Natural Language Understanding, Generation and Machine Translation - Week 8 - Parsing

Antonio León Villares

April 2023

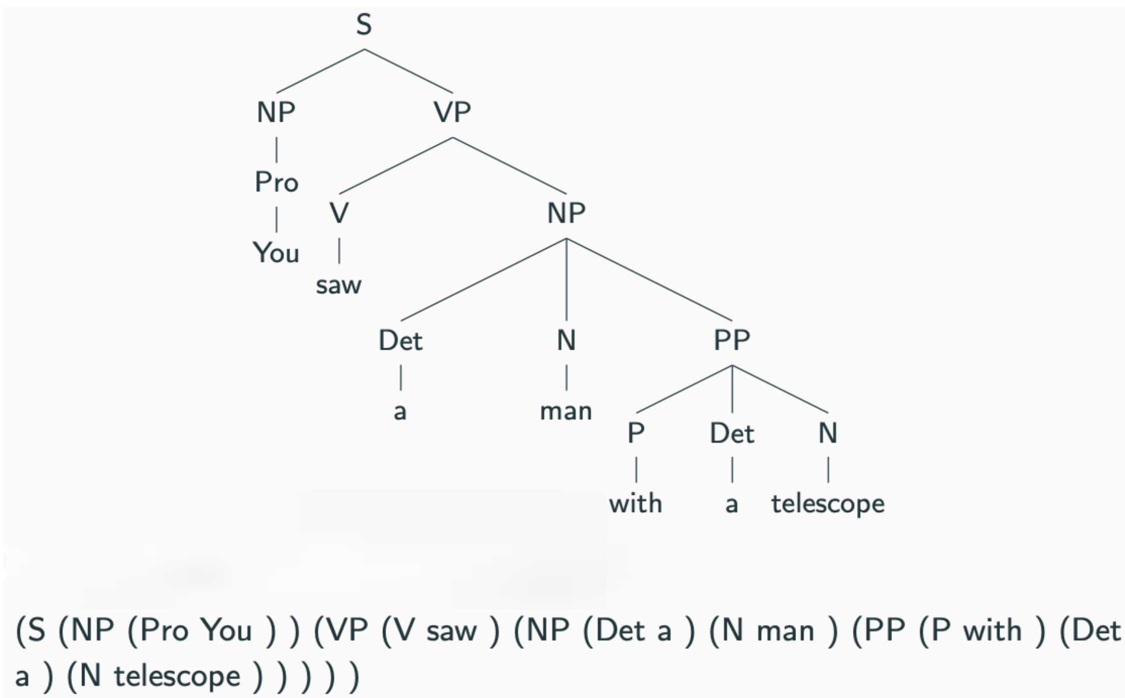
Contents

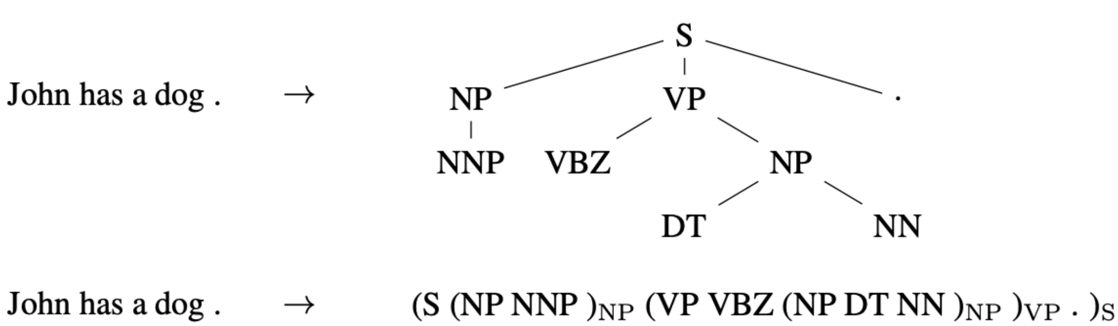
Based on:

- *Grammar as a Foreign Language, by Vinyals et al.*
- *Constituency Parsing with a Self-Attentive Encoder, by Kitaev and Klein*
- *Unsupervised Parsing via Constituency Tests, by Cao et al.*
- *Unsupervised Recurrent neural Network Grammars, by Yoon et al.*
- *Movie Summarisation via Sparse Graph Construction, by Papalampidi, Keller and Lapata*

1 Parsing with Encoder-Decoder Networks

- **How can encoder-decoder networks be used for parsing?**
  - generally, **encoder-decoder networks** can be used for **any** task, where we operate over **sequences of symbols**
  - in **parsing**, we generally operate over **parse trees**
  - however, **parse trees** can be easily **linearised**





- this converts a **parse tree** into a **sequence of symbols**, over which we can operate

- **In general, what other structures can encoder-decoder networks operate over?**
  - any **structured representation** which can be **linearised** into a **symbol sequence** is fair game:
    - formal languages (i.e Python)
    - databases
    - tables
    - graphs
    - images
  - whether **encoder-decoder networks** are suitable for this is another question altogether

2 Neural Parsing with LSTMs

Here we discuss the encoder-decoder network proposed by Vinyals et al. in Grammar as a Foreign Language, which is based on LSTMs.

