Stack-propagation: Improved Representation Learning for Syntax

Yuan Zhang[*]
CSAIL, MIT
Cambridge, MA 02139, USA
yuanzh@csail.mit.edu

David Weiss
Google Inc
New York, NY 10027, USA
djweiss@google.com

Abstract

Traditional syntax models typically leverage part-of-speech (POS) information by constructing features from hand-tuned templates. We demonstrate that a better approach is to utilize POS tags as a *regularizer* of learned representations. We propose a simple method for learning a stacked pipeline of models which we call "stack-propagation". We apply this to dependency parsing and tagging, where we use the hidden layer of the tagger network as a representation of the input tokens for the parser. At test time, our parser does not require predicted POS tags. On 19 languages from the Universal Dependencies, our method is 1.3% (absolute) more accurate than a state-of-the-art graph-based approach and 2.7% more accurate than the most comparable greedy model.

1 Introduction

In recent years, transition-based dependency parsers powered by neural network scoring functions have dramatically increased the state-of-the-art in terms of both speed and accuracy (Chen and Manning, 2014; Alberti et al., 2015; Weiss et al., 2015). Similar approaches also achieve state-of-the-art in other NLP tasks, such as constituency parsing (Durrett and Klein, 2015) or semantic role labeling (FitzGerald et al., 2015). These approaches all share a common principle: replace hand-tuned conjunctions of traditional NLP feature templates with continuous approximations learned by the hidden layer of a feed-forward network.

However, state-of-the-art dependency parsers depend crucially on the use of predicted part-of-speech (POS) tags. In the pipeline or *stacking* (Wolpert, 1992) method, these are predicted from an independently trained tagger and used as features in the parser. However, there are two main disadvantages of a pipeline: (1) errors from the POS tagger cascade into parsing errors, and (2) POS taggers often make mistakes precisely because they cannot take into account the syntactic context of a parse tree. The POS tags may also contain only coarse information, such as when using the universal tagset of Petrov et al. (2011).

One approach to solve these issues has been to avoid using POS tags during parsing, e.g. either using semi-supervised clustering instead of POS tags (Koo et al., 2008) or building recurrent representations of words using neural networks (Dyer et al., 2015; Ballesteros et al., 2015). However, the best accuracy for these approaches is still achieved by running a POS tagger over the data first and combining the predicted POS tags with additional representations. As an alternative, a wide range of prior work has investigated jointly modeling both POS and parse trees (Li et al., 2011; Hatori et al., 2011; Bohnet and Nivre, 2012; Qian and Liu, 2012; Wang and Xue, 2014; Li et al., 2014; Zhang et al., 2015; Alberti et al., 2015). However, these approaches typically require sacrificing either efficiency or accuracy compared to the best pipeline model, and often they simply re-rank the predictions of a pipelined POS tagger.

In this work, we show how to improve accuracy for both POS tagging and parsing by incorporating stacking into the architecture of a feed-forward network. We propose a continuous form of stacking that allows for easy backpropagation down the pipeline across multiple tasks, a process we call

[*] Research conducted at Google.

Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, pages 1557–1566,
Berlin, Germany, August 7-12, 2016. ©2016 Association for Computational Linguistics

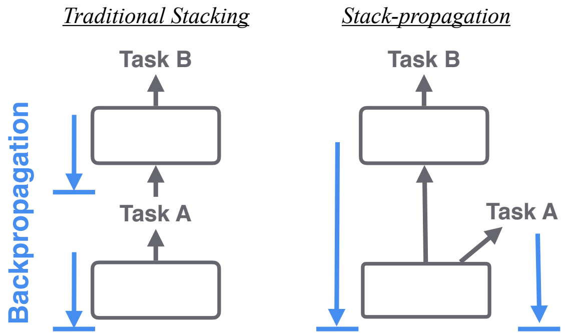


Figure 1: Traditional stacking (left) vs. Stack-propagation (right). Stacking uses the output of Task A as features in Task B, and does not allow backpropagation between tasks. Stack-propagation uses a continuous and differentiable link between Task A and Task B, allowing for backpropagation from Task B into Task A's model. Updates to Task A act as *regularization* on the model for Task B, ensuring the shared component is useful for both tasks.

"stack-propagation" (Figure 1). At the core of this idea is that we use POS tags as *regularization* instead of *features*.

Our model design for parsing is very simple: we use the hidden layer of a window-based POS tagging network as the representation of tokens in a greedy, transition-based neural network parser. Both networks are implemented with a refined version of the feed-forward network (Figure 3) from Chen and Manning (2014), as described in Weiss et al. (2015). We link the tagger network to the parser by translating traditional feature templates for parsing into feed-forward connections from the tagger to the parser (Figure 2). At training time, we unroll the parser decisions and apply stack-propagation by alternating between stochastic updates to the parsing or tagging objectives (Figure 4). The parser's representations of tokens are thus regularized to be individually predictive of POS tags, even as they are trained to be useful for parsing when concatenated and fed into the parser network. This model is similar to the multi-task network structure of Collobert et al. (2011), where Collobert et al. (2011) shares a hidden layer between multiple tagging tasks. The primary difference here is that we show how to unroll parser transitions to apply the same principle to tasks with fundamentally different structure.

