Perspectives on neural proof nets

Richard Moot

Perspectives on neural proof nets

Richard Moot

CNRS, LIRMM, Univeristé de Montpellier

1 Introduction

Proof nets are a way of representing proofs as a type of (hyper)graph. Originally introduced for linear logic (Girard 1987), proof nets can be seen as a parallelised sequent calculus which removes inessential rule permutations, but also as a multi-conclusion natural deduction which simplifies many of the logical rules (notably the $\multimap E$, $\bullet E$, $\diamond E$ rules). This make proof nets a good choice for automated theorem proving: avoiding needless rule permutations entails an important reduction of the search space for proofs (compared to sequent calculus, and to a somewhat lesser extent when compared to natural deduction) but still allows us to compute the lambda terms corresponding to our proofs: enumerating all different proof nets for a sequent is equivalent to enumerating all its different lambda terms.

Proof nets can be adapted to different types of type-logical grammars while preserving their good logical properties (Moot 2021). This makes them an important tool for testing the predictions of different grammars written in type-logical formalisms. Combining the lambda terms produced by proof search with lambda terms assigned to words in the lexicon places type-logical grammars in the formal semantics tradition initiated by Montague (Montague 1974, Moortgat 1997).

In this paper I will present two different ways of combining proof net proof search with neural networks, using two different ways to split the task into two subtasks. The first approach is the 'standard' approach which has been applied to proof search in type-logical grammars in various different forms (Kogkalidis, Moortgat & Moot 2020*b*, De Pourtales, Rabault, Kogkalidis & Moot 2023). Since this approach has been discussed in other places, I will only present it briefly as a way to contrast it with the second, novel approach, which is the main topic of the paper.

The rest of this paper is structured as follows. Section 2 presents the standard view on proof nets for natural language processing, and introduces proof nets in the style of Moot & Puite (2002), which are quite close to natural deduction proofs, but which allow structures to have multiple conclusions. Section 3 shows how the standard approach maps nicely to a neural network architecture: following Kogkalidis et al. (2020*b*), we first map words to formulas, then use a

neural network to find the correct matching between atomic formulas in a way similar to a resolution proof. Section 4 proposes an alternative way to split the task into two: first we generate the graph structure in a way which guarantees it corresponds to a lambda-term, then we obtain the detailed structure using vertex labelling. Vertex labelling is a well-studied task in graph neural networks, so most of Section 4 will be about ways to implement graph generation using neural networks. Some work is needed to fit this task into one of the standard graph neural network paradigms but two possibilities are explored (with a third in Appendix A). Section 5 concludes. Unfortunately, we need to leave the implementation and evaluation of these new strategies to future research.

2 Proof nets

The 'standard' approach to using proof nets for natural language processing is the following.

1. *Lexical lookup* The lexicon maps words to formulas. In general, a word can have multiple formulas assigned to it, so this step is fundamentally non-deterministic.
2. *Unfold* We write down the formula decomposition tree.
3. *Match* We enumerate the 1-1 matchings between atomic formulas of opposite polarity, similar to the way a resolution proof links atoms with their negations.
4. *Check correctness* We check whether the resulting structure satisfies the correctness conditions.

We will return to these different steps below. This approach is standard because it corresponds naturally to the way we write our grammars in type-logical grammars: the grammar writer provides a lexicon and then tests his grammar on sentences using the words in the lexicon. The task of the theorem prover is then to find all different proofs/lambda terms, and this allows type-logical grammarians to test the predictions of their grammars by computing the different meanings of the sentences in their fragment.

To keep the discussion simple we will start with proof nets for the non-associative Lambek calculus NL. Table 1 shows the links for the two implications.

The table give the names of the corresponding natural deduction rule for each of the links. For the elimination (or *tensor*) links, the correspondence with the modus ponens rule is quite immediate: for $/E$ we have a formula C/B, that is a formula which looks for a B to its right to form a C, and we have a formula B to its right, therefore we conclude C. Similarly, the formula $A\backslash C$ combines with an A to its left to form a C.

The introduction (or *par*) links are somewhat more difficult to explain. We start by remarking that the par links are up-down symmetric with the corresponding tensor link. The way to read the $/I$ link is that we connect it to a

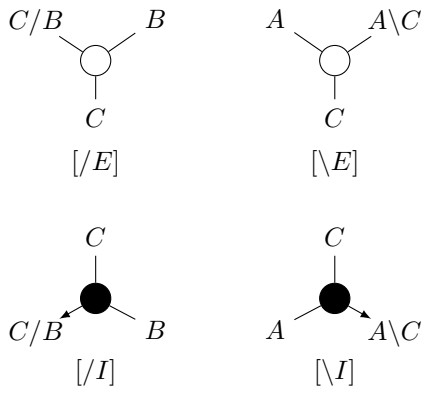


Table 1: Links for proof nets

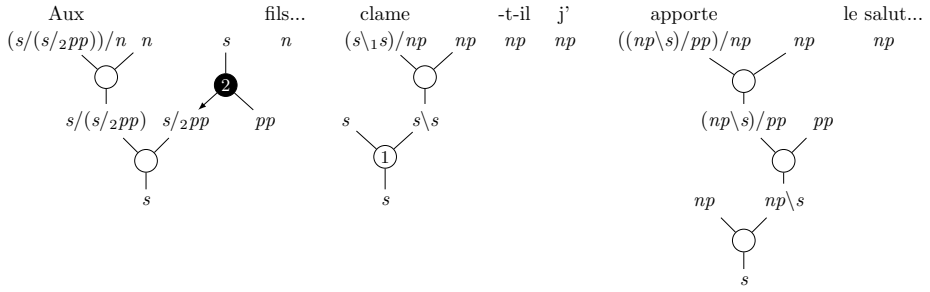


Figure 1: Formula unfolding

structure with conclusion C and hypothesis B to obtain a structure with conclusion C/B. This way the co-indexing between hypotheses and introduction rule in natural deduction rules is replaced by an edge in the hypergraph.

Figure 1 shows a (slightly simplified) example unfolding for the sentence.

(1) Aux fils de la Révolution mexicaine, claime-t-il, j'apport le
To the sons of the revolution Mexican, he exclaimed, I bring the
salut fraternel des fils de la Révolution française.
salvation brotherly of the sons of the revolution French.
To the sons of the Mexican revolution, he exclaimed, I bring the brotherly salvation of the sons of the French revolution.

In the simplified version of Figure 1, the noun "fils de la Révolution mexicaine" has been abbreviated as "fils...". Similarly, "le salut fraternel des fils de la Révolution française" has been abbreviated as the noun phrase "le salut...".

We can verify that each local neighbourhood is an instantiation of one of the links from Table 1. Given a formula, like $((np\backslash s)/pp)/np$ for "apporte"

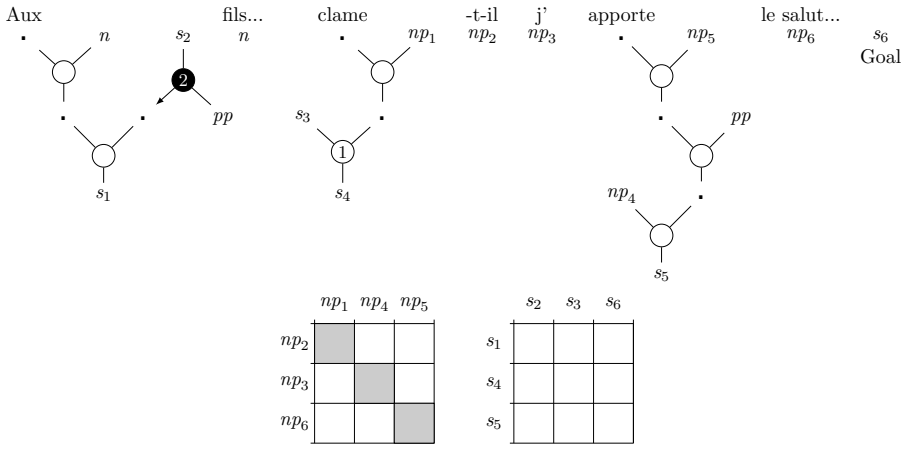


Figure 2: Abstract proof structure of Figure 1, with matrices representing the possible *np* and *s* matchings.

the unfolding is completely deterministic. We simply apply the logical links until we arrive at the atomic formulas. Some of the links have labels inside of the central circle — 2 for the $/I$ link of "Aux", 1 for the bottom most $\backslash E$ link of "clame". These labels are ways to mark certain connections as special; technically, they allows us to introduce controlled versions of structural rules for so-called multimodal versions of the Lambek calculus (Moortgat 1996) and these extensions can be incorporated into the proof net calculus we present here without problems (Moot & Puite 2002). However, the complete formal development of multimodal proof nets is not important for what follows, and the only thing to remember is that the labels can permit some additional operations.

We can simplify Figure 1 further by removing all complex formulas from the vertices, leaving only the atomic formulas. This simplification is information preserving: given that the 'root' of each component is indicated by the corresponding word, we can determine the corresponding link for each local neighbourhood, then compute the formulas from the atomic leaves up towards the lexical root. The structure shown in Figure 2 is called an abstract proof structure. In the figure, we have explicitly added the goal formula *s* as well. In addition, the atomic *np* and *s* formulas have been given integer subscripts. These subscripts are not formally part of the structure and are only there to give us a simple way to refer to the different atomic formulas.