2.1 Architecture

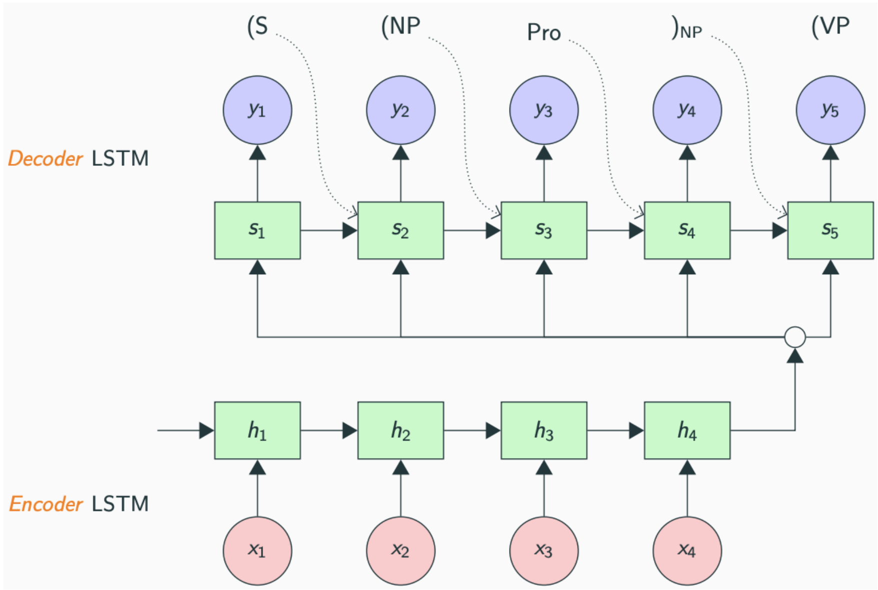
- **How are parse trees further processed for the LSTM network?**
  - to **simplify** the sequence, we can remove the **terminals**:

    (S (NP Pro) (VP V (NP Det N (PP P Det N))))

  - when we parse out the sequence, we can always **fill in** the gaps with the words in order. For example, when the LSTM outputs `Pro )` as the first branch of the tree, we can assume that this corresponds to the first word of the input sequence: `Pro You)`
  - to make **closing brackets** easier to recognise by the network, we can **annotate** them:

    $(S\ (NP\ Pro\ )_{NP}\ (VP\ V\ (NP\ Det\ N\ (PP\ P\ Det\ N\ )_{PP}\ )_{NP}\ )_{VP}\ )_{S}$

- **What is the structure of the encoder-decoder network?**
  - the **basic structure** uses the typical **encoder-decoder** LSTM architecture:



- however, they made the following observations to **improve** on this:
  1. **End of Sequence Symbol**: every **output parse tree** is terminated with an **end-of-sequence token** (since we need to delimit where these variable length sequences end)
  2. **Reverse Input String**: if the input string is reversed ("John has a dog." → ". dog a has John"), there is a small performance improvement (the **parse tree** isn't reversed)
  3. **Deeper Network**: 3 LSTM encoder-decoder blocks were used
  4. **Attention**: was incorporated between **encoder** and **decoder**
  5. **POS Replacement**: all POS tags were replaced with `XX`, which surprisingly improved F_1 performance
  6. **Pretrained Embeddings**: `word2vec` embeddings were used as inputs
  7. **Large Amount of Training Data**: without a lot of **training data**, the model performed poorly (it requires a lot of data to correctly gauge the idiosyncrasies which define **parse trees**)
- the **final architecture** looked thus:

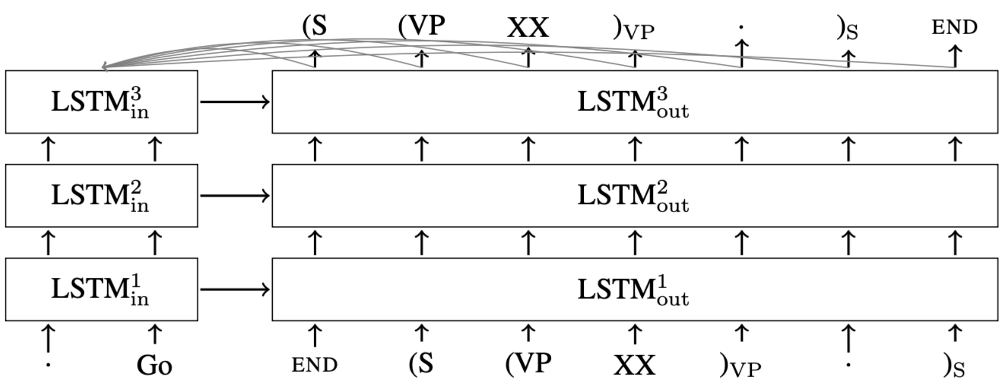


Figure 1: Processing the input sentence "Go.".

2.2 Results

- **What potential problems could this architecture have?**

1. **Invalid Trees**: it could be the case that the **output trees** are wrong (for instance, if there is a **mismatch** in the number of brackets). However, in practice it was found that this only occurred in 0.8-1.5% of the sentences. Even if it were to widely occur, this could be easily fixed in post processing.
2. **Generating the Best Tree**: it could be the case that this network, whilst outputting the **best symbol** at each step doesn't necessarily generate the **best tree**. This can be fixed by using **beam search**, much like in **machine translation**, to ensure that the output tree is (heuristically) the best.

