BUILDING THE LEARNING-FROM-INTERACTION PIPELINE FOR LARGE LANGUAGE MODELS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Shikhar Murty
June 2025

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Christopher Manning, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Tatsunori Hashimoto

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Jacob Andreas

Approved for the Stanford University Committee on Graduate Studies.

Stacey F. Bent, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format.

Abstract

LLMs have demonstrated remarkable capabilities, and there is growing interest in using them as agents—systems that can translate complex human goals, expressed in natural language, into sequences of actions within digital environments like web browsers. Achieving this requires two core competencies: first, the ability to understand arbitrary and compositional language inputs; and second, the capacity to learn about unfamiliar environments so that language goals can be grounded in effective, multi-step decision-making. This thesis addresses both of these challenges.

In the first part, I introduce Tree Projections, a framework for understanding how transformers build compositional structure. I then present a series of results based on Tree Projections that illuminate the mechanisms behind compositional generalization, grokking, and sample-efficient learning in transformers. While Tree Projections help explain successful generalization, prior work has shown that standard transformers struggle with deep recursion due to a lack of mechanisms for unbounded hierarchical depth. To address this, I propose Pushdown Layers, an architectural augmentation that adds a stack-based memory to transformers. Pushdown Layers improve sample efficiency and generalization on tasks requiring nested or recursive reasoning.

In the second party, I introduce NNetNav and BAGEL, methods for unsupervised, open-ended exploration in web environments that enable models to automatically collect training data for new websites, without human supervision. Our best results come from fine-tuning LLMs with demonstrations collected via NNetNav, which uses the hierarchical structure of language to guide exploration policies. Using NNetNav, we collect 10,000 demonstrations from 20 real-world websites and fine-tune an 8B model, setting a new state-of-the-art among unsupervised methods and outperforming zero-shot GPT-4 on multiple browser benchmarks.

Taken together, these contributions bring us closer to digital language agents that can both handle the complexity of language instructions and autonomously learn from interacting with their environments.

Acknowledgements

First and foremost, I want to thank Chris Manning for the immense time, care, and mentorship he's offered me over the past 5.5 years. Nearly everything I've learned about research and scholarship has been shaped by his guidance. Chris taught me that great research isn't about cleverness or optimization—it's about asking the right questions. His research taste and approach to NLP are legendary, and I feel incredibly lucky to have absorbed even a fraction of it. I'm continually amazed by Chris's attention to detail, whether it's correctly post-processing the Penn Treebank or debugging the quirks of the latest transformer variant. His ability to stay current with the latest papers while deeply understanding the historical roots of the field is a rare combination—one I aspire to emulate. He's shown me that to invent the future, you have to understand the past. Beyond research, Chris has taught me how to teach, mentor, and communicate effectively—skills that are just as essential for scholars. I'm grateful that he trusted me to mentor undergraduate and master's students, explore unconventional research directions, and take on the responsibility of head TAing CS 224n.

Next, I would like to thank Jacob Andreas. Jacob has been like a second advisor during the early part of my PhD and has profoundly influenced how I think about compositionality, generalization, and language understanding more broadly. He also taught me a lot about the craft of writing papers and designing clear, effective figures—lessons I'll carry with me throughout my career. I'll always remember his advice to "build the experiment pipeline first"—a principle that has saved me more times than I can count. I'm deeply grateful for the time and care he's invested in mentoring me, the dense and thoughtful feedback on my writing, and the collaborative spirit he brought to every project. I'm incredibly proud of the papers we wrote together.

I would also like to thank Chris Potts, who has influenced my thinking in many important ways—from his online lectures on semantics, which I watched long before starting my PhD, to our early conversations on syntax and grounding during my first year. I still remember how Chris was the only faculty member who signed up for one-on-one random lunch pairings with grad students, back when the NLP group had only 30 students. That openness and generosity left a lasting impression.

I'm also very grateful to Tatsu, who worked with me during my first year and taught me a great deal about statistics, especially how to run careful analyses and interpret results rigorously. I've really appreciated the many thoughtful discussions we've had over the years, and your openness to entertaining (and challenging) my various project ideas. Thank you for your generosity with your time and feedback.

Next, I would like to thank some of my earliest research mentors, Mausam, Andrew McCallum and Aaron Courville. A special thanks to Mausam, who first introduced me to AI and NLP as an undergrad and generously took me on as a researcher in his lab. A huge thank you to Andrew McCallum, who took a chance on me and welcomed me into the IESL lab as an intern. I owe much of what I know about graphical models and information extraction to Andrew and to my PhD mentors at IESL—Luke Vilnis and Pat Verga. Andrew's kindness, curiosity, and creativity have left a deep impression on me. I'm also grateful to Aaron for introducing me to the vibrant deep learning research world of 2018 Montreal, and to my PhD mentor at MILA, Dzmitry Bahdanau for introducing me to systematic generalization.

I've been incredibly fortunate to work with some amazing student mentees over the years: Xinran, Ananjan, Ananth, Ethan, Hongmeng, and Houjun. It's been a joy to watch many of you go on to start your own PhDs, and I'm genuinely proud to have been part of your journey. Working with you has taught me that mentorship is never a one-way street: there's so much I've learned through our collaborations. Thank you!

Research is hardly a one-person effort, and I have been blessed to work with some of the most gifted scientists in the field, both in industry and academia. Thanks to Marco and Scott for teaching me how to debug machine learning models, and to Pete, Mandar, and Kenton for introducing me to the world of grounded NLP. Thanks also to Dima for the great discussions on web agents, reinforcement learning, and collecting human preferences.

Thanks to all my friends and colleagues in the Stanford NLP group. This is a special group that has evolved so much over the years. From my first year to the last, the faces may have rotated, but the energy never did. I'd like to give a special shoutout to my officemates—Róbert, Julie, Dilara, Ashwin, and John—for keeping the NLP lab office culture (and the endless research and non-research discussions) alive in the post-COVID work-from-home world. Thanks to John for always listening to random research pitches, and to labmates Tolúlọpẹ́, Anna, Zen, Eric, Fede, Kaitlyn, and Nelson for your support and camaraderie throughout the PhD journey.

To Pratyusha—knowing you over the last 10 years has been one of life's greatest gifts. Thank you for being by my side throughout the PhD, and for somehow managing to be both my partner in

life and in research as we co-authored four papers together. I couldn't have done this without you. And thanks to Pratyusha's family for all the support and laughs.

And last, but certainly not least, I want to express my deepest gratitude to my family. To Mom and Dad—thank you for your unwavering love and belief in me. Dad, you've been an incredible role model—I aspire to be like you. Mom, your love (and delicious food!) kept me going, especially on the hardest days. I'm so grateful for everything you've done. My younger brother Supransh has been my first reader and most loyal supporter, always eager to dive into whatever I'm working on, even the dense stuff. Thank you for your constant curiosity and encouragement.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

> *"My fondest dream is to someday build a machine that really thinks, learns, communicates with humans and manipulates its environment in a fairly sophisticated way."*
>
> Claude Shannon, 1952

A long-standing goal of artificial intelligence has been to build helpful agents that follow human instructions by manipulating some shared environment [Winograd, 1972]. An especially promising application for such agents lies in digital environments, such as web browsers, computers, and mobile devices, where they can act as copilots, reducing the human burden of repetitive digital tasks. Beyond their practical utility, these environments provide a natural testbed for testing language understanding, multi-step reasoning and grounding abilities. To see why, consider a simple task like *Upload the image to overleaf after resizing it according to stanford thesis requirements*. Successfully completing this task requires the agent to construct a compositional interpretation of the instruction based on English syntax, ground words like "upload" into sequences of concrete actions (e.g., `double click upload icon` → `select file` → `click upload button`), reason over dynamic and visually complex interfaces, and plan out a coordinated series of steps in order to satisfy a latent post-condition.

Recent advances in large language models (LLMs) have brought us closer to this goal, enabling the development of digital agents [Yao et al., 2022, Kim et al., 2023, Murty et al., 2024]. Although these systems have made impressive strides, they remain far from reliable, greatly limiting their applicability to real-world applications. How can we move towards more reliable digital agents?

At a high level, the task of executing language instructions in digital environments requires

two fundamental competencies: a robust understanding of language and deep knowledge of the environment in which the instruction is to be executed. Neither of these competencies comes for free—LLMs, though powerful, are not naturally equipped to reason structurally about complex language instructions (especially instructions with deep nested structure [Hahn, 2020]), and by nature of being completely ungrounded, generally do not have knowledge about unknown digital interfaces. This thesis addresses both of these gaps—by developing methods and architectural innovations to carefully characterize and improve how deep linguistic structure is handled in large language models, and by proposing new learning methods that enables large language model *agents* to quickly acquire knowledge about the digital environments they inhabit.

1.1 A Brief History of Language Agents in AI and NLP

The ambition to build intelligent agents that can understand and act through natural language has shaped the trajectory of AI since its earliest days. This section traces how that goal has evolved—from early symbolic systems to modern large language models—and how the field has repeatedly returned to the challenge of grounding language in perception and action. I begin with two formative systems, ELIZA and SHRDLU, that represent contrasting early approaches to language understanding. I then examine how the field shifted away from grounded interaction in favor of syntactic and statistical modeling through the 1980s and 1990s. Finally, I trace the re-emergence of grounding in the 2000s via semantic parsing.

1.1.1 The Early Days of AI and NLP

ELIZA, developed in the 1960s, was a rule-based conversational agent that mimicked a psychotherapist using pattern matching and templated responses [Weizenbaum, 1966]. Although it appeared to engage in meaningful dialogue, ELIZA lacked any representation of context, memory, or environment. It was an entirely ungrounded system, operating solely on surface-level linguistic patterns without connecting language to perception or action. In contrast, SHRDLU, created in the early 1970s, operated in a constrained blocks world [Winograd, 1972]. SHRDLU could manipulate symbolic representations of physical objects based on user natural language commands like *put the red block on the green cube*. It also maintained a limited dialogue context, enabling referential expressions and basic clarification. SHRDLU was an early grounded language system, in that it connected words to entities and actions in a world. However, its capabilities were strictly limited to a narrow domain, and its rule-based logic made it brittle and inflexible outside its designed environment.

Following these early systems, research in natural language processing (NLP) pivoted away from grounded agents toward corpus-based linguistic analysis. This shift was driven by advances in statistical modeling and the introduction of new datasets supporting tasks such as syntactic parsing, semantic parsing, part-of-speech tagging, and topic modeling.

In syntax, early developments included work on probabilistic context-free grammars (PCFGs) [Jelinek et al., 1992], and the introduction of the Penn Treebank [Marcus et al., 1993], which provided a large-scale hand annotated corpus of syntax trees that enabled data-driven methods [e.g., Collins, 1997, Charniak, 1997]. For machine translation, the IBM alignment models were some of the earliest models for word alignment [Brown et al., 1993], while Brown clustering introduced a distributional method for inducing word classes [Brown et al., 1992]. n-gram language models, often trained on newswire, became foundational to language modeling [Jelinek, 1990, Goodman, 2001]. More sophisticated sequence models, such as Hidden Markov Models (HMMs), were widely adopted for tasks such as tagging, parsing, and speech recognition [Church, 1988]. These probabilistic sequence models—natural precursors to today's LLMs—enabled more robust handling of ambiguity and linguistic variation at scale.

Although this era marked a shift away from language as a medium for action, it laid critical groundwork for the resurgence of grounding that followed. Formal grammars developed during this time—such as Combinatory Categorial Grammar (CCG) [Steedman, 1996]—offered mechanisms for compositional semantics, pairing syntactic categories with logical forms to enable precise, rule-based mappings from surface form to meaning. In parallel, advances in statistical learning—from HMMs and n-gram models to algorithms like Expectation-Maximization for latent-variable modeling—provided a more powerful toolkit for modeling structure in data. Thus, when grounding re-emerged in the 2000s, it drew on both symbolic and statistical corpus-based techniques to model how complex meanings arise from linguistic form.

1.1.2 A Return to Grounding in the 2000s

A return to grounding began to take shape in the mid-2000s through the revival of semantic parsing. In contrast to syntactic parsing, semantic parsing focuses on mapping natural language to executable meaning representations—logical forms, programs, or action sequences that can be grounded in a system's environment. This represented a conceptual shift: rather than analyzing language for its own sake, semantic parsing treated language as a *specification for behavior*.

Early work such as Zelle and Mooney [1996] framed this problem as mapping natural language questions to Prolog-style queries over small databases. This was followed by statistical approaches

[Wong and Mooney, 2006, Zettlemoyer and Collins, 2005] that used training data to induce mappings from language to lambda calculus expressions with limited supervision. These models reintroduced compositional meaning into NLP (e.g. Liang et al. [2013]) in a way that was tightly connected to actions in a domain. Semantic parsing also enabled more explicitly situated grounding by learning to map from language to physical actions or spatial commands for robots [Chen and Mooney, 2011, Artzi and Zettlemoyer, 2013, Tellex et al., 2011].

To make this concrete, we briefly illustrate how Combinatory Categorial Grammar (CCG), a formalism used in early semantic parsing work, builds compositional semantics using lambda calculus and syntactic categories.

CCG assigns each word a syntactic category encoding both its grammatical role and how it composes with others. These categories also carry lambda-calculus semantics, allowing meaning to be constructed incrementally bottom-up. For example, the imperative verb "*open*" is assigned the category S/NP, indicating it takes a noun phrase (NP) to its right to produce a sentence (S), and is associated with the function $\lambda x.\,\mathrm{open}(x)$.

