Neural Lemma Suggestion in Coq

Axel Marmet

January 2022

1 Introduction

The goal of this project is to provide lemma suggestion in the interactive theorem prover Coq [1] using the Transformer architecture [2].

1.1 Coq

Coq offers a formal language, called Gallina, to write mathematical definitions, executable algorithms and theorems. It uses the Curry-Howard isomorphism to mechanically verify the validity of theorems. The isomorphism generally states that "a proof is a program, and the formula it proves is the type for the program". Thus to create a proof in Coq the user must create a term that has the type corresponding to the theorem she wants to prove. To aid in this term creation, Coq provides a second language $\mathrm{L}_{\mathrm{tac}}$ that allows the use of tactic to incrementally create a term without having to write Gallina code. A quick use case of Coq is shown and explained in Figure 1.

1.2 Lemma suggestion

When proving theorems, it is natural to reuse previously proven theorems instead of reproving everything in the local context of the proof. However seeing what theorems may be useful to prove the current goal is not always clear to a human expert, even less so to a machine. This is the problem of premise selection. The usual formulation of this problem is to find a function

$$f : \mathit{goal} \times \mathit{context} \times \mathit{premise} \to [0, 1]$$

where the codomain is the probability that the premise will be useful. We propose to go further and try to solve the problem of premise selection which we now state:

$$g : \mathit{goal} \times \mathit{context} \to \mathit{premise}$$

where the codomain is a premise, created and considered useful by the network

There are multiple reasons to try to solve this. Firstly given a similarity function

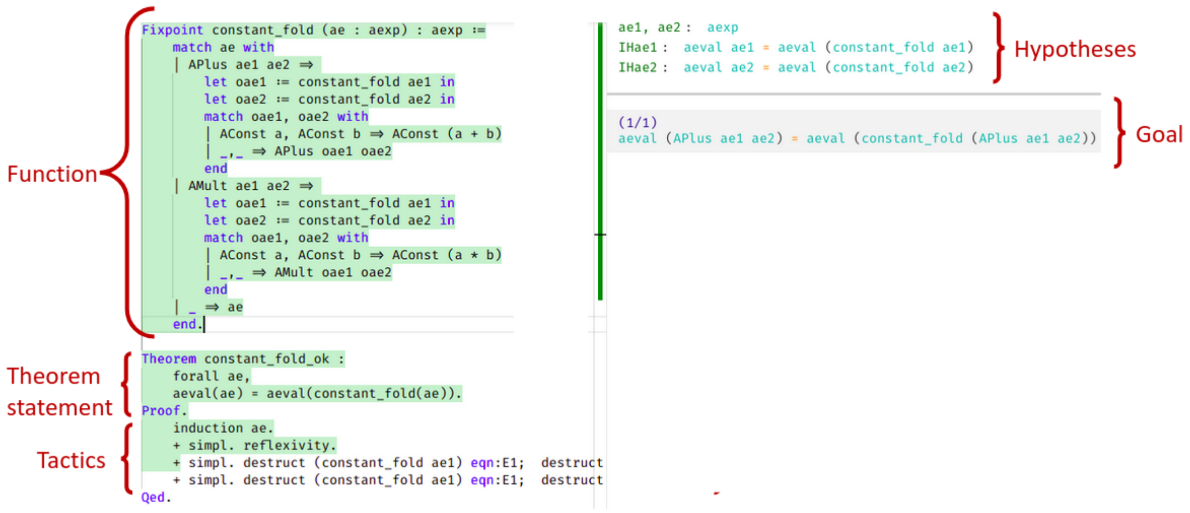
$$s : \mathit{premise} \times \mathit{premise} \to [0, 1]$$

we can combine g and s to obtain

$$\begin{aligned} &f' : \mathit{goal} \times \mathit{context} \times \mathit{premise} \to [0, 1] \\ &f'(\mathit{goal}, \mathit{context}, \mathit{premise}) = s(g(\mathit{goal}, \mathit{context}), \mathit{premise}) \end{aligned}$$

which fulfils premise selection.