- **What different corpora were used for training?**

1. `WSJ`: treebank with $\approx 40k$ **manually annotated sentences** (these are **gold-labels**)
2. `BerkeleyParser Corpus`: $\approx 90k$ sentences from `WSJ` and several other treebanks + $\approx 7M$ sentences from news appearing on the web, tagged by using the high quality `BerkeleyParser`
3. `High-Confidence Corpus`: $\approx 90k$ sentences from `WSJ` and several other treebanks + $\approx 11M$ sentences from news appearing on the web, tagged by using 2 high quality parsers (`BerkeleyParser`, `ZPar`). This includes only those sentences in which both parsers agreed on the tree, and they re-sampled to match the distribution of sentence lengths of the WSJ training corpus (since shorter sentence lengths are easier to parse).

- **What results did the LSTM encoder-decoder achieve?**

Parser	Training Set	WSJ 22	WSJ 23
baseline LSTM+D	WSJ only	< 70	< 70
LSTM+A+D	WSJ only	88.7	88.3
LSTM+A+D ensemble	WSJ only	90.7	90.5
baseline LSTM	BerkeleyParser corpus	91.0	90.5
LSTM+A	high-confidence corpus	93.3	92.5
LSTM+A ensemble	high-confidence corpus	**93.5**	**92.8**
Petrov et al. (2006) [12]	WSJ only	91.1	90.4
Zhu et al. (2013) [13]	WSJ only	N/A	90.4
Petrov et al. (2010) ensemble [14]	WSJ only	92.5	91.8
Zhu et al. (2013) [13]	semi-supervised	N/A	91.3
Huang & Harper (2009) [15]	semi-supervised	N/A	91.3
McClosky et al. (2006) [16]	semi-supervised	92.4	92.1
Huang & Harper (2010) ensemble [17]	semi-supervised	92.8	92.4

Figure 2: F_1 scores for neural parsing using LSTMs. `+D` indicates that **dropout** was used. The results in the lower half of the table are for parsing performed with different iterations of the `BerkeleyParser`. Note that the current state of the art is at 95-96.

- notice, using just `WSJ` with a naive encoder-decoder results in poor results; however, upon adding **attention** or using **ensemble methods** (train a bunch of LSTMs, and use a classifier to decide on tree) leads to significant performance improvements
- using the larger datasets for training also leads to performance improvements (even above those obtained by the high-quality parsers)

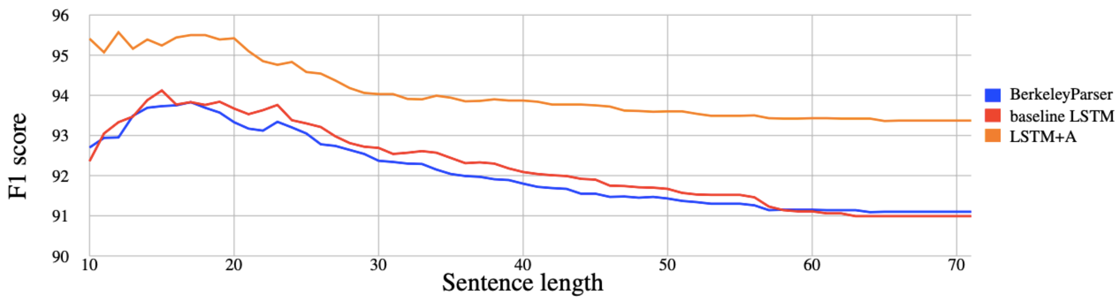


Figure 3: Here, the `BerkeleyParser`, baseline LSTM and LSTM with attention were evaluated one a single sentence, for sentences of varying lengths. Notice, by just using attention we obtain significantly higher F_1 scores, particularly as sentence length increases. Moreover, the performance degradation seems to be lower for the LSTM with attention as sentence length increases.

- **Why is this such an impressive result?**
  - the LSTM was nothing special: it is a general encoder-decoder model
  - it nonetheless seems to capture syntactic structure very well, and can "understand" the structure of the trees (i.e closing brackets), without any explicit change to its architecture
  - its performance rivals, and even outperforms, that of specialised systems, like `BerkeleyParser`
  - moreover, it'll be much faster at **inference** (a **probabilistic chart parser**, such as CYK, has $\mathcal{O}(n^3)$ complexity)
- **What information does the attention matrix revel with regards to how the LSTM is processing the input sentence to generate the parse tree?**

