Structured Neural Models and Structured Decoding for Natural Language Processing

Mitchell Stern

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2020-221
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-221.html

December 18, 2020

Structured Neural Models and Structured Decoding for Natural Language Processing

by

Mitchell Stern

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Dan Klein, Co-chair
Professor Michael Jordan, Co-chair
Assistant Professor David Bamman

Fall 2020

Structured Neural Models and Structured Decoding for Natural Language Processing

Abstract

Structured Neural Models and Structured Decoding for Natural Language Processing

by

Mitchell Stern

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Dan Klein, Co-chair

Professor Michael Jordan, Co-chair

Neural sequence models have been applied with great success to a variety of tasks in natural language processing in recent years. However, because they generate their outputs one token at a time from left to right, they are not immediately suitable for domains where non-trivial output constraints must be satisfied for well-formedness, or where parallel decoding may be desirable for higher throughput. In this dissertation, we explore how we can overcome these limitations through the use of more highly structured models and more flexibly structured decoding algorithms.

On the modeling side, we first introduce a span-based neural model for constituency parsing that permits efficient, globally optimal decoding over the space of parse trees using a chart-based dynamic program. We then present the Abstract Syntax Network, a tree-structured neural model for code generation whose scoring modules are composed together in a way that mirrors the syntactic structure of the program being produced. Next, turning to more flexible decoding algorithms for sequences, we demonstrate how the Transformer sequence model can be extended to accommodate blockwise parallel decoding for significant improvements in decoding speed without compromising accuracy. Finally, we present the Insertion Transformer, an insertion-based sequence model that enables out-of-order generation and logarithmic-time parallel decoding.

This dissertation is dedicated to my advisors, Dan Klein and Michael Jordan, for their guidance and support over the course of my PhD, and to my colleagues at Berkeley and Google, without whom this work would not have been possible.

Contents

Chapter 1

Introduction

Neural sequence models have become a critical component in a wide variety of natural language processing systems in recent years, with successful applications in language modeling, machine translation, dialogue, speech recognition, and other areas. However, while they provide a uniform interface to many tasks in the form of next-token prediction, their unstructured and monotonic nature makes them ill-suited for domains with non-trivial output constraints or alternate decoding requirements. For example, the designer of a syntactic parser must ensure that their system produces valid trees for all inputs, and a company building a low-latency machine translation system may be willing to expend greater computational resources per input in exchange for faster generation speed. Ordinary sequence models cannot accommodate either of these use cases by default. The goal of this dissertation is to explore ways in which we can overcome such limitations through the use of more highly structured neural models (Chapters 2-3) and more flexibly structured decoding algorithms (Chapters 4-5).

In Chapter 2, we introduce a simple neural model for constituency parsing with a span-oriented tree decoder in which all spans and labels are scored independently. This model is not only compatible with a chart-based dynamic program that allows us to recover the globally optimal parse tree, but also admits a novel greedy top-down inference procedure based on recursive input partitioning that yields similar performance with better inference complexity. Our system achieved state-of-the-art performance on the Penn Treebank at the time of publication and continues to serve as a foundation for subsequent research in the parsing community, demonstrating the strength of a structured neural approach.

In Chapter 3, we introduce the Abstract Syntax Network, a model for code generation and semantic parsing that ensures well-formed outputs by directly producing abstract syntax trees in the appropriate domain. Our model contains separate neural scoring modules for each of the target domain's syntactic production rules. These modules are dynamically assembled during inference in a manner that mirrors the structure of the program being predicted. By maintaining a tight coupling between model and output structure, our syntactically grounded approach attains state-of-the-art accuracy on the Hearthstone language-to-code dataset, substantially outperforming the previous best sequence-based model. It also performs competitively on three standard semantic parsing datasets without any task-specific

engineering.

In Chapter 4, we propose a blockwise parallel decoding scheme compatible with a modified Transformer sequence model that can significantly improve generation speed without compromising accuracy. We first augment the Transformer model with additional prediction heads so that it can make parallel predictions for the next several tokens. Then, by making use of the model's ability to process output sequences in parallel, we are able to efficiently determine the longest prefix of these predictions that would match the output of a greedy decode. These steps can be dovetailed for further efficiency improvements. Using this strategy, we achieve wall-clock speedups of up to 4x over standard greedy decoding on machine translation and image super-resolution tasks with minimal effects on output quality.

In Chapter 5, we introduce the Insertion Transformer, a sequence model in which tokens can be inserted anywhere in the output rather than just at the end. Such a model enables out-of-order generation or completion of partially specified sequences, and can be trained to follow an arbitrary generation order by setting the loss appropriately. It also allows for a parallel decoding scheme in which tokens are inserted into multiple slots simultaneously. When we train the model to follow a balanced binary tree ordering, we find that it can achieve logarithmic-time parallel decoding while matching the accuracy of a standard Transformer on a benchmark English-German translation task.

Chapter 2

A Span-Based Model for Constituency Parsing

In this chapter, we present a minimal neural model for constituency parsing based on independent scoring of labels and spans. We show that this model is not only compatible with classical dynamic programming techniques, but also admits a novel greedy top-down inference algorithm based on recursive partitioning of the input. We demonstrate empirically that both prediction schemes are competitive with recent work, and when combined with basic extensions to the scoring model are capable of achieving state-of-the-art single-model performance on the Penn Treebank (91.79 F1) and strong performance on the French Treebank (82.23 F1).[1]

2.1 Introduction

This chapter presents a minimal but surprisingly effective span-based neural model for constituency parsing. Recent years have seen a great deal of interest in parsing architectures that make use of recurrent neural network (RNN) representations of input sentences (Vinyals et al., 2015b). Despite evidence that linear RNN decoders are implicitly able to respect some nontrivial well-formedness constraints on structured outputs (Graves, 2013), researchers have consistently found that the best performance is achieved by systems that explicitly require the decoder to generate well-formed tree structures (Chen and Manning, 2014).

There are two general approaches to ensuring this structural consistency. The most common is to encode the output as a sequence of operations within a transition system which constructs trees incrementally. This transforms the parsing problem back into a sequence-to-sequence problem, while making it easy to force the decoder to take only actions guaranteed to produce well-formed outputs. However, transition-based models do not admit fast dynamic programs and require careful feature engineering to support exact search-based

[1]The material in this chapter is adapted from *A Minimal Span-Based Neural Constituency Parser* (Stern, Andreas, and Klein, 2017).

inference (Thang et al., 2015). Moreover, models with recurrent state require complex training procedures to benefit from anything other than greedy decoding (Wiseman and Rush, 2016).

An alternative line of work focuses on *chart parsers*, which use log-linear or neural scoring potentials to parameterize a tree-structured dynamic program for maximization or marginalization (Finkel et al., 2008; Durrett and Klein, 2015). These models enjoy a number of appealing formal properties, including support for exact inference and structured loss functions. However, previous chart-based approaches have required considerable scaffolding beyond a simple well-formedness potential, e.g. pre-specification of a complete context-free grammar for generating output structures and initial pruning of the output space with a weaker model (Hall et al., 2014). Additionally, we are unaware of any recent chart-based models that achieve results competitive with the best transition-based models.

In this work, we present an extremely simple chart-based neural parser based on independent scoring of labels and spans, and show how this model can be adapted to support a greedy top-down decoding procedure. Our goal is to preserve the basic algorithmic properties of span-oriented (rather than transition-oriented) parse representations, while exploring the extent to which neural representational machinery can replace the additional structure required by existing chart parsers. On the Penn Treebank, our approach outperforms a number of recent models for chart-based and transition-based parsing—including the state-of-the-art models of Cross and Huang (2016) and Liu and Zhang (2016)—achieving an F1 score of 91.79. We additionally obtain a strong F1 score of 82.23 on the French Treebank.

2.2 Model

A constituency tree can be regarded as a collection of labeled spans over a sentence. Taking this view as a guiding principle, we propose a model with two components, one which assigns scores to span labels and one which assigns scores directly to span existence. The former is used to determine the labeling of the output, and the latter provides its structure.

At the core of both of these components is the issue of span representation. Given that a span's correct label and its quality as a constituent depend heavily on the context in which it appears, we naturally turn to recurrent neural networks as a starting point, since they have previously been shown to capture contextual information suitable for use in a variety of natural language applications (Bahdanau et al., 2014; Wang et al., 2015)

In particular, we run a bidirectional LSTM over the input to obtain context-sensitive forward and backward encodings for each position i, denoted by $\mathbf{f}_i$ and $\mathbf{b}_i$, respectively. Our representation of the span (i, j) is then the concatenatation the vector differences $\mathbf{f}_j - \mathbf{f}_i$ and $\mathbf{b}_i - \mathbf{b}_j$. This corresponds to a bidirectional version of the LSTM-Minus features first proposed by Wang and Chang (2016).

On top of this base, our label and span scoring functions are implemented as one-layer feedforward networks, taking as input the concatenated span difference and producing as output either a vector of label scores or a single span score. More formally, letting $\mathbf{s}_{ij}$ denote

the vector representation of span (i, j), we define

$$s_{\text{labels}}(i, j) = \mathbf{V}_\ell g(\mathbf{W}_\ell \mathbf{s}_{ij} + \mathbf{b}_\ell),$$
$$s_{\text{span}}(i, j) = \mathbf{v}_s^\top g(\mathbf{W}_s \mathbf{s}_{ij} + \mathbf{b}_s),$$

where g denotes an elementwise nonlinearity. For notational convenience, we also let the score of an individual label ℓ be denoted by

$$s_{\text{label}}(i, j, \ell) = [s_{\text{labels}}(i, j)]_\ell,$$

where the right-hand side is the corresponding element of the label score vector.

One potential issue is the existence of unary chains, corresponding to nested labeled spans with the same endpoints. We take the common approach of treating these as additional atomic labels alongside all elementary nonterminals. To accommodate n-ary trees, our inventory additionally includes a special empty label $\varnothing$ used for spans that are not themselves full constituents but arise during the course of implicit binarization.

Our model shares several features in common with that of Cross and Huang (2016). In particular, our representation of spans and the form of our label scoring function were directly inspired by their work, as were our handling of unary chains and our use of an empty label. However, our approach differs in its treatment of structural decisions, and consequently, the inference algorithms we describe below diverge significantly from their transition-based framework.

2.3 Chart Parsing

Our basic model is compatible with traditional chart-based dynamic programming. Representing a constituency tree T by its labeled spans,

$$T := \{(\ell_t, (i_t, j_t)) : t = 1, \ldots, |T|\},$$

we define the score of a tree to be the sum of its constituent label and span scores,

$$s_{\text{tree}}(T) = \sum_{(\ell,(i,j)) \in T} \left[s_{\text{label}}(i, j, \ell) + s_{\text{span}}(i, j) \right].$$

To find the tree with the highest score for a given sentence, we use a modified CKY recursion. As with classical chart parsing, the running time of our procedure is $O(n^3)$ for a sentence of length n.

2.3.1 Dynamic Program for Inference

The base case is a span $(i, i + 1)$ consisting of a single word. Since every valid tree must include all singleton spans, possibly with empty labels, we need not consider the span score

in this case and perform only a single maximization over the choice of label:

$$s_{\text{best}}(i, i+1) = \max_{\ell} \left[s_{\text{label}}(i, i+1, \ell) \right].$$

For a general span (i, j), we define the score of the split (i, k, j) as the sum of its subspan scores,

$$s_{\text{split}}(i, k, j) = s_{\text{span}}(i, k) + s_{\text{span}}(k, j). \tag{2.1}$$

For convenience, we also define an augmented split score incorporating the scores of the corresponding subtrees,

$$\tilde{s}_{\text{split}}(i, k, j) = s_{\text{split}}(i, k, j) + s_{\text{best}}(i, k) + s_{\text{best}}(k, j).$$

Using these quantities, we can then write the general joint label and split decision as

$$s_{\text{best}}(i, j) = \max_{\ell, k} \left[s_{\text{label}}(i, j, \ell) + \tilde{s}_{\text{split}}(i, k, j) \right]. \tag{2.2}$$

Because our model assigns independent scores to labels and spans, this maximization decomposes into two disjoint subproblems, greatly reducing the size of the state space:

$$s_{\text{best}}(i, j) = \max_{\ell} \left[s_{\text{label}}(i, j, \ell) \right] + \max_{k} \left[\tilde{s}_{\text{split}}(i, k, j) \right].$$

We also note that the span scores $s_{\text{span}}(i, j)$ for each span (i, j) in the sentence can be computed once at the beginning of the procedure and shared across different subproblems with common left or right endpoints, allowing for a quadratic rather than cubic number of span score computations.

2.3.2 Margin Training

Training the model under this inference scheme is accomplished using a margin-based approach. When presented with an example sentence and its corresponding parse tree T^*, we compute the best prediction under the current model using the above dynamic program,

$$\widehat{T} = \operatorname*{argmax}_{T} \left[s_{\text{tree}}(T) \right].$$

If $\widehat{T} = T^*$, then our prediction was correct and no changes need to be made. Otherwise, we incur a hinge penalty of the form

$$\max \left(0,\ 1 - s_{\text{tree}}(T^*) + s_{\text{tree}}(\widehat{T}) \right)$$

to encourage the model to keep a margin of at least 1 between the gold tree and the best alternative. The loss to be minimized is then the sum of penalties across all training examples.

Prior work has found that it can be beneficial in a variety of applications to incorporate a structured loss function into this margin objective, replacing the hinge penalty above with one of the form

$$\max\left(0,\ \Delta(\widehat{T}, T^*) - s_{\text{tree}}(T^*) + s_{\text{tree}}(\widehat{T})\right)$$

for a loss function Δ that measures the similarity between the prediction $\widehat{T}$ and the reference T^*. Here we take Δ to be a Hamming loss on labeled spans. To incorporate this loss into the training objective, we modify the dynamic program of Section 2.3.1 to support loss-augmented decoding (Taskar et al., 2005). Since the label decisions are isolated from the structural decisions, it suffices to replace every occurrence of the label scoring function $s_{\text{label}}(i, j, \ell)$ by

$$s_{\text{label}}(i, j, \ell) + \mathbf{1}(\ell \neq \ell^*_{ij}),$$

where ℓ^*_{ij} is the label of span (i, j) in the gold tree T^*. This has the effect of requiring larger margins between the gold tree and predictions that contain more mistakes, offering a greater degree of robustness and better generalization.

2.4 Top-Down Parsing

While we have so far motivated our model from the perspective of classical chart parsing, it also allows for a novel inference algorithm in which trees are constructed greedily from the top down. At a high level, given a span, we independently assign it a label and pick a split point, then repeat this process for the left and right subspans; the recursion bottoms out with length-one spans that can no longer be split. Figure 2.1 gives an illustration of the process, which we describe in more detail below.

The base case is again a singleton span $(i, i+1)$, and follows the same form as the base case for the chart parser. In particular, we select the label $\widehat{\ell}$ that satisfies

$$\widehat{\ell} = \operatorname*{argmax}_{\ell} \left[s_{\text{label}}(i, i+1, \ell)\right],$$

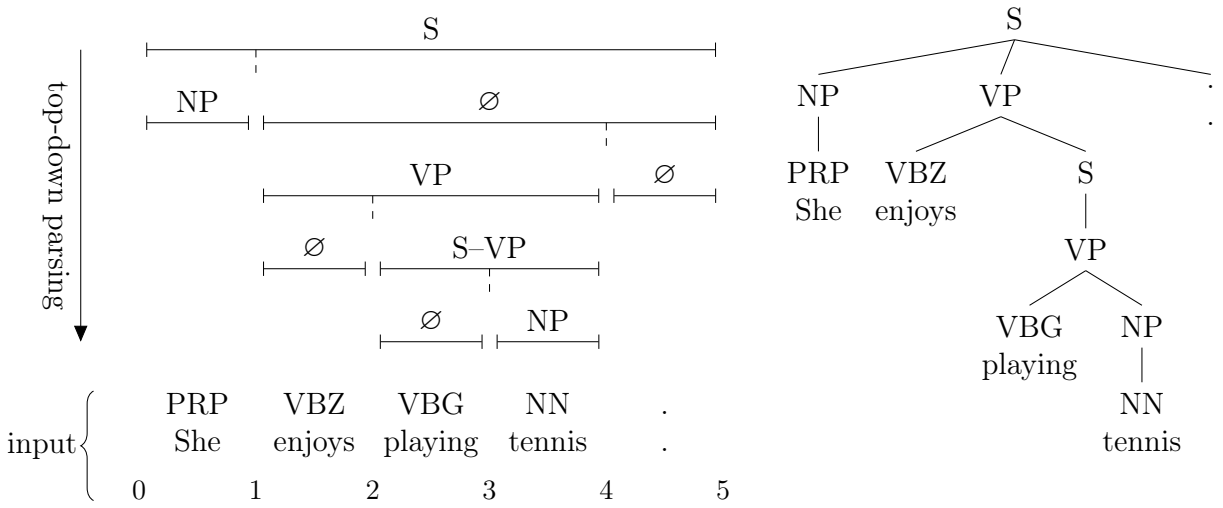
omitting span scores from consideration since singleton spans cannot be split.

To construct a tree over a general span (i, j), we aim to solve the maximization problem

$$(\widehat{\ell}, \widehat{k}) = \operatorname*{argmax}_{\ell, k} \left[s_{\text{label}}(i, j, \ell) + s_{\text{split}}(i, k, j)\right],$$

where $s_{\text{split}}(i, k, j)$ is defined as in Equation (2.1). The independence of our label and span scoring functions again yields the decomposed form

$$\begin{aligned} \widehat{\ell} &= \operatorname*{argmax}_{\ell} \left[s_{\text{label}}(i, j, \ell)\right], \\ \widehat{k} &= \operatorname*{argmax}_{k} \left[s_{\text{split}}(i, k, j)\right], \end{aligned} \tag{2.3}$$



(a) Execution of the top-down parsing algorithm. (b) Output parse tree.

Figure 2.1: An execution of our top-down parsing algorithm (a) and the resulting parse tree (b) for the sentence "She enjoys playing tennis." Part-of-speech tags, shown here together with the words, are predicted externally and are included as part of the input to our system. Beginning with the full sentence span $(0, 5)$, the label S and the split point 1 are predicted, and recursive calls are made on the child spans $(0, 1)$ and $(1, 5)$. The left child span $(0, 1)$ is assigned the label NP, and with no further splits to make, recursion terminates on this branch. The right child span $(1, 5)$ is assigned the empty label $\varnothing$, indicating that it does not represent a constituent in the tree. A split point of 4 is selected, and further recursive calls are made on the grandchild spans $(1, 4)$ and $(4, 5)$. This process of labeling and splitting continues until every branch of recursion bottoms out in singleton spans, at which point the full parse tree can be returned. Note that the unary chain S–VP is produced in a single labeling step.

leading to a significant reduction in the size of the state space.

To generate a tree for the whole sentence, we call this procedure on the full sentence span $(0, n)$ and return the result. As there are $O(n)$ spans each requiring one label evaluation and at most $n - 1$ split point evaluations, the running time of the procedure is $O(n^2)$.

The algorithm outlined here bears a strong resemblance to the chart parsing dynamic program discussed in Section 2.3, but differs in one key aspect. When performing inference from the bottom up, we have already computed the scores of all of the subtrees below the current span, and we can take this knowledge into consideration when selecting a split point. In contrast, when producing a tree from the top down, we can only select a split point based on top-level evaluations of span quality, without knowing anything about the subtrees that will be generated below them. This difference is manifested in the augmented split score $\tilde{s}_{\text{split}}$ used in the definition of s_{best} in Equation (2.2), where the scores of the subtrees associated

with a split point are included in the chart recursion but necessarily excluded from the top-down recursion.

While this apparent deficiency may be a cause for concern, we demonstrate the surprising empirical result in Section 2.6 that there is no loss in performance when moving from the globally-optimal chart parser to the greedy top-down procedure.

2.4.1 Margin Training

As with the chart parsing formulation, we also use a margin-based method for learning under the top-down model. However, rather than requiring separation between the scores of full trees, we instead enforce a local margin at every decision point.

For a span (i, j) occurring in the gold tree, let ℓ^* and k^* represent the correct label and split point, and let $\widehat{\ell}$ and $\widehat{k}$ be the predictions made by computing the maximizations in Equation (2.3). If $\widehat{\ell} \neq \ell^*$, meaning the prediction is incorrect, we incur a hinge penalty of the form

$$\max\left(0, 1 - s_{\text{label}}(i, j, \ell^*) + s_{\text{label}}(i, j, \widehat{\ell})\right).$$

Similarly, if $\widehat{k} \neq k^*$, we incur a hinge penalty of the form

$$\max\left(0, 1 - s_{\text{split}}(i, k^*, j) + s_{\text{split}}(i, \widehat{k}, j)\right).$$

To obtain the loss for a given training example, we trace out the actions corresponding to the gold tree and accumulate the above penalties over all decision points. As before, the total loss to be minimized is the sum of losses across all training examples.

Loss augmentation is also beneficial for the local decisions made by the top-down model, and can be implemented in a manner akin to the one discussed in Section 2.3.2.

