Representation Learning and Dynamic Programming for Arc-Hybrid Parsing

Joseph Le Roux **Antoine Rozenknop** **Mathieu Lacroix**
Laboratoire d'Informatique de Paris Nord,
Université Paris 13 – SPC, CNRS UMR 7030,
F-93430, Villetaneuse, France
{leroux,rozenknop,lacroix}@lipn.fr

Abstract

We present a new method for transition-based parsing where a solution is a pair made of a dependency tree and a derivation graph describing the construction of the former. From this representation we are able to derive an efficient parsing algorithm and design a neural network that learns vertex representations and arc scores. Experimentally, although we only train via local classifiers, our approach improves over previous arc-hybrid systems and reach state-of-the-art parsing accuracy.

1 Introduction

While transition-based dependency parsing is usually implemented as a beam-search procedure, *e.g.* (Kiperwasser and Goldberg, 2016), some recent work such as (Shi et al., 2017) showed that global inference can be performed efficiently with dynamic programming. To this end, the stack representing pending subparses and the buffer representing the unconsumed input must both be abstracted into equivalence classes, while remaining rich enough to help with accurate predictions.

In this paper we first explicitly consider that a solution in transition-based parsing is represented as a pair made of a derivation graph and a derived dependency tree allowing the scoring function to be expressed naturally as a sum over these 2 structures. While we restrict our presentation to arc-hybrid systems, our method can be applied quite directly to other transition rule systems.

Secondly we show that this representation leads to an exact $O(n^4)$ parsing algorithm using dynamic programming. This algorithm can be seen as an extension of the minimal feature set arc-hybrid parsing algorithm presented in (Shi et al., 2017) where the contribution of the dependency arcs can be explicitly added to the scoring function as in the Eisner parsing algorithm (Eisner, 1996).

We then propose an alternative approach to global inference where derivation steps are represented as dense vectors based on the number and type of steps in a derivation. With this abstraction we design a neural architecture based on non-local networks (Wang et al., 2017) related to self-attention mechanism (Vaswani et al., 2017; Gu et al., 2018) to learn these representations while maintaining the possibility for exact decoding.

Our contribution can be summarized as follows: *(i)* a representation of arc-hybrid parsing as maximum subgraphs selection where a solution contains dependencies and derivation information; *(ii)* a polynomial dynamic programming algorithm to solve this problem exactly; *(iii)* a neural architecture able to learn representations for the subgraph vertices and compute arc scores without explicit stack and buffer representations.

These contributions are validated empirically by experimental results on the Penn Treebank where our system reaches state-of-the-art accuracy (94.8% UAS) for arc-hybrid parsing with networks of comparable size.

We first review arc-hybrid dependency parsing (§2) then present a deductive scheme to solve it (§3). The neural architecture is presented in §4 and experiments reported in §5. Finally we discuss some related work in §6.

2 Arc-Hybrid Dependency Parsing

2.1 Arc-Hybrid Derivations

Intuitively, the arc-hybrid parsing strategy (Gómez-Rodríguez et al., 2008; Kuhlmann et al., 2011) builds dependency parses incrementally by reading the sentence from left to right. The pending words are words which have been given all their left modifiers but may have not been given all their right dependents yet. The pending words are stored in the stack which

Proceedings of the 23rd Conference on Computational Natural Language Learning, pages 238–248
Hong Kong, China, November 3-4, 2019. ©2019 Association for Computational Linguistics

initially contains a dummy root word. The words which have not been read yet are stored in order in the buffer. The first word in the buffer is called the frontier word.

The algorithm proceeds by reducing the most recent pending word (the top of the stack) which means assigning it a governor. This governor is either the current frontier word, creating a left arc, or the previously most recent pending word (below in the stack), creating a right arc. Once given a governor, the most recent pending word is popped out. When the frontier word has been given all his left dependents, it is shifted which means it is pushed in the stack, becomes the most recent pending word and the next word in the sentence becomes the frontier word.

More formally, the arc-hybrid parsing algorithm is defined using configurations. Following standard definition, a configuration is a triplet $[\sigma, \beta, A]$ where σ is a stack of indexes of words which have not been given a governor yet, β a list (buffer) of indexes of words still to be read[1], and A the set of dependency arcs that have been constructed so far.

Given sentence $w = w_1, ..., w_n$ and dummy root token w_0, there are one initial configuration $c_1 = [0, 1..., \emptyset]$ and many goal configurations of the form $[0, \emptyset, A]$. There exist 3 transition rules to pass from one configuration to another[2]:

shift $[\sigma, b|\beta, A] \rightarrow^S [\sigma|b, \beta, A]$

left $[\sigma|d, h|\beta, A] \rightarrow^L [\sigma, h|\beta, A \cup \{(h, d)\}]$

right $[\sigma|h|d, \beta, A] \rightarrow^R [\sigma|h, \beta, A \cup \{(h, d)\}]$

A derivation for w is a sequence $\gamma = c_1, t_1, c_2, t_2, \ldots, t_{2n}, c_{2n+1}$ of $2n$ transitions from c_1 leading to a goal configuration c_{2n+1}.

The set of derivations for a sentence w is noted D_w. It can be shown that derivations all generate projective dependency trees and that, for a sentence with n words, they each contain n shift operations and n left or right reductions[3].

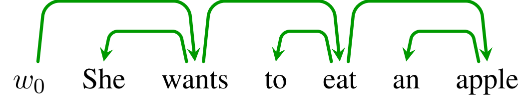
Shifts and reductions can be seen as forming a well-parenthesized expression. Each word is associated with a kind of parenthesis and shifting and reducing this word correspond to opening and closing parentheses of this kind.

