Utrecht University

Department of Information and Computing Science

Master thesis: Artificial Intelligence

Breaking left-to-right generation in Transformer models:

arbitrary orderings on tagging tasks

Supervisor & first examiner:
Tejaswini Deoskar

Second examiner:
Lasha Abzianidze

Candidate:
Giacomo Bais
Student Number: 5355583

October 10, 2024

Abstract

Transformers (Vaswani et al., 2017) have achieved great success in various natural language processing tasks. Traditionally, generation happens in a left-to-right fashion in such models, mimicking how we process and produce text as humans. However, for tasks in which tokens do not exhibit strong left-to-right sequential dependencies such as Combinatory Categorical Grammar (CCG) tagging, alternative generation orders may be more effective. This thesis explores a novel approach to break away from the traditional left-to-right ordering in Transformer-based models. We introduce TagInsert, a sequence-to-sequence model designed to perform tagging tasks with an ordering that is learned by the model itself, potentially reducing error propagation typically associated with auto-regressive models.

We evaluate the performance of our architecture across three tasks: Part-of-Speech tagging, CCG non-constructive tagging, and CCG constructive supertagging. Our results show that arbitrary generation order improves performance in Part-Of-Speech and CCG non-constructive tagging when compared to a left-to-right model. Notably, for CCG non-constructive tagging, we observe a statistically significant advantage over the standard left-to-right approach, indicating that breaking the traditional ordering may yield better results for tasks with weak left-to-right sequential dependencies. In the case of constructive supertagging, another version of TagInsert adapted for the task is presented. While reaching comparable results with the existing literature, better solutions to introduce arbitrary ordering in the task are likely needed to more effectively exploit the benefits the architecture brings.

We also present extensive qualitative analyses to understand the behaviour of both the non-constructive and constructive model, in order to identify properties in which there is a benefit from breaking the left-to-right ordering.

Contents

1 Introduction **4**

2 Background and related work **8**
2.1 Auto-regressive models and left-to-right approach 8
2.2 The Insertion Transformer (Stern et al., 2019) 10
2.3 Combinatory categorical grammar 18
2.4 Constructive supertagging 20
2.5 Breaking the left-to-right approach 23

3 Models **26**
3.1 Models for Part-Of-Speech and CCG Tagging 26
3.2 Models for CCG constructive supertagging 36

4 Experiment 1: Part of Speech Tagging **48**
4.1 Data and Pre-processing 49
4.2 Results and discussion 50

5 Experiment 2: CCG Tagging **57**
5.1 Data and Pre-processing 58
5.2 Results and discussion 58

6 Experiment 3: CCG Constructive supertagging **63**
6.1 Data and Pre-processing 65
6.2 Results and discussion 67

7 Summary and Conclusion **83**
7.1 Part-Of-Speech Tagging 83
7.2 CCG Tagging 84
7.3 Constructive Supertagging 85
7.4 Future work 87
7.5 Infrastructure and computation 88
7.6 Ethics and Reproducibility 89

Appendix

A Appendix A **90**

B Appendix B **91**

1. Introduction

The field of Natural Language Processing has seen enormous advances in the last few years. With the introduction of the Transformer architecture (Vaswani et al., 2017), we have seen an exciting surge of ideas and applications that pushed the limits of what was thought possible only a decade ago. Models that make use of such ideas are now ubiquitous, and it is difficult to predict what new advances can be achieved even in the near future, as the core concepts of the architecture are applied in a wide range of applications and domains.

The original Transformer is an encoder-decoder model, developed for sequence-to-sequence tasks. In this approach, there is an encoder side tasked to representing an input sequence and a decoder side tasked to generate a target sequence using information from the encoder. Among many tasks, models adopting such an architecture can be trained to do machine translation, summarization and question-answering. Experimentation with the architecture has led to two other approaches on the architecture: encoder-only and decoder-only models. In encoder-only models, the input sequence is processed to produce a meaningful and compact representation of the data. The representation can then be used for many purposes, like classification tasks and fine-tuning. The most well-known example of such models is BERT (Devlin et al., 2019), which has found its way in the pipeline of many deep learning models. In strong contrast, decoder-only models are trained to generate tokens in an auto-regressive manner. Traditionally, at each time step one token is generated, and the sequence is expanded from left to right. These architectures can be used for any type of text generation task, with the most well-known example being the GPT family of Large Language Models (Radford and Narasimhan, 2018).

Indeed, most of the architectures adopting a decoder are auto-regressive, left-to-right generation models. In such models, tokens predicted at each time step are conditioned on what was previously generated, and the sequence grows from left to right. Auto-regressive models have shown many advantages. For example, because of the constant conditioning on previous generations at each time step, they can maintain good contextual consistency. On the other hand, this also creates a chain of sequential dependencies that can lead to an unpredictable propagation of

errors. Auto-regressive models are not forced to follow a left-to-right order of generation but, in the case of the Transformer, this is frequently the method of choice. Indeed, this order of generation had a prominent role in the success of the architecture. There is a great synergy between the pre-determined order of generation and the attention mechanism, which is at the core of the Transformer. By knowing what *future* tokens are at each time step, we can mask the communication mechanism during training and efficiently process all sub-sequences in parallel. Effectively, this means that all the processing that needs to be done by the decoder can happen in one fell swoop. While this is obviously very desirable, it also foregoes the possibility of letting the model learn a **dynamic order of generation**. While the left-to-right choice is reasonable, being akin to what we do as humans when processing and producing text, it is interesting to see what kind of ordering a model would choose if given the opportunity. Moreover, it may be possible that, depending on the task, using a left-to-right order of generation limits the model in the dependencies it can learn. For example, in text generation, left-to-right sequential dependencies are plenty and the assumption holds, but does the same apply in classification tasks? To correctly classify difficult categories, information present in classes later on in the sequence may be crucial, and it will never be available to the model when the order is pre-determined this way.

In this thesis we explore the idea of breaking the left-to-right ordering on a number of tagging tasks, in which the left-to-right sequential dependencies are weaker when compared to text generation. Specifically, the aim of this research is to present a novel Transformer-like encoder-decoder architecture that allows for tagging in arbitrary ordering: **TagInsert**. By letting the model decides on its own ordering, we aim to show that advantages can be gained, spanning from alleviating the effect of error propagation of auto-regressive models to more accurate generation in selected tasks. Moreover, as TagInsert is an Encoder-Decoder model, we also inspect whether a sequence-to-sequence model can perform as a tagger. To these ends, we designed three experiments on different tagging tasks: Part-Of-Speech tagging, Combinatory Categorical Grammar tagging and constructive supertagging. We chose to start with Part-of-Speech tagging as it is one of the most fundamental tasks of Natural Language Processing. We then focus on CCG tagging: while being the same task in essence, the underlying grammar of the categories presents sequential dependencies that do not follow a strictly left-to-right order, which the arbitrary ordering could exploit. For example, a transitive verb is usually assigned a *((S \NP) /*

NP) category, indicating that it expects an object to the right and a subject to the left to form a sentence. Before successfully tagging the verb, the model would ideally tag the subject and the object to its left and right. Moreover, the huge number of categories makes it a much harder task when compared to Part-of-Speech tagging. All in all, CCG tagging offers a powerful and linguistically motivated way to parse and understand natural language. It is thus very relevant to accurately perform the task, as it implies the use of models with great syntactical and semantic knowledge of the language used. Finally, we move on to CCG constructive supertagging, which has been the preferred way of going about the task in recent years. In this approach, the complex categories of the grammar are broken down to atomic categories and predicted piece by piece, implying an even greater understanding of the underlying language structure. Moreover, in this task the model is explicitly exposed to the bidirectional dependencies contained in the categories, which may allow for a more fine-tuned order of generation suited for the task. Constructive supertagging also offers an elegant solution to one of the major problems of the task, which is the presence of the so called *long tail*. The issue stems from the fact that many of the complex categories in the formalism are very rare, and some are even not present in the training data. By constructing each category block by block, the constructive supertaggers retain the capacity of predicting virtually every possible category, even the ones it has not been exposed to during training.

By analysing the results on these three experiments, we investigate what the effect of allowing for arbitrary ordering of production is in various contexts. Furthermore, we include an extensive investigation of the order learned by our architecture, providing reasoning as for why breaking the left-to-right assumption could be beneficial. It is, in essence, an exploration of how a Transformer architecture behaves when given the chance of learning its own ordering of generation in a context where the left-to-right one could potentially not be the best approach.

The thesis is organised as follows. In Chapter 2, we first present an overview of the core concepts that are relevant to the research. Namely, we present important aspects of auto-regressive, left-to-right models; the CCG formalism; previous work on constructive supertagging and pertinent concepts we will use to break the left-to-right ordering. In Chapter 3, we present the TagInsert architecture we developed for tagging in arbitrary ordering. Moreover, we will present all the additional models we use in the three tagging experiments, including a constructive version of TagInsert for the task of CCG constructive supertagging. In Chapter 4 we present

the first experiment using the models, trained on the task of Part-of-Speech tagging. We include an analysis on the ordering TagInsert learned as well as beneficial instances of arbitrary ordering. Chapter 5 reuses the same framework but trains the model for non-constructive CCG tagging. Chapter 6 explores how arbitrary ordering can be used in the context of constructive CCG supertagging, presenting the results of the constructive variant of TagInsert. Finally, Chapter 7 summarizes our findings and the future work that can stem from this project.

In the quickly evolving field of Artificial Intelligence, and specifically in Natural Language Processing, our research inserts itself as a way to rethink about the ever so popular Transformer architecture. We contributed to the research in the field by breaking the left-to-right predetermined ordering in such architecture, presenting a more flexible model that allows for arbitrary ordering of generation. By inspecting the effect of our approach in tasks in which there is no strong sequential ordering, we provide a deeper understanding of how flexible generation can improve the sequence-to-sequence model performance.

2. Background and related work

In this Chapter, we will provide most of the notions and concepts relevant to this research. Namely, we start by providing an extensive explanation on auto-regressive models and their common left-to-right order of generation. In doing so, we present the main inspiration of this research, the Insertion Transformer (Stern et al., 2019). The model is a great example of a Transformer-like architecture for sequence-to-sequence tasks that does away with the pre-determined order of generation. Then, we present a more extensive explanation of the CCG formalism that will be used in our experiments. This includes some of the more common ways the task has been handled in the literature. In doing so, we introduce the concept of constructive supertagging, being the preferred way to tackle the task in recent years.

2.1 Auto-regressive models and left-to-right approach

In the context of Machine Learning, auto-regressive models are a class of models that rely on previously generated outputs in order to predict what comes next. Indeed, the reason as to why they are called auto-regressive is because of their dependency on their own output. With the Transformer model from Vaswani et al. (2017) being a prime example, auto-regressive models have been used for a great number of different tasks and domains, such as image generation and Natural Language Processing. An example of the former is Stable Diffusion (Rombach et al., 2022), which is able to transform text prompts to high quality images. In the latter domain, a great variety of auto-regressive models have been successfully used for a staggering amount of tasks, like translation, summarization, question-answering and more.

This class of models has many advantages. For example, they are very flexible in the task they can learn. As long as there is a way to formulate what is needed to be produced as a sequence of elements, auto-regressive models can be used to learn it. The model's reliance on the previously generated data can also be viewed as an advantage, as it helps the model stay more consistent with itself. Naturally, there are also some disadvantages with the approach. Among others,

we choose to focus on the so called *error propagation*. With error propagation we describe a phenomenon caused by the strong dependency on past generations of the auto-regressive models. Basically, as soon as the model makes a mistake during generation, all that will be generated afterwards will be conditioned on an error. Sometimes, this may have little impact, if the first error happens towards the end of the sequence. Some other times, let us imagine a mistake in the very first time step, it can be very impactful and cause a domino-like effect.

Most of the times, especially in Natural Language Processing, auto-regressive model produce their outputs in a left-to-right manner. The choice is made, among other reasons, because of its natural alignment with how we process and produce language as humans. Indeed, when tasked to generate text, it is a very sensible approach to follow. However, the concepts of auto-regressive models and left-to-right generation are distinct, and should not be confused. There is no reason why a model cannot be auto-regressive, while ordering its generation in an arbitrary way. A big focus of this research will be on breaking the left-to-right ordering, and exploring the possible benefits it provides. Most importantly, we will focus on how letting an auto-regressive model choose its own ordering with a sensible criterion could help alleviate the limitation brought by error propagation. Specifically, if the model orders its production by increasing difficulty, it is reasonable to expect the effect of error propagation to be alleviated.

Efforts in this direction have been made through the Insertion Transformer (Stern et al., 2019). The model, designed for machine translation, is a variant on the original encoder-decoder Transformer. The decoder side is designed to perform insertions between tokens in a sequence that is not limited to left to right expansion. By modelling a joint probability distribution on the tokens and slots between them, the model can be taught to generate its output in many custom orderings. Much like the architecture we present in this research, the Insertion Transformer is given the burden to handle both tokens generation and their ordering. Indeed, the architecture has been experimented with in a variety of settings, like text generation (Lu et al., 2021), semantic parsing (Zhu et al., 2020) and Named-Entity Recognition (Ke et al., 2023). As a preliminary research, we implemented the model and trained it for tagging. Unfortunately, the resulting model could not perform for the task. The TagInsert model presented in this thesis was born with the intent of having an insertion-based architecture for tagging tasks. Because of the nature of tagging tasks, we could simplify the more general architecture of the Insertion Transformer,

thus it is not necessary to understand the original architecture of the model to understand TagInsert. However, some overlap is still present. For this reason, we include an explanation of the Insertion Transformer, to introduce the idea behind our research.

2.2 The Insertion Transformer (Stern et al., 2019)

In order to offer a concrete example of a model that escapes the fully auto-regressive setting, we present a detailed description of the Insertion Transformer. The model was a strong inspiration to the work presented in this research, being the skeleton to the architecture that breaks the left-to-right ordering that is later presented.

2.2.1 Basic concepts and functionality

The Insertion Transformer is a variant of the Transformer architecture. It is a semi-auto-regressive model, designed to generalize the original architecture so that arbitrary generation ordering and parallel decoding can be achieved, allowing for more efficient and context-aware processing. Being a more general version of the original Transformer, it can even be trained to perform left-to-right insertions, emulating the original Transformer when the task is deemed to be more suited. While the architecture offers very promising benefits, it also comes with some downsides, like less efficient training. All in all, the model is a promising step forward on breaking the auto-regressive left-to-right approach that is ever so present in most modern architectures, while being relatively under-explored.

Insertion Transformer: Flexible Sequence Generation via Insertion Operations

Serial generation:

t	Canvas	Insertion
0	[]	(ate, 0)
1	[ate]	(together, 1)
2	[ate, together]	(friends, 0)
3	[friends, ate, together]	(three, 0)
4	[three, friends, ate, together]	(lunch, 3)
5	[three, friends, ate, lunch, together]	($\langle$EOS$\rangle$, 5)

Parallel generation:

t	Canvas	Insertions
0	[]	(ate, 0)
1	[ate]	(friends, 0), (together, 1)
2	[friends, ate, together]	(three, 0), (lunch, 2)
3	[three, friends, ate, lunch, together]	($\langle$EOS$\rangle$, 5)

Figure 2.1: Sequence generation in the Insertion Transformer. Left side shows serial generation, in which only one token is inserted at each time step. Right side shows parallel generation, making the model semi auto-regressive. Figure taken from Stern et al. (2019).

Figure 2.1 from Stern et al. (2019) shows the basic functionality of the Insertion Transformer. At each time step, the model needs to decide both on *what* and *where* to insert tokens. Effectively, it has to model a joint probability distribution over

Trajectories	L2R Transformer	Insertion Transformer
t = 1	<s>	love
t = 2	<s> I	pizza !
t = 3	<s> I love	I pizza
t = 4	<s> I love pizza	I love !

Figure 2.2: Comparison of a potential trajectory for the sentence "I love pizza !" between the original Transformer and the Insertion Transformer. The original Transformer has a fixed trajectory determined by the left-to-right order, while the Insertion Transformer has to randomly sample subsequences to process at each time step.

tokens and positions. In order to achieve this, the original Transformer architecture needs to be modified. The original Transformer already provides a way to generate a probability distribution over tokens, but a method to represent all possible positions in which a token can be inserted is also needed. In their work, the authors conceptualize each position as a slot, which is effectively a gap between two tokens. Assuming slots can be meaningfully represented and there is an effective way of producing a joint distribution of both tokens and slots, the architecture can be trained to perform any kind of ordering. From Figure 2.1 it is also possible to see one of the main motivations of the work: parallel insertions. By making the model able to predict more than one token at each time step, the model shies away from a full-on auto-regressive structure.

Figure 2.1 shows how the model will need to iteratively insert new tokens in a progressively increasing sequence. To train it with meaningful data, it is necessary for it to be exposed to many different subsequences of the original training set. Indeed, in contrast with the left-to-right framework, it is not possible to anticipate how generation will happen at inference time. Many different combinations of subsequences from the target should be considered to get the most out of training. For this reason, a randomly sampled subsequence is calculated at each time step during training. We call the set of subsequences forwarded to the model a *trajectory*. Figure 2.2 shows the difference between trajectories in the original Transformer and the Insertion Transformer.

Each subsequence that makes up the trajectory needs to be forwarded to the decoder at each time step, which is a big difference with the original Transformer, in which the pre-determined order made it possible to process all subsequences in one go. That was possible because of the unidirectional expansion of the trajectory, which guarantees that the absolute positional encoded is unchanged between time

steps and enables the reuse of previously processed subsequences.

The model can be trained to generate tokens in many different orderings. In the paper, the authors experiment with a traditional left-to-right ordering, a balanced binary tree ordering designed for parallel insertions and a uniform ordering in which the model is more free to learn the ordering by itself. What determines the type of ordering the model will learn is the loss function. Thus, for each ordering there is a corresponding loss function that teaches the model the best way to perform insertions.

The next few Sections will focus on explaining how all of the concepts mentioned have been implemented in practice, giving a full overview of the architecture and design choices behind the Insertion Transformer.

2.2.2 Representing slots and modeling the joint distribution

Representing slots As explained, to make it possible to perform an insertion, the model not only needs to predict which token to generate, but also the position in which said token is to be generated. For example, given a training sequence $y = [x_1, x_2, x_3, x_4]$, a valid subsequence would be $\hat{y} = [x_2, x_4]$, and the model would learn to represent slots between such tokens. This way, the model can make informed decisions on the positions in which to perform an insertion which are also based on their surrounding tokens. The authors exploited the positional representations the decoder of the original Transformer already produces before the final output layer. The slots representations are calculated by concatenating each adjacent positional representation: by averaging them it is possible to build vectors that aggregate the information on their left and right positions. Additionally, in order to represent the slots at the beginning and at the end of the current sequence, two special tokens are added to the decoder input. Thus, if the current subsequence has length N, the Insertion Transformer ultimately produces N+2 vectors encoding the information on each position of the subsequence (special tokens included), as well as N+1 vectors encoding the information on each slot between words in the subsequence.

Joint distribution Once each token and slot has its own representation, all that is left to train the model is producing a joint distribution over both. The authors offered a couple of ways of calculating it. For example, it can be calculated by multiplying the newly extracted slots representation matrix and the standard soft-

max projection matrix in the final layer of the original Transformer. This operation effectively calculates the logits of both tokens and locations combined, which are then flattened and softmaxed in order to obtain the desired joint distribution. Thus, given a hidden size H, a vocabulary size $|V|$, the slots representation matrix $S \in \mathbb{R}^{(N+1,H)}$ and the standard softmax projection matrix $W \in \mathbb{R}^{(H,|V|)}$, the joint probability distributions over content C and locations L is defined as

$$p(C, L) = Softmax(Flatten(S \cdot W)).$$

Figure 2.3 shows a complete diagram of the architecture of the Insertion Transformer. Block B of the diagram shows how each adjacent positional vector of the current hypothesis sequence calculated by the decoder is aggregated, to then perform a matrix multiplication with the softmax projection matrix and obtain the joined probability distribution. The Figure includes a running example in which a single subsequence is processed. Calculation of the loss function is also included and is expanded on in the next sections.

Trajectory sampling As mentioned in the previous Section, one of the main differences that stems from abandoning a pre-determined order of generation is the necessity of generating the subsequences to feed to the decoder at each time step. Because there is no way of knowing how the sequence will be produced a priori, the model needs to be exposed to many different subsequences of the full target sequence. Thus, given a sequence, the authors randomly sampled a subsequence of the full target at each time step. The way the subsequence is sampled can be flexible, but in their work the authors simply extracted a random number of indices from a uniform distribution. Block A of Figure 2.3 shows an example of how the sampling is done in the Insertion Transformer.

2.2.3 Loss functions

Having calculated the joint distribution, the model can be trained to perform insertions of tokens over all slots of the sequence. Depending on the loss function, different orders of the positions in which the tokens are inserted can be learned. Namely, it is possible to train the model to make insertions in a traditional left-to-right manner, or in more fancy orderings, starting in the middle of the sequence

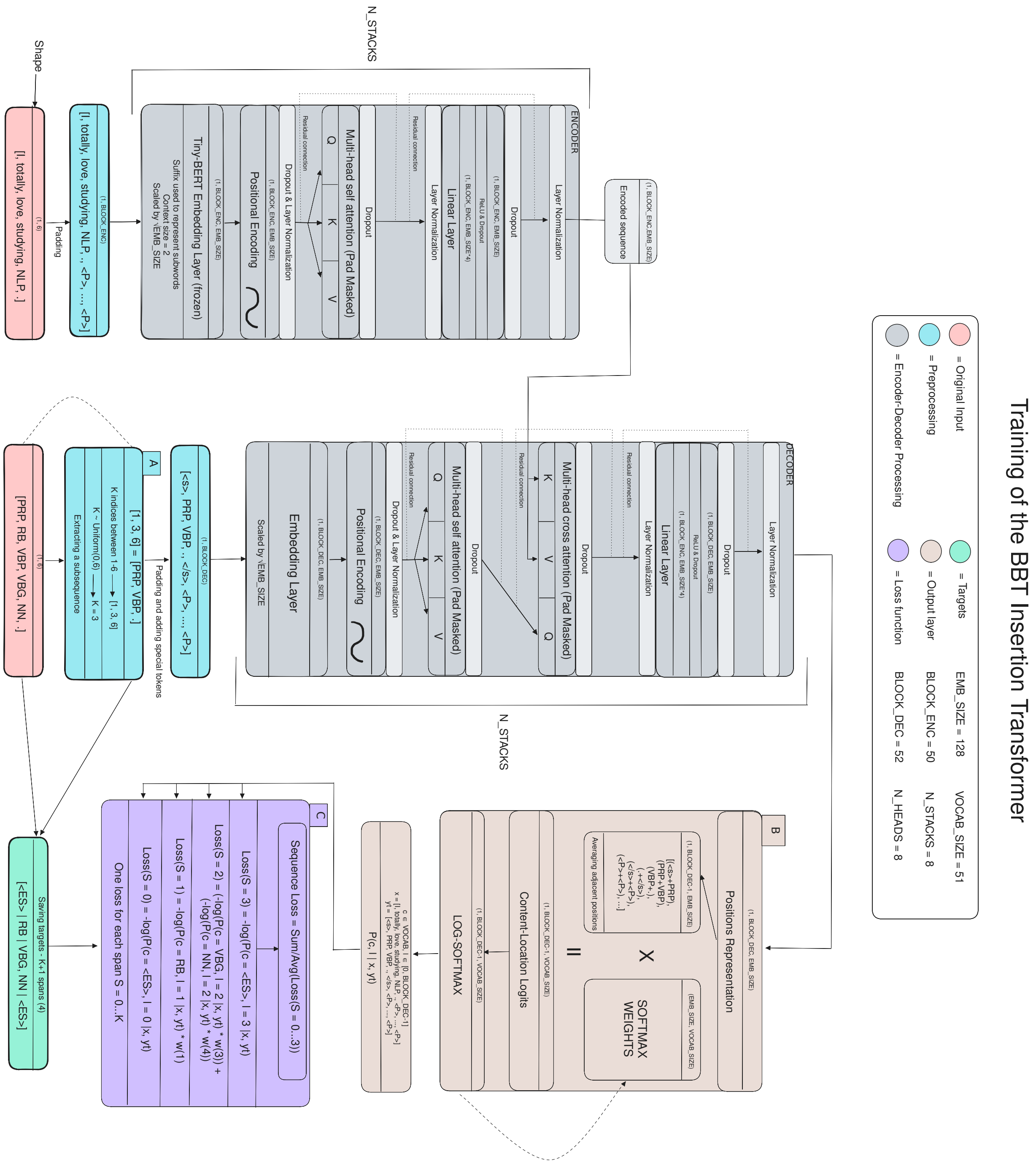

Figure 2.3: Visualization of the training procedure of the Insertion Transformer during a single trajectory step. Block A shows the process of handling trajectories. Block B shows how the joint distribution between slots and tokens is calculated. Block C shows how the Balanced Binary tree order is obtained through the loss function.

and then inserting in parallel in both directions. Because of this feature, the Insertion Transformer can be seen as a more general implementation of the original Transformer architecture. Crucially, in order for the model to insert in all positions of the sequence, the subsequent attention mask present in the self-attention

mechanism in the original Transformer decoder needs to be removed, so that all positions can attend to all the other positions and not only to the ones that follow the pre-determined left-to-right attention mask.