2.4.2 Training with Exploration

The hinge penalties given above are only defined for spans (i, j) that appear in the example tree. The model must therefore be constrained at training time to follow decisions that exactly reproduce the gold tree, since supervision cannot be provided otherwise. As a result, the model is never exposed to its mistakes, which can lead to a lack of calibration and poor performance at test time.

To circumvent this issue, a *dynamic oracle* can be defined to inform the model about correct behavior even after it has deviated from the gold tree. Cross and Huang (2016) propose such an oracle for a related transition-based parsing system, and prove its optimality for the F1 metric on labeled spans. We adapt their result here to obtain a dynamic oracle for the present model with similar guarantees.

The oracle for labeling decisions carries over without modification: the correct label for a span is the label assigned to that span if it is part of the gold tree, or the empty label $\varnothing$ otherwise.

For split point decisions, the oracle can be broken down into two cases. If a span (i, j) appears as a constituent in the gold tree T, we let $b(i, j)$ denote the collection of its interior boundary points. For example, if the constituent over $(1, 7)$ has children spanning $(1, 3)$, $(3, 6)$, and $(6, 7)$, then we would have the two interior boundary points, $b(1, 7) = \{3, 6\}$. The oracle for a span appearing in the gold tree is then precisely the output of this function. Otherwise, for spans (i, j) not corresponding to gold constituents, we must instead identify the smallest enclosing gold constituent:

$$(i^*, j^*) = \min\{(i', j') \in T : i' \leq i < j \leq j'\},$$

where the minimum is taken with respect to the partial ordering induced by span length. The output of the oracle is then the set of interior boundary points of this enclosing span that also lie inside the original, $\{k \in b(i^*, j^*) : i < k < j\}$. The proof of correctness is similar to the proof in Cross and Huang (2016); we refer to the Dynamic Oracle section in their paper for a more detailed discussion.

As presented, the dynamic oracle for split point decisions returns a collection of one or more splits rather than a single correct answer. Any of these is a valid choice, with different splits corresponding to different binarizations of the original n-ary tree. We choose to use the leftmost split point for consistency in our implementation, but remark that the oracle split with the highest score could also be chosen at training time to allow for additional flexibility.

Having defined the dynamic oracle for our system, we note that training with exploration can be implemented by a single modification to the procedure described in Section 2.4.1. Local penalties are accumulated as before, but instead of tracing out the decisions required to produce the gold tree, we instead follow the decisions predicted by the model. In this way, supervision is provided at states within the prediction procedure that are more likely to arise at test time when greedy inference is performed.

2.5 Scoring and Loss Alternatives

The model presented in Section 2.2 is designed to be as simple as possible. However, there are many variations of the label and span scoring functions that could be explored; we discuss some of the options here.

2.5.1 Top-Middle-Bottom Label Scoring

Our basic model treats the empty label, elementary nonterminals, and unary chains each as atomic units, obscuring similarities between unary chains and their component nonterminals or between different unary chains with common prefixes or suffixes. To address this lack of structure, we consider an alternative scoring scheme in which labels are predicted in three parts: a top nonterminal, a middle unary chain, and a bottom nonterminal (each of which

is possibly empty).[2] This not only allows for parameter sharing across labels with common subcomponents, but also has the added benefit of allowing the model to produce novel unary chains at test time.

More precisely, we introduce the decomposition

$$s_{\text{label}}(i, j, (\ell_t, \ell_m, \ell_b)) = s_{\text{top}}(i, j, \ell_t) + s_{\text{middle}}(i, j, \ell_m) + s_{\text{bottom}}(i, j, \ell_b),$$

where s_{top}, s_{middle}, and s_{bottom} are independent one-layer feedforward networks of the same form as s_{label} that output scores for all label tops, label middle chains, and label bottoms encountered in the training corpus, respectively. The best label for a span (i, j) is then computed by solving the maximization problem

$$\max_{\ell_t, \ell_m, \ell_b} \left[s_{\text{label}}(i, j, (\ell_t, \ell_m, \ell_b)) \right],$$

which decomposes into three independent subproblems corresponding to the three label components. The final label is obtained by concatenating ℓ_t, ℓ_m, and ℓ_b, with empty components being omitted from the concatenation.

2.5.2 Left and Right Span Scoring

The basic model uses the same span scoring function s_{span} to assign a score to the left and right subspans of a given span. One simple extension is to replace this by a pair of distinct left and right feedforward networks of the same form, giving the decomposition

$$s_{\text{split}}(i, k, j) = s_{\text{left}}(i, k) + s_{\text{right}}(k, j).$$

2.5.3 Span Concatenation Scoring

Since span scores are only used to score splits in our model, we also consider directly scoring a split by feeding the concatenation of the span representations of the left and right subspans through a single feedforward network, giving

$$s_{\text{split}}(i, k, j) = \mathbf{v}_s^\top g \left(\mathbf{W}_s [\mathbf{s}_{ik}; \mathbf{s}_{kj}] + \mathbf{b}_s \right).$$

This is similar to the structural scoring function used by Cross and Huang (2016), although whereas they additionally include features for the outside spans $(0, i)$ and (j, n) in their concatenation, we omit these from our implementation, finding that they do not improve performance.

[2] In more detail, $\varnothing$ decomposes as $(\varnothing, \varnothing, \varnothing)$, X decomposes as $(X, \varnothing, \varnothing)$, X–Y decomposes as $(X, \varnothing, Y)$, and X–Z_1–$\cdots$–Z_k–Y decomposes as $(X, Z_1$–$\cdots$–$Z_k, Y)$.

2.5.4 Deep Biaffine Span Scoring

Inspired by the success of deep biaffine scoring in recent work by Dozat and Manning (2016) for dependency parsing, we also consider a split scoring function of a similar form for our model. Specifically, we let $\mathbf{h}_{ik} = f_{\text{left}}(\mathbf{s}_{ik})$ and $\mathbf{h}_{kj} = f_{\text{right}}(\mathbf{s}_{kj})$ be deep left and right span representations obtained by passing the child vectors through corresponding left and right feedforward networks. We then define the biaffine split scoring function

$$s_{\text{split}}(i, k, j) = \mathbf{h}_{ik}^{\top}\mathbf{W}_s\mathbf{h}_{kj} + \mathbf{v}_{\text{left}}^{\top}\mathbf{h}_{ik} + \mathbf{v}_{\text{right}}^{\top}\mathbf{h}_{kj},$$

which consists of the sum of a bilinear form between the two hidden representations together with two inner products.

2.5.5 Structured Label Loss

The three-way label scoring scheme described in Section 2.5.1 offers one path towards the incorporation of label structure into the model. We additionally consider a structured Hamming loss on labels. More specifically, given two labels ℓ_1 and ℓ_2 consisting of zero or more nonterminals, we define the loss as $|\ell_1 \setminus \ell_2| + |\ell_2 \setminus \ell_1|$, treating each label as a multiset of nonterminals. This structured loss can be incorporated into the training process using the methods described in Sections 2.3.2 and 2.4.1.

2.6 Experiments

We first describe the general setup used for our experiments. We use the Penn Treebank (Marcus et al., 1993) for our English experiments, with standard splits of sections 2-21 for training, section 22 for development, and section 23 for testing. We use the French Treebank from the SPMRL 2014 shared task (Seddah et al., 2014) with its provided splits for our French experiments. No token preprocessing is performed, and only a single `<UNK>` token is used for unknown words at test time. The inputs to our system are concatenations of 100-dimensional word embeddings and 50-dimensional part-of-speech embeddings. In the case of the French Treebank, we also include 50-dimensional embeddings of each morphological tag. We use automatically predicted tags for training and testing, obtaining predicted part-of-speech tags for the Penn Treebank using the Stanford tagger (Toutanova et al., 2003) with 10-way jackknifing, and using the provided predicted part-of-speech and morphological tags for the French Treebank. Words are replaced by `<UNK>` with probability $1/(1 + \text{freq}(w))$ during training, where $\text{freq}(w)$ is the frequency of w in the training data.

We use a two-layer bidirectional LSTM for our base span features. Dropout with a ratio selected from $\{0.2, 0.3, 0.4\}$ is applied to all non-recurrent connections of the LSTM, including its inputs and outputs. We tie the hidden dimension of the LSTM and all feedforward networks, selecting a size from $\{150, 200, 250\}$. All parameters (including word and tag embeddings) are randomly initialized using Glorot initialization (Glorot and Bengio, 2010), and are tuned

WSJ Dev, Atomic Labels, Basic 0-1 Label Loss

Parser	Minimal	Left-Right	Concat.	Biaffine
Chart	91.95	92.09	92.15	91.96
Top-Down	92.16	92.25	92.24	92.14

(a)

WSJ Dev, Atomic Labels, Structured Label Loss

Parser	Minimal	Left-Right	Concat.	Biaffine
Chart	91.86	92.12	92.09	91.95
Top-Down	92.12	92.31	92.26	92.20

(b)

WSJ Dev, 3-Part Labels, Basic 0-1 Label Loss

Parser	Minimal	Left-Right	Concat.	Biaffine
Chart	92.08	92.05	91.94	91.79
Top-Down	92.12	92.18	92.14	92.02

(c)

WSJ Dev, 3-Part Labels, Structured Label Loss

Parser	Minimal	Left-Right	Concat.	Biaffine
Chart	91.92	91.96	91.97	91.78
Top-Down	91.98	92.27	92.17	92.06

(d)

Table 2.1: Development F1 scores on the Penn Treebank. Each table corresponds to a particular choice of label loss (either the basic 0-1 loss or the structured Hamming label loss of Section 2.5.5) and labeling scheme (either the basic atomic scheme or the top-middle-bottom labeling scheme of Section 2.5.1). The columns within each table correspond to different split scoring schemes: basic minimal scoring, the left-right scoring of Section 2.5.2, the concatenation scoring of Section 2.5.3, and the deep biaffine scoring of Section 2.5.4.

on development set performance. We use the Adam optimizer (Kingma and Ba, 2014) with its default settings for optimization, with a batch size of 10. Our system is implemented in C++ using the DyNet neural network library (Neubig et al., 2017).

We begin by training the minimal version of our proposed chart and top-down parsers on the Penn Treebank. Out of the box, we obtain test F1 scores of 91.69 for the chart parser and 91.58 for the top-down parser. The higher of these matches the recent state-of-the-art score of 91.7 reported by Liu and Zhang (2016), demonstrating that our simple neural parsing system is already capable of achieving strong results.

Building on this, we explore the effects of different split scoring functions when using either the basic 0-1 label loss or the structured label loss discussed in Section 2.5.5. Our results are presented in Tables 2.1a and 2.1b.

We observe that regardless of the label loss, the minimal and deep biaffine split scoring schemes perform a notch below the left-right and concatenation scoring schemes. That the minimal scoring scheme performs worse than the left-right scheme is unsurprising, since the latter is a strict generalization of the former. It is evident, however, that joint scoring of left and right subspans is not required for strong results—in fact, the left-right scheme which scores child subspans in isolation slightly outperforms the concatenation scheme in all but one case, and is stronger than the deep biaffine scoring function across the board.

Comparing results across the choice of label loss, however, we find that fewer trends are apparent. The scores obtained by training with a 0-1 loss are all within 0.1 of those obtained using a structured Hamming loss, being slightly higher in four out of eight cases and slightly lower in the other half. This leads us to conclude that the more elementary approach is sufficient when selecting atomic labels from a fixed inventory.

Final Parsing Results on Penn Treebank

Parser	LR	LP	F1
Durrett and Klein (2015)	–	–	91.1
Vinyals et al. (2015b)	–	–	88.3
Dyer et al. (2016)	–	–	89.8
Cross and Huang (2016)	90.5	92.1	91.3
Liu and Zhang (2016)	91.3	92.1	91.7
Best Chart Parser	90.63	92.98	91.79
Best Top-Down Parser	90.35	93.23	91.77

Table 2.2: Comparison of final test F1 scores on the Penn Treebank. Here we only include scores from single-model parsers trained without external parse data.

Final Parsing Results on French Treebank

Parser	LR	LP	F1
Björkelund et al. (2014)	–	–	82.53
Durrett and Klein (2015)	–	–	81.25
Cross and Huang (2016)	81.90	84.77	83.11
Best Chart Parser	80.26	84.12	82.14
Best Top-Down Parser	79.60	85.05	82.23

Table 2.3: Comparison of final test F1 scores on the French Treebank.

We also perform the same set of experiments under the setting where the top-middle-bottom label scoring function described in Section 2.5.1 is used in place of an atomic label scoring function. These results are shown in Tables 2.1c and 2.1d.

A priori, we might expect that exposing additional structure would allow the model to make better predictions, but on the whole we find that the scores in this set of experiments are worse than those in the previous set. Trends similar to before hold across the different choices of scoring functions, though in this case the minimal setting has scores closer to those of the left-right setting, even exceeding its performance in the case of a chart parser with a 0-1 label loss.

Our final test results are given in Table 2.2, along with the results of other recent single-model parsers trained without external parse data. We achieve a new state-of-the-art F1 score of 91.79 with our best model. Interestingly, we observe that our parsers have a noticeably higher gap between precision and recall than do other top parsers, perhaps in part owing to the structured label loss which penalizes mismatching nonterminals more heavily than it does a nonterminal and empty label mismatch. In addition, there is little difference between the best top-down model and the best chart model, indicating that global normalization is not required to achieve strong results. Processing one sentence at a time on a `c4.4xlarge` Amazon EC2 instance, our best chart and top-down parsers operate at speeds of 20.3 sentences per second and 75.5 sentences per second, respectively, as measured on the test set.

We additionally train parsers on the French Treebank using the same settings from our English experiments, selecting the best model of each type based on development performance. We list our test results along with those of several other recent papers in Table 2.3. Although we fall short of the scores obtained by Cross and Huang (2016), we achieve competitive performance relative to the neural CRF parser of Durrett and Klein (2015).

2.7 Related Work

Many early successful approaches to constituency parsing focused on rich modeling of correlations in the *output space*, typically by engineering proabilistic context-free grammars with state spaces enriched to capture long-distance dependencies and lexical phenomena (Collins, 2003; Klein and Manning, 2003; Petrov and Klein, 2007). By contrast, the approach we have described here continues a recent line of work on direct modeling of correlations in the *input space*, by using rich feature representations to parameterize local potentials that interact with a comparatively unconstrained structured decoder. As noted in the introduction, this class of feature-based tree scoring functions can be implemented with either a linear transition system (Chen and Manning, 2014) or a global decoder (Finkel et al., 2008). Kiperwasser and Goldberg (2016) describe an approach closely related to ours but targeted at dependency formalisms, and which easily accommodates both sparse log-linear scoring models (Hall et al., 2014) and deep neural potentials (Henderson, 2004; Ballesteros et al., 2016).

The best-performing constituency parsers in the last two years have largely been transition-based rather than global; examples include the models of Dyer et al. (2016), Cross and Huang (2016) and Liu and Zhang (2016). The present work takes many of the insights developed in these models (e.g. the recurrent representation of spans (Kiperwasser and Goldberg, 2016), and the use of a dynamic oracle and exploration policy during training (Goldberg and Nivre, 2013)) and extends these insights to span-oriented models, which support a wider range of decoding procedures. Our approach differs from other recent chart-based neural models (e.g. Durrett and Klein (2015)) in the use of a recurrent input representation, structured loss function, and comparatively simple parameterization of the scoring function. In addition to the globally optimal decoding procedures for which these models were designed, and in contrast to the left-to-right decoder typically employed by transition-based models, our model admits an additional greedy top-to-bottom inference procedure.

2.8 Conclusion

We have presented a minimal span-oriented parser that uses a recurrent input representation to score trees with a sum of independent potentials on their constituent spans and labels. Our model supports both exact chart-based decoding and a novel top-down inference procedure. Both approaches achieve state-of-the-art performance on the Penn Treebank, and our best model achieves competitive performance on the French Treebank. Our experiments show

that many of the key insights from recent neural transition-based approaches to parsing can be easily ported to the chart parsing setting, resulting in a pair of extremely simple models that nonetheless achieve excellent performance.

Chapter 3

Abstract Syntax Networks for Code Generation

Tasks like code generation and semantic parsing require mapping unstructured (or partially structured) inputs to well-formed, executable outputs. In this chapter, we introduce Abstract Syntax Networks, a modeling framework for these problems. The outputs are represented as abstract syntax trees (ASTs) and constructed by a decoder with a dynamically-determined modular structure paralleling the structure of the output tree. On the benchmark Hearthstone dataset for code generation, our model obtains 79.2 BLEU and 22.7% exact match accuracy, compared to previous state-of-the-art values of 67.1 and 6.1%. Furthermore, we perform competitively on the Atis, Jobs, and Geo semantic parsing datasets with no task-specific engineering.[1]

3.1 Introduction

Tasks like semantic parsing and code generation are challenging in part because they are *structured* (the output must be well-formed) but not *synchronous* (the output structure diverges from the input structure).

Sequence-to-sequence models have proven effective for both tasks (Dong and Lapata, 2016; Ling et al., 2016), using encoder-decoder frameworks to exploit the sequential structure on both the input and output side. Yet these approaches do not account for much richer structural constraints on outputs—including well-formedness, well-typedness, and executability. The well-formedness case is of particular interest, since it can readily be enforced by representing outputs as abstract syntax trees (ASTs) (Aho et al., 2006), an approach that can be seen as a much lighter weight version of CCG-based semantic parsing (Zettlemoyer and Collins, 2005).

In this work, we introduce *Abstract Syntax Networks* (ASNs), an extension of the standard encoder-decoder framework utilizing a modular decoder whose submodels are composed to

[1]The material in this chapter is adapted from *Abstract Syntax Networks for Code Generation and Semantic Parsing* (Rabinovich, Stern, and Klein, 2017).


```
class DireWolfAlpha(MinionCard):
  def __init__(self):
    super().__init__(
      "Dire Wolf Alpha", 2, CHARACTER_CLASS.ALL,
      CARD_RARITY.COMMON, minion_type=MINION_TYPE.BEAST)
  def create_minion(self, player):
    return Minion(2, 2, auras=[
      Aura(
        ChangeAttack(1),
        MinionSelector(Adjacent()))
    ])
```

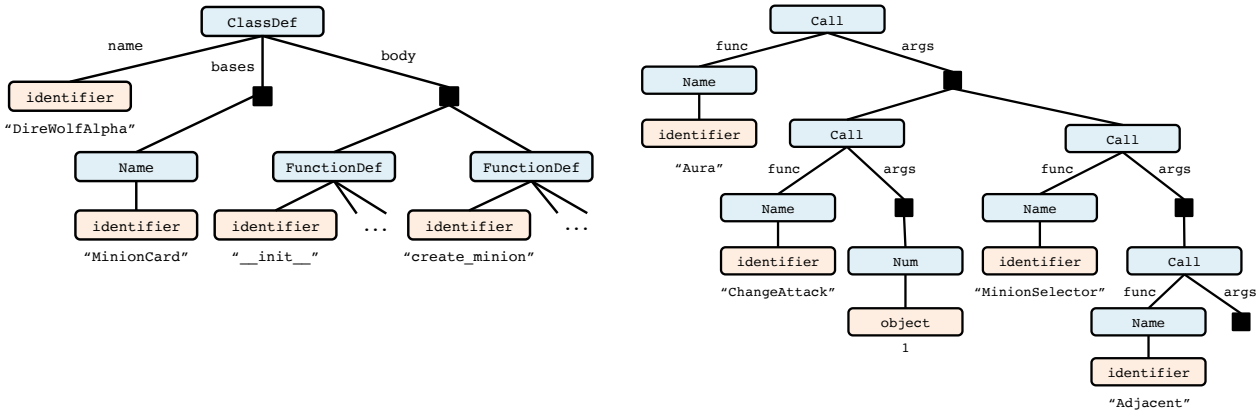
```
name: ['D', 'i', 'r', 'e', ' ', 'W', 'o', 'l', 'f', ' ', 'A', 'l', 'p', 'h', 'a']
cost: ['2']
type: ['Minion']
rarity: ['Common']
race: ['Beast']
class: ['Neutral']
description: ['Adjacent', 'minions', 'have', '+', '1', 'Attack', '.']
health: ['2']
attack: ['2']
durability: ['-1']
```

Figure 3.1: Example code and attributes for the "Dire Wolf Alpha" Hearthstone card.

```
show me the fare from ci0 to ci1

lambda $0 e
  ( exists $1 ( and ( from $1 ci0 ) ( to $1 ci1 ) ( = ( fare $1 ) $0 ) ) )
```

Figure 3.2: Example of a query and its logical form from the ATIS dataset. The `ci0` and `ci1` tokens are entity abstractions introduced in preprocessing (Dong and Lapata, 2016).

natively generate ASTs in a top-down manner. The decoding process for any given input follows a dynamically chosen mutual recursion between the modules, where the structure of the tree being produced mirrors the call graph of the recursion. We implement this process using a decoder model built of many submodels, each associated with a specific construct in the AST grammar and invoked when that construct is needed in the output tree. As is common with neural approaches to structured prediction (Chen and Manning, 2014; Vinyals



(a) The root portion of the AST.