[1] A buffer containing $(i, ..., n)$ will be denoted by "i..".

[2] Shift and left transitions require a non-empty buffer and the stack has to be of length at least 2 for left and right transitions since w_0 cannot be the governor of a left arc.

[3] Note that derivation is not unique for a dependency tree.

For instance, in *She wants to eat an apple*, the derivation $(_{\text{She}})^L(_{\text{wants}}(_{\text{to}})^L(_{\text{eat}}(_{\text{an}})^L(_{\text{apple}})^R)^R)^R$, where shifting a word is represented by a subscripted opening parenthesis and reductions are closing parentheses typed either L or R, will generate the following dependency tree:



Arc-hybrid parsing amounts to finding the highest-scoring derivation for a sentence:

$$\hat{\gamma} = \arg\max_{\gamma \in D_w} s(\gamma)$$

If we assume s decomposes over transition scores $s_\ell(c_\ell, t_\ell)$, we retrieve the well-studied cumulative sum of its transition scores.

$$s(\gamma) = \sum_{1 \leq \ell \leq 2n} s_\ell(c_\ell, t_\ell) \tag{1}$$

2.2 General Formulation for Dynamic Programming

We present a dynamic programming (DP) algorithm for arc-hybrid parsing with cumulative transition scores as in Eq. 1. This algorithm cannot be used as such since states references complete stack contents, of which there is an exponential number, leading to an intractable complexity. To use this algorithm in practice we would need to resort to beam search in order to approximate solutions, see for instance (Dyer et al., 2015).

However, this constitutes a general framework from which efficient algorithms can be derived by considering various equivalence classes over states and independence assumptions in the scoring function. For instance we can retrieve the *Minimal Feature Set* algorithm of Shi et al. (2017), or the new algorithm presented in Section 3.

In our algorithm, an item ${}^{\sigma,A}\langle \boldsymbol{i}, \boldsymbol{j}\rangle^B$ represents the following set of subsequences of a derivation:

$${}^{\sigma,A}\langle \boldsymbol{i}, \boldsymbol{j}\rangle^B : [\sigma, i..., A] \xrightarrow{*} [\sigma|i, j..., B]$$

i.e. subsequences which start with *shifting* w_i and end with w_i on top of the stack and w_j on top of the buffer[4]. As a special case, we will note ${}^{\emptyset,\emptyset}\langle \boldsymbol{0}, \boldsymbol{j}\rangle^B$ the set of subderivations starting from the initial configuration $[0, 1..., \emptyset]$ and leading to $[0, j..., B]$.

[4] such a subderivation is only possible if $i < j \leq n+1$ and $A \subset B$.

Goal: Goal items have the form: ${}^{\emptyset,\emptyset}\langle \boldsymbol{0}, \boldsymbol{n{+}1}\rangle^{A} : [\emptyset, 0..., \emptyset] \xrightarrow{*} [0, \emptyset, A]$.

Axioms: The algorithm starts with the set of items corresponding to possible shifts[5]: ${}^{\sigma,A}\langle \boldsymbol{i}, \boldsymbol{i{+}1}\rangle^{A} : [\sigma, i..., A] \rightarrow [\sigma|i, i{+}1..., A]$. The first axiom[6] ${}^{\emptyset,\emptyset}\langle \boldsymbol{0}, \boldsymbol{1}\rangle^{\emptyset}$ pictures a dummy subderivation that would put w_0 on top of the stack and lead to the initial configuration $[0, 1..., \emptyset]$.

DP Steps: A DP step consists in building an item ${}^{\sigma,A}\langle \boldsymbol{i}, \boldsymbol{j}\rangle^{B}$ by composing ${}^{\sigma,A}\langle \boldsymbol{i}, \boldsymbol{k}\rangle^{C}$, ${}^{\sigma|i,C}\langle \boldsymbol{k}, \boldsymbol{j}\rangle^{D}$, and a *reduce* operation on word w_k:

$$\begin{array}{rrcl} {}^{\sigma,A}\langle \boldsymbol{i}, \boldsymbol{k}\rangle^{C}: & [\sigma, i..., A] & \xrightarrow{*} & [\sigma|i, k..., C] \\ {}^{\sigma|i,C}\langle \boldsymbol{k}, \boldsymbol{j}\rangle^{D}: & [\sigma|i, k..., C] & \xrightarrow{*} & [\sigma|i|k, j..., D] \\ \textit{reduce}: & [\sigma|i|k, j..., D] & \rightarrow & [\sigma|i, j..., B] \end{array}$$

We thus have a *right reduction* rule:

$$\frac{{}^{\sigma,A}\langle \boldsymbol{i}, \boldsymbol{k}\rangle^{C} \quad {}^{\sigma|i,C}\langle \boldsymbol{k}, \boldsymbol{j}\rangle^{D}}{{}^{\sigma,A}\langle \boldsymbol{i}, \boldsymbol{j}\rangle^{D\cup\{(i,k)\}}} \qquad [(i \rightarrow k)]$$

and a *left reduction* rule:

$$\frac{{}^{\sigma,A}\langle \boldsymbol{i}, \boldsymbol{k}\rangle^{C} \quad {}^{\sigma|i,C}\langle \boldsymbol{k}, \boldsymbol{j}\rangle^{D}}{{}^{\sigma,A}\langle \boldsymbol{i}, \boldsymbol{j}\rangle^{D\cup\{(k,j)\}}} \qquad [(k \leftarrow j)]$$

Scoring Items: We trivially score an axiom ${}^{\sigma,A}\langle \boldsymbol{i}, \boldsymbol{i{+}1}\rangle^{A}$ with the score of a *shift* transition occurring in configuration $[\sigma, i..., A]$.