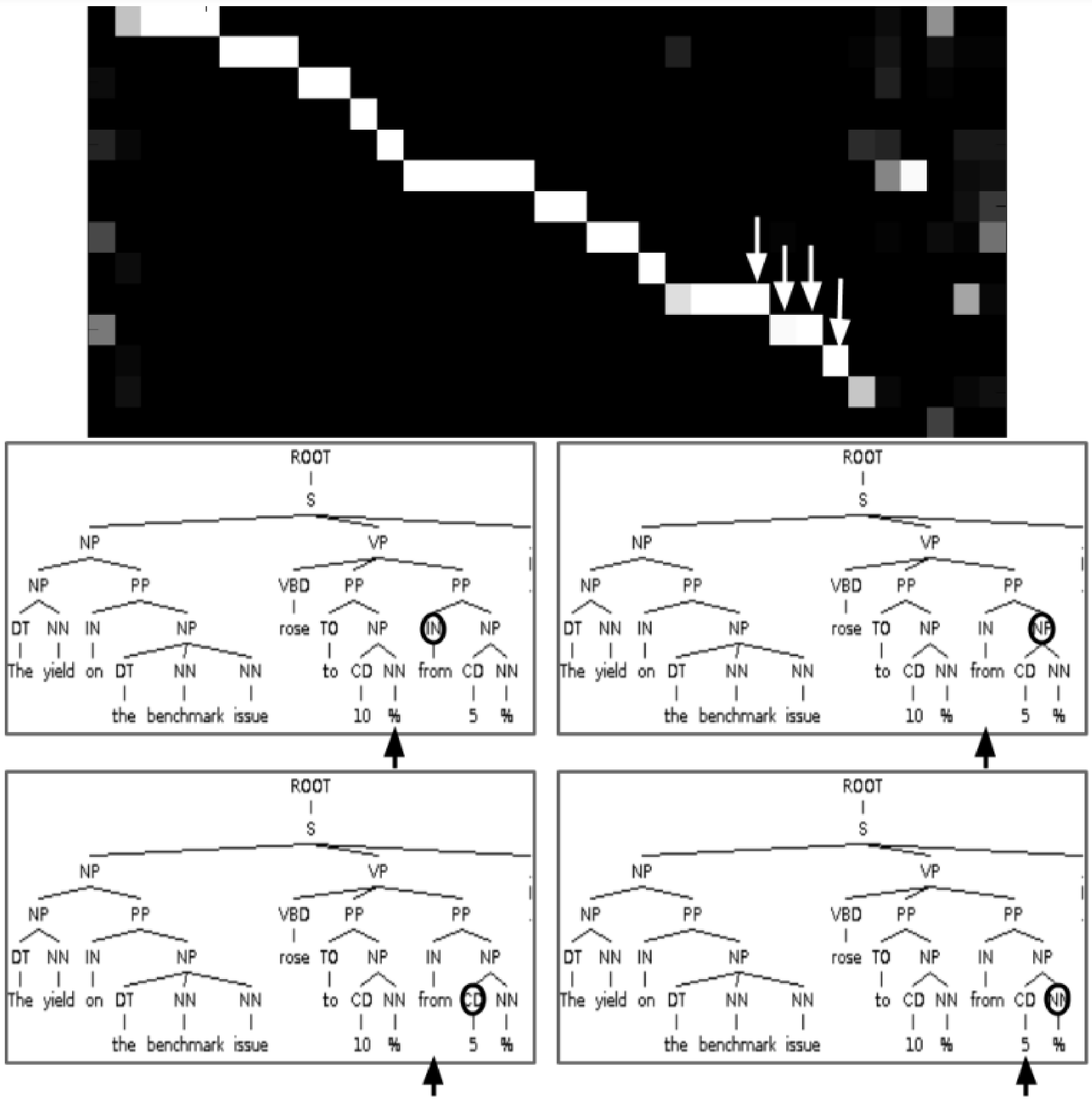


Figure 4: Circles on nodes denote the current output being decoded in the tree, whereas the arrow denotes where the model is attending. This showcases 4 consecutive steps in the decoding process. The columns each denote the attention vector for each word in the input.

- this shows how the model focuses on a **single word** as it is decoding
- it considers the input sequence **monotonically**, from left to right
- sometimes the model skips words
- each time a **terminal** is **consumed**, the attention pointer moves to the right

3 Neural Parsing with Transformers

Here we discuss the transformer-based neural parser developed in Constituency Parsing with a Self-Attentive Encoder

3.1 Architecture

- **Why are transformers well-suited for neural parsing?**
  - **self-attention** is an integral part of transformers
  - this allows models to understand a word, based on which words it attends to from the input
  - thus, the transformer learns which sort of relations exist between input words, which provides a more **natural** way of understanding the **syntactic relationship** between these words
  - this is greatly beneficial, as this model requires **less training data** than the LSTM-based model, and obtains **better results**
- **What are the 3 main features of the transformer-based neural parser?**
  1. **Factored Attention Heads**: these generate separate representations for **position** and **content** information. This results in a **context aware summary vector**, which encompasses word, POS tag and position information.
  2. **Span Scores**: the embedding layers are combined to produced **span scores**: they assign scores to "chunks" of words (i.e "I have a dog", then there'd be attention scores for "I", "I have", "have a", "have a dog", etc...)
  3. **Decoding**: for decoding, the span scores are used alongside **CYK** (an efficient parsing algorithm, see my FNLP notes) to generate the output tree. This ensures that generated trees are always valid.