The design of the loss function and the training procedure are the areas in which the model differs the most from the original Transformer architecture. During training, the vectors corresponding to each position in the final decoder layer need to be recalculated after each insertion operation. The reason for this is that each insertion can potentially add new tokens anywhere in the sequence, changing the absolute position of all tokens. In contrast, in a traditional auto-regressive architecture the previously generated positional vectors can be reused, as the information used to calculate them does not change by just expanding the sequence to the right. This key change makes it so the loss cannot be calculated simultaneously for all the generation steps, and positional representations need to be re-computed after each insertion operation. In this view, a sequence of length n as to be forwarded to the decoder a total of n times for the full loss to be calculated. This is a big difference with the original Transformer, in which a single pass is sufficient. Naturally, this impacts the training efficiency of the model. However, parallel generation somewhat balances it out by allowing for faster decoding times, as a sequence of length n can theoretically be generated in at minimum $\log_2 n$ insertion steps. The authors experimented on **three different orderings**: a traditional left-to-right order, a balanced binary tree order and a uniform order, explained in brief below.

1. Left-to-right order To train the model to insert in a left-to-right fashion, the loss function is defined as the negative log-likelihood of inserting the correct next token x in the correct next position k, given a sequence truncated at a uniformly sampled position. Formally, given an instance (x, y) and a $k \sim Uniform(0, |y|)$ indexing where the truncation is to be performed, the loss is defined as

$$Loss(X = x, Y = y_1...y_k) = -\log p(y_{k+1}, k | x, y_1...y_k),$$

where $p(C = c, L = l | X = x, Y = y)$ is the joined probability distribution modeled by the decoder for inserting content c in location l. K is sampled from a uniform distribution so that the model is exposed to sequences of many different lengths. If $k = |y|$, y_{k+1} is set to be the special end-of-sequence token.

2. Balanced Binary Tree The balanced binary tree ordering encourages the model to insert tokens in the center of the spans between previously inserted tokens. In the context of the Insertion Transformer, a span represents a slot in which any number of token is to be inserted. In contrast with a left-to-right order, this approach allows for multiple parallel insertions. Given an instance (x, y) and a $k \sim Uniform(0, |y|)$, a subsequence $\hat{y}$ of length k is sampled by extracting random indices from the full sequence. The extracted indices are then sorted and used to construct the subsequence. Block A in Figure 2.3 shows the sub-sequencing process, first extracting the subsequence length k and then extracting as many indices from the original sequence. The loss of each slot is then defined as

$$SlotLoss(X = x, Y = \hat{y}, L = l) = \sum_{i=i_l}^{j_l} -\log p(y_i, l|x, \hat{y}) \cdot w_l(i),$$

where $p(C = c, L = l|X = x, Y = y)$ is the joined probability distribution modeled by the decoder for inserting content c in location l; i_l and j_l are the indices for the start and end of the span in location l; and $w_l(i)$ is a weighting function for the index i of the span in location l. The weighting function $w_l(i)$ is defined as the softmax of the negative distance of index i from the center of the span in location l

$$w_l(i) = \frac{exp(-|\frac{i_l+j_l}{2} - i|/\tau)}{\sum_{i'=i_l}^{j_l} exp(-|\frac{i_l+j_l}{2} - i'|/\tau)},$$

where τ is a temperature hyperparameter to adjust the weighting influence on the loss. Effectively, this loss makes the model more focused on the center of the spans thanks to the weighting function. For example, given a sequence $y = [x_1, x_2, x_3, x_4, x_5]$ and a subsequence $\hat{y} = [x_1, x_5]$, the model will likely start decoding by inserting x_3 between x_1 and x_5. The loss of the whole insertion operation is then simply defined as the average slot loss in each position p.

3. Uniform The uniform order aims to insert tokens into spans with equal probability, without assigning higher weights to their centers. In practice, the loss function is calculated the same way as the balanced binary tree order, with the only difference being the τ hyperparameter being set to ∞.

2.2.4 Termination conditions

Finally, both balanced binary tree and uniform orderings need a termination condition when generating. The original paper explored two approaches, slot finalization and sequence finalization. In slot finalization, when a span is empty the loss is computed using a special end of slot token, and during inference generation terminates as soon as all slots predict this special token. In sequence finalization, loss is not calculated for empty spans. Once all spans are empty, the loss is defined as the negative log-likelihood of a special end of sequence token.

A visualization of the loss calculation using slot finalization is shown in Figure 2.3: in Block C, a loss is calculated for each slot in the subsequence as the weighted sum of the negative log-likelihoods of inserting each missing target in that slot. Weights are calculated depending on the ordering chosen (Balanced Binary Tree will put focus on the center token of each span while uniform will not have any preferences). All the slot losses calculated this way are then averaged to get the complete subsequence loss.

2.2.5 Recent Work using the Insertion Transformer

Recently, the Insertion Transformer architecture has been used to perform various tasks. This section shortly reports some of this research, covering the most notable results. In order to improve on the overhead needed to recalculate positional vectors at every generation step, Zhang et al. (2023) proposed a dynamic way of extracting positional encodings of tokens (Fractional Positional Encoding). Essentially, each encoding is calculated by exploiting the encoding of the left and right neighbors, ensuring that the relative positions of the tokens stays the same through all the generation process. In their work, they reported similar results to the original implementation, making it promising for the Insertion Transformer to be more efficient. The architecture is deployed for text generation in Lu et al. (2021), in which the authors improve on both efficiency through relative positional encoding and a more adaptive parallelization of the insertion operation.

As an attempt to generalize the model even more, Ruis et al. (2020) proposed an Insertion-Deletion framework, in which tokens can be deleted rather than only inserted. This feature makes it possible for the model to correct mistakes made in previous generation steps. In their results, they show a performance improvement on simple tasks like shifting alphabetic sequences. Notably, this model shares many

similarities with the Levehnstein Transformer, another architecture that focuses on both insertion and deletion operations (Gu et al., 2019).

In the field of semantic parsing, Zhu et al. (2020) reached new states of the art with a non-auto-regressive approach based on the Insertion Transformer. Finally, the non-auto-regressive approach proved relevant in data augmentation in Ke et al. (2023), where the authors use the framework to generate more syntactic aware NER (Named-Entity Recognition) data.

2.3 Combinatory categorical grammar

Combinatory Categorical Grammar (CCG) is a formalism in which each word of a sentence is assigned a lexical category. Each category can either be atomic or complex. Atomic categories are the simplest building block of the category language, like S, N or NP. On the other hand, complex categories effectively define functions of atomic categories. Through the use of the slash operators ('\' or '/'), atomic categories are combined to form *supertags*. Each supertag can carry a rich representation of the function of each word in the sentence. Moreover, as supertags can be recursively combined, the number of unique categories can grow to a very large amount. Once each word is correctly tagged, the sentence can be easily parsed through combinatory rules that merge supertags to ultimately derive an atomic category for the whole sequence. Thus, assigning the correct lexical category to each word is the most crucial aspect of the task, which has proven to be quite challenging as a consequence of the large number of complex categories that can be formed.

2.3.1 CCG supertagging

Traditionally, most supertaggers follow the recent trend of fine-tuning a pre-trained transformer based neural model so that it learns to sequentially predict categories for each token in a sequence. For example, Tian et al. (2020) added a BERT pre-trained encoder to a Graph Convolutional Neural Network (GCNN) and trained it for CCG supertagging. Normally, these models are pre-trained for language modeling, which is a highly generic task that allows for holistic knowledge about the target language to be learned. Fine-tuning these large language models makes it possible to exploit such knowledge and skew it in favor of the desired task, yielding staggering results. For example, one could use a pre-trained BERT encoder to produce highly context-aware embeddings and simply add an output layer to map

each token to a supertag. While such models showed impressive results, it is worth noting that they have two important limitations. First, they are usually trained by frequency-cutting the long-tailed distribution of the supertags in the data. Indeed, when using CCG, the high number of categories that can be constructed is a common challenge when it comes to learning about the rarest supertags. Essentially, the high performances shown may be slightly deceptive, as the models get very good at predicting 'easy' categories, but struggle to predict or outright ignore rare ones, which make up a large percentage of the supertags. The second limitation pertains to the recursive structure of the CCG supertags, which is ignored in such models. Indeed, simply fine-tuning a pre-trained model results in treating the target labels as opaque categories, ignoring the fact that some are a complex construction of several building blocks from the language. For instance, a transitive verb is usually assigned a *((S\NP) / NP)* category, indicating that it expects an object to the right and a subject to the left to form a sentence. These syntactic notions that come with each supertag may help with the task of supertagging, and should ideally be exploited instead of being ignored. Furthermore, using this approach makes it impossible for the model to predict supertags that do not occur in the training set. Finally, by making this concession, it is implicit that such a model would perform as shown only in grammars with a small set of heavily used categories, which may not be true for all languages. Research has been made to solve these issues, and some tentative solutions are presented in the next section.

2.3.2 CCG parsing

Once a sequence has been supertagged, all that is left is to parse it. Notably in CCG, the categories are so informative and inherently resolve syntactical and positional ambiguities that some of the work of the parser is already done during tagging. Thus, the focus in this task is usually on the sequence tagger. Regardless, the job of the parser in this case is to combine the assigned categories using the CCG rules, to form a tree that represents the syntactical analysis of the sequence. During the process, each analysis retrieved must be scored and a search algorithm must be deployed to efficiently find the best scoring parse. To do the parsing, one could use a chart-based algorithm like CKY (Clark et al., 2002) or a shift-reduce algorithm (Ambati et al., 2016). Furthermore, it is also possible to adapt a transformer-like model to perform the task, by making the model produce embeddings not only for the tokens, but also for the spans between each of them (Stern et al., 2017). Once

the model has learned an internal representation of both, it can explore the different parsing branches and score each span accordingly. To search the best performing parse, one could use a beam search algorithm, which performs a parallel search on the possible sequence space by pruning on the top performing ones, or a A* search algorithm, which limits the exploration based on heuristic function that estimates how each currently generated parse is distant from the optimal one (Lewis and Steedman, 2014). Finally, regardless of the method, it is apparent that the performances of a CCG parser are very reliant on the accuracy of the CCG supertagger, hinting on the fact that once the most difficult challenges on supertagging are overcome, the parser will also improve on their performances.

2.4 Constructive supertagging

Having explained the task of supertagging, the most simple way to solve it would be to train a model to predict each supertag as if it was a fixed label, just like a classic Part-Of-Speech tagging task. As mentioned in the previous section, this is the approach that most supertaggers currently use. While this approach would perform well for the frequent categories encountered during training, it is a method that yields an inherent weakness when it comes to rare or out of vocabulary supertags. In contrast, POS tagging is a simpler task, in which the number of tags to predict is low. Namely, the Penn Treebank, which is one of the most used datasets for the task, includes 36 POS tags, plus 12 for punctuation. The CCGBank, containing the same sentences as the Penn Treebank, includes 1286 supertags in section 02-21 only, with an average number of 19.2 different categories for each token (Hockenmaier and Steedman, 2007). With so many categories, the distribution is heavy-tailed, with a high number of supertags being rare. A common approach to solve the issue is to truncate the rarest categories with an arbitrary threshold. However, this mostly just serves as band-aid rather than solving the underlying complex task that is supertagging. Thus, using the simplest approach not only would fail to overcome this problem, but would also not exploit the internal structure the CCG formalism introduces, as mentioned before. Indeed, each supertags is built recursively and have an internal compositional structure. This points to the importance of constructive supertagging, in which these complex categories are not just predicted as fixed labels, but are instead constructed at inference time through modeling their structure. Such an approach would make it so that no truncation of data is needed, and would make it possible to correctly construct rare supertags by learning to

exploit learned properties of the structured categories. Examples of prior works that make use of this approach include a top-down constructive supertagger which models each supertag as a tree (Prange et al., 2020); and an inductive constructive supertagger, which learns a context-free grammar used to build categories (Kogkalidis et al., 2019).

2.4.1 Constructing categories (Kogkalidis et al., 2019)

In their paper, Kogkalidis et al. (2019) use a standard Transformer architecture to produce categories of a categorical grammar. While the grammar of choice is not CCG but rather a type-logical one, they are both categorical grammars, and as such similar in nature. The transformer model turns the input words to the encoder block into types from a fixed vocabulary of atomic categories, which then form the rules (or supertags) of the grammar. Thus, the task is essentially a sequence-to-sequence task. The rules produced in the output sequence are in polish notation, making it possible to easily separate and retrace them. In their method, they had to extract the rules of the grammar from a corpus first, so that each word can be assigned a category. Then, through the use of ELMo embeddings, they encode the input sequence and pass them through the encoder block. The decoder block then simply predicts the next atomic symbol to be produced, which can either be a type of the grammar or a special separator token. This approach makes it possible to produce every possible supertag that can be built using the atomic categories given by the grammar, so that the model can be trained and evaluated on the whole corpus, rather than arbitrarily cutting the rarest categories. In their experiments, the authors show that such an architecture can perform and has the potential to remove the need to ignore rare categories from the training and evaluation process. In their analysis, they showed that the model outperformed all the traditional constructive models on the rare supertags. Not only that, but it is also proactive in producing unseen categories, with a remarkable 44% accuracy when doing so. It is noteworthy that the model was trained on a Dutch corpus, which has a more complex syntax than English. Finally, it is also notable how the model seemed to achieve an understanding of the given grammar, with all predicted types having the correct structure and showing higher-order consistency on unseen supertags.

2.4.2 Exploiting the tree structure (Prange et al., 2020)

In their paper, Prange et al. (2020) seek to exploit the structural nature of CCG supertags in order to train a model to construct such categories in a top-down hierarchical manner. Namely, they aim to teach the model to make decisions that are syntactically informed, as when building the tree predictions on new labels to attach to the tree would take into account previously predicted ones and its positions.

In their method, the authors train their model to construct each tree starting from the root, which can be either a single atomic category (e.g. NP) or a slash operator ('\' or '/'), indicating the outermost operator. Every time a slash operator is produced, the node is considered non-terminal with exactly two children, that can in turn be single atomic categories or slash operators. Every time a single atomic category is produced, it is considered a leaf, terminating the production on the branch. This way, the tree generation process always yields structurally sound supertags. In order to predict each node, the model makes use of the node's ancestors, its position and the current word's embedding. The word's embedding are produced by using a fine-tuned RoBERTa encoder. The authors experimented with two different methods of using these information to represent a node. The first method (TreeRNN) makes use of a structure reminiscent of RNNs to calculate the representation based on the node's parent's label and its (previously calculated) representation. The second method (AddrMLP) builds a feature vector based on a depth-first tree traversal, in which the values are dependent on the label of each traversed node. A linear layer is then used to project the feature vector in the space of the word embedding, to then add them together and produce the node representation. Regardless of the method, once a representation of a node in a certain position of the tree is produced, a MLP which includes a softmax output layer on all possible atomic categories and operators is used for the prediction. In their experiments, the authors compared performances with different types of supertagger: some ignoring rare supertags in a more traditional manner, while others using the novel constructive approach. In general, their model could match performances on frequent categories while improving on the long-tail. TreeRNN and AddrMLP are found to have different behaviors and some interesting perks. An extensive analysis on the performance on the long-tail distribution of the supertag is presented in the paper. While both variants showed improvements on rare categories, AddrMLP was especially noteworthy as it could outperform all traditional taggers in the long tail of the distribution, while matching their results in the frequent su-

pertags. Additionally, the results show that the models are indeed making use of their ability to generate supertags, rather than just imitating what was seen during training, with the most common mistakes in the process being a single atomic categories or slash when compared to the gold label.

2.5 Breaking the left-to-right approach

As the focus of this research relies on breaking the left-to-right ordering frequently used in auto-regressive model, we provide an overview of how such models work and what exactly makes them left-to-right. Specifically, we will focus on the Transformer model and the attention mechanism that made it so popular, which plays a role on the unidirectional ordering. Once these aspects are clear, it will be easier to show how we designed the TagInsert model. Indeed, the left-to-right ordering needs to be enforced by the architecture, and there are two key elements that are responsible for it: *masked attention* and *next token prediction*.

Masked attention In essence, the breakthrough of the Transformer was the idea that the attention mechanism was all you needed to model dependencies in a given sequence. Indeed, what attention offers is a communication tool, which is not hindered by long-range dependencies between tokens. As a communication mechanism, all tokens are allowed to exchange information between each other, creating a communication network that is invariant to position. However, when we want to train our models, we need to block some communication channels so that *future tokens* are not involved. In the context of left-to-right ordering, future tokens refers to elements of the sequence that are further along the current time step, which moves from left to right. As what we intend with *future* is determined a priori in this setting, it is easy enough to know which communication channels to block at each time step, and the resulting mechanism is called *masked attention*. Figure 2.4 serves as a simple visualization of the attention mechanism and how masking transforms it. At the time-step showed on the right of the Figure, the future is comprised of the tokens *pizza* and *!*. Thus, those tokens are left out of the communication, and training follows the proper flow. On the other hand, the left side of the Figure shows how unmasked attention would look in this case. All tokens can communicate with each other, and at each time step we cannot determine what the future is, making it impossible to block communication properly. Masking attention is the first step to enforce a strict left-to-right ordering to generation. Ultimately, we *can* mask atten-

tion *because* we decide a priori that generation will follow the left-to-right ordering.

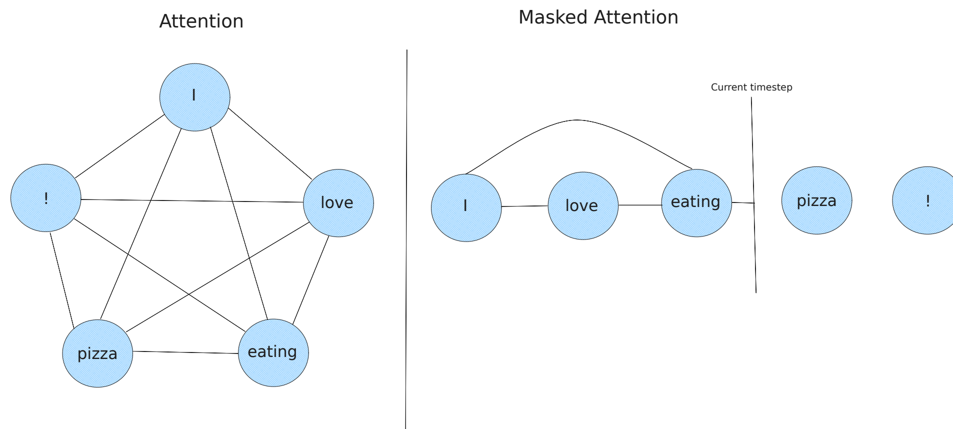

Figure 2.4: Attention and masked attention. By introducing time steps and left-to-right ordering, we can block communication between 'future' tokens.

Predicting the next token The other key element to enforce the ordering is to only consider the position to the immediate right of the current time step when predicting a target to generate. This effectively teaches the model to predict the *next best token*. It is essentially sparing the model of the burden of handling the order of generation. By deciding it ourselves a priori, the model can just focus on the next time-step. Referencing Figure 2.4 again, this concept translates to only considering the vector corresponding to the token *pizza* when training at that time step. The position corresponding to the token *!* will be considered at the next time step.

These two concepts work in tandem to create a mechanism in which the ordering is pre-determined.

Having a clearer picture of how the left-to-right ordering is enforced, we now ask ourselves what would need to happen for the ordering to be decided by the model and not a priori. The first implication is that we would have no a priori concept of what *future tokens* are. Thus, we could no longer easily mask attention. The second implication is that the model would need to be able to decide which token to predict next, rather than being forced to predict the token to the right of the current time step. Thus, at each time step the model would need to be given access to the full sequence to make that decision. Ultimately, by unmasking attention and exposing the model to the full sequence, we could relieve the model from the left-to-right generation. However, there are two problems that needs to be solved first.

A By unmasking attention we allow for communication between all of the to-

kens, even the ones that the model should not have access to yet.

B Having access to the full sequence to decide which token to predict next, one needs to decide on a criterion for the choice to be made.

In the next Chapter, we will present the TagInsert model and how it solves both of this issues in order to allow for arbitrary ordering of generation.

3. Models

In this Chapter, we present the models used for the tagging experiments we designed, with a focus on our novel TagInsert architecture, which modifies the original Transformer in order to allow for arbitrary ordering of generation. First, we will present the models that will be used for Part-of-Speech and CCG tagging in Chapter 4 and Chapter 5 respectively. For each model, we mention the motivation behind their use and briefly present their architecture, training methods and inference procedure. For TagInsert, we also present a detailed explanation of the design and methods that are relevant to the architecture. As Part-of-Speech and CCG tagging are essentially the same task, we used the same models to analyze whether the bidirectional sequential dependencies present in the CCG formalism would be better exploited when compared to POS tagging. Then, we present the models for CCG constructive supertagging, including a variant of the TagInsert model adapted for constructive supertagging. The model uses part of the mechanism of the supertagger presented in Prange et al. (2020). Thus, we also provide motivation and an explanation of the constructive model.

3.1 Models for Part-Of-Speech and CCG Tagging

For the task of Part-of-Speech tagging (Chapter 4) and CCG tagging (Chapter 5), four different models were trained:

1. Finetuned DistilBERT
2. Vanilla Encoder-Decoder Transformer
3. TagInsert
4. TagInsert left-to-right

The first two models are essentially baseline models we use to compare performances with TagInsert, in order to inspect the potential benefits of the arbitrary ordering it offers. The finetuned DistilBERT model is a highly flexible architecture that proved to be successful in a wide range of tasks. The Vanilla Encoder-Decoder Transformer was chosen to explore how a sequence-to-sequence model

can perform on a tagging task. This is an important step, as TagInsert is also an encoder-decoder model. The TagInsert model is a novel architecture that breaks the traditional left-to-right order of target generation, choosing its own preferred ordering. TagInsert left-to-right is a proof of concept model that is halfway between TagInsert and the Vanilla Encoder-Decoder Transformer, in which it retains the left-to-right ordering of the latter. By comparing results of the two TagInsert models, our intention is to analyse the benefits of breaking the traditional ordering of decoding.

3.1.1 Finetuning a BERT model

Motivation A common approach to solve various Natural Language Processing tasks is to make use of the holistic knowledge aggregated in Large Language Models (LLM). Namely, finetuning a pre-trained language model in order to make it proficient in the task of choice has been proven to be quite effective (Devlin et al., 2019). In the context of this research, this approach serves to provide a baseline model that is suitable and often used for tagging tasks, as opposed to the Vanilla Encoder-Decoder Transformer. By comparing the results of this model and the results of the encoder-decoder based models, it will be possible to analyse the effectiveness of the sequence-to-sequence models as taggers.

Architecture In the case of finetuning a BERT-like model, the standard approach is to add an additional output softmax layer to the transformer-based encoder block, making it so the encodings produced by the model for each word can be used to model a probability distribution over all possible tags. By doing this, each word is mapped to exactly one tag, and the behavior of the model is much more similar to that of an actual tagger. As for the ordering, because the BERT model is an encoder block, it is not an auto-regressive model and there is no sequence generation happening.