We compute the score of a DP step as the sum of the scores of the combined items and the score of *reducing* w_k from configuration $[\sigma|i|k, j..., D]$. When several DP steps produce the same item, it is assigned the highest score.

2.3 Minimal Feature Set Algorithm

For this algorithm (Shi et al., 2017), the score of a transition t_ℓ does not depend on the whole configuration but only on the index on top of the stack and the first index of the buffer. In other words, local scores s_ℓ only depend on word indexes (i,j,k). This assumption is crude but it allows for quite large items equivalence classes. We can retrieve this algorithm from the one above by removing unnecessary information in the items. Items of the form ${}^{\sigma,A}\langle \boldsymbol{i}, \boldsymbol{j}\rangle^{B}$ will simply reduce to $\langle \boldsymbol{i}, \boldsymbol{j}\rangle$, there will be $O(n^2)$ such items, and the DP complexity will be $O(n^3)$. More concretely, we have the following schemata:

[5]There is one axiom for each possible configuration $[\sigma, i..., A]$ with $0 \leq i < n$, σ a valid stack and A the set of corresponding arcs.

[6]Other axioms can be generated lazily from stack and arcs set pairs of items created by DP steps.

Goal: $\langle \boldsymbol{0}, \boldsymbol{n{+}1}\rangle$.

Axioms: $\langle \boldsymbol{i}, \boldsymbol{i{+}1}\rangle$.

DP Steps:

$$\frac{\langle \boldsymbol{i}, \boldsymbol{k}\rangle \quad \langle \boldsymbol{k}, \boldsymbol{j}\rangle}{\langle \boldsymbol{i}, \boldsymbol{j}\rangle} \qquad [(i \rightarrow k)]$$

and

$$\frac{\langle \boldsymbol{i}, \boldsymbol{k}\rangle \quad \langle \boldsymbol{k}, \boldsymbol{j}\rangle}{\langle \boldsymbol{i}, \boldsymbol{j}\rangle} \qquad [(k \leftarrow j)]$$

One of the issues with this parsing scheme is the difficulty to interpret item scores consistently. In the case of a left reduction (producing $k \leftarrow j$ above) the score of $\langle \boldsymbol{k}, \boldsymbol{j}\rangle$ can be interpreted as the score of word j being the governor of word k and the score of $\langle \boldsymbol{i}, \boldsymbol{k}\rangle$ as the score of shifting word k in the context of i being the most recent pending word.

On the contrary, in the case of a right reduction (producing $i \rightarrow k$ above) if the score of $\langle \boldsymbol{i}, \boldsymbol{k}\rangle$ may well be interpreted analogously as the score of having i as a governor of k, we cannot interpret $\langle \boldsymbol{k}, \boldsymbol{j}\rangle$ as the score of shifting k, and it may be difficult to interpret this item score in terms of a transition operation.

3 Parsing with Derivations and Dependencies

3.1 A New Score for Derivations

We depart form previous work and make the dependency arc contribution explicit in the score function:

$$\hat{\gamma}, \hat{\tau} = \underset{\gamma,\tau \in S_w}{\arg\max}\, s(\gamma) + s(\tau) \tag{2}$$

where S_w is the set of pairs (γ, τ) with $\gamma \in D_w$ and τ the dependency tree corresponding to γ. We use an arc-factored model for τ from now on and discuss scores for γ.

We define an equivalence class (i, q) containing all configurations c_ℓ such that the first index i_ℓ of β_ℓ equals i and the size of stack $|\sigma_\ell|$ equals q. Thus, we rewrite cumulative transition scores as:

$$s^{(u)}(\gamma) = \sum_{1 \leq \ell \leq 2n} s_u(i_\ell, |\sigma_\ell|, t_\ell). \tag{3}$$

We also consider a score function based on the nestedness property of shift/reduce derivations. A score is given to each pair consisting of a shift and its corresponding (left or right) reduction. In other

words, we exploit the perfect matching induced by a derivation between shift and reduce transitions: if t_ℓ is a shift transition and $t_{\ell'}$ its matching reduction, we consider a score depending on the equivalence classes of c_ℓ and $c_{\ell'+1}$. Note that the nestedness property implies that the stacks σ_ℓ and $\sigma_{\ell'+1}$ are equal. Moreover, $i_{\ell'+1} = i_{\ell'}$ as $t_{\ell'}$ is a reduction, an operation which does not modify the buffer. Denoting by M the set of matching shift/reduce pairs $(t_\ell, t_{\ell'})$ in γ, this gives:

$$s^{(m)}(\gamma) = \sum_{(t_\ell, t_{\ell'}) \in M} s_m(i_\ell, |\sigma_\ell|, i_{\ell'}). \quad (4)$$

Note that $s_m(i_\ell, |\sigma_\ell|, i_{\ell'})$ can be seen as the score of performing a reduction when i_ℓ is on the top of the stack of size $|\sigma_\ell|$ and $i_{\ell'}$ is the first word position of the buffer. Hence, this gives a score similar to the reduction score used in the minimal feature set algorithm. The difference is that the score takes into account the size of the stack but not the type (left or right) of reduction that is performed. The direction information will be given by the dependency arc score.

Finally, the derivation score is:

$$s(\gamma) = s^{(u)}(\gamma) + s^{(m)}(\gamma) \quad (5)$$

3.2 Graph Representation of a Derivation

In this section we represent the derivation of the arc-hybrid parsing using graphs.

A derivation is a sequence of n shifts and n reductions such that no more reductions than shifts are performed at each step. Such a sequence is a Dyck word and can then be represented as a path in an $(n+1) \times (n+1)$ grid starting at the lower left corner, ending at the lower right corner, using only up-right diagonal arcs and downward arcs (Roman, 2015). Such representation is the starting point of our derivation graph.