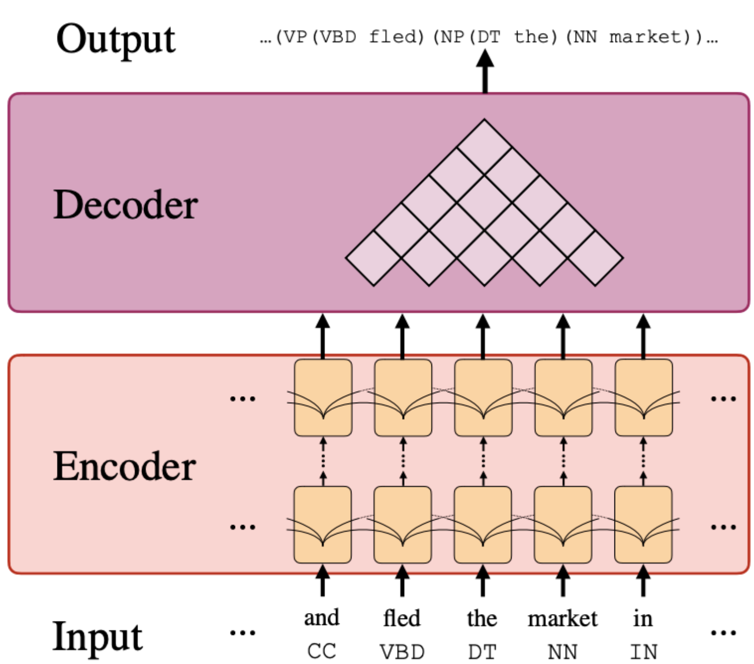
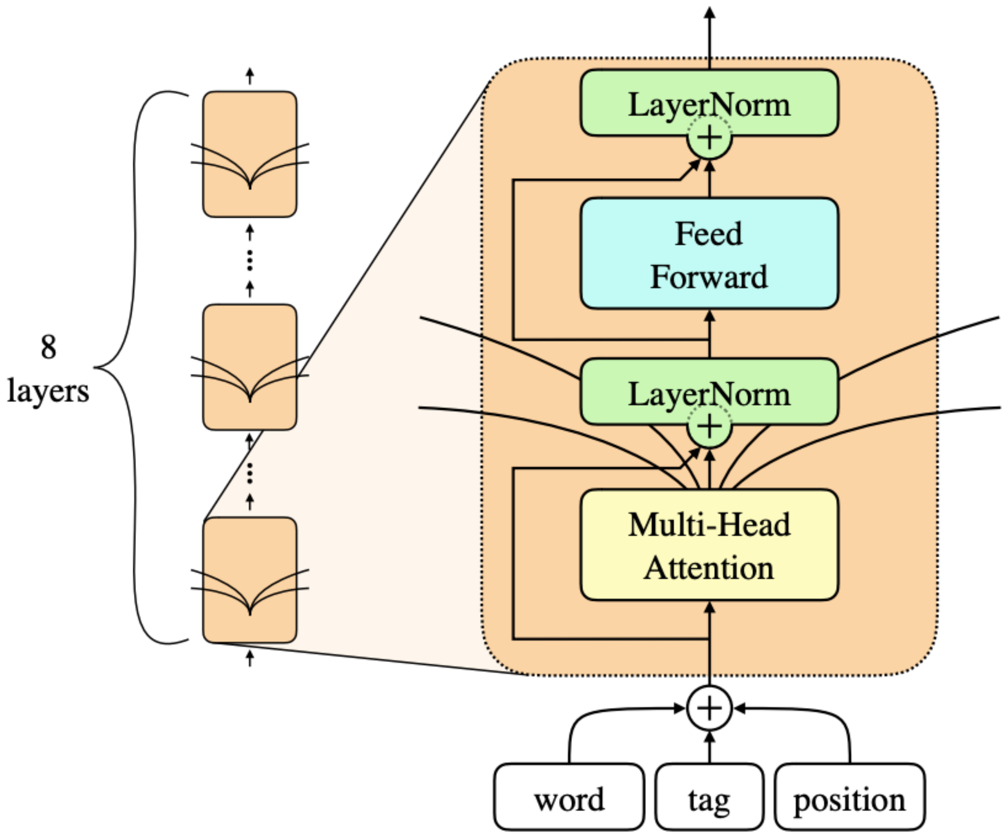


Figure 5: The **transformer** parser **encodes** using the **self-attention** mechanisms, and generates a parse tree by using a **chart decoder**. Notice, here POS tags are used, and special start/end tokens are appended to each input sequence.

3.1.1 Factored Attention Heads

- **How are factored attention heads used for neural parsing?**
  - the **transformer parser** is composed by 8 **transformer blocks**