3.1.2 Vanilla Encoder-Decoder Transformer

Motivation When talking about auto-regressive, left-to-right decoding models, the original Transformer from Vaswani et al. (2017) is perhaps the most relevant at present. Core concepts from it are used in many of the major Natural Language Processing applications. For example, the ubiquitous family of GPT models (Radford and Narasimhan, 2018) uses a slightly customized version of the decoder pre-

sented in the original paper. In the context of this research, the model was chosen in order to test how an architecture that was originally intended to do sequence-to-sequence tasks can perform for tagging. It is important to establish what the limitations are when using such models for the task, as the TagInsert model that allows for arbitrary ordering that is later introduced is based on the same architecture. Once the limitations are clear, it is possible to better analyze the effect of breaking the left-to-right ordering on a task in which it is not clear how forcing such an ordering would help the model. Thus, this Section presents the Vanilla Encoder-Decoder Transformer, as it was implemented in the original paper and deployed for Part-Of-Speech tagging.

Architecture In the original paper, the Transformer model was used for machine translation. In general, the architecture can be used for any sequence-to-sequence task, and for this reason is comprised of an encoder side and a decoder side. The encoder side takes the source sequence as its input and, through the multi-head attention mechanism, provides an encoding of the sequence that ideally accounts for all the dependencies between tokens. Indeed, in the encoder, attention is unmasked, meaning that all the tokens are free to consider all the others in the sequence when computing attention weights. On the other hand, the decoder side takes the target sequence as its input. As the decoder is tasked to generate the sequence in a left-to-right manner, the input sequence is shifted right through the use of a special start token, and the model is effectively trained to predict the next token to the right at each time step. As mentioned, if the desired behavior of the model is to generate tokens in a predetermined order, masking becomes necessary so that the model is not pre-emptively exposed to tokens it should not have access to (yet). After going through a number of self-attention (contained to the target sequence), cross-attention layers (extended to the source sequence as produced by the encoder) and linear layers, the decoder generates logits over the output labels for each position in the sequence. Then, the model is trained by means of the loss function of choice to minimize the error when greedily predicting the token from the logits of the *next position*. Of course, in the context of Part-of-Speech tagging, the encoder will take the word embeddings from the previously mentioned DistilBERT model as its input, while the decoder takes the POS tags. Then, the decoder is trained to generate one tag at each time step in a left-to-right manner. Appendix B shows a visualization of the Vanilla Encoder-Decoder Transformer for completion.

Tagging as sequence transduction This is a crucial aspect for the premises of our experiments: when using an encoder-decoder architecture to perform a tagging task, we are re-imagining the concept of tagging into sequence transduction, where a sequence of words is transformed into a sequence of tags. In practice, this means that the basic Transformer model has no concept of tagging. That is, there is no explicit dependency between a word and its tag, other than the fact that they are in the same position in the sequence and what the cross-attention mechanism can capture. The model simply learns to produce a target tag sequence that is somehow coherent with the input word sequence. Then, it stops when the amount of tags generated is the same as the amount of words it was fed on the encoder side. While this does not mean that the model is unable to behave like a tagger, one might argue that there should be a more explicit connection between a word and its tag. In TagInsert, we will provide a solution for this issue. This will also help to make the comparison between left-to-right and arbitrary ordering more focused.

Training Training is kept as close as possible to the original Vaswani et al. (2017) paper. The loss function of choice is Cross Entropy Loss, and it is calculated on the logits of each subsequent position as they are produced by the decoder. This teaches the model to maximize the accuracy of the next token prediction, and makes it possible to decode in a left-to-right manner. It is important to note that a Transformer model exploits the fixed ordering by calculating all the logits in a single pass to the decoder, making training very efficient. In principle, every loss calculation corresponds to a tag generation that is added to the sequence. The newly added tag has the potential to change the logits produced by the decoder and should require another pass at each time step. However, in this specific case, all can be done in one pass, as the order of generation is predetermined and the attention mechanism is masked, making sure that each position's logits are only dependent on what has been generated in previous time steps.

Inference For inference, the encoder is fed the source word embeddings, while the decoder is fed a single special start token. Each time step, a tag is greedily predicted from the probability distribution produced by a softmax output layer for the next position. The sequence is then expanded this way until the number of tags predicted equals the number of words in the source sequence.

3.1.3 TagInsert

Motivation In this Section we present a novel architecture to perform tagging with arbitrary orderings called TagInsert. The model was inspired by the Insertion Transformer (Stern et al., 2019) presented in Chapter 2, in which it does not follow a left-to-right order of decoding. By solving **Problem A** and **Problem B** presented in Chapter 2 in order to allow for arbitrary ordering of insertions, we explore what are the effects of breaking the commonly used left-to-right order of generation. We then compare the results to a more traditional tagger (finetuned DistilBERT) and an encoder-decoder, left-to-right model (Vanilla Encoder-Decoder Transformer), aiming to show that using a left-to-right approach is not always the best way to tackle a task. Rather, the choice should be made depending on inherent properties of the problem that is at hand. That is way, for instance, we expect the model to be able to exploit the preferred ordering expressed by the supertags in the context of CCG tagging.

Breaking left-to-right decoding

The idea behind the model is that at each time step t, both a tag and its respective position are chosen to be inserted in a target sequence partially filled with t tags. Each one of those tags can be anywhere in the sequence, breaking the left-to-right ordering. For this to be possible, we need to unmask the attention mechanism in the decoder and give the model access to all the position in the target sequence at every time step. Like mentioned, this implies solving **Problem A** and **Problem B** introduced in Chapter 2.

First, we solve **Problem A** by exploiting a crucial property of any tagging task: the length of the target sequence is fixed and known a priori, unlike what happens in a language generation task. By knowing exactly how many tags we will be predicting, we can use placeholder tokens that indicate positions for which a tag is yet to be inserted. Those tokens are never seen in training and will not provide any relevant information to the attention mechanism. This can be seen as a workaround akin to masked attention, but instead of shutting down communication channels, we mask the tokens that are still considered *future* (yet to be inserted) at each time step. The placeholder tokens we use are annotated with *<UNK>*. The unmasked attention mechanism is visualized in Figure 3.1 for a time step $t = 3$.

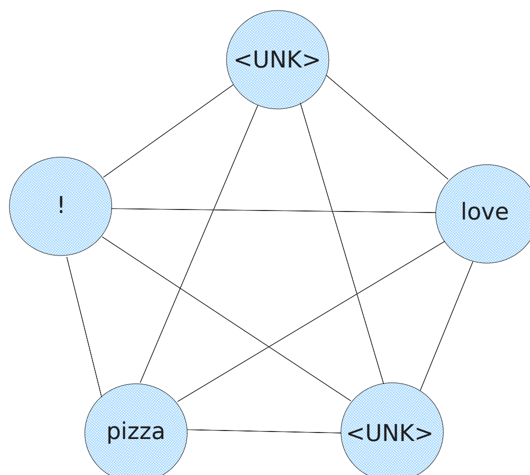

Figure 3.1: Unmasking attention by masking the tokens. The figure depicts a possible timestep (t = 3) using the unmasked attention mechanism in TagInsert, following the example sequence in Figure 2.4. The figure shows words rather than their corresponding tags words for the sake of clarity.

Next, we solve **Problem B** by giving the model a criterion to choose the next token and position to insert. Once the target sequence has been processed by the decoder and the model goes through the output layer, each position is mapped to a probability distribution over all possible tags. This also happens in the Vanilla Encoder-Decoder Transformer but, because of the left-to-right enforcing, only the next position to the immediate right is considered. We let the model decide which token to predict next by simply extracting the maximum probability token over all positions corresponding to an *<UNK>* token. This way, the model will learn to choose the tag it is most confident with at each time step, and the position will follow. Other ways to decide the next word to tag are also possible,

Having solved both proposed problem, the model can now generate the target sequence with dynamic ordering, and the left-to-right decoding is not enforced anymore. The following is an example of decoding at an arbitrary time step $t = 3$

<s>	<UNK>	PRP	<UNK>	VBN	CD	<UNK>	<UNK>	</s>
			Insertion here					
<s>	<UNK>	PRP	VBZ	VBN	CD	<UNK>	<UNK>	</s>

Example 1: Decoding a sequence at time step $t = 3$. The model greedily decides that the tag at position 3 will be inserted next.

In the Example, the model is exposed to a number of tags and *<UNK>* tokens. The <UNK> tokens effectively mask the attention mechanism for future tokens and

the model chooses the maximum probabilty tag across all position, which was the *VBZ* tag in position 3. After the insertion is performed, the model can proceed to the next time step. As depicted, the target sequence also contains special start and end sentence tokens. In principle, these are not needed as generation can start at any position and can end as soon as all the special <UNK> slots have been filled. However, it might be that the model learns specific correlations with tags near the start and end of the sequence. Thus, the tokens are kept, as they do not interfere with the learning process.

Architecture

The architecture is that of an Encoder-Decoder model. For what is shown in the example to be possible, all that needs to be modified from the Vanilla Encoder-Decoder Transformer lies in the decoder side. Naturally, the most important change is the unmasked attention mechanism. The way the communication mechanism has been adapted to work for arbitrary ordering of insertions has been previously explained. However, what has not been mentioned is the significant impact the change of the ordering has on efficiency and decoder processing.

Trajectories and efficiency Once the left-to-right decoding is broken and we dynamically mask the attention mechanism, it is important to note that the model can no longer reuse previously calculated vector representations from the decoder at each time step. This happens because attention scores should be updated in order to account for newly inserted tags in the target sequence. Indeed, at each timestep, a special <UNK> tokens gets replaced with an actual tag that should be now exploited in the attention mechanism. A similar problem is also present in Stern et al. (2019), although for different reasons. In their model, it is the absolute positional encoding that needs to be recalculated, because of insertions changing the absolute positions of tags in the target sequence. Similarly to what the authors did in their work, what we are forced to do is forward the target sequence to the decoder at each time step, rather than conveniently doing a single forward pass that reuses all vectors representations for each trajectory step like it is done in Vaswani et al. (2017). This leads to a significant decrease in efficiency during training. In practice, this means that each sequence is forwarded to the decoder a number of times that is equal to its length. In the case of TagInsert, training from a single sequence starts from a sequence full of <UNK> tokens, and it ends with a sequence in which

only a single <UNK> token is present. Much like what was explained for the Insertion Transformer in Chapter 2, each of these individual subsequences makes up what we call a *trajectory*. In principle, multiple ways of determining the next subsequence of the trajectory at each time step are possible. In this implementation, the trajectories are randomly determined through a uniform extraction before-hand.

Reinforcing word-tag dependencies Another important change to the decoder is to add the source words embeddings produced by the DistilBERT model at each position, before going through the attention mechanism. This way, the model is given a more concrete understanding of the task of tagging, by directly linking each target tag to its source word. This was done in the hope of making aiding the sequence-to-sequence model to behave like a tagger.

A simple visualization of the architecture is shown in Figure 3.2. Note that the decoder is forwarded the tags sequence multiple times, represented by the loop on the decoder side, as after each trajectory step a new tag is inserted and the vectors calculated by the decoder may change because of the positional encoding. A more detailed graph is included in Appendix A for the sake of reproduction and clarity.

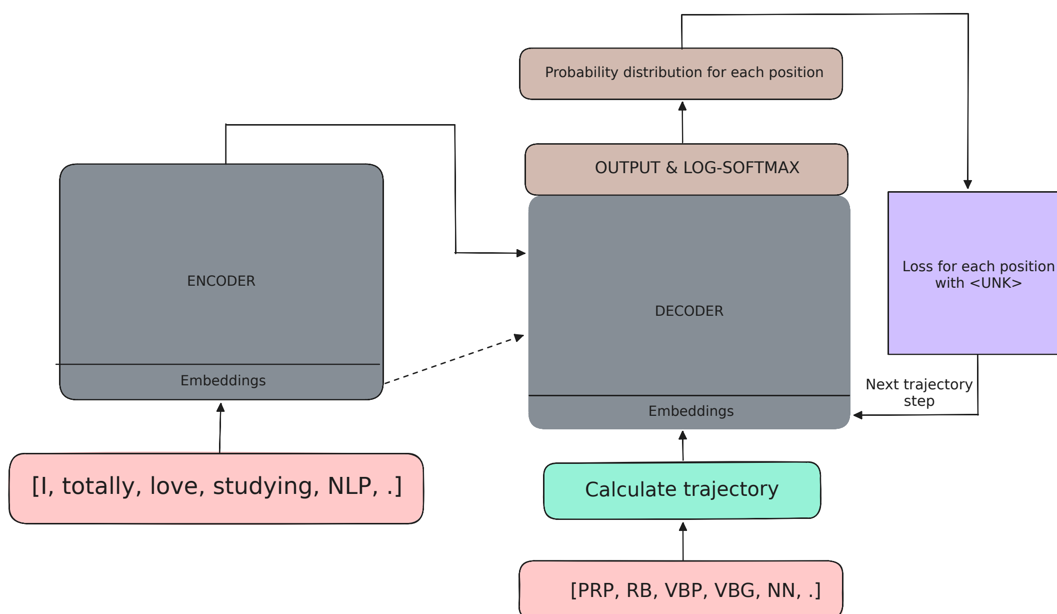

Figure 3.2: Architecture of the TagInsert model. The first trajectory is a full <UNK> sequence. After each pass to the decoder, a new tag is inserted in place of an <UNK> tokens, forming the new trajectory and continuing until the sequence is complete.

Training

Once the decoder outputs a representation for each position in the sequence, an output layer followed by a log-softmax layer maps *each position* to a probability distribution over all possible tags. In order for the model to learn, a loss function that penalizes the model for inserting incorrect tags is needed. The simplest way is to calculate the negative log-likelihood of inserting the correct tag at each position in which a <UNK> appears. Then, one can average over all the individual losses to obtain the full sequence loss. Formally, given a list of positions U corresponding to n <UNK> tokens in the original sequence $U = [u_1, ..., u_n]$, a target sequence y, and l probability distributions over tags for all positions in the sequence $P = [p(c_1), ..., p(c_l)]$ generated by the final output layer, the sequence loss is defined as

$$SequenceLoss = \frac{\sum_{i=1}^{n} -log(P_{u_i}(c = y_{u_i}))}{n}, \tag{3.1}$$

Where P_{u_i} is the probability distribution at position u_i corresponding to an <UNK> token and y_{u_i} is the correct tag at that same position from the target sequence. This way, the model learns to insert the correct tag at each position. At each trajectory step, this loss is calculated, leading to having multiple losses for the same position over different trajectory steps, depending on how the trajectory was sampled. Another approach could be to calculate the loss at specific positions rather than at all of them. For example, one could only calculate the loss corresponding to positions at the middle of the spans delimited by actual tags, mimicking Stern et al. (2019) with their Balanced Binary Tree ordering. However, other modifications to the architecture would also need to be implemented to allow for parallel insertions like in the original paper. Moreover, it is also possible to only calculate one loss at each trajectory step, corresponding to the position of the tag that will be inserted in the next one. This way, there is no repetition of losses over the same positions, and training mimics what happens during inference by learning to insert exactly one tag at each time step. Moreover, we applied *teacher forcing* (Williams and Zipser, 1989) during training using the same method of Vaswani et al. (2017). By inserting only targets coming from the gold sequence at each time step, we ensure that the model learns as much as possible from the data and does not lose focus by making mistakes at any point.

Inference

Once the model is trained, it can be used to generate a sequence of tags. Decoding starts with a sequence of <UNK> of the same length as the source sequence. Special start and end tokens are used as delimiters. Then, at each time step, one tag and its corresponding position is greedily chosen to be inserted in the sequence. Only positions corresponding to <UNK> special tokens are considered when greedily predicting the next tag. Decoding ends when t is equal to the source sequence length. It is important to remark that the position chosen at each time step is purely determined by the token the model wants to predict. The model does not generate a joint probability distribution over both positions and tags, but rather greedily chooses the tag with the maximum probability out of all positions. This naturally translates to the model choosing what it deems to be the safest next tag for maximum accuracy.

Differences with the the Insertion Transformer

Many of the differences from Stern et al. (2019) have been illustrated in this Chapter as design choices, but for the sake of clarity we will list all changes we made to the original Insertion Transformer in this Section. Most of the changes are simplifications that can be made thanks to the model being trained specifically for tagging. For example, there is no need to represent slots between words, and the sequence of targets to be produced has a pre-determined length. Instead, both training and inference start with a sequence full of *<UNK>* special tokens, used to represent positions in which a tag has yet to be inserted. Next, there is no explicit modelling of a probability distribution for both the tokens and the position, the model simply uses the token logits at each position to greedily choose the position simultaneously. This relieves the architecture of the burden of modelling another layer of information, with the downside of only being able to be used for generating sequences for which the length is known in advance. TagInsert currently does not implement parallel decoding. While it is possible to do it in TagInsert, we did not focus on this aspect of the original model for this research. Ultimately, TagInsert is a model that lies in between the Vanilla Encoder-Decoder Transformer and the Insertion Transformer, in which the architecture very much resembles the original Transformer, while the training and decoding procedure resembles the Insertion Transformer.

TagInsert L2R

Motivation In order to provide a fair comparison between the Vanilla Encoder-Decoder Transformer presented earlier in the chapter and TagInsert, a specific case of TagInsert is presented. Because TagInsert is forced to predict in a left-to-right ordering in this model, it is also a variant of the Vanilla Encoder-Decoder Transformer, in which word embeddings are added to the decoder while preserving its architecture. Essentially, the model differs from TagInsert in which it has a left-to-right ordering, and it differs from the Vanilla Encoder-Decoder Transformer in which it has the benefit of being given more explicit word-tags dependencies. By comparing it with TagInsert, it is possible to investigate the effects of breaking the left-to-right ordering.

3.2 Models for CCG constructive supertagging

For the task of CCG constructive supertagging, three main architectures can be considered. Experiments were also made on three variants of the original Prange model, resulting in a total of five models trained for the task.

1. Prange et al. (2020) reproduction
2. Constructive TagInsert
3. Prange Variants (3 variants)

First, the Prange et al. (2020) model was chosen as a simple yet effective example of a constructive supertagger. In their original implementation, Prange et al. (2020) designed the model so that the construction of each supertag would only depend on its associated word representation. Indeed, each supertag in the sequence is build in parallel. This implies that the order in which words are supertagged is irrelevant for this model. Second, the Constructive TagInsert model was designed to be a constructive supertagger that implements the arbitrary ordering of generation. Indeed, the TagInsert architecture proposed previously in this Chapter can only be used for opaque labelling. There is no mechanism that allows it to represent or utilize each of the atomic categories of a CCG supertag. In our solution to obtain a constructive version of TagInsert, we merge the TagInsert and Prange models. By doing so, not only we obtain a variant of TagInsert capable of constructive supertagging, but it is possible to inspect the effect of arbitrary ordering on the Prange architecture. The idea is to let TagInsert make an informed decision

on the next position to construct in based on the previously generated supertags, which also guide category construction through the Prange module. Finally, three variants of Prange are also proposed, in which all the supertags are built simultaneously like in the original model, but the presence of previously predicted atomic categories in the sequence is tentatively exploited. While this approach lacks the arbitrary ordering that the TagInsert model offers, it is an interesting adaptation of the Prange model that exploits a feature of supertags that was not present in the original work. Specifically, it is an attempt of capturing the intuition that each atomic category in a supertag has a strong dependency on the atomic categories present in the other supertags in the sequence.

3.2.1 Prange et al. (2020) reproduction

Motivation We re-implemented the *AddrMLP* variant of Prange et al. (2020). The model is a simple implementation of a constructive supertagger, being a finetuned BERT-like encoder at its core. Moreover, the original paper showed good performances, with results outperforming non-constructive models. Although the model is invariant to ordering, it was chosen in order to investigate whether adding ordering through TagInsert would help the model exploit the information held by previously predicted tags in the sequence.

Supertags representation When building a constructive supertagger, it is crucial to decide on a way to represent the supertags. Indeed, it is not sufficient to simply map each supertag to a label. To construct a category means to build each of its atomic components block by block. Not only that, but the relationship between each block, as expressed by the slashes, must also be represented. Prange et al. (2020) makes a straight-forward choice by using binary trees. In this view, each supertag is seen as a collection of terminal and non-terminal nodes. If the node is terminal (leaf), that implies an atomic category (S, NP, ...). If the node is non-terminal, that implies a slash (\or /). Figure 3.3 shows an example for the supertag *(S\NP)/NP*. In Prange's case, the categories are built in a top-down fashion, starting by the root and stopping when all nodes are terminal.

Architecture The original architecture of the model is that of a BERT-like encoder with a somewhat sophisticated output layer. An encoder block of choice is used to calculate a contextualized representation of each word in the sequence. In the paper, the BERT-like model they chose to produce the encodings is RoBERTa (Liu

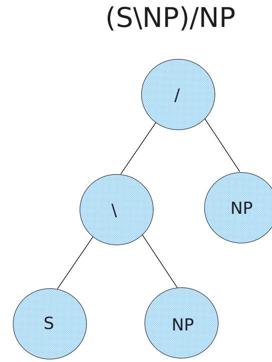

Figure 3.3: Representation of the supertag *(S\NP)/NP* as a binary tree.

et al., 2019). Once the sequence is encoded, the model proceeds to build the category in a top-down manner. First, the root is predicted using only the encoded word produced by the BERT-like model. The encoding goes through a fully connected 2-layer perceptron and a softmax layer, that maps it to the vocabulary of the atomic categories and slashes. The label for the node is greedily predicted and construction moves on the next depth. For depths higher than 0, the process is the same with the exception that an encoding for each node's position and its ancestor's slashes is calculated and added to the word's representation. As described in the paper, both the node's position and its ancestors are represented as a binary string. In the case of the position, the string corresponds to a top-down traversal of the tree, in which every left turn is assigned a 1 and each right turn a 0. In the case of the node's ancestors, it also correspond to a top-down traversal, but each value depends on the encountered slashes: 1 for forward slashes and -1 for backward slashes. The two binary strings are then concatenated and passed through a linear layer that projects them into the space of the BERT-like encoded word. Now, the two can be added together and go through the previous 2-layer perceptron and softmax layer, so that a label can be predicted for each node. Formally, each node at depth d of the tree corresponding to word w of a given sequence is given a hidden representation $H_{w,d}$ calculated as follows:

$$H_{w,d} = E_w + Linear(pos + slashes) \tag{3.2}$$

Where E_w is the encoded word from the BERT-like model and *pos* and *slashes* are binary strings calculated from the position of the node in the tree and the direction of its ancestors' slashes. When $d = 0$ and the root's representation is being

calculated, both *pos* and *slashes* are zero strings. Once the hidden representation is calculated, the node's label is greedily predicted from the probability distribution over the atomic categories and slashes $P(l)$ calculated as follows:

$$P(l) = Softmax(MLP(H_{w,d})) \tag{3.3}$$

From Equation 3.2 and Equation 3.3, it is apparent that each node's label is only dependant on the word's representation, its position in the tree and its ancestors. No information is extrapolated by the other trees in the sequence, albeit the contextualized words embeddings from the BERT-like model are context aware. Construction continues depth by depth, until all nodes are terminal. Because all trees are independent of each other, construction can be parallelized throughout the whole sequence, granting good efficiency. Figure 3.4 shows a diagram of the model's architecture and training.

Because there is inherently a strong dependency between terminal nodes (atomic categories) of the other supertags in the sequence, it seems that some information could be exploited there. This motivated us for the Prange variants that will be presented in later Sections. The original paper tried to more explicitly exploit the information of the other words in the sequence by using attention over the encoder's hidden state. However, the reported results do not differ very much.