Also the number of premises can sometimes be in the order of multiple thousandths (for example the *CompCert* project [3] proofs in our dataset have, on average, 13'340 premises in their context [4]) which limits the realistic computational power the network can possess. In the domain of Natural Language Processing (NLP), most of the recent successes have been achieved by some variants of the Transformer architecture [2] and those models often benefit from immense amount of parameters [5, 6]. Confirming the intuition that transformer can be applied to code are, for example, the Code Transformer [7] that achieves state-of-the-art results on code



```
Fixpoint constant_fold (ae : aexp) : aexp :=
    match ae with
    | APlus ae1 ae2 ⇒
        let oae1 := constant_fold ae1 in
        let oae2 := constant_fold ae2 in
        match oae1, oae2 with
        | AConst a, AConst b ⇒ AConst (a + b)
        | _,_ ⇒ APlus oae1 oae2
        end
    | AMult ae1 ae2 ⇒
        let oae1 := constant_fold ae1 in
        let oae2 := constant_fold ae2 in
        match oae1, oae2 with
        | AConst a, AConst b ⇒ AConst (a * b)
        | _,_ ⇒ AMult oae1 oae2
        end
    | _ ⇒ ae
    end.

Theorem constant_fold_ok :
    forall ae,
    aeval(ae) = aeval(constant_fold(ae)).
Proof.
    induction ae.
    + simpl. reflexivity.
    + simpl. destruct (constant_fold ae1) eqn:E1;  destruct
    + simpl. destruct (constant_fold ae1) eqn:E1;  destruct
Qed.
```

```
ae1, ae2 :  aexp
IHae1 :  aeval ae1 = aeval (constant_fold ae1)
IHae2 :  aeval ae2 = aeval (constant_fold ae2)

(1/1)
aeval (APlus ae1 ae2) = aeval (constant_fold (APlus ae1 ae2))
```

Figure 1: A small example of using Coq. On the top left one can see a simple definition of some constant folding function. Below it is a theorem stating that evaluating a term that has been constant folded is equivalent to evaluating the initial term. On the right we can see the current state of the proof with on top the hypotheses and the current goal under the horizontal line.

summarization or the Skip-tree training task [8] that shows the ability to do mathematical reasoning.

Finally, contrary to premise selection, premise suggestion is not limited to existing premises. This means that a user could write the main proof she wants, and then be suggested intermediary lemmas to prove, helping structure the overarching proof architecture.

1.3 Architecture overview

We now quickly present the general architecture of the project, also shown in Figure 2. All the datasets are obtained by processing the CoqGym dataset [4]. There are three different training tasks, each resulting in parameters that are used in the next architecture. This allows for an easier optimisation. The first task uses Word2Vec [9] to create subtoken embeddings. The second task, *CounT*, uses the resulting embeddings and focuses on understanding Gallina terms one at a time, it uses a specific transformer architecture called the *Code Transformer* [7] that is suited to understand AST structured data. Finally the last task *Griffon* uses the pretrained parameters of *CounT* to take in the entire context as input and output useful lemmas.

2 Data Preprocessing

The data used in this project can be split in a pipeline as shown in Figure 3.

2.1 Initial data

We use the Coq proof dataset CoqGym [4]. It includes 71K human-written proofs from 123 open-source software projects. Each project belongs to only one of the splits so as to ensure that no test or validation proof comes from a project that is used in training, which also allows the dataset to verify how the models generalise.
In the dataset the Gallina terms are stored in their "kernel form" which is completely stripped of syntactic sugar, which removes the burden of learning this syntactic sugar. A proof sample is composed of

- its environment : all the premises in the scope, either defined in the same file as the proof or imported from a library
- its proof tree : a tree representing the proof steps where each node is a goal and its corresponding hypotheses, and tactics are edges connecting a goal to further sub-goals

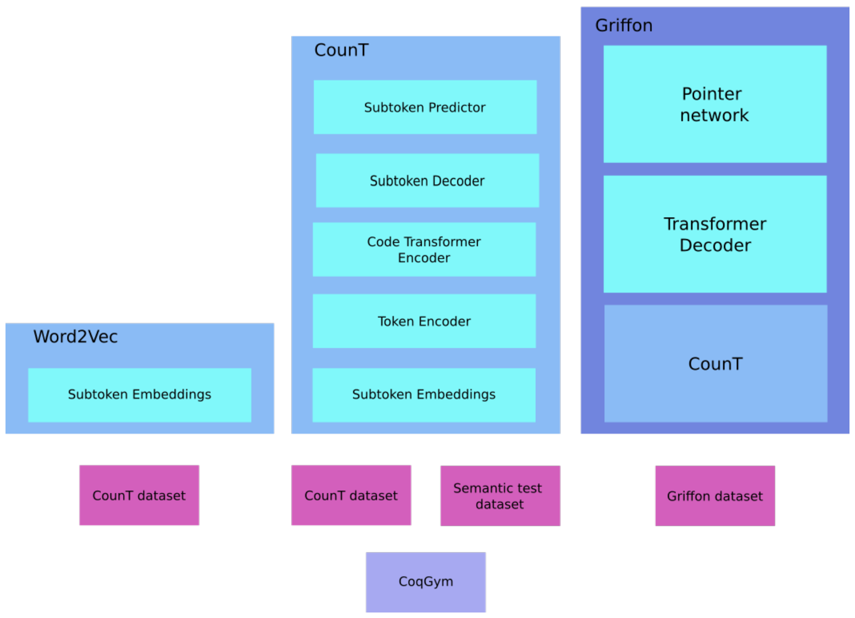


Figure 2: General organisation of the project, showing the various datasets created from the original dataset and the three distinct training tasks