Proof search now consists of matching the negative formulas — formulas which are not connected from below such as the formula np_2 of "-t-il" and *n* of "fils..." — with the positive formulas — formulas which are not connected from the top, such as np_4 and np_5 of "apporte" and the np_1 formula of "clame". The two square matrices on the bottom of Figure 2 summarise the possible matchings for the *np* and *s* formulas (the *n* and *pp* formulas have only a single possibility

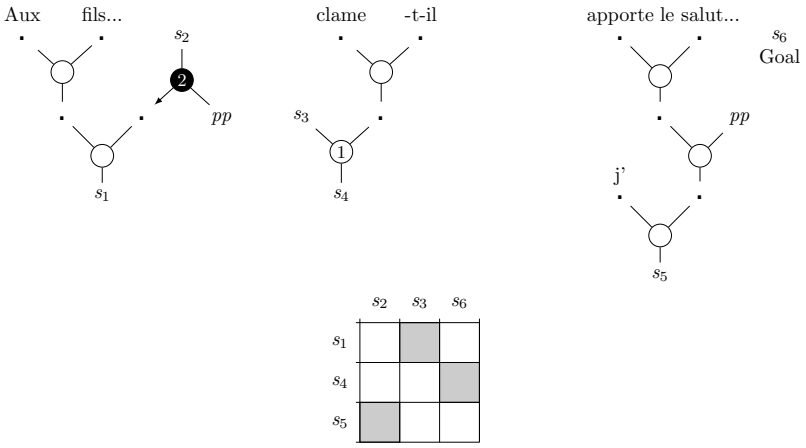


Figure 3: Abstract proof structure of Figure 2 after identification of the nouns and noun phrases.

in the structure). The negative (conclusion) atomic formulas are shown at the row labels, the positive (hypothesis) atomic formulas are shown at the column labels. A valid matching is a 1-1 matching between rows (negative atoms) and columns (positive atoms).

The simplification of "fils..." and "le salut..." reduces the combinatorics of proof search by quite a bit. For example, there are only two nouns left in the structure, the one of "fils..." and the one of "Aux" so there is only one way to connect the n formulas, as opposed to $11! = 39\,916\,800$ for the complete sentence. We can similarly use the information about the word order to connect the noun phrase of "-t-il" to "clame" and the two noun phrases "j" and "le salut..." to "apporte" as subject and object respectively. We should be careful about this kind of identification in the presence of structural rules, which can change the structure of the trees: we will see later how such a rewrite will allow us to put "clame-t-il" in the desired position.

The cells shaded grey (bottom left of Figure 2) show the correct vertex identifications for the np formulas: vertex np_1 with vertex np_2, np_3 with np_4 and np_5 with np_6.

Performing the vertex identifications for the nouns and noun phrases produces the abstract proof structure shown in Figure 3. For the s vertices, there are three possibilities for s_5 corresponding to the tree "j'apport le salut pp". We can identify it with the goal formula s_6, connect it to the tree for "clame-t-il" at s_3, or to the tree for "aux fils..." at s_2. In the current context, we prefer the identification $s_5 - s_2$ and the identification $s_6 - s_4$ as shown on the left of Figure 4.

For connecting the pp vertices, there is only one possibility, shown on the right of Figure 4. The curved line plays the same role as the co-indexing of hy-

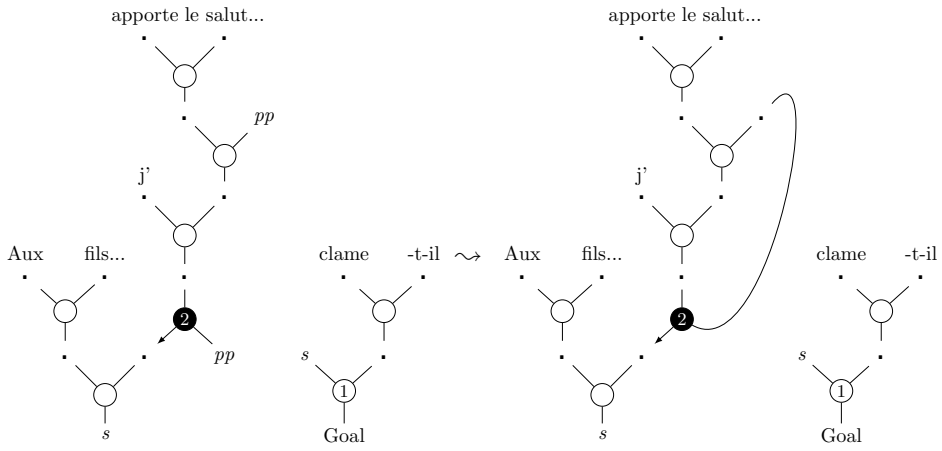


Figure 4: Connecting the *pp* hypothesis

potheses and introduction rules in natural deduction proofs. The *pp* hypothesis in the abstract proof structure on the left of the figure is withdrawn, and the curved line connects it to the introduction rule (represented by the filled circle marked '2') withdrawing it.

We complete the matching stage by connecting "clame-t-il" to "Aux fils ... j'apport le salut...". In this matching, "clame-t-il" takes the rest of the sentence as its argument, which corresponds to the intended reading for the complete sentence, shown as Figure 5.

It is important to note here that the graph of Figure 5 is uniquely determined by the formula unfolding of Figure 2 together with the matching of the atomic formulas, and therefore uniquely determined by just the sequence of formulas together with the matching of the atoms.

2.1 Contractions

While proof structures are good for representing the search space in automated theorem proving contexts — proof search amounts simply to finding matchings of atomic formulas — proof structures do not necessarily correspond to proofs. Following Moot & Puite (2002), we will define proof nets as proof structures whose abstract proof structures contract to a tree.

We will present a simplified version of the contraction condition from Moot & Puite (2002), which is nonetheless powerful enough for the rest of this paper. The simplest contractions are those for the non-associative Lambek calculus NL shown in Figure 6.

The contractions have the same general pattern. They take a pair of links — one corresponding to an elimination (tensor) rule and one corresponding to an introduction (par) rule — connected to each other at two vertices. The vertex

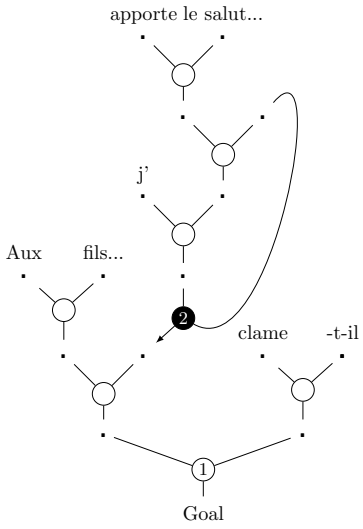


Figure 5: Continuing from the abstract proof structure of Figure 4 to obtain the completed linking for the abstract proof structure of Figure 2

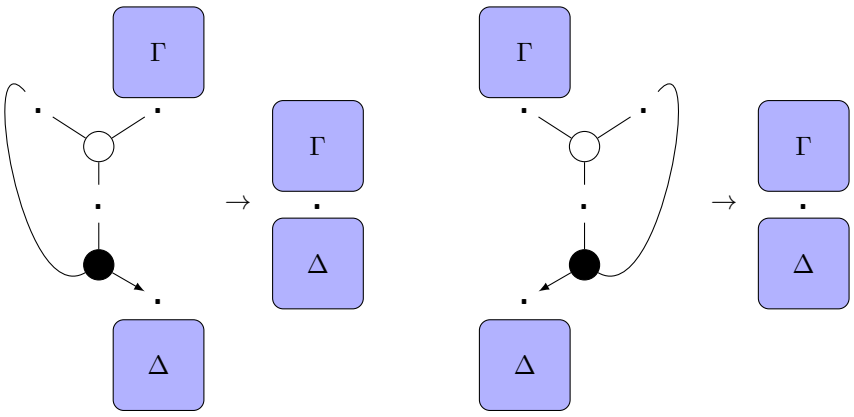


Figure 6: Contractions for the non-associative Lambek calculus

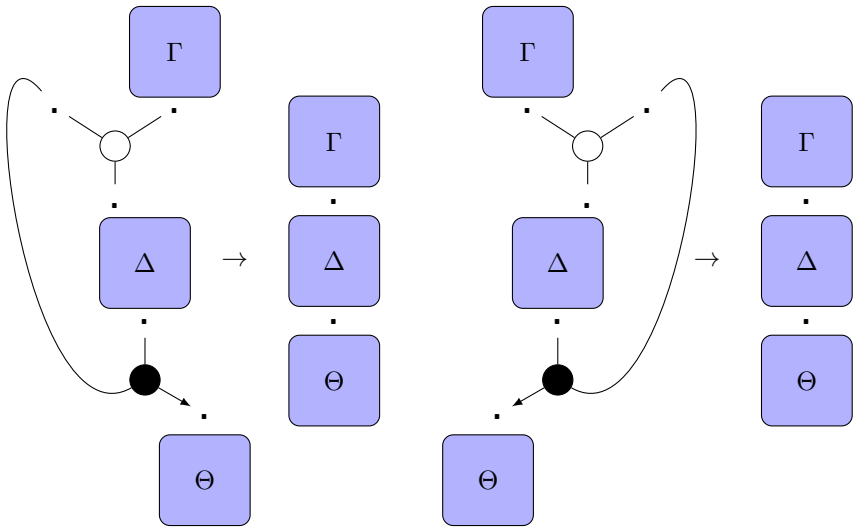


Figure 7: Contractions for the associative Lambek calculus.

of the par link with the arrow is not attached to the tensor link, but the other two vertices are. Moreover, the connections respect up/down and left/right.

The contraction for $\backslash I$, shown on the left of Figure 6, amounts to withdrawing the hypothesis which is the immediate left daughter of the root node, whereas the contraction for $/I$, shown on the right, withdraws the immediate right daughter.

Looking back at the abstract proof structure for our example in Figure 5, we see that it is not of the appropriate form to apply the $/I$ contraction for NL. When we look at the par link, it is connected top-bottom to a tensor link, but the curved line is connected the right branch of a different tensor link. We can therefore not apply the $/I$ contraction. In a non-associative context, this is the correct behaviour.

Moving to the associative Lambek calculus, we change the contractions to the ones shown in Figure 7. Where for the NL contractions, the top node of the par link was connected to the bottom node of the tensor link, for the L contractions a structure Δ has been inserted between the two. The contractions now have the conditions that for $\backslash I$, shown on the left of the figure, the path through Δ only passes through tensor links and moves left each time. For $/I$, shown on the right of the figure, the path through Δ only passes through tensor links and moves right each time.