Training Once all the trees in the sequence have been constructed, the model needs to be corrected through the loss function so that each individual node is correctly predicted. In the original paper, the authors make use of the Cross Entropy Loss applied to each node in the tree. An important point to be made is that, to correctly predict a supertag, every single node that it is comprised of must be correct. This means that we want the contribution of the loss function to be balanced across all nodes in the tree. In other words, whether the tree shows a large number of nodes or just a single one, we want the model to uniformly learn to be accurate in each one of the atomic decision that make up the tree. Hence, the loss of a given tree cannot be normalized over all the atomic decisions, otherwise shorter trees would be favored in the learning process. This is an especially relevant point in CCG grammar, as the simplest trees are also the most frequent ones. Indeed, the goal of constructing categories block by block is to be able to accurately generate

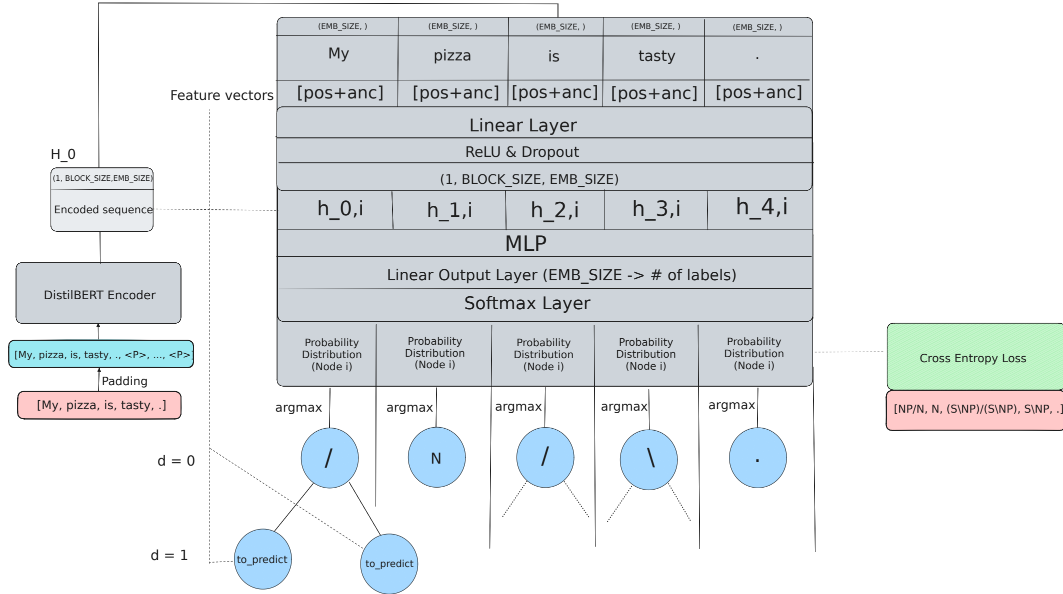

Figure 3.4: Model architecture and training of the Prange *AddrMLP* constructive CCG supertagger. The running example shows the model predicting nodes at depth 1.

uncommon categories. The already imbalanced learning process would be skewed even more in favor of frequent categories.

Another important design choice that the authors have made is that of using *teacher forcing* (Williams and Zipser, 1989). Because each node's prediction depends on its ancestors, it is important that, when building a tree, the model is not impeded by errors made along the way. For instance, if the tree *N/N* is to be built, but the root is incorrectly predicted as an atomic category rather than a slash, constructions immediately halts and the amount of learning that can be done for this tree is limited to one node. For this reason, each node is also assigned a teacher label that contains its gold target, in this case the forward slash. This helps the model to maximize learning from the training data. Formally, given a tree T comprised of n nodes, a sequence of n gold targets Y_n representing all the correct nodes from the data and n probability distributions P_n over the target vocabulary (labels and slashes) produced by the model, the tree loss is calculated as follows:

$$TreeLoss(T, Y_n, P_n) = \sum_{i=1}^{n} CrossEntropyLoss(Y_i, P_i) \tag{3.4}$$

The full sequence loss is then calculated as the sum of each tree loss. Similarly, the batch loss is calculated as a sum of all sequence losses, normalizing only by the number of words in the batch.

From the loss function, the gradient is allowed to flow through the model all the way to the encoder block. As each individual node's prediction is greatly dependent on the words representations, the model is essentially an instance of fine-tuning a BERT-like model. The main difference being in the output layer: instead of simply mapping each word to an opaque label, the model is taught to shape its representation to accurately build a binary tree in top down order. The information held by the node's position and ancestors makes the model aware of which node exactly it is trying to predict.

Inference During inference, the sequence of words is forwarded to the model. As there is no directionality in the approach, trees can be built in parallel depth by depth. Inference stops as soon as all trees reached terminal nodes (atomic categories) in all their branches.

Changes from the original paper Most of the aspects of the models are retained in this reproduction, with a few subtle differences. First, an embedding layer was added to enhance the representation of each node in a tree. In contrast, the original paper used a binary encoded feature vector to represent both the position of each node in the tree and its ancestors. In the model presented here, each combination of position and ancestor is represented by a trainable embedding, hopefully enriching the model's representation of each node. Second, the model is only exposed to sequences of length shorter or equal than 70. In earlier Chapters, the maximum length permitted was 50, but for the sake of reproduction we chose to minimize the amount of cut sentences from the data. Third, only the AddrMLP version of the model was implemented, as it was more consistently performing model in their work. However, we did not experiment with the way in which the authors tried to exploit the encoder's attention weights to have better node's representation and more consistent outputs. In their results, there is only a slight improvement when using such method and reproducing this result was not deemed of primary importance.

3.2.2 Constructive TagInsert

Motivation Investigating the effect of arbitrary ordering in the context of constructive supertagging is a challenging task. The most straightforward way to do it is to adapt the previously presented TagInsert architecture for the task. However, being a sequence-to-sequence model, it is not easy to directly translate from a non-constructive to a constructive framework. We decided to use the simplest approach. We exploited the fact that the Prange architecture only makes use of words representations to build its tags, and we went for what is essentially a modular approach to the architecture that merges Prange and TagInsert together. This way, we were able to have a simple implementation of a constructive supertagger that allows for arbitrary ordering, while also making use of a constructive framework that we proved could perform.

Architecture The model is effectively a combination of TagInsert and Prange. First, the TagInsert model handles both the words and the supertags in the encoder and decoder sides respectively. As shown in Figure 3.5, the decoder side produces positional vectors, one for each element of the sequence, exactly as it was done before. Immediately after producing the content-location logits, the model greedily decides which position to build a tree on based on the content logits, and passes the corresponding vector to the Prange mechanism. From here, supertag construction functions exactly the same as in our replication of Prange et al. (2020). The main draw of the approach is that the Prange module uses the positional vectors calculated on the decoder side through the unmasked attention mechanism by TagInsert. Conceptually, these vectors should hold relevant information on the other supertags in the sequence, and it is our intention to investigate how and if the constructive model can make use of such information. Figure 3.5 shows a simplified version of the architecture, highlighting how the two modules work together.

TagInsert and Prange As intuitive as this approach is, it holds a significant inconsistency. Our ultimate goal is to make use of arbitrary ordering mechanism offered by TagInsert so that relevant information of the other trees in the sequence could be used for a more informed supertag construction. However, there is a discordance in how this information is represented by the two modules. TagInsert assumes that each supertag is an opaque label, and as such it has no concept of the individual nodes that make up each category. Prange, on the other hand, wants to use

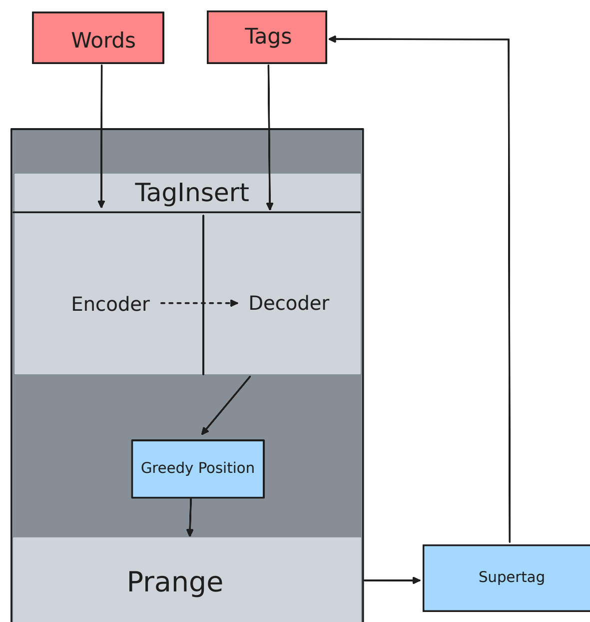

Figure 3.5: Model architecture of the Constructive TagInsert supertagger.

what TagInsert produces to build a collection of nodes that its collaborator ignores. Essentially, the two modules speak a different language. While this does not necessarily have to be an issue, it is a conceptual inconsistency that may hinder the model. As the simple merging of the two architectures is the most straightforward way to convert the TagInsert model into a constructive supertagger, we decided to use this approach. Moreover, while the conceptual difference between the two could indeed be a problem, it is not guaranteed. Thus, we experiment with this limitation in mind, and leave the exploration of other approaches that overcome this drawback for future works.

Training Training works the same way it was presented in previous Sections. The loss function is handled like in 3.4. The only difference is in how trajectories are calculated. In the TagInsert model described earlier in the Chapter, trajectories for each timestep were randomly sampled in advance for each sequence. Then, a loss would be calculated in each position corresponding to a special <UNK> token before moving onto the next trajectory step. In this case, in order to improve efficiency, the next trajectory step is determined at each timestep by the TagInsert model, corresponding to the greedy choice of the position to build the next tree in. Then, only the loss corresponding to the single built tree is calculated. Essentially,

TagInsert chooses the position on which the next supertag is to be built, and once it is constructed the corresponding loss is calculated. After that, the gold target corresponding to that position is added to the target sequence and everything is forwarded to the decoder again. This process continues until the sequence is complete, just like in TagInsert.

Inference Inference also works very similarly to the standalone TagInsert model. There is however a small detail that stems from the discordance between the two modules that was mentioned above. In contrast with the non-constructive TagInsert model, the Prange module makes it possible for OOV (out of vocabulary) supertags to be predicted. This is, as mentioned, an intended and much desired behavior from a constructive supertagger, as one of the main appeal of the approach is to be able to tag the long-tail of numerous but infrequent supertags of the grammar. The TagInsert side of the model however, cannot operate on OOV supertags, as by viewing each tag as an opaque label it is simply not aware of the existence of them. Thus, when an OOV tag is predicted for a sequence at any timestep, it is converted into a special <U> token. While Prange et al. (2020) results show that their approach does not make the model eager to predict OOV tags anyway, it is one of the limitations of this architecture. Again, further work on this issue can help in finding a better way of converting TagInsert to a constructive supertagger.

3.2.3 Prange variants

Motivation Having presented the Prange architecture and the way arbitrary ordering can be used to exploit information about other trees in the sequence, we turn to exploring variants of what was originally proposed in Prange et al. (2020). While adding ordering to the model gives us a way to access the other trees, there are alternatives to make use of such information. The original model allows for parallel, depth by depth, tree construction. While the main appeal for this is efficient decoding, it is also possible to exploit the access that this gives to atomic categories predicted in previous depths of the other trees. The models will still be invariant to ordering, but the information held by atomic categories on surrounding trees will be used for a more informed construction. Intuitively, the models will try to make use of the idea that most atomic categories that make up a supertag are directly referencing terminal nodes in other trees.

For example, let us consider the sentence from Figure 3.4, *My pizza is tasty*, with

supertags *NP/N, N, (S\NP)/(S\NP), S\NP*. When predicting the tag for the word *My*, the two atomic categories *NP* and *N* are directly referenced in the other trees. Indeed, the *N* refers to the noun *Pizza*, and the *NP* creates a noun phrase that will be later consumed by the verb *is* to form a sentence. Because the *N* for *pizza* has already been predicted at depth 0, it is possible to make use of this information when predicting the two labels at depth 1. From the example it is noticeable that this is not always possible, as sometimes referenced labels are deeper down the tree. For example, in the case of the word *tasty*, both the *S* and the *NP* are referenced in deeper nodes of the tree for the verb *is*. Thus, in these variants the model will not be constrained by the existence of the other nodes by masking the probability distribution over atomic categories, but rather the information will be given to it and it is up to the model to exploit it or not. This approach embraces the idea that supertagging should be viewed as *almost parsing*. Indeed, it is during parsing that the referencing of each atomic category is used to build the derivation trees.

This Section presents three different approaches on this idea, each of them handling the information that comes from the atomic categories present in previous depths in a different way. To reasonably limit the amount of nodes the model can access at any depths, only trees in a context window of 3 to the current one are considered. That means that for any given tree, a maximum of 6 neighboring trees are taken into consideration.

V1: Concatenating the leaves The first approach is the most straightforward one. Each leaf node already predicted in other trees represent an atomic category (not a slash), which can potentially be used by the model to make a more informed prediction of the current label. Thus, all the leaves that can be accessed in the context window are concatenated and passed through an Embedding layer and a Linear layer that maps them back to the space of the encoded words. The resulting vector is then simply added to the encoded word and the position and ancestor's slashes information. Effectively, this turns Equation 3.2 to calculate a node's hidden representation into

$$H_{w,d} = E_w + Linear(pos + slashes) + Linear(Embedding(Concat(Leaves))). \quad (3.5)$$

where the term *Leaves* is used to reference previously predicted leaves in the context window, in depths *(0, ..., d-1)*. When the root is predicted and $d = 0$, no leaves are extracted and the equation reverts back to Equation 3.2.

V2: Weighting the leaves In this approach, all the L leaves that are accessible are passed through an embedding layer and then mapped back with a Linear layer to a single tensor of length L. Then, the tensor is passed through a Softmax layer that computes a weight for each leaf, to then be used for a weighted sum with their embeddings, effectively calculating a weighted representation of the surrounding leaves. In this idea, the model would learn to prioritize certain leaves when exploiting the information that it is given. This turns Equation 3.2 to calculate a node's hidden representation into

$$H_{w,d} = E_w + Linear(pos + slashes) + W(Linear(Embedding(Leaves))). \quad (3.6)$$

where W indicates the described process of calculating a weight for each leaf and performing a weighted sum with each encoded leaf.

V3: Informed weighting This final variant uses the same approach as the previous one, but it is slightly constrained in the amount of leaves the model has access to. Up until now, all available leaves have been used by the model to encode them as information. However, following the logic of parsing, there is a distinction to be made when we are trying to predict the label for a right child or a left child. In CCG, what comes to the right of a slash (i.e. right child in a binary tree representation) is known as the *argument*, while what comes to the left (left child) is known as the *result*. The argument indicates an atomic category that is to be consumed with another atomic category during parsing, while the result indicates what will be left after said consumption. In practice, this means that when predicting an argument, only *results* and roots from the other trees contain useful information, as they may be consumed as the argument at any point during parsing. Conversely, when predicting a result, only *arguments* and roots should be considered. While this is an oversimplification of how parsing actually works, as in practice there are also more complex compositional rules to build the parse tree that escape this logic, this is a fair rule of thumb that we may be able to exploit. Thus, in this approach we

discriminate between argument and result when predicting a node, and the leaves that are given to the model are dependent on which type of leaf we are predicting.

4. Experiment 1: Part of Speech Tagging

In this Chapter we consider the task of Part-Of-Speech tagging, one of the most fundamental tasks in the field of Natural Language Processing. A large variety of models and architectures have been trained for the task, and the state of the art offers very good results, reaching up to 98% accuracy (Bohnet et al., 2018). Although Encoder-Decoder models, like the original Transformer, can also be used for the task, the best performing models commonly use BERT-like encoder only architectures. Indeed, the Encoder-Decoder paradigm was intended for broader sequence-to-sequence tasks. On the other hand, using only the encoder side translates into a model that effectively behaves like a tagger, mapping each word to a specific tag. Because sequence-to-sequence architectures are not commonly used for tagging, this research also serves as an exploration on how such models may perform in this context. We chose this task as a well documented setting to experiment on a model that breaks the left-to-right ordering and allows the model to come up with its own, preferred ordering.

The models we chose for the experiment were previously presented in Chapter 3, but are also shortly summarized here as a reminder. We first trained a baseline encoder-only model that accurately represents what the state of the art on the task is. This way, it would be possible to determine whether our model could match or even outperform such models. Then, we experiment on the task using the original Vanilla Encoder-Decoder Transformer architecture, as it was presented in Vaswani et al. (2017). The experiment serves to show whether an architecture thought for sequence-to-sequence tasks can perform on tagging. Finally, we train the proposed TagInsert model for the task. Stemming from the Transformer architecture, the model chooses its own production order. A variant of the model which is forced to generate tags in the left-to-right ordering is also trained, in order to see the impact of the arbitrary ordering.

4.1 Data and Pre-processing

In order to train the models for Part-Of-Speech tagging, the standard Penn Tree-Bank was used (Marcus et al., 1993). All the training was done on Sections 01-22, while validation was done on Section 00. The results reported for all of the models are for Section 00 and Section 23. The best performing epoch from evaluating on Section 00 is used to test on Section 23. In order to keep the memory needs contained, only sentences of length up until 50 were kept for all sections. This means that 777 sentences were cut from the training data (about 1.8% of the total number of training sentences) and 36 sentences were cut from both Section 00 and Section 23 (about 1.8% and 1.5% of the sentences in the respective sections). While this may make it a bit difficult to directly compare results on other models outside of this research, we hypothesize that the general performances of the models should not be affected. After the process, the training data is comprised of 42748 sentences. The validation data is comprised of 1885 sentences in Section 00 and of 2416 sentences in Section 23. The total amount of Part-Of-Speech tags contained in the data is 45, with no out of vocabulary categories. The average sequence length is 23 for all of the sections.

Words representation and tokenization Before forwarding the data to the models, words that appear in each sentence are processed through a DistilBERT-base cased model (Sanh et al., 2020). This is done in order to extract contextualized words embeddings that the models can exploit for better performances. Albeit conceptually straightforward, in order to complete this step a choice needs to be made in regards to tokenization. That is, BERT-like models split each word into sub-words and produce a separate vector for each. In the context of Part-Of-Speech tagging, each word is assigned exactly one category, and ideally the whole representation of the word from BERT is a desirable information for the model to process. A few approaches come to mind to solve the issue, such as keeping only prefixes or suffixes as representative of the whole word. Moreover, it is also possible to average over all the different sub-words produced by BERT in the hopes of aggregating all the relevant information the model produces. Ultimately, as we were not able to reference any standard way of handling the issue, experimentation was performed for all three of the listed methods. As no meaningful difference was found among the three, the prefix was arbitrarily chosen as representative of each word. Finally, in order to extract the contextualized embedding from the language model, the last

four layers of the model were summed together, as to retain as much information as possible. The resulting embeddings are then forwarded to the models and are not updated further.

For last, both the words and the tags in the sentences are padded up to a fixed block size before being forwarded to the model. This is made to account for the maximum sequence length of 50 in the data, plus however many special start and end tokens are needed for the models. These steps apply for all the models presented in the next section, except for the fine-tuned DistilBERT model. As this was a fundamentally different architecture when compared to the others, no preprocessing step was necessary, except for padding.

Hyperparameters All of the models, except for the fine-tuned DistilBERT, have the same hyperparameters. Namely, all models used for this experiment were built with 6 encoder-decoder stacks and 8 attention heads. The optimizer used was Adam, with a starting learning rate of 0.5, β_1 of 0.9, β_2 of 0.98 and ϵ of $1e^{-9}$. The learning rate was adapted throughout training by using the LambdaLR scheduler with the same parameters of the original Transformer model. Finally, the dropout probability for all layers was set at 0.1. Conversely, the fine-tuned DistilBERT model had a learning rate of $2e^{-5}$ and a weight decay of 0.01.

4.2 Results and discussion

Table 4.1 summarizes the results of Part-Of-Speech tagging with the models presented in Chapter 3. The results for Section 23 are obtained by evaluating the best performing model out of all the trained epochs on Section 00.

	Epochs	Batch size	Accuracy (Sec. 00)	Accuracy (Sec. 23)
FineTuned DistilBERT	3	24	97.69%	97.81%
Vanilla Encoder-Decoder Transformer	3	64	93.57%	93.83%
TagInsert	3	64	**97.24%**	**97.41%**
TagInsert L2R	3	64	97.16%	97.31%

Table 4.1: Results for the Part-Of-Speech taggers, evaluated on both Section 00 and Section 23 of the Penn TreeBank. The baseline model is highlighted in bold, and the best results for each section out of the other three models are also highlighted.

4.2.1 Finetuned DistilBERT

The model converges at about at 97.69% for Section 00 and at about 97.81% accuracy for Section 23. The result is comparable with the state of the art for Part-Of-

Speech tagging. Thus, it can serve as a good baseline to investigate whether the implemented models can compare in terms of performances.

4.2.2 Vanilla Encoder-Decoder Transformer

The model converges at about 93.57% accuracy for Section 00 and 93.83% accuracy for Section 23. The result falls short when compared to the state of the art on Part-Of-Speech tagging. This was to be expected, as the original purpose of the architecture was not tagging, and the dependency between words and tags is only modelled by the cross-attention mechanism and the positional encoding. In the best performing models for the task, the objective is to assign categories to the words, rather than generating an open-ended sequence of tags. The issue that this brings can be observed when analyzing the errors the model makes. It is frequent that, once a difficult tag is reached during the generation of the sequence, the model offsets the word-tags predictions. Once that happens, all the subsequent predictions are shifted to the left, resulting in a domino effect. Following is an example from Section 00 that shows this behavior.

Word	Japan	's	Commission	has	said	it	is	considering	investigating	the	bids	for	possible	antitrust-law	violations	.
Gold Tag	NNP	POS	NNP	VBZ	VBD	PRP	VBZ	VBG	VBG	DT	NNS	IN	JJ	JJ	NNS	.
Predicted Tag	NNP	POS	NNP	VBZ	VBN	PRP	VBZ	VBG	DT	NNS	IN	JJ	JJ	JJ	NNS	.

In this example, there are two different types of mistake the model makes. First, the model misclassifies the word *said*. This is likely to be a labelling error in the data, as the verb is indeed in its past participle form rather than simple past, like suggested in the gold label. Moreover, it is clear that this type of mistake will not cause the domino effect previously descripted, as the model did not attempt to skip any tag. More interestingly, once the model attempts to generate the tag for the word *investigating*, it predicts a determiner (DT) for it. This was clearly meant for the next word *the*, but the model has no concept of word-tag assignment, and thus moves on with generation with a misalignment that may or may not get fixed. It is likely that either the word *investigating* was too difficult to predict using only the prefix for its word embedding, or that the model got confused by the two verbs in gerund forms in sequence. Regardless, while this issue is caused by the model architecture and its original purpose, it is also a general weakness of auto-regressive models, in which the full dependency on the previously generated tokens may result in a propagation of an error to subsequent generations.

4.2.3 TagInsert

The model converges at about 97.24% accuracy for Section 00 and 97.41% accuracy for Section 23. While TagInsert shows that breaking the left-to-right approach and reinforcing the word-tag dependencies on the decoder side yields good results, it is likely that the fact that tagging is still treated as a sequence-to-sequence task may still impede it to reach the performances of the fine-tuned model. Still, the result is quite good and is comparable to the state of the art on the task. In order to quantify how much of this improvement is due to the arbitrary ordering, the next section discusses the L2R TagInsert model. The model also reinforces the word-tag dependencies on the decoder side, making it a more direct comparison with the DistilBERT model.

