cases. For the largest corpora the performance of BILTY is stronger, frequently outperforming EFSELAB's perceptron model—but this is by no means absolute, and EFSELAB has the lowest error rate for the largest (Czech) corpus.

5 Discussion

Given the results above, we should ask ourselves *why* one method is better than another for tagging a particular language. Given that the dataset represents a relatively diverse set of languages, the first explanation to consider is that some models are more suited for certain types of languages. However, when we look at the results on very similar datasets, this explanation is not supported. For instance, in some cases (Ancient Greek and Latin) different systems are better at different treebanks in the same language, and the same holds for closely related languages (Norwegian, Danish and Swedish; Catalan, Portuguese and Spanish).

In the previous section we showed that training data size has a stronger effect on the accuracy of the neural network tagger, which is expected since this model does not have

Northern European Journal of Language Technology, 2018, Vol. 5, Article 1, pp 1–15
DOI 10.3384/nejlt.2000-1533.1851

Table 5: Error rate (in percent) for universal part-of-speech tags on the official test sets from the Universal Dependency treebanks version 2.0.

Language	BILTY	UDPIPE	OUR	Language	BILTY	UDPIPE	OUR
Anc. Greek	**10.4**	17.1	12.3	Indonesian	7.0	**6.6**	7.1
Anc. Greek (PROIEL)	4.0	4.2	**3.5**	Irish	29.3	18.5	**15.1**
Arabic	**5.2**	5.7	**5.2**	Italian	2.5	2.7	**2.3**
Basque	6.7	7.6	**5.8**	Japanese	**3.1**	3.3	3.8
Belarusian	32.4	15.4	**9.8**	Korean	**5.3**	5.8	6.0
Bulgarian	**1.9**	2.3	2.0	Latin	21.3	20.3	**15.6**
Catalan	**1.9**	**1.9**	2.4	Latin (ITTB)	**2.2**	2.7	2.7
Chinese	**7.4**	7.9	9.0	Latin (PROIEL)	5.2	4.6	**4.2**
Coptic	5.5	5.7	**4.9**	Latvian	9.9	10.8	**8.9**
Croatian	4.1	4.0	**3.6**	Lithuanian	43.2	28.3	**20.9**
Czech	1.6	1.9	**1.4**	Norwegian (Bokmål)	**2.7**	3.0	2.9
Czech (CAC)	1.3	1.9	**1.2**	Norwegian (Nynorsk)	**2.8**	3.4	3.3
Czech (CLTT)	4.0	3.6	**2.5**	Old Church Slavonic	5.4	6.2	**4.7**
Danish	4.0	4.4	**3.7**	Persian	3.5	3.8	**3.4**
Dutch	8.4	8.8	**7.7**	Polish	3.8	4.3	**3.0**
Dutch (LassySmall)	2.8	**2.3**	2.4	Portuguese	**3.1**	3.2	3.4
English	5.9	5.4	**5.2**	Portuguese (BR)	**2.7**	2.8	3.1
English (LinES)	5.6	5.0	**4.7**	Romanian	3.3	3.1	**2.9**
English (ParTUT)	7.3	6.2	**5.3**	Russian	4.5	5.2	**4.1**
Estonian	11.8	11.7	**9.7**	Sanskrit	50.8	47.5	**39.0**
Finnish	5.7	5.3	**4.2**	Slovak	5.6	7.4	**4.8**
Finnish (FTB)	9.1	7.4	**6.8**	Slovenian	**2.8**	3.5	3.2
French	3.7	3.5	**3.4**	Slovenian (SST)	14.9	13.3	**11.0**
French (ParTUT)	11.7	7.1	**5.9**	Spanish	4.8	4.2	**4.1**
French (Sequoia)	3.1	3.3	**2.6**	Spanish (AnCora)	2.0	**1.8**	2.1
Galician	3.2	3.1	**2.6**	Swedish	3.9	4.2	**3.7**
Galician (TreeGal)	17.5	12.8	**9.6**	Swedish (LinES)	5.6	5.6	**5.0**
German	**7.0**	8.6	7.2	Tamil	22.8	14.5	**13.4**
Gothic	5.2	5.4	**4.3**	Turkish	6.8	6.7	**5.7**
Greek	3.8	4.6	**3.2**	Ukrainian	63.8	41.0	**38.1**
Hebrew	**4.3**	4.9	4.5	Urdu	7.7	8.3	**6.7**
Hindi	3.9	4.2	**3.5**	Vietnamese	13.7	12.4	**11.7**
Hungarian	8.2	9.5	**6.6**				
Summary statistics below refer to all 65 treebanks, in both columns above							
Mean error rate	9.0	7.9	6.7	No. of (shared) top ranks	15	4	48