(b) Excerpt from the same AST, corresponding to the code snippet `Aura(ChangeAttack(1),MinionSelector(Adjacent()))`.

Figure 3.3: Fragments from the abstract syntax tree corresponding to the example code in Figure 3.1. Blue boxes represent composite nodes, which expand via a constructor with a prescribed set of named children. Orange boxes represent primitive nodes, with their corresponding values written underneath. Solid black squares correspond to constructor fields with `sequential` cardinality, such as the body of a class definition (Figure 3.3a) or the arguments of a function call (Figure 3.3b).

et al., 2015b), our decoder proceeds greedily and accesses not only a fixed encoding but also an attention-based representation of the input (Bahdanau et al., 2014).

Our model significantly outperforms previous architectures for code generation and obtains competitive or state-of-the-art results on a suite of semantic parsing benchmarks. On the Hearthstone dataset for code generation, we achieve a token BLEU score of 79.2 and an exact match accuracy of 22.7%, greatly improving over the previous best results of 67.1 BLEU and 6.1% exact match (Ling et al., 2016).

The flexibility of ASNs makes them readily applicable to other tasks with minimal adaptation. We illustrate this point with a suite of semantic parsing experiments. On the Jobs dataset, we improve on previous state-of-the-art, achieving 92.9% exact match accuracy as compared to the previous record of 90.7%. Likewise, we perform competitively on the Atis and Geo datasets, matching or exceeding the exact match reported by Dong and Lapata (2016), though not quite reaching the records held by the best previous semantic parsing approaches (Wang et al., 2014).

3.1.1 Related Work

Encoder-decoder architectures, with and without attention, have been applied successfully both to sequence prediction tasks like machine translation and to tree prediction tasks like constituency parsing (Cross and Huang, 2016; Dyer et al., 2016; Vinyals et al., 2015b). In the

latter case, work has focused on making the task look like sequence-to-sequence prediction, either by flattening the output tree (Vinyals et al., 2015b) or by representing it as a sequence of construction decisions (Cross and Huang, 2016; Dyer et al., 2016). Our work differs from both in its use of a recursive top-down generation procedure.

Dong and Lapata (2016) introduced a sequence-to-sequence approach to semantic parsing, including a limited form of top-down recursion, but without the modularity or tight coupling between output grammar and model characteristic of our approach.

Neural (and probabilistic) modeling of code, including for prediction problems, has a longer history. Allamanis et al. (2015) and Maddison and Tarlow (2014) proposed modeling code with a neural language model, generating concrete syntax trees in left-first depth-first order, focusing on metrics like perplexity and applications like code snippet retrieval. More recently, Shin et al. (2017) attacked the same problem using a grammar-based variational autoencoder with top-down generation similar to ours instead. Meanwhile, a separate line of work has focused on the problem of program induction from input-output pairs (Balog et al., 2016; Liang et al., 2010; Menon et al., 2013).

The prediction framework most similar in spirit to ours is the doubly-recurrent decoder network introduced by Alvarez-Melis and Jaakkola (2017), which propagates information down the tree using a vertical LSTM and between siblings using a horizontal LSTM. Our model differs from theirs in using a separate module for each grammar construct and learning separate vertical updates for siblings when the AST labels require all siblings to be jointly present; we do, however, use a horizontal LSTM for nodes with *variable* numbers of children. The differences between our models reflect not only design decisions, but also differences in data—since ASTs have labeled nodes and labeled edges, they come with additional structure that our model exploits.

Apart from ours, the best results on the code-generation task associated with the HEARTHSTONE dataset are based on a sequence-to-sequence approach to the problem (Ling et al., 2016). Abstract Syntax Networks greatly improve on those results.

Previously, Andreas et al. (2016) introduced neural module networks (NMNs) for visual question answering, with modules corresponding to linguistic substructures within the input query. The primary purpose of the modules in NMNs is to compute deep features of images in the style of convolutional neural networks (CNN). These features are then fed into a final decision layer. In contrast to the modules we describe here, NMN modules do not make decisions about what to generate or which modules to call next, nor do they maintain recurrent state.

3.2 Data Representation

3.2.1 Abstract Syntax Trees

Our model makes use of the Abstract Syntax Description Language (ASDL) framework (Wang et al., 1997), which represents code fragments as trees with typed nodes. *Primitive types*

correspond to atomic values, like integers or identifiers. Accordingly, *primitive nodes* are annotated with a primitive type and a value of that type—for instance, in Figure 3.3a, the `identifier` node storing `"create_minion"` represents a function of the same name.

Composite types correspond to language constructs, like expressions or statements. Each type has a collection of *constructors*, each of which specifies the particular language construct a node of that type represents. Figure 3.4 shows constructors for the statement (`stmt`) and expression (`expr`) types. The associated language constructs include function and class definitions, return statements, binary operations, and function calls.

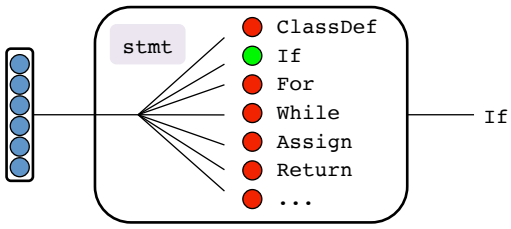
```
primitive types: identifier, object, ...

stmt
 = FunctionDef(identifier name, arg* args, stmt* body)
 | ClassDef(identifier name, expr* bases, stmt* body)
 | Return(expr? value)
 | ...

expr
 = BinOp(expr left, operator op, expr right)
 | Call(expr func, expr* args)
 | Str(string s)
 | Name(identifier id, expr_context ctx)
 | ...

...
```

Figure 3.4: A simplified fragment of the Python ASDL grammar.[2]

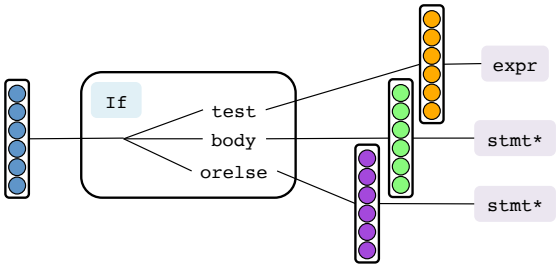
Composite types enter syntax trees via *composite nodes*, annotated with a composite type and a choice of constructor specifying how the node expands. The root node in Figure 3.3a, for example, is a composite node of type `stmt` that represents a class definition and therefore uses the `ClassDef` constructor. In Figure 3.3b, on the other hand, the root uses the `Call` constructor because it represents a function call.

Children are specified by named and typed *fields* of the constructor, which have *cardinalities* of `singular`, `optional`, or `sequential`. By default, fields have `singular` cardinality, meaning they correspond to exactly one child. For instance, the `ClassDef` constructor has a `singular` name field of type `identifier`. Fields of `optional` cardinality are associated with zero or one children, while fields of `sequential` cardinality are associated with zero or more children—these are designated using `?` and `*` suffixes in the grammar, respectively.

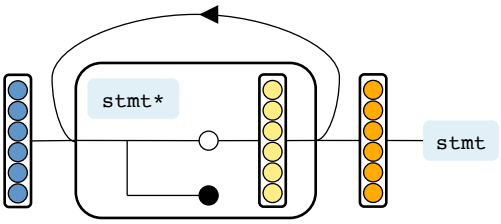
[2]The full grammar can be found online on the documentation page for the Python `ast` module: https://docs.python.org/3/library/ast.html#abstract-grammar



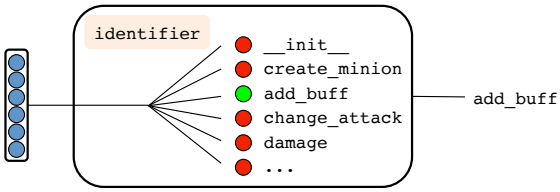
(a) A composite type module choosing a constructor for the corresponding type.



(b) A constructor module computing updated vertical LSTM states.



(c) A constructor field module (`sequential` cardinality) generating children to populate the field. At each step, the module decides whether to generate a child and continue (white circle) or stop (black circle).



(d) A primitive type module choosing a value from a closed list.

Figure 3.5: The module classes constituting our decoder. For brevity, we omit the cardinality modules for `singular` and `optional` cardinalities.

Fields of `sequential` cardinality are often used to represent statement blocks, as in the `body` field of the `ClassDef` and `FunctionDef` constructors.

The grammars needed for semantic parsing can easily be given ASDL specifications as well, using primitive types to represent variables, predicates, and atoms and composite types for standard logical building blocks like lambdas and counting (among others). Figure 3.2 shows what the resulting λ-calculus trees look like. The ASDL grammars for both λ-calculus and Prolog-style logical forms are quite compact, as Figures 3.9 and 3.10 in the appendix show.

3.2.2 Input Representation

We represent inputs as collections of named components, each of which consists of a sequence of tokens. In the case of semantic parsing, inputs have a single component containing the query sentence. In the case of Hearthstone, the card's name and description are represented as sequences of characters and tokens, respectively, while categorical attributes are represented as single-token sequences. For Hearthstone, we restrict our input and output vocabularies to values that occur more than once in the training set.

3.3 Model Architecture

Our model uses an encoder-decoder architecture with hierarchical attention. The key idea behind our approach is to structure the decoder as a collection of mutually recursive modules. The modules correspond to elements of the AST grammar and are composed together in a manner that mirrors the structure of the tree being generated. A vertical LSTM state is passed from module to module to propagate information during the decoding process.

The encoder uses bidirectional LSTMs to embed each component and a feedforward network to combine them. Component- and token-level attention is applied over the input at each step of the decoding process.

We train our model using negative log likelihood as the loss function. The likelihood encompasses terms for all generation decisions made by the decoder.

3.3.1 Encoder

Each component c of the input is encoded using a component-specific bidirectional LSTM. This results in forward and backward token encodings $(\overrightarrow{\mathbf{h}^c}, \overleftarrow{\mathbf{h}^c})$ that are later used by the attention mechanism. To obtain an encoding of the input as a whole for decoder initialization, we concatenate the final forward and backward encodings of each component into a single vector and apply a linear projection.

3.3.2 Decoder Modules

The decoder decomposes into several classes of modules, one per construct in the grammar, which we discuss in turn. Throughout, we let $\mathbf{v}$ denote the current vertical LSTM state, and use f to represent a generic feedforward neural network. LSTM updates with hidden state $\mathbf{h}$ and input $\mathbf{x}$ are notated as $\text{LSTM}(\mathbf{h}, \mathbf{x})$.

Composite type modules Each composite type `T` has a corresponding module whose role is to select among the constructors `C` for that type. As Figure 3.5a exhibits, a composite type module receives a vertical LSTM state $\mathbf{v}$ as input and applies a feedforward network $f_{\mathtt{T}}$ and a softmax output layer to choose a constructor:

$$p\left(\mathtt{C} \mid \mathtt{T}, \mathbf{v}\right) = \left[\text{softmax}\left(f_{\mathtt{T}}\left(\mathbf{v}\right)\right)\right]_{\mathtt{C}}.$$

Control is then passed to the module associated with constructor `C`.

Constructor modules Each constructor `C` has a corresponding module whose role is to compute an intermediate vertical LSTM state $\mathbf{v}_{u,\mathtt{F}}$ for each of its fields `F` whenever `C` is chosen at a composite node u.

For each field F of the constructor, an embedding $\mathbf{e}_{\mathtt{F}}$ is concatenated with an attention-based context vector $\mathbf{c}$ and fed through a feedforward neural network $f_{\mathtt{C}}$ to obtain a context-dependent field embedding:

$$\tilde{\mathbf{e}}_{\mathtt{F}} = f_{\mathtt{C}}\left(\mathbf{e}_{\mathtt{F}},\ \mathbf{c}\right).$$

An intermediate vertical state for the field F at composite node u is then computed as

$$\mathbf{v}_{u,\mathtt{F}} = \mathrm{LSTM}^{v}\left(\mathbf{v}_{u},\ \tilde{\mathbf{e}}_{\mathtt{F}}\right).$$

Figure 3.5b illustrates the process, starting with a single vertical LSTM state and ending with one updated state per field.

Constructor field modules Each field F of a constructor has a corresponding module whose role is to determine the number of children associated with that field and to propagate an updated vertical LSTM state to them. In the case of fields with `singular` cardinality, the decision and update are both vacuous, as exactly one child is always generated. Hence these modules forward the field vertical LSTM state $\mathbf{v}_{u,\mathtt{F}}$ unchanged to the child w corresponding to F:

$$\mathbf{v}_{w} = \mathbf{v}_{u,\mathtt{F}}. \tag{3.1}$$

Fields with `optional` cardinality can have either zero or one children; this choice is made using a feedforward network applied to the vertical LSTM state:

$$p(z_{\mathtt{F}} = 1 \mid \mathbf{v}_{u,\mathtt{F}}) = \mathrm{sigmoid}\left(f_{\mathtt{F}}^{\mathrm{gen}}(\mathbf{v}_{u,\mathtt{F}})\right). \tag{3.2}$$

If a child is to be generated, then as in (3.1), the state is propagated forward without modification.

In the case of `sequential` fields, a horizontal LSTM is employed for both child decisions and state updates. We refer to Figure 3.5c for an illustration of the recurrent process. After being initialized with a transformation of the vertical state, $\mathbf{s}_{\mathtt{F},0} = \mathbf{W}_{\mathtt{F}}\mathbf{v}_{u,\mathtt{F}}$, the horizontal LSTM iteratively decides whether to generate another child by applying a modified form of (3.2):

$$p\left(z_{\mathtt{F},i} = 1 \mid \mathbf{s}_{\mathtt{F},i-1},\ \mathbf{v}_{u,\mathtt{F}}\right) = \mathrm{sigmoid}\left(f_{\mathtt{F}}^{\mathrm{gen}}\left(\mathbf{s}_{\mathtt{F},i-1},\ \mathbf{v}_{u,\mathtt{F}}\right)\right).$$

If $z_{\mathtt{F},i} = 0$, generation stops and the process terminates, as represented by the solid black circle in Figure 3.5c. Otherwise, the process continues as represented by the white circle in Figure 3.5c. In that case, the horizontal state $\mathbf{s}_{u,i-1}$ is combined with the vertical state $\mathbf{v}_{u,\mathtt{F}}$ and an attention-based context vector $\mathbf{c}_{\mathtt{F},i}$ using a feedforward network $f_{\mathtt{F}}^{\mathrm{update}}$ to obtain a joint context-dependent encoding of the field F and the position i:

$$\tilde{\mathbf{e}}_{\mathtt{F},i} = f_{\mathtt{F}}^{\mathrm{update}}(\mathbf{v}_{u,\mathtt{F}},\ \mathbf{s}_{u,i-1},\ \mathbf{c}_{\mathtt{F},i}).$$

The result is used to perform a vertical LSTM update for the corresponding child w_i:

$$\mathbf{v}_{w_i} = \mathrm{LSTM}^{v}(\mathbf{v}_{u,\mathtt{F}},\ \tilde{\mathbf{e}}_{\mathtt{F},i}).$$

Finally, the horizontal LSTM state is updated using the same field-position encoding, and the process continues:

$$\mathbf{s}_{u,i} = \text{LSTM}^h(\mathbf{s}_{u,i-1},\ \tilde{\mathbf{e}}_{\mathtt{F},i}).$$

Primitive type modules Each primitive type `T` has a corresponding module whose role is to select among the values `y` within the domain of that type. Figure 3.5d presents an example of the simplest form of this selection process, where the value `y` is obtained from a closed list via a softmax layer applied to an incoming vertical LSTM state:

$$p\left(\mathtt{y} \mid \mathtt{T}, \mathbf{v}\right) = \left[\text{softmax}\left(f_{\mathtt{T}}\left(\mathbf{v}\right)\right)\right]_{\mathtt{y}}.$$

Some string-valued types are open class, however. To deal with these, we allow generation both from a closed list of previously seen values, as in Figure 3.5d, and synthesis of new values. Synthesis is delegated to a character-level LSTM language model (Bengio et al., 2003), and part of the role of the primitive module for open class types is to choose whether to synthesize a new value or not. During training, we allow the model to use the character LSTM only for unknown strings but include the log probability of that binary decision in the loss in order to ensure the model learns when to generate from the character LSTM.

3.3.3 Decoding Process

The decoding process proceeds through mutual recursion between the constituting modules, where the syntactic structure of the output tree mirrors the call graph of the generation procedure. At each step, the active decoder module either makes a generation decision, propagates state down the tree, or both.

To construct a composite node of a given type, the decoder calls the appropriate composite type module to obtain a constructor and its associated module. That module is then invoked to obtain updated vertical LSTM states for each of the constructor's fields, and the corresponding constructor field modules are invoked to advance the process to those children.

This process continues downward, stopping at each primitive node, where a value is generated but no further recursion is carried out.

3.3.4 Attention

Following standard practice for sequence-to-sequence models, we compute a raw bilinear attention score q_t^{raw} for each token t in the input using the decoder's current state $\mathbf{x}$ and the token's encoding $\mathbf{e}_t$:

$$q_t^{\text{raw}} = \mathbf{e}_t^\top \mathbf{W}\mathbf{x}.$$

The current state $\mathbf{x}$ can be either the vertical LSTM state in isolation or a concatentation of the vertical LSTM state and either a horizontal LSTM state or a character LSTM state (for string generation). Each submodule that computes attention does so using a separate matrix $\mathbf{W}$.

A separate attention score q_c^{comp} is computed for each component of the input, independent of its content:

$$q_c^{\text{comp}} = \mathbf{w}_c^\top \mathbf{x}.$$

The final token-level attention scores are the sums of the raw token-level scores and the corresponding component-level scores:

$$q_t = q_t^{\text{raw}} + q_{c(t)}^{\text{comp}},$$

where $c(t)$ denotes the component in which token t occurs. The attention weight vector $\mathbf{a}$ is then computed using a softmax:

$$\mathbf{a} = \text{softmax}\left(\mathbf{q}\right).$$

Given the weights, the attention-based context is given by:

$$\mathbf{c} = \sum_t a_t \mathbf{e}_t.$$

Certain decision points that require attention have been highlighted in the description above; however, in our final implementation we made attention available to the decoder at all decision points.

Supervised Attention In the datasets we consider, partial or total copying of input tokens into primitive nodes is quite common. Rather than providing an explicit copying mechanism (Ling et al., 2016), we instead generate alignments where possible to define a set of tokens on which the attention at a given primitive node should be concentrated.[3] If no matches are found, the corresponding set of tokens is taken to be the whole input.

The attention supervision enters the loss through a term that encourages the final attention weights to be concentrated on the specified subset. Formally, if the matched subset of component-token pairs is S, the loss term associated with the supervision would be

$$\log \sum_t \exp\left(a_t\right) - \log \sum_{t \in S} \exp\left(a_t\right), \tag{3.3}$$

where a_t is the attention weight associated with token t, and the sum in the first term ranges over all tokens in the input. The loss in (3.3) can be interpreted as the negative log probability of attending to some token in S.

3.4 Experimental Evaluation

3.4.1 Semantic Parsing

Data We use three semantic parsing datasets: Jobs, Geo, and Atis. All three consist of natural language queries paired with a logical representation of their denotations. Jobs

[3] Alignments are generated using an exact string match heuristic that also included some limited normalization, primarily splitting of special characters, undoing camel case, and lemmatization for the semantic parsing datasets.

Atis		Geo		Jobs	
System	Accuracy	System	Accuracy	System	Accuracy
ZH15	84.2	ZH15	88.9	ZH15	85.0
ZC07	84.6	KCAZ13	89.0	PEK03	88.0
WKZ14	**91.3**	WKZ14	**90.4**	LJK13	90.7
DL16	84.6	DL16	87.1	DL16	90.0
ASN	85.3	ASN	85.7	ASN	**91.4**
+ SupAtt	85.9	+ SupAtt	87.1	+ SupAtt	**92.9**

Table 3.1: Accuracies for the semantic parsing tasks. ASN denotes our Abstract Syntax Network framework. SupAtt refers to the supervised attention mentioned in Section 3.3.4.

System	Accuracy	BLEU	F1
Nearest	3.0	65.0	65.7
LPN	6.1	67.1	–
ASN	**18.2**	**77.6**	**72.4**
+ SupAtt	**22.7**	**79.2**	**75.6**

Table 3.2: Results for the Hearthstone task. SupAtt refers to the system with supervised attention mentioned in Section 3.3.4. LPN refers to the system of Ling et al. (2016). Our nearest neighbor baseline Nearest follows that of Ling et al. (2016), though it performs somewhat better; its nonzero exact match number stems from spurious repetition in the data.

consists of 640 such pairs, with Prolog-style logical representations, while Geo and Atis consist of 880 and 5,410 such pairs, respectively, with λ-calculus logical forms. We use the same training-test split as Zettlemoyer and Collins (2005) for Jobs and Geo, and the standard training-development-test split for Atis. We use the preprocessed versions of these datasets made available by Dong and Lapata (2016), where text in the input has been lowercased and stemmed using NLTK (Bird et al., 2009), and matching entities appearing in the same input-output pair have been replaced by numbered abstract identifiers of the same type.