By analysing the errors the models makes during inference, it is clear that the behavior shown by the Encoder-Decoder model is no longer there, and no skipping of tags occurs. This is likely due to the word embeddings added at each position of the decoder, strengthening the relation of tags and words pairs. As for the ordering that the model chooses, there is a clear pattern based on confidence and frequency of the tags. Namely, the model often chooses to start decoding with very easy tags, in which the confidence is very high. For example, the punctuation tags like the periods, commas and columns are almost always the starting point for generation. Other tags that are frequently chosen to start with are pronouns (PRP), determiners (DT) and special symbols ($, ", ...). On the other hand, when the model encounters a hard tag for which its confidence in the prediction is low, it usually keeps it for last. Indeed, when the model makes a mistake, it often occurs that the last inserted tag is the one that was mistaken. Tags that are commonly mistaken include adjectives (JJ), different types of verbs (VB, VBN, VBZ, VBG, VBD) and proper nouns (NNP) which can be ambiguous. Figure 4.1 shows a more detailed analysis of the chosen ordering by the model. This behavior of the model that prioritizes easy tags and leaves problematic ones for last leads to an interesting effect. One of the main drawback of auto-regressive models is the fact that errors are always propagated in future predictions, as generation is always conditioned on previous time steps. As an auto-regressive model, TagInsert is not immune to this problem, but making the active choice of leaving problematic tags for last can potentially help alleviate the issue. By contrast, a left-to-right fixed ordering model cannot make such a choice, and is forced to make decisions as they come. Figure 4.2 shows the behavior of the model on aggregated Part-Of-Speech tags. As different types of verbs, nouns,

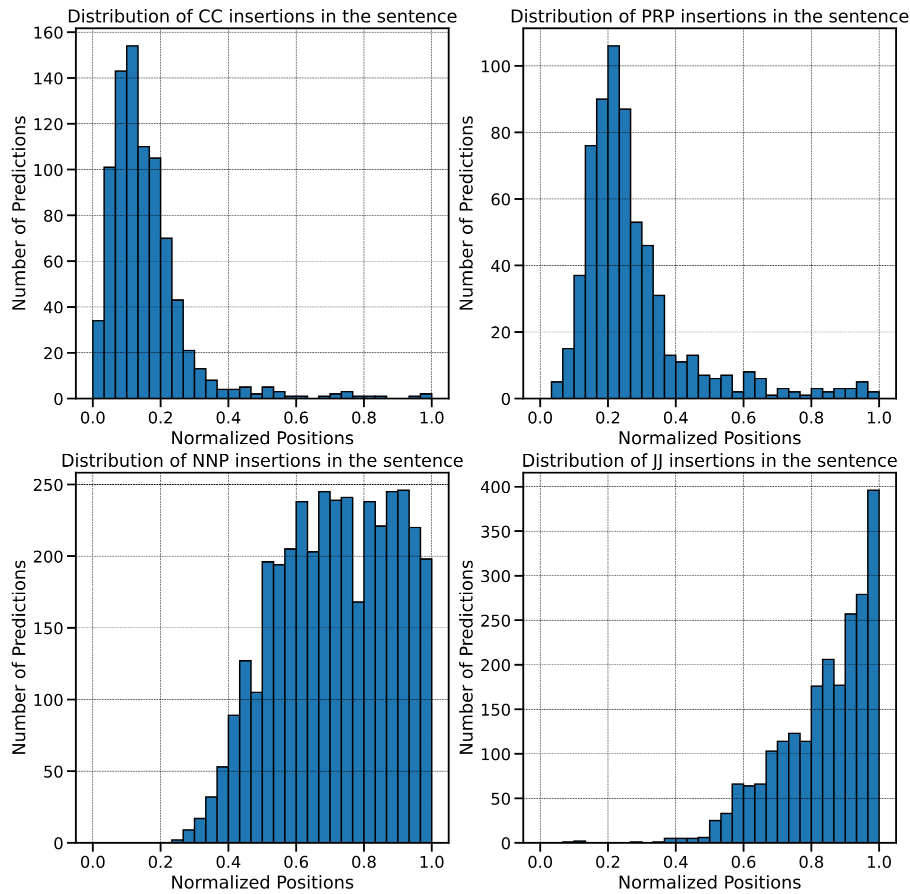

Figure 4.1: Ordering normalized on sequence length for selected tags. It is clear that easy tags like *PRP* or *CC* are prioritized over difficult ones like *NNP* and *JJ*.

adjectives and punctuation have a number of different Part-Of-Speech tags, the graph further shows how after aggregating each category, the model gives priority to non-ambiguous ones like symbols and punctuations. Conversely, verbs, nouns and adjectives are kept for last.

4.2.4 TagInsert L2R

The model converges at about 97.16% for Section 00 and 97.31% accuracy for Section 23. The difference with the TagInsert model is not very large, and it seems like most of the issues seen in the Vanilla Encoder-Decoder Transformer were again solved by adding the words embedding on the decoder side.

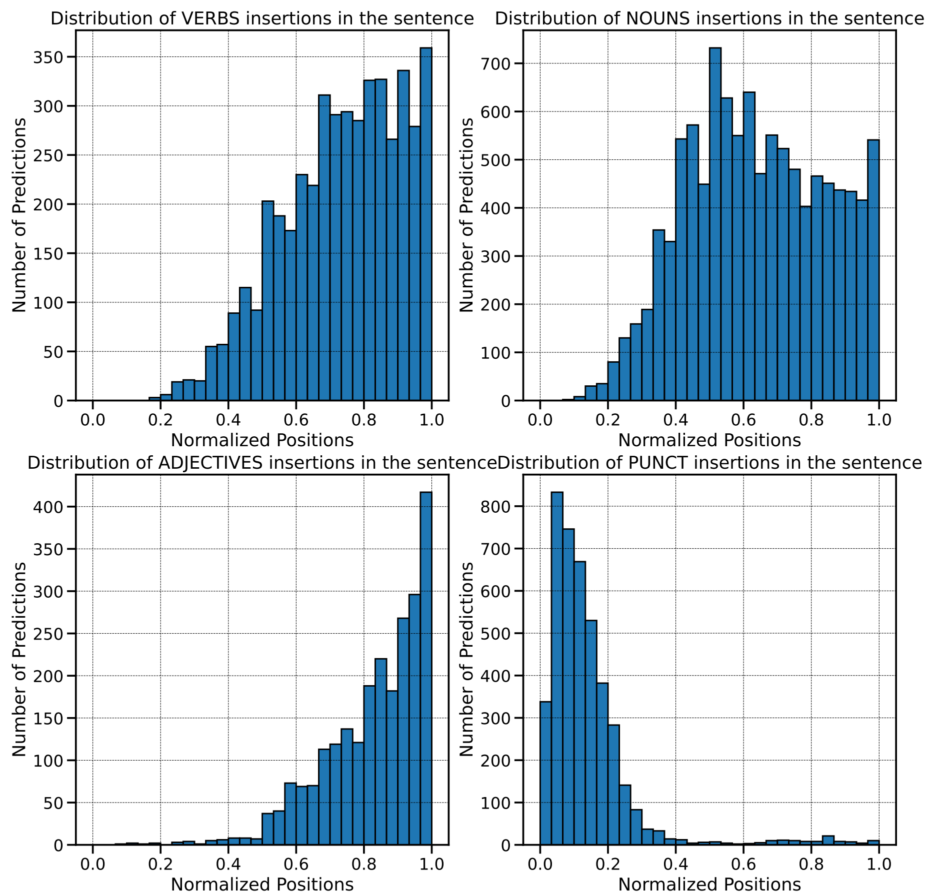

Figure 4.2: Ordering normalized on sequence length for selected aggregated categories of tags.

4.2.5 Comparisons

In this Section, the models will be compared based on their performances on Section 23. A two proportion Z-test is used to test whether the difference in accuracies between models is significant.

As mentioned earlier, the Vanilla Encoder-Decoder Transformer falls short when compared to the other models. This is likely due to its nature as a sequence-to-sequence model that, while usable for tagging, needs some more explicit constraints on the direct dependencies between words and their tags. Although TagInsert retains a similar architecture, by adding the source words embeddings on the decoder side, this issue is successfully alleviated.

The Fine-tuned DistilBERT model is the best performing model. The accuracy is much higher than the Vanilla Encoder-Decoder Transformer's. When compared to the TagInsert model, there is a statistically significant difference between the accuracies (p-value: $1.49e^{-5}$). Naturally, when compared to the TagInsert L2R lower accuracy, there is also a statistically significant difference between the accuracies (p-value: $8.38e^{-8}$). All in all, it is not too surprising that the finetuned model performs so well, as in its way of explicitly mapping each word to a single tag, it effectively behaves like a tagger.

The TagInsert model greatly improves over the Encoder-Decoder model. As mentioned, most of the improvements seems to come from the word embeddings added on the decoder side. Indeed, the difference in accuracy between the model and its left-to-right variant is not significant (p-value: 0.302). This suggests that breaking the left-to-right directionality in favour of an arbitrary ordering is not sufficient for the model to improve in a meaningful way. However, the fact that there is no significant difference does not mean that there is no point in breaking the left-to-right ordering. Moreover, it is possible to show that it provides some advantages. In fact, one could argue that the fact that both models perform should be in favor of the arbitrary ordering model, as it is a more general approach. While Part-Of-Speech tagging may not majorly benefit from it, other tasks, whose nature may suffer from being constrained to a left-to-right ordering, might widen the gap.

Regardless, below we present an example in which allowing for arbitrary ordering benefits the model in the context of Part-Of-Speech tagging. In the Example, the TagInsert model recognizes the ambiguity on the word *trade*, which could be both a noun and a verb, and leaves it for last. By doing so, it uses as much context as possible to do an informed prediction. By being able to exploit the notion that between a TO and a preposition (IN) the probability of encountering a verb is high, the model can successfully tag the word *trade*. The left-to-right model cannot make use of such information and makes a mistake.

t	Words (full sequence is always visible)	POS Tags TagInsert
1	[Cray]	[NNP]
2	[Cray Computer]	[NNP NNP]
3	[Cray Computer has]	[NNP NNP VBZ]
4	[Cray Computer has applied]	[NNP NNP VBZ VBN]
5	[Cray Computer has applied to]	[NNP NNP VBZ VBN TO]
6	[Cray Computer has applied to trade]	[NNP NNP VBZ VBN TO NN]
7	[Cray Computer has applied to trade on]	[NNP NNP VBZ VBN TO NN IN]
8	[Cray Computer has applied to trade on Nasdaq]	[NNP NNP VBZ VBN TO NN IN NNP]
9	[Cray Computer has applied to trade on Nasdaq .]	[NNP NNP VBZ VBN TO NN IN NNP .]

t	Words (full sequence is always visible)	POS Tags TagInsert
1	[to]	[TO]
2	[to .]	[TO .]
3	[has to .]	[VBZ TO .]
4	[has to on .]	[VBZ TO IN .]
5	[Computer has to on .]	[NNP VBZ TO IN .]
6	[Computer has applied to on .]	[NNP VBZ VBN TO IN .]
7	[Cray Computer has applied to on .]	[NNP NNP VBZ VBN TO IN .]
8	[Cray Computer has applied to on Nasdaq .]	[NNP NNP VBZ VBN TO IN NNP .]
9	[Cray Computer has applied to trade on Nasdaq .]	[NNP NNP VBZ VBN TO VB IN NNP .]

Example of benefit of breaking the left-to-right ordering. The left-to-right version of TagInsert misclassifies the word *trade*, while the arbitrary ordering model recognizes the ambiguity that comes with the word by leaving it for last. This makes it possible to have maximum information for more informed predictions in difficult cases.

5. Experiment 2: CCG Tagging

By training TagInsert for the task of Part-Of-Speech tagging, we inspected the effect that arbitrary ordering of decoding has on the task. While the difference in overall accuracy between the model and its left-to-right counterpart was not statistically significant, we were able to provide concrete examples that showed how the model could benefit from dynamically choosing its ordering. However, we mentioned in earlier Chapters how the arbitrary ordering could be especially useful in tasks in which a preferred order of generation is more explicit. In Part-Of-Speech tagging, no such mechanism exists, and thus it is not surprising that TagInsert was not able to meaningfully improve on the task. Moreover, especially when considering the English language, the task is quite saturated, and it is difficult to show any kind of statistical improvement over the state of the art.

In this Chapter however, we focus on a tagging task in which said mechanism is present, and there is an explicit reason as for why the left-to-right generation may be problematic. The same model trained for Part-Of-Speech tagging in Chapter 4 are now used for Combinatory Categorical Grammar (CCG) tagging. In this task, the underlying (syntactic) analysis and the supertags it is comprised of imply that a specific ordering of generation should be preferred. Indeed, each category is so informative that it is normally called a *supertag*. As a reminder, a common supertag is *(S\NP)/NP*, and it identifies a transitive verb. The supertag is made of a number of atomic categories, each one playing a specific role in representing the function a word has in relation to the other words in the sentence. Specifically, this supertag indicates that in the presence of a Noun Phrase (NP) to the right and a Noun Phrase to the left, the word makes up a sentence (S). It is clear how this explicitly models the behavior of transitive verbs, but most importantly, it is clear how the NPs should be tagged first, as they are directly referenced by the transitive verb's supertag.

For this reason, we hypothesize that repeating the experiment of Chapter 4 for the task of CCG tagging would show a wider gap in the performances between the left-to-right models and TagInsert.

5.1 Data and Pre-processing

In order to train the models for CCG tagging, the CCGBank (Hockenmaier and Steedman, 2007) and the Rebank (Honnibal et al., 2010) datasets were used. The former is a direct translation of the Penn TreeBank for Combinatory Categorical Grammar, while the latter is a reworked version of the CCGBank, and is, for all intents and purposes, a more complete version of the dataset. Specifically, the Rebank contains a total of 1675 unique supertags, as opposed as the 1323 contained in CCGBank. Most of these new categories belong to the *long tail*, which is the large population of rare supertags in the data. When it comes to CCG, tagging the long tail has been the most challenging aspect of the task, as traditional machine learning model have difficulty generalizing over infrequent data. In general, the supertags are more rigorous in the Rebank, and a lot of work has been made on the NP argument structure. The Rebank dataset also contains more atomic categories that make up the supertags. Specifically, Rebank shows 37 atomic categories, while CCGBank only 34.

All the training was done on Sections 02-21, while validation was done on Section 00. The results reported for all the models are for Section 00 and Section 23. Similarly to the previous Chapter, the best performing epoch from validation on Section 00 is used to test on Section 23. Again, only sentences with length up until 50 were kept for all sections. Thus, for both datasets, 735 sentences were cut from the training data (about 1.9% of the total sentences), 33 sentences were cut from Section 00 (about 1.7% of the total sentences) and 28 sentences were cut from Section 23 (about 1.2% of the total sentences). After the process, the training data is comprised of 38869 sentences, Section 00 is comprised of 1880 sentences and Section 23 is comprised of 2379 sentences. The average sequence length is 23 for all of the sections. The same pre-processing steps from Chapter 4 are performed. For the sake of comparison, the same hyperparameters from the previous Chapter were used for all the models.

5.2 Results and discussion

The results for the CCGBank and the Rebank datasets are discussed separately.

5.2.1 CCGBank

The results for Section 00 and Section 23 of the CCGBank are shown in Table 5.1. As Section 23 functions as our test set, the analysis will mostly be focused on it.

CCGBank	Epochs	Batch size	Accuracy (Sec. 00)	Accuracy (Sec. 23)
FineTuned DistilBERT	3	24	95.81%	95.33%
Vanilla Encoder-Decoder Transformer	3	64	91.68%	90.81%
TagInsert	4	64	**95.15%**	**94.57%**
TagInsert L2R	5	64	94.88%	94.44%

Table 5.1: Results for the CCG taggers, evaluated on both Section 00 and Section 23 of the CCGBank. The baseline model is highlighted in bold, and the best results from the other three models for each section are also highlighted.

Finetuned DistilBERT

The model converges at 95.81% accuracy for Section 00 and at about 95.33% accuracy for Section 23. The results falls a bit short from what is shown in the literature, which is about 96.22% for Section 23 (Prange et al., 2020). The difference may be explained by different training settings and BERT-like model. As these were all consistent in our research and because we use this model as a baseline for the architecture we implemented, we deem the slight difference with the literature acceptable.

Vanilla Encoder-Decoder Transformer

As the Vanilla Transformer model did not perform well in the previous Chapter, we expected that for the more difficult task of CCG tagging it would also underperform. However, we report the results for the sake of completeness. The model converges at about 91.68% accuracy for Section 00 and at about 90.81% accuracy for Section 23. Indeed, much like in Part-Of-Speech tagging, the model is not performing. The reason is, as expected, the same from the previous task: the architecture is not explicitly made for tagging. We still observe the phenomenon of tag skipping that was described earlier. All in all, it is evident that the original Transformer architecture needs modifications when used for tagging, as the word-tag dependencies are too loosely modelled this way.

TagInsert

The model converges at about 95.15% accuracy for Section 00 and at about 94.57% accuracy for Section 23. Again showing a big improvement over the Vanilla Encoder-

Decoder Transformer, the architecture proves it can perform as a tagger.

The ordering that the model chooses is, again, dictated by its confidence of predicting the correct category. In the context of training, confidence is a product of frequency and ambiguity of the supertags in the corpus. When compared to Part-Of-Speech tagging, the big vocabulary of Combinatory Categorical Grammar makes it more difficult to identify patterns, but when considering clearly ambiguous and unambiguous categories, we can still see a sensible behavior.

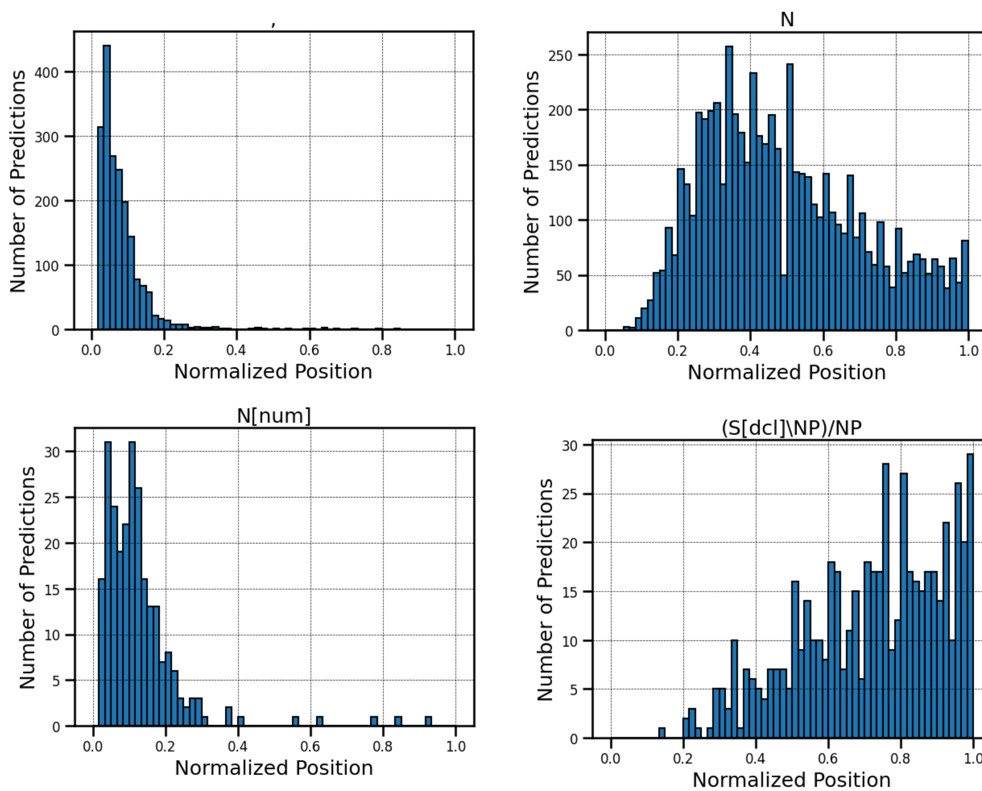

Figure 5.1: Ordering normalized on sequence length for selected CCG supertags. Unambiguous categories like , and *N[num]* are consistently inserted at the start of decoding, while more ambiguous ones like *N* and *(S\NP)/NP* are inserted towards the second half of decoding.

Figure 5.1 shows said pattern, in which very unambiguous categories like commas and *N[num]* (representing digits and numbers) are consistently inserted at the start of decoding. Conversely, supertags like *N* (nouns) and *(S\NP)/NP* (transitive verbs) which are generally ambiguous and more difficult to accurately classify, are inserted towards the second half of decoding.

TagInsert L2R

The model converges at 94.88% accuracy for Section 00 and at about 94.44% accuracy for Section 23. Like in the previous Chapter, the benefit of adding word's embedding to the decoder side is evident.

Comparisons

In this Section, the models will be compared based on their performances on Section 23. A two proportion Z-test is used to test whether the difference in accuracies between models is significant.

Most of the same patterns from Chapter 4 are present on the task of CCG tagging. As the task is more challenging to learn due to the big target vocabulary and the sparsity of the long tail, accuracies are lower all-around for all the models. The most glaring similarity is how the Vanilla Encoder-Decoder Transformer cannot compete with all the other architectures, because of its original purpose of a sequence-to-sequence model. Adding the word's embedding in the TagInsert L2R model mostly fixes the issue once again.

The fine-tuned DistilBERT model is the best performing model once again. When compared to the TagInsert model, there is a statistically significant difference between the accuracies (p-value: $1.25e^{-8}$). As the TagInsert L2R model performs slightly worse, the difference is also non-significant in that case. The superiority of the fine-tuned model as a tagger was already documented in Chapter 4, and this keep being the case in the context of CCG tagging.

The difference between accuracies of TagInsert and TagInsert L2R is, once again, not significant (p-value: 0.3495). This is slightly disappointing, as we hypothesized that in CCG tagging the arbitrary ordering would benefit the model more when compare to Part-Of-Speech tagging. In a way, this is also not completely unexpected, as the complex supertags are treated as opaque labels that ignore the information contained within the categories themselves. In this information is contained the ordering logic that we were hoping to exploit, and as such, it may have been optimistic to expect a stronger difference. We now move to the Rebank results, to see whether the same apply in the more polished dataset.

5.2.2 Rebank

The results for Section 00 and Section 23 of the CCGBank are shown in Table 5.2.

Rebank	Epochs	Batch size	Accuracy (Sec. 00)	Accuracy (Sec. 23)
FineTuned DistilBERT	5	24	94.44%	93.96%
Vanilla Encoder-Decoder Transformer	3	64	89.49%	88.35%
TagInsert	7	64	**93.84**%	**93.47%**
TagInsert L2R	7	64	93.54%	92.90%

Table 5.2: Results for the CCG taggers, evaluated on both Section 00 and Section 23 of the Rebank. The baseline model is highlighted in bold, and the best results from the three implemented models for each section are also highlighted.

The results are all around lower when compared to the CCGBank, remarking how the dataset has more categories, mostly belonging to the long tail. As the patterns of the individual models do not change from the CCGBank, we only focus on the comparisons between the models.

Comparisons

The FineTuned DistilBERT model and the TagInsert model show once again a difference in performances. Namely, the difference in accuracy is statistically significant (p-value: $0.9e^{-3}$).

A more interesting result is shown by the comparison of the TagInsert and the TagInsert L2R model. The difference in accuracy for Section 23 is now statistically significant, with a p-value of $0.2e^{-3}$. This hints towards the arbitrary ordering having some sort of meaningful positive effect on the model. The fact that the difference was not significant in the CCGBank may be attributed to the Rebank being more polished and complex, letting the benefits of the ordering shine. At the same time, there could just be some variance in convergence that would need to be explored by performing multiple training runs. As it would have proven to be quite demanding for our resources, we could not conduct such an analysis. However, at the very least, this indicates that there is a possibility of the arbitrary ordering being beneficial for the model. It is also worth pointing out that, while we could not run training multiple times to estimate an error of the accuracy reached, across all results and settings, the TagInsert model was consistently performing better than the left-to-right model.

All of this points to promising prospects for the model, and as such we continue our exploration on a more advanced setting for Combinatory Categorical Grammar: CCG constructive supertagging.

6. Experiment 3: CCG Constructive supertagging

In Chapter 4 and Chapter 5, we reported on the effects of breaking the left-to-right assumption in two tagging tasks. When it comes to Part-Of-Speech tagging, there was no inherent mechanism in the tags that suggested any benefit in breaking the ordering. This was remarked by the fact that no significant improvement was found when using TagInsert rather than using its left-to-right version. Conversely, we hypothesized that Combinatory Categorical Grammar would show a significant difference, as there is an underlying mechanism in the grammar expressed by the formalism that should not be restrained by a pre-determined ordering. The results of the experiment showed a significant statistical difference between TagInsert and its left-to-right version on the Rebank dataset. This was a promising direction to confirm our hypotheses. The next natural step is to investigate the effect of the proposed models on constructive supertagging.