Consider the sentence: *"Open the folder that contains the latest report."* Each phrase is incrementally composed using function application. The determiner "*the*" has category NP/N and semantics $\lambda f.\,\iota x.f(x)$, mapping a noun predicate to a unique entity. The operator ι is known as the iota operator, and denotes *definite description*—it refers to the unique entity x such that a given property holds. For instance, the phrase "*the folder*" corresponds to $\iota x.\,\mathrm{folder}(x)$, and "*the latest report*" corresponds to $\iota x.\,\mathrm{latest}(\mathrm{report})(x)$.

The relative clause "*that contains the latest report*" modifies the noun via a category $\mathrm{N}\backslash\mathrm{N}$, and the verb "*contains*" has type $\mathrm{S}/\mathrm{NP}\backslash\mathrm{NP}$, seeking both object and subject. Composing "*contains the latest report*" yields:

$$\lambda x.\,\mathrm{contains}(x, \iota x.\,\mathrm{latest}(\mathrm{report})(x))$$

which in turn modifies "*folder*" to form:

$$\lambda x.\,\mathrm{folder}(x) \wedge \mathrm{contains}(x, \iota x.\,\mathrm{latest}(\mathrm{report})(x))$$

Applying the determiner "*the*" gives:

$$\iota x.\,\mathrm{folder}(x) \wedge \mathrm{contains}(x, \iota x.\,\mathrm{latest}(\mathrm{report})(x))$$

Finally, applying the verb "*open*" yields the complete semantic interpretation of the command:

$$\text{open}(\iota x.\, \text{folder}(x) \wedge \text{contains}(x, \iota x.\, \text{latest}(\text{report})(x)))$$

This lambda calculus expression does more than merely represent the sentence's meaning——it directly specifies an executable action: opening a particular folder in the environment. In early semantic parsing systems, such formal expressions were typically converted into database queries or robot commands, bridging the gap between natural language and real-world behavior. This execution step highlighted the power of structured representations: by explicitly modeling how meaning is constructed, these systems could ground language in tangible actions.

By the late 2010s, semantic parsing had evolved to incorporate neural network architectures and reinforcement learning, allowing language models to generate programs, navigation plans, or tool use sequences from free-form inputs and environmental feedback [Andreas et al., 2016, Misra et al., 2017]. Neural approaches to semantic parsing generally followed three modeling traditions. The first retained symbolic grammars and used neural networks to learn probabilistic parsing rules [Misra and Artzi, 2016, Lee et al., 2016]. The second organized neural components hierarchically, as in seq2tree architectures [Andreas and Klein, 2015, Dong and Lapata, 2016, Rabinovich et al., 2017]. The third dispensed with symbolic structure entirely, training sequence-to-sequence models that directly mapped utterances to meaning representations using flat LSTM-based encoders and decoders [Jia and Liang, 2016], with data augmentation techniques used to inject weak forms of compositionality.

After the introduction of the transformer architecture, explicit injection of structure was largely abandoned in favor of training large pre-trained models on trillions of tokens from the internet. These models are capable of few-shot generalization, multi-turn dialogue, and a wide range of tasks via prompting. Fundamentally, large language models remain ungrounded: they are trained only on text, with no direct connection to perception, action, or physical context [Bisk et al., 2020]. While they can simulate grounded reasoning through text alone, they do not experience or interact with an environment in the way a true agent would.

Historically, the appeal of structured representations—such as those used in semantic parsing—was their ability to support strong generalization by explicitly modeling how meaning is composed from parts. In contrast, transformers often succeed at compositional tasks despite lacking explicit structure, but it remains unclear why this is the case, or when it will fail. Recent work has shown that self-attention architectures have fundamental limitations in their ability to recognize

or compute certain types of recursive and hierarchical structure [Hahn, 2020]. This raises open questions about whether large language models can reliably handle deeply nested or systematically compositional inputs—or whether structural biases remain necessary for robust generalization.

Both the ability to ground language in the world and being compositionally robust to linguistic variations are important for building reliable grounded agents. In the next section, I formally introduce the grounded instruction-following problem, argue why these two factors are essential, and give a brief overview of prior work and concerns in these directions.

1.2 Building Reliable Grounded Agents

Given a natural language goal g, an agent issues actions $\{a_1, a_2, \ldots, a_T\}$, where each action a_t is taken in response to the observation o_t at time step t. The full episode can be represented as a *trajectory*:

$$\tau = \{o_1, a_1, o_2, a_2, \ldots, o_T, a_T, o_{T+1}\}.$$

We formalize an instruction-following agent as a *language-conditioned policy* $\pi(a_t \mid \tau_{<t}, g)$, where $\tau_{<t} = \{o_1, a_1, \ldots, o_t\}$ denotes the trajectory observed so far. Executing an action leads to a new observation via an environment-specific transition function that is typically unknown and potentially non-deterministic.

At a high level, effective instruction following in diverse, unseen environments requires two key capabilities:

1. **Compositional language understanding:** the ability to put together the meaning of some potentially unseen utterance by combining the meaning of individual words, by using rules of syntax. This includes understanding how complex commands are built from simpler parts (*open the file and then upload it*) and interpreting modifiers and nested structure (*rename the folder inside Projects/2023 that contains the word draft in its name to final*). Crucially, agents must generalize to unseen combinations of known concepts, reflecting the kind of *systematicity* that underlies human generalization in language.

2. **Grounding language in the environment:** the ability to connect linguistic expressions to concrete concepts in the environment. This includes:

    - *Referential grounding:* resolving phrases like *upload button* or *new document* to specific UI elements based on visual features, DOM labels, or context.

- *Action grounding:* determining what sequence of low-level actions (e.g. mouse clicks, keyboard shortcuts, menu selections) corresponds to high-level verbs such as "copy", "upload" or "save as PDF" and composing these action sequences as necessary. For example, interpreting *copy the contents of data.txt and paste it into a new google doc* as a series of steps: opening a file manager, locating data.txt, opening it, selecting the text, and issuing a copy command to copy to clipboard
- *State tracking and post-condition reasoning:* maintaining an internal model of how actions change the environment, both to verify progress and to plan ahead. For instance, after executing `Ctrl+C`, the agent should infer that the clipboard now contains the selected text, and that issuing `Ctrl+V` in a text field will paste it. More broadly, grounding instructions like *paste the contents of* `data.txt` *into a Google Doc* requires reasoning about the desired post-condition (the Google Doc contains the file's contents), and planning a sequence of actions that reliably brings about that state-even across different applications.

The rest of this section expands on each of these capabilities, reviewing some important related work. Section 1.2.1 discusses compositionality in neural language models. Section 1.2.2 explores approaches to grounding language in environments.

1.2.1 Compositional Language Understanding

Compositionality is a core property of human language that allows us to understand entirely novel sentences by combining familiar elements in systematic ways. For example, once a person has learned how to *rename a folder*, they are able to generalize to more complex instructions such as *rename the second folder inside the Projects directory* or *rename all folders that contain the word 'draft' to 'final'*, even without having seen these exact commands before, as long as individual concepts like "second folder" are known. This ability to systematically combine familiar linguistic components in novel ways is central to human cognition. For AI agents, acquiring such compositional generalization capabilities is essential: it enables robust out-of-distribution performance and efficient learning from limited data.

Studying compositional generalization in deep neural networks has become a central thread in language understanding research. A canonical testbed for this is semantic parsing, where the goal is to translate natural language instructions into executable logical forms. One of the earliest benchmarks in this line of work is the SCAN dataset [Lake and Baroni, 2018], which creates compositional

test splits and shows that standard sequence-to-sequence models struggle to generalize beyond the training distribution. This work inspired a flurry of follow-ups, including SQOOP [Bahdanau et al., 2019], gSCAN [Ruis et al., 2020], ReaSCAN [Wu et al., 2021]. In particular gSCAN and ReaSCAN extend the SCAN dataset to grounded environments. In these settings, the meaning of an utterance depends not only on syntactic form, but also on context—for example, the phrase *big table* may refer to different objects depending on which tables are visible and their relative sizes.

These grounded variants emphasize that language understanding is not merely a matter of symbolic recombination; it requires contextual interpretation, integrating linguistic structure with perceptual cues. This perspective is especially important for instruction-following agents.

1.2.2 Grounding Language into the Environment

Referential grounding has often been studied in the context of visual question answering (VQA). Early VQA datasets employed relatively simple language [Antol et al., 2015], prompting a shift toward synthetic, compositional language datasets like CLEVR [Johnson et al., 2017] and GQA [Hudson and Manning, 2019], which pair controlled language with either synthetic or natural images. These datasets, originally designed for multimodal understanding and compositional reasoning, have also been repurposed for improving agent grounding [Majumdar et al., 2020]. However, such datasets are limited in their applicability to digital agents operating over user interfaces. To address this, newer datasets such as ScreenSpot [Cheng et al., 2024] and ScreenSpot-Pro [Li et al., 2025] have been introduced to evaluate and improve referential grounding for screen-based environments.

Beyond resolving referents, instruction following also requires knowledge of how actions affect the environment. Some works tackle this by explicitly modeling *environment dynamics*. For instance, Branavan et al. [2010] use past state transitions to train a look-ahead model that predicts the outcome of possible actions. This model, integrated with a policy learner, helps estimate whether a candidate action will lead to a high-reward state under a given instruction. A popular approach has been to train neural-network-based forward models for planning—either in latent space, or using video-based forward models that predict future frames along with inverse dynamics models to extract actions. In the context of digital agents, Chae et al. [2024] train large language model-based world models to simulate future states, using them in conjunction with a value function to choose the most promising action among candidates.

The most direct learning signal for instruction-following comes from interaction—either via expert demonstrations or the agent's own exploration. Historically, one modeling tradition has framed instruction following as a semantic parsing task: mapping natural language into a structured plan that

a symbolic executor can execute. These plans could take the form of formal logical specifications induced from data [Chen and Mooney, 2011], or be derived from compositional models of natural language meaning [Artzi and Zettlemoyer, 2013, Andreas and Klein, 2015]. These methods leverage the compositional structure of language as a scaffold for generalization and data-efficient learning.

In contrast, a second line of work frames instruction following as a direct sequence-to-sequence mapping problem, bypassing explicit intermediate representations by directly learning policies that map instructions to actions. A notable early example is Branavan et al. [2009], which maps natural language troubleshooting instructions to low-level commands in the Windows OS. Their system learns a log-linear policy over hand-crafted features derived from the UI state and interaction history. The environment is represented via categorical features (e.g., element labels, positions, parent windows), and the reward function penalizes mismatches between instructions and UI descriptions. More recent work eschews hand-engineered features in favor of end-to-end neural policies that operate on raw observations [Mei et al., 2016, Misra et al., 2017, inter alia].

1.3 Using LLMs for Grounded Instruction Following

While these end-to-end neural policies achieved impressive results, they typically operated on non pre-trained representations—learning everything from scratch and requiring substantial data. In the 2020s, large language models (LLMs) such as Gemini and GPT-4 introduced a new paradigm: they can serve as powerful feature extractors or as starting points for fine-tuning policies, offering a more data-efficient path to building grounded instruction-following agents. This section reviews some of these efforts

Early efforts in using LLMs for grounded agents leveraged them in a zero-shot manner. For instance, Huang et al. [2022] prompt language models to produce mid-level plans that can be executed in virtual home environments. Similarly, Ahn et al. [2022] propose SayCan, where an LLM scores possible subgoals based on a given instruction, and these scores are combined with a learned affordance model to identify feasible next actions. In the domain of digital agents, Yao et al. [2022] introduced ReAct, which combined chain-of-thought reasoning with action execution in a loop, enabling LLMs to act as shopping agents on the web.

These approaches share a common strategy: they repurpose the ungrounded world knowledge of LLMs through clever prompting to generate executable plans or action sequences. While promising, such zero-shot methods suffer from two main limitations. First, they can be unreliable, hallucinating steps and producing actions misaligned with the current state. Second, they often assume access

to a predefined library of mid-level skills, which may not be available in more open-ended or under-instrumented environments.

To address these limitations, recent work has begun to explore fine-tuning LLMs using standard behavioral cloning or reinforcement learning techniques in grounded settings. However, this introduces a new challenge: the lack of large-scale, high-quality datasets of demonstrations.

1.4 This Thesis

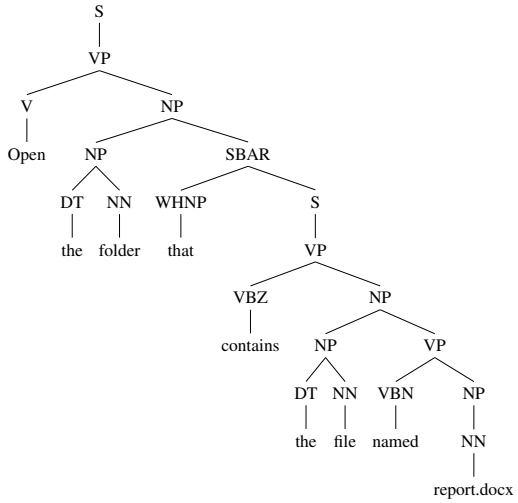
In the previous section, I argue that compositionality and grounding are both essential for building effective instruction-following agents. Yet, while large language models (LLMs) have become powerful tools for a broad range of natural language processing tasks, it remains an open question whether they can reason about complex language structures in the same way as the explicitly compositional models of the 2000s. Furthermore, these models are *primarily* trained on web text—data that contains only a small fraction of explicit grounding, and even then, it's not tailored to the kinds of digital environments instruction-following agents must navigate. This raises the critical question: how can we develop post-training methods that specifically instill environment-relevant grounding into LLMs?