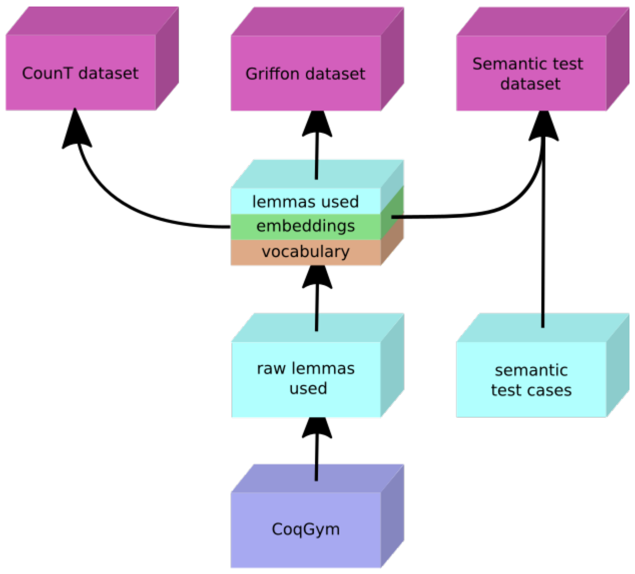


Figure 3: The data processing pipeline and its various dataset

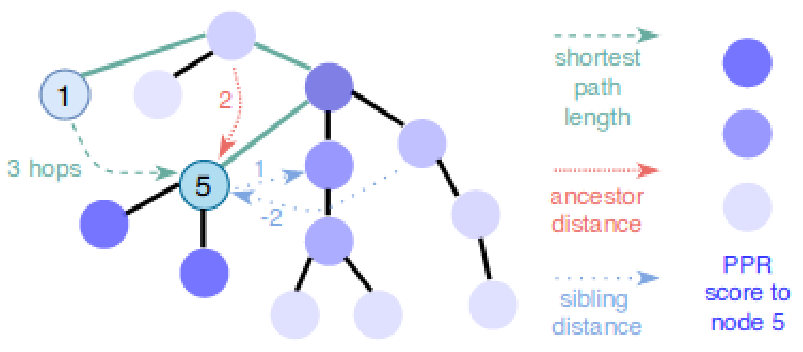


Figure 4: Illustration of the various distances used, taken from [7]

2.2 Base data

To acquire training samples we iterate over all the proofs in the CoqGym dataset: every time a premise is used, either by the tactic `apply` or `rewrite`, we store all the hypotheses, the current goal and the premise used itself. Note that if the premise used is one of the hypotheses, we remove it from the context given to the model. Otherwise the optimisation process could easily collapse to always copying the hypothesis that overlaps the most with the goal.

To make batching more regular, we discard all hypotheses or goals longer than 128 tokens, lemmas longer than 256 and we clip the total number of hypotheses at 64. This filtering retains 98% of the potential samples to amount to 75K suggestions. Because we keep the same splits as CoqGym to avoid test set pollution we have

- 37271 training samples
- 19647 validation samples
- 17935 test samples

At this point the data corresponds to the *raw lemmas used* box in Figure 3.

2.3 Preprocessing

The preprocessing follows the language agnostic preprocessing guidelines from the Code Transformer paper.

To tokenize our inputs we first create tokens by splitting on whitespace. Once we have those original tokens, we split them into at most 5 subtokens according to the snake_case naming convention of Coq. For example `reverse_list` (`my_list`) would be tokenized to [[`"reverse"`, `"list"`], [`"("`], [`"my"`, `"list"`], [`")"`]. We also recreate the inputs' ASTs by parsing the CoqGym s-expression. Then using the AST we compute multiple token-to-token distances:

- shortest path length : number of hops necessary for a node to reach another one, considers the AST as an undirected graph.
- ancestor distance : number of hops to reach an ancestor or child node. Distance is positive when accessing ancestors and negative when accessing children.
- sibling distance : maintain an order in the children of a node, not usual in graphs but ubiquitous in ASTs. Allows to differentiate between $f(x, y)$ and $f(y, x)$
- Personalized PageRank (PPR) [10] : PPR is a well-studied proximity measure which has been shown to be very effective in learning with graphs [11, 12], PPR captures the local graph structure around a pair of nodes (i, j)

The various distances are illustrated in the Figure 4. Feeding the distances to the *Code Transformer* allows it to be aware of the AST structure of the input.