In constructive supertagging, each category is not treated as an opaque symbol anymore. In this task, models are not trained to map each word to a simple label that ignores the rich structure of what a supertag represents, but they rather learn to build each category block by block. Indeed, each CCG supertag is comprised of a number of slashes and atomic categories that define the function of each word in a sentence. Furthermore, each atomic category is meant to be combined during parsing with atomic categories contained in other trees. How these labels are combined is strongly dictated by the direction of the slashes contained in each supertag, meaning that each category should be aware of its neighbors in both directions. As mentioned in Chapter 5, this implies the existence of an ideal order of generation, and is the main motivation for applying arbitrary ordering in this context. When it comes to constructive supertagging, the referencing of specific labels in the sequence is a lot more relevant, as the models have access to the actual atomic categories that are referenced. Thus, we hypothesize that the effect of arbitrary ordering in this task should be even more prominent when compared to non-constructive supertagging.

The models we chose for the experiment were previously presented in details in Chapter 3, but are also shortly summarized here as a reminder. In a similar

manner to the previous Chapters, the first step was to select a good baseline for a constructive supertagger. This way, we could inspect whether the arbitrary ordering of the TagInsert model could improve performances on a reliable constructive architecture. For this, we chose the *AddrMLP* constructive model from Prange et al. (2020). Because the architecture is relatively simple and the reported performances surprisingly high, it was a natural choice for the task. As we implement the architecture ourselves, this research also serves as a tentative reproduction of the results shown in the paper. Moreover, we propose a number of variations of the model that try to make more use of the neighboring supertags. It is important to note, that the Prange model is invariant to ordering, in which each category is independently constructed and has no effect on the construction of the other supertags in the sequence. Thus, it is not a left-to-right model. This implies that the focus of this Chapter will be on allowing *informed* arbitrary ordering, rather than breaking the left-to-right decoding like in the previous experiments. To elaborate, we investigate on whether the model can learn an ordering that can exploit the information held by each complex category.

For this purpose, in Chapter 3 we proposed a variant of the previously presented TagInsert model that can be used for CCG supertagging. To make it possible, we chose to make use of the Prange architecture and use their mechanism to build supertags. Effectively, this merges the TagInsert architecture of the previous Chapters with the constructive mechanism of the Prange model. The TagInsert module of the architecture chooses the next position to construct a supertag in, based on what other categories have been previously built in the sequence. Next, the Prange mechanism builds the category in said position using the representation produced by TagInsert. In this view, ordering is crucial to the task, and we hypothesize that an informed ordering can help the construction of the categories by exploiting the information contained in every supertag in the sequence.

For the purpose of replicating the results of Prange et al. (2020), models in this Chapter will be evaluated on both the CCGBank and Rebank dataset. Moreover, the experiment also investigates the effect of arbitrary ordering on the *long-tail* (Hockenmaier and Steedman, 2007). As a reminder, the most challenging aspect of CCG tagging is, indeed, correctly classifying the large amount of rare supertags that appear in the treebank. As machine learning models function by fitting the training data, it is naturally challenging for them to be accurate on sparse data. Depending on the architecture, considering for example non-constructive models, it may even

been outright impossible to correctly classify out-of-vocabulary categories. One of the main motivations to build a constructive supertagger is to do away with these limitation. By giving the model a deeper understand of the logic of the supertags and the capability of building each of its block individually, any category can be predicted, and the long-tail can, in principle, be successfully handled. For these reasons, when evaluating a constructive supertagger, the long-tail is of much interest. Thus, the behavior of the presented models based on supertags frequency is also reported in the Chapter.

6.1 Data and Pre-processing

In order to be able to compare results across all the models presented in this research, both the CCGBank and the Rebank dataset are used again to train and evaluate constructive supertaggers. As Prange et al. (2020) also evaluated their model on these datasets, we are able to reasonably compare results for the sake of the paper reproduction.

In the case of constructive supertagging, sentences up to length 70 were kept for training, validation and testing. This is in contrast with the previous maximum length of 50. The decision was made to be able to better compare results with previous research and Prange et al. (2020) specifically. At this point, the number of sentences cut is very minimal, while still keeping the memory constraint at a comfortable level. Regardless of design choices, one could argue that sentences longer than 70 words may include noisy data, as they likely contain some sort of error that led to such anomalous length. All the training was performed in Section 02-21 for both datasets, and all the models were evaluated on Section 00 and Section 23, again, for both datasets. The choice was made to stay in line with Prange et al. (2020) and prior work on the datasets. In terms of statistics, after cutting long sentences, the training data is comprised of 39553 sentences, Section 00 is comprised of 1907 sentences and Section 23 is comprised of 2407 sentences. By cutting sentences longer than 70 words, 51 sentences were removed from Section 02-21 (about 0.1% of the total number of training sentences); 6 sentences were removed from Section 00 (about 0.3% of the total number of sentences) and no sentences were cut from Section 23.

As the supertags are not treated as opaque labels anymore, it is not sufficient to simply map each one to its own label, and an additional pre-processing step is re-

quired. Following Prange et al. (2020), each category is processed and represented as a binary tree, where each node is either a slash operator (/ or \) or an atomic category (*S*, *NP*, ...). In the former case, the node always has exactly two children, the left child being the result and the right child being the argument. In the latter case, the node is always a leaf. Figure 3.3 showed an example of this representation. With the implementation of a proper data structure, it is then possible to build supertags in a top-down manner. In strong contrast to non-constructive supertagging, this entails the potential of constructing supertags that are unseen in the training data and makes the task of addressing the long-tail of the categories' distribution possible. Not only that, this method guarantees the construction of structurally sound supertags.

Finally, an additional training and testing setting was arranged for the constructive models. In line with Prange et al. (2020), we redistributed the Rebank dataset training data so that it could serve as a test for the ability of the model to generalize on the long tail. In practice, only sentences containing supertags that appeared at least 10 times in Sections 02-21 are kept for training, while the rest are used for testing. As depicted in the original paper, this redistribution captures most of the long tail in the test data, and forces the model to be tried on a sufficient number of out-of-vocabulary categories. Indeed, because of the implied property of the long tail, while there are many rare supertags, they appear only a handful of times. Table 6.1 shows how redistributing the dataset this way helps to augment the amount of infrequent supertags for the model to be tried on. In terms of sentences, we end up with 37336 train sentences and 2217 test sentences after the redistribution.

	All tags		Frequent tags >= 100		Common tags 10-99		Rare tags 1-9		OOV tags	
	Unique	Count	Unique	Count	Unique	Count	Unique	Count	Unique	Count
Rebank Sec 00	475	45460	199	44911	183	434	76	97	17	18
Rebank redistributed	1519	64926	188	61211	239	1079	34	114	1058	**2522**

Table 6.1: Effect on the long tail of the Rebank redistributed test set. By keeping only frequent and common supertags in the training data, the amount of OOV categories in the test set is greatly increased.

Hyperparameters The results shown in this Chapter for the Prange model and its variants have been obtained by using the same hyperparameters from the original paper. Namely, we used the AdamW optimizer, with a starting learning rate of $1e^{-4}$, β_1 of 0.9, β_2 of 0.999 and ϵ of $1e^{-6}$. The learning rate was adapted throughout

training by using the 1cycle learning rate policy (Smith and Topin, 2018), with a maximum learning rate of $1e^{-4}$. Because of how the scheduler is implemented, the amount of epochs to run is needed in advance. Thus, all the models are trained for a maximum of 6 epochs. All models seem to have converged by that point. The dropout probability in all layers is of 0.1. Like Prange et al. (2020), an upper bound of 6 for the maximum depth a tree can reach is in place. Even though conceptually there should be no need for such a limit, as it might desirable for the model to construct the supertag it deems most appropriate, regardless of its depth, in practice no observed supertag exceeds this limit. It is then convenient to limit the memory use to process and store the trees without influencing the model performance. Regardless, the approach used in Prange et al. (2020) and in all the other models presented in this Chapter are easily adaptable to changing this hyperparameter, so that trees of arbitrary depth can be constructed. The Constructive TagInsert model keeps the same hyperparameters from the previous Chapters, while adopting the optimizer and scheduler from the Prange model.

6.2 Results and discussion

The results are presented in three different sections, each one focusing on a different dataset. First, we present the results on the CCGBank, then we present the results on the Rebank, and finally on the Redistribution that augments the number of infrequent supertags in the test set. In the Rebank Section, an extensive analysis of the arbitrary ordering the TagInsert model learned is included.

6.2.1 CCGBank

The results for Section 00 and Section 23 of the CCGBank are shown in Table 6.2 and Table 6.3 respectively. For the sake of replication, Section 23 results include the accuracies reported in Prange et al. (2020). Like in previous Chapters, the best performing epoch after evaluating on Section 00 is used to test on Section 23. Thus, the analysis will mostly be focused on Section 23, as it serves effectively as our test set.

CCGBank (Section 00)	Epochs	Batch size	Overall	Frequent Tags (≥ 100)	Common Tags (10-99)	Rare Tags (1-9)	OOV
			N = 392, n = 44611	N = 171, n = 44172	N = 150, n = 357	N = 56, n = 66	N = 15, n = 16
Prange Replication	5	64	96.46%	96.80%	70.03%	34.85%	6.25%
Constructive TagInsert	5	64	96.28%	96.66%	63.59%	36.36%	**12.50**%
Variant 1 (concatenation)	6	64	96.62%	96.94%	**71.99**%	36.36%	0%
Variant 2 (weighted sum)	6	64	96.62%	96.95%	71.71%	31.82%	0%
Variant 3 (informed weighted sum)	6	64	**96.64%**	**96.96**%	**71.99**%	**37.88**%	6.25%

Table 6.2: Results of the constructive models on Section 00 of the CCGBank. Best results for each frequency are highlighted in bold.

CCGBank (Section 23)	Epochs	Batch size	Overall	Frequent Tags (≥ 100)	Common Tags (10-99)	Rare Tags (1-9)	OOV
			N = 435, n = 55371	N = 170, n = 54817	N = 177, n = 450	N = 66, n = 81	N = 22, n = 23
(Prange et al., 2020)	Up to 10	8	96.09%	96.44%	68.10%	**37.40**%	3.03%
Prange Replication	5	64	96.03%	96.42%	65.33%	27.16%	**4.35**%
Constructive TagInsert	5	64	95.77%	96.17%	64.89%	28.40%	**4.35**%
Variant 1 (concatenation)	6	64	96.10%	96.47%	67.33%	29.63%	**4.35**%
Variant 2 (weighted sum)	6	64	**96.17**%	**96.52**%	**70.89**%	32.10%	0%
Variant 3 (informed weighted sum)	6	64	96.09%	96.44%	70%	28.40%	**4.35**%

Table 6.3: Results of the constructive models on Section 23 of the CCGBank. Best results for each frequency are highlighted in bold.

Prange replication

The replication of Prange et al. (2020) seems to match the reported results. This is noticeable when looking at the evaluation on Section 23, where most of the bins dictated by frequency from the training data are comparable to the original model. On rare supertags, there is a noticeable gaps between the two, with the original paper reporting an accuracy of 37.4% and our reproduction falling a bit short with 27.16%. However, the difference in accuracy is likely not significant because of the small sample size. Unfortunately, it is not possible to test the hypothesis because of the slight difference in sample size that was brought by cutting some sentences in the reproduction. It is also possible that the original model was ran for some more epochs, as in the paper the authors do not report the actual number but rather state that it was trained for up to 10 epochs. Our replication also looks to be doing better on out of vocabulary categories, but this can again be attributed to slightly different datasets and insignificant sample size. In reality, both models only get one OOV category right, but the sample size is so small that no real analysis can be performed anyway. The Rebank redistribution will provide a better framework to test the models on the long tail.

Regardless, the replication seems to have been a success for the CCGBank, with the two models behaving in a very similar way across all frequencies. Note that

the model did not perform well on out of vocabulary categories when compared to, for example, Kogkalidis et al. (2019), which was able to reach up to 19.2% in accuracy. That is to be expected from the Prange architecture. Indeed, as we are essentially finetuning a BERT-like model, words are being tuned to fit the training data as much as possible, which in turn penalizes unseen data. This is of course a common property of machine learning models, but different architectures may encourage the model to generalize over unseen data better than this one.

Constructive TagInsert

The Constructive TagInsert model shows an improvement over its non-constructive counterpart. While in a way this was to be expected, as the constructive approach should, in general, be considered a better way to tackle the task, it is good to see that the discordance between its two modules discussed in Chapter 3 is not preventing it from functioning as a supertagger. Across frequency bins, results are slightly worse than all the other models. It is interesting that OOV supertags seems to be more accurately classified in Section 00, when compared to all the other models. However, like mentioned above, accuracies for this frequency bin are very misleading and should be (for now) ignored, due to the extremely low sample size.

What is most interesting is whether the model learned a specific ordering for generating the supertags. The ordering based on frequency and confidence was apparent in Chapter 4 and Chapter 5, but does the same apply in the constructive settings? In order to perform a good analysis, we opted to explore this point on the Rebank dataset, where the data is cleaner and improved. Thus, results on this will be reported in a later Section.

Prange variants

All the Prange variants seem to be performing in the task, although at first glance it does not look like adding the leaves information is being exploited in a very meaningful way. While there potentially seems to be some improvement over the other models, it is not much. The fact that between Section 00 and Section 23 the best performing model seems to swap between the weighted sum and the informed weighted sum variant also seems to point at some variance in the results, which makes it difficult to draw convincing conclusions.

Comparisons

Out of all the results in Section 23, it seems like the Prange Variant 2 that uses a weighted sum for the neighboring leaves is outperforming the others in most frequency bins. To compare results, we performed a paired t-test on all frequency bins. In order to have a more reliable statistical test, we also performed a permutation test (Good, 2000) for common, rare and OOV categories, where the sample size might falsify the normality assumption of the test. The permutation test is a non-parametric statistical test which verifies whether randomly permutating individual predictions between models does not change a statistic of choice. In our case, we calculated the difference in accuracy after 1000 permutations. There is no normality assumption in this test, which makes it ideal for small sample sizes.

When comparing the Variant 2 to our Prange replication, the paired t-test finds a significant statistical difference between the accuracies (p-value: 0.002). The difference between accuracies for frequent supertags is also significant (p-value: 0.037). For common supertags, both the t-test and the permutation test show a significant difference, with p-values of 0.002 and $5e^{-3}$ respectively. Rare categories show a non-significant difference from both tests (p-values: 0.346 and 0.208). Of course, OOV categories are also not significant.

The Constructive TagInsert model seems to be falling short of all the others, with only the long tail supertags accuracy being seemingly comparable. Namely, when compared to our Prange replication, the paired t-test finds a significant difference for general and frequent supertags accuracy (p-value: $6.79e^{-5}$ and $4.25e^{-5}$ respectively). In the long tail, accuracies over common categories hold no significant difference for both the t-test and the permutation test (p-value: 0.819 and 0.895 respectively). Rare supertags also are not statistically different, with p-values of 0.656 and 0.983 for both tests. Obviously, there is no significant difference for the OOV categories.

6.2.2 Rebank

The results for Section 00 and Section 23 of the Rebank are shown in Table 6.4 and Table 6.5 respectively. For the sake of replication, Section 23 results include what was reported in Prange et al. (2020).

Rebank (Section 00)	Epochs	Batch size	Overall	Frequent Tags (≥ 100)	Common Tags (10-99)	Rare Tags (1-9)	OOV
			N = 475, n = 45460	N = 199, n = 44911	N = 183, n = 434	N = 76, n = 97	N = 17, n = 18
Prange Replication	6	64	95.48%	95.95%	66.36%	24.74%	**11.11%**
Constructive TagInsert	6	64	94.85%	95.31%	64.98%	**30.93%**	5.56%
Variant 1 (concatenation)	5	64	95.42%	95.88%	67.44%	24.74%	5.56%
Variant 2 (weighted sum)	6	64	**95.53%**	**95.99%**	67.97%	24.74%	0%
Variant 3 (informed weighted sum)	6	64	95.46%	95.89%	**68.66%**	**30.93%**	**11.11%**

Table 6.4: Results of the constructive models on Section 00 of the Rebank. Best results for each frequency are highlighted in bold.

Rebank (Section 23)	Epochs	Batch size	Overall	Frequent Tags (≥ 100)	Common Tags (10-99)	Rare Tags (1-9)	OOV
			N = 538, n = 56395	N = 199, n = 55698	N = 222, n = 563	N = 90, n = 106	N = 27, n = 28
(Prange et al., 2020)	Up to 10	8	94.58%	95.01%	67.44%	**34.89%**	3.70%
Prange Replication	6	64	95.08%	95.57%	65.01%	25.47%	3.57%
Constructive TagInsert	6	64	94.44%	94.91%	64.30%	29.25%	0%
Variant 1 (concatenation)	5	64	94.92%	95.39%	66.07%	26.42%	3.57%
Variant 2 (weighted sum)	6	64	**95.21%**	**95.67%**	66.79%	28.30%	3.57%
Variant 3 (informed weighted sum)	6	64	95.17%	95.59%	**69.80%**	29.25%	**7.14%**

Table 6.5: Results of the constructive models on Section 23 of the Rebank. Best results for each frequency are highlighted in bold.

Prange replication

When evaluating on the Rebank dataset, the replication still looks comparable to Prange et al. (2020). In this case, the model seems to be even outperforming the original paper on overall accuracy and common supertags. Again, it is not possible to perform a reliable hypothesis test due to the slight difference in test data, but at the very least we can say that the results are appropriate. In a similar manner to the CCGBank, rare categories seems to underperform in our reproduction, with the original paper reporting 34.89% accuracy and our reproduction settling on 25.47%. Again, the low sample size and potential difference in number of trained epochs makes it difficult to draw any conclusion. On OOV categories, the model seems to be performing as exactly like in the CCGBank, getting only one correct out of the total.

Constructive TagInsert

Maintaining the trend from the CCGBank, the constructive TagInsert model improves over its non-constructive counterpart. Curiously, the model seems to be under-performing in general, with the exception of the rare supertags, in which other models struggle a bit more. This behavior was also present in the CCGBank, although again, the low sample size of the frequency bin makes it hard to reach conclusions. In general, there seems to be no major differences with the CCGBank

when evaluating on the Rebank, except for an all around decrease in accuracy, which happens to all the models and is easily explained by the larger supertag vocabulary of the dataset.

Ordering As opposed to the non-constructive TagInsert model, which learned from the data by inserting opaque labels and decided on the ordering based on the confidence and frequency that such insertions are made, the constructive version learns its orderings in a much more unpredictable way. One would expect for the ordering to also be based on confidence, but it is difficult to say how the loss function influences learning in this case. Indeed, each tree loss is a composition of losses from a number of atomic decision that corresponds to the number of nodes in the tree. Moreover, each individual decision is based on not only the vector produced by the decoder, but also the position and slashes ancestors of each individual node. The way the TagInsert module interacts with the Prange module is also puzzling, in which it is difficult to say how the architectures handles the difference in the understanding of the supertags (opaque vs composite labels). Regardless, because of the TagInsert module is common between the models, ordering is chosen by extracting the maximum probability supertag across all positions. This means that orderings represent what the model thinks are the easiest, in the case of early insertions, and the hardest, in the case of late insertions, category to predict. Thus, we ran an explorative research comparing the ordering of the constructive and non-constructive versions of TagInsert. By searching for patterns and comparing the models, we look for possibly interesting orderings the model learns.

Frequency based ordering Figure 6.1 shows a comparison of the ordering chosen by the models for three of the most frequent supertags in the corpus. It is immediately apparent how the non-constructive model bases its ordering on a product of frequency and ambiguity of the category. The supertag N is both frequent and ambiguous, and thus the non-constructive model usually inserts half way through decoding. On the other hand, the constructive model seems to have learned that the category is easy to predict, and usually inserts it at the start. As the constructive model learns as a product of individual decisions identified by the nodes, there might be a correlation with the number of nodes a supertag possesses. The behavior of the non-constructive model is made even clearer when inserting a comma, one of the most frequent and unambiguous category in the corpus. Interestingly, the constructive module seems to recognize this by usually inserting it towards the

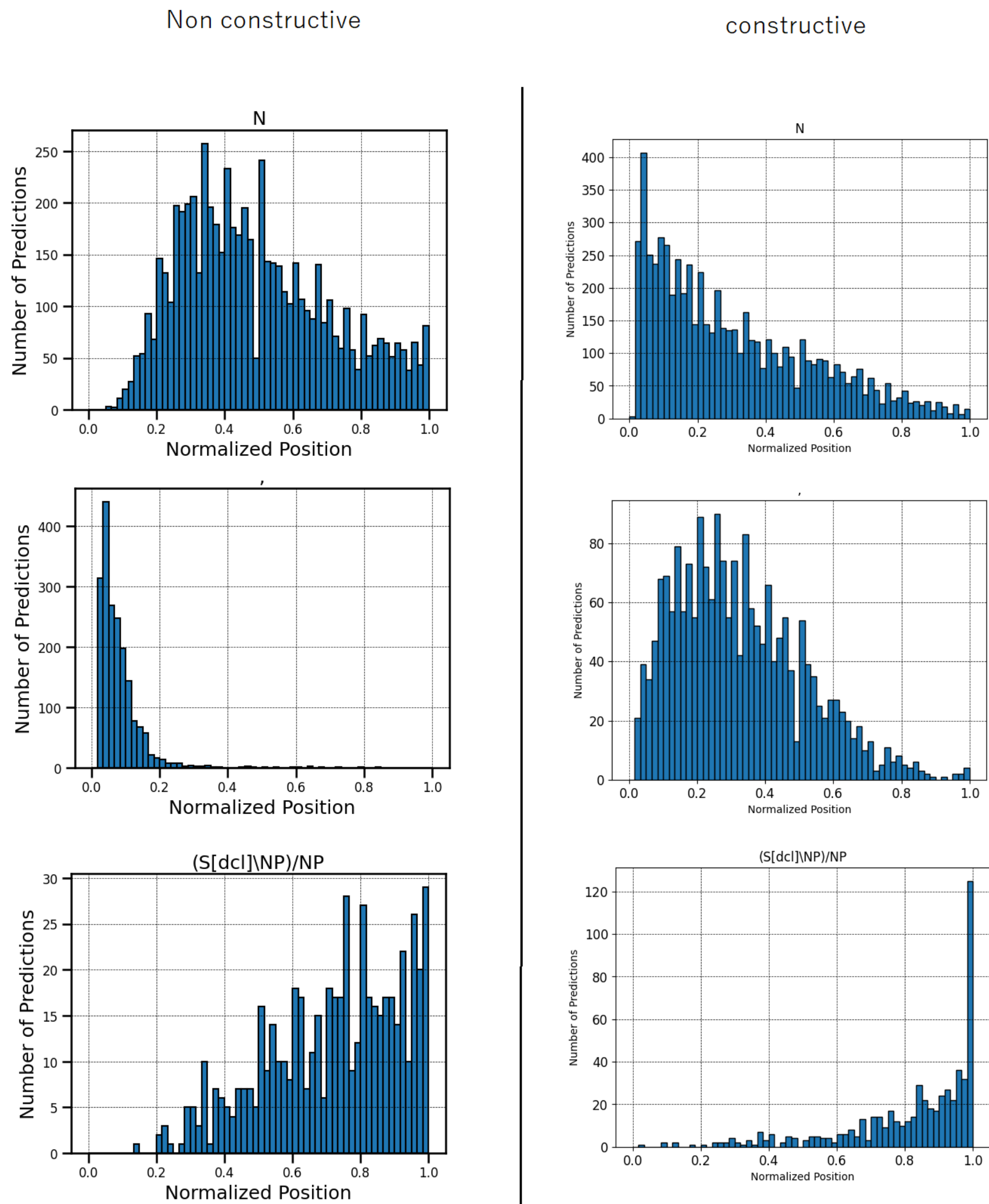

Figure 6.1: Some of the most frequent supertags and their normalized ordering during decoding for the non-constructive and constructive TagInsert model.

start of decoding, but is not as committed as its non-constructive counterpart. Finally, the supertag *(S\NP)/NP* of transitive verbs is recognized by both model to be a difficult category, albeit very frequent. The constructive model however shows a stronger tendency for inserting it at the very last chance it gets, indicating that it is one of the hardest categories for it.