The key advantage of our approach is that at test time, we do not require predicted POS tags for parsing. Instead, we run the tagger network up to the hidden layer over the entire sentence, and then dynamically connect the parser network to the tagger network based upon the discrete parser configurations as parsing unfolds. In this way, we avoid cascading POS tagging errors to the parser. As we show in Section 5, our approach can be used in conjunction with joint transition systems in the parser to improve both POS tagging as well as parsing. In addition, because the parser re-uses the representation from the tagger, we can drop all lexicalized features from the parser network, leading to a compact, faster model.

The rest of the paper is organized as follows. In Section 2, we describe the layout of our combined architecture. In Section 3, we introduce stack-propagation and show how we train our model. We evaluate our approach on 19 languages from the Universal Dependencies treebank in Section 4. We observe a >2% absolute gain in labeled accuracy compared to state-of-the-art, LSTM-based greedy parsers (Ballesteros et al., 2015) and a >1% gain compared to a state-of-the-art, graph-based method (Lei et al., 2014). We also evaluate our method on the Wall Street Journal, where we find that our architecture outperforms other greedy models, especially when only coarse POS tags from the universal tagset are provided during training. In Section 5, we systematically evaluate the different components of our approach to demonstrate the effectiveness of stack-propagation compared to traditional types of joint modeling. We also show that our approach leads to large reductions in cascaded errors from the POS tagger.

We hope that this work will motivate further research in combining traditional pipelined structured prediction models with deep neural architectures that learn intermediate representations in a task-driven manner. One important finding of this work is that, even *without* POS tags, our architecture outperforms recurrent approaches that build custom word representations using character-based LSTMs (Ballesteros et al., 2015). These results suggest that learning rich embeddings of words may not be as important as building an intermediate representation that takes multiple features of the *surrounding* context into account. Our results also suggest that deep models for dependency parsing may not discover POS classes when trained solely for parsing, even when it is fully within the capacity of the model. Designing architectures to apply stack-propagation in other coupled NLP tasks might yield significant accuracy improvements for deep learning.

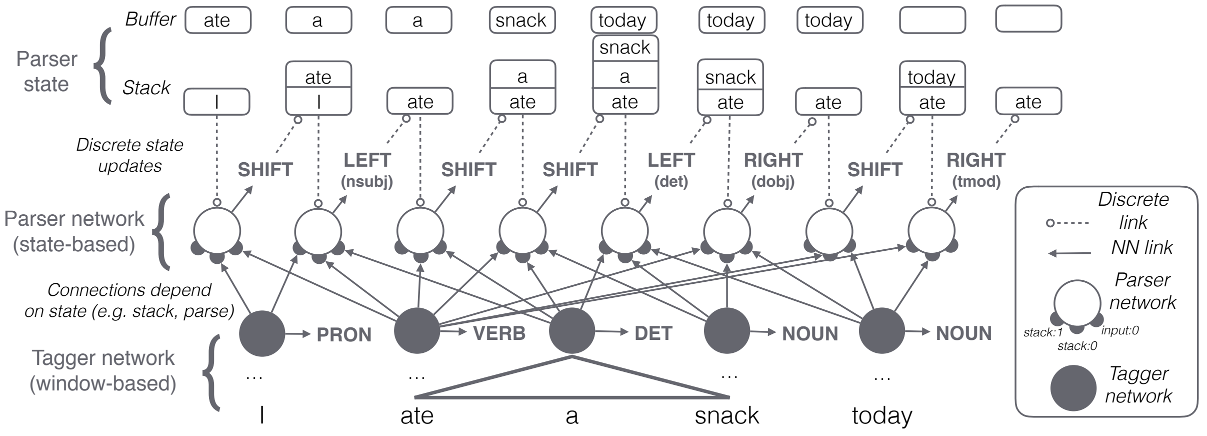


Figure 2: Detailed example of the stacked parsing model. *Top:* The discrete parser state, consisting of the stack and the buffer, is updated by the output of the parser network. In turn, the feature templates used by the parser are a function of the state. In this example, the parser has three templates, `stack:0`, `stack:1`, and `input:0`. *Bottom:* The feature templates create many-to-many connections from the hidden layer of the tagger to the input layer of the parser. For example, the predicted root of the sentence ("ate") is connected to the input of most parse decisions. At test time, the above structure is constructed dynamically as a function of the parser output. Note also that the predicted POS tags are not directly used by the parser.

2 Continuous Stacking Model

In this section, we introduce a novel neural network model for parsing and tagging that incorporates POS tags as a regularization of learned implicit representations. The basic unit of our model (Figure 3) is a simple, feed-forward network that has been shown to work very well for parsing tasks (Chen and Manning, 2014; Weiss et al., 2015). The inputs to this unit are feature matrices which are embedded and passed as input to a hidden layer. The final layer is a softmax prediction.