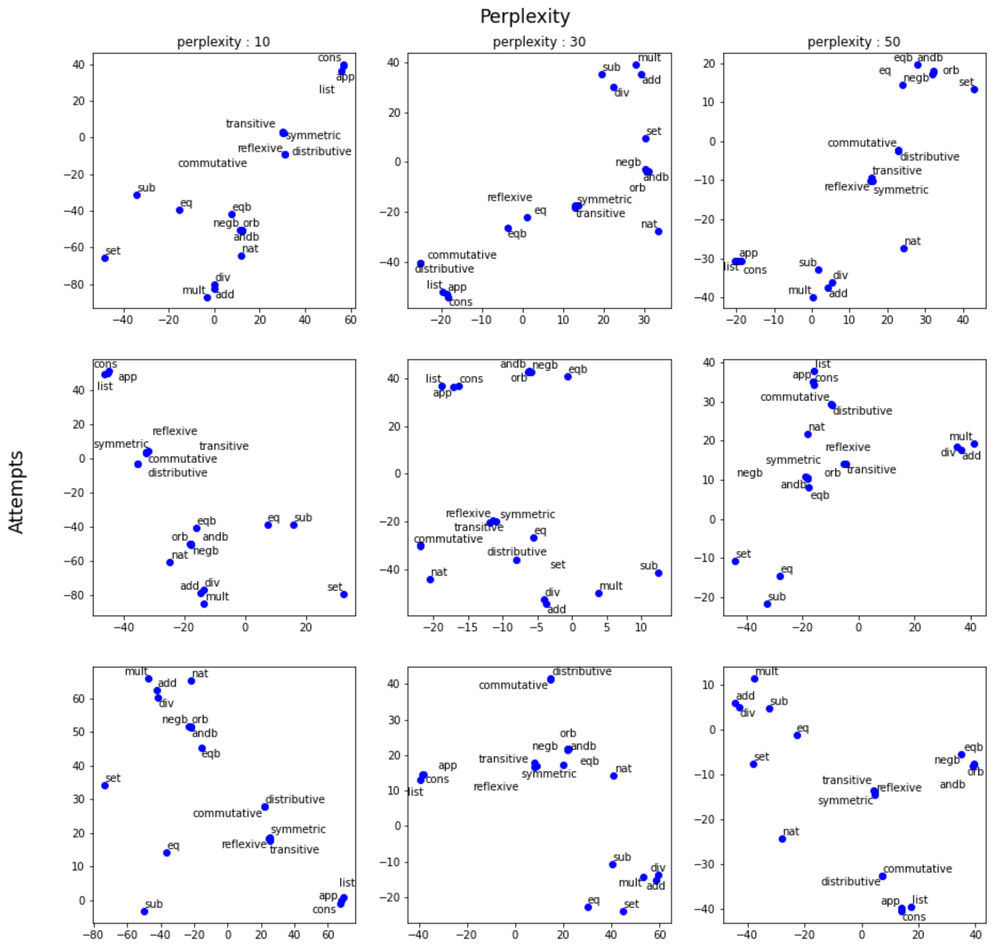


Figure 5: t-SNE projection of a few created embeddings. The horizontal axis represents various values of perplexity, a t-SNE hyperparameter, the higher it is the higher the dimension reduction will pay attention to non local information The vertical axis is simply multiple attempts as t-SNE is non-deterministic

2.4 Word2Vec

Once the subtokens are obtained we can create the vocabulary our models will use. However, such a vocabulary maps each token to a single integer. This low dimensionality is not appropriate for an input to a neural network. Thus, we use Word2Vec [9] to associate high dimension vectors to each subtoken. A property of the embeddings generated is that high cosine similarity correlates with high semantic similarity. To test this we use the dimensionality reduction technique t-SNE [13]. The results can be seen in Figure 5. The results are quite pleasing as we can see various cluster that make sense. Firstly operations operating on the same type tends to be clustered together, for example [`andb`, `orb`, `negb`, `eqb`]. More interestingly, because it's a clustering not directly related to types, the properties of operations [`distributive`, `commutative`] form a different cluster than the properties of relations [`transitive`, `reflexive`, `symmetric`].

When applied to large NLP corpora, the embeddings also display some form of "arithmetic", such as $Rome - Italy + France = Paris$. We weren't able to replicate such results with our embeddings. This is probably due to the small size of our dataset, compared to that of those used in the NLP world.

3 CounT

As explained earlier, our *Code Transformer* encoder will take in each of the hypotheses and the goal (referred to as "statements" when grouped together from now on). Its main goal is to encode useful information in the statements so that it is easily accessible by the decoder when outputting lemmas.

It is common in deep learning to use pretraining on a self-supervised task before finetuning on a downstream task. Perhaps the most famous example of a pretrained model is BERT [5], a standard transformer encoder architecture pretrained with the Masked Language Model (MLM) task.