Figure 6.2 shows the proportion of supertags inserted in the first half of de-

coding a sequence. Each histogram represent one of the frequency bins of Table 6.4. In the most frequent supertags, both models predict about 50% of categories in the first half of decoding. However, when supertags get more rare, the non-constructive model will almost never insert them in the first half, while the constructive model seems to be relatively unfazed.

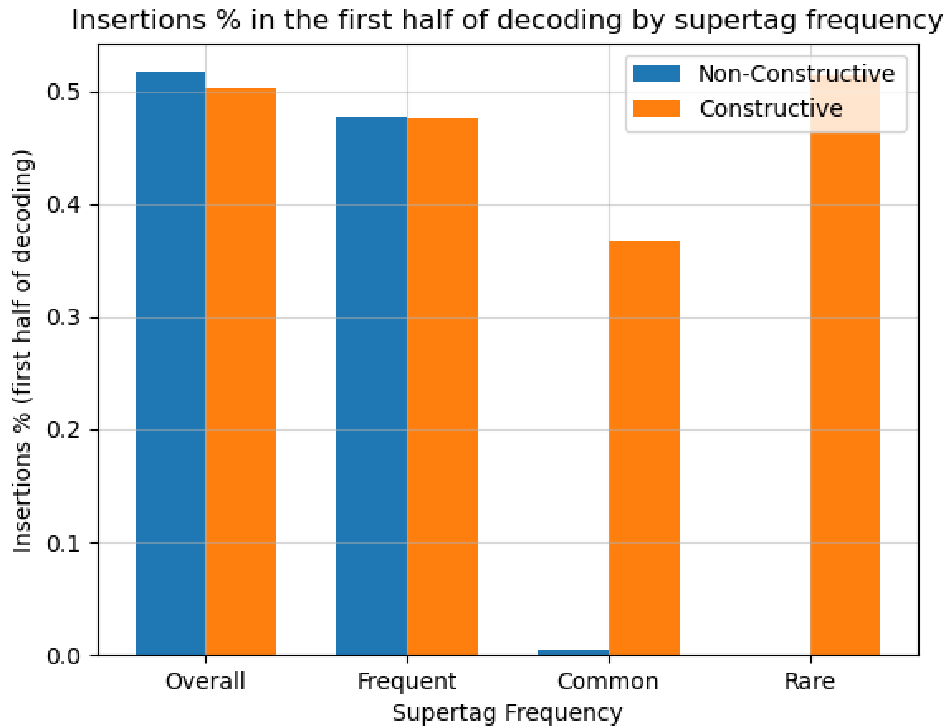

Figure 6.2: Proportion of supertags inserted in the first half of decoding the sequence, aggregated by frequency.

Depth and nodes length based ordering Following the hint from the frequency based orderings, we explore whether the constructive model has a tendency of considering supertags that have low depth or low number of nodes as easier to predict. Figure 6.3 shows the ordering of both versions of TagInsert based on the depth of the supertags. It is worth noting that the lower the depth, the more frequent the category is in the corpus. When the depth is very high, the supertags are frequently in the long tail. Interestingly, both model share a similar pattern up until trees of depth 3. Supertags of depth 0 (single nodes) are deemed easy and are inserted at the start of decoding, but at depth 1 both models already agree that difficulty increases. Starting from depth 3, the models begin to diverge, with the non-constructive model recognizing the difficulty of the long tail and the construc-

tive model opting for a more uniform ordering.

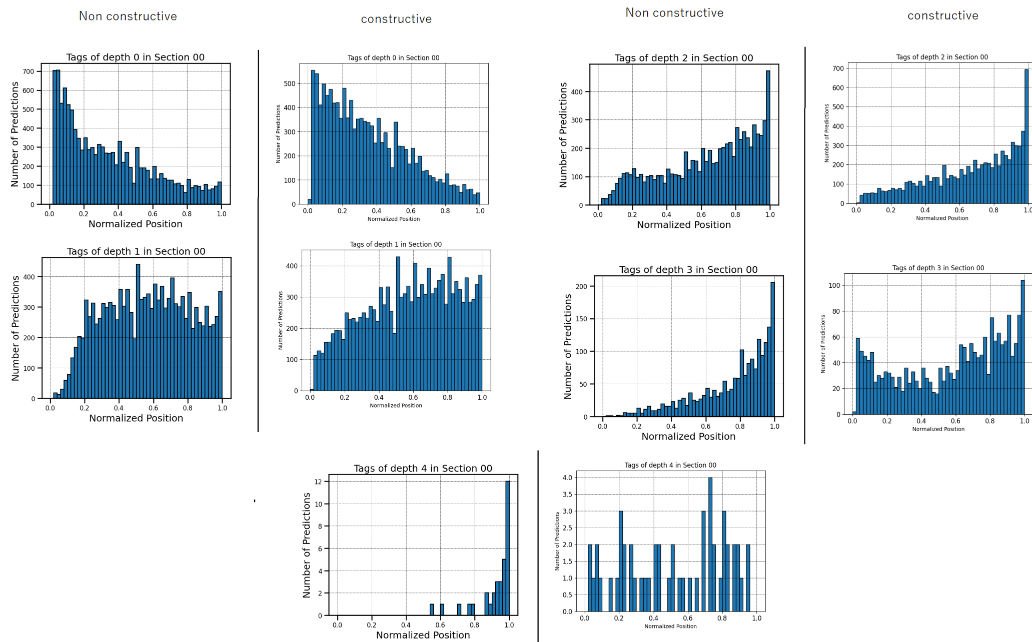

Figure 6.3: Normalized ordering aggregated by depth of supertags for both the constructive and non-constructive TagInsert models.

As another point of view, we can also inspect the ordering based on the number of nodes in supertags. Figure 6.4 shows the analysis for some notable lengths, which shows a very similar pattern to the depth analysis. This is to be expected, as the two concept are obviously highly correlated. However, some differences are implied, as for instance supertags of depth 2 can vary significantly in the number of nodes they possess. When looking at the ordering for trees that contain 15 nodes, it is clear how the two models diverge, with the constructive model seemingly inverting the trend. We can conclude that there is some correlation with depth and node number in the ordering of the constructive supertagger, albeit its behavior becomes puzzling with more complex categories.

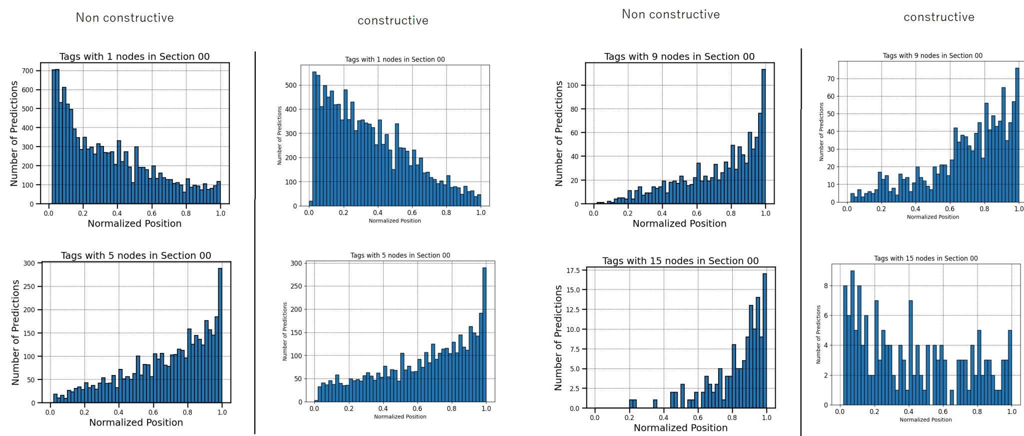

Figure 6.4: Normalized ordering aggregated by number of nodes of supertags for both the constructive and non-constructive TagInsert models.

Ambiguous transitive verbs Intrigued by the stark difference shown by the models on transitive verbs in the frequency analysis, we turn to an investigation on transitive verbs that are also very ambiguous. Up until now, the analysis has been focused on the supertags, but it may be that the constructive model pays more attention to the words themselves when compared to the non-constructive one. The main reason to believe this would be because of the Prange model essentially finetuning the BERT-like words embeddings. Through learning, each word's representation is shaped to construct accurate trees. Thus, Figure 6.5 shows orderings of three of the most ambiguous transitive verbs in the corpus. When looking at the word *is*, it is apparent how the non-constructive model learned it to be a difficult word. The constructive model also deems it difficult, but the difference is noticeable. A very interesting find can be seen with the contraction *'s*. The non-constructive model recognizes it as a difficult word, likely for its ambiguity as both a verb and its use as English possessive. On the other hand, the constructive model thinks the opposite and is quite happy to start predicting with it. This example is particularly interesting, as the difference between using the word for its possessive role or for its role as a verb would be clearly expressed in the contextualized word's embedding produced by the RoBERTa model.

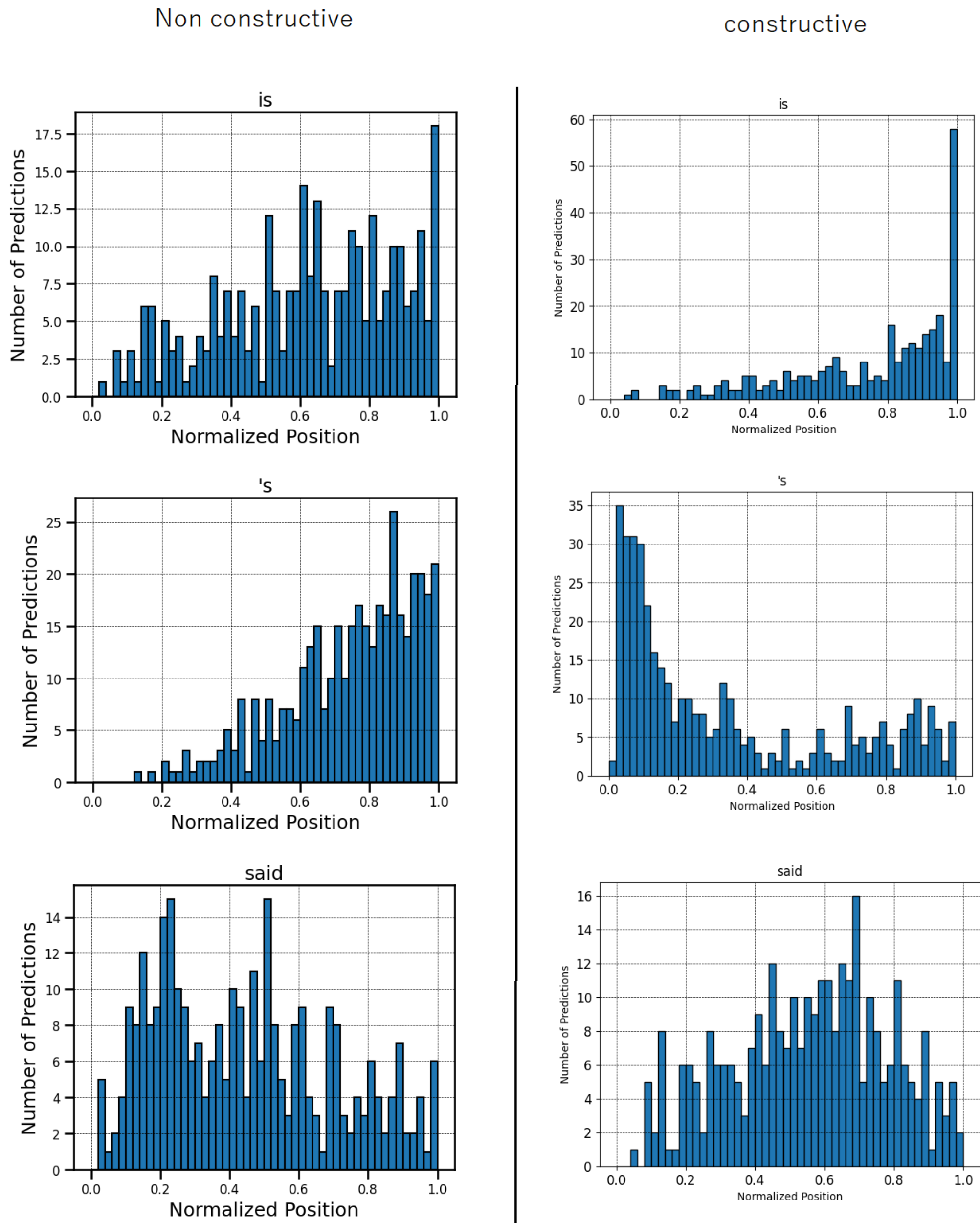

Figure 6.5: Normalized ordering of some ambiguous transitive verbs for both the constructive and non-constructive TagInsert models.

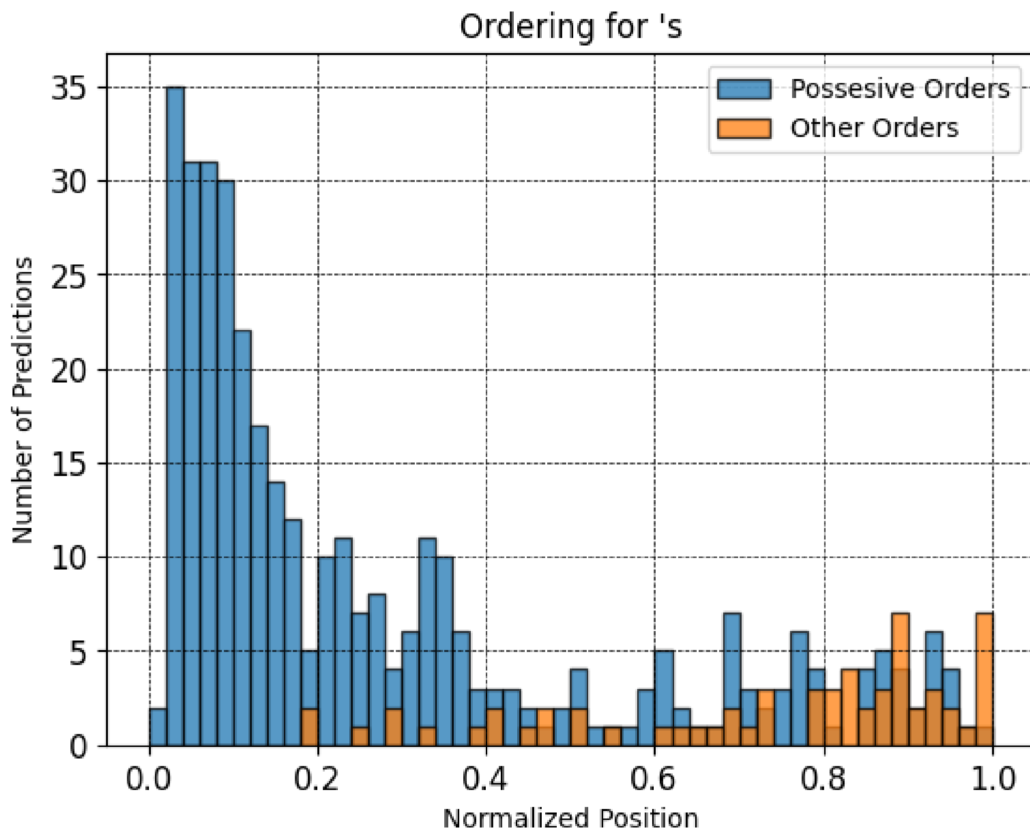

Figure 6.6: Normalized ordering of the word *'s*, differentiating on when it was used as a possessive construct or not. It is evident that the model recognizes the difference and learns that the possessive case is easier.

Figure 6.6 shows the difference of the ordering in cases where *'s* was used as a possessive construct or whichever different one. It is now apparent how the contextualized word embeddings fine-tuned during learning play a major role in how ordering is determined. This is a very different approach from the non-constructive model. Although it is still based on the confidence of the model about correctly tagging a certain word, what is considered difficult to the constructive model is also a product of the word's representation and role in the sentence.

Prange variants

The Prange variants retain a good performance, with little variance between them. When compared to Prange et al. (2020), they seem to outperform the reported results. This does not hold in the rare supertags, in which the original model really shined.

Comparisons

Similarly to the CCGBank evaluation, the Prange Variant 2 that computes a weighted sum of the surrounding leaves is the best performing model overall. However, in the long tail, it is the Variant 3 that slightly takes the edge. First, let us compare the Variant 2 with the Prange replication. The difference in overall accuracy (95.21% and 95.08%) is statistically significant, with a p-value of 0.018 from the paired t-test. Difference in frequent supertags accuracy (95.67% and 95.57%) is also significant, with a p-value of 0.044. On common categories, the difference (66.79% and 65.01%) is not significant for both the paired t-test (p-value: 0.2043) and the permutation test (p-value: 0.2837). On rare supertags, the difference (28.30% and 25.47%) is not significant for both the paired t-test (p-value: 0.3196) and the permutation test (p-value: 0.4956). As the two models perform the same in OOV categories, there is no difference to test.

Next, we compare the Variant 3 model with the Prange replication, with the intention to see whether its good performance in the long tail is significant. Interestingly, when testing on common supertags accuracy (69.80% and 65.01%), there is a significant difference for both the paired t-test (p-value: $3e^{-4}$) and the permutation test (p-value: 0.002). On rare categories, the difference in accuracy (29.25% and 25.47%) is not significant for both the paired t-test (p-value: 0.25) and the permutation test (p-value: 0.3956).

The Constructive TagInsert model seems to be underperforming on general accuracy and frequent supertags, while on the long tail is comparable with all the other models. Specifically, when compared to the Prange replication on overall accuracy (94.44% and 95.08), the paired t-test shows a significant difference, with a p-value of $1.30e^{-18}$. On frequent supertags, the accuracy difference (94.91% and 95.57%) is also significant, with a p-value of $1.02e^{-19}$. On the long tail, no accuracy difference was found to be significant.

6.2.3 Redistribution

Rebank Redistribution	Epochs	Batch size	Overall	Frequent Tags ($\geq$ 100)	Common Tags (10-99)	Rare Tags (1-9)	OOV
			N = 1519, n = 64926	N = 188, n = 61211	N = 239, n = 1079	N = 34, n = 114	N = 1058, n = 2522
Prange et al. (2020)	Up to 10	8	89.01%	92.70%	**54.03%**	**26.48%**	**10.96%**
Prange Replication	6	64	**89.20%**	**93.25%**	**54.03%**	21.05%	8.84%
Constructive TagInsert	6	64	88.10%	92.21%	48.75%	21.93%	8.33%
Variant 1 (concatenation)	6	64	89.16%	93.22%	53.38%	17.54%	9.24%
Variant 2 (weighted sum)	5	64	89.10%	93.14%	52.83%	17.54%	9.79%
Variant 3 (informed weighted sum)	6	64	89.15%	93.17%	53.85%	22.81%	9.64%

Table 6.6: Results of the constructive models on the Rebank redistribution, where only sentences with supertags appearing more than 9 times in Sections 02-21 were used for training. The rest of the sentences were used for the evaluation. Best results for each frequency are highlighted in bold.

Prange replication

By redistributing the dataset and evaluating our replication in a context in which the amount of OOV is much larger, we can inspect the ability of the model to generalize on the long-tail. This was not really possible before, as the sample sizes were way too low for OOV categories. The results show that, once again, the replication is working as intended, especially in overall accuracy and supertags that do not stand in the long tail. In the rare categories frequency bin, we again observe a considerable gap in accuracy when compared with Prange et al. (2020). The redistribution however was not able to populate this category very significantly, and thus the result is likely not significant. On OOV categories, the model cannot reach the accuracy of the original. As we could not match the sample sizes for the frequency bins on the original paper, albeit following their method of redistribution, we assume that there was a mistake in reporting. Thus, we perform a tentative two proportion Z-test to verify whether the difference in accuracy for OOV categories is is significant. Indeed, with a p-value of $1.55e^{-15}$, the difference is significant, indicating that our reproduction could not match the original results.

Constructive TagInsert

The Constructive TagInsert model retains its behavior from the previous datasets. Overall, the model is slightly underperforming. The rare supertags are, again, the most notable result of the architecture. However, like mentioned the redistribution did not particularly benefit the sample size of this frequency bin. Thus, we still find it difficult to draw any conclusion. Having a larger sample size for OOV categories did not help the model, which performs the worst when compared to all the other

models.

As for the ordering, while most of the previously reported behavior on the Rebank is also present in the redistribution, there are some specific differences that are quite puzzling. Figure 6.7 shows one of the most extreme examples of this in the period tag, with what should be a strongly unambiguous category in both the word and the supertag side. The model opts to insert it at the end of decoding, hinting that it learned for it to be a difficult category. We could not explain this behavior, but it is an interesting question of how learning and convergence change when the underlying distribution of the training data is heavily modified.

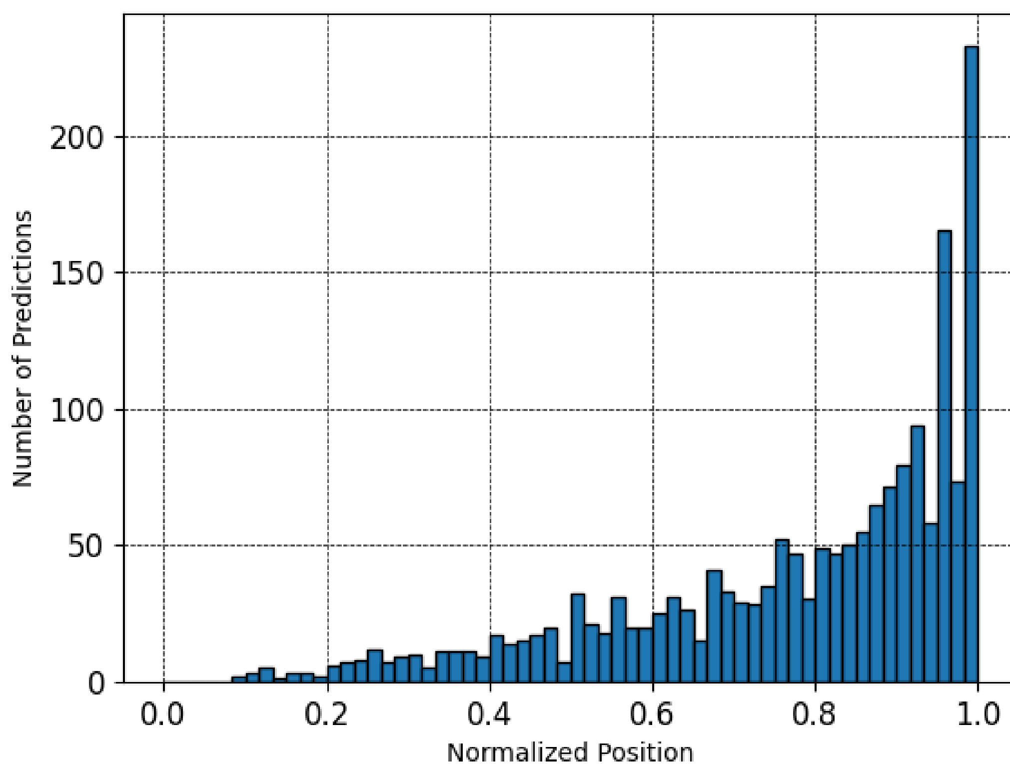

Figure 6.7: Normalized ordering of the period in the redistribution of Rebank. In this case, the period is frequently inserted last.

Prange variants

All the Prange variants can perform in the redistribution, much like in the previous datasets. We do not notice any difference of note in the performance between the variants. Variant 3 seems like it does slightly better overall, but like in previous datasets, the difference is minimal.

Comparisons

In the redistribution, the better model on the long tail is the original Prange et al. (2020). The replication is comparable to it, with no statistically significant difference in any bins except for the OOV categories, as mentioned above. This may be caused by the differences in implementation we noted in this Chapter. At the same time, we noticed some inconsistencies with the sample sizes in the various frequency bins for the redistribution. As we followed the straightforward method the authors chose the redistribute the Rebank, we are not sure whether there was a mistake on reporting or if there are some actual differences in our replication that were not reported in the paper.