This means that the $\backslash I$ contraction withdraws the leftmost hypothesis, whereas the $/I$ contraction withdraws the rightmost hypothesis, which is the desired behaviour for the Lambek calculus introduction rule.

We can relax the contractions even further, keeping only the constraint that

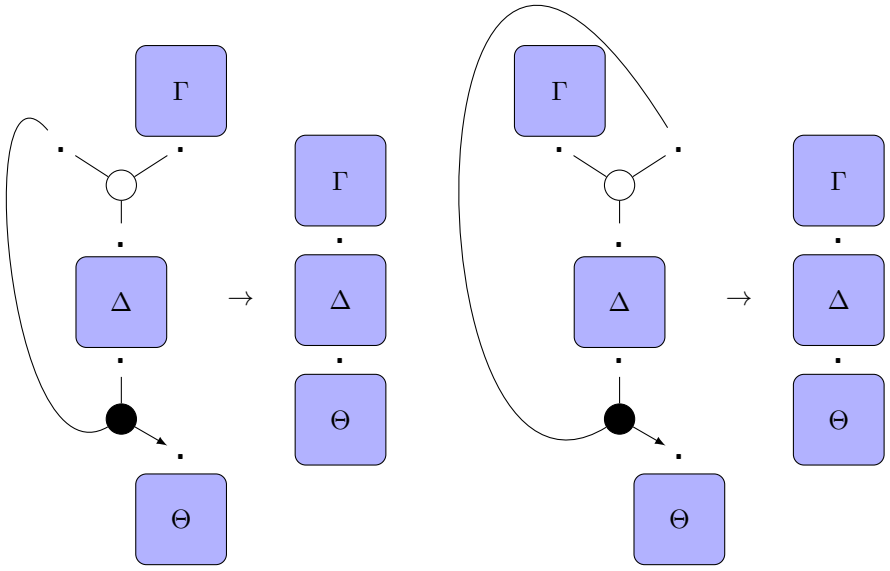


Figure 8: Contractions for the associative, commutative Lambek calculus LP

the path through Δ doesn't pass any par links. This produces the left branch and right branch extraction package of Moortgat (1999) and Oehrle (2011).

Finally, moving to a fully commutative system makes to the implications interderivable. In the commutative context, we keep only the tensor link for / and write it $\multimap$ as its linear logic counterpart. For the par link, we now treat the conclusions as unordered, which turns the two par links into different ways of writing the same link. Figure 8 shows the contractions for $\multimap$. The difference with Figure 7 is that we now have two contractions for a single connective, depending on whether the hypothesis is on a left or right branch.

If we denote a move left from the current node (and through a tensor link) by l and right by r, then starting from the root node, the path constraints for the contractions look as follows for the logics discussed.

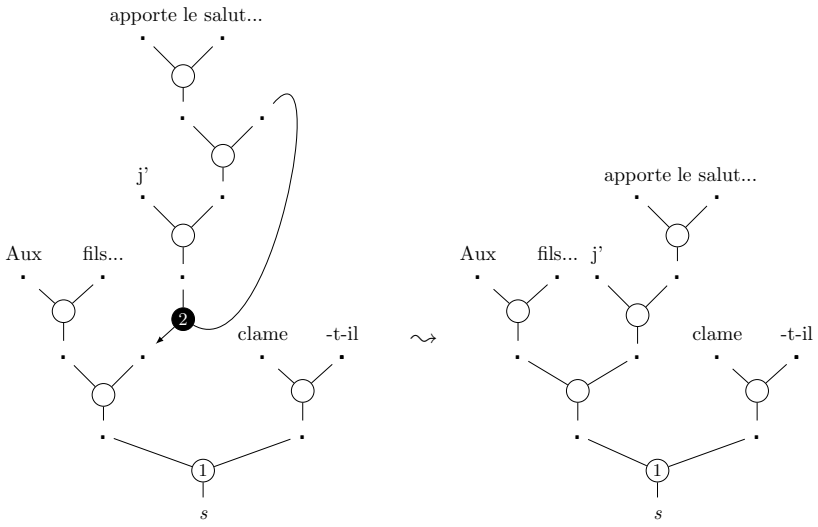


Figure 9: Right branch contraction for the abstract proof structure of Figure 5

	Left branch \		Right branch /
path	logic	path	logic
l	NL	r	NL
l^+	L	r^+	L
$(lr)^*l$	left branch extraction	$(lr)^*r$	right branch extraction
$(lr)^+$	LP	$(lr)^+$	LP

For NL, we must arrive at the node corresponding to the withdrawn hypothesis after one step (left for \, right for /). For L, we take at least one step, but all steps must be in the same direction. For the left/right branch extraction package, we only verify the last step is left or right. For LP, we take at least one step, but there are no constraints on the direction.

Going back to our previous example, repeated on the right of Figure 9, we can now state more clearly why we have marked the par link with '2'. The marking indicates that a right branch contraction is allowed here[1].

Verifying that the path from the root to the withdraw hypotheses doesn't pass any other par links (trivially true, since there are no others) and that the withdrawn hypothesis is on a right branch, we apply the contraction to obtain the structure shown on the right of Figure 9.

While we have contracted the initial proof structure to a tree, the yield of the tree is different from the input sentence. An additional tree rewrite, shown in Figure 10, allows us to move structures on the right of branches labeled '1' inside. This produces a tree with the correct yield, as shown on the right of

[1]For this example, a Lambek calculus $/I$ contraction would also work, but for more complicated cases we need the right branch extraction rule.

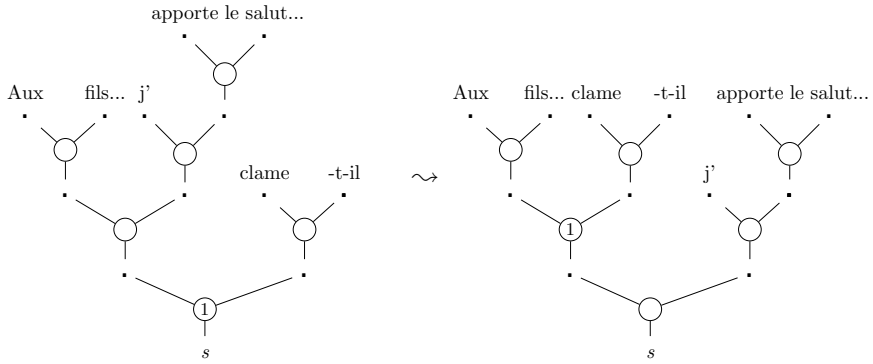


Figure 10: Tree rewrite to insert "clame-t-il" at its correct place

Figure 10. This is a fairly standard type of rule in multimodal type-logical grammars (Moortgat 1997).

2.2 Proof nets and lambda terms

Proof nets, like natural deduction proofs, correspond to lambda terms. These are linear lambda terms, because we operate in a fragment of multiplicative linear logic. Since multiplicative linear logic with just the '$\multimap$' connective is our semantic language, we can see the derivational meaning as a homomorphism from Lambek calculus proofs (or proofs in any of the variants of the Lambek calculus) into proofs of multiplicative linear logic.

Concretely, this means we replace the Lambek calculus implications '/' and '\' (and their variants, like '$\backslash_1$' and '$/_2$') with linear implication '$\multimap$'. Meanings are computed at the level of proof nets, that is, proof structures whose abstract proof structures can be contracted using the LP contractions or any restriction of these contractions. However, the actual contractions do not influence the meaning, only the initial proof structure does.

Going back to the abstract proof structure of Figure 5, before any contractions, we take this previous figure but annotate each link with the direction of the implication. This gives the figure shown on the left of Figure 11. Even though the directional information is superfluous, it makes the transformation to a linear logic proof more easy to explain. To transform this abstract proof structure into a structure representing the meaning, we simply swap the order of the premisses of the tensor links marked \. Since, by convention, we treat the conclusions of the LP par link as unordered, nothing has to be done for the par link. This produces the structure shown on the right of Figure 11.

We have changed the labels on the tensor links to @ for application ($\multimap E$), and the par links to λ for abstraction ($\multimap I$), indicating the interpretation in terms of meaning composition.

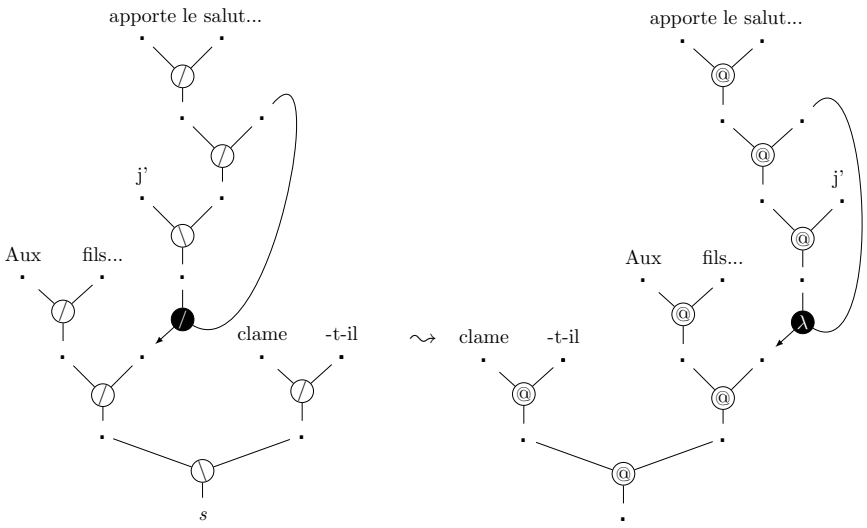


Figure 11: Semantic version of the proof net of Figure 5

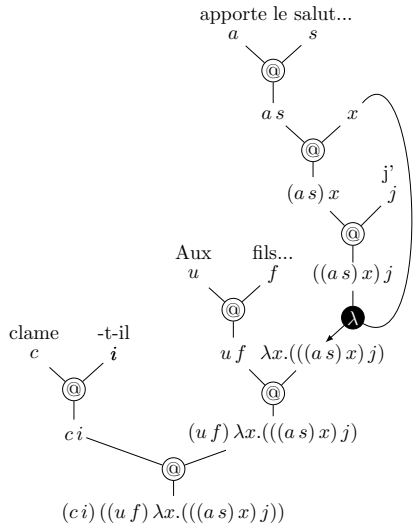


Figure 12: Figure 11 with terms at the vertices

Using variables u for "Aux", f for "fils", c, for "clame", etc., we can then compute the meaning corresponding to this proof as shown in Figure 12. The computed term

$$(c\, i)\, ((u\, f)\, \lambda x.(((a\, s)\, x)\, j))$$

is just a flat representation of the information in the LP proof net. Given a sequence of variables u, f, c, i, j, a, s (we will generally just use $x_1, \ldots, x_n$ for an n-word sentence) we can — by the Curry-Howard isomorphism between LP proofs and linear lambda terms — uniquely reconstruct the proof net from the lambda term and vice versa[2].