3.1 Masked language model

The MLM task consists of applying some noise to 15% of the tokens and asking the network to predict the original tokens. This task has two main advantages.

Firstly, this does not require human labels, hence the *self-supervised* taxonomy. This generally allows to use orders of magnitude more data than a true supervised task.

Secondly, although seemingly simple, this task requires a lot of understanding of the language and the underlying domain. For example to correctly predict "The woman walked across the street, checking for traffic over [mask] shoulder." requires an understanding of grammar and "EPFL is located in [mask], Switzerland" requires factual knowledge.

We use this pretraining task to train as many parts of our final architecture as possible on a much bigger quantity of data than would otherwise be available. A visual description of the MLM task is shown in Figure 6b.

3.2 Semantic test cases

Our MLM task is simply a proxy to make our encoder understand the Gallina language and its world, using only validation on the same task does not offer much insight. Of course, if the MLM task is perfectly solved, then our encoder has acquired a good understanding of the language, but the converse does not hold. It is possible that the network has acquired a good understanding but still fails on the validation set due to project-specific information it has no access to.

We use *behavioral* testing as proposed by Ribeiro et al. [14]. We verify the capacity of the network to do type inference and its understanding of natural deduction. For example a unit test of natural deduction is the elimination of the disjunction.

```
forall (X Y Z : Prop), (X ∨ Y) → (X → [mask]) → (Y → [mask]) → Z.
```

Similarly, a test of type inference capabilities is

```
forall (A B : Type) (b : B) (x : A), exists (f : [mask] → B), f x = b.
```

As of writing this report there are only 15 test cases as they were written entirely by hand. Interesting future work would be to write a program that could automatically generate such test cases by composing the rules of natural deduction or type inference.

3.3 Architecture

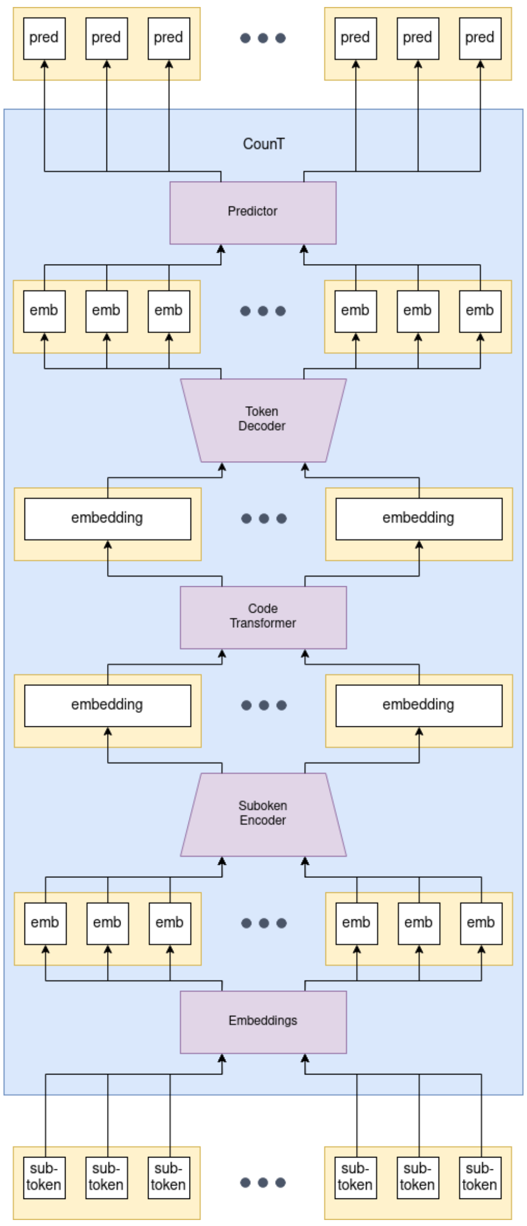
The *CounT* network is composed of multiple modules that will be reused at various places in the final network. It is a linear sequence of transformations. First, each subtoken gets replaced by its embedding. Then, all subtokens belonging to the same token are fed through a small Multi Layer Perceptron (MLP) to obtain a single embedding. Those token embeddings are then fed through the *Code Transformer*. The resulting embeddings are split into subtokens by another MLP. Then a prediction head assigns to each of those subtokens a probability distribution over the entire vocabulary. A visual representation of the architecture can be seen in Figure 6a.