The challenges of language understanding and environment interaction are not merely technical hurdles for building intelligent agents. Rather, they reflect deep philosophical questions about how language users acquire meaning. Wittgenstein [1953] argues that understanding language is inseparable from its use in goal-directed, situated activity. Meaning does not emerge from static definitions but from use within a shared world—famously encapsulated in the quote *in most cases, meaning is use*. According to this view, language is not an abstract symbol system but a tool whose significance arises through interaction. If we are to build systems that truly understand language, we must move beyond the passive training paradigm that dominates LLM development today and instead, immerse them in interactive, feedback-rich environments. This thesis adopts this perspective: it seeks to bridge the gap between language and use for LLMs by building a *learning-from-interaction pipeline*.

1.4.1 Chapters in the Thesis

The first part of this thesis seeks to answer the question: *Are LLMs capable of compositional generalization?*. While many methods have been proposed to uncover syntactic structure in transformers, they do not directly reveal whether the *meaning composition function* implemented by a transformer

resembles these syntax trees. To illustrate, consider the sentence $x =$ *Open the folder that contains the file named report.docx*. A constituency parse reveals how nested structures (like the relative clause "that contains the file named report.docx") modify the main noun phrase "the folder," showing how the meaning of the instruction is systematically built:



A transformer encoder operating on this input produces contextual vectors $\{h_{\text{the}}, h_{\text{folder}}, \ldots, h_{\text{named}}, h_{\text{report.docx}}\}$. How do we know if the transformer is combining these contextual vectors by following the rules of syntax? More concretely, does the representation for the constituent phrase *the file named report.docx*—built from $\{h_{\text{the}}, h_{\text{file}}, h_{\text{named}}, h_{\text{report.docx}}\}$—reflect the same hierarchical structure as the corresponding subtree in the syntax parse?

One influential line of work seeks to address this by training linear probes, simple neural networks that takes hidden states as inputs and output corresponding parses as outputs. These works argue that if such a probe can be trained to obtain a high-enough accuracy on some test set, it implies the existence of syntactic meaning composition in transformers. I argue that Probing only reveals that the final representations contain enough information to reconstruct the syntax, but it does not capture how the transformer incrementally composes meaning during the forward pass. In other words, probes can find correlations between representations and tree structures, but they do not show that the underlying composition function itself has a tree-like structure. I argue that this is a fundamentally different notion—one that requires new tools.

Chapter 2 introduces *Tree Projection*, a method to directly fit a tree-structured function to a transformer's output on a small corpus. The resulting goodness-of-fit score quantifies how closely the transformer's meaning composition resembles a tree structure. We show that this measure

is a strong predictor of compositional generalization performance across a range of grounded instruction-following datasets. This work first appeared at ICLR 2023 in *Characterizing Intrinsic Compositionality in Transformers with Tree Projections* [Murty et al., 2023c]

Building on this, Chapter 3 uses Tree Projections to uncover a new grokking [Power et al., 2022] phenomenon in transformer models, where out-of-distribution performance continues to improve long after in-domain accuracies saturate—coinciding with an increase in the tree projection score. This work was first presented at ACL 2023 in *Grokking of Hierarchical Structure in Vanilla Transformers* [Murty et al., 2023a].

While results in these chapters results that transformers have a good inductive bias for learning compositional structure, prior work reveal fundamental limitations: transformers lack the *representational capacity* for recursion. To address this limitation, Chapter 4 introduces *Pushdown Layers*, a new self-attention mechanism that augments transformers with a stack data structure. Much like programming languages use stacks to implement recursion, Pushdown Layers use stacks to track recursive state for linguistic constituents—pushing and popping tokens to form nested structures as the language model processes text. The stack features themselves softly bias self-attention, enabling transformers to more naturally handle recursive phenomena in both formal as well as natural languages. This work was published at EMNLP 2024 in *Pushdown Layers: Encoding Recursive Structure in Transformer Language Models* [Murty et al., 2023b].

The second part of this thesis turns to the question of grounding these fundamentally ungrounded models in digital environments where they must act. We present methods for injecting grounding through direct environment interaction, using demonstration data obtained via autonomous model exploration. Chapter 5 introduces BAGEL, a method that converts a seed set of randomly explored trajectories or synthetic instructions into demonstrations via round-trips between two noisy components: a labeler module, which converts a trajectory into a synthetic instruction, and a zero-shot LLM agent, which maps the synthetic instruction into a refined trajectory. By iteratively performing these round-trips, BAGEL quickly converts the initial distribution of trajectories toward those that are well-described by natural language. These demonstrations are then used to adapt a zero-shot LLM agent at test time via in-context learning over retrieved demonstrations. We find that BAGEL improves performance by 2–13% absolute on ToolQA and MiniWob++. This work was published in ICML 2024 as *BAGEL: Bootstrapping Agents by Guiding Exploration with Language* [Murty et al., 2024].

Finally, Chapter 6 presents `NNetNav` [Murty et al., 2025], a method for unsupervised interaction with websites that generates synthetic demonstrations for training browser agents. Unlike BAGEL,

which is applied primarily to simpler web interfaces, `NNetNav` conducts a much larger-scale data collection effort, producing over 10,000 demonstrations across 20 real websites. Furthermore, while BAGEL does not explicitly address the exponential search space of interactions, `NNetNav` leverages a hierarchical exploration policy that generates trajectory prefixes aligned with meaningful sub-goals. This hierarchical structure allows `NNetNav` to automatically prune infeasible interaction episodes, making the search tractable in more complex, real-world environments. Another key difference is that BAGEL grounds pre-trained LLMs through in-context learning at test time, while `NNetNav` uses its generated demonstrations for direct fine-tuning of LLMs. We show that LLama-3.1-8b fine-tuned on 10k `NNetNav` demonstrations achieves a 16% success rate on WebArena and 35% on WebVoyager, improvements of 15 and 31 percentage points respectively over the zero-shot LLama-3.1-8b baseline, outperforming zero-shot GPT-4 and achieving state-of-the-art performance among unsupervised methods on both benchmarks.

Part I

Compositionality in Transformer-based Sequence Models

Chapter 2

Characterizing Intrinsic Compositionality in Transformers with Tree Projections

2.1 Introduction

Consider the sentence *Jack has more apples than Saturn has rings*, which you have almost certainly never encountered before. Such *compositionally novel* sentences consist of known words in unknown contexts, and can be reliably interpreted by humans. One leading hypothesis suggests that humans process language according to hierarchical tree-structured computation and that such a restricted computation is, in part, responsible for compositional generalization. Meanwhile, popular neural network models of language processing, such as the transformer, can, in principle, learn an arbitrarily expressive computation over sentences, with the ability to route information between any two pieces of the sentence. In practice, when trained on language data, do transformers instead constrain their computation to look equivalent to a tree-structured bottom-up computation?

While generalization tests on benchmarks [Lake and Baroni, 2018, Bahdanau et al., 2019, Hupkes et al., 2019, Kim and Linzen, 2020, among others] assess if a transformer's *behavior* is aligned with tree-like models, they do not measure if the transformer's *computation* is tree-structured, largely because model behavior on benchmarks could entirely be due to orthogonal properties of the dataset [Patel et al., 2022]. Thus, to understand if transformers implement tree-structured computations, the approach we take is based on *directly approximating them* with a separate, tree-structured

computation. Prior methods based on this approach [Andreas, 2019, McCoy et al., 2019] require putatively gold syntax trees, which not only requires committing to a specific theory of syntax, but crucially, may not exist in some domains due to syntactic indeterminacy. Consequently, these methods will fail to recognize a model as tree-like if it is tree-structured according to a different notion of syntax. Moreover, all of these approaches involve an expensive training procedure for explicitly fitting a tree-structured model [Socher et al., 2013, Smolensky, 1990] to the neural network.

This Chapter presents a method that is completely unsupervised (no gold syntax needed) and parameter-free (no neural network fitting needed). At a high level, the proposed method *functionally projects*[1] transformers into the space of all tree-structured models, via an implicit search over the joint space of tree structures and parameters of corresponding tree-structured models (Figure 2.1). The main intuition behind our approach is to appeal to the notion of *representational invariance*: bottom-up tree-structured computations over sentences build intermediate representations that are invariant to outside context, and so we can approximate transformers with a tree-structured computation by searching for a "bracketing" of the sentence where transformer representations of intermediate brackets are maximally invariant to their context. Concretely, the main workhorse of our approach is a subroutine that computes distances between contextual and context-free representations of all spans of a sentence. We use these distances to induce a *tree projection* of the transformer using classical chart parsing (Section 2.3), along with a score that estimates tree-structuredness.

First, we prove that our approach can find the *best* tree-structured account of a transformer's computation under mild assumptions (Theorem 1). Empirically, we find transformer encoders of varying depths become more tree-like as they train on three sequence transduction datasets, with corresponding tree projections gradually aligning with gold syntax on two of three datasets (Section 2.5). Then, we use tree projections as a tool to predict behaviors associated with compositionality: induced trees reliably reflect contextual dependence structure implemented by encoders (Section 2.6.1) and both tree scores as well as parsing F1 of tree projections better correlate with compositional generalization to configurations unseen in training than in-domain accuracy on two of three datasets (Section 2.6.2).

2.2 Background

How can we compute the meaning of *red apples are delicious*? Substantial evidence [Crain and Nakayama, 1987, Pallier et al., 2011, Hale et al., 2018] supports the hypothesis that semantic

[1]We provide a *functional* account of the transformer's computation and not a *topological* account, i.e., we are agnostic to whether the attention patterns of the transformer themselves look tree structured—see Murty et al. [2023c] for examples.

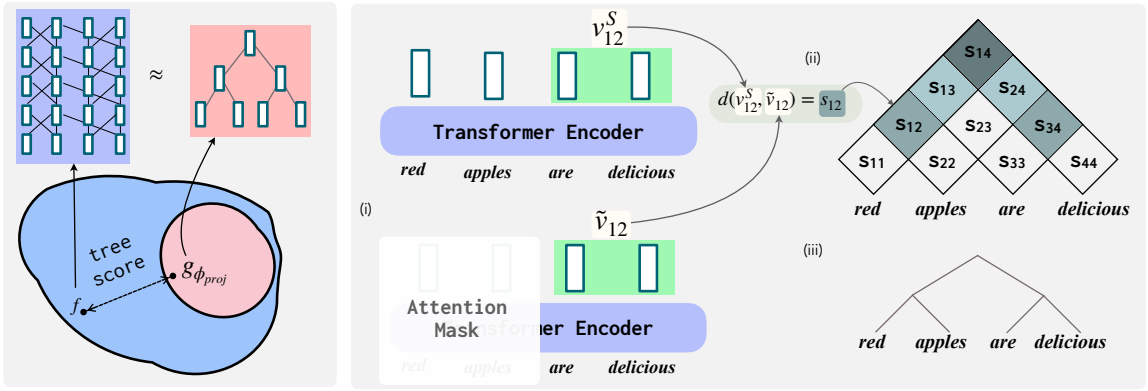


Figure 2.1: (a) Given a transformer model f , our method finds the *tree projection* of f i.e., binary trees corresponding to the tree-structured neural network $g_{\phi_{\text{proj}}}$ (in the space of all tree-structured models) that best approximates the outputs of f on a given set of strings. (b) (i) Given a string, we compute context-free representations ($\tilde{\boldsymbol{v}}_{ij}$) for all spans of the string via attention masking (Section 2.3). (ii) We use the distance between (average-pooled) context-free and contextual representations ($\boldsymbol{v}_{ij}$) to populate a chart data structure. (iii) We decode a tree structure from chart entries.

interpretation of sentences by humans involves a *tree-structured*, hierarchical computation, where smaller constituents (*red*, *apples*) recursively combine into larger constituents (*red apples*), until we reach the full sentence. Concretely, suppose we have a sentence $S \triangleq \{w_1, w_2, \ldots, w_{|S|}\}$. Let T be a function that returns a binary tree for any sentence S, defined recursively as $T(S) \triangleq \langle T(S_{1,j}), T(S_{j+1,|S|})\rangle$ where $T(S_{a,b})$ refers to a subtree over the span $S_{a,b} \triangleq \{w_a, w_{a+1}, \ldots, w_b\}$. We say that a span $S_{a,b} \in T(S)$ if the node $T(S_{a,b})$ exists as a subtree in $T(S)$. For notational convenience, we sometimes use S_l and S_r as the left and right subtrees for $T(S)$ i.e., $T(S) = \langle S_l, S_r\rangle$.

Compositionality in Meaning Representations. While theories of compositional meaning formation might differ on specifics of syntax, at a high-level, they propose that computing the meaning of S must involve a bottom-up procedure along some syntax tree $T(S)$ of the sentence S. Formally, we say that a meaning representation system m is compositional if the meaning $m(s)$ of some expression s is a *homomorphic image* of the syntax of s i.e., $m(s) = \phi(m(s_l), m(s_r))$ for some ϕ following Montague [1970]. Crucially, we note that such a ϕ exists only if $m(s)$ can be fully determined by the contents of s, that is, if $m(s)$ is *contextually invariant*. While there are several phenomena that necessarily require a non-compositional context-sensitive interpretation (indexicals, idioms, pronouns, lexical ambiguity among others), compositional interpretation remains a central component in explanations of the human ability to systematically interpret novel sentences.