Define the *derivation graph* $G = (V, A)$ as follows. $V = \{v_i^q | 1 \leq q \leq i \leq n+1\}$ represents the set of equivalence classes. A vertex v_i^q corresponds to the class (i, q) of configurations, where w_i is the first word in the buffer[7] and the stack is of size q. The arc set A is given by $A = T \cup E$ where T represents transitions between states and E matches between shifts and reductions. We note $T = T^S \cup T^L \cup T^R$:

- $T^S = \{(v_i^q, v_{i+1}^{q+1}) | 1 \leq q \leq i \leq n\}$,
- $T^L = \{(v_i^q, v_i^{q-1})^L | 2 \leq q \leq i \leq n\}$,
- $T^R = \{(v_i^q, v_i^{q-1})^R | 2 \leq q \leq i \leq n+1\}$.

T^S, T^L and T^R represent shifts and tagged reductions respectively. We set $E = \{(v_i^q, v_j^q) | 1 \leq q \leq i < j \leq n+1\}$ because a shift can be matched to a reduction only if the size of the stack is the same before the shift and after the reduce. Note that for a sentence with n words, G will have $O(n^2)$ vertices and $O(n^3)$ arcs.

A derivation γ obtained by the arc-hybrid parsing will be represented by a pair (P, M) where $P \subseteq T$ is a path representing the derivation γ and $M \subseteq E$ is the set of arcs matching the shifts with their associated reductions in the derivation.

More formally, (P, M) is a solution if and only if we have the following. P is a path from v_1^1 to v_{n+1}^1 in G using only arcs of T. By construction, it contains n arcs of T^S and n arcs of $T^L \cup T^R$. It then corresponds to the sequence of transitions $t_1, \ldots, t_{2n}$. Note that any arc (v_i^q, v_{i+1}^{q+1}) in P consists in pushing word w_i on the top of the stack and any arc (v_i^q, v_i^{q-1}) in P consists in popping the top of the stack. One can retrieve the sequence of configurations $c_1, \ldots, c_{2n+1}$ thanks to the transitions.

Remark that each vertex v_i^q covered by P corresponds to configuration c_{2i-q} since $i-1$ shifts and $i-q$ reduces have been performed. An arc (v_i^q, v_j^q) of E belongs to M if the transition t_{2i-q} is a shift, transition t_{2j-q-1} is a reduction and these shift and reduce operations are matched together. Hence, $(v_i^q, v_j^q) \in M$ implies that (v_i^q, v_{i+1}^{q+1}) and (v_j^{q+1}, v_j^q) belong to P.

One can associate a score s_u from Eq. (3) with each arc of T and a score s_m from Eq. (4) with each arc of E. In this case, $s(\gamma)$ corresponds to

$$s(\gamma) = \sum_{a \in P} s_u(a) + \sum_{a \in M} s_m(a) \quad (6)$$

As an illustration, the derivation presented in the introduction can be represented by the set of black arcs P and red arcs M in Figure 1.

3.3 Dynamic Programming Algorithm

From the graph representation above we can derive a DP algorithm which computes the score of the optimal solution subgraph. This algorithm can be seen as a specialization of the general framework presented in Section 2.2 for our scoring functions.

[7] The value $n+1$ for i indicates that the buffer is empty.

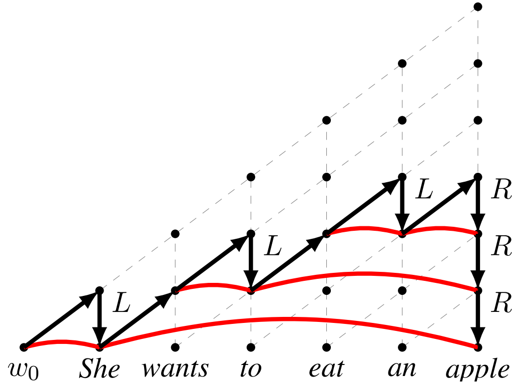


Figure 1: A Derivation for *She wants to eat an apple*. Shifts (resp. reductions) are represented in black diagonal (resp. vertical) arcs. Red curved edges represent matchings. Vertices are configuration classes v_i^q.

Since score functions s_u and s_m take the size of the stack into account, items with different (initial) stack sizes cannot be equivalent. Hence, items of the form ${}^{q}\langle \boldsymbol{i}, \boldsymbol{j} \rangle$ are needed, where $q = |\sigma|$ represents the size of the stack before shifting w_i. Leaving out the resulting dependency arcs, both *reduction* rules can be written with these equivalent items:

$$\frac{{}^{q}\langle \boldsymbol{i}, \boldsymbol{k} \rangle \quad {}^{q+1}\langle \boldsymbol{k}, \boldsymbol{j} \rangle}{{}^{q}\langle \boldsymbol{i}, \boldsymbol{j} \rangle}$$

There are $O(n^3)$ equivalent items and the DP complexity will be $O(n^4)$. Such items will be scored in the following way :

- the score of an axiom ${}^{q}\langle \boldsymbol{i}, \boldsymbol{i{+}1} \rangle$ is $s_u(i, q, \mathit{shift})$. Axioms appear as black diagonal arcs (v_i^q, v_{i+1}^{q+1}) in Figure 1.
- In a DP step, the score of the *reduce* transition is $s_u(j, q{+}2, \mathit{reduce})$. The transition is represented as a black vertical arc (v_j^{q+2}, v_j^{q+1}) in Figure 1.
- Finally the *matching* score in a DP step is $s_m(k, q{+}1, j)$. This score corresponds to red curved arcs (v_k^{q+1}, v_j^{q+1}) in Figure 1.