3.4 Data

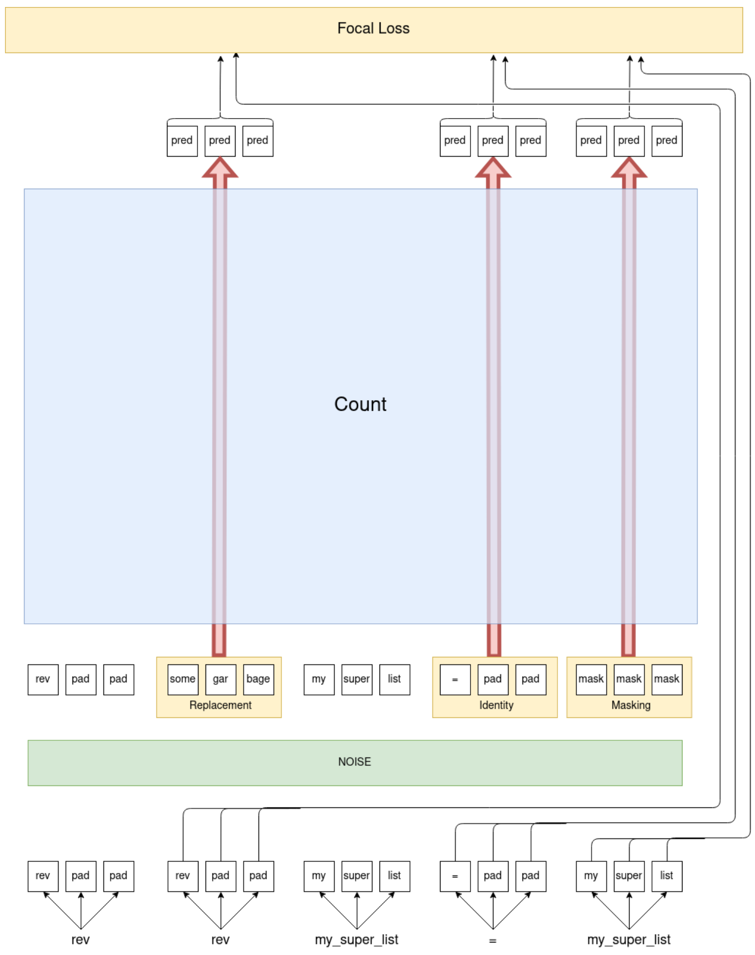
The data used for this part is the *CounT* dataset. It is easily obtained by iterating over all base samples and taking each statement as its own *CounT* sample.

3.5 Results

The network was trained for 50 epochs for which the validation metrics are shown in Figure 7. One can see that the training quickly plateaus. We will use the model with the highest semantic test case accuracy, at the end of epoch 17. The test set results are shown in Table 1.



(a) Architecture of the *CounT* network



(b) Illustration of the MLM taks when used on the *CounT* architecture

Figure 6: The *CounT* architecture and how it is trained

	accuracy [%]	top 3 accuracy [%]
checking padding	82.58	84.92
not checking padding	43.64	50.48
semantic tests	21.43	26.57

Table 1: *CounT* results on the test set

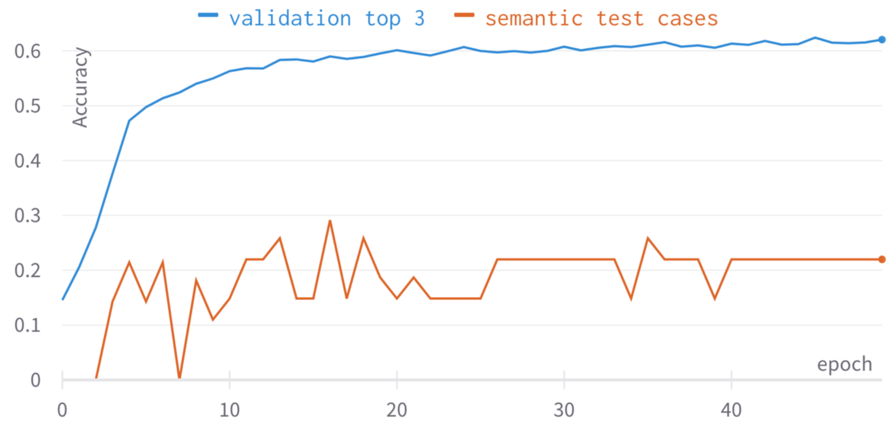


Figure 7: Validation metrics of the CounT training run

The difference in accuracy between the MLM task itself and the semantic test case is unexpected, one would expect the opposite ratio as explained earlier. It seems to indicate that the training task is not as well designed as it could be. The main culprit is probably that all tokens have the same exact 0.15 probability of being picked, which means that frequent tokens will be over-represented while rare tokens will be under-represented. Potential future work would be to investigate on the best way to mask tokens.

4 Griffon

The *Griffon* network is the final architecture of this project. It reuses all parts of the *CounT* network. The two additional modules are a modified transformer decoder and a pointer network [15]. An overview of the architecture can be seen on Figure 8.