3 Neural proof nets

Proof enumeration for type-logical grammars works well in a research context, with relatively small lexicons and sentences. But how can we 'scale' type-logical grammar parsing to use these grammars for natural language understanding tasks? The standard approach has two main bottlenecks. First, lexical ambiguity (that is, words can be assigned many different formulas) becomes a major problem. Second, the matching step is formally equivalent to finding a correct permutation (under some constraints). For both of these bottlenecks, we are no longer interesting in enumerating the different possibilities, but in finding the correct solution, where we mean "correct" in the sense that it corresponds to the intended interpretation of the sentence (more prosaically, this means it should be the same as the proof of the sentence we find in a given corpus).

Different machine learning approaches have been successfully applied to the first task under the label 'supertagging' (Bangalore & Joshi 2011). As in many other research areas, deep learning has greatly improved the accuracy on this task. For example, for the French TLGbank (Moot 2015), a maximum-entropy supertagger obtains 88.8% of the assigned formulas correct whereas the current state-of-the-art using neural networks is at 95.9% (Kogkalidis & Moortgat 2022).

The second task, linking the atomic formulas, has only recently been implemented using a neural network (Kogkalidis et al. 2020*b*). This provides a complete neural network solution for parsing arbitrary text using type-logical grammars: neural proof nets. Neural proof nets have been successfully applied to the Dutch Æthel dataset (Kogkalidis, Moortgat & Moot 2020*a*), and the French TLGbank (De Pourtales et al. 2023).

4 Different perspectives

While we have seen important improvements in supertagging, it remains an important bottleneck for neural proof nets: a single error in the formula assignment can cause a proof to fail and not produce a lambda term at all. In addition, compared to standard parser evaluation metrics, it is hard to measure partial successes in neural proof nets: we can calculate the percentage of

[2] On the condition that the term has no closed subterms.

words assigned the correct formula, and, given the correct formulas, we can calculate the percentage of correct links, but it is not obvious how to calculate a score combining these two percentages into a single useful metric other than the extremely severe 'percentage of sentences without any errors'.

I therefore propose a different perspective on neural proof nets: instead of dividing the task into a supertagging and a linking task, we divide the task in a graph generation and graph labelling task. The graph generation task is set up in such a way it generates a proof net, step by step, ensuring at each step that the structure is valid. The graph labelling task then assigns labels to the graph vertices, where the labels are the logical connectives and atomic formulas.

The advantage of this setup is that it ensures by construction we obtain a proof net (and therefore a lambda term) for our input sentence, and that errors in the labelling phrase will have a relatively minor impact on downstream tasks. We can also see the inductive proof net construction steps as parser actions and explore multiple solutions (for example, using beam search). This makes it easy for a model to predict the k-best proof nets for a sentence.

We can set up this alternative vision of neural proof nets in such a way that each proof net is generated by a unique tree of parser actions. This also provides a way to solve the proof comparison problem for neural proof nets. If each proof net can be uniquely described by the actions used to create it, then we can compare two sets of actions and apply standard statistical measures (such as f-scores) for comparing these two sets.

4.1 Graph generation

The graph generation task is the most important one, given that it constructs the linear logic proof representing the meaning of the phrase, which is generally what interests us for applications in natural language processing. Graph generation is also the more complicated task given that it doesn't fit as neatly into a well-understood graph neural network architecture.

While we can set up the graph generation component as a form of backward chaining proof search (as we will see in Appendix A), it is most convenient to set it up as a form of forward chaining proof search. For forward chaining, the starting point is a set of isolated vertices, one for each word in the sentence. There is, of course, an implicit linear order on these vertices, which corresponds to the linear order of the corresponding words in the sentence.

The invariant we want to maintain is that each component of the graph corresponds to a proof and a lambda term. Among other things, this means that each component has a unique root node. In our initial structure, we have an axiom $A_i \vdash A_i$ and a free variable x_i for each of the initial vertices.

The basic operation is graph composition, shown in Figure 13. It takes the root node of two disjoint structures, and combines them into a single structure. Logically, this operation corresponds to the $\multimap E$ rule without explicit formula labels, and to application on the level of terms. Since Γ and Δ are disjoint, this is a valid rule application and one producing a linear lambda term.

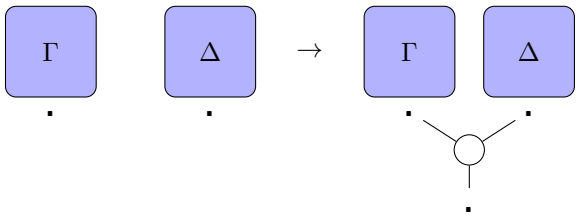


Figure 13: Graph composition

The introduction rule and abstraction operation are a bit trickier: we abstract over a variable, but this cannot be one of the variables corresponding to one of the words in the sentence, since these need to be free in the final term we compute. The solution is therefore to introduce a new variable as the sister of a node in the given structure, then introduce an abstraction at an ancestor node of this new variable. There are then two cases, depending on whether the newly introduced (and abstracted) variable is a functor or an argument[3].

Figure 14 shows the two graph rewrite operations corresponding the introduction rule, with abstraction over a functor shown on the left of the figure and abstraction over an argument shown on the right of the figure.

Given that the graph expansions could apply indefinitely, there is a final graph operation 'stop', which can apply only when the graph is connected, and which ends the graph generation process.

The reader will surely have remarked that the graph rewrite operations of Figure 14 are simply the graph contractions of the Lambek-van Benthem calculus LP of Figure 8 but applied as graph expansions. This makes it very easy to see the rewrite operations are correct. To show that a graph is a correct LP proof, we need to show it contracts to a tree using the contractions of Figure 8. Take a graph G constructed by application of the graph rewrites of Figures 13 and 14. We can order the rewrites such that all composition operations precede all expansion operations. The composition operations produce a tree, and the expansions produce a structure which can be contracted to a tree, because the expansions are simply the inverses of the contractions[4].

We can also look at the graph rewrite operations from the perspective of the lambda calculus, where they correspond to the following (slightly odd) term

[3]The attentive reader will note that this excludes terms of the form $\lambda x.x$. If we need these terms, we can add a specific graph rewrite for this case. However, type-logical grammar proofs are generally restrict to proofs without empty antecedents, and for proofs without empty antecedents the two given cases suffice.

[4]The condition that the path through Δ does not pass through another par link is superfluous in LP without the tensor product '$\otimes$', since we can always reorder the operations in such a way that it is respected by starting at the outermost abstraction, then moving inside. However, this no longer holds when we add the product or move to the Lambek calculus.

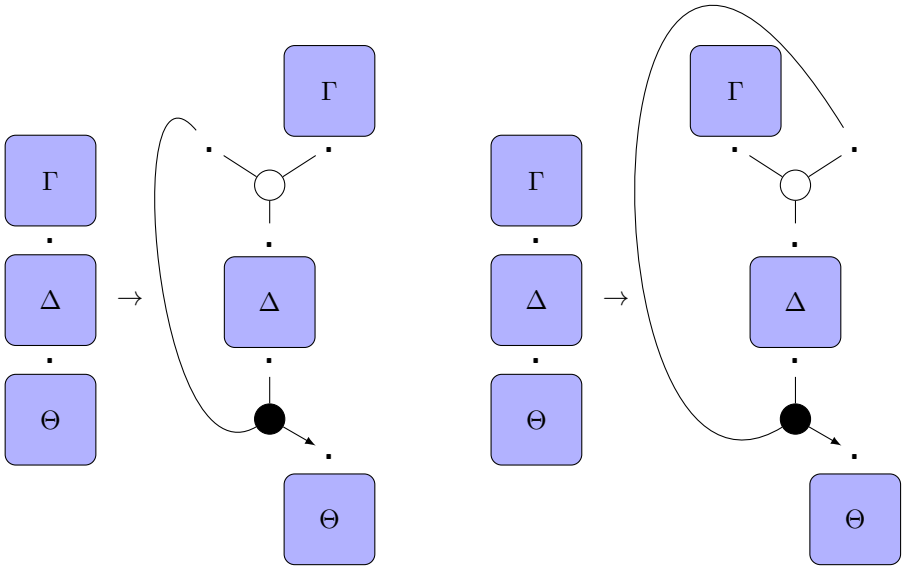


Figure 14: Graph expansions

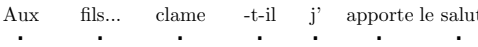


Figure 15: Initial configuration for the proof net generation

construction rules.

$$M, N \rightsquigarrow (M\,N) \qquad M,\ N \text{ do not share variables} \tag{1}$$
$$P[M[N]] \rightsquigarrow P[\lambda x.M[(x\,N)]] \tag{2}$$
$$P[M[N]] \rightsquigarrow P[\lambda x.M[(N\,x)]] \tag{3}$$

The term construction rules which are missing from a combinatorial perspective are the following.

$$M[N]] \rightsquigarrow M[(\lambda x.x)\,N)]] \tag{4}$$
$$M[N]] \rightsquigarrow M[(N\,(\lambda x.x))]] \tag{5}$$

We can exclude 4 because it only produces terms which are not beta normal, but the more important reason to exclude these two construction rules is that they are only needed to produce closed subterms, which correspond to empty antecedent derivations, and these are generally excluded in type-logical grammars. If want empty subterms, we can simply add 5 and the corresponding expansion[5].