Figure 2 depicts both reduction rules. An item is represented as an arrow for the initial shift, and a triangle for the well-nested part of the subderivation. A reduction builds a new item by extending the well-nested part of the left antecedent with a new matching arc obtained from the right antecedent and the new reduction arc.

A dependency arc is added to assign a head to k, the midpoint of the reduction rule, depending on the direction of the reduction.

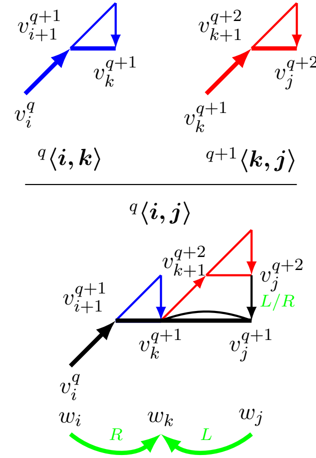


Figure 2: *Illustration of reduction rules.* Items are represented by a diagonal arc (first shift) and an horizontal edge (well-nested part). When combining two items in a reduction rule, a (curved) matching edge and a (vertical) reduction arc are added. The type of the reduction leads to a new dependency arc for the modifier w_k.

4 Learning Derivation Scores

We first present our network architecture inspired by recent work on self attention (Vaswani et al., 2017; Wang et al., 2017) which is able to learn representations of arc-hybrid configuration classes, that we call step representations. These representations are then used to compute derivation scores.

Our network is an encoder/decoder. The encoder computes word and step representations in the specific context of the sentence to be parsed, while the decoder computes arc scores.

We borrow from the transformer layer (Vaswani et al., 2017) the idea of global attention but extend it to the case where the size of the output is different from the size of the input. This has already been explored in (Gu et al., 2018) in the context of Machine Translation. However our problem is simpler because the size of the output is always twice the size of the input, in other words we do not have to estimate the size of the output.

4.1 Notation

A feed-forward layer is a sequence of an affine transformation, a ReLU filter and a linear transformation, *i.e.* $\text{FF}(\mathbf{x}) = V(\max(0, (W\mathbf{x} + b)))$, with V, W, b trainable parameters.

We call interpolation layers functions like $I(\mathbf{x}, \mathbf{y}) = C(\mathbf{x})\cdot\mathbf{x}+(1-C(\mathbf{x}))\cdot\mathbf{y}$ where C is a linear transformation followed by a sigmoid squashing, and $\cdot$ denotes the component-wise product.

Combining the previous two, we define highway layers (Greff et al., 2017) as functions $H(\mathbf{x}) = I(\mathbf{x}, FF(\mathbf{x}))$, i.e. an interpolation of input $\mathbf{x}$ and a feed-forward transformation of $\mathbf{x}$.

We also make use of biaffine functions following (Dozat and Manning, 2017) that we define as functions of the form $B(\mathbf{e}, \mathbf{f}) = \mathbf{e}^\top M\mathbf{f}+V\mathbf{e}$, with matrix M and vector V learnable parameters.

4.2 Layer Structure

Each layer L_i is the composition of two sublayers A_i, computing a generalized attention, and B_i, performing a feed-forward transformation. As is the case in previous approaches, each sublayer is followed by a layer normalization (Ba et al., 2016) and a residual connection to prevent underflows.

In more details, each layer takes as input a sequence of size n of dense vectors of size d packed as a matrix $\mathbf{X}$ in $\mathbb{R}^{n\times d}$ and a query vector sequence of size o, either equal to n or $2n$ depending on the layer (see infra) packed as a matrix $\mathbf{Y}$ in $\mathbb{R}^{o\times d}$. When $\mathbf{X}$ and $\mathbf{Y}$ are the same, we recover the self-attention mechanism of the transformer layer. The layer forward value is given by the following equations, where LN is a layer normalization:

$$\begin{aligned} a_i(\mathbf{X}, \mathbf{Y}) &= \mathbf{X} + A_i(\mathbf{X}, \mathbf{Y}) \\ b_i(\mathbf{X}) &= \mathbf{X} + B_i(\mathbf{X}) \\ L_i(\mathbf{X}, \mathbf{Y}) &= LN(b_i(LN(a_i(\mathbf{X}, \mathbf{Y})))). \end{aligned}$$

The first sublayer A_i computes for each output position in Y a multi-head scaled dot-product attention over input query Y and key/value X, with m attention heads.

$$A_i(\mathbf{X}, \mathbf{Y}) = \sum_{h=1}^{m}(A(Q_i^h\mathbf{Y}, K_i^h\mathbf{X}, V_i^h\mathbf{X}))$$

Each attention head A takes a query as input queries, keys and values, and computes for each query vector a convex combination of value vectors, the coefficients of which are given by an operation between the query and the key vectors:

$$A(\mathbf{Q}, \mathbf{V}, \mathbf{K}) = \mathrm{softmax}(\mu\mathbf{Q}\mathbf{K}^\top)\mathbf{V}$$

where softmax is applied row-wise and μ is a smoothing factor between 0 and 1, which is set to $d^{-0.5}$, where d is the size of a query vector, following previous implementations of dot-product attention (Luong et al., 2015).

The second sublayer B_i applies the same feed-forward transformation to each element of the sequence of vectors returned by A_i.

4.3 Word and Position Embeddings

Each word in the train set is associated with a real vector stored in a lookup table E. In order to cope with unseen words, we follow (Kiperwasser and Goldberg, 2016) and at training time words are randomly UNK-ed with a probability inversely proportional to their frequency in the train set.

Contrarily to recurrent networks such as LSTMs, attention networks do not have a built-in notion of position so it must be provided externally. In our systems, we have two types of positions, namely word positions and step positions. We use position embeddings stored in lookup tables called respectively T and S.