Evaluation We compute accuracies using tree exact match for evaluation. Following the publicly released code of Dong and Lapata (2016), we canonicalize the order of the children within conjunction and disjunction nodes to avoid spurious errors, but otherwise perform no transformations before comparison.

3.4.2 Code Generation

Data We use the HEARTHSTONE dataset introduced by Ling et al. (2016), which consists of 665 cards paired with their implementations in the open-source Hearthbreaker engine.[4] Our training-development-test split is identical to that of Ling et al. (2016), with split sizes of 533, 66, and 66, respectively.

Cards contain two kinds of components: textual components that contain the card's name and a description of its function, and categorical ones that contain numerical attributes (attack, health, cost, and durability) or enumerated attributes (rarity, type, race, and class). The name of the card is represented as a sequence of characters, while its description consists of a sequence of tokens split on whitespace and punctuation. All categorical components are represented as single-token sequences.

Evaluation For direct comparison to the results of Ling et al. (2016), we evaluate our predicted code based on exact match and token-level BLEU relative to the reference implementations from the library. We additionally compute node-based precision, recall, and F1 scores for our predicted trees compared to the reference code ASTs. Formally, these scores are obtained by defining the intersection of the predicted and gold trees as their largest common tree prefix.

3.4.3 Settings

For each experiment, all feedforward and LSTM hidden dimensions are set to the same value. We select the dimension from {30, 40, 50, 60, 70} for the smaller JOBS and GEO datasets, or from {50, 75, 100, 125, 150} for the larger ATIS and HEARTHSTONE datasets. The dimensionality used for the inputs to the encoder is set to 100 in all cases. We apply dropout to the non-recurrent connections of the vertical and horizontal LSTMs, selecting the noise ratio from {0.2, 0.3, 0.4, 0.5}. All parameters are randomly initialized using Glorot initialization (Glorot and Bengio, 2010).

We perform 200 passes over the data for the JOBS and GEO experiments, or 400 passes for the ATIS and HEARTHSTONE experiments. Early stopping based on exact match is used for the semantic parsing experiments, where performance is evaluated on the training set for JOBS and GEO or on the development set for ATIS. Parameters for the HEARTHSTONE experiments are selected based on development BLEU scores. In order to promote generalization, ties are broken in all cases with a preference toward higher dropout ratios and lower dimensionalities, in that order.

Our system is implemented in Python using the DyNet neural network library (Neubig et al., 2017). We use the Adam optimizer (Kingma and Ba, 2014) with its default settings for optimization, with a batch size of 20 for the semantic parsing experiments, or a batch size of 10 for the HEARTHSTONE experiments.

[4] Available online at `https://github.com/danielyule/hearthbreaker`.


```
class IronbarkProtector(MinionCard):
  def __init__(self):
    super().__init__(
      'Ironbark Protector', 8, CHARACTER_CLASS.DRUID,
      CARD_RARITY.COMMON)
  def create_minion(self, player):
    return Minion(8, 8, taunt=True)
```

Figure 3.6: Cards with minimal descriptions exhibit a uniform structure that our system almost always predicts correctly, as in this instance.

3.4.4 Results

Our results on the semantic parsing datasets are presented in Table 3.1. Our basic system achieves a new state-of-the-art accuracy of 91.4% on the Jobs dataset, and this number improves to 92.9% when supervised attention is added. On the Atis and Geo datasets, we respectively exceed and match the results of Dong and Lapata (2016). However, these fall short of the previous best results of 91.3% and 90.4%, respectively, obtained by Wang et al. (2014). This difference may be partially attributable to the use of typing information or rich lexicons in most previous semantic parsing approaches (Zettlemoyer and Collins, 2007; Kwiatkowski et al., 2013; Wang et al., 2014; Zhao and Huang, 2015).

On the Hearthstone dataset, we improve significantly over the initial results of Ling et al. (2016) across all evaluation metrics, as shown in Table 3.2. On the more stringent exact match metric, we improve from 6.1% to 18.2%, and on token-level BLEU, we improve from 67.1 to 77.6. When supervised attention is added, we obtain an additional increase of several points on each scale, achieving peak results of 22.7% accuracy and 79.2 BLEU.

3.4.5 Error Analysis and Discussion

As the examples in Figures 3.6-3.8 show, classes in the Hearthstone dataset share a great deal of common structure. As a result, in the simplest cases, such as in Figure 3.6, generating the code is simply a matter of matching the overall structure and plugging in the correct values in the initializer and a few other places. In such cases, our system generally predicts the correct code, with the exception of instances in which strings are incorrectly transduced.


```
class ManaWyrm(MinionCard):
  def __init__(self):
    super().__init__(
      'Mana Wyrm', 1, CHARACTER_CLASS.MAGE,
      CARD_RARITY.COMMON)
  def create_minion(self, player):
    return Minion(
      1, 3, effects=[
        Effect(
          SpellCast(),
          ActionTag(
            Give(ChangeAttack(1)), SelfSelector()))
      ])
```

Figure 3.7: For many cards with moderately complex descriptions, the implementation follows a functional style that seems to suit our modeling strategy, usually leading to correct predictions.

Introducing a dedicated copying mechanism like the one used by Ling et al. (2016) or more specialized machinery for string transduction may alleviate this latter problem.

The next simplest category of card-code pairs consists of those in which the card's logic is mostly implemented via nested function calls. Figure 3.7 illustrates a typical case, in which the card's effect is triggered by a game event (a spell being cast) and both the trigger and the effect are described by arguments to an `Effect` constructor. Our system usually also performs well on instances like these, apart from idiosyncratic errors that can take the form of under- or overgeneration or simply substitution of incorrect predicates.

Cards whose code includes complex logic expressed in an imperative style, as in Figure 3.8, pose the greatest challenge for our system. Factors like variable naming, nontrivial control flow, and interleaving of code predictable from the description with code required due to the conventions of the library combine to make the code for these cards difficult to generate. In some instances (as in the figure), our system is nonetheless able to synthesize a close approximation. However, in the most complex cases, the predictions deviate significantly from the correct implementation.

In addition to the specific errors our system makes, some larger issues remain unresolved. Existing evaluation metrics only approximate the actual metric of interest: functional equivalence. Modifications of BLEU, tree F1, and exact match that canonicalize the code—for example, by anonymizing all variables—may prove more meaningful. Direct evaluation of functional equivalence is of course impossible in general (Sipser, 2006), and practically challenging even for the HEARTHSTONE dataset because it requires integrating with the game


```
class MultiShot(SpellCard):
  def __init__(self):
    super().__init__(
      'Multi-Shot', 4,
      CHARACTER_CLASS.HUNTER,
      CARD_RARITY.FREE)
  def use(self, player, game):
    super().use(player, game)
    targets = copy.copy(
        game.other_player.minions)
    for i in range(0, 2):
      target = game.random_choice(targets)
      targets.remove(target)
      target.damage(
        player.effective_spell_damage(3),
        self)
  def can_use(self, player, game):
    return (
      super().can_use(player, game) and
      (len(game.other_player.minions) >= 2))
```

```
class MultiShot(SpellCard):
  def __init__(self):
    super().__init__(
      'Multi-Shot', 4,
      CHARACTER_CLASS.HUNTER,
      CARD_RARITY.FREE)
  def use(self, player, game):
    super().use(player, game)
    minions = copy.copy(
        game.other_player.minions)
    for i in range(0, 3):
      minion = game.random_choice(minions)
      minions.remove(minion)
  def can_use(self, player, game):
    return (
      super().can_use(player, game) and
      len(game.other_player.minions) >= 3)
```

Figure 3.8: Cards with nontrivial logic expressed in an imperative style are the most challenging for our system. In this example, our prediction comes close to the gold code, but misses an important statement in addition to making a few other minor errors. *(Left)* gold code; *(right)* predicted code.

engine.

Existing work also does not attempt to enforce *semantic* coherence in the output. Long-distance semantic dependencies, between occurrences of a single variable for example, in particular are not modeled. Nor is well-typedness or executability. Overcoming these evaluation and modeling issues remains an important open problem.

3.5 Conclusion

ASNs provide a modular encoder-decoder architecture that can readily accommodate a variety of tasks with structured output spaces. They are particularly applicable in the presence of recursive decompositions, where they can provide a simple decoding process that closely parallels the inherent structure of the outputs. Our results demonstrate their promise for tree prediction tasks, and we believe their application to more general output structures is an interesting avenue for future work.

3.6 Appendix

```
expr
  = Apply(pred predicate, arg* arguments)
  | Not(expr argument)
  | Or(expr left, expr right)
  | And(expr* arguments)

arg
  = Literal(lit literal)
  | Variable(var variable)
```

Figure 3.9: The Prolog-style grammar we use for the JOBS task.

```
expr
  = Variable(var variable)
  | Entity(ent entity)
  | Number(num number)
  | Apply(pred predicate, expr* arguments)
  | Argmax(var variable, expr domain, expr body)
  | Argmin(var variable, expr domain, expr body)
  | Count(var variable, expr body)
  | Exists(var variable, expr body)
  | Lambda(var variable, var_type type, expr body)
  | Max(var variable, expr body)
  | Min(var variable, expr body)
  | Sum(var variable, expr domain, expr body)
  | The(var variable, expr body)
  | Not(expr argument)
  | And(expr* arguments)
  | Or(expr* arguments)
  | Compare(cmp_op op, expr left, expr right)

cmp_op = Equal | LessThan | GreaterThan
```

Figure 3.10: The λ-calculus grammar used by our system.

Chapter 4

Blockwise Parallel Decoding for Sequences

Deep autoregressive sequence-to-sequence models have demonstrated impressive performance across a wide variety of tasks in recent years. While common architecture classes such as recurrent, convolutional, and self-attention networks make different trade-offs between the amount of computation needed per layer and the length of the critical path at training time, generation still remains an inherently sequential process. To overcome this limitation, we propose a novel blockwise parallel decoding scheme in this chapter in which we make predictions for multiple time steps in parallel then back off to the longest prefix validated by a scoring model. This allows for substantial theoretical improvements in generation speed when applied to architectures that can process output sequences in parallel. We verify our approach empirically through a series of experiments using state-of-the-art self-attention models for machine translation and image super-resolution, achieving iteration reductions of up to 2x over a baseline greedy decoder with no loss in quality, or up to 7x in exchange for a slight decrease in performance. In terms of wall-clock time, our fastest models exhibit real-time speedups of up to 4x over standard greedy decoding.[1]

4.1 Introduction

Neural autoregressive sequence-to-sequence models have become the de facto standard for a wide variety of tasks, including machine translation, summarization, and speech synthesis (Vaswani et al., 2017; Rush et al., 2015; van den Oord et al., 2016). One common feature among recent architectures such as the Transformer and convolutional sequence-to-sequence models is an increased capacity for parallel computation, making them a better fit for today's massively parallel hardware accelerators (Vaswani et al., 2017; Gehring et al., 2017). While advances in this direction have allowed for significantly faster training, outputs are still

[1]The material in this chapter is adapted from *Blockwise Parallel Decoding for Deep Autoregressive Models* (Stern, Shazeer, and Uszkoreit, 2018).

generated one token at a time during inference, posing a substantial challenge for many practical applications (Oord et al., 2017).

In light of this limitation, a growing body of work is concerned with different approaches to accelerating generation for autoregressive models. Some general-purpose methods include probability density distillation (Oord et al., 2017), subscaling (Kalchbrenner et al., 2018), and decomposing the problem into the autoregressive generation of a short sequence of discrete latent variables followed by a parallel generation step conditioned on the discrete latents (Kaiser et al., 2018). Other techniques are more application-specific, such as the non-autoregressive Transformer for machine translation (Gu et al., 2018). While speedups of multiple orders of magnitude have been achieved on tasks with high output locality like speech synthesis, to the best of our knowledge, published improvements in machine translation either show much more modest speedups or come at a significant cost in quality.

In this work, we propose a simple algorithmic technique that exploits the ability of some architectures, such as the Transformer (Vaswani et al., 2017), to score all output positions in parallel. We train variants of the autoregressive model to make predictions for multiple future positions beyond the next position modeled by the base model. At test time, we employ these proposal models to independently and in parallel make predictions for the next several positions. We then determine the longest prefix of these predictions that would have generated under greedy decoding by scoring each position in parallel using the base model. If the length of this prefix is greater than one, we are able to skip one or more iterations of the greedy decoding loop.

In our experiments, our technique approximately doubles generation speed at no loss in quality relative to greedy decoding from an autoregressive model. Together with knowledge distillation and approximate decoding strategies, we can increase the speedup in terms of decoding iterations to up to five-fold at a modest sacrifice in quality for machine translation and seven-fold for image super-resolution. These correspond to wall-clock speedups of three-fold and four-fold, respectively.

In contrast to the other previously mentioned techniques for improving generation speed, our approach can furthermore be implemented on top of existing models with minimal modifications. Our code is publicly available in the open-source Tensor2Tensor library (Vaswani et al., 2018).

4.2 Greedy Decoding

In a sequence-to-sequence problem, we are given an input sequence $x = (x_1, \ldots, x_n)$, and we would like to predict the corresponding output sequence $y = (y_1, \ldots, y_m)$. These sequences might be source and target sentences in the case of machine translation, or low-resolution and high-resolution images in the case of image super-resolution. One common approach to this problem is to learn an autoregressive scoring model $p(y \mid x)$ that decomposes according

to the left-to-right factorization

$$\log p(y \mid x) = \sum_{j=0}^{m-1} \log p(y_{j+1} \mid y_{\leq j}, x).$$

The inference problem is then to find $y^* = \text{argmax}_y \, p(y \mid x)$.

Since the output space is exponentially large, exact search is intractable. As an approximation, we can perform greedy decoding to obtain a prediction $\hat{y}$ as follows. Starting with an empty sequence $\hat{y}$ and $j = 0$, we repeatedly extend our prediction with the highest-scoring token $\hat{y}_{j+1} = \text{argmax}_{y_{j+1}} \, p(y_{j+1} \mid \hat{y}_{\leq j}, x)$ and set $j \leftarrow j + 1$ until a termination condition is met. For language generation problems, we typically stop once a special end-of-sequence token has been generated. For image generation problems, we simply decode for a fixed number of steps.

4.3 Blockwise Parallel Decoding

Standard greedy decoding takes m steps to produce an output of length m, even for models that can efficiently score sequences using a constant number of sequential operations. While brute-force enumeration of output extensions longer than one token is intractable when the size of the vocabulary is large, we can still attempt to exploit parallelism within the model by training a set of auxiliary models to propose candidate extensions.

Let the original model be $p_1 = p$, and suppose that we have also learned a collection of auxiliary models $p_2, \ldots, p_k$ for which $p_i(y_{j+i} \mid y_{\leq j}, x)$ is the probability of the $(j+i)$th token being y_{j+i} given the first j tokens. We propose the following blockwise parallel decoding algorithm (illustrated in Figure 4.1), which is guaranteed to produce the same prediction $\hat{y}$ that would be found under greedy decoding but uses as few as m/k steps. As before, we start with an empty prediction $\hat{y}$ and set $j = 0$. Then we repeat the following three substeps until the termination condition is met:

- **Predict:** Get the block predictions $\hat{y}_{j+i} = \text{argmax}_{y_{j+i}} \, p_i(y_{j+i} \mid \hat{y}_{\leq j}, x)$ for $i = 1, \ldots, k$.
- **Verify:** Find the largest $\hat{k}$ such that $\hat{y}_{j+i} = \text{argmax}_{y_{j+i}} \, p_1(y_{j+i} \mid \hat{y}_{\leq j+i-1}, x)$ for all $1 \leq i \leq \hat{k}$. Note that $\hat{k} \geq 1$ by the definition of $\hat{y}_{j+1}$.
- **Accept:** Extend $\hat{y}$ with $\hat{y}_{j+1}, \ldots, \hat{y}_{j+\hat{k}}$ and set $j \leftarrow j + \hat{k}$.

In the **predict** substep, we find the local greedy predictions of our base scoring model p_1 and the auxiliary proposal models $p_2, \ldots, p_k$. Since these are disjoint models, each prediction can be computed in parallel, so there should be little time lost compared to a single greedy prediction.

Next, in the **verify** substep, we find the longest prefix of the proposed length-k extension that would have otherwise been produced by p_1. If the scoring model can process this sequence

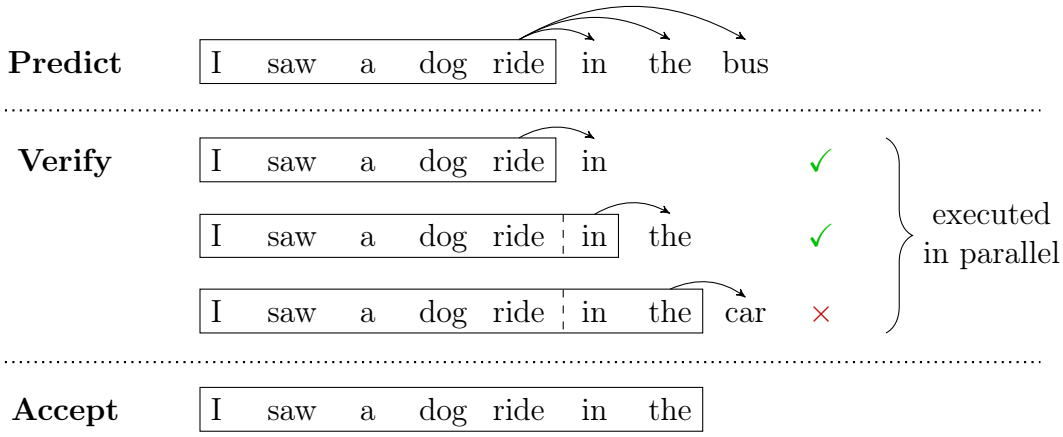


Figure 4.1: The three substeps of blockwise parallel decoding. In the **predict** substep, the greedy model and two proposal models independently and in parallel predict "in", "the", and "bus". In the **verify** substep, the greedy model scores each of the three independent predictions, conditioning on the previous independent predictions where applicable. When using a Transformer or convolutional sequence-to-sequence model, these three computations can be done in parallel. The highest-probability prediction for the third position is "car", which differs from the independently predicted "bus". In the **accept** substep, $\hat{y}$ is hence extended to include only "in" and "the" before making the next k independent predictions.

of k tokens in fewer than k steps, this substep will help save time overall provided more than one token is correct.

Lastly, in the **accept** substep, we extend our hypothesis with the verified prefix. By stopping early if the base model and the proposal models start to diverge in their predictions, we ensure that we will recover the same output that would have been produced by running greedy decoding with p_1.

The potential of this scheme to improve decoding performance hinges crucially on the ability of the base model p_1 to execute all predictions made in the **verify** substep in parallel. In our experiments we use the Transformer model (Vaswani et al., 2017). While the total number of operations performed during decoding is quadratic in the number of predictions, the number of necessarily sequential operations is constant regardless of output length. This allows us to execute the **verify** substep for a number of positions in parallel without spending additional wall-clock time.

4.4 Combined Scoring and Proposal Model

When using a Transformer for scoring, the version of our algorithm presented in Section 4.3 requires two model invocations per step: one parallel invocation of $p_1, \ldots, p_k$ in the prediction substep, and an invocation of p_1 in the verification substep. This means that even with

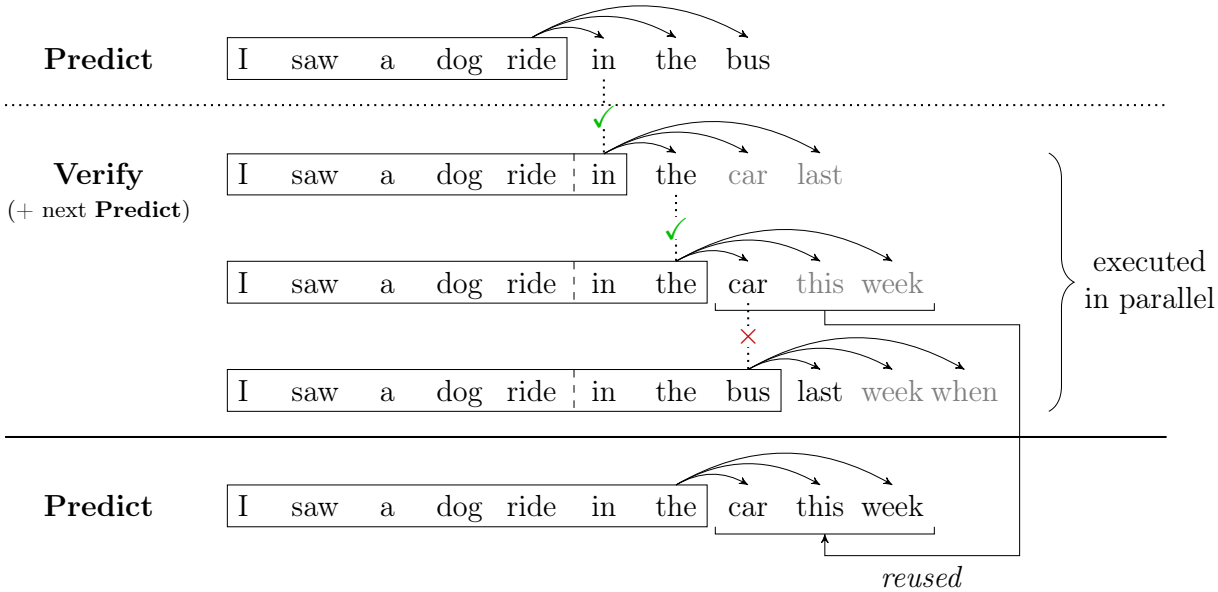


Figure 4.2: Combining the scoring and proposal models allows us to merge the previous verification substep with the next prediction substep. This makes it feasible to call the model just once per iteration rather than twice, halving the number of model invocations required for decoding.

perfect auxiliary models, we will only reduce the number of model invocations from m to $2m/k$ instead of the desired m/k.