4.2 Example

Returning to our previous example, we will now show how to generate the semantic proof net of Figure 11 using the graph rewrite operations. Our initial configuration is shown in Figure 15. In this configuration there are 42 possible applications of the composition/application rule and 14 possibilities for the expansion/abstraction rules. The expansion rules can apply to single vertices, where they correspond to replacing a variable y by either $\lambda x.(x\,y)$ (this operation is generally called lifting) or $\lambda x.(y\,x)$ (eta expansion).

In this configuration, the correct applications are shown at the top row of Figure 16. The bottom row shows how we can then combine the tree for "apporte le salut..." with the tree for "j". The three applications on the top row are unordered. We can apply them one at a time, or all at once. The application on the bottom row can only be correctly performed after "apporte le salut..." has been created.

However, there is nothing which forbids any of the other connections. It will be up to the neural model to learn that combining "-t-il" as functor with "Aux" is not a very promising start for the graph generation process.

[5]The attentive reader will remark that this still excludes the case where, rather than just some of its subterms, the complete term assigned to a sentence is closed. However, we start from the assumption that a sentence contains at least one word, and therefore the computed term must have at least one variable.

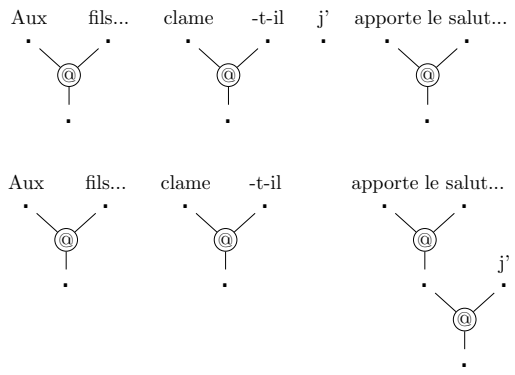


Figure 16: The top row shows first series of applications starting from Figure 15. On the bottom row, an additional application has been performed.

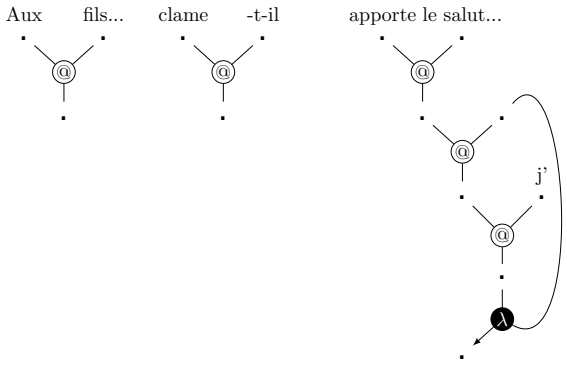


Figure 17: The structure of Figure 16 after an argument expansion

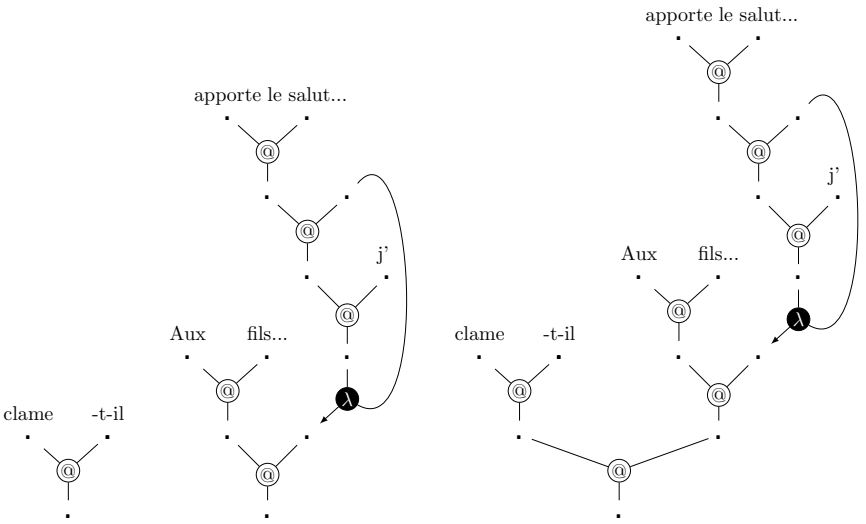


Figure 18: The final two applications

Given the graph on the bottom of Figure 16, we can apply one of the expansion rules, choosing the sister node of "j" to expand, creating a two new links, a tensor link with its root as the sister of "j" and a par link at the root of the complete structure. This par link connects to the right daughter of the newly introduced link, and is therefore an instance of an abstraction over an argument (rather than a function).

We can them combine the three components with two more graph composition/application rules to obtain the graph shown on the right of Figure 18. This is the same graph as the semantic graph of Figure 11 and it therefore encodes the correct semantics.

Before continuing, let's give some basic analysis of the combinatorics for the different steps. Given c components, there are exactly $c(c-1)$ ways to combine them using the application rule. For the expansion rules, given a component with n nodes, there are $\leq n^2+n-1$ ways to select a node and one of its descendants[6] and double that number once we distinguish between abstraction over a functor and an argument. For the graph shown on the right of Figure 18, there are 16 vertices and a total of 60 possibilities of node-descendant combination, which is considerably less than the maximum of 3840 for the number of vertices.

Given the combinatorics of the expansion step, it makes sense to restrict its application as much as possible. One simple but useful restriction is that we

[6]The worst case occurs when the tree is maximally unbalanced, such as the subtree "j'apporte le salut..." of Figure 18, since it maximises the number of nodes related by the descendant relation. The best case occurs when all leaves of the tree are at the same depth, which minimises the nodes related by the descendant relation and results in less than $\lceil n \log_2 n \rceil$ possibilities for selection of a vertex and one of its descendants.

never do expansions when the root of the expansion is on the left branch on an application link. These expansions produce a beta redex, and when we perform expansions from the root node upwards, the beta redex will remain in the final term[7].

We could also be tempted to think we can restrict expansions to the root node of components. However, although this is true for a system with expansions 1-3 and 5, such a restriction would need more careful analysis in a system with only expansions 1-3. To illustrate the potential pitfall, consider the term $\lambda x.f(\lambda y.(y\,x))$. Terms of this form are sometimes called 'argument lowering', corresponding at the formula level to $((A \multimap B) \multimap B) \multimap C \vdash A \multimap C$. When we want to produce this term starting from f, the only valid step we can take is argument expansion to produce $\lambda x.(f\,x)$. To produce the final term, we need to perform function expansion at x, but this produces an abstraction which is not at the root node. There appear to be ways around this, for example by considering newly introduced variables such as x in our example terms as a type of root node as well, allowing expansions to apply there.

4.3 Graph neural networks

We have seen how graph generation can be done in a way which generates only (linear logic) proof nets, and that these graphs/proofs correspond to the derivational meaning of the sentence. What remains to be shown is how we can integrate graph generation with neural networks.

Graph neural networks are an active research area, and would seem a priori well-suited for generating graph-based representations of proofs. However, our graphs have a number of particularities which restrict the architectures we can use. First of all, there is a linear order on the initial sequence of vertices. We will assume in what follows that the positional information that is part of the BERT encodings for each word (Devlin, Chang, Lee & Toutanova 2018) suffices for taking word order into account, but we could envisage architectures encoding word order differently. We also need some way to distinguish the three vertices adjacent to a hyperlink in the graph, which seems to require some form of edge labelling, for example representing the link shown below on the left as shown on the right.

This means we are using heterogeneous graphs, with (at least) two types of vertices, the first type corresponding to the vertices of the original abstract proof

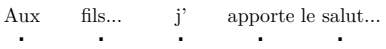
[7]For example, we can remove a beta redex $(\lambda x.M)\,N$ by a functor expansion as follows $\lambda y.((y\,\lambda x.M)\,N)$, but an outermost expansion would have produced $\lambda y.((yM')\,N)$ first (where M' is $\lambda x.M$ before the expansion introducing x), followed by the expansion of M' to $\lambda x.M$.

structure, and the second to the hyperedges of the abstract proof structure. This is a fairly common way of interpreting a hypergraph as a standard graph.

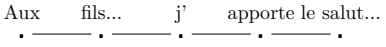
Transforming the graph generation mechanism into a neural network requires us to make choices for how to represent the initial state, how to represent the possible parse actions, implement a way to select an action among the alternatives, and finally a way to label the vertices in order to obtain the detailed formula information. We will discuss each of these in turn.

Initial states

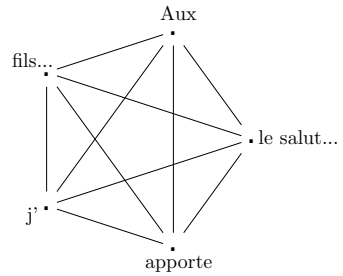
The simplest way to do represent the initial state is to represent each word by an isolated vertex, with BERT embedding of the word — a vector of floating point values encoding information about the context and relative word positions — at the corresponding vertex.



We can connect each word to its neighbours, producing the following initial configuration.



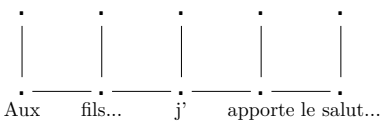
We can also use a fully connected graph.