4.4 Word Encoder and Dependency Scores

Our encoder is the composition of e self-attention layers starting from word and position vectors.

$$\begin{aligned} X_0 &= [E(w_1) + T(1); \ldots; E(w_n) + T(n)] \\ X_i &= L_i(X_{i-1}, X_{i-1}), 1 \le i \le e \end{aligned}$$

After encoding, we get X_e that we interpret as a sequence of contextualized word vectors. We can use these vectors to predict arc scores. In the following, we use a biaffine function B_{dep} to define the raw score between head at position i and modifier at position j: $s_{w_i\to w_j} = B_{\text{dep}}(X_e[i], X_e[j])$.

These raw scores are used in two ways.First and most obviously they score dependency arcs of the derived tree in this model. Second, for each modifier we use incoming arc scores to weigh potential heads and compute an expected head vector. We use normalized scores via softmax to interpolate head vectors. As a result for each vector word $X_e[i]$ we obtain an expected head vector $Y_e[i]$, which will be used hereafter.

4.5 Step Encoder

For steps, we distinguish the first layer from the others. For the first layer, input is the sequence returned by the last word encoder noted X_e and expected heads Y_e, and the query is initialized with

the increasing sequence of valid step position embeddings called $P = [S(1); \ldots ; S(2n)]$.

$$D_0 = X_e + Y_e,\ P_1 = P,\ D_1 = L_{e+1}(D_0, P_1)$$

For $k > 1$, we set $D_k = L_{e+k}(D_{k-1}, D_{k-1})$.

Given sentence w, we note $h(w)$ the sequence of vectors returned by the last decoder layer.

4.6 Decoders as Local Classifiers

In this section we present how the score function can be decomposed as local probabilities. Given a sentence w, we assume the probability of a derivation γ is conditioned upon its corresponding derived tree τ. This condition prevents inconsistencies between τ and γ but plays no other role in the scoring function. The probability of a derived tree decomposes as independent head predictions computed by a logistic regression over head scores.

$$\begin{aligned} p(\gamma, \tau|w) =& p(\gamma|w, \tau) \times p(\tau|w) \\ =& p(\gamma|w, \tau) \times \prod_{h \to m \in \tau} p(h|m, w) \end{aligned}$$

The probability of a derivation in D_w is the probability at each independent step ℓ of 2 events: the transition t_ℓ and the step position r_ℓ of the corresponding reduction for a shift, or a dummy position for a reduction. We consider these two events to be independent but requiring the knowledge of the difference q_ℓ between the number of shifts and reductions already performed before the current step. This difference can also be interpreted in the context of the arc-hybrid algorithm as the depth of the stack. The difference is then used as a parameter for the potentials. This gives:

$$\begin{aligned} p(\gamma|w) &= \prod_{\ell=1}^{2|w|} p(t_\ell, r_\ell, q_\ell|w, \ell) \\ &= \prod_{\ell=1}^{2|w|} p_d(q_\ell|w, \ell) \times p(t_\ell, r_\ell|w, \ell, q_\ell) \\ &= \prod_{\ell=1}^{2|w|} p_d(q_\ell|w, \ell) \\ &\qquad \times p_u(t_\ell|w, \ell, q_\ell) \times p_m(r_\ell|w, \ell, q_\ell) \end{aligned}$$

The condition on sentence and step index w, ℓ is implemented via functions taking the ℓth step representation of w computed as described in the previous section, denoted $h(w)_\ell$ or simply h_ℓ if the sentence is clear from the context.

In practice we restrict the set of values for q_ℓ as the set of natural numbers between one and nine, and a special value for differences greater or equal to ten[8]. We use a lookup table F to convert discrete difference values to dense representations.

Note that although the 3 distributions are conditioned on the same step representations introduced in the previous section, these representations are first passed through highway layers, $\{H_i\}_{i=d,u,m}$, parameterized for each distribution. This helps with the specialization of step representations while keeping the possibility to share information between tasks.

The first two distributions are categorical distributions of the exponential family. They are computed as normalized potentials given by feed-forward transformations of steps and differences.

$$\begin{aligned} p_d(q|h_\ell) &\propto \exp \mathrm{FF}_d(F(q) + H_d(h_\ell)), \\ p_u(t|q, h_\ell) &\propto \exp \mathrm{FF}_t(F(q) + H_u(h_\ell)). \end{aligned}$$

The third distribution is computed with a bi-affine function B_m followed by a softmax, as in (Dozat and Manning, 2017). We reserve an extra value for the result random variable which encodes the absence of corresponding reduction.This is used when s is not a shift step. Note the difference embedding is only used on the left side of M.

$$p_m(r|h_\ell, q) \propto \exp B_m(F(q) + H_m(h_\ell), h_{\ell+r})$$

Learning is performed by simply maximizing the conditional log-likelihood of the 4 distributions over the correct derivations given a set of sentences. Once parameterized, it is straightforward to see how these distributions can fit the score model of Equation 5.

Conditional log-likelihood minimization requires to compute values for each distribution along gold solutions, which means it is a $O(n^2)$ procedure, because of distributions on dependency arcs and matching transitions, which both require 2 position parameters in order to be computed.

5 Experiments

Data We ran experiments on the Wall Street sections of the English Penn Treebank (Marcus

[8] We found almost all train sentences could be parsed with stack size below 10.