4.1 Transformer Decoder

A standard decoder only reads from one encoded sequence at a time, which is not the case in our setup where the decoder has access to up to 65 statements (64 hypotheses and 1 goal). Thus, we add an attention layer that only looks at the name of the statements. The hypotheses always have a name and the goal term is simply assigned the name *Goal*. With this additional layer each token can decide how much it cares about a specific statement based on its name.

4.2 Pointer Network

A pointer network allows to copy an input subtoken instead of having to generate it from a fixed length vocabulary. This creates an extended vocabulary containing the original vocabulary and all unknown tokens in the input. This is vital for *Griffon* if it ever wants to be used on real projects. This is because Coq projects can have wildly different domains and the fixed length vocabulary learned from the training set cannot represent all potential domains. For each output subtoken t the pointer network outputs both a probability distribution a_i^t over the input subtokens and a copying probability p_{copy}^t. Then the probability of choosing the word w from the extended vocabulary is

$$p^t(w) = (1 - p_{\text{copy}}^t)p_{\text{vocab}}^t(w) + p_{\text{copy}}^t \sum_{i:w_i=w} a_i^t$$

4.3 Results

We train *Griffon* for 20 epochs, for which the validation accuracy is shown in Figure 9. The validation set indicates that the best model is the last. However, note that is seems that the network hasn't fully converged, and future work could look into training it for longer. The results are shown in Table 2.

An accuracy of 90% is quite satisfying, but as can be seen when checking only actual subtokens, the networks predicts the expected subtoken only 75% of the time. This is good but probably not yet good enough to be used