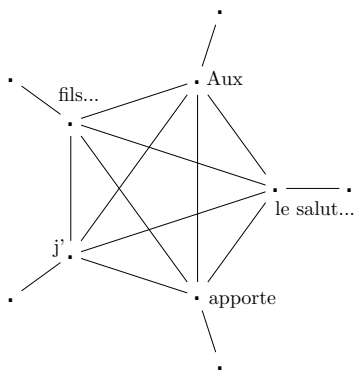
The advantage of the initial graph being more connected rather than a set of isolated vertices is that we can use a graph convolution or message passing step to aggregate the combined information of the BERT vectors of our words.

The drawback of having connections in the initial graph is that when we want to cast the problem as an edge prediction task, we must use multigraphs (i.e. graphs allowing multiple connections between the same vertices), otherwise we cannot predict connections at all for the case above (because all connections are already there) or exclude connections between neighbouring vertices for the line graph, which would exclude many correct linkings.

Given that few graph neural network architectures support multigraphs, we can therefore choose to use the following structure for the line graph instead.



And the structure below for the fully connected graph.



In these structures, we keep the advantages of allowing message passing between the different word representations, but can still cast the problem as an edge prediction task without having to use multigraphs.

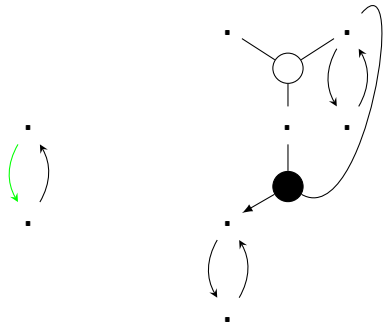
Parser actions as edge prediction

The easiest way to implement the graph generation parser actions as graph neural networks is to implement them as directed edge predictions. In this type of graph neural network, we represent graph composition as a directed edge from functor to argument. The expansion/abstraction rule is slightly more complicated. As we have seen in Section 4.1, we need to select two vertices, one the ancestor of the other. There are two complications: the first is that we need to distinguish between the two different expansion rules, and the second is that we can select the same vertex twice. We can distinguish between two edges between the same pair of nodes using labelled edges in multigraphs, and we can represent the root and leaf node being identical by a self-loop (again labeled) in the graph, but there are relatively few graph neural networks permitting both self-loops and multi-graphs.

We can avoid these complications by adding new vertices representing possible abstractions to the graph, one for each root of a component and one for each introduced variable at a previous expansion step.

The figure below sketches what this would look like. The simplest possible situation is shown on the left. Here we have a single vertex (at the top of the figure) representing variable f and the abstraction root below it. The two possibilities for abstraction are then represented by the directed arrows. The arrow in green transforms f into $\lambda x.(fx)$ (f is a functor in this situation, and

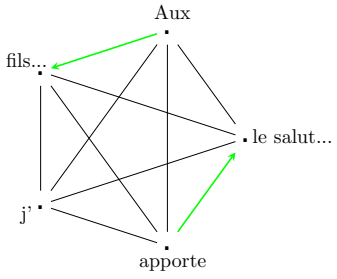
the arrow goes from functor to argument), whereas the other arrow transforms f to $\lambda x.(x\, f)$ (where f is an argument and the arrow goes from functor to argument).



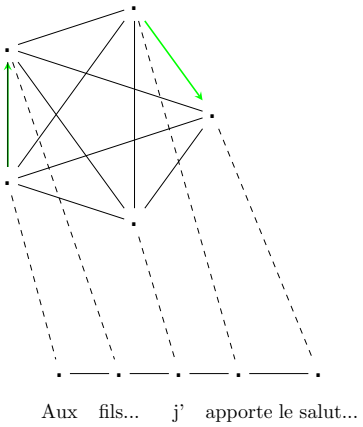
On the right of the figure, we see the result after f has been transformed into $\lambda x.(f\, x)$. We can now perform abstraction at the root vertex and at the newly created x node. From the root node, all its descendants (the four vertices corresponding to f, x, $f\, x$ and $\lambda x.(f\, x)$) are potential sisters to a newly introduced variable, but only the root node has been shown to avoid cluttering the figure. From x, given that it's a leaf node, there are only two possibilities, corresponding respectively to $\lambda x.(f\, \lambda y.(y\, x))$ and $\lambda x.(f\, \lambda y.(x\, y))$.

Using this trick, all possible parsing actions are directed edges. The task for the neural network is then to assign weights to all edges, maximising the weights assigned to edges representing correct actions, and minimising the weights assigned to incorrect ones.

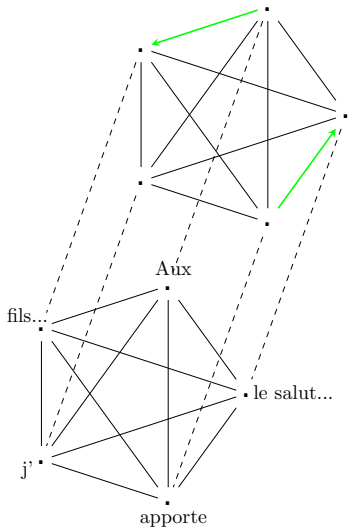
From the starting situation in the 'isolated vertices' scenario, all connections are possible an we should maximise those shown in green below, which indicate that 'Aux' is functor of 'fils...' and 'apporte' of 'le salut...'. The connections for the abstractions are not shown in the figure, but should be represented as possiblities.



In the line scenario, it would look as follows. The abstraction vertices and connections are again not shown.

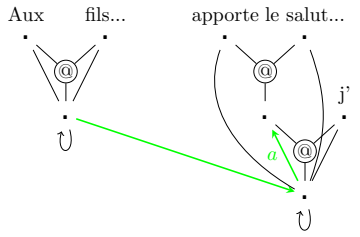


And finally in the fully connected scenario (again without the abstraction vertices/connections), we would have two isomorphic cliques, connected two each other at their matching points, the first clique (shown at the bottom) representing the fully connected input graph, and the second one the possible parser actions.



After we have performed some composition operations, we obtain the configuration shown below. Given that there are only two disconnected substructures, we can still perform a last composition operation with either structure as the functor (in this case, the structure "Aux fils..." should be the functor). Alternatively, we can perform a number of abstractions. We have only shown the possibilities from the root node, using a self-loop to help distinguish the computed structure from the parser actions.

There are two possible compose actions, and 16 possible expansion actions: two for each node at both substructures. To keep the figure simple, only a few edges have been shown. The edges which move towards the correct structure are shown in green.



This gives the following global strategy for neural proof net generation.

1. start with an initial configuration,
2. propagate information using message passing,
3. predict directed edges,
4. update the graph with the graph operation corresponding to the best edge (or all edges over a certain weight),
5. update the action edges: delete action edges which are no longer possible in the updated graph, and add actions which have become possible,
6. continue to 2 with the updated graph, or stop if no edges are assigned weights above a threshold.

We can also add 'stop' as a global graph property to predict separately, but only when the graph is already connected (and therefore sure to correspond to an appropriate lambda term). This has the advantage that the neural network can learn to calibrate the stop condition against the edge prediction weights.

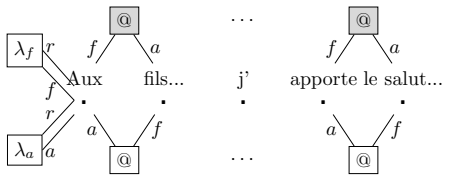
During training, we take different stages of the graph generation, and in each case, we maximise the weights of the correct edges and minimise the incorrect ones (using values between 0 and 1).

This global strategy still has quite a few blanks to fill in, but we can try to adapt some of the known edge prediction architectures to the current context (Wu, Cui, Pei, Zhao & Song 2022, Chapters 10, 19, 24). However, we should be aware of a potential problem, and that is that for some of the main use cases of edge prediction in neural networks, we have a single, large graph and want to predict missing edges and their weights/labels. This is the case for recommender systems and knowledge graphs. By comparison, our graphs will be quite small, but we will have many of them.

Parser actions as vertex labelling

As an alternative to generating parser actions as edge predictions, we can represent parser actions by vertices. There are application vertices labeled '@' and connected by an f and a edge to two different vertices in the term representation. Similarly there are vertices labeled 'λ_f' and 'λ_a' for abstraction over functor and arguments respectively, connected to two (not necessarily different) vertices.

This would represent the initial graph — for the case with isolated vertices — and possible parser actions as shown below. Only a few of the possible actions have been shown. The correct actions have been shaded grey.



Like before, we want the graph neural network to take an input graph and assign weights between 0 and 1 to all possible actions, maximising the weights assigned to the correct actions and minimising the weights assigned to the incorrect ones. The actions include the 'stop' action when the graph is connected.

The structure of the neural proof net generation algorithm does not change much with respect to the edge prediction strategy.

1. start with an initial configuration,
2. propagate information using message passing,
3. predict weights for all action vertices,
4. update the graph with the graph operation corresponding to the best action vertex,
5. continue to 2 with the updated graph, until the 'stop' action is selected.

There are graph neural network architectures available in standard libraries which seem well-suited for our node-labeled, vertex-labeled graphs (Xie & Grossman 2018, Maguire, Grattarola, Mulligan, Klyshko & Melo 2021).

Comments, alternatives, evaluation

Both of the previous options to generate the graph structure iterate between an action prediction stage and a graph update stage. The graph update stage has quite a bit of overhead: actions which are no longer possible need to be removed (generally, this is because a vertex is no longer a root node), new actions need to be added: new applications for new root nodes, new abstractions for all new vertices.