We use two such networks in this work: a window-based version for tagging and a transition-based version for dependency parsing. In a traditional stacking (pipeline) approach, we would use the discrete predicted POS tags from the tagger as features in the parser (Chen and Manning, 2014). In our model, we instead feed the continuous hidden layer activations of the tagger network as input to the parser. The primary strength of our approach is that the parser has access to all of the features and information used by the POS tagger during training time, but it is allowed to make its own decisions at test time.

To implement this, we show how we can reuse feature templates from Chen and Manning (2014) to specify the feed-forward connections from the tagger network to the parser network. An interesting consequence is that because this structure is a function of the derivation produced by the parser, the final feed-forward structure of the stacked model is not known until *run-time*. However, because the connections for any specific parsing decision are fixed given the derivation, we can still extract examples for training off-line by unrolling the network structure from gold derivations. In other words, we can utilize our approach with the same simple stochastic optimization techniques used in prior works. Figure 2 shows a fully unrolled architecture on a simple example.

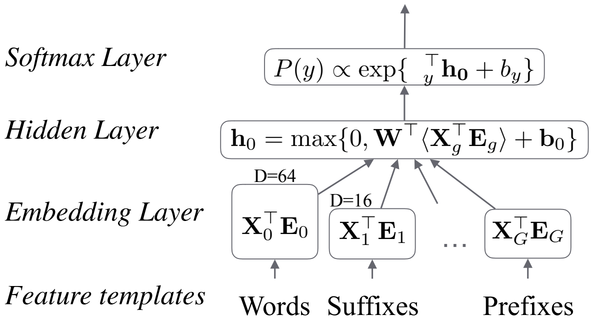
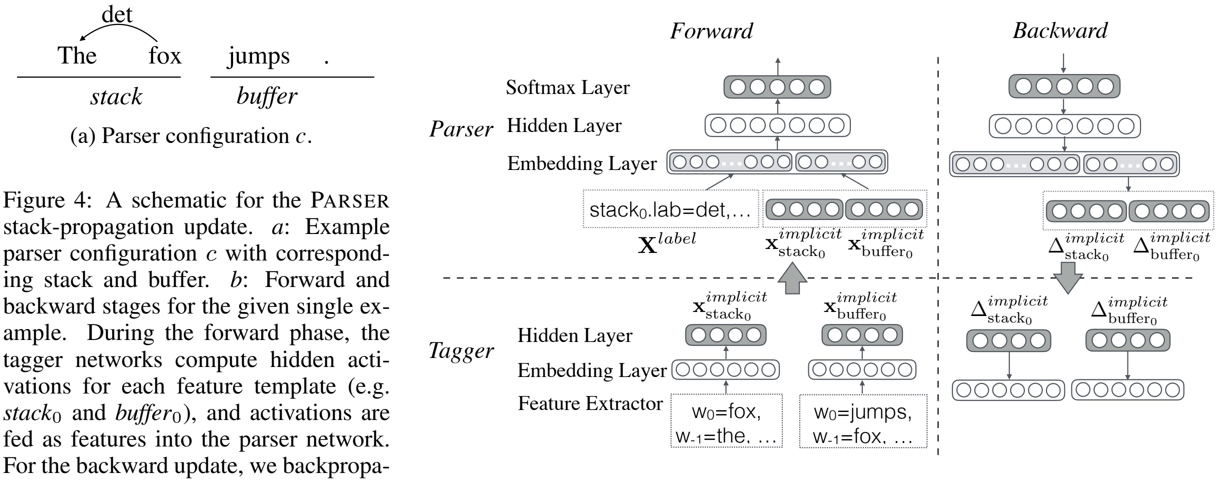

Softmax Layer
$P(y) \propto \exp\{ {}_y^\top \mathbf{h_0} + b_y\}$
Hidden Layer
$\mathbf{h}_0 = \max\{0, \mathbf{W}^\top \langle \mathbf{X}_g^\top \mathbf{E}_g \rangle + \mathbf{b}_0\}$
Embedding Layer
D=64
D=16
$\mathbf{X}_0^\top \mathbf{E}_0$
$\mathbf{X}_1^\top \mathbf{E}_1$
...
$\mathbf{X}_G^\top \mathbf{E}_G$
Feature templates
Words
Suffixes
Prefixes

Figure 3: Elementary NN unit used in our model. Feature matrices from multiple channels are embedded, concatenated together, and fed into a rectified linear hidden layer. In the parser network, the feature inputs are continuous representations from the tagger network's hidden layer.

2.1 The Tagger Network

As described above, our POS tagger follows the basic structure from prior work with embedding, hidden, and softmax layers. Like the "window-approach" network of Collobert et al. (2011), the tagger is evaluated per-token, with features extracted from a window of tokens surrounding the target. The input consists of a rich set of fea-



(a) Parser configuration c.

(b) Forward and backward schema for (a).