Northern European Journal of Language Technology, 2018, Vol. 5, Article 1, pp 1–15
DOI 10.3384/nejlt.2000-1533.1851

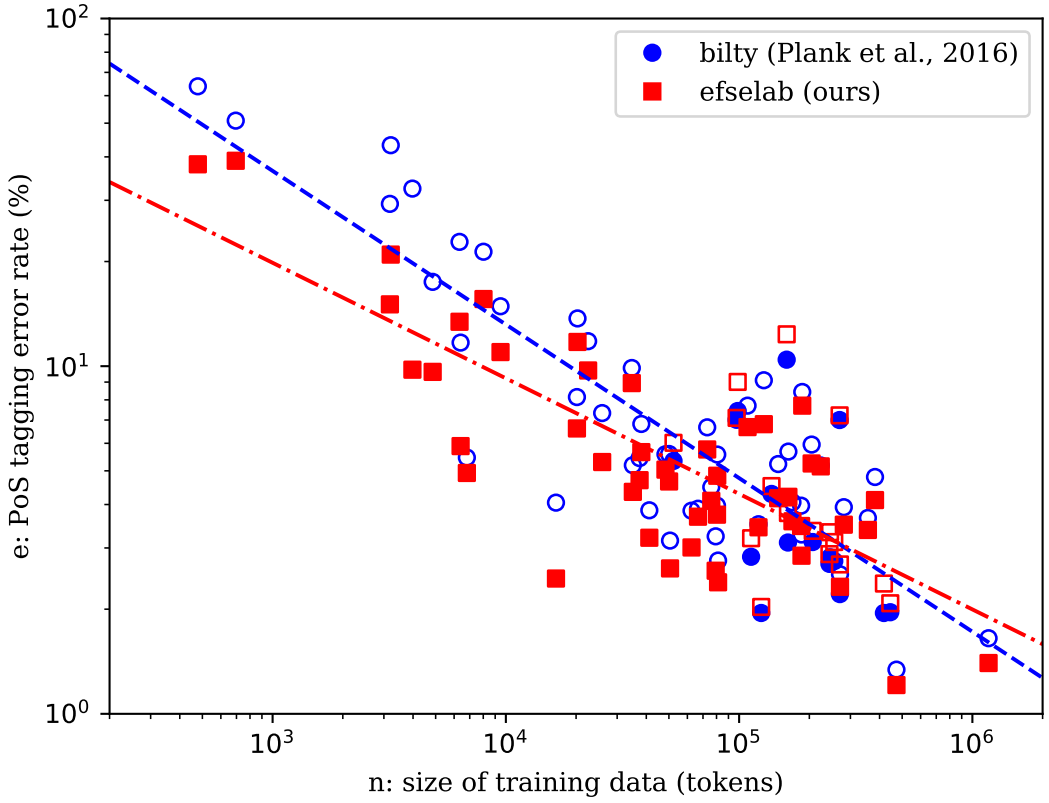

Figure 1: Relationship between training data size and model error rate. Filled markers represent the best system for each dataset. Regression parameters: $e = 771 \cdot n^{-0.44}$ (BILTY), $e = 197 \cdot n^{-0.33}$ (EFSELAB), where e is the error rate in percent and n is the training data size in tokens.

access to manually extracted features. From Figure 1, it is clear that the neural network model is less suitable in nearly all cases with less than 100,000 tokens of training data. For treebanks with more training data available, the differences in accuracy are small and non-systematic. An in-depth analysis of tagging errors might reveal some patterns in which types of morphological or syntactic structures that are challenging to the different models, but that would be beyond the scope of this work. Since the difference in accuracy is not systematic between treebanks in similar or even the same languages, we do not expect such an analysis to reveal particularly strong patterns. Thus, our conclusion is that with respect to PoS tagging accuracy, the model of Plank et al. (2016) is less suitable than a perceptron tagger for small datasets (less than 100,000 tokens), and equally suitable for larger datasets.

We initially set out to investigate the tradeoff between accuracy and performance in PoS tagging, but the results indicate that this might be a false choice—it is in fact possible to have both. Perhaps our most important finding is that a perceptron-based PoS tagger using decades-old technology outperforms a state-of-the-art neural network tagger. Narrowly, in terms of accuracy, but by orders of magnitude in terms of computational efficiency. By evaluating this on the most recent version of the Universal Dependencies treebanks, we can make these conclusions with a high degree of certainty by basing them on results from 65 treebanks in 48 languages. Supervised PoS tagging is often mentioned as a typical example of a "solved" problem in NLP where progress has plateaued, and this would seem to be supported by our results.