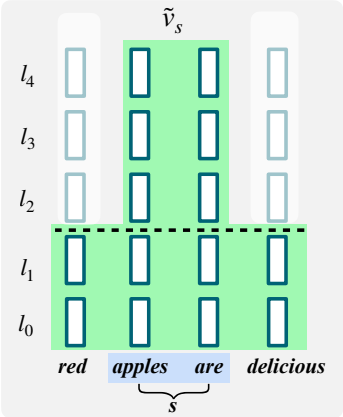


Figure 2.2: We use a T-shaped attention mask with a threshold layer to obtain approximate context-free vectors for transformers.

Compositionality in Neural Models. A class of neural networks that are obviously compositional are tree-structured models such as Socher et al. [2013], that obtain *vector representations* of sentences by performing a bottom-up computation over syntax. Specifically, given S and a corresponding binary tree $T(S)$, the output of the tree-structured network g_ϕ is defined recursively—for any span $p \in T(S)$, $g_\phi(p, T(p)) \triangleq h_\theta(g_\phi(p_l, T(p_l)), g_\phi(p_r, T(p_r))$ where $h_\theta : \mathbb{R}^d \times \mathbb{R}^d \mapsto \mathbb{R}^d$ is some feedforward neural network. For leaf nodes w_i, $g_\phi(w_i, T(w_i)) \triangleq \eta_{w_i}$, where $\eta_w \in \mathbb{R}^d$ represents the word embedding for w. The parameters of the network are $\phi = \{\theta, \eta_{w_1}, \eta_{w_2}, \ldots\}$.

2.3 Our Approach

While tree-structured networks were built to reflect the compositional structure of natural language, they have been superseded by relatively unstructured transformers [Vaswani et al., 2017a]. How can we measure if the *computation* implemented by a transformer is compositional and tree-like? We start by noting that in any bottom-up tree computation over a sentence, representation of an intermediate constituent depends only on the span it corresponds to, while being fully invariant to outside context. Thus, one way to assess tree-structuredness of a computation over some span is to measure *contextual invariance* of the resulting representation. Consequently, we construct a tree-structured approximation of a transformer's computation over a sentence by searching for a bracketing of the sentence where spans have maximal contextual invariance.

2.3.1 Span Contextual Invariance

Suppose f is a transformer model that produces contextual vectors of words in S as $f(S) \triangleq \{\boldsymbol{v}^S_{w_1}, \boldsymbol{v}^S_{w_2}, \ldots, \boldsymbol{v}^S_{w_{|S|}}\}$ where $\boldsymbol{v}^S_w$ is a contextual vector representation of w. Given a span p, let $\boldsymbol{v}^S_p$ be the span representation of the contextual vectors of words in p, $\boldsymbol{v}^S_p = \sum_{w \in p} \boldsymbol{v}^S_w$. Similarly, let $\tilde{\boldsymbol{v}}_p$ be a *context-free* representation of the span p. For transformers, we obtain context-free representations through a simple attention masking scheme. In particular, to obtain $\tilde{\boldsymbol{v}}_p$, we apply a "T-shaped" attention mask and take the pooled representation of the words in p at the final layer (Figure 2.2). The mask ensures that attention heads do not attend to tokens outside of p *after an optional threshold layer*[2]

We define span contextual invariance (SCI) of a span p in the sentence S as $\text{SCI}(S, p) \triangleq d(\boldsymbol{v}^S_p, \tilde{\boldsymbol{v}}_p)$ for some distance function d. Similarly, we define the cumulative SCI score for a tree T to be:

$$\text{SCI}(S, T) \triangleq \sum_{s \in T} d(\boldsymbol{v}^S_p, \tilde{\boldsymbol{v}}_p). \tag{2.1}$$

2.3.2 Computing Tree Projections by minimizing SCI

Consider the collection of strings, $\mathcal{D} = \{(S)\}$, and some function T that produces binary trees for any $S \in \mathcal{D}$. The cumulative error from approximating outputs of the transformer f with outputs of a tree-structured network g_ϕ structured according to T can be written as

$$\mathcal{L}(f, g_\phi, T) \triangleq \sum_{S \in \mathcal{D}} \sum_{p \in T(S)} d(g_\phi(p, T(p)), \boldsymbol{v}^S_p). \tag{2.2}$$

Suppose we are interested in finding the best tree-structured approximation to f over all possible trees i.e. a configuration of tree structures and corresponding model parameters that best approximate the transformer's behavior. We define this as the *exact tree projection* of f,

$$\phi_{\text{proj}}, T_{\text{proj}} \triangleq \arg\min_{\phi, T} \mathcal{L}(f, g_\phi, T). \tag{2.3}$$

Theorem 1. $\min_{\phi,T} \mathcal{L}(f, g_\phi, T) \leq \sum_{S \in \mathcal{D}} \min_{T(S)} \text{SCI}(S, T(S))$. *In other words, the best tree structured approximation to f has an error upper bounded by cumulative* SCI *scores.*

[2]This procedure outputs vectors that are entirely context-free only if the threshold is exactly 0, but we find that tuning the threshold layer often leads to significantly better induced parses.

In general, finding tree projections involves a joint search over all discrete tree structures $T(S)$ as well as over continuous parameters ϕ, which is intractable. However, we substantially simplify this search using Theorem 1, since the upper bound depends only on parses $T(S)$ and properties of the transformer, and can be exactly minimized for a given f in polynomial time, with efficient parsing algorithms. We minimize this upper bound itself to approximately recover the best tree-structured approximation to f, over all choices of trees and parameters. The output of this minimization is an approximate tree projection,

$$\widehat{T}_{\text{proj}}(S) = \arg\min_{T(S)} \text{SCI}(S, T(S)) \tag{2.4}$$

for every $S \in \mathcal{D}$. Under a mild assumption[3], SCI minimization leads to tree projections *exactly*.

Assumption 1. *Let S_p denote the collection of sentences that contain the span p. Then, for every span p, we have $\min_{\boldsymbol{v}} \sum_{S \in S_p} d(\boldsymbol{v}_p^S, \boldsymbol{v}) = \sum_{S \in S_p} d(\boldsymbol{v}_p^S, \tilde{\boldsymbol{v}}_p)$. That is, context-free vectors minimize the cumulative distance to their contextual counterparts.*

Corollary 1.1. *Under Assumption 1, $\min_{\phi,T} \mathcal{L}(f, g_\phi, T) = \sum_{S \in \mathcal{D}} \min_{T(S)} \text{SCI}(S, T(S))$. Moreover, $T_{proj}(S) = \arg\min_{T(S)} \text{SCI}(S, T(S))$ for any $S \in \mathcal{D}$.*

2.3.3 Measuring Intrinsic Compositionality

SCI minimization provides two natural ways to measure intrinsic compositionality of f on $\mathcal{D}$. To measure tree-structuredness, we use

$$t_{\text{score}} \triangleq \frac{\sum_{S \in \mathcal{D}} \mathbb{E}_T \text{SCI}(S, T) - \text{SCI}(S, \widehat{T}_{\text{proj}}(S))}{|\mathcal{D}|}, \tag{2.5}$$

which computes the averaged SCI score of induced trees, normalized against the expected SCI score under a uniform distribution over trees. We find normalization to be necessary to prevent our method from spuriously assigning high tree-structuredness to entirely context-free encoders (that have high SCI scores for *all* trees). When gold syntax T_g is available, we use

$$t_{\text{parseval}} \triangleq \text{PARSEVAL}(\widehat{T}_{\text{proj}}, T_g, \mathcal{D}), \tag{2.6}$$

to measure bracketing F1 score (PARSEVAL; Black et al. [1991]) score of $\widehat{T}_{\text{proj}}$ against T_g on $\mathcal{D}$.

[3]Figure A.1 shows that this assumption approximately holds in practice.

2.4 Experimental Setup

Our experiments[4] are organized as follows. First, we show that on 3 sequence transduction tasks, transformers of varying depths become more tree-like over the course of training, and sometimes learn tree projections that progressively evolve towards ground truth syntax. Then, we show how tree projections can be used to assess various model behaviors related to compositionality.

Datasets. We consider three datasets (Table 2.1) commonly used for benchmarking compositional generalization: COGS [Kim and Linzen, 2020], M-PCFGSET [Hupkes et al., 2019] and GeoQuery [Zelle and Mooney, 1996]. COGS consists of automatically generated sentences from a context-free grammar paired with logical forms, split into in-domain examples (for training) and a compositionally challenging evaluation set. M-PCFGSET is a slightly modified version [5] of PCFGSET [Hupkes et al., 2019], where inputs are a nested sequence of expresssions that specify a unary or binary operation over lists. The objective is to execute the function specified by the input to obtain the final list. We focus on the "systematicity split" for measuring compositional generalization. Finally, GeoQuery consists of natural language queries about US geography paired with logical forms. To measure compositional generalization, we use the "query" split from Finegan-Dollak et al. [2018].

Implementation Details. We use greedy top down chart parsing to approximately minimize SCI. In particular, we use SCI scores for all $O(|S|^2)$ spans of a string S to populate a chart data structure, which is used to induce a tree by minimizing SCI via a top down greedy procedure (see Murty et al. [2023c] for more details), similar to Stern et al. [2017a]. Our procedure outputs a tree and simultaneously returns normalized SCI score of the tree, computing a sampling estimate of expected SCI score (Equation 2.5). We train transformer encoder-decoder models with encoders of depths {2, 4, 6} and a fixed decoder of depth 2. We omit 6-layer transformer results for GeoQuery as this model rapidly overfit and failed to generalize, perhaps due to the small size of the dataset. We choose a shallow decoder to ensure that most of the sentence processing is performed on the encoder side. We train for 100k iterations on COGS, 300k iterations on M-PCFGSET and 50k iterations on GeoQuery. We collect checkpoints every 1000, 2000 and 500 gradient updates and use the encoder at these checkpoints to obtain parses as well as tree scores. In all experiments, d is cosine distance i.e., $d(\boldsymbol{x}, \boldsymbol{y}) = 1 - \frac{\boldsymbol{x}^\top \boldsymbol{y}}{\|\boldsymbol{x}\|\|\boldsymbol{y}\|}$. All transformer layers have 8 attention heads and a hidden dimensionality of 512. We use a learning rate of 1e-4 (linearly warming up from 0 to 1e-4 over 5k steps) with the

[4] Code and data are be available at https://github.com/MurtyShikhar/TreeProjections

[5] see Murty et al. [2023c] for details.

AdamW optimizer. All accuracies refer to exact match accuracy against the gold target sequence. For all seq2seq transformers, we tune the threshold layer based on t_{parseval}.

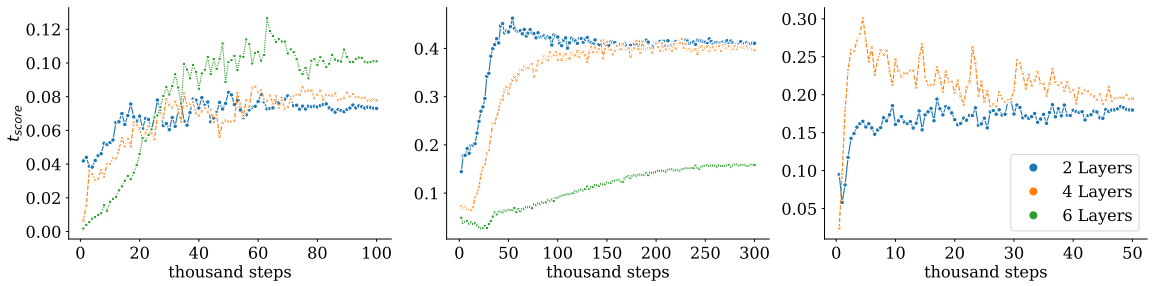
	Inputs	Outputs
i.	*The ball was found* *A cookie was blessed*	`ball(`x_1`) AND find.theme(`x_3, x_1`)` `cookie(`x_1`) AND bless.theme(`x_3, x_1`)`
ii.	*copy interleave_second reverse shift H13 C19 H9 O20* *repeat interleave_second interleave_first S1 E3 W3 N11 H4 Y3*	`H9 H13 O20 C19` `L8 E1 R13 T12 E1 T12 L8 E1 R13 T12 E1 T12`
iii.	*Which state has the lowest population density?* *What is the population density of Wyoming?*	`(A, _smallest(B, (_state(A), _density(A, B))))` `(A, (_density(B, A), _const(B, _stateid(wyoming))))`

Table 2.1: Example (x, y) pairs from COGS (i), M-PCFGSET (ii) and GeoQuery (iii). See Murty et al. [2023c] for more details on pre-processing as well as dataset statistics.

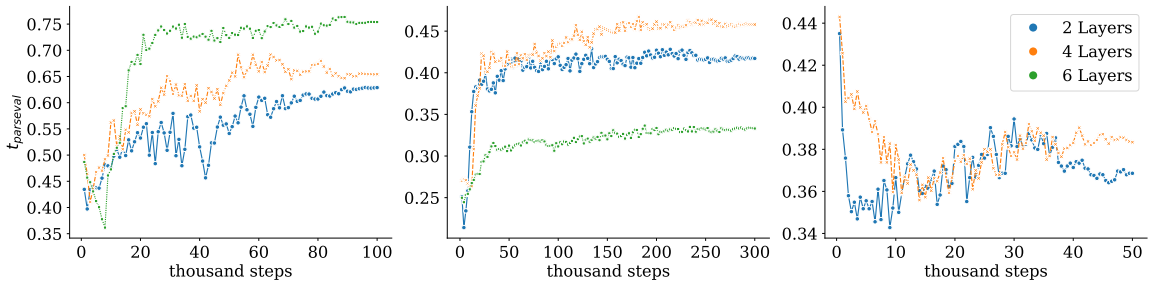
2.5 Trained Transformers implement a tree-like computation

How does intrinsic compositionality of a transformer encoder evolve during the course of training on sequence transduction tasks? To study this, we plot t_{score} (*how tree-like is a model?*) and t_{parseval} (*how accurate is the tree projection of a model?*) of encoder checkpoints throughout training. As a comparison, we track how well a supervised probe recovers syntax from encoders—that is, we train a 1 layer transformer decoder to autoregressively predict linearized *gold* parse trees of S from transformer outputs $f(S)$ at various points of training, and measure the PARSEVAL score of probe outputs (p_{parseval}) on a test set.