POS embedding size	28
Other embedding size	100
Encoder input/output size	256
Decoder input/output size	256
Hidden layer size in B subnetworks	512
attention heads	8
Maximum step numbers	200
Maximum difference D (stack height)	10
Number of word encoder layers	4
Number of step encode layers	4

Table 1: Network hyperparameters

Setting	UAS
Ours	**94.82** $\pm$ 0.10
Ours, joint training, but decoding with	
dependency scores only	94.80 $\pm$ 0.09
derivation scores only	93.95
Ours, training and decoding with	
dependency scores only	94.73 $\pm$ 0.06
derivation scores only (no stack size)	84.81
(Shi et al., 2017) best local (4 features)	93.89
(Shi et al., 2017) global (2 features)	94.43
(Shi et al., 2017) global Eisner	94.50
(Kiperwasser and Goldberg, 2016) greedy	93.8

Table 2: Comparisons on PTB test set

et al., 1994) converted to Stanford Dependencies (de Marneffe and Manning, 2008). Transition sequences were obtained from dependencies and in case of ambiguity right reductions were always performed before shift if possible. We followed the standard split (02-21 for training, 23 for testing and 22 for development purposes) and used POS tags predicted by the Stanford MaxEnt tagger trained using 10-way jackknifing (Toutanova et al., 2003). We evaluate on unlabeled attachment without punctuation, with CoNLL evaluation script.

Implementation Hyperparameters are given in Table 1. In addition to word embeddings parameterized on the PTB, we use pretrained Glove vectors (Pennington et al., 2014). Our prototype is written in c++ with DyNet[9] for neural computations (Adam optimizer and default values) and UDPipe[10] for reading and writing data files. Minibatches contain around 1,000 tokens. We train each model for 100 epochs and select our best model according to its development UAS. We follow previous works with transformers to set the learning rate (Vaswani et al., 2017). For the first $8,000$ updates the learning increases linearly with the number of steps, then it decreases proportionally to the squared root of the number of steps. We use the formula of Strubell et al. (2018).

Results Results on the PTB test set are presented in Table 2 and comparisons with previous work on arc-hybrid parsing with comparable network sizes. Parsing results are obtained by averaging 5 models initialized with different random seeds and standard deviation is also provided. Our system reaches 94.8% UAS and parses the whole section 23 in 1.13 seconds (DP only) on an Intel Xeon 2.10 GHz. Although our system is trained via local classifiers, we can see that it improves over global systems trained without step encodings.

Ablations indicate that the major part of scoring comes from dependencies. We may also conclude that (i) derivation information is useful per se but most importantly as an auxiliary task to improve dependencies and (ii) the stack size is paramount in this model since otherwise the network has no indication of the stack content. If not considered, accuracy drop considerably: step indexes alone are too vague as they can correspond to many different stack and buffer contents.

6 Discussion and Related Work

Derivation Parsing Maximum subgraph selection has played a central role in dependency parsing since the MST reduction by McDonald et al. (2005) and can also be traced back to the parsing-as-intersection tradition in phrase-based parsing – see for instance (Billot and Lang, 1989) – where the goal is to find, starting from a generic grammar, a graph-structure (a shared forest) that recognizes the input presented as a string or an automaton. In dependency parsing, this approach has since been extended to more complex dependencies such as non-crossing and 1-endpoint-crossing dependencies (Kuhlmann and Jonsson, 2015; Cao et al., 2017).

There is a long line of research which solve the different variants of transition-based dependency parsing algorithms with dynamic programming. Recent work showed that this can be performed efficiently, in $O(n^3)$, for arc-hybrid parsers (Gómez-Rodríguez et al., 2008) and have since been extended with non-linear classifiers (Kiperwasser and Goldberg, 2016; Shi et al., 2017) to reach state-of-the-art parsing accuracy.

We depart from both in the following way. In most works on maximum subgraph selection, the class of valid subgraph is a class defined only by

[9] https://github.com/clab/dynet
[10] http://ufal.mff.cuni.cz/udpipe

properties on dependencies. Here we represent *derivations* instead of derived structures. In that respect, we are closer to approaches developed for mildly-context sensitive formalisms such as Tree Adjoining Grammars which work primarily on the derivation tree (Corro et al., 2017) and consider the derived tree, i.e. the parse structure, as a by-product. Compared to other dynamic programming approaches to arc-hybrid parsing, we therefore work on a richer model, and have more expressive power to take a representation of states into account in the scoring scheme. This comes at a cost since the time complexity of our parsing algorithm is $O(n^4)$, an order of magnitude higher. The stack information (size) is minimal and is used to parameterize access to information available from step embeddings.

Compared to joint parsing systems working on both constituents and dependencies, our approach doesn't require external linguistic knowledge such as head percolation rules. On the other hand, since derivations don't add new information, but merely offer a new vision of the problem, the potential accuracy gain is lower.

Machine Learning Aspects Self-attention networks have been used in parsing, see for instance (Kitaev and Klein, 2018), whether based on dependencies or syntagms. Curiously we found few models of transition-based parsing based on these networks, and bidirectional recurrent network are still preferred in most architectures, where they are believed to capture some information about the sequential nature of transition-based algorithms. Instead we present a non-sequential model of transition-based parsing where representation vectors are obtained via unrolled iterative estimation (Greff et al., 2017).

Our encoder-decoder architecture together with independence assumptions made in the probabilistic model which decomposes a derivation score in several subtasks can be seen as auxiliary tasks as in (Coavoux et al., 2018).

The use of expected head vectors as input of the step encoder is related to the syntactic head attention of the SRL neural architecture in (Strubell et al., 2018).

7 Conclusion

We presented the arc-hybrid parsing transition rule system as a subgraph selection problem and showed how this can be solved exactly by a dynamic programming algorithm. This theoretical result is backed up by state-of-the-art results on the PTB.

This new representation of the problem is the basis of a novel neural architecture which learns vertex representations (for derivation steps) and edge scores (for derivation features).