Figure 4: A schematic for the PARSER stack-propagation update. *a*: Example parser configuration c with corresponding stack and buffer. *b*: Forward and backward stages for the given single example. During the forward phase, the tagger networks compute hidden activations for each feature template (e.g. *stack*$_0$ and *buffer*$_0$), and activations are fed as features into the parser network. For the backward update, we backpropagate training signals from the parser network into each linked tagging example.

tures for POS tagging that are deterministically extracted from the training data. As in prior work, the features are divided into *groups* of different sizes that share an embedding matrix $\mathbf{E}$. Features for each group g are represented as a sparse matrix $\mathbf{X}^g$ with dimension $F^g \times V^g$, where F^g is the number of feature templates in the group, and V^g is the vocabulary size of the feature templates. Each row of $\mathbf{X}^g$ is a one-hot vector indicating the appearance of each feature.

The network first looks up the learned embedding vectors for each feature and then concatenates them to form the embedding layer. This embedding layer can be written as:

$$\mathbf{h}_0 = [\mathbf{X}^g \mathbf{E}^g \mid \forall g] \qquad (1)$$

where $\mathbf{E}^g$ is a learned $V^g \times D^g$ embedding matrix for feature group. Thus, the final size $|\mathbf{h}_0| = \sum_g F^g D^g$ is the sum of all embedded feature sizes. The specific features and their dimensions used in the tagger are listed in Table 1. Note that for all features, we create additional *null* value that triggers when features are extracted outside the scope of the sentence. We use a single hidden layer in our model and apply rectified linear unit (ReLU) activation function over the hidden layer outputs. A final softmax layer reads in the activations and outputs probabilities for each possible POS tag.

2.2 The Parser Network

The parser component follows the same design as the POS tagger with the exception of the features and the output space. Instead of a window-based classifier, features are extracted from an arc-standard parser configuration[1] c consisting of the stack s, the buffer b and the so far constructed dependencies (Nivre, 2004). Prior implementations of this model used up to four groups of discrete features: words, labels (from previous decisions), POS tags, and morphological attributes (Chen and Manning, 2014; Weiss et al., 2015; Alberti et al., 2015).

Features (g)	Window	D
Symbols	1	8
Capitalization	+/- 1	4
Prefixes/Suffixes ($n = 2, 3$)	+/- 1	16
Words	+/-3	64

Table 1: Window-based tagger feature spaces. "Symbols" indicates whether the word contains a hyphen, a digit or a punctuation.

In this work, we apply the same design principle but we use an *implicitly* learned intermediate representation in the parser to replace traditional discrete features. We only retain discrete features over the labels in the incrementally constructed tree (Figure 4). Specifically, for any token of interest, we feed the hidden layer of the tagger network evaluated for that token as input to the parser. We implement this idea by re-using the feature templates from prior work as indexing functions.

We define this process formally as follows. Let $f_i(c)$ be a function mapping from parser configurations c to indices in the sentence, where i denotes each of our feature templates. For example, in Figure 4(a), when $i = \texttt{stack}_0$, $f_i(c)$ is the in-

[1] Note that the "stack" in the parse configuration is separate from the "stacking" of the POS tagging network and the parser network (Figure 1).

dex of "fox" in the sentence. Let $\mathbf{h}_1^{tagger}(j)$ be the hidden layer activation of the tagger network evaluated at token j. We define the input $\mathbf{X}^{implicit}$ by concatenating these tagger activations according to our feature templates:

$$\mathbf{x}_i^{implicit} \triangleq \mathbf{h}_1^{tagger}(f_i(c)). \qquad (2)$$

Thus, the feature group $\mathbf{X}^{implicit}$ is the row-concatenation of the hidden layer activations of the tagger, as indexed by the feature templates. We have that $F^{implicit}$ is the number of feature templates, and $V^{implicit} = H^{tagger}$, the number of possible values is the number of hidden units in the tagger. Just as for other features, we learn an embedding matrix $\mathbf{E}^{implicit}$ of size $H^{implicit} \times F^{implicit}$. Note that as in the POS tagger network, we reserve an additional *null* value for out of scope feature templates. A full example of this lookup process, and the resulting feed-forward network connections created, is shown for a simple three-feature template consisting of the top two tokens on the stack and the first on the buffer in Figure 2. See Table 1 for the full list of 20 tokens that we extract for each state.

3 Learning with Stack-propagation

In this section we describe how we train our stacking architecture. At a high level, we simply apply backpropagation to our proposed continuous form of stacking (hence "stack-propagation.") There are two major issues to address: (1) how to handle the dynamic many-to-many connections between the tagger network and the parser network, and (2) how to incorporate the POS tag labels during training.

Addressing the first point turns out to be fairly easy in practice: we simply unroll the gold trees into a derivation of (state, action) pairs that produce the tree. The key property of our parsing model is that the *connections* of the feed-forward network are constructed incrementally as the parser state is updated. This is different than a generic recurrent model such as an LSTM, which passes activation vectors from one step to the next. The important implication at training time is that, unlike a recurrent network, the parser decisions are conditionally independent given a fixed history. In other words, if we unroll the network structure ahead of time given the gold derivation, we do not need to perform inference when training with respect to these examples. Thus, the overall training procedure is similar to that introduced in Chen and Manning (2014).