As it turns out, we can further reduce the number of model invocations from $2m/k$ to $m/k + 1$ if we assume a combined scoring and proposal model, in which case the nth verification substep can be merged with the $(n+1)$st prediction substep.

More specifically, suppose we have a single Transformer model which during the verification substep computes $p_i(y_{j+i'+i} \mid \hat{y}_{\leq j+i'}, x)$ for all $i = 1, \ldots, k$ and $i' = 1, \ldots, k$ in a constant number of operations. This can be implemented for instance by increasing the dimensionality of the final projection layer by a factor of k and computing k separate softmaxes per position. Invoking the model after plugging in the k future predictions from the prediction substep yields the desired outputs.

Under this setup, after $\hat{k}$ has been computed during verification, we will have already computed $p_i(y_{j+\hat{k}+i} \mid y_{\leq j+\hat{k}}, x)$ for all $i = 1, \ldots, k$, which is exactly what is required for the prediction substep in the next iteration of decoding. Hence these substeps can be merged together, reducing the number of model invocations by a factor of two for all but the very first iteration.

Figure 4.2 illustrates the process. Note that while proposals have to be computed for every position during the verification substep, all predictions can still be made in parallel.

4.5 Approximate Inference

The approach to block parallel decoding we have described so far produces the same output as a standard greedy decode. By relaxing the criterion used during verification, we can allow for additional speedups at the cost of potentially deviating from the greedy output.

4.5.1 Top-k Selection

Rather than requiring that a prediction exactly matches the scoring model's prediction, we can instead ask that it lie within the top k items. To accomplish this, we replace the verification criterion with

$$\hat{y}_{j+i} \in \text{top-}k_{y_{j+i}} p_1(y_{j+i} \mid \hat{y}_{\leq j+i-1}, x).$$

4.5.2 Distance-Based Selection

In problems where the output space admits a natural distance metric d, we can replace the exact match against the highest-scoring element with an approximate match:

$$d\left(\hat{y}_{j+i},\ \operatorname*{argmax}_{y_{j+i}} p_1(y_{j+i} \mid \hat{y}_{\leq j+i-1}, x)\right) \leq \epsilon.$$

In the case of image generation, we let $d(u, v) = |u - v|$ be the absolute difference between intensities u and v within a given color channel.

4.5.3 Minimum Block Size

It is possible that the first non-greedy prediction within a given step is incorrect, in which case only a single token would be added to the hypothesis. To ensure a minimum speedup, we could require that at least $1 < \ell \leq k$ tokens be added during each decoding step. Setting $\ell = k$ would correspond to parallel decoding with blocks of fixed size k.

4.6 Implementation and Training

We implement the combined scoring and proposal model described in Section 4.4 for our experiments. Given a baseline Transformer model pre-trained for a given task, we insert a single feedforward layer with hidden size $k \times d_{\text{hidden}}$ and output size $k \times d_{\text{model}}$ between the decoder output and the final projection layer, where d_{hidden} and d_{model} are the same layer dimensions used in the rest of the network. A residual connection between the input and each of the k outputs is included. The original projection layer is identically applied to each of the k outputs to obtain the logits for $p_1, \ldots, p_k$. See Figure 4.3 for an illustration.

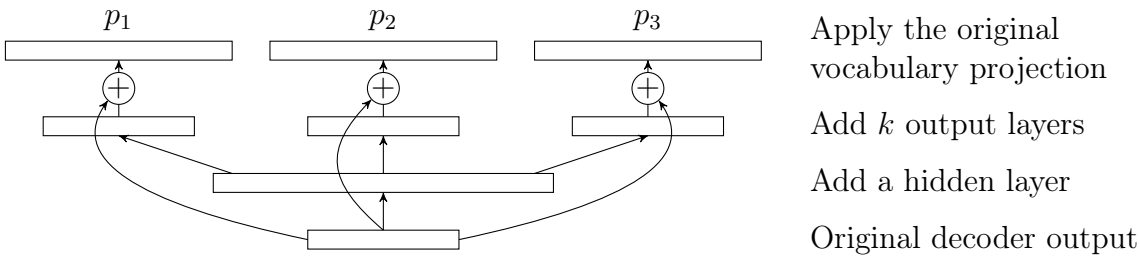


Figure 4.3: The modification we make to a Transformer to obtain a combined scoring and prediction model. To make predictions for the next k positions instead of one position, we insert a multi-output feedforward layer with residual connections after the original decoder output layer, then apply the original vocabulary projection to all outputs.

Due to memory constraints at training time, we are unable to use the mean of the k cross entropy losses corresponding to $p_1, \ldots, p_k$ as the overall loss. Instead, we select one of these sub-losses uniformly at random for each minibatch to obtain an unbiased estimate of the full loss. At inference time, all logits can be computed in parallel with marginal cost relative to the base model.

4.6.1 Fine Tuning

An important question is whether or not the original parameters of the pre-trained model should be fine tuned for the modified joint prediction task. If they are kept frozen, we ensure that the quality of the original model is retained, perhaps at the cost of less accurate future prediction. If they are fine tuned, we might improve the model's internal consistency but suffer a loss in terms of final performance. We investigate both options in our experiments.

4.6.2 Knowledge Distillation

The practice of knowledge distillation (Hinton et al., 2015; Kim and Rush, 2016), in which one trains a model on the outputs of another model, has been shown to improve performance on a variety of tasks, potentially even when teacher and student models have the same architecture and model size (Furlanello et al., 2017). We posit that sequence-level distillation could be especially useful for blockwise parallel decoding, as it tends to result in a training set with greater predictability due to consistent mode breaking from the teacher model. For our language task, we perform experiments using both the original training data and distilled training data to determine the extent of the effect. The distilled data is produced via beam decoding using a pre-trained model with the same hyperparameters as the baseline but a different random seed. The beam search hyperparameters are those from Vaswani et al. (2017).

4.7 Experiments

We implement all our experiments using the open-source Tensor2Tensor framework (Vaswani et al., 2018). Our code is publicly available within this same library.

4.7.1 Machine Translation

For our machine translation experiments, we use the WMT 2014 English-German translation dataset. Our baseline model is a Transformer trained for 1,000,000 steps on 8 P100 GPUs using the `transformer_base` hyperparameter set in Tensor2Tensor. Using greedy decoding, it attains a BLEU score of 25.56 on the newstest2013 development set.

On top of this, we train a collection of combined scoring and proposal Transformer models for various block sizes k; see Section 4.6 for implementation details. Each model is trained for an additional 1,000,000 steps on the same hardware, either on the original training data or on distilled data obtained from beam search predictions from a separate baseline run. Optimizer accumulators for running averages of first and second moments of the gradient are reset for the new training runs, as is the learning rate schedule.

We measure the BLEU score and the mean accepted block size $\hat{k}$ on the development set under a variety of settings. Results are reported in Table 4.1.[2]

k	Regular ●	Distillation ■	Fine Tuning ▲	Both ♦
1	26.00 / 1.00	26.41 / 1.00		
2	25.81 / 1.51	26.52 / 1.55	25.74 / 1.78	26.58 / 1.88
4	25.84 / 1.73	26.31 / 1.85	25.05 / 2.69	26.36 / 3.27
6	26.08 / 1.76	26.26 / 1.90	24.69 / 2.98	26.18 / 4.18
8	25.82 / 1.76	26.25 / 1.91	24.27 / 3.01	26.11 / 4.69
10	25.69 / 1.74	26.34 / 1.90	23.51 / 2.87	25.60 / 4.95

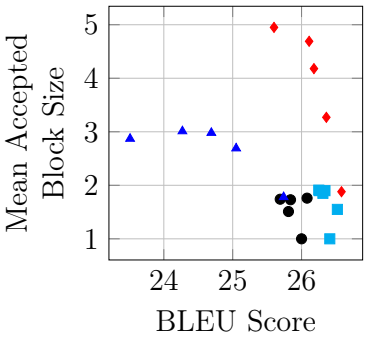


Table 4.1: Results on the newstest2013 development set for English-German translation. Each cell lists BLEU score and mean accepted block size. Larger BLEU scores indicate higher translation quality, and larger mean accepted block sizes indicate fewer decoding iterations. The data from the table is also visually depicted in a scatter plot on the right.

From these results, we make several observations. For the regular setup with gold training data and frozen baseline model parameters, the mean block size reaches a peak of 1.76, showing that speed can be improved without sacrificing model quality. When we instead use distilled data, the BLEU score at the same block size increases by 0.43 and the mean block

[2]The BLEU scores in the first two columns vary slightly with k. This is because the final decoder layer is processed by a learned transformation for all predictions $p_1, p_2, \ldots, p_k$ in our implementation rather than just $p_2, \ldots, p_k$. Using an identity transformation for p_1 instead would result in identical BLEU scores.

size reaches 1.91, showing slight improvements on both metrics. Next, comparing the results in the first two columns to their counterparts with parameter fine tuning in the last two columns, we see large increases in mean block size, albeit at the expense of some performance for larger k. The use of distilled data lessens the severity of the performance drop and allows for more accurate forward prediction, lending credence to our earlier intuition. The model with the highest mean block size of 4.95 is only 0.81 BLEU points worse than the initial model trained on distilled data.

We visualize the trade-off between BLEU score and mean block size in the plot next to Table 4.1. For both the original (●, ▲) and the distilled (■, ♦) training data, one can select a setting that optimizes for highest quality, fastest speed, or something in between. Quality degradation for larger k is much less pronounced when distilled data is used. The smooth frontier in both cases gives practitioners the option to choose a setting that best suits their needs.

We also repeat the experiments from the last column of Table 4.1 using the top-k approximate selection criterion of Section 4.5.1. For top-2 approximate decoding, we obtain the results $k = 2$: 26.49 / 1.92, $k = 4$: 26.22 / 3.47, $k = 6$: 25.90 / 4.59, $k = 8$: 25.71 / 5.34, $k = 10$: 25.04 / 5.67, demonstrating additional gains in accepted block size at the cost of further decrease in BLEU. Results for top-3 approximate decoding follow a similar trend: $k = 2$: 26.41 / 1.93, $k = 4$: 26.14 / 3.52, $k = 6$: 25.56 / 4.69, $k = 8$: 25.41 / 5.52, $k = 10$: 24.68 / 5.91. On the other hand, experiments using a minimum block size of $k = 2$ or $k = 3$ as described in Section 4.5.3 exhibit much larger drops in BLEU score with only minor improvements in mean accepted block size, suggesting that the ability to accept just one token on occasion is important and that a hard lower bound is somewhat less effective.

4.7.2 Image Super-Resolution

For our super-resolution experiments, we use the training and development data from the CelebA dataset (Liu et al., 2015). Our task is to generate a 32×32 pixel output image from an 8×8 pixel input. Our baseline model is an Image Transformer (Parmar et al., 2018) with 1D local attention trained for 1,000,000 steps on 8 P100 GPUs using the `img2img_transformer_b3` hyperparameter set. As with our machine translation experiments, we train a collection of additional models with warm-started parameters for various block sizes k, both with and without fine tuning of the base model's parameters. Here we train for an additional 250,000 steps.

We measure the mean accepted block size on the development set for each model. For the Image Transformer, an image is decomposed into a sequence of red, green, and blue intensities for each pixel in raster scan order, so each output token is an integer between 0 and 255. During inference, we either require an exact match with the greedy model or allow for an approximate match using the distance-based selection criterion from Section 4.5.2 with $\epsilon = 2$. Our results are shown in Table 4.2.

We find that exact-match decoding for the models trained with frozen base parameters is perhaps overly stringent, barely allowing for any speedup for even the largest block size.

k	Regular	Approximate	Fine Tuning	Both
1	1.00			
2	1.07	1.24	1.59	1.96
4	1.08	1.36	2.11	3.75
6	1.09	1.38	2.23	5.25
8	1.09	1.49	2.17	6.36
10	1.10	1.40	2.04	6.79

Table 4.2: Results on the CelebA development set. Each cell lists the mean accepted block size during decoding; larger values indicate fewer decoding iterations.

Relaxing the acceptance criterion helps a small amount, though the mean accepted block size remains below 1.5 in all cases. The models with fine-tuned parameters fare somewhat better when exact-match decoding is used, achieving a mean block size of slightly over 2.2 in the best case. Finally, combining approximate decoding with fine tuning yields results that are substantially better than when either modification is applied on its own. For the smaller block sizes, we see mean accepted block sizes very close to the maximum achievable bound of k. For the largest block size of 10, the mean accepted block size reaches an impressive 6.79, indicating a nearly 7x reduction in decoding iterations.

To evaluate the quality of our results, we also ran a human evaluation in which workers on Mechanical Turk were shown pairs of decoder outputs for examples from the development set and were asked to pick which one they thought was more likely to have been taken by a camera. Within each pair, one image was produced from the model trained with $k = 1$ and frozen base parameters, and one image was produced from a model trained with $k > 1$ and fine-tuned base parameters. The images within each pair were generated from the same underlying input, and were randomly permuted to avoid bias. Results are given in Table 4.3.

In all cases we obtain preference percentages close to 50%, indicating little difference in perceived quality. In fact, subjects generally showed a weak preference toward images generated using the fine-tuned models, with images coming from a fine-tuned model with approximate decoding and a medium block size of $k = 6$ obtaining the highest scores overall. We believe that the more difficult training task and approximate acceptance criterion both helped lead to outputs with slightly more noise and variation, giving them a more natural appearance when compared to the smoothed outputs that result from the baseline. See Section 4.7.4 for examples.

4.7.3 Wall-Clock Speedup

So far we have framed our results in terms of the mean accepted block size, which is reflective of the speedup achieved relative to greedy decoding in terms of number of decoding iterations. Another metric of interest is actual wall-clock speedup relative to greedy decoding, which takes into account the additional overhead required for blockwise parallel prediction. We plot

Method 1	Method 2	$1 > 2$	Confidence Interval
Fine tuning, exact, $k = 2$	Regular, exact, $k = 1$	52.8%	(50.8%, 54.9%)
Fine tuning, exact, $k = 4$	Regular, exact, $k = 1$	54.4%	(52.5%, 56.3%)
Fine tuning, exact, $k = 6$	Regular, exact, $k = 1$	53.2%	(51.3%, 55.0%)
Fine tuning, exact, $k = 8$	Regular, exact, $k = 1$	55.1%	(53.3%, 56.8%)
Fine tuning, exact, $k = 10$	Regular, exact, $k = 1$	54.5%	(53.1%, 56.0%)
Fine tuning, approximate, $k = 2$	Regular, exact, $k = 1$	50.0%	(48.4%, 51.5%)
Fine tuning, approximate, $k = 4$	Regular, exact, $k = 1$	53.3%	(51.7%, 55.0%)
Fine tuning, approximate, $k = 6$	Regular, exact, $k = 1$	56.8%	(55.4%, 58.2%)
Fine tuning, approximate, $k = 8$	Regular, exact, $k = 1$	55.2%	(53.5%, 56.7%)
Fine tuning, approximate, $k = 10$	Regular, exact, $k = 1$	50.3%	(48.9%, 51.8%)

Table 4.3: Human evaluation results on the CelebA development set. In each row, we report the percentage of votes cast in favor of the output from Method 1 over that of Method 2, along with a 90% bootstrap confidence interval.

these two quantities against each other for the best translation and super-resolution settings in Figure 4.4.

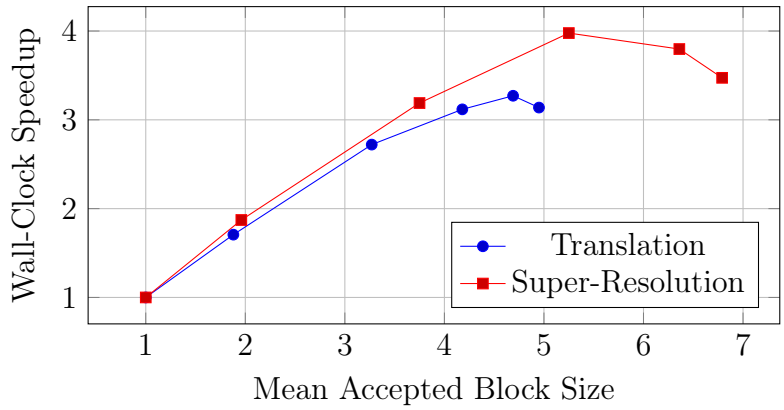


Figure 4.4: A plot of the relative wall-clock speedup achieved for various mean accepted block sizes, where the latter measure a reduction in iterations required for decoding. This data comes from the final column of Table 4.1 (translation results using fine tuning and distillation) and the final column of Table 4.2 (super-resolution results using fine tuning and approximate decoding).

For translation, the wall-clock speedup peaks at 3.3x, corresponding to the setting with $k = 8$ and mean accepted block size 4.7. For super-resolution, the wall-clock speedup reaches 4.0x, corresponding to the setting with $k = 6$ and mean accepted block size 5.3. In both cases, larger block sizes k continue to improve in terms of iteration count, but start to decline

Model	Source	BLEU	Wall-Clock Speedup
Transformer (beam size 4)	(Vaswani et al., 2017)	28.4	
Transformer (beam size 1)	(Gu et al., 2018)	22.71	
Transformer (beam size 4)	(Gu et al., 2018)	23.45	
Non-autoregressive Transformer	(Gu et al., 2018)	17.35	
Non-autoregressive Transformer (+FT)	(Gu et al., 2018)	17.69	
Non-autoregressive Transformer (+FT + NPD $s = 10$)	(Gu et al., 2018)	18.66	
Non-autoregressive Transformer (+FT + NPD $s = 100$)	(Gu et al., 2018)	19.17	
Transformer (beam size 1)	(Lee et al., 2018)	23.77	1.20x
Transformer (beam size 4)	(Lee et al., 2018)	24.57	1.00x
Iterative refinement Transformer ($i_{\text{dec}} = 1$)	(Lee et al., 2018)	13.91	11.39x
Iterative refinement Transformer ($i_{\text{dec}} = 2$)	(Lee et al., 2018)	16.95	8.77x
Iterative refinement Transformer ($i_{\text{dec}} = 5$)	(Lee et al., 2018)	20.26	3.11x
Iterative refinement Transformer ($i_{\text{dec}} = 10$)	(Lee et al., 2018)	21.61	2.01x
Iterative refinement Transformer (Adaptive)	(Lee et al., 2018)	21.54	2.39x
Latent Transformer without rescoring	(Kaiser et al., 2018)	19.8	
Latent Transformer rescoring top-10	(Kaiser et al., 2018)	21.0	
Latent Transformer rescoring top-100	(Kaiser et al., 2018)	22.5	
Transformer with distillation (greedy, $k = 1$)	This work	29.11	1.00x
Blockwise parallel decoding for Transformer ($k = 2$)	This work	28.95	1.72x
Blockwise parallel decoding for Transformer ($k = 4$)	This work	28.54	2.69x
Blockwise parallel decoding for Transformer ($k = 6$)	This work	28.11	3.10x
Blockwise parallel decoding for Transformer ($k = 8$)	This work	27.88	3.31x
Blockwise parallel decoding for Transformer ($k = 10$)	This work	27.40	3.04x

Table 4.4: A comparison of results on the newstest2014 test set for English-German translation. The reported speedups are for wall-clock time for single-sentence decoding averaged over the test set. Our approach exhibits relatively little loss in quality compared to prior work. We achieve a BLEU score within 0.29 of the original Transformer with a real-time speedup over our baseline exceeding 3x.

in terms of wall-clock improvement due to their higher computational cost.

Using our best settings for machine translation (distilled data and fine-tuned models), we also ran a test set evaluation on the newstest2014 dataset. These results along with others from related approaches are summarized in Table 4.4. Our technique exhibits much less quality degradation relative to our baseline when compared with other approaches, demonstrating its efficacy for faster decoding with minimal impact on end performance.