Acknowledgments

We would like to thank the three anonymous reviewers, who all provided constructive feedback that contributed to improving the present work. Part of this work has been supported by an infrastructure grant from the Swedish Research Council (SWE-CLARIN, project 821-2013-2003).

References

Appleyard, Jeremy, Tomás Kociský, and Phil Blunsom. 2016. Optimizing performance of recurrent neural networks on GPUs. *CoRR* abs/1604.01946.

Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *CoRR* abs/1409.0473.

Chen, Danqi and Christopher Manning. 2014. A fast and accurate dependency parser using neural networks. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 740–750. Doha, Qatar: Association for Computational Linguistics.

Collins, Michael. 2002. Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms. In *Proceedings of the ACL-02 Conference on Empirical Methods in Natural Language Processing - Volume 10*, EMNLP '02, pages 1–8. Stroudsburg, PA, USA: Association for Computational Linguistics.

Daumé, Hal, III and Daniel Marcu. 2005. Learning as search optimization: Approximate large margin methods for structured prediction. In *Proceedings of the 22Nd International Conference on Machine Learning*, ICML '05, pages 169–176. New York, NY, USA: ACM. ISBN 1-59593-180-5.

Ganchev, Kuzman and Mark Dredze. 2008. Small statistical models by random feature mixing. In *Proceedings of the ACL-2008 Workshop on Mobile Language Processing*. Association for Computational Linguistics.

Goodfellow, Ian J., Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron C. Courville, and Yoshua Bengio. 2014. Generative adversarial networks. *CoRR* abs/1406.2661.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Identity mappings in deep residual networks. *CoRR* abs/1603.05027.

Hochreiter, Sepp and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Computation* 9(8):1735–1780.

Lample, Guillaume and Devendra Singh Chaplot. 2017. Playing FPS games with deep reinforcement learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*, pages 2140–2146.

Neubig, Graham, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, Kevin Duh, Manaal Faruqui, Cynthia Gan, Dan Garrette, Yangfeng Ji, Lingpeng Kong, Adhiguna Kuncoro, Gaurav Kumar, Chaitanya Malaviya, Paul Michel, Yusuke Oda, Matthew Richardson, Naomi Saphra, Swabha Swayamdipta, and Pengcheng Yin. 2017. Dynet: The dynamic neural network toolkit. *CoRR* abs/1701.03980.

Nivre, Joakim et al. 2017. Universal dependencies 2.0 – CoNLL 2017 shared task development and test data. LINDAT/CLARIN digital library at the Institute of Formal and Applied Linguistics, Charles University.

Nivre, Joakim, Johan Hall, Jens Nilsson, Atanas Chanev, Gülsen Eryigit, Sandra Kübler, Svetoslav Marinov, and Erwin Marsi. 2007. MaltParser: A language-independent system for data-driven dependency parsing. *Natural Language Engineering* 13:95–135.

Plank, Barbara, Anders Søgaard, and Yoav Goldberg. 2016. Multilingual part-of-speech tagging with bidirectional long short-term memory models and auxiliary loss. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 412–418. Berlin, Germany: Association for Computational Linguistics.

Ratnaparkhi, Adwait. 1996. A maximum entropy model for part-of-speech tagging. In *Conference on Empirical Methods in Natural Language Processing*, EMNLP 1996, pages 133–142. Philadelphia, PA, USA.

Rosenblatt, Frank. 1957. The perceptron: a perceiving and recognizing automaton. Tech. rep., Cornell Aeronautical Laboratory, inc.

Shen, Libin, Giorgio Satta, and Aravind Joshi. 2007. Guided learning for bidirectional sequence classification. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*, pages 760–767. Prague, Czech Republic: Association for Computational Linguistics.

Silver, David, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529(7587):484–489.

Straka, Milan, Jan Hajič, and Straková Jana. 2016. UDPipe: trainable pipeline for processing CoNLL-U files performing tokenization, morphological analysis, pos tagging and parsing. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*. Paris, France: European Language Resources Association (ELRA).

Straková, Jana, Milan Straka, and Jan Hajič. 2014. Open-Source Tools for Morphology, Lemmatization, POS Tagging and Named Entity Recognition. In *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 13–18. Baltimore, Maryland: Association for Computational Linguistics.

Viterbi, Andrew. 1967. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory* 13(2):260–269.

Wu, Yonghui, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR* abs/1609.08144.