To incorporate the POS tags as a regularization during learning, we take a fairly standard approach from multi-task learning. The objective of learning is to find parameters Θ that maximize the data log-likelihood with a regularization on Θ for both parsing and tagging:

$$\max_{\Theta} \lambda \sum_{\mathbf{x},y \in \mathcal{T}} \log(P_\Theta(y \mid \mathbf{x})) + \sum_{c,a \in \mathcal{P}} \log\left(P_\Theta(a \mid c)\right), \qquad (3)$$

where $\{\mathbf{x}, y\}$ are POS tagging examples extracted from individual tokens and $\{c, a\}$ are parser (configuration, action) pairs extracted from the unrolled gold parse tree derivations, and λ is a trade-off parameter.

We optimize this objective stochastically by alternating between two updates:

- TAGGER: Pick a POS tagging example and update the tagger network with backpropagation.
- PARSER: (Figure 4) Given a parser configuration c from the set of gold contexts, compute both tagger and parser activations. Backpropagate the parsing loss through the stacked architecture to update both parser and tagger, ignoring the tagger's softmax layer parameters.

While the learning procedure is inspired from multi-task learning—we only update each step with regards one of the two likelihoods—there are subtle differences that are important. While a traditional multi-task learning approach would use the final layer of the parser network to predict both POS tags and parse trees, we predict POS tags from the first hidden layer of our model (the "tagger" network) only. We treat the POS labels as regularization of our parser and simply discard the softmax layer of the tagger network at test time. As we will show in Section 4, this regularization leads to dramatic gains in parsing accuracy. Note that in Section 5, we also show experimentally that stack-propagation is more powerful than the traditional multi-task approach, and by combining them together, we can achieve better accuracy on both POS and parsing tasks.

Method	ar	bg	da	de	en	es	eu	fa	fi	fr	hi	id	it	iw	nl	no	pl	pt	sl	AVG
NO TAGS																				
B'15 LSTM	75.6	**83.1**	69.6	**72.4**	77.9	78.5	67.5	74.7	**73.2**	77.4	85.9	**72.3**	**84.1**	73.1	69.5	82.4	**78.0**	**79.9**	**80.1**	76.6
Ours (window)	**76.1**	82.9	**70.9**	71.7	**79.2**	**79.3**	**69.1**	**77.5**	72.5	**78.2**	**87.1**	71.8	83.6	**76.2**	**72.3**	**83.2**	77.8	79.0	79.8	**77.3**
UNIVERSAL TAGSET																				
B'15 LSTM	74.6	82.4	68.1	73.0	77.9	77.8	66.0	75.0	73.6	78.0	86.8	72.2	84.2	74.5	68.4	83.3	74.5	80.4	78.1	76.2
Pipeline P_{tag}	73.7	83.6	72.0	73.0	79.3	79.5	63.0	78.0	66.9	78.5	87.8	73.5	84.2	75.4	70.3	83.6	73.4	79.5	79.4	76.6
RBGParser	75.8	83.6	**73.9**	73.5	79.9	79.6	68.0	**78.5**	65.4	78.9	87.7	**74.2**	84.7	**77.6**	72.4	83.9	75.4	**81.3**	80.7	77.6
Stackprop	**77.0**	**84.3**	73.8	**74.2**	**80.7**	**80.7**	**70.1**	**78.5**	**74.5**	**80.0**	**88.9**	74.1	**85.8**	77.5	**73.6**	**84.7**	**79.2**	80.4	**81.8**	**78.9**

Table 2: Labeled Attachment Score (LAS) on Universal Dependencies Treebank. *Top:* Results without any POS tag observations. "B'15 LSTM" is the character-based LSTM model (Ballesteros et al., 2015), while "Ours (window)" is our window-based architecture variant without stackprop. *Bottom:* Comparison against state-of-the-art baselines utilizing the POS tags. Paired t-tests show that the gain of Stackprop over all other approaches is significant ($p < 10^{-5}$ for all but RBGParser, which is $p < 0.02$).

3.1 Implementation details

Following Weiss et al. (2015), we use mini-batched averaged stochastic gradient descent (ASGD) (Bottou, 2010) with momentum (Hinton, 2012) to learn the parameters Θ of the network. We use a separate learning rate, moving average, and velocity for the tagger network and the parser; the PARSER updates all averages, velocities, and learning rates, while the TAGGER updates only the tagging factors. We tuned the hyperparameters of momentum rate μ, the initial learning rate η_0 and the learning rate decay step γ using held-out data. The training data for parsing and tagging can be extracted from either the same corpus or different corpora; in our experiments they were always the same.

To trade-off the two objectives, we used a random sampling scheme to perform 10 epochs of PARSER updates and 5 epochs of TAGGER updates. In our experiments, we found that pre-training with TAGGER updates for one epoch before interleaving PARSER updates yielded faster training with better results. We also experimented using the TAGGER updates solely for initializing the parser and found that interleaving updates was crucial to obtain improvements over the baseline.