4.7.4 Examples

Machine translation. Here we show the generation process for a typical machine translation example. Generation occurs at the level of subwords, with underscores indicating word

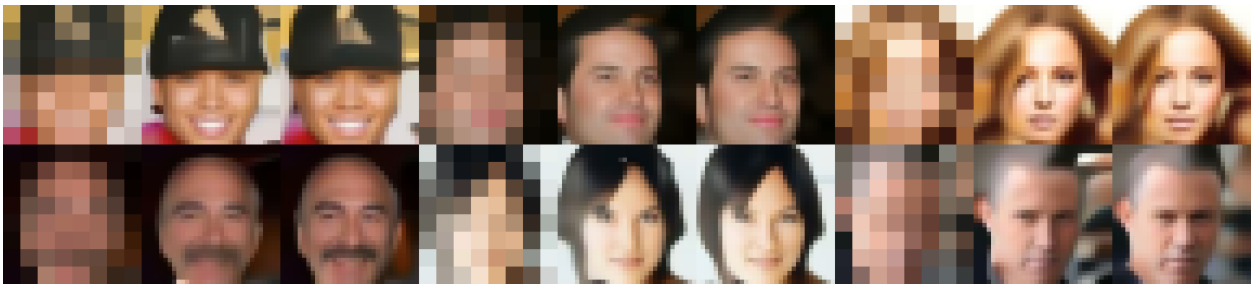
boundaries.

Input: The James Webb Space Telescope (JWST) will be launched into space on board an Ariane5 rocket by 2018 at the earliest.

Output: Das James Webb Space Teleskop (JWST) wird bis spätestens 2018 an Bord einer Ariane5-Rakete in den Weltraum gestartet.

Step 1	10 tokens	`[Das_, James_, Web, b_, Space_, Tele, sko, p_, (_, J]`
Step 2	5 tokens	`[W, ST_, ) _, wird_, bis_]`
Step 3	4 tokens	`[späte, stens_, 2018_, an_]`
Step 4	10 tokens	`[Bord_, einer_, Ari, ane, 5_, -_, Rak, ete_, in_, den_]`
Step 5	2 tokens	`[Weltraum, _]`
Step 6	3 tokens	`[gestartet_, ._, <EOS>]`

Super-resolution. Here we provide a selection of typical examples from the development set. As suggested by the human evaluations in Section 4.7.2, the blockwise parallel decodes are largely comparable in quality to the standard greedy decodes. For each triple, the left image is the low-resolution input, the middle image is the standard greedy decode, and the right image is the approximate greedy decode using the fine-tuned model with block size $k = 10$.



4.8 Conclusion

In this work, we proposed blockwise parallel decoding as a simple and generic technique for improving decoding performance in deep autoregressive models whose architectures allow for parallelization of scoring across output positions. It is comparatively straightforward to add to existing models, and we demonstrate significant improvements in decoding speed on machine translation and a conditional image generation task at no loss or only small losses in quality.

In future work we plan to investigate combinations of this technique with potentially orthogonal approaches such as those based on sequences of discrete latent variables (Kaiser et al., 2018).

Chapter 5

Insertion-Based Decoding for Sequences

In this chapter, we present the Insertion Transformer, an iterative, partially autoregressive model for sequence generation based on insertion operations. Unlike typical autoregressive models which rely on a fixed, often left-to-right ordering of the output, our approach accommodates arbitrary orderings by allowing for tokens to be inserted anywhere in the sequence during decoding. This flexibility confers a number of advantages: for instance, not only can our model be trained to follow specific orderings such as left-to-right generation or a binary tree traversal, but it can also be trained to maximize entropy over all valid insertions for robustness. In addition, our model seamlessly accommodates both fully autoregressive generation (one insertion at a time) and partially autoregressive generation (simultaneous insertions at multiple locations). We validate our approach by analyzing its performance on the WMT 2014 English-German machine translation task under various settings for training and decoding. We find that the Insertion Transformer outperforms many prior non-autoregressive approaches to translation at comparable or better levels of parallelism, and successfully recovers the performance of the original Transformer while requiring only logarithmically many iterations during decoding.[1]

5.1 Introduction

Neural sequence models (Sutskever et al., 2014; Cho et al., 2014) have been successfully applied to many applications, including machine translation (Bahdanau et al., 2014; Luong et al., 2015), speech recognition (Bahdanau et al., 2016; Chan et al., 2016), speech synthesis (Oord et al., 2016a; Wang et al., 2017), image captioning (Vinyals et al., 2015c; Xu et al., 2015) and image generation (Oord et al., 2016b,c). These models have a common theme: they rely on the chain-rule factorization and have an autoregressive left-to-right structure. This formulation bestows many advantages in both training and inference. Log-likelihood computation is tractable, allowing for efficient maximum likelihood learning. Efficient approximate inference

[1]The material in this chapter is adapted from *Insertion Transformer: Flexible Sequence Generation via Insertion Operations* (Stern, Chan, Kiros, and Uszkoreit, 2019).

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]	(⟨EOS⟩, 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]	(⟨EOS⟩, 5)

Figure 5.1: Examples demonstrating how the clause "three friends ate lunch together" can be generated using our insertion framework. On the left, a serial generation process is used in which one insertion is performed at a time. On the right, a parallel generation process is used with multiple insertions being allowed per time step. Our model can either be trained to follow specific orderings or to maximize entropy over all valid actions. Some options permit highly efficient parallel decoding, as shown in our experiments.

is also made possible through beam search decoding. However, the autoregressive framework does not easily accommodate for parallel token generation or more elaborate generation orderings (e.g., tree orders).

More recently, there has been work on non-autoregressive sequence models such as the Non-Autoregressive Transformer (NAT) (Gu et al., 2018) and the Iterative Refinement model (Lee et al., 2018). In both of these models, the decoder is seeded with an initial input derived from the source sequence, then produces the entire target sequence in parallel. Lee et al. (2018) adds an iterative refinement stage to the decoder in which a new hypothesis is produced conditioning on the input and the previous output.

While allowing for highly parallel generation, there are a few drawbacks to such approaches. The first is that the target sequence length needs to be chosen up front, preventing the output from growing dynamically as generation proceeds. This can be problematic if the chosen length is too short to accommodate the desired target, or can be wasteful if it is too long. In the case of Gu et al. (2018), there is also a strong conditional independence assumption between output tokens, limiting the model's expressive power. Lee et al. (2018) relaxes this assumption but in turn requires two separate decoders for the initial hypothesis generation and the iterative refinement stage.

In this work, we present a flexible sequence generation framework based on insertion operations. The Insertion Transformer is an iterative, partially autoregressive model which can be trained in a fully end-to-end fashion. Generation is accomplished by repeatedly making insertions into an initially-empty output sequence until a termination condition is met. Our approach bypasses the problem of needing to predict the target sequence length ahead of time by allowing the output to grow dynamically, and also permits deviation from classic left-to-right generation, allowing for more exotic orderings like balanced binary trees.

During inference, the Insertion Transformer can be used in an autoregressive manner for serial decoding, with one insertion operation being applied at a time, or in a partially autoregressive manner for parallel decoding, with insertions at multiple locations being applied

simultaneously. This allows for the target sequence to grow exponentially in length. In the case of a balanced binary tree order, our model can use as few as $\lfloor \log_2 n \rfloor + 1$ operations to produce a sequence of length n, which we find achievable in practice using an appropriately chosen loss function during training.

5.2 Sequence Generation via Insertion Operations

In this section, we describe the abstract framework used by the Insertion Transformer for sequence generation. The next section then describes the concrete model architecture we use to implement this framework.

We begin with some notation. Let x be our source canvas and y be our target canvas. In the regime of sequence modeling, a canvas is a sequence and we use the terms interchangeably. While this work focuses on sequence generation, we note that our framework can be generalized to higher-dimensional outputs (e.g., image generation).

Let $\hat{y}_t$ be the hypothesis canvas at time t. Because our framework only supports insertions and not reordering operations, it must be a subsequence of the final output hypothesis $\hat{y}$. For example, if the eventual output were $\hat{y} = [A, B, C, D, E]$, then $\hat{y}_t = [B, D]$ would be a valid intermediate canvas while $\hat{y}_t = [B, A]$ would not. We do not restrict ourselves to one insertion per step, meaning $\hat{y}_t$ could have more than t tokens.

Further, let $\mathcal{C}$ be our content vocabulary (i.e., token vocabulary for sequences). At each iteration t, the Insertion Transformer produces a joint distribution over the choice of content $c \in \mathcal{C}$ and all available insertion locations $l \in [0, |\hat{y}_t|]$ in the current hypothesis canvas $\hat{y}_t$. In other words, the Insertion Transformer models both what to insert and where to insert relative to the current canvas hypothesis $\hat{y}_t$:

$$p(c, l \mid x, \hat{y}_t) = \text{InsertionTransformer}(x, \hat{y}_t). \tag{5.1}$$

As an example, suppose our current hypothesis canvas is $\hat{y}_t = [B, D]$ and we select the insertion operation $(c = C, l = 1)$. This will result in the new hypothesis canvas $\hat{y}_{t+1} = [B, C, D]$. Also see Figure 5.1 for an example showing the full generation process for a typical English sentence.

The permitted insertion locations allow for insertions anywhere in the canvas from the leftmost slot $(l = 0)$ to the rightmost slot $(l = |\hat{y}_t|)$. Generation always begins with an empty canvas $\hat{y}_0 = []$ with just a single insertion location $l = 0$, and concludes when a special marker token is emitted. Exact details on termination handling can be found in Section 5.4.4, where we describe two variants.

5.3 Insertion Transformer Model

The concrete model we use for the Insertion Transformer is a modified version of the original Transformer (Vaswani et al., 2017), with the decoder having been altered to induce a

distribution over insertions anywhere in the current output rather than just at the end. We outline the key changes below.

Full Decoder Self-Attention. We remove the causal self-attention mask from the decoder so that all positions can attend to all other positions, as opposed to just those to the left of the current position. This allows each decision to condition on the full context of the canvas hypothesis for the current iteration.

Slot Representations via Concatenated Outputs. The standard Transformer decoder produces n vectors for a sequence of length n, one per position, with the last one being used to pick the next word. Our model instead requires $n+1$ vectors, one for each of the $n-1$ slots between words plus 2 for the beginning and end slots. We achieve this by adding special marker tokens at the beginning and end of the decoder input to extend the sequence length by two. We then take the resulting $n+2$ vectors in the final layer and concatenate each adjacent pair to obtain $n+1$ slot representations. Hence each slot is summarized by the final representations of the positions to its immediate left and right.

5.3.1 Model Variants

Beyond the required structural changes above, there are several variations of our model that we explore within our experiments.

Content-Location Distribution. We need to model the joint content-location distribution for the insertion operations. We present two approaches: the first directly models the joint distribution, the second relies on a factorization.

Let $H \in \mathbb{R}^{(T+1)\times h}$ be the matrix of slot representations, where h is the size of the hidden state and T is the length of the current partial hypothesis. Let $W \in \mathbb{R}^{h\times|\mathcal{C}|}$ be the standard softmax projection matrix from the Transformer model. We can simply use this projection matrix to compute the content-location logits, then flatten this matrix into a vector and directly take the softmax over all the content-location logits to obtain a jointly normalized distribution:

$$p(c, l) = \text{softmax}(\text{flatten}(HW)). \tag{5.2}$$

Another approach is to model the joint distribution using a conditional factorization, $p(c, l) = p(c \mid l)p(l)$. We can model the conditional content distribution as is done in the normal Transformer:

$$p(c \mid l) = \text{softmax}(h_l W), \tag{5.3}$$

where $h_l \in \mathbb{R}^h$ is the l-th row of H. In other words, we apply the softmax per-row in the matrix HW. We separately model the location distribution by taking the softmax of the dot

product of the hidden states and a learnable query vector $q \in \mathbb{R}^h$:

$$p(l) = \text{softmax}(Hq). \tag{5.4}$$

This approach requires a small number of additional parameters h compared to modeling the joint distribution directly.

Contextualized Vocabulary Bias. To increase information sharing across slots, we can perform a max pooling operation over the final decoder hidden vectors H to obtain a context vector $g \in \mathbb{R}^h$. We then project g into the vocabulary space using a learned projection matrix $V \in \mathbb{R}^{h \times |\mathcal{C}|}$ to produce a shared bias $b \in \mathbb{R}^{|\mathcal{C}|}$. We then add b to the result to the vocabulary logits at each position as an additional shared bias. We believe this may be useful in providing the model with coverage information, or in propagating count information about common words that should appear in multiple places in the output. Formally, we have

$$g = \text{maxpool}(H) \tag{5.5}$$

$$b = gV \tag{5.6}$$

$$B = \text{repmat}(b, [T+1, 1]) \tag{5.7}$$

$$p(c, l) = \text{softmax}(HW + B) \tag{5.8}$$

Mixture-of-Softmaxes Output Layer. Unlike the output vectors of a typical autoregressive model which only need to capture distributional information about the next word, the slot vectors in our model are responsible for representing entire bags of words. Moreover, depending on the order of generation, they might correspond to any contiguous span of the final output, making this a highly nontrivial modeling problem. We posit that the language modeling softmax bottleneck identified by Yang et al. (2018) poses even greater challenges for our setup. We try including the mixture-of-softmaxes layer proposed in their work as one means of addressing the issue.

5.4 Training and Loss Functions

The Insertion Transformer framework is flexible enough to accommodate arbitrary generation orders, including those which are input- and context-dependent. We discuss several order loss functions that we can optimize for.

5.4.1 Left-to-Right

As a special case, the Insertion Transformer can be trained to produce its output in a left-to-right fashion, imitating the conventional setting where this ordering is enforced by construction.

To do so, given a training example (x, y), we randomly sample a length $k \sim \text{Uniform}([0, |y|])$ and take the current hypothesis to be the left prefix $\hat{y} = (y_1, \ldots, y_k)$. We then aim to maximize the probability of the next content in the sequence $c = y_k$ in the rightmost slot location $l = k$, using the negative log-likelihood of this action as our loss to be minimized:

$$\text{loss}(x, \hat{y}) = -\log p(y_{k+1}, k \mid x, \hat{y}). \tag{5.9}$$

When the sequence is complete, i.e. $k = n$, we take y_{k+1} to be the end-of-sequence token $\langle\text{EOS}\rangle$. We note that there are several differences between our left-to-right order loss and a standard autoregressive Transformer log-probability loss. We describe them in detail in Section 5.4.5.

5.4.2 Balanced Binary Tree

A left-to-right strategy only allows for one token to be inserted at a time. On the other end of the spectrum, we can train for maximal parallelism by using a balanced binary tree ordering. The centermost token is produced first, then the center tokens of the spans on either side are produced next, and this process is recursively continued until the full sequence has been generated. As an example, for the target output $[A, B, C, D, E, F, G]$, the desired order of production would be $[] \rightarrow [D] \rightarrow [B, D, F] \rightarrow [A, B, C, D, E, F, G]$, where multiple insertions are executed in parallel. See Section 5.5 for more details on parallel decoding.

To achieve this goal, we use a soft binary tree loss encouraging the model to assign high probability to tokens near the middle of the span represented by a given slot. Partial canvas hypotheses are generated randomly so as to improve robustness and reduce exposure bias.

In more detail, given a training example (x, y), we first sample a subsequence $\hat{y}$ from the set of all subsequences of the target y. One option would be to sample uniformly from this set, which could be accomplished by iterating through each token and keeping or throwing it out with probability 1/2. Though simple, this approach would overexpose the model to partial outputs with length close to $|y|/2$ and would underexpose it to hypotheses that are nearly empty or nearly complete.

To circumvent this issue, we instead use a biased sampling procedure that gives uniform treatment to all lengths. In particular, we first sample a random length $k \sim \text{Uniform}([0, |y|])$, then sample a random subsequence of y of length k. The latter step is carried out by constructing an index list $[1, \ldots, |y|]$, shuffling it, and extracting the tokens corresponding to the first k indices in the order they appear in the target sequence y.

Once we have our randomly chosen hypothesis $\hat{y}$, it remains to compute the loss itself. For each of the $k + 1$ slots at locations $l = 0, \ldots, k$, let $(y_{i_l}, y_{i_l+1}, \ldots, y_{j_l})$ be the span of tokens from the target output yet to be produced at location l. We first define a function d_l giving the distance from the center of the span corresponding to location l:

$$d_l(i) = \left| \frac{i_l + j_l}{2} - i \right|. \tag{5.10}$$

Figure 5.2: A visualization of the weighting of the per-token negative log-likelihoods in the balanced binary tree and uniform losses. The balanced binary tree loss strongly incentivizes the generation of the center word or center words within each slot.

We use the negative distance function $-d_l$ as the reward function for a softmax weighting policy w_l (Rusu et al., 2016; Norouzi et al., 2016) (see Figure 5.2 for an illustration):

$$w_l(i) = \frac{\exp(-d_l(i)/\tau)}{\sum_{i'=i_l}^{j_l} \exp(-d_l(i')/\tau)}. \tag{5.11}$$

Next we define the slot loss at location l as a weighted sum of the negative log-likelihoods of the tokens from its corresponding span:

$$\text{slot-loss}(x, \hat{y}, l) = \sum_{i=i_l}^{j_l} -\log p(y_i, l \mid x, \hat{y}) \cdot w_l(i). \tag{5.12}$$

In other words, the loss encourages the model to prioritize the tokens closest to the center based on d_l. The temperature hyperparameter τ allows us to control the sharpness of the weight distribution, with $\tau \to 0$ approaching a peaked distribution placing all the weight on the centermost token (or centermost two tokens in the case of an even-length span), and $\tau \to \infty$ approaching a uniform distribution over all the missing content for a slot.

Finally, we define the full loss as the average of slot losses across all locations:

$$\text{loss}(x, \hat{y}) = \frac{1}{k+1} \sum_{l=0}^{k} \text{slot-loss}(x, \hat{y}, l). \tag{5.13}$$

5.4.3 Uniform

In addition to encouraging the model to follow a particular generation order, we can also train it to learn an agnostic view of the world in which it assigns equal probability mass to each correct action with no special preference. This neutral approach is useful insofar as it forces the model to be aware of all valid actions during each step of decoding, providing a rich learning signal during training and maximizing robustness.

Such an approach also bears resemblance to the principle of maximum entropy, which has successfully been employed for maximum entropy modeling across a number of domains in machine learning.

To implement this loss, we simply take $\tau \to \infty$ in the binary tree loss of the previous section, yielding a slot loss of

$$\text{slot-loss}(x, \hat{y}, l) = \frac{1}{j_l - i_l + 1} \sum_{i=i_l}^{j_l} -\log p(y_i, l \mid x, \hat{y}). \tag{5.14}$$

This is the mean of the negative log-probabilities of the correct actions for the given slot, which we note is maximized by a uniform distribution. Then as before, we take the full loss to be the mean of the slot losses.

5.4.4 Termination

We experiment with two termination conditions for the binary tree and uniform losses, slot finalization and sequence finalization, and compare their empirical performance in our experiments.

For slot finalization, when computing the slot loss for a location corresponding to an empty span in the true output, we take the target to be a single end-of-slot token. Then, all slot losses are always well-defined, and at generation time we can cease decoding when all slots predict an end-of-slot. We note for clarity that this special token appears in the vocabulary of the model but is never actually produced; see Section 5.5 for more details.

Alternatively, for sequence finalization, we leave the slot losses undefined for empty spans and exclude them from the overall loss. Once the entire sequence has been produced and all locations correspond to empty spans, we take the slot loss at every location to be the negative log-likelihood of an end-of-sequence token. This is identical to the slot finalization approach at the very end, but differs while generation is ongoing as no signal is provided for empty slots.

5.4.5 Training Differences

In a typical neural autoregressive model, there is a unidirectional flow of information in the decoder. This allows hidden states to be propagated (and reused) across time steps during the generation process, since they will remain unaltered as the hypothesis is extended rightward. In contrast, because we allow for insertions anywhere in the sequence, our approach lacks this unidirectional property and we must recompute the decoder hidden states for each position after every insertion.

This has several consequences. First, there is no state (or gradient) propagation between generation steps. Next, instead of being able to efficiently compute the losses for all generation steps of an example in one fell swoop as is usually done, we can only compute the loss for one generation step at a time under the same memory constraints. Accordingly, our batch size is effectively reduced by a factor of the average sequence length, which has the potential to affect convergence speed and/or model quality. Finally, since we need to subsample generation steps during training, as opposed to a standard Transformer that can compute all

the generation steps in a sequence for free, our gradient suffers from extra variance due to the sampling process. Under the right training conditions, however, we find these not to be major hindrances.

5.5 Inference

Recall that at each time step t, the Insertion Transformer yields a distribution $p(c, l \mid x, \hat{y}_t)$ over content c and location l given the input sequence x and current partial output sequence $\hat{y}_t$. This highly flexible model opens the door for both sequential and parallel inference techniques, which we describe in more detail below.