  - each **transformer block** uses **factored self-attention heads**

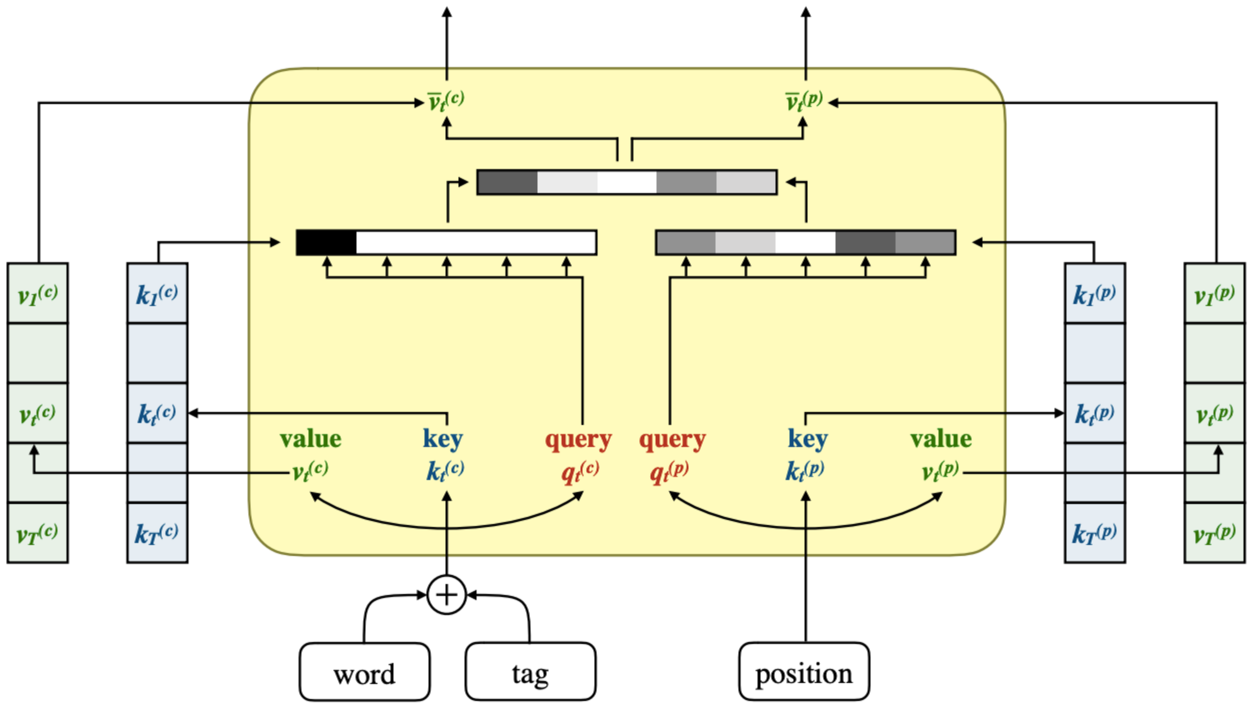


Figure 6: Sketch of how **factored self-attention heads** are employed in the parser. **Factored attention** is a common design pattern for transformer models.

- the idea is to learn separate representations for **content** (word and tag) and **position** information
- **position** information is very important for parsing, since the model needs to understand span of texts and the interplay of words at different position
- **word**, **tag** and **position** are encoded by using separate **embeddings** (learnt independently)
- a **self-attention distribution** is learnt for both the **word+tag** and **position** encodings, by using separate query and key vectors
- the distributions are combined to generate a **single** attentional distribution
- finally, separate representations for the 2 types of informations are produced, by using the different value vectors, alongside the unified attentional distribution
- the **context aware summary vector** is obtained by **concatenating** these 2 **content** and **position** representations

- **What benefits does using factored self-attention provide?**

  1. **Interpretability**: attentional results are more **interpretable**, since we can obtain separate distributions for **content** and **position** information
  2. **Reduced Complexity**: the model will require **less parameters** (no longer have to model interactions between the 2 types of information; the paper makes it much clearer, but essentially we learn block-sparse parameter matrices)

3.1.2 Computing Span Scores

- **How can span scores be computed from the context aware summary vectors?**

  - let $\underline{y}_k$ be the **context aware summary vector** for the kth input word
  - it can be split into 2 parts:

  - $\overrightarrow{\underline{y}}_k$ for the upper half
  - $\overleftarrow{\underline{y}}_k$ for the lower half

- we can compute a **span vector** for a span (i, j) via:

$$\underline{v} = [\overrightarrow{\underline{y}}_j - \overrightarrow{\underline{y}}_i; \overleftarrow{\underline{y}}_{j+1} - \overleftarrow{\underline{y}}_{i+1}]$$

  where the first half of the span vector corresponds to **content information** of the words defining the span, whilst the second half of the span vector corresponds to **position information** of the words after the words defining the span

- this is purely heuristic: the context words tend to have useful information for defining a good constituent span
- to compute the **span score**, a small network is used:

$$s(i, j, \cdot) = M_2\texttt{ReLU}(\texttt{LayerNorm}(M_1\underline{v} + \underline{c}_1)) + \underline{c}_2$$

  where $M_1, M_2, \underline{c}_1, \underline{c}_2$ are learnable parameters.

3.1.3 Decoding

- **How is decoding performed in this model?**
  - say we have a candiate **parse tree** T
  - we can assign a **score** to T, by scoring each of its **constituents**:

$$s(T) = \sum_{(i,j,I) \in T} s(i, j, I)$$

    where $s(i, j, I)$ is the **span score** for a **constituent** located between position i and j with label I (i.e POS tags, NP, VP, D, etc ...)

  - at inference, select the tree which obtains the **highest score**:

$$\hat{T} = \underset{T}{argmax}\ s(T)$$

  - we can efficiently compute this using CYK, which will give an **optimal output sequence** (unlike with **beam search** for LSTMs)

3.2 Results

- **How well does the transformer parser perform?**
  - this **transformer parser** was trained **solely** on the WSJ corpus
  - a bunch of additions were compared for the model (using **factored self-attention**, using `CharLSTM` (which generates embeddings for the POS tags) and using `ELMo` (a pre-trained language model similar to `BERT`) to generate the word, position and tag embeddings)
  - the transformer model was also compared with other neural parsers

Encoder Architecture	F1 (dev)	Δ
LSTM (Gaddy et al., 2018)	92.24	-0.43
Self-attentive (Section 2)	92.67	0.00
+ Factored (Section 3)	93.15	0.48
+ CharLSTM (Section 5.1)	93.61	0.94
+ ELMo (Section 5.2)	95.21	2.54

	LR	LP	F1
Single model, WSJ only			
Vinyals et al. (2015)	–	–	88.3
Cross and Huang (2016)	90.5	92.1	91.3
Gaddy et al. (2018)	91.76	92.41	92.08
Stern et al. (2017b)	92.57	92.56	92.56
Ours (CharLSTM)	**93.20**	**93.90**	**93.55**

Figure 7: Results for the transformer-based neural parser. It obtains significantly higher test set performance than all previous models. Notice the drastic difference in development performance when using a pretrained language model to generate embeddings.