Results. We plot t_{parseval} and t_{score} over the course of training in Figure 2.3. *We observe that 7/8 encoders gradually become more tree-like* i.e., increase t_{score} over the course of training, with the 4 layer transformer on GeoQuery being the exception. Interestingly, we note that t_{parseval} also increases over time for all encoders on COGS and M-PCFGSET suggesting that the *tree projection of trained transformers progressively becomes more like ground-truth syntax*. In other words, all encoders trained on COGS and M-PCFGSET learn a computation that is gradually more "syntax aware". Can supervised probing also reveal this gradual syntactic enrichment? We plot PARSEVAL score of parse trees predicted by the probe on held out sentences (p_{parseval}) in Figure 2.4—while p_{parseval} does improve over time on both COGS and M-PCFGSET, we observe that all checkpoints after some threshold have similar probing accuracies. We quantitatively compare gradual syntactic enrichment by computing the Spearman correlation between t_{parseval} (p_{parseval}) and training step and find that $\rho_{p_{\text{parseval}}}$ is significantly smaller than $\rho_{t_{\text{parseval}}}$ for both datasets. Interestingly, we also find that our unsupervised procedure is able to produce better trees than the *supervised* probe on M-PCFGSET as



(a) Normalized Tree Scores for COGS, M-PCFGSET and GeoQuery (↑ is better).

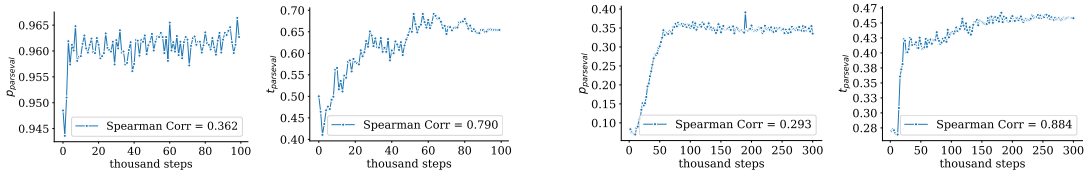


(b) Parsing Accuracies for COGS, M-PCFGSET and GeoQuery (↑ is better).

Figure 2.3: We plot t_{score} and t_{parseval} by computing approximate tree projections at various checkpoints. 7/8 models become more tree-structured (increased t_{score}) and all models on COGS and M-PCFGSET learn tree projections that gradually align with ground truth syntax (increased t_{parseval}).

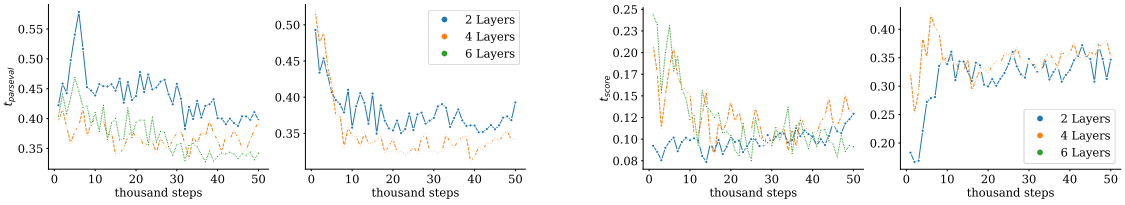
observed by comparing p_{parseval} and t_{parseval}. Overall, we conclude that supervised probing is unable to discover latent tree structures as effectively as our method.

How does supervisory signal affect compositionality? Could a purely self-supervised objective (i.e., no output logical form supervision) also lead to similar emergent tree-like behavior? To test this, we experiment with training the transformer encoder with a masked language modeling objective, similar to Devlin et al. [2019] for COGS and GeoQuery. Concretely, for every S, we mask out 15% of input tokens and jointly train a transformer encoder and a 1 layer feedforward network, to produce contextual embeddings from which the feedforward network can decode word identities for masked out words. As before, we collect checkpoints during training and plot both t_{parseval} and t_{score} over time in Figure 2.5. We find that t_{parseval} does not improve over time for any of the models. Additionally, we find that t_{score} increases for all models on GeoQuery, but only for the 2 layer model on COGS. Taken together, these results suggest that under the low data regime studied here, transformers trained with a self-supervised objective do not learn tree-structured computations.



(a) COGS (b) M-PCFGSET

Figure 2.4: We plot p_{parseval} and t_{parseval} over time for the 4 layer transformer encoder on COGS and M-PCFGSET. We find that t_{parseval} improves gradually over time suggesting that the model becomes more "syntax aware". Such gradual syntax enrichment is not uncovered well by the probe since all checkpoints after 4000 (for COGS) and 50000 (for M-PCFGSET) iterations have similar p_{parseval}.



(a) Parsing Accuracies (b) Normalized Tree Scores

Figure 2.5: We plot t_{parseval} and t_{score} at various checkpoints for models trained with a masked language modeling objective on COGS (first) and GeoQuery (second). Only 2/5 models become tree-structured and none learn tree projections aligned with gold syntax, suggesting that self-supervision may fail to produce tree-like computation in a relatively low data regime.

2.6 Tree Projections and Model Behavior

Given S, and corresponding contextual vectors $f(S)$, the *contextual dependence structure* captures the dependence between contextual vectors and words in S i.e., how much does $\boldsymbol{v}^S_{w_i}$ change when w_j is perturbed to a different word. Contextual dependence structure is important for assessing compositional behavior. For instance, consider the span p = *red apples* appearing in some sentences. If the contextual vectors for p have large dependence on outside context, we expect the model to have poor generalization to the span appearing in *novel contexts* i.e., poor compositional generalization.

We first show that tree projections reflect the contextual dependence structure implemented by a transformer. Next, we show that both t_{score} and t_{parseval} are better predictors of compositional generalization than in-domain accuracy.

2.6.1 Induced trees correspond to Contextual dependence structure

Intuitively, greedily decoding with a SCI populated chart makes split point decisions where resulting spans are maximally invariant with one other. Thus, for a given constituent c and a word $w \in c$, we expect $\boldsymbol{v}^S_w$ to depend more on words within the same constituent than words outside the constituent. Thus, we compare the change in $\boldsymbol{v}^S_w$ when another word inside c is perturbed (*in-constituent* perturbations) to the change when a word outside c is perturbed (*out-of-constituent* perturbations), where word perturbations are performed by adding gaussian noise to corresponding word vectors in layer 0 (see Figure 2.6). We ensure that both perturbations are made to words at the same *relative distance* from w. As a control, we also compute changes to $\boldsymbol{v}^S_w$ when perturbations are made with respect to constituents from random trees.

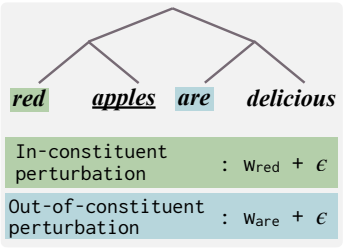


Figure 2.6: For word w (*apples*) in constituent c, an in-constituent perturbation adds noise $\epsilon \sim \mathcal{N}(0, 0.01)$ to another word's vector within c (*red*) while an out-of-constituent perturbation adds noise to a word vector at same relative distance outside c (*are*).

Setup and Results. We sample 500 random inputs from each of COGS, M-PCFGSET and GeoQuery and consider encoders from all transformer models. We obtain the *mean* L_2 distance between the contextual vector of w in the original and perturbed sentence for in-constituent perturbations (Δ_{ic}) and out-of-constituent perturbations (Δ_{oc}) and plot the relative difference between the two

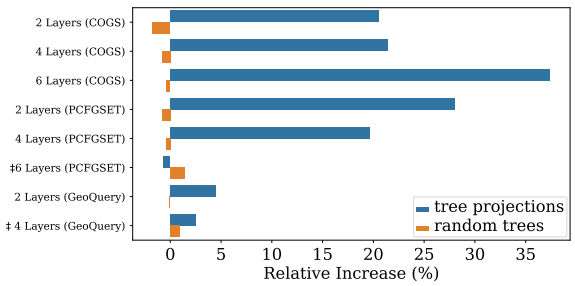


Figure 2.7: We measure the mean L_2 distance in the contextual vector of words when *in-constituent* and *out-of-constituent* words are perturbed. We plot the relative difference between Δ_{ic} and Δ_{oc} when constituents are obtained from tree projections (in blue). As a control, we also compute Δ_{ic} and Δ_{oc} when constituents are chosen from random trees (in orange). For all models except those marked with ‡, *in-constituent* perturbations lead to significantly (as measured by a t-test, $p < 10^{-5}$) larger change to contextual vectors compared to *out-of-constituent* perturbations.

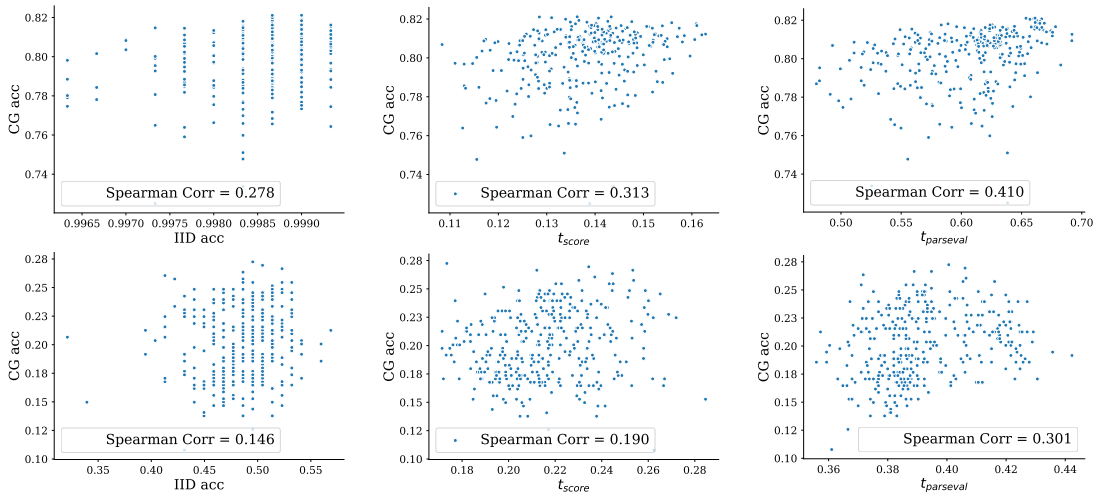
in Figure 2.7. For 6/8 models, in-constituent perturbations result in larger L_2 changes than out-of-constituent perturbations (statistically significant according to a two-sided t-test, $p < 10^{-4}$). Meanwhile, when constituents are chosen according to random trees, changes resulting from both perturbations are similar. Overall, this suggests that *induced trees reflect the contextual dependence structure learnt by a transformer*.

2.6.2 Tree-structuredness correlates better with generalization than in-domain accuracy

We study the connection between compositionality and generalization for the 4 layer transformer encoder on COGS and GeoQuery [6]. On each dataset, we train the model with 5 different random seeds and collect checkpoints every 1000/500 iterations. For each checkpoint, we measure accuracy on the in-domain validation set (*IID acc*) and accuracy on the out-of-domain compositional generalization set (*CG acc*). Additionally, we also compute t_{parseval} and t_{score} for the encoders at each of these checkpoints. To measure the relationship between compositionality and generalization, we compute the Spearman correlation between t_{parseval} (t_{score}) and *CG acc* and denote that as $\rho^{\text{CG}}_{t_{\text{parseval}}}$ ($\rho^{\text{CG}}_{t_{\text{score}}}$). As a comparison, we also compute the correlation between *IID acc* and *CG acc* ($\rho^{\text{CG}}_{\text{IID}}$).

Results. We plot the relationship between various properties and generalization along with corresponding correlations in Figure 2.8. In general, we expect both *IID acc* and *CG acc* to improve

[6] IID acc perfectly predicts generalization for M-PCFGSET so we omit it in these experiments.



(a) IID acc vs CG acc (b) t_{score} vs CG acc (c) t_{parseval} vs CG acc

Figure 2.8: We plot the Spearman correlation between (a) *IID acc* and *CG acc*, (b) t_{score} and *CG acc*, (c) t_{parseval} and *CG acc*. We find that both t_{parseval} and t_{score} correlate better with generalization than in-domain accuracy. All correlations are statistically significant (p-values $< 10^{-3}$)

.

together over time, and so it is unsurprising to see that $\rho^{\text{CG}}_{\text{IID}} > 0$. Moreover, for COGS, both t_{parseval} and t_{score} increase over time, and so it is expected that both $\rho^{\text{CG}}_{t_{\text{parseval}}}$ and $\rho^{\text{CG}}_{t_{\text{score}}}$ are positive. Crucially, however, we find that both $\rho^{\text{CG}}_{t_{\text{parseval}}}$ and $\rho^{\text{CG}}_{t_{\text{score}}}$ are greater than $\rho^{\text{CG}}_{\text{IID}}$ on both COGS and GeoQuery. Thus, tree-like behavior (t_{score}) as well as the *right* tree-like behavior (t_{parseval}) are better predictors of compositional generalization than in-domain accuracy. This result gives simple *model selection* criteria to maximize CG accuracy in the absence of a compostional generalization test set (true for most practical scenarios)—given a collection of checkpoints with similar in-domain accuracies, choose the checkpoint with highest t_{score} or t_{parseval} (if syntactic annotations are available) to get the model with best generalization behavior, in expectation.