To transform a correct sequence of parser action into the input for our training stage, we need to cut the sequences into their individual steps, indicating *all* correct actions in each case. Each step has only the BERT embeddings of the leaves, and the graph labels as its input. However, it would seem advantageous cast the task as a sequence prediction task, where all weights of the previous step are available to be updated. Treating proof net generation as a type of neural network for graph transformation would be a promising possibility (Wu et al. 2022, Chapter 12). However, there doesn't appear to be an easy fit between our problem and existing architectures for this case. For example, Wang, Che, Guo & Liu (2018) use a sequence-to-graph neural network for parsing semantic dependencies, but this approach is not easily adapted to our task since it is essentially an edge prediction task between vertices corresponding to words. This means it operates modulo associativity (not distinguishing between $(np\backslash s)/np$ and $np\backslash(s/np)$ for example) and that there is no easy way to incorporate the expansion rules. The parsing model which corresponds best to our needs is the sequence-to-graph model of Chen, Sun & Han (2018), who use a graph generation network to translate sequences of text to parser actions generating a semantic graph. However, even this model would need to be changed to ensure only proof *nets* are generated, otherwise we would lose one of the major attractive points of our current approach to neural proof nets.

Whatever the strategy chosen for graph generation, we want to compare the results with the current state-of-the-art for the 'standard' strategy of neural proof nets (De Pourtales et al. 2023). This means evaluating it on the no-errors in the linear logic proof criterion, or, equivalently the correct (untyped) lambda term criterion.

4.4 Vertex labelling to obtain the surface structure graph

For the second task, we have already produced the graph representing the meaning, and we want to obtain the more detailed graph representing the syntactic structure.

Figure 19 repeats the structure of Figure 11, but with additional arrows pointing to the functor of the application nodes. These arrows are not formally a part of the structure, but they make it easier to see some of its properties. The arrows always point to the main formula of their link, meaning that the arrow is always a formula $A \multimap B$, with A its sister and B its mother node (for tensor links the mother is displayed below the central node, for par links above it).

With this in mind, we can simply compute the generic structure of the formulas which ensures it respects the restrictions on the formulas imposed by the links. This is morally equivalent to computing a principal type in type theory: we make the minimal assumptions about the formulas which guarantees

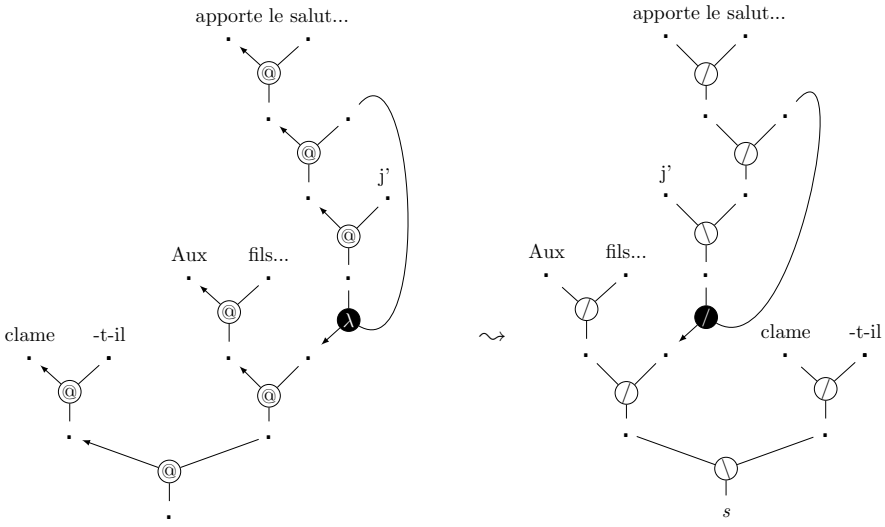


Figure 19: Figure 11 used for vertex labeling

the proof is correct.

$$\begin{aligned} \mathit{Aux} &= A \multimap (E \multimap F) \multimap G \\ \mathit{fils...} &= A \\ \mathit{clame} &= B \multimap G \multimap H \\ \text{-}t\text{-}\mathit{il} &= B \\ j' &= C \\ \mathit{apporte} &= D \multimap E \multimap C \multimap F \\ \mathit{le\ salut...} &= D \end{aligned}$$

The conclusion is assigned the formula H. Since each formula occurs exactly twice, we can uniquely recover the proof net from these formulas, as there is only one way to connect the positive and negative atoms.

To transform these minimal lexical assignments to the fully detailed ones we seek, we need to do two things:

1. replace the type variables by atomic formulas[8],
2. replace the linear logic connectives by their directional variants.

[8]We assume all proofs are in long normal form. Without this restriction, it would be possible to instantiate a variable by a complex formula. For long normal form proofs, we can restrict ourselves to replacing variables by atomic formulas.

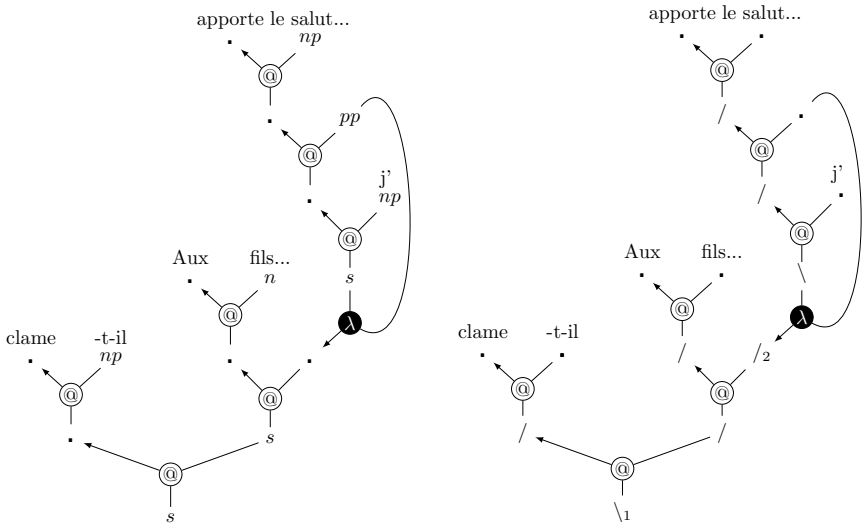


Figure 20: The two vertex labelling tasks: atomic formulas (left) and connectives (right)

We can see task 2 as an *edge* labelling problem, as shown in Figure 19 on the right. Each central node now uses the labels '/' and '\' instead of '@' and 'λ'.

However, it is easier to treat both tasks as vertex labelling problems, allowing us to train them in parallel. For task 1, we want to label all vertices where no arrow arrives with one of the fixed set of atomic formulas in our grammar. For task 2, we want to label all vertices which are not leaves of the structure with one of the fixed set of binary connectives.

From the structure of the input graph, we can determine exactly which vertices need an atom label and which vertices need a connective label. This means that the graph convolutions can propagate the information through vertices which will not be labelled in any way which optimises the information available at the vertices which *will* be labelled.

Figure 20 shows the correct result for the first task on the left of the figure. It amounts to choosing $A = n$, $B = C = D = np$, $E = pp$ and $F = G = H = s$,

which produces the following lexical assignments.

$$\begin{aligned} \mathit{Aux} &= n \multimap (\mathit{pp} \multimap s) \multimap s \\ \mathit{fils...} &= n \\ \mathit{clame} &= \mathit{np} \multimap s \multimap s \\ \text{-}\mathit{t}\text{-}\mathit{il} &= \mathit{np} \\ \mathit{j'} &= \mathit{np} \\ \mathit{apporte} &= \mathit{np} \multimap \mathit{pp} \multimap \mathit{np} \multimap F \\ \mathit{le\ salut...} &= \mathit{np} \end{aligned}$$

Figure 20 shows the correct results for the second task on the right of the figure. It corresponds to replacing linear implication with the directional implications of the Lambek calculus. This task also recovers the labels for '$\backslash_1$' and '$/_2$'.

$$\begin{aligned} \mathit{Aux} &= (G/(F/_2E))/A \\ \mathit{fils...} &= A \\ \mathit{clame} &= (G\backslash_1 H)/B \\ \text{-}\mathit{t}\text{-}\mathit{il} &= B \\ \mathit{j'} &= C \\ \mathit{apporte} &= ((C\backslash F)/E)/D \\ \mathit{le\ salut...} &= D \end{aligned}$$

Combining these two results together then gives the full formula information.

$$\begin{aligned} \mathit{Aux} &= (s/(s/_2\mathit{pp}))/n \\ \mathit{fils...} &= n \\ \mathit{clame} &= (s\backslash_1 s)/\mathit{np} \\ \text{-}\mathit{t}\text{-}\mathit{il} &= \mathit{np} \\ \mathit{j'} &= \mathit{np} \\ \mathit{apporte} &= ((\mathit{np}\backslash\mathit{np})/\mathit{pp})/\mathit{np} \\ \mathit{le\ salut...} &= \mathit{np} \end{aligned}$$

Compared to the graph generation task, the labelling tasks in this section are relatively simple: our vertex labelling needs to decide for the first task that "fils..." is a noun n and "le salut...." is a noun phrase etc., but also, for the second task, that "fils..." is to the immediate right of "Aux", and "le salut" to the immediate right of "apporte".

These seem to be tasks where standard graph architectures for vertex labelling would work well (Wu et al. 2022, Chapter 4).

Generating the labels allows for a direct comparison with the state-of-the-art in supertagging for the same dataset (Kogkalidis & Moortgat 2022, De Pourtales et al. 2023).

5 Conclusion

I have sketched a novel way of generating proof nets using graph neural networks. Even though there are a number of implementation details to work out and experiment with, I believe this approach has a number of important advantages over the 'standard' way of Kogkalidis et al. (2020*b*).

By changing the split of the task from a formula prediction plus link predication task to a graph generation plus vertex labelling task, we can generate our proof nets in a way which guarantees we always obtain a proof (that is, a proof *net*, not a proof structure) and therefore a lambda term for downstream natural language processing tasks.

The graph generation task is also easily adapted to a k-best or beam search setting: at each prediction step, we can take the k most promising actions (those with highest weight/probability), batch generate the possible next actions for all of them, then prune until we are again left with k-best different graphs. This requires some way to ensure different ways of generating the same graph are identified, with only the one with the highest weight kept[9].