4 Unsupervised Parsing

4.1 The Need for Unsupervised Parsing

- **Why are annotated treebanks inconvenient?**
  - **expensive**
  - **cumbersome** to create
  - **unavailable** for most languages
  - requires **trained syntactician** to annotate them
- **Why would unsupervised parsing be useful?**
  1. **Treebank Scarcity**: **annotated treebanks** are rare, and difficult to generate; on the other hand, unannotated text data is freely available
  2. **Low-Resource**: there are only treebanks in a few dozen languages - but there are around 6,000 total languages (many of which barely have any online data, let alone written data)
  3. **Preliminary Annotation**: **unsupervised parsers** can be used to **preliminarily** annotate treebank data, allowing us to generate **larger** treebanks
- **What are the challenges with unsupervised parsing?**
  1. **Gold's Theorem**: in simple terms, implies that a full **grammar** for a **natural language** can't be learned just from **raw text**; this makes **unsupervised parsing** particularly challenging. However, children are capable of gauging **syntactic structure** with little supervision (using text/speech/images, but definitely no parse trees), which indicates that decent unsupervised models can be potentially learnt.
  2. **Evaluation**: it is unclear how to evaluate unsupervised models, particularly when parsing, since even syntacticians might disagree on a particular parse tree. This can be somewhat fixed if we take **annotated treebanks** as "gold labels".

4.2 Unsupervised Parsing via Constituency Tests

We discuss the model proposed by Cao et al. in Unsupervised Parsing via Constituency Tests.

- **How can constituency tests be used for unsupervised parsing?**
  - **constituency tests** allow us to determine whether a **span** of words form a **constituent**
  - these are based on the fact that long spans of words can often times be **substituted** by shorter forms:

    "John's friend bought a book, but John's friend didn't read the book

    ↓

    "John's friend bought a book, but he didn't read it
  - **constituents** can be used to **segment** sentences into **spans**, such as NP, VP, D, etc..., which can then be used to generate a **parse tree**
- **Why are pre-trained language models important for unsupervised parsing?**
  - when we perform **constituency tests**, we need to verify that we have produced a **grammatical** sentence
  - hopefully, **pretrained language models** have an **inherent syntactic structure**, which can be used to assess grammaticality in an unsupervised manner

4.2.1 Constituency Tests

- **What is a constituent?**
  - a **constituent** refers to a syntactic unit that is composed of one or more words and functions as a single unit within a sentence
- **What are constituent tests?**
  - a set of tests (involving substitution and replacement) which constituents must pass (see this for some other examples)
  - for this paper, they used 5 **constituency tests**
  - for example, if given the sentence:

    "by midday, the London market was in full retreat"

    and we wanted to check whether "the London market" is a **constituent**, we'd break the sentence into parts at either side of the span:
    * A : "by midday"
    * B : "the London market"
    * C : "was in full retreat"

    and we'd apply:

Name	Applied to "A [B] C"	Example
Clefting	it {is, was} B that A C	*it {is, was} the london market that by midday , was in full retreat*
Coordination	A B and B C	*by midday , the london market and the london market was in full retreat*
Substitution	A {it, ones, did so} C	*by midday , {it, ones, did so} was in full retreat*
Front Movement	B , A C	*the london market , by midday , was in full retreat*
End Movement	A C B	*by midday , was in full retreat the london market*

- this shows that “the London market” is indeed a constituent, since the examples are **grammatical**
- on the other hand, if we tried using “market was” as a constituent, we’d obtain nonsensical examples. For example, with substitution, we’d obtain:

“by midday, the London **it/ones/did so** in full retreat ”

4.2.2 Grammaticality Model

- **What is the purpose of the grammaticality model?**
  - testing if after applying a **constituency test**, we obtain a **grammatical sentence**
  - for this, we can use **pre-trained language models**
- **How can we use the pre-trained models to assess gramamticality?**
  - a bunch of non-ideas:
    1. **Perplexity**: if a sentence has **high perplexity** according to the model, this could indicate low grammaticality. In practice, the difference between gramamtical and ungrammatical sentences isn’t too significant.
    2. **Prompting**: we can **prompt** the model (i.e “is this sentence grammatical”). However, **prompting** hadn’t been developed when this paper was developed (although nowadays with `ChatGPT` we can easily ask if a sentence is grammatical or not)
    3. **Incremental Prediction**: we can feed chunks of the sentence to the model, and see if it generates an output similar to that of the sentence
  - however, in the paper they opted for **fine-tuning** the **pre-trained language model** for the task of predicting wehther a sentence was grammatical or not
  - as we’ll discuss, they had to further enhance the model via a **refinement step** in order to get good results
- **How can we train the grammaticality model, if we don’t have data on whether a sentence is grammatical or not?**
  - the key idea is that we can **generate examples of ungrammatical sentences**
  - we have a plethora of (unannotated) sentences, which we know are **grammatical**
  - we can apply a series of **corruptions** to the sentence, which will make it **ungrammatical**
  - we can then fine-tune the LM to predict whether a sentence was real or predicted
  - for this task, they used 5M sentences in English from the Gigaword dataset; they fine-tuned `RoBERTa`, a variant of `BERT`

Name	Description
Shuffle	Choose a random subset of words in the sentence and randomly permute them.
Swap	Choose two words and swap them.
Drop	Choose a random subset of words in the sentence and drop them.
Span Drop	Choose a random contiguous span of words and drop it.
Span Movement	Choose a random contiguous span of words and move it to the front or back.
Bigram	Generate a sentence of the same length using a bigram language model trained on the source corpus.