5.5.1 Greedy Decoding

First we have a standard greedy approach to decoding, in which the action with the highest probability across all choices of content c and location l is selected:

$$(\hat{c}_t, \hat{l}_t) = \operatorname*{argmax}_{c,l} p(c, l \mid x, \hat{y}_t). \tag{5.15}$$

Once the best decision has been identified, we insert token $\hat{c}_t$ at location $\hat{l}_t$ to obtain the next partial output $\hat{y}_{t+1}$.

For models trained towards sequence finalization, this process continues until an end-of-sequence token gets selected at any location, at which point the final output is returned.

For models trained towards slot finalization, we restrict the argmax to locations whose maximum-probability decision is not end-of-slot, and finish only when the model predicts an end-of-slot token for every location.

5.5.2 Parallel Decoding

If we train an Insertion Transformer towards slot finalization, we can also parallelize inference across slots within each time step to obtain a simple partially autoregressive decoding algorithm.

In more detail, for each location l we first compute the following maximum-probability actions:

$$\hat{c}_{l,t} = \operatorname*{argmax}_{c} p(c \mid l, x, \hat{y}_t). \tag{5.16}$$

For the version of the model whose joint distribution factors as $p(c, l) = p(l)p(c \mid l)$, the required conditional distribution $p(c \mid l)$ is already available. For the jointly normalized model, we can either obtain the conditional via renormalization as $p(c \mid l) = p(c, l)/p(l) = p(c, l)/\sum_{c'} p(c', l)$, or compute it directly by taking a softmax over the subset of logits at

Loss	Termination	BLEU (+EOS)	BLEU (+EOS) +Distillation	BLEU (+EOS) +Distillation, +Parallel
Left-to-Right	Sequence	20.92 (20.92)	23.29 (23.36)	-
Binary Tree ($\tau = 0.5$)	Slot	20.35 (21.39)	24.49 (25.55)	25.33 (25.70)
Binary Tree ($\tau = 1.0$)	Slot	21.02 (22.37)	24.36 (25.43)	25.43 (25.76)
Binary Tree ($\tau = 2.0$)	Slot	20.52 (21.95)	24.59 (25.80)	25.33 (25.80)
Uniform	Sequence	19.34 (22.64)	22.75 (25.45)	-
Uniform	Slot	18.26 (22.16)	22.39 (25.58)	24.31 (24.91)

Table 5.1: Development BLEU scores obtained via greedy decoding for our basic models trained with various loss functions and termination strategies. The +EOS numbers are the BLEU score obtained when an EOS penalty is applied during decoding to discourage premature stopping. The +Distillation numbers are for models trained with distilled data. The +Parallel numbers are obtained with parallel decoding, which is applicable to models trained with the slot finalization termination condition.

location l. In both cases, all the required conditional distributions can be computed in parallel.

Next, we filter out the locations for which the maximum-probability decision is an end-of-slot token, and for each location l that remains, insert the selected token $\hat{c}_{l,t}$ into that slot. The resulting sequence becomes the next partial output $\hat{y}_{t+1}$. This process continues until an end-of-slot token is predicted at every location.

Since the parallel decoding scheme described here allows for a token to be inserted in every slot at every time step, a sequence of length n could theoretically be generated in as few as $\lfloor \log_2 n \rfloor + 1$ steps. We find that this logarithmic complexity is attainable in practice in our experiments.

5.6 Experiments

In this section, we explore the efficacy of our approach on a real-world machine translation task, analyzing its performance under different training conditions, architectural choices, and decoding procedures. We experiment on the WMT 2014 English-German translation dataset, using newstest2013 for development and newstest2014 for testing, respectively. All our experiments are implemented in TensorFlow (Abadi et al., 2015) using the Tensor2Tensor framework (Vaswani et al., 2018). We use the default `transformer_base` hyperparameter set reported by Vaswani et al. (2018) for all hyperparameters not specific to our model. We perform no additional hyperparameter tuning. All our models are trained for 1,000,000 steps on eight P100 GPUs.

5.6.1 Baseline Results

We first train the baseline version of our model with different choices of loss functions and termination strategies. Greedy decoding results on the development set are given for each setting in the third column of Table 5.1.

We observe that the binary tree loss performs the best when standard greedy decoding is used, attaining a development BLEU score of 21.02. We also find that our left-to-right models do poorly compared to other orderings. One explanation is that the gradients of the binary tree and uniform losses are much more informative, in that they capture information on all the missing tokens, whereas left-to-right only provides information about the next one. We note that in all cases, even after 1M steps the models are still improving and do not appear to overfit.

Upon inspecting the outputs of these models, we found that some of the most common and severe mistakes were due to the model assigning high probability to the terminal token (end-of-slot or end-of-sequence, both abbreviated as EOS) too early in the decoding process, resulting in artificially short outputs. To rectify this, we introduce an **EOS penalty** hyperparameter, which is a scalar subtracted from the log-probability assigned by the model to an EOS at each location during decoding. Using a penalty of β prevents the model from selecting an EOS unless there is a difference of at least β between the log-probability of EOS and the log-probability of the second-best choice. This approach is similar the length normalization techniques used in many sequence models (Graves, 2012). We perform a sweep over the range $[0, 7]$ and report the best result for each model in parentheses. A well-chosen EOS penalty can have a sizable effect, increasing the BLEU score by nearly 4 points in some cases, and its inclusion brings the highest development score to 22.64 for the uniform loss with sequence-level finalization.

5.6.2 Knowledge Distillation

One technique shown to improve model performance on a wide variety of tasks is knowledge distillation (Hinton et al., 2015; Kim and Rush, 2016), wherein a model is trained on the outputs of another model. We use the base Transformer model from Vaswani et al. (2017) with beam search as our teacher model, and rerun a subset of the baseline experiments from the previous section on the resulting distilled data. The results are given in the fourth column of Table 5.1.

We observe improvements of 3 to 4 BLEU points across the board, showing that distillation is remarkably effective for our setting. As before, the models trained with a binary tree loss are approximately 2 BLEU points better than those trained with a uniform loss when standard decoding is performed, but the differences largely vanish when using a properly-tuned EOS penalty for each model. The best model by a small margin is the one trained with a binary tree loss with temperature $\tau = 2.0$, which achieves a 25.80 BLEU score on the development set.

Joint	Contextual	Mixture	BLEU (+EOS)
✗	✗	✗	22.39 (25.58)
✓	✗	✗	22.92 (25.14)
✗	✓	✗	23.00 (25.41)
✗	✗	✓	22.19 (25.58)
✓	✓	✗	23.22 (25.44)
✓	✗	✓	20.17 (24.19)
✗	✓	✓	23.29 (25.48)
✓	✓	✓	22.16 (25.44)

Table 5.2: Development BLEU scores obtained via greedy decoding when training models with the architectural variants discussed in Section 5.3.1. All models are trained with a uniform loss and slot finalization on distilled data.

5.6.3 Architectural Variants

Next we explore different combinations of the architectural variants described in Section 5.3.1. Using the uniform loss, slot finalization, and distillation as a neutral baseline configuration, we train each variant and decode on the development set to obtain the results given in Table 5.2.

Many of the configurations help improve performance when decoding without an EOS penalty. In particular, using joint normalization, a contextualized vocabulary bias, or both leads to improvements of 0.5-0.8 BLEU over the baseline. Once we tune the EOS penalty for each setting, however, the improvements largely disappear. The best configurations, primarily those involving mixture-of-softmaxes, are within 0.1 BLEU of the baseline. This suggests that the core architecture is already sufficiently powerful when decoding is well-tuned, but that it may be useful to consider some variations when looking at other inference settings.

5.6.4 Parallel Decoding

Thus far, all our experiments have used greedy decoding. However, as described in Section 5.5, models trained towards slot finalization also permit a parallel decoding scheme in which tokens are simultaneously inserted into every unfinished slot at each time step until no such slots remain. We decode the development set using this strategy for some of our more promising models, giving results in Table 5.3. Some example decodes are provided in Figure 5.4 for reference.

First and foremost, we observe that all scores are on par with those obtained via greedy decoding, and in some cases are even better. This demonstrates that with a proper training objective, our model can seamlessly accommodate parallel insertions with little effect on end performance. The fact that some scores are improved suggests that greedy search may suffer from issues related to local search that are circumvented by making multiple updates to the hypothesis at once. We leave this as an interesting topic for future investigation.

Model	**BLEU (+EOS)**
Binary Tree ($\tau = 0.5$)	25.33 (25.70)
Binary Tree ($\tau = 1.0$)	25.43 (25.76)
Binary Tree ($\tau = 2.0$)	25.33 (25.80)
Uniform	24.31 (24.91)
Uniform + Contextual	24.54 (24.74)
Uniform + Mixture	24.33 (25.11)
Uniform + Contextual + Mixture	24.68 (25.02)

Table 5.3: Parallel decoding results on the development set for some of our stronger models. All numbers are comparable to or even slightly better than those obtained via greedy decoding, demonstrating that our model can perform insertions in parallel with little to no cost for end performance.

In addition, we find that parallel decoding also helps close the gap between results obtained with and without an EOS penalty. We believe this may be due in part to the fact that the number of decoding iterations is reduced substantially, thereby giving fewer opportunities for the model to erroneously stop at an intermediate state.

We also perform a more careful analysis of the extent of the parallelism achieved by our highest-scoring models. In Figure 5.3, we plot the number of decoding iterations taken vs. the output length n for each development sentence. We also plot the theoretical lower bound of $\lfloor \log_2 n \rfloor + 1$ and the upper bound of n on the number of iterations. Note that greedy decoding takes n steps by definition. Our best model comes impressively close to the lower bound across the entire development set, rarely deviating by more than 1 or 2 iterations. This demonstrates that our framework is capable of producing high-quality output using a sub-linear (i.e. logarithmic) number of generation steps.

5.6.5 Test Results

Finally we report results in Table 5.4 on the newstest2014 test set using our best hyperparameters as measured on the development set. When compared with related approaches, we find that we match the high quality of models requiring a linear number of iterations while using a logarithmic number of generation steps. In practice, as shown in Figure 5.3, we rarely require more than 10 generation steps, meaning our empirical complexity even matches that of Lee et al. (2018) who use a constant 10 steps. When trained with the binary tree loss, we find that the Insertion Transformer is able to match the standard Transformer model while requiring substantially fewer generation iterations.

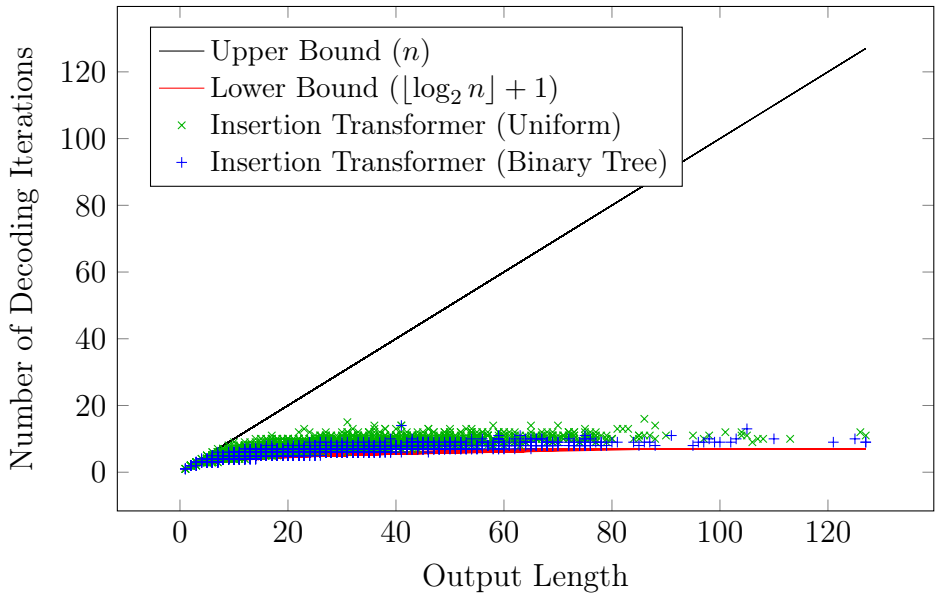


Figure 5.3: Plot showing number of decoding iterations versus output length as measured on the development set for our best models. To produce an output of length n, an insertion-based model requires at least $\lfloor \log_2 n \rfloor + 1$ iterations and at most n. While greedy decoding cannot do better than the upper bound, our parallel decoding scheme nearly achieves the lower bound in all cases.

5.7 Related Work

There has been prior work on non-left-to-right autoregressive generation. Vinyals et al. (2015a) explores the modeling of sets, where generation order does not matter. Ford et al. (2018) explores language modeling where select words (i.e., functional words) are generated first, and the rest are filled in using a two-pass process. There has also been prior work in hierarchical autoregressive image generation (Reed et al., 2017), where $\log n$ steps are required to generate n tokens. This bears some similarity to our balanced binary tree order.

Shah et al. (2018) also recently proposed generating language with a dynamic canvas. Their work can be seen as a continuous relaxation version of our model, wherein their canvas is an embedding space, while our canvas contains discrete tokens. They applied their approach to language modeling tasks, whereas we apply ours to conditional language generation in machine translation.

In addition, there has been recent work on non-autoregressive machine translation (Gu et al., 2018; Lee et al., 2018) and semi-autoregressive translation (Stern et al., 2018; Wang et al., 2018). The key difference between our work and prior work is that the Insertion Transformer framework can accommodate for a dynamically growing canvas size while still achieving sub-linear generation complexity. Other models also tend to degrade with increasing parallelism, while our model trained with the balanced binary tree loss suffers no model

Model	BLEU	Iterations
Autoregressive Left-to-Right		
Transformer (Vaswani et al., 2017)	27.3	n
Semi-Autoregressive Left-to-Right		
SAT (Wang et al., 2018)	24.83	$n/6$
Blockwise Parallel (Stern et al., 2018)	27.40	$\approx n/5$
Non-Autoregressive		
NAT (Gu et al., 2018)	17.69	1
Iterative Refinement (Lee et al., 2018)	21.61	10
Our Approach (Greedy)		
Insertion Transformer + Left-to-Right	23.94	n
Insertion Transformer + Binary Tree	27.29	n
Insertion Transformer + Uniform	27.12	n
Our Approach (Parallel)		
Insertion Transformer + Binary Tree	27.41	$\approx \log_2 n$
Insertion Transformer + Uniform	26.72	$\approx \log_2 n$

Table 5.4: BLEU scores on the newstest2014 test set for the WMT 2014 English-German translation task. Our parallel decoding strategy attains the same level of accuracy reached by linear-complexity models while using only a logarithmic number of decoding iterations.

degradation under parallel decoding.

We must also mention the concurrent work of Gu et al. (2019). They similarly use an insertion-based framework to generate sequences, but there are some differences and tradeoffs between our approaches. The main difference is that we model each successive canvas explicitly after a set of insertions, while Gu et al. (2019) model the canvas implicitly by conditioning on the insertion sequence. Consequently, Gu et al. (2019)'s approach is autoregressive, can rely on cached decoder states, and permits standard beam search, while our approach must recompute the decoder states with each iteration, but is partially autoregressive and thereby allows for parallel decoding. Gu et al. (2019) also explored tree-based orders, but while they found the syntactic tree order from a dependency parser to do slightly worse than a left-to-right baseline, we find our balanced binary tree approach to match the standard Transformer even when using parallel decoding.

Finally, we also note that Welleck et al. (2019)[1] concurrently explored generation using a tree formulation, similar to our Insertion Transformer implementation. However, they did not explore the balanced binary tree policy examined in this work, nor did they adapt their model for parallel generation, instead opting to use a serialized in-order traversal. Moreover, on a machine translation task, Welleck et al. (2019) found left-to-right generation to be superior to their learned orderings, while our balanced binary tree approach is able to match the

performance of the standard Transformer.

5.8 Conclusion

In this chapter, we presented the Insertion Transformer, a partially autoregressive model for sequence generation based on insertion operations. Our model can be trained to follow arbitrary generation orderings, such as a left-to-right order or a balanced binary tree order, or can be optimized to learn all possible orderings, making it also applicable to completion or infilling tasks. The model can be decoded serially, producing one token at a time, or it can be decoded in parallel with simultaneous insertions at multiple locations. When using the binary tree loss, we find empirically that we can generate sequences of length n using close to the asymptomatic limit of $\lfloor \log_2 n \rfloor + 1$ steps without any quality degradation. This allows us to match the performance of the standard Transformer on the WMT 2014 English-German translation task while using substantially fewer iterations during decoding.

Input: But on the other side of the state, that is not the impression many people have of their former governor.

Output: Aber auf der anderen Seite des Staates ist das nicht der Eindruck, den viele von ihrem ehemaligen Gouverneur haben.

Parallel decode (binary tree loss):

Aber_ auf_ der_ anderen_ Seite_ des_ Staates_ ist_ das_ nicht_ der_ Eindruck_ ,_ den_ viele_ von_ ihrem_ ehemaligen_ Gouverneur_ haben_ ._
Aber_ auf_ der_ anderen_ Seite_ des_ Staates_ ist_ das_ nicht_ der_ Eindruck_ ,_ den_ viele_ von_ ihrem_ ehemaligen_ Gouverneur_ haben_ ._
Aber_ auf_ der_ anderen_ Seite_ des_ Staates_ ist_ das_ nicht_ der_ Eindruck_ ,_ den_ viele_ von_ ihrem_ ehemaligen_ Gouverneur_ haben_ ._
Aber_ auf_ der_ anderen_ Seite_ des_ Staates_ ist_ das_ nicht_ der_ Eindruck_ ,_ den_ viele_ von_ ihrem_ ehemaligen_ Gouverneur_ haben_ ._
Aber_ auf_ der_ anderen_ Seite_ des_ Staates_ ist_ das_ nicht_ der_ Eindruck_ ,_ den_ viele_ von_ ihrem_ ehemaligen_ Gouverneur_ haben_ ._

. . . . . .

Input: Everyone has the Internet, an iPad and eBooks.

Output: Jeder hat das Internet, ein iPad und eBooks.

Greedy decode (uniform loss):

Jeder_ hat_ das_ Internet_ ,_ ein_ i Pad_ und_ eB oo ks_ ._
Jeder_ hat_ das_ Internet_ ,_ ein_ i Pad_ und_ eB oo ks_ ._
Jeder_ hat_ das_ Internet_ ,_ ein_ i Pad_ und_ eB oo ks_ ._
Jeder_ hat_ das_ Internet_ ,_ ein_ i Pad_ und_ eB oo ks_ ._
Jeder_ hat_ das_ Internet_ ,_ ein_ i Pad_ und_ eB oo ks_ ._
Jeder_ hat_ das_ Internet_ ,_ ein_ i Pad_ und_ eB oo ks_ ._
Jeder_ hat_ das_ Internet_ ,_ ein_ i Pad_ und_ eB oo ks_ ._
Jeder_ hat_ das_ Internet_ ,_ ein_ i Pad_ und_ eB oo ks_ ._

(continued)
Jeder_ hat_ das_ Internet_ ,_ ein_ i Pad_ und_ eB oo ks_ ._
Jeder_ hat_ das_ Internet_ ,_ ein_ i Pad_ und_ eB oo ks_ ._
Jeder_ hat_ das_ Internet_ ,_ ein_ i Pad_ und_ eB oo ks_ ._
Jeder_ hat_ das_ Internet_ ,_ ein_ i Pad_ und_ eB oo ks_ ._
Jeder_ hat_ das_ Internet_ ,_ ein_ i Pad_ und_ eB oo ks_ ._
Jeder_ hat_ das_ Internet_ ,_ ein_ i Pad_ und_ eB oo ks_ ._
Jeder_ hat_ das_ Internet_ ,_ ein_ i Pad_ und_ eB oo ks_ ._

. . . . . .

Input: They want to create a post on the college's equal opportunities committee to ensure that their opinions can be aired freely.

Output: Sie wollen einen Posten im Ausschuss für Chancengleichheit des Kollegiums einrichten, um sicherzustellen, dass ihre Meinungen frei zur Sprache gebracht werden können.