The parser actions perspective also makes it easier to evaluate partial successes. For example, we can look at two different parse trees for a sentence an compare the two by evaluating how many of the applications and abstractions are correct.

A weakness of the current proposal is that adding product formulas makes the task quite a bit harder. First of, we need to distinguish between two cases for each tensor link, application and pairing (corresponding to the product introduction rule). Second, the product elimination rule is complicated: we need to expand two vertices (by expanding either one or two vertices) and attach the par link for the product to these two vertices. On the other hand, the product is extremely rare in our dataset (Moot 2015) and it appears all occurrences $A \otimes B$ can be replaced by $(A \multimap (B \multimap s)) \multimap s$, which is both a standard way to implement products as implications in the lambda calculus, and the standard treatment of argument cluster coordination in type-logical grammars.

Given the weakness at treating product types, the skeptic would be right to ask whether we're not simply doing natural deduction proof search, thereby losing some of the most important benefits of using proof nets. I think that in a sense, proof nets — especially in the version presented here — *are* natural deduction proofs, but the product elimination rule is tricky in combination with the graph expansion approach advocated here.

[9] Using some form of canonical vertex numbering (e.g. left-to-right, bottom-to-top), we can avoid having to check for graph isomorphism and simply check for identity.

Taken together, I think the benefits largely outweigh the drawbacks. The main problems with standard neural proof nets are that they can fail to produce any meaning at all because of a single supertag error, and that they have problems in the k-best scenario because of the task split: it is not hard to obtain the k-best supertags but it is hard to obtain the k-best proofs.

This new method will of course have to be evaluated against the current state-of-the-art for neural proof nets, in terms of supertagger scores, but, most importantly, in terms of the percentage of sentences receiving the correct meaning.

References

Bangalore, S. & Joshi, A. (2011), *Supertagging: Using Complex Lexical Descriptions in Natural Language Processing*, MIT Press.

Chen, B., Sun, L. & Han, X. (2018), Sequence-to-action: End-to-end semantic graph generation for semantic parsing, *in* 'Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)', Association for Computational Linguistics, Melbourne, Australia, pp. 766–777.
URL: *https://aclanthology.org/P18-1071*

De Pourtales, C., Rabault, J., Kogkalidis, K. & Moot, R. (2023), Deepgrail: Neural proof nets for French, Technical report, LIRMM. forthcoming.

Devlin, J., Chang, M.-W., Lee, K. & Toutanova, K. (2018), 'BERT: Pre-training of deep bidirectional transformers for language understanding', *arXiv preprint arXiv:1810.04805* .

Girard, J.-Y. (1987), 'Linear logic', *Theoretical Computer Science* **50**, 1–102.

Kogkalidis, K. & Moortgat, M. (2022), 'Geometry-aware supertagging with heterogeneous dynamic convolutions', *arXiv preprint arXiv:2203.12235* .

Kogkalidis, K., Moortgat, M. & Moot, R. (2020*a*), Æthel: Automatically extracted typelogical derivations for dutch, *in* 'Proceedings of The 12th Language Resources and Evaluation Conference', pp. 5257–5266.

Kogkalidis, K., Moortgat, M. & Moot, R. (2020*b*), Neural proof nets, *in* 'Proceedings of the 24th Conference on Computational Natural Language Learning', pp. 26–40.

Kogkalidis, K., Moortgat, M., Moot, R. & Tziafas, G. (2019), Deductive parsing with an unbounded type lexicon, *in* 'Proceedings of SEMSPACE 2019'.

Maguire, J. B., Grattarola, D., Mulligan, V. K., Klyshko, E. & Melo, H. (2021), 'XENet: Using a new graph convolution to accelerate the timeline for protein design on quantum computers', *PLoS computational biology* **17**(9).

Montague, R. (1974), The proper treatment of quantification in ordinary English, *in* R. Thomason, ed., 'Formal Philosophy. Selected Papers of Richard Montague', Yale University Press, New Haven.

Moortgat, M. (1996), 'Multimodal linguistic inference', *Journal of Logic, Language and Information* **5**(3,4), 349–385.

Moortgat, M. (1997), Categorial type logics, *in* J. van Benthem & A. ter Meulen, eds, 'Handbook of Logic and Language', Elsevier/MIT Press, chapter 2, pp. 93–177.

Moortgat, M. (1999), Constants of grammatical reasoning, *in* G. Bouma, E. Hinrichs, G.-J. Kruijff & R. T. Oehrle, eds, 'Constraints and Resources in Natural Language Syntax and Semantics', CSLI, Stanford, pp. 195–219.

Moot, R. (2015), 'A type-logical treebank for French', *Journal of Language Modelling* **3**(1), 229–264.

Moot, R. (2021), 'Type-logical investigations: proof-theoretic, computational and linguistic aspects of modern type-logical grammars', Habilitation à Diriger des Recherches, Université de Montpellier.

Moot, R. & Puite, Q. (2002), 'Proof nets for the multimodal Lambek calculus', *Studia Logica* **71**(3), 415–442.

Oehrle, R. T. (2011), Multi-modal type-logical grammar, *in* R. Borsley & K. Börjars, eds, 'Non-transformational Syntax: Formal and Explicit Models of Grammar', Wiley-Blackwell, chapter 6, pp. 225–267.

Wang, Y., Che, W., Guo, J. & Liu, T. (2018), A neural transition-based approach for semantic dependency graph parsing, *in* 'Proceedings of the AAAI conference on artificial intelligence', Vol. 32.

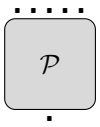
Wu, L., Cui, P., Pei, J., Zhao, L. & Song, L. (2022), *Graph Neural Networks: Foundations, Frontiers, and Applications*, Springer.

Xie, T. & Grossman, J. C. (2018), 'Crystal graph convolutional neural networks for an accurate and interpretable prediction of material properties', *Physical review letters* **120**(14), 145301.

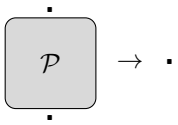
A Backward chaining proof net generation

We can do the graph generation procedure as backward chaining proof search as well. While the forward chaining proof search of Section 4.1 is essentially a type of natural deduction proof search, the backward chaining method is more like sequent calculus proof search. At each stage, we have a structure we are constructing about which we know only the hypotheses and the conclusion vertices. This is also essentially the same strategy as the one used by Kogkalidis, Moortgat, Moot & Tziafas (2019), with the difference that we do not have the formula information in the current case.

The starting position therefore has exactly one hypothesis for each word in the sentence, and a single conclusion.



We can stop the proof any time we have a single hypothesis, in which case we turn the proof box with a single hypotheses and a single conclusion into a single vertex.



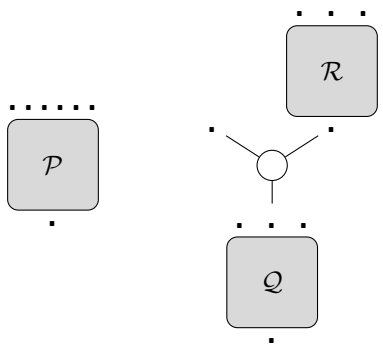
The complicated operation for backward chaining proof search is the elimination rule for linear implication. We are computing a structure $\mathcal{P}$ which hypotheses $x_1, \ldots, x_k$ and conclusion z. We select a unique hypothesis y as the main formula, then split the remaining hypotheses into two groups $v_1, \ldots, v_n$ and $w_1, \ldots w_m$. We have $n + m + 1 = k$, since each x_i is in exactly one of the three groups. The v_i, together with the conclusion of the new link, will be the premisses of new structure $\mathcal{Q}$ whereas the w_j will be the premisses of $\mathcal{R}$.

If this sounds complicated, it is essentially the $\multimap L$ rule from sequent calculus, where we similarly divide the formulas into Γ (our v formulas), Δ (our w formulas) and $A \multimap B$ (y).

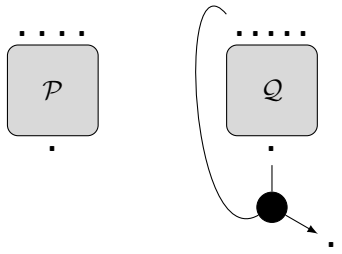
$$\frac{\Delta \vdash A \quad \Gamma, B \vdash C}{\Gamma, \Delta, A \multimap B \vdash C} \multimap L$$

There are many possibilities here, since for k formulas, there are k choices for the main formula, then for the remaining $k-1$ formulas we assign them to one of two groups. This gives a total of 2^k different ways to apply this rule. However, this might still be an effective strategy for neural parsing. As indicated before, the backward chaining proof search procedure is essentially the same as the one

of Kogkalidis et al. (2019) without explicit formula information at the vertices during the graph construction stage.



The final operation is the introduction rule. We start with a box $\mathcal{P}$ with premisses $x_1, \ldots, x_n$ for $n > 0$, add a new hypothesis x_0 and attach a new link withdrawing this hypothesis. Like the graph expansion rules, this rule increases the number of hypotheses, but unlike the expansion rule it competes with the 'stop' rule only when $n = 1$.



The backward chaining graph generation strategy therefore has two major drawbacks compared to the forward chaining strategy.

1. it front-loads the combinatorics for the $\multimap E$ rule; where the forward chaining strategy needs to consider n^2 combinations for this rule (it selects a functor from n roots and an argument from the $n-1$ remaining ones), the backward chaining strategy needs to consider 2^n. We are also handicapped because we cannot use any formula-based ways to restrict our partitions, such as the 'count check'.

2. it can more easily get stuck in a loop; in the forward chaining strategy, the $\multimap I$ rule always has the 'stop' rule as an alternative, whereas this is not the case for backward chaining.

For these reasons, the forward chaining strategy appears to be preferred, but ultimately we would need to compare the performance of both approaches,

and study other graph construction methods which mix ideas from forward and backward chaining.