4 Experiments

In this section, we evaluate our approach on several dependency parsing tasks across a wide variety of languages.

4.1 Experimental Setup

We first investigated our model on 19 languages from the Universal Dependencies Treebanks v1.2.[2] We selected the 19 largest currently spoken languages for which the full data was freely available. We used the coarse universal tagset in our experiments with no explicit morphological annotations. To measure parsing accuracy, we report unlabeled attachment score (UAS) and labeled attachment score (LAS) computed on all tokens (including punctuation), as is standard for non-English datasets.

For simplicity, we use the arc-standard (Nivre, 2004) transition system with greedy decoding. Because this transition system only produces projective trees, we first apply a projectivization step to all treebanks before unrolling the gold derivations during training. We make an exception for Dutch, where we observed a significant gain on development data by introducing the SWAP action (Nivre, 2009) and allowing non-projective trees.

For models that required predicted POS tags, we trained a window-based tagger using the same features as the tagger component of our stacking model. We used 5-fold jackknifing to produce predicted tags on the training set. We found that the window-based tagger was comparable to a state-of-the-art CRF tagger for most languages. For every network we trained, we used the development data to evaluate a small range of hyperparameters, stopping training early when UAS no longer improved on the held-out data. We use $H = 1024$ hidden units in the parser, and $H = 128$ hidden units in the tagger. The parser embeds the tagger activations with $D = 64$. Note that following Ballesteros et al. (2015), we did not use any auxiliary data beyond that in the treebanks, such as pre-trained word embeddings.

For a final set of experiments, we evaluated on the standard Wall Street Journal (WSJ) part of the Penn Treebank (Marcus et al., 1993)), dependencies generated from version 3.3.0 of the Stanford

[2] http://universaldependencies.org

Method	UAS	LAS
NO TAGS		
Dyer et al. (2015)	92.70	90.30
Ours (window-based)	**92.85**	**90.77**
UNIVERSAL TAGSET		
Pipeline (P_{tag})	92.52	90.50
Stackprop	**93.23**	**91.30**
FINE TAGSET		
Chen & Manning (2014)	91.80	89.60
Dyer et al. (2015)	93.10	90.90
Pipeline (P_{tag})	93.10	91.16
Stackprop	**93.43**	**91.41**
Weiss et al. (2015)	93.99	92.05
Alberti et al. (2015)	94.23	92.36

Table 3: WSJ Test set results for greedy and state-of-the-art methods. For reference, we show the most accurate models from Alberti et al. (2015) and Weiss et al. (2015), which use a deeper model and beam search for inference.

converter (De Marneffe et al., 2006). We followed standard practice and used sections 2-21 for training, section 22 for development, and section 23 for testing. Following Weiss et al. (2015), we used section 24 to tune any hyperparameters of the model to avoid overfitting to the development set. As is common practice, we use pretrained word embeddings from the `word2vec` package when training on this dataset.

4.2 Results

We present our main results on the Universal Treebanks in Table 2. We directly compare our approach to other baselines in two primary ways. First, we compare the effectiveness of our learned continuous representations with those of Alberti et al. (2015), who use the predicted distribution over POS tags concatenated with word embeddings as input to the parser. Because they also incorporate beam search into training, we re-implement a greedy version of their method to allow for direct comparisons of token representations. We refer to this as the "Pipeline (P_{tag})" baseline. Second, we also compare our architecture trained without POS tags as regularization, which we refer to as "Ours (window-based)". This model has the same architecture as our full model but with no POS supervision and updates. Since this model never observes POS tags in any way, we compare against a recurrent character-based parser (Ballesteros et al., 2015) which is state-of-the-art when no POS tags are provided.[3] Finally, we compare to RGBParser (Lei et al., 2014), a state-of-the art graph-based (non-greedy) approach.

Model Variant	UAS	LAS	POS
Arc-standard transition system			
Pipeline (P_{tag})	81.56	76.55	95.14
Ours (window-based)	82.08	77.08	-
Ours (Stackprop)	**83.38**	**78.78**	-
Joint parsing & tagging transition system			
Pipeline (P_{tag})	81.61	76.57	95.30
Ours (window-based)	82.58	77.76	94.92
Ours (Stackprop)	**83.21**	**78.64**	**95.43**

Table 4: Averaged parsing and POS tagging results on the UD treebanks for joint variants of stackprop. Given the window-based architecture, stackprop leads to higher parsing accuracies than joint modeling (83.38% vs. 82.58%).

Our greedy stackprop model outperforms all other methods, including the graph-based RBGParser, by a significant margin on the test set (78.9% vs 77.6%). This is despite the limitations of greedy parsing. Stackprop also yields a 2.3% absolute improvement in accuracy compared to using POS tag confidences as features (Pipeline P_{tag}). Finally, we also note that adding stackprop to our window-based model improves accuracy in *every* language, while incorporating predicted POS tags into the LSTM baseline leads to occasional drops in accuracy (most likely due to cascaded errors.)