Parallel decode (uniform loss):

Sie_ wollen_ einen_ Posten_ im_ Ausschuss_ für_ Chancengleichheit_ des_ Koll egi ums_ einrichten_ ,_ um_ sicherzustellen_ ,_ dass_ ihre_ Mein ungen_ frei_ zur_ Sprache_ gebracht_ werden_ können_ ._
Sie_ wollen_ einen_ Posten_ im_ Ausschuss_ für_ Chancengleichheit_ des_ Koll egi ums_ einrichten_ ,_ um_ sicherzustellen_ ,_ dass_ ihre_ Mein ungen_ frei_ zur_ Sprache_ gebracht_ werden_ können_ ._
Sie_ wollen_ einen_ Posten_ im_ Ausschuss_ für_ Chancengleichheit_ des_ Koll egi ums_ einrichten_ ,_ um_ sicherzustellen_ ,_ dass_ ihre_ Mein ungen_ frei_ zur_ Sprache_ gebracht_ werden_ können_ ._
Sie_ wollen_ einen_ Posten_ im_ Ausschuss_ für_ Chancengleichheit_ des_ Koll egi ums_ einrichten_ ,_ um_ sicherzustellen_ ,_ dass_ ihre_ Mein ungen_ frei_ zur_ Sprache_ gebracht_ werden_ können_ ._
Sie_ wollen_ einen_ Posten_ im_ Ausschuss_ für_ Chancengleichheit_ des_ Koll egi ums_ einrichten_ ,_ um_ sicherzustellen_ ,_ dass_ ihre_ Mein ungen_ frei_ zur_ Sprache_ gebracht_ werden_ können_ ._
Sie_ wollen_ einen_ Posten_ im_ Ausschuss_ für_ Chancengleichheit_ des_ Koll egi ums_ einrichten_ ,_ um_ sicherzustellen_ ,_ dass_ ihre_ Mein ungen_ frei_ zur_ Sprache_ gebracht_ werden_ können_ ._
Sie_ wollen_ einen_ Posten_ im_ Ausschuss_ für_ Chancengleichheit_ des_ Koll egi ums_ einrichten_ ,_ um_ sicherzustellen_ ,_ dass_ ihre_ Mein ungen_ frei_ zur_ Sprache_ gebracht_ werden_ können_ ._
Sie_ wollen_ einen_ Posten_ im_ Ausschuss_ für_ Chancengleichheit_ des_ Koll egi ums_ einrichten_ ,_ um_ sicherzustellen_ ,_ dass_ ihre_ Mein ungen_ frei_ zur_ Sprache_ gebracht_ werden_ können_ ._

Figure 5.4: Example decodes using models trained with the binary tree loss and the uniform loss. Within each row, the underlined blue words are those being inserted, and the gray words represent those from the final output that have not yet been generated. We observe that both models are able to achieve a high degree of parallelism, with the binary tree model matching the logarithmic theoretical lower bound on the number of parallel decoding iterations thanks to its training objective.

Chapter 6

Conclusion

In this dissertation, we explored how structured neural models and structured decoding procedures can be used to tackle problems in domains that are not immediately accessible to standard sequence models. The first part of the dissertation focused on models for problems with non-trivial output constraints. In Chapter 2, we proposed a model for constituency parsing that ensures well-formed outputs by means of a span-oriented decoder. This idea was taken a step further in Chapter 3 through the introduction of Abstract Syntax Networks, whose modular structure enables them to directly model distributions over syntactically correct Python programs, lambda expressions, or Prolog expressions. The second part of the dissertation then focused on what can be accomplished through more flexible decoding strategies for sequence models. In Chapter 4, we devised a blockwise parallel decoding algorithm for Transformer models that allows us to predict the same output as a greedy decode in a fraction of the decoding iterations. Even further improvements were achieved with the Insertion Transformer model from Chapter 5, which permits out-of-order generation and is capable of logarithmic-time parallel decoding when following a balanced binary tree ordering.

The ideas presented here suggest a number of promising avenues for further exploration. For instance, while the structured models we propose enforce proper syntactic structure, additional conditions must be met to guarantee fully executable outputs for a program synthesis task, e.g. variables should be declared before being used and function arguments should have the appropriate types. Designing systems that take more of these constraints into account will undoubtedly yield dividends by eliminating additional classes of invalid output structures from the search space. Turning to sequence generation, we can imagine moving beyond the capabilities of our Insertion Transformer architecture by implementing additional types of operations, e.g. multi-token insertions, swaps, or mutations. Such extensions would not only provide additional tools for exploring the trade-off between speed and accuracy, but could also prove useful in editing-oriented applications such as post-editing and grammar correction. We hope the community will continue to explore the utility of structured neural models and structured decoding procedures in future work.

Bibliography

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Miltiadis Allamanis, Daniel Tarlow, Andrew D. Gordon, and Yi Wei. 2015. Bimodal modelling of source code and natural language. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*. pages 2123–2132.

David Alvarez-Melis and Tommi S. Jaakkola. 2017. Tree-structured decoding with doubly-recurrent neural networks. In *Proceedings of the International Conference on Learning Representations (ICLR) 2017*.

Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. 2016. Neural module networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Oral.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *CoRR* abs/1409.0473. http://arxiv.org/abs/1409.0473.

Dzmitry Bahdanau, Jan Chorowski, Dmitriy Serdyuk, Philemon Brakel, and Yoshua Bengio. 2016. End-to-End Attention-based Large Vocabulary Speech Recognition. In *ICASSP*.

Miguel Ballesteros, Yoav Goldberg, Chris Dyer, and Noah A Smith. 2016. Training with exploration improves a greedy stack-lstm parser. *arXiv preprint arXiv:1603.03793* .

Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. Deepcoder: Learning to write programs. *CoRR* abs/1611.01989.

Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. 2003. A neural probabilistic language model. *J. Mach. Learn. Res.* 3:1137–1155. http://dl.acm.org/citation.cfm?id=944919.944966.

Steven Bird, Ewan Klein, and Edward Loper. 2009. *Natural Language Processing with Python*. O'Reilly Media, Inc., 1st edition.

Anders Björkelund, Ozlem Cetinoglu, Agnieszka Falenska, Richárd Farkas, Thomas Müller, Wolfgang Seeker, and Zsolt Szántó. 2014. The IMS-Wrocław-Szeged-CIS entry at the SPMRL 2014 shared task: Reranking and morphosyntax meet unlabeled data. *Notes of the SPMRL* .

William Chan, Navdeep Jaitly, Quoc Le, and Oriol Vinyals. 2016. Listen, Attend and Spell: A Neural Network for Large Vocabulary Conversational Speech Recognition. In *ICASSP*.

Danqi Chen and Christopher D. Manning. 2014. A fast and accurate dependency parser using neural networks. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*. pages 740–750. http://aclweb.org/anthology/D/D14/D14-1082.pdf.

Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. In *EMNLP*.

Michael Collins. 2003. Head-driven statistical models for natural language parsing. *Computational linguistics* 29(4):589–637.

James Cross and Liang Huang. 2016. Span-based constituency parsing with a structure-label system and provably optimal dynamic oracles. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, EMNLP 2016, Austin, Texas, USA, November 1-4, 2016*. pages 1–11.

Li Dong and Mirella Lapata. 2016. Language to logical form with neural attention. *CoRR* abs/1601.01280.

Timothy Dozat and Christopher D. Manning. 2016. Deep biaffine attention for neural dependency parsing. *CoRR* abs/1611.01734. http://arxiv.org/abs/1611.01734.

Greg Durrett and Dan Klein. 2015. Neural crf parsing. *arXiv preprint arXiv:1507.03641* .

Chris Dyer, Adhiguna Kuncoro, Miguel Ballesteros, and Noah A Smith. 2016. Recurrent neural network grammars. *arXiv preprint arXiv:1602.07776* .

Jenny Rose Finkel, Alex Kleeman, and Christopher D Manning. 2008. Efficient, feature-based, conditional random field parsing. In *ACL*. volume 46, pages 959–967.

Nicolas Ford, Daniel Duckworth, Mohammad Norouzi, and George E. Dahl. 2018. The Importance of Generation Order in Language Modeling. In *EMNLP*.

Tommaso Furlanello, Zachary C Lipton, AI Amazon, Laurent Itti, and Anima Anandkumar. 2017. Born again neural networks. In *NIPS Workshop on Meta Learning*.

Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N. Dauphin. 2017. Convolutional sequence to sequence learning. *CoRR* abs/1705.03122. http://arxiv.org/abs/1705.03122.

Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics*.

Yoav Goldberg and Joakim Nivre. 2013. Training deterministic parsers with non-deterministic oracles. *Transactions of the association for Computational Linguistics* 1:403–414.

Alex Graves. 2012. Sequence Transduction with Recurrent Neural Networks. In *ICML Representation Learning Workshop*.

Alex Graves. 2013. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850* .

Jiatao Gu, James Bradbury, Caiming Xiong, Victor O.K. Li, and Richard Socher. 2018. Non-autoregressive neural machine translation. In *International Conference on Learning Representations*. https://openreview.net/forum?id=B1l8BtlCb.

Jiatao Gu, Qi Liu, and Kyunghyun Cho. 2019. Insertion-based Decoding with Automatically Inferred Generation Order. In *arXiv*.

David Leo Wright Hall, Greg Durrett, and Dan Klein. 2014. Less grammar, more features. In *ACL (1)*. pages 228–237.

James Henderson. 2004. Discriminative training of a neural network statistical parser. In *Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics*. Association for Computational Linguistics, page 95.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531* .

Łukasz Kaiser, Aurko Roy, Ashish Vaswani, Niki Pamar, Samy Bengio, Jakob Uszkoreit, and Noam Shazeer. 2018. Fast decoding in sequence models using discrete latent variables. *arXiv preprint arXiv:1803.03382* .

Nal Kalchbrenner, Erich Elsen, Karen Simonyan, Seb Noury, Norman Casagrande, Edward Lockhart, Florian Stimberg, Aaron van den Oord, Sander Dieleman, and Koray Kavukcuoglu. 2018. Efficient neural audio synthesis. *arXiv preprint arXiv:1802.08435* .

Yoon Kim and Alexander M. Rush. 2016. Sequence-level knowledge distillation. *arXiv preprint arXiv:1606.07947* .

Diederik P. Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *CoRR* abs/1412.6980. http://arxiv.org/abs/1412.6980.

Eliyahu Kiperwasser and Yoav Goldberg. 2016. Simple and accurate dependency parsing using bidirectional lstm feature representations. *arXiv preprint arXiv:1603.04351* .

Dan Klein and Christopher D Manning. 2003. Accurate unlexicalized parsing. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*. pages 423–430.

Tom Kwiatkowski, Eunsol Choi, Yoav Artzi, and Luke S. Zettlemoyer. 2013. Scaling semantic parsers with on-the-fly ontology matching. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing, EMNLP 2013, 18-21 October 2013, Grand Hyatt Seattle, Seattle, Washington, USA, A meeting of SIGDAT, a Special Interest Group of the ACL*. pages 1545–1556.

Jason Lee, Elman Mansimov, and Kyunghyun Cho. 2018. Deterministic non-autoregressive neural sequence modeling by iterative refinement. *CoRR* abs/1802.06901. http://arxiv.org/abs/1802.06901.

Percy Liang, Michael I. Jordan, and Dan Klein. 2010. Learning programs: A hierarchical bayesian approach. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel*. pages 639–646.

Percy Liang, Michael I. Jordan, and Dan Klein. 2013. Learning dependency-based compositional semantics. *Comput. Linguist.* 39(2):389–446. https://doi.org/10.1162/COLI_a_00127.

Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomás Kociský, Fumin Wang, and Andrew Senior. 2016. Latent predictor networks for code generation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*.

Jiangming Liu and Yue Zhang. 2016. Shift-reduce constituent parsing with neural lookahead features. *CoRR* abs/1612.00567. http://arxiv.org/abs/1612.00567.

Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. 2015. Deep learning face attributes in the wild. In *Proceedings of International Conference on Computer Vision (ICCV)*.

Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective Approaches to Attention-based Neural Machine Translation. In *EMNLP*.

Chris J. Maddison and Daniel Tarlow. 2014. Structured generative models of natural source code. In *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014*. pages 649–657.

Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. 1993. Building a large annotated corpus of english: The penn treebank. *Comput. Linguist.* 19(2):313–330. http://dl.acm.org/citation.cfm?id=972470.972475.

Aditya Krishna Menon, Omer Tamuz, Sumit Gulwani, Butler W. Lampson, and Adam Kalai. 2013. A machine learning framework for programming by example. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*. pages 187–195.

Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, Kevin Duh, Manaal Faruqui, Cynthia Gan, Dan Garrette, Yangfeng Ji, Lingpeng Kong, Adhiguna Kuncoro, Gaurav Kumar, Chaitanya Malaviya, Paul Michel, Yusuke Oda, Matthew Richardson, Naomi Saphra, Swabha Swayamdipta, and Pengcheng Yin. 2017. Dynet: The dynamic neural network toolkit. *arXiv preprint arXiv:1701.03980* .

Mohammad Norouzi, Samy Bengio, Navdeep Jaitly Zhifeng Chen, Mike Schuster, Yonghui Wu, and Dale Schuurmans. 2016. Reward Augmented Maximum Likelihood for Neural Structured Prediction. In *NIPS*.

Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. 2016a. WaveNet: A Generative Model for Raw Audio. In *arXiv*.

Aaron van den Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. 2016b. Pixel Recurrent Neural Networks. In *ICML*.

Aaron van den Oord, Nal Kalchbrenner, Oriol Vinyals, Lasse Espeholt, Alex Graves, and Koray Kavukcuoglu. 2016c. Conditional Image Generation with PixelCNN Decoders. In *NIPS*.

Aaron van den Oord, Yazhe Li, Igor Babuschkin, Karen Simonyan, Oriol Vinyals, Koray Kavukcuoglu, George van den Driessche, Edward Lockhart, Luis C Cobo, Florian Stimberg, et al. 2017. Parallel wavenet: Fast high-fidelity speech synthesis. *arXiv preprint arXiv:1711.10433* .

Niki Parmar, Ashish Vaswani, Jakob Uszkoreit, Lukasz Kaiser, Noam Shazeer, and Alexander Ku. 2018. Image transformer. *CoRR* abs/1802.05751. http://arxiv.org/abs/1802.05751.

Slav Petrov and Dan Klein. 2007. Improved inference for unlexicalized parsing. In *Proceedings of the Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics*. Assocation for Computational Linguistics.

Ana-Maria Popescu, Oren Etzioni, and Henry Kautz. 2003. Towards a theory of natural language interfaces to databases. In *Proceedings of the 8th international conference on Intelligent user interfaces*. ACM, pages 149–157.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Vancouver, Canada, pages 1139–1149. https://doi.org/10.18653/v1/P17-1105.

Scott Reed, Aäron van den Oord, Nal Kalchbrenner, Sergio Gómez Colmenarejo, Ziyu Wang, Dan Belov, and Nando de Freitas. 2017. Parallel Multiscale Autoregressive Density Estimation. In *ICML*.

Alexander M Rush, Sumit Chopra, and Jason Weston. 2015. A neural attention model for abstractive sentence summarization. *arXiv preprint arXiv:1509.00685* .

Andrei A. Rusu, Sergio Gomez Colmenarejo, Caglar Gulcehre, Guillaume Desjardins, James Kirkpatrick, Razvan Pascanu, Volodymyr Mnih, Koray Kavukcuoglu, and Raia Hadsell. 2016. Policy Distillation. In *ICLR*.

Djamé Seddah, Sandra Kübler, and Reut Tsarfaty. 2014. Introducing the spmrl 2014 shared task on parsing morphologically-rich languages. In *Proceedings of the First Joint Workshop on Statistical Parsing of Morphologically Rich Languages and Syntactic Analysis of Non-Canonical Languages*. Dublin City University, Dublin, Ireland, pages 103–109. http://www.aclweb.org/anthology/W14-6111.

Harshil Shah, Bowen Zheng, and David Barber. 2018. Generating Sentences Using a Dynamic Canvas. In *AAAI*.

Richard Shin, Alexander A. Alemi, Geoffrey Irving, and Oriol Vinyals. 2017. Tree-structured variational autoencoder. In *Proceedings of the International Conference on Learning Representations (ICLR) 2017*.

Michael Sipser. 2006. *Introduction to the Theory of Computation*. Course Technology, second edition.

Mitchell Stern, Jacob Andreas, and Dan Klein. 2017. A minimal span-based neural constituency parser. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Vancouver, Canada, pages 818–827. https://doi.org/10.18653/v1/P17-1076.

Mitchell Stern, William Chan, Jamie Kiros, and Jakob Uszkoreit. 2019. Insertion transformer: Flexible sequence generation via insertion operations. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*. PMLR, Long Beach, California, USA, volume 97 of *Proceedings of Machine Learning Research*, pages 5976–5985. http://proceedings.mlr.press/v97/stern19a.html.

Mitchell Stern, Noam Shazeer, and Jakob Uszkoreit. 2018. Blockwise parallel decoding for deep autoregressive models. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, Curran Associates, Inc., pages 10086–10095. http://papers.nips.cc/paper/8212-blockwise-parallel-decoding-for-deep-autoregressive-models.pdf.

Ilya Sutskever, Oriol Vinyals, and Quoc Le. 2014. Sequence to Sequence Learning with Neural Networks. In *NIPS*.

Ben Taskar, Vassil Chatalbashev, Daphne Koller, and Carlos Guestrin. 2005. Learning structured prediction models: A large margin approach. In *Proceedings of the 22Nd International Conference on Machine Learning*. ACM, New York, NY, USA, ICML '05, pages 896–903. https://doi.org/10.1145/1102351.1102464.

Le Quang Thang, Hiroshi Noji, and Yusuke Miyao. 2015. Optimal shift-reduce constituent parsing with structured perceptron. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*. volume 1, pages 1534–1544.

Kristina Toutanova, Dan Klein, Christopher D. Manning, and Yoram Singer. 2003. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology - Volume 1*. Association for Computational Linguistics, Stroudsburg, PA, USA, NAACL '03, pages 173–180. https://doi.org/10.3115/1073445.1073478.

Aäron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alexander Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. 2016. WaveNet: A generative model for raw audio. *CoRR* abs/1609.03499. http://arxiv.org/abs/1609.03499.

Ashish Vaswani, Samy Bengio, Eugene Brevdo, Francois Chollet, Aidan N. Gomez, Stephan Gouws, Llion Jones, Łukasz Kaiser, Nal Kalchbrenner, Niki Parmar, Ryan Sepassi, Noam Shazeer, and Jakob Uszkoreit. 2018. Tensor2tensor for neural machine translation. *CoRR* abs/1803.07416. http://arxiv.org/abs/1803.07416.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *CoRR* http://arxiv.org/abs/1706.03762.

Oriol Vinyals, Samy Bengio, and Manjunath Kudlur. 2015a. Order Matters: Sequence to sequence for sets. In *ICLR*.

Oriol Vinyals, Łukasz Kaiser, Terry Koo, Slav Petrov, Ilya Sutskever, and Geoffrey Hinton. 2015b. Grammar as a foreign language. In *Advances in Neural Information Processing Systems*. pages 2773–2781.

Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. 2015c. Show and Tell: A Neural Image Caption Generator. In *CVPR*.

Adrienne Wang, Tom Kwiatkowski, and Luke S Zettlemoyer. 2014. Morpho-syntactic lexical generalization for ccg semantic parsing. In *EMNLP*. pages 1284–1295.

Chunqi Wang, Ji Zhang, and Haiqing Chen. 2018. Semi-Autoregressive Neural Machine Translation. In *EMNLP*.

Daniel C. Wang, Andrew W. Appel, Jeff L. Korn, and Christopher S. Serra. 1997. The zephyr abstract syntax description language. In *Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL), 1997*. USENIX Association, Berkeley, CA, USA, DSL'97, pages 17–17.

Peilu Wang, Yao Qian, Frank K. Soong, Lei He, and Hai Zhao. 2015. A unified tagging solution: Bidirectional LSTM recurrent neural network with word embedding. *CoRR* abs/1511.00215. http://arxiv.org/abs/1511.00215.

Wenhui Wang and Baobao Chang. 2016. Graph-based dependency parsing with bidirectional LSTM. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. http://aclweb.org/anthology/P/P16/P16-1218.pdf.

Yuxuan Wang, RJ Skerry-Ryan, Daisy Stanton, Yonghui Wu, Ron J. Weiss, Navdeep Jaitly, Zongheng Yang, Ying Xiao, Zhifeng Chen, Samy Bengio, Quoc Le, Yannis Agiomyrgiannakis, Rob Clark, and Rif A. Saurous. 2017. Tacotron: Towards End-to-End Speech Synthesis. In *INTERSPEECH*.

Sean Welleck, Kiante Brantley, Hal Daume, and Kyunghyun Cho. 2019. Non-Monotonic Sequential Text Generation. In *arXiv*.

Sam Wiseman and Alexander M Rush. 2016. Sequence-to-sequence learning as beam-search optimization. *arXiv preprint arXiv:1606.02960* .

Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhutdinov, Richard Zemel, and Yoshua Bengio. 2015. Show, Attend and Tell: Neural Image Caption Generation with Visual Attention. In *ICML*.

Zhilin Yang, Zihang Dai, Ruslan Salakhutdinov, and William W. Cohen. 2018. Breaking the Softmax Bottleneck: A High-Rank RNN Language Model. In *ICLR*.

Luke S. Zettlemoyer and Michael Collins. 2005. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *UAI '05, Proceedings of the 21st Conference in Uncertainty in Artificial Intelligence, Edinburgh, Scotland, July 26-29, 2005*. pages 658–666.

Luke S. Zettlemoyer and Michael Collins. 2007. Online learning of relaxed ccg grammars for parsing to logical form. In *In Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL-2007*. pages 678–687.

Kai Zhao and Liang Huang. 2015. Type-driven incremental semantic parsing with polymorphism. In *NAACL HLT 2015, The 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Denver, Colorado, USA, May 31 - June 5, 2015*. pages 1416–1421.