4.2.3 Parsing Algorithm

- **How does the unsupervised parsing algorithm operate?**
  - we define 2 functions:
    * a **transformation function**, which given a **sentence span**, applies **constituency tests** to the sentence:
    $$\texttt{c} : (\texttt{sent}, i, j) \mapsto \texttt{sent'}$$
    * a **judgement function**, which given a **sentence**, determines whether it is **grammatical** or not (this has parameters θ):
    $$g_\theta : \texttt{sent} \mapsto [0, 1]$$
  - then, the **unsupervised parsing** algorithm works as follow:
    1. For each sentence, we can consider a **span** (i, j)
    2. If C is our set of **constituency tests**, we evaluate the span on each $c \in C$, and average the **grammaticality score** of the results:
    $$s_\theta(\texttt{sent}, i, j) = \frac{1}{|C|} \sum_{c \in C} g_\theta(\texttt{c}(\texttt{sent}, i, j))$$
    3. We can **score** a given **tree**, by adding the **scores** of it **spans**, and then choose the **highest scoring** binary tree by using CYK:
    $$t^*(\texttt{sent}) = \underset{t \in T(\texttt{len(sent)})}{argmax} \sum_{(i,j) \in t} s_\theta(\texttt{sent}, i, j)$$
    where $T(\texttt{len(sent)})$ denotes the set of binary trees with `len(sent)` leaves

4.2.4 Refining the Grammaticality Model

- **Why is refining the grammaticality model important?**
  - whilst the **grammaticality model** works well with the **true/corrupted** prediction task, its scores aren't fully accurate
  - for instance, it sometimes thought that some spans were invalid, simply because it was very sure that other spans were a lot more valid
  - moreover, sometimes spans tended to have fairly similar scores, which made syntactic discrimination difficult
  - with **refinement**, they were capable of obtaining more reliable span scores
- **How does refinement work?**
  1. Select a batch B of sentences, and parse them
  2. Treat the generated trees as **pseudo gold-labels** (i.e correct parse trees). We can define an **optimisation** problem, by maximising the **binary cross entropy** over span scores:
  $$\sum_{(i,j) \in t^*(\texttt{sent})} \log(s_\theta(\texttt{sent}, i, j)) + \sum_{(i,j) \notin t^*(\texttt{sent})} \log(1 - s_\theta(\texttt{sent}, i, j))$$
  According to the paper/slides, this induces that **spans** in the generated trees (i.e **constituents**) get a higher **grammaticality score**, whereas spans outside of the tree will get a reduced **grammaticality score**
  3. Repeat for the next batch of sentences. Over time, g_θ will learn to be more discriminative about the features which make a span more or less grammatical.

- **What additional benefit does refinement have?**
  - **refinement** operates over **all** spans in the tree
  - hence, we can learn about **wider context** (i.e how all spans compose together to generate a tree) as opposed to only focusing on assessing the suitability of a single span

4.2.5 Results

- **How well does the unsupervised parser perform?**
  - performance was compared with:
    - previous unsupervised neural models
    - baselines, where branching was monotonic (i.e each sentence was parsed as a left branching, balanced or right branching tree)
  - since this model can only parse **binary trees** (due to CYK), the percentage of binary trees is also included, to signal what the highest possible score could've been

Model	PTB F1 Mean	PTB F1 Max
PRPN† (Shen et al., 2018)	37.4	38.1
URNNG (Kim et al., 2019b)	–	45.4
ON† (Shen et al., 2019)	47.7	49.4
Neural PCFG† (Kim et al., 2019a)	50.8	52.6
DIORA (Drozdov et al., 2019)	–	58.9
Compound PCFG† (Kim et al., 2019a)	55.2	60.1
Left Branching	8.7	
Balanced	18.5	
Right Branching	39.5	
Ours (before refinement)	48.2	
Ours (after refinement)	**62.8**	**65.9**
Oracle Binary Trees	84.3	

Figure 8: Both mean and max over a number of runs is reported (since these models can be sensitive to initialisation). Notice, there is a drastic performance improvement upon including refinement. The highest F_1 score that could've been achieved is 84.3. Notice how poor these results are when compared to supervised parsing, where we obtain 90+ performance.

- **How were these results further improved?**
  - **Unsupervised RNNG** is a variant of Recurrent Neural Network Grammar proposed by Kim et al. in Unsupervised Recurrent Neural Network Grammars
  - URNNG can be trained by using trees from some other **unsupervised parser**, followed by fine-tuning with an LM objective
  - combining URNNG with this unsupervised parser leads to results comparable to **supervised models**

Model	PTB F1 Initial (Max)	+URNNG
PRPN	47.9	51.6
ON	50.0	55.1
Neural PCFG	52.6	58.7
Compound PCFG	60.1	66.9
Ours (after refinement)	65.9	**71.3**
Supervised Binary RNNG	71.9	72.8

4.2.6 Visualising Parses: Before and After Refinement and After Refinement + URNNG