From a parsing perspective, understanding why derivation prediction is a good auxiliary task to learn syntactic dependencies could prove insightful to explain how transitions and dependencies are related.

The derivation/derived pair is a very powerful concept that remains to be fully exploited. In particular, there are two promising avenues for future improvements. First, the learning framework could be enriched in a setting where the derivation graph is modeled as a latent variable and marginalized over. It remains to be seen if this can be done exactly or if sampling is required for efficiency. Second, since the score of a solution is the sum of the scores of elements in a pair, it should be possible to design an approximate solver based on lagrangian decomposition more efficient in practice.

Acknowledgments

This work is partially supported by a public grant overseen by the French National Research Agency (ANR) as part of the program *Investissements d'Avenir* (ANR-10-LABX-0083). It contributes to the IdEx Université de Paris (ANR-18-IDEX-0001). This work is partially supported by a public grant overseen by the French ANR (ANR-16-CE33-0021).

References

Lei Jimmy Ba, Ryan Kiros, and Geoffrey E. Hinton. 2016. Layer normalization. *CoRR*, abs/1607.06450.

Sylvie Billot and Bernard Lang. 1989. The structure of shared forests in ambiguous parsing. In *Proceedings of the 27th annual meeting on Association for Computational Linguistics*, pages 143–151. Association for Computational Linguistics.

Junjie Cao, Sheng Huang, Weiwei Sun, and Xiaojun Wan. 2017. Quasi-second-order parsing for 1-endpoint-crossing, pagenumber-2 graphs. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 24–34. Association for Computational Linguistics.

Maximin Coavoux, Shashi Narayan, and Shay B. Cohen. 2018. Privacy-preserving neural representa-

tions of text. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1–10. Association for Computational Linguistics.

Caio Corro, Joseph Le Roux, and Mathieu Lacroix. 2017. Efficient discontinuous phrase-structure parsing via the generalized maximum spanning arborescence. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1644–1654. Association for Computational Linguistics.

Timothy Dozat and Christopher D. Manning. 2017. Deep biaffine attention for neural dependency parsing. In *International Conference on Learning Representations*.

Chris Dyer, Miguel Ballesteros, Wang Ling, Austin Matthews, and Noah A. Smith. 2015. Transition-based dependency parsing with stack long short-term memory. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 334–343, Beijing, China. Association for Computational Linguistics.

Jason M. Eisner. 1996. Three new probabilistic models for dependency parsing: An exploration. In *COLING 1996 Volume 1: The 16th International Conference on Computational Linguistics*.

Carlos Gómez-Rodríguez, John Carroll, and David Weir. 2008. A deductive approach to dependency parsing. In *Proceedings of ACL-08: HLT*, pages 968–976, Columbus, Ohio. Association for Computational Linguistics.

Klaus Greff, Rupesh Kumar Srivastava, and Jürgen Schmidhuber. 2017. Highway and residual networks learn unrolled iterative estimation. In *International Conference on Learning Representations*.

Jiatao Gu, James Bradbury, Caiming Xiong, Victor OK Li, and Richard Socher. 2018. Non-autoregressive neural machine translation. In *International Conference on Learning Representations*.

Eliyahu Kiperwasser and Yoav Goldberg. 2016. Simple and accurate dependency parsing using bidirectional lstm feature representations. *Transactions of the Association for Computational Linguistics*, 4:313–327.

Nikita Kitaev and Dan Klein. 2018. Constituency parsing with a self-attentive encoder. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2676–2686. Association for Computational Linguistics.

Marco Kuhlmann, Carlos Gómez-Rodríguez, and Giorgio Satta. 2011. Dynamic programming algorithms for transition-based dependency parsers. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 673–682, Portland, Oregon, USA. Association for Computational Linguistics.

Marco Kuhlmann and Peter Jonsson. 2015. Parsing to noncrossing dependency graphs. *Transactions of the Association for Computational Linguistics*, 3:559–570.

Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1412–1421. Association for Computational Linguistics.

Mitchell Marcus, Grace Kim, Mary Ann Marcinkiewicz, Robert MacIntyre, Ann Bies, Mark Ferguson, Karen Katz, and Britta Schasberger. 1994. The penn treebank: annotating predicate argument structure. In *HLT'94: Proceedings of the workshop on Human Language Technology*, pages 114–119, Morristown, NJ, USA. Association for Computational Linguistics.

Marie-Catherine de Marneffe and Christopher D. Manning. 2008. Stanford typed dependencies manual. Technical report, Stanford University.

Ryan McDonald, Fernando Pereira, Kiril Ribarov, and Jan Hajic. 2005. Non-projective dependency parsing using spanning tree algorithms. In *Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing*, pages 523–530, Vancouver, British Columbia, Canada. Association for Computational Linguistics.

Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543. Association for Computational Linguistics.

Steven Roman. 2015. *An Introduction to Catalan Numbers*, 1st edition. Birkhäuser Basel.

Tianze Shi, Liang Huang, and Lillian Lee. 2017. Fast(er) exact decoding and global training for transition-based dependency parsing via a minimal feature set. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 12–23. Association for Computational Linguistics.

Emma Strubell, Patrick Verga, Daniel Andor, David Weiss, and Andrew McCallum. 2018. Linguistically-informed self-attention for semantic role labeling. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 5027–5038. Association for Computational Linguistics.

Kristina Toutanova, Dan Klein, Christopher D. Manning, and Yoram Singer. 2003. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of the 2003 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008.

Xiaolong Wang, Ross Girshick, Abhinav Gupta, and Kaiming He. 2017. Non-local neural networks. *arXiv preprint arXiv:1711.07971*, 10.