2.7 Related Work

Measuring Linguistic Structure. A common analysis tool for assessing a model's competence in a specific linguistic phenomenon is *behavioral testing* [Linzen et al., 2016, Marvin and Linzen, 2018, Ribeiro et al., 2020], where the model's performance on a curated test set is used as the measure of competence. Widely used in prior work to assess compositionality of neural models [Lake and Baroni, 2018, Bahdanau et al., 2019, Yu and Ettinger, 2020], behavioral tests are inherently *extrinsic*, since they are agnostic to whether the model implements an appropriately constrained, tree-like

computation. While most prior approaches for assessing intrinsic compositionality [Andreas, 2019, McCoy et al., 2019] require putatively gold syntax trees, our proposed approach does not require any pre-determined ground truth syntax, since we search over the space of *all* possible trees to find the best tree structure that approximates a transformer's computation.

Tree-structured Neural Networks. Inspired by the widely accepted belief that natural language is mostly tree-structured [Chomsky, 1957], there have been several attempts to construct tree shaped neural networks for various NLP tasks, such as Recursive Neural Networks [Socher et al., 2013], Tree RNNs [Tai et al., 2015a], Recurrent Neural Network Grammars [Dyer et al., 2016], Neural Module Networks [Andreas et al., 2016], Ordered Neuron [Shen et al., 2019] among others. These approaches have largely been superseded by transformers [Vaswani et al., 2017a], often pre-trained on a large corpus of text (Devlin et al. [2019], *inter alia*). We show that transformers, though not explicitly tree-structured, may still learn to become tree-like when trained on language data.

Invariances and Generalization. The general problem of studying model performance under domain shifts has been widely studied under domain generalization [Blanchard et al., 2011]. When domain shift is a result of changing feature covariates only, an effective strategy for domain generalization is to learn *domain invariant representations* [Muandet et al., 2013, Ganin et al., 2016]. We apply the notion of domain invariance in the context of compositional generalization, and posit that models that produce span representations that are more contextually invariant can generalize better to inputs where the span appears in a novel context, which is precisely the motivation behind SCI.

2.8 Conclusion

When trained on language data, how can we know whether a transformer learns a compositional, tree structured computation as is hypothesized to underlie human language processing? While extrinsic behavioral tests only assess if the model is capable of the same generalization capabilities as those expected from tree-structured models, this work proposes an *intrinsic* approach that directly estimates how well a parametric tree-structured computation approximates the model's computation. Our method is unsupervised and parameter-free and provably upper bounds the representation building process of a transformer with any tree-structured neural network, effectively providing a *functional projection* of the transformer into the space of all tree structured models. The central conceptual notion in this chapter is *span contextual invariance* (SCI) that measures how much the contextual

representation of a span depends on the context of the span vs. the content of the span. SCI scores of all spans are plugged into a standard top-down greedy parsing algorithm to induce a binary tree along with a corresponding tree score. From experiments, we show that tree projections uncover interesting training dynamics that a supervised probe is unable to discover—we find that on 3 sequence transduction tasks, transformer encoders tend to become *more tree-like* over the course of training, with tree projections that become *progressively closer to true syntactic derivations* on 2/3 datasets. We also find that tree-structuredness as well as parsing F1 of tree projections is a better predictor of generalization to a compositionally challenging test set than in-domain accuracy i.e., given a collection of models with similar in-domain accuracies, select the model that is most tree-like for best compositional generalization. Overall, our results suggest that making further progress on human-like compositional generalization might require inductive biases that encourage the emergence of latent tree-like structure.

Discussion. Tree Projections has since been applied in several follow-up studies to better understand and improve the compositional structure in transformers. Singh et al. [2023] used it to analyze compositionality in vision-language models, while in the next chapter, we show how Tree Projections enabled the discovery of a new grokking phenomenon in transformers. Nandi et al. [2025] built on these ideas to develop a regularizer that directly intervenes in the training of 1B+ parameter language models, leading to improved out-of-distribution perplexities and better syntactic generalization. syntactic generalization. This work is particularly relevant given recent questions about the validity of syntactic probes: while probes can reveal the presence of syntactic structure, this does not guarantee that the model can actually use it. Consequently, methods that correlate compositional structure with downstream behavior, like those discussed here, are becoming essential tools for analyzing LLMs.

Chapter 3

Grokking of Hierarchical Structure in Vanilla Transformers

3.1 Introduction

Although human language is produced as a linear sequence, it is hierarchically organized. Smaller units compose to form larger constituents. The ability to infer this hierarchical structure underlies our ability to produce and understand new sentences [Chomsky, 1965, Crain and Nakayama, 1987]. In this paper, we investigate whether standard neural transformer models Vaswani et al. [2017b] can also generalize hierarchically when trained on language processing tasks (Fig 3.1). Our main finding is that hierarchical generalization in transformers does occur, but very slowly: performance on structurally novel sentences increases gradually, long after performance on sentences from the training distribution has plateaued. We term this phenomenon *structural grokking*, by analogy to existing findings on simple classification tasks [Power et al., 2022].

On two datasets, we show that structural grokking exhibits inverted U-shaped scaling behavior as a function of model depth: hierarchical generalization improves, then declines, as we train deeper models. Prior work suggests that a number of model-internal properties might track the emergence of hierarchical structure in transformers, including weight norms [Merrill et al., 2021, Liu et al., 2022b, Power et al., 2022], attention sparsity [Merrill et al., 2021], and functional tree-structuredness (from Chapter 2). We find that functional tree-structuredness is uniquely able to predict structural grokking—while weight norms and attention sparsity increase monotonically in model depth, tree-structuredness is highest for models of the optimal depth for structural grokking.

Our results challenge findings from prior work [Mueller et al., 2022, Petty and Frank, 2021]

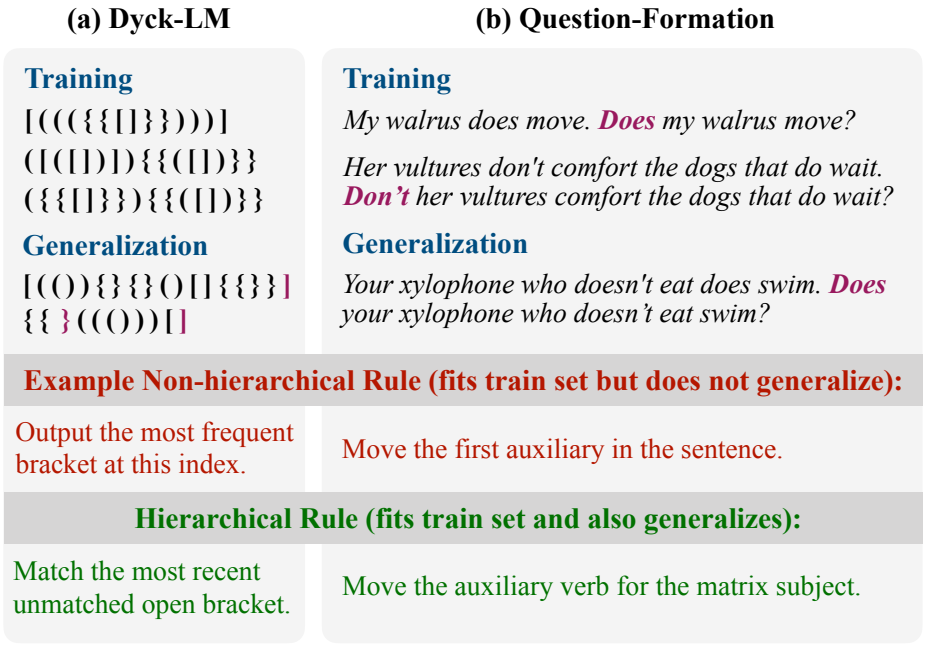


Figure 3.1: Examples from language modeling datasets we use to assess hierarchical generalization in vanilla transformers. These datasets are constructed so that both a non-hierarchical as well as a hierarchical rule can perfectly fit the training set, but only the hierarchical rule generalizes to structurally novel inputs.

claiming that ordinary transformers completely fail on the tests of hierarchical generalization that we study. We attribute these failures to early stopping based on in-domain validation performance, which significantly underestimates hierarchical generalization due to structural grokking. On the datasets where this prior work reports generalization accuracies below 20%, *simply by training for longer*, mean accuracy across random seeds reaches 80%, and several seeds achieve near-perfect generalization performance. Past findings are also partially explained by U-shaped scaling: this work uses models that are too shallow [Mueller et al., 2022, Petty and Frank, 2021] or too deep [Mueller et al., 2022]. Our results align with past findings on the role of extended training in other language processing problems [Csordás et al., 2021, Hoffmann et al., 2022].

3.2 Background

Transformers Given a sequence of tokens $w_{\leq i} = w_1, w_2, \ldots, w_i$, where each token is drawn from a fixed vocabulary V, an L-layer transformer language model (LM) f_θ^L outputs a distribution over the next token $w_{i+1} \in V$, $f_\theta^L(w_{\leq i}) \in \mathbb{R}^{|V|}$. A key part of the architecture is a sequence of L

self-attention layers, where layer p computes contextual vectors of token k as a non-linear parametric function of a convex combination of contextual vectors of tokens $w_{\leq k}$ from the previous layer, where coefficients $\mathbf{a}_k^p \in \mathbb{R}^k$ are known as the *attention distribution*. The LM weights are learned by maximizing the log probability of the correct continuation w_{k+1}, given prefix $w_{\leq k}$.

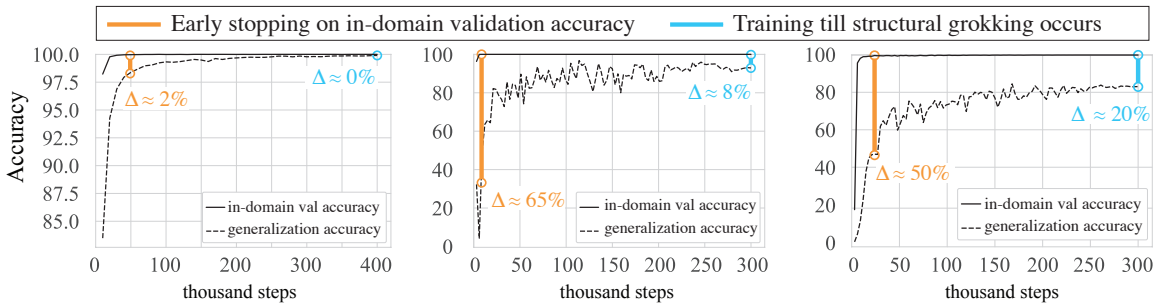
Hierarchical structure in transformers While unsupervised pre-training of transformers has led to state-of-the-art transfer learning results across NLP, the architecture itself has been claimed to lack human-like inductive biases toward hierarchical structure [Tran et al., 2018, Hahn, 2020, Petty and Frank, 2021, Mueller et al., 2022]. We revisit these claims in this work.

To understand whether a given model has a bias for acquiring hierarchical structure, we follow McCoy et al. [2020] and evaluate generalization in models trained on ambiguous tasks in which training data is consistent with both a "hierarchical rule" as well as a "non-hierarchical rule" (Fig 3.1). To test if the hierarchical rule has been acquired, we test generalization on a separate out-of-distribution test set, constructed such that only learners that have acquired the hierarchical rule are successful.

Grokking Power et al. [2022] identify the phenomenon of *grokking* on small algorithmic datasets where they find that test performance improves long after training performance has saturated. We hypothesize a similar *structural grokking*, where the model groks hierarchical structure long after in-domain validation performance has saturated, and consequently, hierarchical generalization can continue to improve with extended training.

3.3 Experiments

Datasets Since our goal is to understand hierarchical generalization in transformers, we use two datasets from [McCoy et al., 2020] and additionally evaluate on a simple bracket-tracking task. For *Dyck*, models are trained to predict next tokens in strings drawn from $\text{Dyck}_{20,10}$, the language of well-nested brackets with 20 types and max nesting depth of 10. We evaluate generalization to structurally unobserved strings in $\text{Dyck}_{20,10}$ (see Fig 3.1 for examples and Murty et al. [2023a] for details). For the McCoy et al. [2020] datasets, in *Question-Formation*, models must convert English sentences into questions and, in *Tense-Inflection*, models must map from sentences and tense markers to appropriately re-inflected sentences. We evaluate generalization on the out-of-distribution test set from McCoy et al. [2020].