5 Discussion

Stackprop vs. other representations. One unexpected result was that, even *without* the POS tag labels at training time, our stackprop architecture achieves better accuracy than either the character-based LSTM or the pipelined baselines (Table 2). This suggests that adding window-based representations–which aggregate over many features of the word and surrounding context–is more effective than increasing the expressiveness of individual word representations by using character-based recurrent models. In future work we will explore combining these two complementary approaches.

We hypothesized that stackprop might provide larger gains over the pipelined model when the

[3] We thank Ballesteros et al. (2015) for their assistance running their code on the treebanks.

Token	married by a **judge**.	Don't **judge** a book by	and **walked** away satisfied	when I **walk** in the door
Neighbors	mesmerizing as a *rat*. A *staple!* day at a *bar*, then go	doesn't *change* the company's won't *charge* your phone don't *waste* your money	tried, and *tried* hard and *incorporated* into and *belonged* to the	upset when I *went* to I *mean* besides me I *felt* as if I
Pattern	a [noun]	'nt [verb]	and [verb]ed	I [verb]

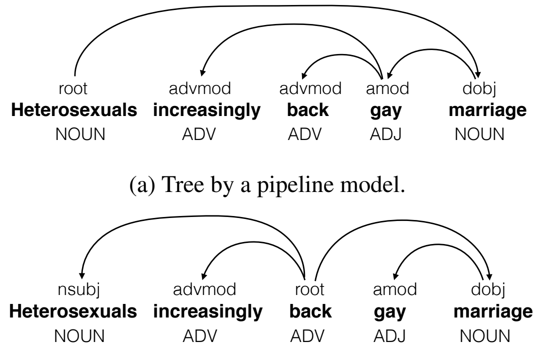
Table 5: Four of examples of tokens in context, along with the three most similar tokens according to the tagger network's activations, and the simple pattern exhibited. Note that this model was trained with the Universal tagset which does not distinguish verb tense.

POS tags are very coarse. We tested this latter hypothesis on the WSJ corpus by training our model using the coarse universal tagsets instead of the fine tagset (Table 3). We found that stackprop achieves similar accuracy using coarse tagsets as the fine tagset, while the pipelined baseline's performance drops dramatically. And while stackprop doesn't achieve the highest reported accuracies on the WSJ, it does achieve competitive accuracies and outperforms prior state-of-the-art for greedy methods (Dyer et al., 2015).

Stackprop vs. joint modeling. An alternative to stackprop would be to train the *final* layer of our architecture to predict both POS tags and dependency arcs. To evaluate this, we trained our window-based architecture with the integrated transition system of Bohnet and Nivre (2012), which augments the SHIFT transition to predict POS tags. Note that if we also apply stackprop, the network learns from POS annotations twice: once in the TAGGER updates, and again the PARSER updates. We therefore evaluated our window-based model both with and without stack-propagation, and with and without the joint transition system.

We compare these variants along with our reimplementation of the pipelined model of Alberti et al. (2015) in Table 4. We find that stackprop is always better, even when it leads to "double counting" the POS annotations; in this case, the result is a model that is significantly better at POS tagging while marginally worse at parsing than stackprop alone.

Reducing cascaded errors. As expected, we observe a significant reduction in cascaded POS tagging errors. An example from the English UD treebank is given in Figure 5. Across the 19 languages in our test set, we observed a 10.9% gain (34.1% vs. 45.0%) in LAS on tokens where the pipelined POS tagger makes a mistake, compared to a 1.8% gain on the rest of the corpora.



(a) Tree by a pipeline model.

(b) Tree by Stackprop model.

Figure 5: Example comparison between predictions by a pipeline model and a joint model. While both models predict a wrong POS tag for the word "back" (ADV rather than VERB), the joint model is robust to this POS error and predict the correct parse tree.

Decreased model size. Previous neural parsers that use POS tags require learning embeddings for words and other features on top of the parameters used in the POS tagger (Chen and Manning, 2014; Weiss et al., 2015). In contrast, the number of total parameters for the combined parser and tagger in the Stackprop model is reduced almost by half compared to the Pipeline model, because the parser and tagger share parameters. Furthermore, compared to our implementation of the pipeline model, we observed that this more compact parser model was also roughly twice as fast.

Contextual embeddings. Finally, we also explored the significance of the representations learned by the tagger. Unlike word embedding models, the representations used in our parser are constructed for each token based on its surrounding context. We demonstrate a few interesting trends we observed in Table 5, where we show the nearest neighbors to sample tokens in this contextual embedding space. These representations tend to represent syntactic patterns rather than individual words, distinguishing between the form (e.g. "judge" as a noun vs. a verb') and context of tokens (e.g. preceded by a personal pronoun).

6 Conclusions

We present a stacking neural network model for dependency parsing and tagging. Through a simple learning method we call "stack-propagation," our model learns effective intermediate representations for parsing by using POS tags as regularization of implicit representations. Our model outperforms all state-of-the-art parsers when evaluated on 19 languages of the Universal Dependencies treebank and outperforms other greedy models on the Wall Street Journal.