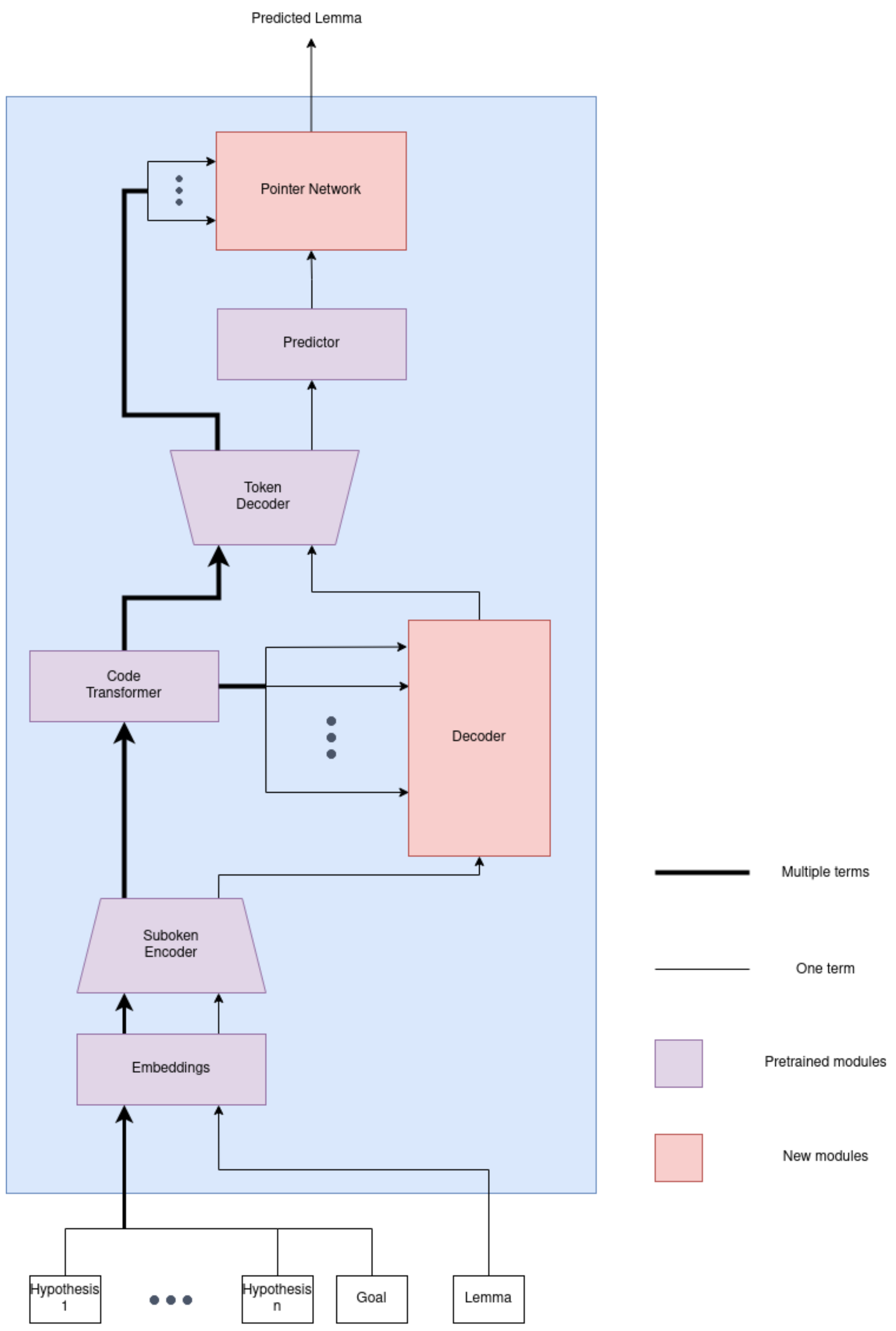


Figure 8: Architecture of the *Griffon* network

	accuracy [%]	top 3 accuracy [%]
checking padding	90.28	93.23
not checking padding	76.46	83.48

Table 2: Test set results for *Griffon*

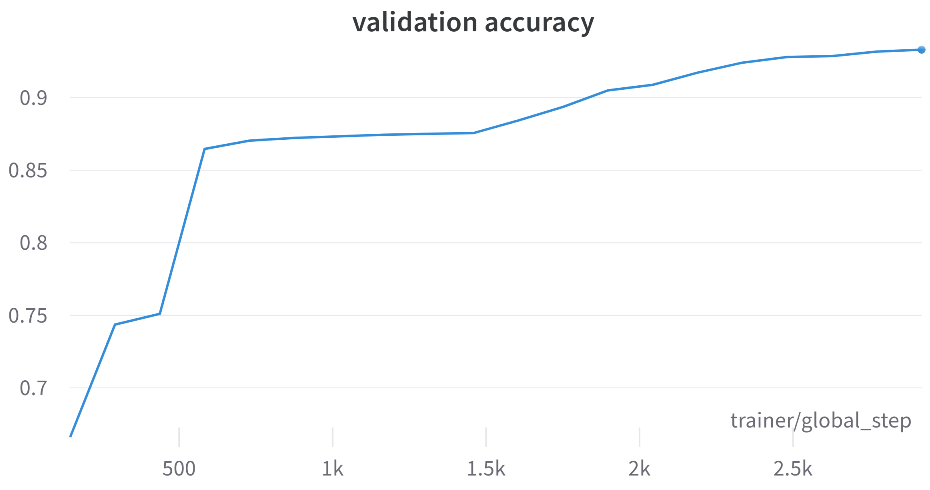


Figure 9: The validation metric of the training run

in real world cases. Unfortunately, inference and beam search are not yet implemented so we cannot inspect the predictions qualitatively.

5 Conclusion

We implemented multiple architectures that, when combined together, display good quantitative results. As the research on this intersection of Proof Assistant and Deep Learning is still in its infancy, we weren't able to compare our results to another implementation. The project was successful in investigating the feasibility of lemma suggestion, as framed in this report, with current deep learning techniques.

The next steps will be to integrate this into a real Coq plugin, then incrementally improve it in the areas highlighted along the report.

References

[1] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[2] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[3] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. Compcert - a formally verified optimizing compiler. 2016.

[4] Kaiyu Yang and Jia Deng. Learning to prove theorems via interacting with proof assistants. *CoRR*, abs/1905.09381, 2019.

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[6] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[7] Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. Language-agnostic representation learning of source code from structure and context. In *International Conference on Learning Representations (ICLR)*, 2021.

[8] Markus Norman Rabe, Dennis Lee, Kshitij Bansal, and Christian Szegedy. Mathematical reasoning via self-supervised skip-tree training. In *International Conference on Learning Representations*, 2021.

[9] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

[10] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[11] Johannes Klicpera, Aleksandar Bojchevski, and Stephan Günnemann. Personalized embedding propagation: Combining neural networks on graphs with personalized pagerank. *CoRR*, abs/1810.05997, 2018.

[12] Aleksandar Bojchevski, Johannes Klicpera, Bryan Perozzi, Amol Kapoor, Martin Blais, Benedek Rózemberczki, Michal Lukasik, and Stephan Günnemann. Scaling graph neural networks with approximate pagerank. *CoRR*, abs/2007.01570, 2020.

[13] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(86):2579–2605, 2008.

[14] Marco Tulio Ribeiro, Tongshuang Wu, Carlos Guestrin, and Sameer Singh. Beyond accuracy: Behavioral testing of NLP models with CheckList. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4902–4912, Online, July 2020. Association for Computational Linguistics.

[15] Abigail See, Peter J. Liu, and Christopher D. Manning. Get to the point: Summarization with pointer-generator networks. *CoRR*, abs/1704.04368, 2017.