(a) Dyck (6 layers) (b) Question-Formation (6 layers) (c) Tense-Inflection (4 layers)

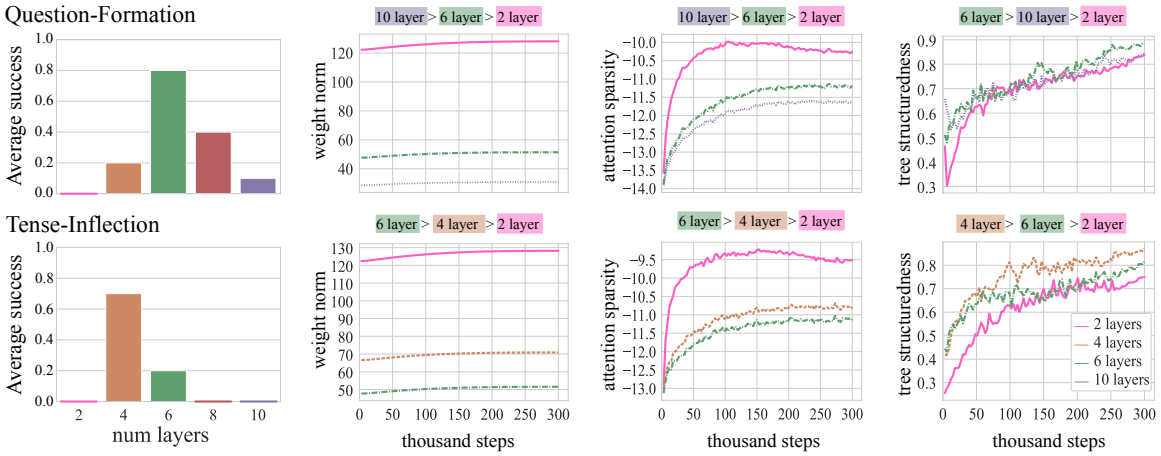
Figure 3.2: Average accuracy across 10 random seeds on the in-domain val set (solid) and generalization set (dashed) for all datasets. Generalization performance improves even after in-domain accuracies have saturated, showing *structural grokking*. We highlight with orange and blue lines the gap between in-domain and generalization accuracies at the point of early stopping based on the in-domain val set performance vs. at the end of training, noting that prior work stops training at the orange line. Stopping training prior to structural grokking can result in a vast underestimation of generalization performance.

Model We train transformer LMs with {2, 4, 6, 8, 10} layers (see Murty et al. [2023a] for more details). For each depth, we train models with 10 random seeds for 300k (400k for Dyck) steps. Given the input sentence (or prefix in the case of Dyck) we decode greedily from the model at test time. For Dyck, we report the accuracy of generating the correct closing bracket type by ranking among closing brackets, given an input prefix from the language. As done in prior work [McCoy et al., 2020, Petty and Frank, 2021, Mueller et al., 2022], for Question-Formation, we report first word accuracy of the decoded question, and for Tense-Inflection, we report the fraction of test inputs for which the target verb is correctly inflected.

3.3.1 Main Results

Transformers exhibit structural grokking We first present results obtained with the best model depth on all datasets in Fig 3.2. We find clear evidence of structural grokking: Across datasets, generalization improves many training steps after in-distribution accuracy has saturated, sometimes approaching perfect accuracy.

Early stopping considered harmful Next, we compare generalization accuracy obtained by early stopping on in-domain validation accuracy (as done in Petty and Frank [2021], Mueller et al. [2022]) to longer training runs (Fig 3.2). Early stopping leads to vastly underestimating generalization. For instance, average generalization goes up from <40%, <50% to <90%, <80% on Question-Formation



(a) Inverted U-shaped scaling (b) Dynamics of model internal properties

Figure 3.3: (a) Inverted U-shaped laws for grokking: On Question-Formation (top) and Tense-Inflection (bottom), we find that both very small and very deep models either fail to exhibit structural grokking or do so infrequently, compared to an in-between optimal model depth. (b) While weight norms and attention sparsity increase for all models and do not differentiate between different sizes, tree-structuredness is highest for the optimal model depth.

and Tense-Inflection, respectively.

Inverted U-shaped scaling On Question-Formation and Tense-Inflection, we train models of increasing depths from 2 to 10 layers. For each depth, we report the fraction of seeds (out of 10) where generalization accuracy eventually crosses 80%, in Fig 3.3a. We find an inverted U-shaped scaling behavior—very shallow and very deep models are unsuccessful, while most seeds generalize in models of intermediate depth. This may also explain why prior work that either used very shallow models (1–3-layer transformers in Petty and Frank [2021], Mueller et al. [2022]) or very deep models (12-layer transformers in Mueller et al. [2022]) failed to generalize well.

3.4 Analysis

Given that structural grokking occurs only in a subset of model architectures, can we identify when it has happened (or predict when it will occur)? Several model-internal properties have been claimed to relate to either grokking or emergent hierarchical structure in transformers.

Weight Norms Recent work [Power et al., 2022, Liu et al., 2022b] identifies the L_2 norm of parameter weights as an important quantity for grokking. For instance, Power et al. [2022] find

weight decay to improve grokking speed and Liu et al. [2022b] identify a "goldilocks zone" in weight norm space where grokking occurs. More generally, norm growth over the course of training has been studied as a key factor in neural network generalization [Soudry et al., 2018].

Attention Sparsity Merrill et al. [2021] prove that norm growth in transformers leads to attention saturation, an important property for emergent linguistic structure [Merrill et al., 2022]. As a proxy for attention sparsity of f_θ^L, we compute the negative mean entropy of all distributions $\{\mathbf{a}_k^p\}$.

Tree-structuredness McCoy et al. [2020] show that tree-structured encoders such as Tai et al. [2015b] show near perfect hierarchical generalization. While transformers are relatively unconstrained, recent evidence suggests that, when trained on language data, they implictly implement (approximately) tree-structured computations. In particular, the *tree projection* method from Chapter 2 precisely characterizes the extent to which a transformer's internal computation on an input can be approximated with a tree-structured neural encoding, providing a tree-structuredness score (t_{score}) for any transformer, and a binary tree that best approximates its computation on an input string (see Appendix in Murty et al. [2023a] for details). To evaluate whether these trees correspond to human notions of syntax, we additionally compare recovered trees to gold-standard ones [t_{parseval}, Black et al., 1991].

3.4.1 Results

We characterize the *dynamics* of weight norms (normalized by number of layers to compare different model depths), attention sparsity, and tree-structuredness, by computing these quantities every 3k gradient updates for Question-Formation and Tense-Inflection. For data-dependent properties such as attention sparsity and tree-structuredness, we sample 10k examples from the training data. We plot these quantities for the smallest model, the largest model for which at least one run shows successful grokking, and for the optimal model depth, in Fig 3.3b.

Optimal models are most tree-structured Weight norms and attention sparsity grow for all model settings in both datasets. However, these properties by themselves are unable to predict that both shallow and deep models fail—shallow models learn the sparsest solutions as well as solutions with largest weight norms, but never generalize hierarchically. Similar to the results in Chapter 2, t_{score} improves over time for all models, indicating increased tree-structuredness over time. For both datasets, the "optimal" model learns the most tree-structured solution compared to both deep

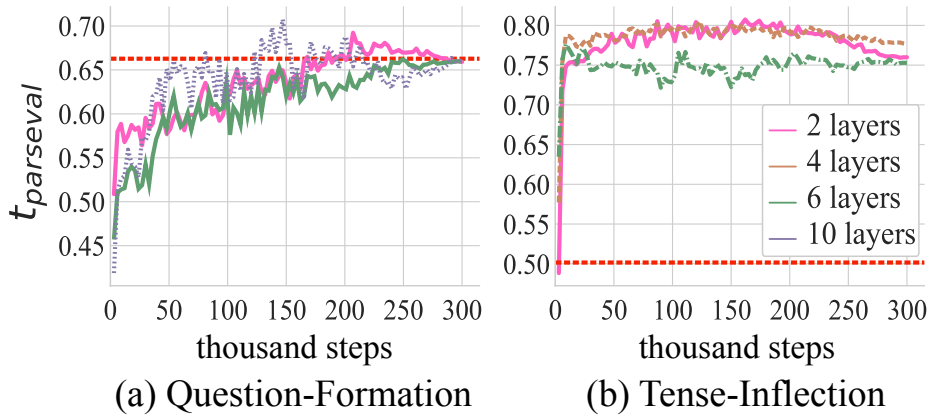


Figure 3.4: While some models fail to generalize hierarchically, all models are effective at learning computations whose closest tree structures progressively evolve towards ground truth syntax, matching or outperforming a right branching baseline (in dashed red).

and shallow models. Liu et al. [2022a] note that, on algorithmic tasks, grokking "coincides with the emergence of structure in embeddings". Similarly, for language tasks, we find that structural grokking coincides with the emergence of tree structured internal computations.

Transformers are surprisingly effective at structure induction From the dynamics of t_{parseval} in Fig 3.4, we note that all models, *regardless of whether they generalize or not*, learn structures that are close to ground truth syntax, sometimes outperforming a right-branching baseline. McCoy et al. [2020] note that tree-structured encoders only generalize when structured according to correct parse trees. Here, we find that all transformers learn correct tree structures, but only the ones that are the most tree-structured generalize best.

3.5 Conclusion

This work shows that transformers are capable of exhibiting *structure-sensitive* "hierarchical generalization" via a grokking mechanism. Their overall learning behavior gradually shifts from memorization (high in-domain accuracy, poor out-of-domain accuracy) to generalization (high in-domain and out-of-domain accuracy). While we show such behavior on relatively small datasets with small models, we believe these results may have broader implications, as training for longer has been shown to help even for web-scale language modeling [Hoffmann et al., 2022] and on compositional generalization tasks [Csordás et al., 2021]. Structural grokking happens most often at "medium-sized"

model depths, and both very shallow and very deep models fail to exhibit it. While properties previously connected with linguistic generalization in transformers such as weight norms and attention sparsity do not differentiate good architectures from bad ones, functional tree-structuredness of the transformer can well predict the optimal model depth. While there are clear limitations to the transformer architecture (such as the inability to implement unbounded recursion, which we address in Chapter 4), our results show that it may have stronger inductive biases than previously believed: With sufficient training, transformers can represent hierarchical sentence structure and use this structure to generalize correctly.

Later Work. To the best of my knowledge, Structural Grokking was the first work to demonstrate a grokking mechanism in transformer language models trained on text. Later, Wang et al. [2024a] showed that grokking also emerges in transformers on multi-step reasoning tasks. Ahuja et al. [2025] further examined the settings for structural grokking and found that consistent grokking behavior only appears in language modeling, while masked language modeling, sequence classification, and sequence-to-sequence tasks do not exhibit the same phenomenon. Interestingly, Yedetore and Kim [2024] showed that pairing examples with lambda calculus meaning representations leads to faster and more frequent grokking compared to training on forms alone (as in this chapter), highlighting the limitations of LLM training, which is *mostly* (though not always) focused on form rather than meaning. Although obtaining meaning annotations at scale can be challenging, grounding language in action sequences offers a concrete and interpretable form of meaning representation. Chapter 6 explores training on such grounded datasets. Finally, Nandi et al. [2025] proposed a new regularizer that can accelerate structural grokking by up to $10\times$.

Chapter 4

Pushdown Layers: Encoding Recursive Structure in Transformer Language Models

4.1 Introduction

An important property of human language and thought is *recursion*, which allows us to compose and reason about complex objects in terms of simpler constituents [Hauser et al., 2002]. While extensively studied in natural language syntax and semantics, recursion is also a key component of several other aspects of intelligent behaviors including mathematical reasoning, programming, and goal-directed planning. Most recursion-capable systems model recursive processes via a stack memory, which is updated as new computation is performed. For instance, a programming language may implement recursion by maintaining a run-time stack of caller-callee frames, storing intermediate outputs in the stack, and updating the stack as new function calls are made. Similarly, a shift-reduce parser implements recursion through a stack of intermediate constituents, shifting tokens onto the stack as they are observed, and occasionally reducing stack elements into constituents as they are completed.

In contrast, the self-attention mechanism underlying modern neural sequence models has no explicit mechanism to maintain a stack memory as it generates strings, and instead relies on hidden representations to implicitly but imperfectly encode such information [Manning et al., 2020]. While this encoding can model bounded recursive structure in formal languages [Yao et al., 2021], it is unclear if it is sufficient for robust syntactic generalization, especially under data-constrained settings.

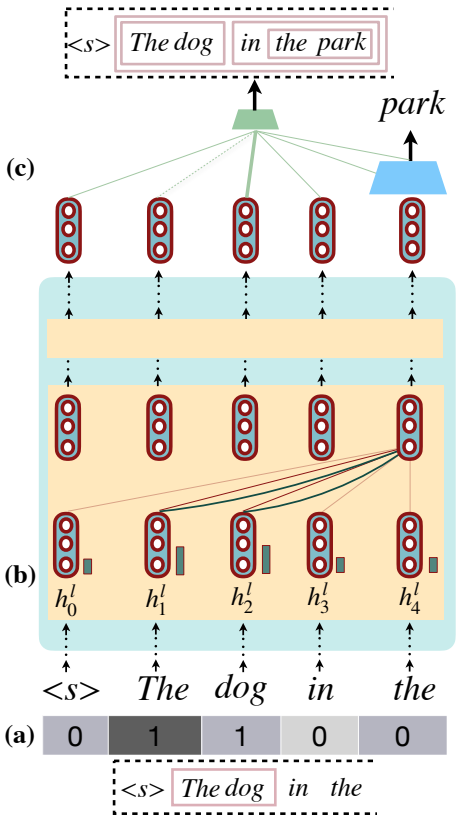


Figure 4.1: (a) Pushdown Layers use a stack-tape to featurize contents of an explicit stack, in terms of estimated token depths, where the stack represents incremental parses. (b) These depths map onto depth embeddings (in blue) that are added to token keys before computing attention scores, softly biasing attention towards a recursive syntactic computation. (c) The stack is updated *synchronously* with the newly predicted word, via an attachment head that selects a constituent to reduce the newly predicted word with, via attention.