The Constructive TagInsert model still underperforms. In this case, it is particularly apparent in the OOV categories, in which it is the worst performing model. When compared to our replication, the difference in accuracy (8.84% and 8.33%) is significant, with a p-value of 0.025 from the z-test.

All the Prange variants show good results, although in the case of the redistribution it is our replication that has the better accuracy in most cases. However, in the case of OOV categories, the variants consistently outperform the Prange replication, with the best model being Variant 2. The difference in accuracy between this variant and the replication on OOV categories (9.79% vs 8.84%) is significant, with a p-value of $1e^{-4}$.

7. Summary and Conclusion

In this thesis we explored a method to introduce arbitrary ordering in Transformer models, the TagInsert model. Thanks to its design, this architecture can break the pre-determined left-to-right ordering that is so often used in application that use the Transformer.

We deployed the model in the context of sequence tagging, hypothesizing that depending on the nature of the task, decoding in an informed ordering learned by the model could provide a number of benefits. For example, we hinted towards mitigating the inherent problem presented by most auto-regressive, left-to-tight models, that propagates potential mistakes done early in the sequence until the end of generation. Most notably, by using the model for CCG tagging, in which supertags hold an inherent mechanism that suggests a preferred order of decoding, we provided an extensive analysis on both the effect of arbitrary ordering and on what ordering the model settles on. Specifically, in the context of CCG constructive supertagging, we introduced yet another architecture, with the purpose of adapting TagInsert to the task of building each category in a block by block, top-down manner.

We summarise our findings on each task here, while also noting the limitations we had to face. Finally, we discuss a number of proposals for future research that could use the arbitrary ordering set up.

7.1 Part-Of-Speech Tagging

In Chapter 4, we implemented three models for the task of Part-Of-Speech tagging, while also fine-tuning a BERT-like model to serve as a baseline.

First, we showed how the original Encoder-Decoder Transformer from Vaswani et al. (2017) can be used for tagging, effectively re-imagining the task as a sequence-to-sequence problem. Although the architecture is able to do the task with acceptable results, it is clear that models specifically designed for tagging are much better suited for it. Specifically, the results and analysis provided showed how the architecture does not model a strict enough dependency between the words and their

tags, simply trying to produce a sequence of tags as consistent as possible to the sequence of words. In practice, this resulted in tags being skipped and misalignment in the sequence.

Second, we introduced TagInsert, providing a method to do away with the left-to-right ordering the architecture assumes. We reported how, while not reaching the performances of the encoder-side tagger baseline, it largely improves on the Vanilla Encoder-Decoder Transformer. Moreover, by inspecting the ordering the model learned, we show how the produced tag at each time step is determined by the confidence of the model. This naturally translates into easier tags being predicted first and more difficult ones being kept for last. We make it a point that this could help alleviate the error propagation typical of auto-regressive models, in cases in which hard tags are found at the start of the sequence.

Finally, by comparing TagInsert with TagInsert L2R, a slightly modified version in which the model is forced to learn a left-to-right ordering, we show how in the context of Part-Of-Speech tagging we could not find a significant difference in performances when using arbitrary ordering. However, qualitative analysis showed instances in which the TagInsert model predicted better tags. As no strong underlying mechanism in Part-Of-Speech tags is present that would suggest a preferred ordering of decoding, we turn to other tasks in which such feature is present.

7.2 CCG Tagging

In Chapter 5 we re-used the same setup of Chapter 4 in order to explore the effect of arbitrary ordering on a tagging task which could better benefit from it. As each Combinatory Categorical Grammar supertag contains information about direct dependencies with the other element in the sequence, we presented the task as a better framework to test TagInsert. By evaluating the architectures on both the CCGBank and the Rebank, we investigated the performances of the models and we checked for better effects of the arbitrary ordering.

Most of the models that were also trained for Part-Of-Speech tagging showed similar behavior for this task. There was no significant difference observed on the CCGBank dataset. However, on the Rebank dataset, the TagInsert model showed a statistically significant improvement over its left-to-right counterpart. This suggested that there could indeed be some benefit in allowing for arbitrary ordering in this task. Although there may be some variance in convergence to be tested with

more extensive training, it was motivation to further investigate the task. We also presented an analysis on the ordering learned by TagInsert, which was similar to the ordering learned for Part-Of-Speech tagging, in which priority was given to easy and unambiguous supertags.

Thus, we moved to a more complex context for CCG tagging, in which each category is treated as a collection of atomic categories: constructive supertagging. By making the model aware of each component the supertags are made of, it was plausible that it could more effectively exploit the bi-directional information they hold.

7.3 Constructive Supertagging

In Chapter 6 we explored the task of CCG constructive supertagging and the effect of arbitrary ordering with TagInsert. Again, we evaluated on both the CCGBank and the Rebank. As one of the most appealing part of constructive supertagging is being able to predict any supertag, even out of vocabulary, we paid special attention to analysing results on the long tail of the treebank distribution. Moreover, we adopted the same approach as Prange et al. (2020) and redistributed the Rebank dataset to have a better distribution of OOV categories to evaluate the models on.

First, we implemented the AddrMLP constructive supertagger described in Prange et al. (2020). The model plays a key role in adapting TagInsert to being a constructive model.

Second, we adapted the TagInsert model into Constructive TagInsert, a constructive supertagger that decides which position to build a supertag in at each time step. Once the position is decided, the category is constructed with the mechanism of Prange et al. (2020). As the original Prange model was not a left-to-right model, but rather a more refined version of a fine-tuned BERT-like model, this Chapter does not make a direct comparison between left-to-right and arbitrary ordering like the previous ones did. However, because dependencies between categories in the sequence are so strong in CCG, we rather explore what is the effect of introducing an informed ordering and potentially exploiting such dependencies in the Prange et al. (2020) approach. The Constructive TagInsert model is close but underperforms Prange et al. (2020), with statistically significant differences in most settings. We hypothesized that this is due to the apparent discordance between TagInsert and the Prange et al. (2020) model. Indeed, TagInsert still perceives each

supertag as an opaque label, and the information encoded by the model with this in mind is passed to the Prange model to build supertags with. Regardless, the model still improves over its non-constructive counterpart. We also presented an extensive analysis on the learned ordering by the constructive model, drawing comparisons with the ordering learned in Chapter 5 for non-constructive CCG tagging. The analysis showed a different behavior when compared to the non-constructive model, in which the word's contextualized embeddings produced by the BERT-like model plays a more prominent role on determining what makes a supertag easy to predict. At the same time, we also found some puzzling results that we could not explain when it comes to ordering, which may again stem from the fact that learning is more unpredictable in a setting where the two modules perceive the task differently.

Finally, we included three variants of the Prange et al. (2020) model. By exploiting the top-down approach of supertag construction, we embraced the idea that a supertagger should be almost parsing, and fed the information of previously predicted atomic categories in the sequence to predict each node of a supertag. In this view, the concept of ordering shifts on a vertical sense rather then horizontal, while keeping consistent with the rules of CCG parsing. All three variants perform similarly to the original model, with some cases even significantly outperforming it according to the statistical tests. The improvements are small and the variants were small and easy to implement modifications on the Prange model. We leave it to future research to explore further possibilities of a constructive supertagger that makes use of the logic of parsing to build its categories.

This Chapter also served as a replication of the results reported in Prange et al. (2020). The replication is mostly successful, with the only statistically significant differences present in the redistributed dataset. However, as the original paper showed some inconsistencies with our data in that case, it is hard to gauge whether that is a meaningful results.

All in all, the experiment was able to show that there is potential in using informed orderings in constructive supertagging. While the Constructive TagInsert model was not the best implementation to reach that goal, there is a lot of design space to explore the subject.

7.4 Future work

First, while it could be argued that letting the model decide on the order of decoding is a strictly more general approach than left-to-right decoding, it must be recognized that the efficiency complication that arise from the arbitrary ordering are not practical. Some research can be done to overcome the issue. For example, by calculating the trajectory before-hand, it is theoretically possible to build a future mask to use during training. The sampled trajectory would make the next position to insert in explicit, so that the mask can be built and training can proceed like it is done in Vaswani et al. (2017). However, this would not work for inference, in which the ordering cannot ever be known in advance. It is worth noting that the construction of a future mask would only be possible in a setting in which we know the target sequence in advance, like tagging. In other architectures that focus on, for example, text generation like Stern et al. (2019), other techniques to solve the problems brought by absolute positional encoding need to be introduced. For example, Zhang et al. (2023) offers a method to reuse positional information across timesteps.

Second, we note again that our exploration of the Constructive TagInsert was strongly limited by the inherent discordance between its TagInsert and Prange modules. Other ways to adapt TagInsert to the constructive paradigm should be explored, so that the architecture can better solve the differences between the two approaches. Moreover, as Prange et al. (2020) was invariant to position, some work can be done on investigating the effect of arbitrary ordering as opposed as left-to-right ordering, much like we did in Chapter 4 and Chapter 5. For example, it would be interesting to explore the approach presented by Kogkalidis et al. (2019), in which they also used a Transformer architecture to generate the supertag sequence in a left-to-right manner, almost as if it was text generation. As they were able to achieve good results and reworking their model with TagInsert would be pretty straightforward, it would be interesting to see whether breaking their left-to-right approach would benefit the model.

Third, we would like to encourage further work on constructive supertagger working as a parser. Actually, supertagging was first described as *almost parsing* in Bangalore and Joshi (1999), which illustrates quite well all the dependencies a supertagger is supposed to model to accurately perform. The three Prange variants presented in this research are very simple takes on this approach, but much more

refined techniques can surely be developed and explored.

Fourth, it is worth noting that all the data we used are from English corpora. Besides being a saturated task for the language, it would be interesting to explore how the arbitrary ordering effects languages in which dependencies and constructs are more free in terms of ordering. For example, Kogkalidis et al. (2019) tested their model in a Dutch corpus.

Finally, in order to keep the demand of resources low, we could not run training multiple times for all the architectures and frameworks. This means that all our statistical testing was limited over hypothesis testing on single runs. This is a limitation of our work. However, there are hints that even though TagInsert does not significantly improve other its left-to-right variant, it can consistently outperform it. For this reason, we believe that reproducing our result through the code base we provided and estimating the variance in convergence could prove useful to highlight whether it is generally a better approach when compared to the traditional fixed ordering.

All in all, we hope that with this work shows the potential in exploring the decoding order that is frequently overlooked when using Transformer models. As such models take over the landscape of Natural Language Processing, it is important to consider that not all tasks are necessarily suited to be solved in a left-to-right manner. Although arbitrary ordering is currently under-explored, likely because of problems like less-efficient training, there is potential in exploring the effect of models learning their own preferred ordering. In the context of Natural Language Processing, most tasks naturally follow the left-to-right way in which human produce and analyze text, but with Transformer being so easily adaptable to many domains, it could prove useful to develop an architecture in which the ordering is not pre-determined, but rather learned by the model itself.

7.5 Infrastructure and computation

The experiments for this research were conducted using two primary computational environments. Initial development and smaller-scale experiments were run on Google Colaboratory (Colab), which provided a convenient and flexible environment. For more computationally intensive tasks, such as training TagInsert and its constructive variant, we utilized the Snellius supercomputer cluster, generously offered by SURF.

The runtime of the models varied greatly depending on the architecture. As mentioned, TagInsert cannot make use of the benefits that come from masking attention, and is forced to inefficiently forward the sequence to the decoder at each time step. Thus, the model needs a significant amount of time to converge, when compared to the Vanilla Encoder-Decoder Transformer and the FineTuned DistilBERT model. In Experiment 1 and 2, TagInsert took about 20 hours to converge. In contrast, the left-to-right version of the model only took about 3 hours. In Experiment 3, in which the model also had to train the Prange et al. (2020) module, convergence took about 50 hours. The Prange reproduction took about 10 hours to converge, which is comparable to what was reported in the original paper.

7.6 Ethics and Reproducibility

The Ethics and Privacy Quick Scan of the Utrecht University Research Institute of Information and Computing Sciences was conducted and no issues were found. It classified this research as low-risk with no fuller ethics review or privacy assessment required. All the code for the implementation of the models and experiments presented in this research is available at GitHub Repository.

A. Appendix A

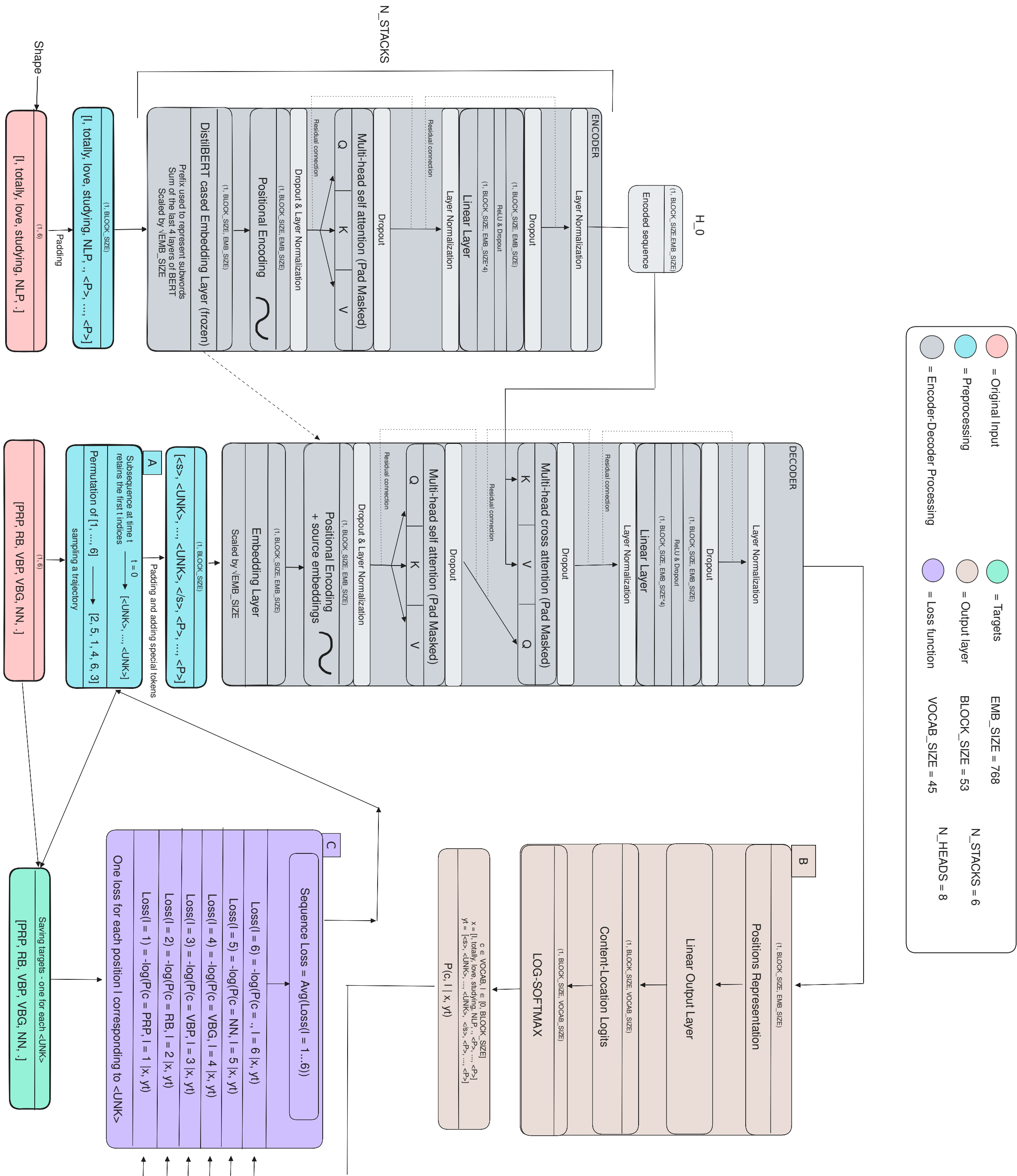

Figure A.1: Visualization of the training procedure of TagInsert during a single trajectory step. Hyperparameters refer to the POS tagging experiment reported in Chapter 4.

B. Appendix B

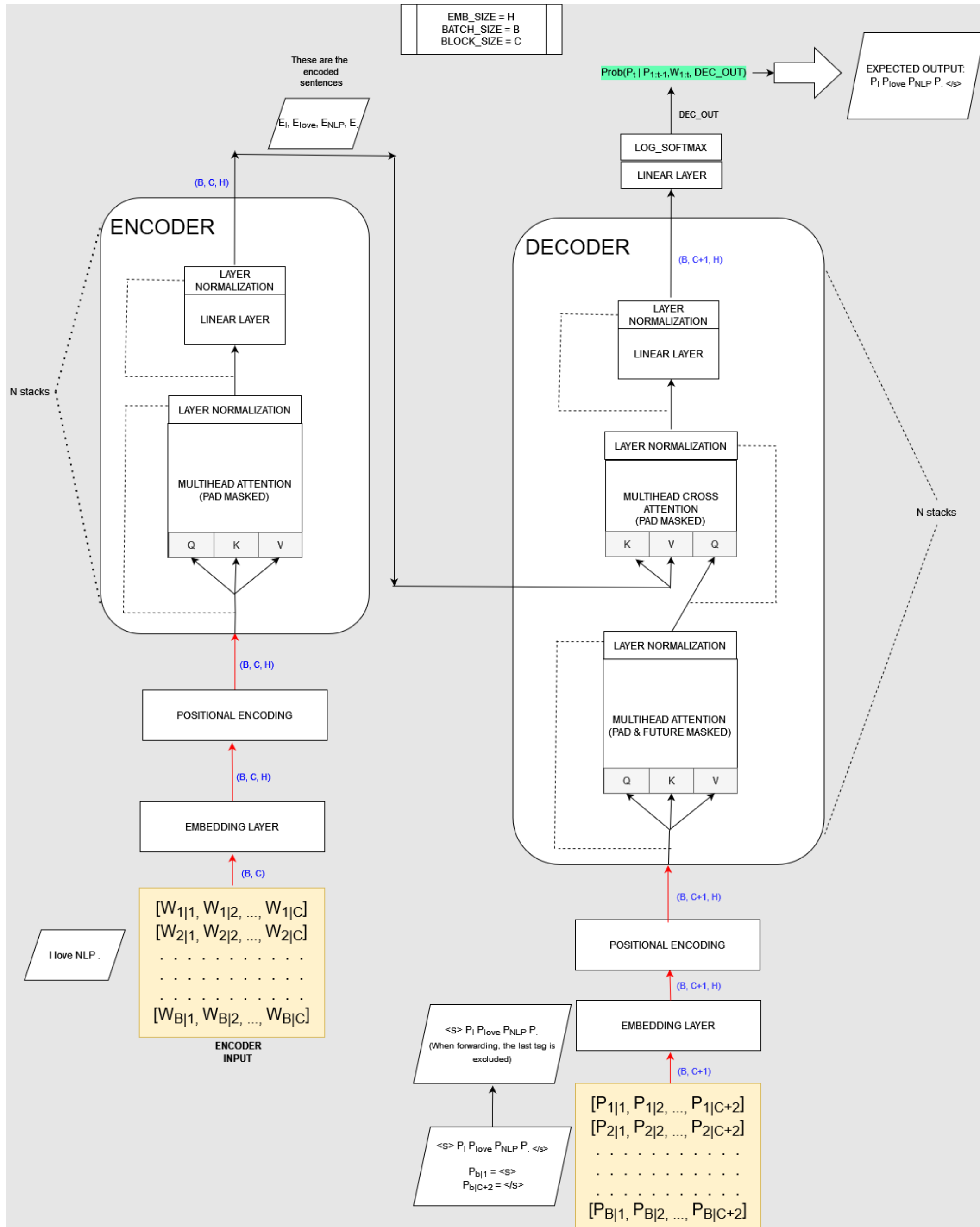

Figure B.1: Architecture of the Encoder-Decoder Transformer.

Bibliography

Ambati, B. R., Deoskar, T., and Steedman, M. (2016). Shift-reduce CCG parsing using neural network models. In Knight, K., Nenkova, A., and Rambow, O., editors, *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 447–453, San Diego, California. Association for Computational Linguistics.

Bangalore, S. and Joshi, A. K. (1999). Supertagging: an approach to almost parsing. *Comput. Linguist.*, 25(2):237–265.

Bohnet, B., McDonald, R. T., Simões, G., Andor, D., Pitler, E., and Maynez, J. (2018). Morphosyntactic tagging with a meta-bilstm model over context sensitive token encodings. *CoRR*, abs/1805.08237.

Clark, S., Hockenmaier, J., and Steedman, M. (2002). Building deep dependency structures with a wide-coverage ccg parser.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2019). Bert: Pre-training of deep bidirectional transformers for language understanding.

Good, P. (2000). *Permutation Tests*. Springer New York.

Gu, J., Wang, C., and Zhao, J. (2019). Levenshtein transformer.

Hockenmaier, J. and Steedman, M. (2007). Ccgbank: A corpus of ccg derivations and dependency structures extracted from the penn treebank. *Computational Linguistics*, 33:355–396.

Honnibal, M., Curran, J. R., and Bos, J. (2010). Rebanking CCGbank for improved NP interpretation. In Hajič, J., Carberry, S., Clark, S., and Nivre, J., editors, *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, pages 207–215, Uppsala, Sweden. Association for Computational Linguistics.

Ke, W., Tian, Z., Liu, Q., Wang, P., Gao, J., and Qi, R. (2023). Towards incremental ner data augmentation via syntactic-aware insertion transformer. pages 5104–5112. Main Track.

Kogkalidis, K., Moortgat, M., and Deoskar, T. (2019). Constructive type-logical

supertagging with self-attention networks. *ArXiv*, abs/1905.13418.

Lewis, M. and Steedman, M. (2014). A* CCG parsing with a supertag-factored model. In Moschitti, A., Pang, B., and Daelemans, W., editors, *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 990–1000, Doha, Qatar. Association for Computational Linguistics.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. (2019). Roberta: A robustly optimized bert pretraining approach.

Lu, S., Meng, T., and Peng, N. (2021). Insnet: An efficient, flexible, and performant insertion-based text generation model. In *Neural Information Processing Systems*.

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

Prange, J., Schneider, N., and Srikumar, V. (2020). Supertagging the long tail with tree-structured decoding of complex categories. *CoRR*, abs/2012.01285.

Radford, A. and Narasimhan, K. (2018). Improving language understanding by generative pre-training.

Rombach, R., Blattmann, A., Lorenz, D., Esser, P., and Ommer, B. (2022). High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 10684–10695.

Ruis, L., Stern, M., Proskurnia, J., and Chan, W. (2020). Insertion-deletion transformer.

Sanh, V., Debut, L., Chaumond, J., and Wolf, T. (2020). Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter.

Smith, L. N. and Topin, N. (2018). Super-convergence: Very fast training of neural networks using large learning rates.

Stern, M., Andreas, J., and Klein, D. (2017). A minimal span-based neural constituency parser. In Barzilay, R. and Kan, M.-Y., editors, *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 818–827, Vancouver, Canada. Association for Computational Linguistics.

Stern, M., Chan, W., Kiros, J., and Uszkoreit, J. (2019). Insertion transformer: Flexi-

ble sequence generation via insertion operations. *CoRR*, abs/1902.03249.

Tian, Y., Song, Y., and Xia, F. (2020). Supertagging Combinatory Categorial Grammar with attentive graph convolutional networks. In Webber, B., Cohn, T., He, Y., and Liu, Y., editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6037–6044, Online. Association for Computational Linguistics.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need. *CoRR*, abs/1706.03762.

Williams, R. J. and Zipser, D. (1989). A Learning Algorithm for Continually Running Fully Recurrent Neural Networks. *Neural Computation*, 1(2):270–280.

Zhang, Z., Zhang, Y., and Dolan, B. (2023). Towards more efficient insertion transformer with fractional positional encoding. pages 1564–1572.

Zhu, Q., Khan, H., Soltan, S., Rawls, S., and Hamza, W. (2020). Don't parse, insert: Multilingual semantic parsing with insertion based decoding.