We observe that the ideas presented in this work can also be as a principled way to optimize up-stream NLP components for down-stream applications. In future work, we will extend this idea beyond sequence modeling to improve models in NLP that utilize parse trees as features. The basic tenet of stack-propagation is that the hidden layers of neural models used to generate annotations can be used instead of the annotations themselves. This suggests a new methodology to building deep neural models for NLP: we can design them from the ground up to incorporate multiple sources of annotation and learn far more effective intermediate representations.

Acknowledgments

We would like to thank Ryan McDonald, Emily Pitler, Chris Alberti, Michael Collins, and Slav Petrov for their repeated discussions, suggestions, and feedback, as well all members of the Google NLP Parsing Team. We would also like to thank Miguel Ballesteros for assistance running the character-based LSTM.

References

Chris Alberti, David Weiss, Greg Coppola, and Slav Petrov. 2015. Improved transition-based parsing and tagging with neural networks. In *Proceedings of EMNLP 2015*.

Miguel Ballesteros, Chris Dyer, and Noah A. Smith. 2015. Improved transition-based parsing by modeling characters instead of words with lstms. *Proceddings of EMNLP*.

Bernd Bohnet and Joakim Nivre. 2012. A transition-based system for joint part-of-speech tagging and labeled non-projective dependency parsing. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 1455–1465. Association for Computational Linguistics.

Léon Bottou. 2010. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer.

Danqi Chen and Christopher D 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)*, volume 1, pages 740–750.

Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel P. Kuksa. 2011. Natural language processing (almost) from scratch. *JMLR*.

Marie-Catherine De Marneffe, Bill MacCartney, and Christopher D. Manning. 2006. Generating typed dependency parses from phrase structure parses. In *Proceedings of Fifth International Conference on Language Resources and Evaluation*, pages 449–454.

Greg Durrett and Dan Klein. 2015. Neural crf parsing. In *Proceedings of the Association for Computational Linguistics*. Association for Computational Linguistics.

Chris Dyer, Miguel Ballesteros, Wang Ling, Austin Matthews, and Noah A Smith. 2015. Transition-based dependency parsing with stack long short-term memory. *ACL*.

Nicholas FitzGerald, Oscar Tckstrm, Kuzman Ganchev, and Dipanjan Das. 2015. Semantic role labeling with neural network factors. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP '15)*.

Jun Hatori, Takuya Matsuzaki, Yusuke Miyao, and Jun'ichi Tsujii. 2011. Incremental joint pos tagging and dependency parsing in chinese. In *IJCNLP*, pages 1216–1224. Citeseer.

Geoffrey E Hinton. 2012. A practical guide to training restricted boltzmann machines. In *Neural Networks: Tricks of the Trade*, pages 599–619. Springer.

Terry Koo, Xavier Carreras Pérez, and Michael Collins. 2008. Simple semi-supervised dependency parsing. In *46th Annual Meeting of the Association for Computational Linguistics*, pages 595–603.

Tao Lei, Yu Xin, Yuan Zhang, Regina Barzilay, and Tommi Jaakkola. 2014. Low-rank tensors for scoring dependency structures. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*, volume 1, pages 1381–1391.

Zhenghua Li, Min Zhang, Wanxiang Che, Ting Liu, Wenliang Chen, and Haizhou Li. 2011. Joint models for chinese pos tagging and dependency parsing. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 1180–1191. Association for Computational Linguistics.

Zhenghua Li, Min Zhang, Wanxiang Che, Ting Liu, and Wenliang Chen. 2014. Joint optimization for chinese POS tagging and dependency parsing. *IEEE/ACM Transactions on Audio, Speech & Language Processing*, pages 274–286.

Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. 1993. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330.

Joakim Nivre. 2004. Incrementality in deterministic dependency parsing. In *Proceedings of the Workshop on Incremental Parsing: Bringing Engineering and Cognition Together*, IncrementParsing '04, pages 50–57, Stroudsburg, PA, USA. Association for Computational Linguistics.

Joakim Nivre. 2009. Non-projective dependency parsing in expected linear time. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, pages 351–359.

Slav Petrov, Dipanjan Das, and Ryan McDonald. 2011. A universal part-of-speech tagset. *arXiv preprint arXiv:1104.2086*.

Xian Qian and Yang Liu. 2012. Joint chinese word segmentation, pos tagging and parsing. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 501–511. Association for Computational Linguistics.

Zhiguo Wang and Nianwen Xue. 2014. Joint pos tagging and transition-based constituent parsing in chinese with non-local features. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*, pages 733–742.

David Weiss, Chris Alberti, Michael Collins, and Slav Petrov. 2015. Structured training for neural network transition-based parsing. In *Proceedings of ACL 2015*, pages 323–333.

David H Wolpert. 1992. Stacked generalization. *Neural networks*.

Yuan Zhang, Chengtao Li, Regina Barzilay, and Kareem Darwish. 2015. Randomized greedy inference for joint segmentation, POS tagging and dependency parsing. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics*, pages 42–52.