In this work, we show that an explicit stack memory mechanism can improve syntactic generalization in Transformer language models (LMs). We introduce *Pushdown Layers*[1], a drop-in replacement for standard self-attention that augments Transformer LMs with stack memory. This memory is modeled using a *stack tape* that stores estimated depths of every token in an incremental parse of the observed prefix. The stack tape is updated autoregressively: as new tokens are predicted, Transformers with Pushdown Layers (Pushdown Transformers) synchronously make probabilistic *attachment decisions* to either "shift", thus assigning the newly predicted token a depth of 0, or "reduce" with one of the constituents in the prefix so far, updating token depths accordingly (see

[1]We borrow this term from pushdown automata, which are finite state machines augmented with stacks.

Figure 4.1). This stack tape is used to additively and softly modulate the attention of the Transformer over tokens—for instance, Pushdown Layers may guide the LM to only attend to head words of constituents, or skip over reduced constituents by decreasing attention.

Pushdown Transformer LMs are *syntactic language models* that learn joint probabilities of sequences and parses in terms of individual word predictions and structure-building operations, and can be trained on any text corpus annotated with constituency parses. But unlike other syntactic language models with structural supervision [Vinyals et al., 2015, Choe and Charniak, 2016, Qian et al., 2021, Sartran et al., 2022], Pushdown Layers do not change the output space of the underlying sequence model, and impose no constraints on attention mechanisms—the manner in which Pushdown Layers use syntactic structure for representation building is learnt purely via gradient descent.

Pushdown Transformers obtain strong generalization improvements over standard Transformer LMs. When trained on depth-bounded Dyck strings and evaluated on deeper Dyck strings, Pushdown Transformers improve performance over baseline LMs by over 25% (Section 4.4.1). When trained on sentence-level language modeling on the BLLIP-LG datasets of Hu et al. [2020], Pushdown Transformers improve syntactic generalization over standard Transformer LMs by 5–13 points as well as other joint models of strings and parses such as Qian et al. [2021], Sartran et al. [2022] by 0.3–4 points (Section 4.4.2). When trained on a new, 100-million-token dataset of parsed Wikipedia articles we call WIKITREES, Pushdown Transformers match the syntactic generalization of ordinary Transformers with 3–5$\times$ less data. Finally, when Pushdown Layers are inserted into a pre-trained GPT-2 (medium) model and fine-tuned on WIKITREES they yield improvements of 0.3–1 points on several GLUE text classification tasks.

4.2 Background

Multi-Head Self-Attention. Transformer language models [Vaswani et al., 2017a] are a class of neural sequence models that use multi-head *self-attention* to obtain contextualized representations of tokens in a sequence, which are then used to predict the next token. In particular, let $x = \{x_1, x_2, \ldots, x_n\}$ be an input sequence. Let $\boldsymbol{h}_i^l \in \mathbb{R}^d$ be the hidden representation of the ith token at the lth attention block. Then, the hidden representation of the ith token is updated as

$$\boldsymbol{h}_i^{l+1} = \text{FF}(O \cdot [\text{A}_1(\boldsymbol{h}_{\leq i}^l), \cdots, \text{A}_K(\boldsymbol{h}_{\leq i}^l)]), \tag{4.1}$$

where $O \in \mathbb{R}^{d \times d}$ is a learnt matrix, FF denotes a feed-forward + residual + layer-norm block, and A_p is the pth self-attention head. Each attention head performs a weighted average over its inputs,

$$\mathrm{A}_p(\boldsymbol{h}^l_{\leq i}) = \sum_{j=1}^{i} \alpha_{ij} W^p_{\text{value}} \boldsymbol{h}^l_j, \tag{4.2}$$

where α_{ij} is the *attention weight* assigned to the jth token by the ith token. These attention weights are computed as

$$\alpha_{ij} = \mathrm{softmax}[(W^p_{\text{key}} \boldsymbol{h}^l_j)^\top W^p_{\text{query}} \boldsymbol{h}^l_i]. \tag{4.3}$$

Each self-attention head introduces learnt parameters $W^p_{\text{key}}, W^p_{\text{query}}, W^p_{\text{value}} \in \mathbb{R}^{d/K \times d}$.

Limitations of Self-Attention. When trained on text corpora, transformers implicitly encode several aspects of linguistic structure unsupervisedly (e.g. Clark et al. [2019], Hewitt and Manning [2019] and Chapter 2). However, there is mounting evidence that recursion, a key feature of human language, remains a challenge. Hahn [2020] shows theoretically that hard-attention cannot model simple recursive structures like 2DYCK (see Section 4.6 for an extended discussion). Empirically, Lakretz et al. [2022] show that self-attention struggles on center embedding phenomenon, and Zhang et al. [2023] show poor performance on simple recursive tree-traversal problems. We hypothesize that a key reason for poor modeling of recursive structure in self-attention is a lack of an explicit structural inductive bias. One common way to add such an inductive bias is via joint modeling of strings and syntactic structure, which we introduce next.

Syntactic Language Models. Let y be the ground-truth syntactic parse of x. A long line of work [Vinyals et al., 2015, Dyer et al., 2016, Choe and Charniak, 2016, Qian et al., 2021, Sartran et al., 2022] considers learning joint distributions $p(x, y)$ to incorporate explicit syntactic structure into neural language models, by learning to output a sequence of *transition actions*,

$$p(x, y) = p(\mathbf{a}_{xy}) = \prod_i p(a_i \mid a_{<i}) \tag{4.4}$$

where actions a_i correspond to both word-level predictions as well as *structural actions* corresponding to opening and closing of constituents, building up the parse tree in a top-down, left-to-right manner. Recent work explores using Transformers to parameterize these joint distributions. For instance, Qian et al. [2021], Sartran et al. [2022] train Transformer LMs over transition actions (Parsing

as Language Modeling or PLM), sometimes with constrained attention heads (PLM-mask), and Transformer Grammars (TG; Sartran et al., 2022) model transition actions with Transformers, also with hard constraints on attention to model shift/reduce actions.

These models have several limitations that motivate our proposed approach. First, their outputs are sequences of transition actions that include both text and tree-building operations; as each constituent in a parse tree has an opening and closing transition action, and there are $\approx n$ constituents for x, this increases input length by a factor of 3, leading to significant computation and memory overhead. Second, inference in neural models operating on transitions require bespoke decoding procedures that carefully balance tradeoffs between high-entropy word-level predictions and low-entropy structural predictions [Stern et al., 2017b]. Finally, to explicitly bias Transformer computations to mirror the recursive structure of parse trees, some approaches like PLM-mask [Qian et al., 2021] and TGs [Sartran et al., 2022] impose hard constraints on attention patterns. Pushdown Layers provide a softer syntactic bias that is amenable to gradient-based learning, while having broader applicability to phenomena beyond local tree-structuredness, such as topical dependencies, coreference, etc.

4.3 Pushdown Layers

Transformer LMs with *Pushdown Layers* are syntactic language models that generate strings while simultaneously building a parse tree over these strings from left to right. This parse tree is built incrementally by tracking the recursive state of every token, which is synchronously updated along with word-level predictions. This recursive state is represented via our *stack tape* as tree-depths of every prefix token, and updates are realized with a stack. The contents of the stack tape are used to *softly modulate* attention over prefix tokens via additive offsets to attention logits (Figure 4.2).

4.3.1 Stack Tape

Like ordinary self-attention, Pushdown Layers take a sequence of hidden states $\{\boldsymbol{h}_k^l\}$ as input, and output a sequence $\{\boldsymbol{h}_k^{l+1}\}$. Additionally, Pushdown Layers use a *stack tape* $\mathcal{W}_k \in \{0, k\}^k$ to simulate a pushdown automaton that performs shift/reduce operations over tokens as they are predicted (Figure 4.2). The contents of the stack tape encode recursive state by tracking the depth of each token within reduced constituents in the stack. Concretely, after observing the prefix $x_{\leq k} = \{x_1, x_2, \ldots, x_k\}$, $\mathcal{W}_k[j] = 0$ if token x_j has not been reduced with any other token, while $\mathcal{W}_k[j] = p$ means that x_j has appeared in p reduce operations such that the resulting *constituent* has token x_j at depth p—in Figure 4.2, the stack tape encodes `[1, 1, 0]` for the incremental parse

[*The dog*] *is*.

Updating the Stack Tape. As shown in Figure 4.2, along with predicting the next word *happy*, Transformers with Pushdown Layers (*Pushdown Transformers*) make an attachment decision to update their stack tape. In our running example, this is done by selecting a constituent from the incremental parse [*The dog*] *is happy*.

Concretely, given prefix $x_{<k}$, Pushdown Transformers predict the next token x_k as well as an update to the stack tape $\mathcal{W}_{k-1}$. This is done by selecting a token r_k to reduce with, out of candidate tokens $\{x_1, x_2, \ldots, x_k\}$, via attention over hidden states $\{\boldsymbol{h}_1^L, \boldsymbol{h}_2^L, \ldots, \boldsymbol{h}_{k-1}^L, \tilde{\boldsymbol{h}}_k^L\}$, where L is the final layer of the Transformer, and $\tilde{\boldsymbol{h}}_k^L$ is a vector representation for the newly predicted token x_k, obtained as $\tilde{\boldsymbol{h}}_k^L = \mathrm{MLP}(x_k, \boldsymbol{h}_{k-1}^L)$. This vector attends to all tokens to make a probabilistic attachment decision,

$$p(r_k = j \mid x_{<k}; \mathcal{W}_{k-1}) \propto \begin{cases} (\boldsymbol{h}_j^{L\top} W^\top \tilde{\boldsymbol{h}}_k^L) & \textit{if } j \neq k\textit{, shift + reduce} \\ (\tilde{\boldsymbol{h}}_k^{L\top} W^\top \tilde{\boldsymbol{h}}_k^L) & \textit{shift only} \end{cases} \tag{4.5}$$

where $W \in \mathbb{R}^{d \times d}$ is a learnt parameter matrix. We use these probabilities to select token $r_k = \arg\max p(j \mid x_{<k}; \mathcal{W}_{k-1})$ to reduce x_k with, and the stack tape is updated accordingly via Algorithm 1. Note that attachment decisions to constituents are made by computing the attachment score for the rightmost token in the constituent. In our running example, the model selects the constituent [*The dog*] by selecting the word *dog*, forming the parse [[*The dog*] [*is happy*]] and updating the stack tape from `[1, 1, 0]` → `[2, 2, 2, 2]`.

4.3.2 Computing Attention Scores

We map contents of $\mathcal{W}_k$ onto a *per-layer* depth embedding $\boldsymbol{d}_{kj}^l$ for every token $j \in \{0, 1, \ldots, k\}$. These depth embeddings are added to attention keys, resulting in a *locally additive* modulation to attention scores,

$$\tilde{\alpha}_{kj}^l = \mathrm{softmax}\big([\boldsymbol{h}_j^l + \boldsymbol{d}_{kj}^l]^\top W_{\text{key}}^\top W_{\text{query}} \boldsymbol{h}_k^l\big). \tag{4.6}$$

Of course, since these logits are themselves part of a softmax and non-linearities, the overall effect can be arbitrarily non-linear. These modified attention weights are used to compute contextualized vectors using Eq 4.2 and Eq 4.1.

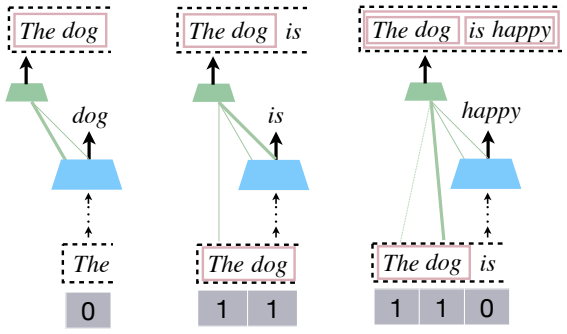


Figure 4.2: Illustration of how the parse [[*The dog* [*is happy*]] is built as a unique sequence of stack-tape updates in Pushdown LMs. Here, as the word *happy* is predicted, the attachment head chooses a constituent (bolded) from the current incremental parse, via attention. Attachment decisions are made to constituents by attending to their rightmost token, and none of the other tokens of a constituent can be attended to (shown as dashed lines). These attachment decisions are used to update depth values in the tape.

Algorithm 1 Stack Tape Update

Require: $\mathcal{W}_{k-1}$, k, r_k, `stack`
Ensure: $\mathcal{W}_k$, `stack`

$\mathcal{W}_k \leftarrow \mathcal{W}_{k-1}$
`constituent` $\leftarrow$ [k]
if $r_k = k$ **then**
    `stack.push(constituent)`
    return
end if
while True **do**
    `top` $\leftarrow$ `stack.pop()`
    ▷ *Perform a reduce*
    `constituent` $\leftarrow$ `top` $+$ `constituent`
    ▷ *Update depths in stack tape*
    for all $d \in$ `constituent` **do**
        $\mathcal{W}_k[d] \leftarrow \mathcal{W}_k[d] + 1$
    end for
    if `top` $= r_k$ **then**
        `break`
    end if
end while
`stack.push(constituent)`

4.3.3 Training and Inference

Training. Given a corpus of strings annotated with parses, we first extract ground-truth values of $\mathcal{W}_k$ for every prefix $x_{\leq k}$. We also extract ground-truth attachment decisions for x_k, given prefix $x_{<k}$. With these quantities precomputed, we can train Pushdown Transformers in *parallel*, like standard Transformers. Attachment probabilities (Eq 4.5) are supervised with ground-truth attachments, along with the standard LM objective, all using hidden states that are contextualized using the Pushdown Layer attention mechanism that uses the precomputed stack tape.

Inference. For any string x and parse y, joint probability $p(x, y)$ factorizes as a product of word-level and attachment scores as

$$p(x,y) = \prod_{k=1}^{n} \Big(p(x_k \mid x_{<k}; \mathcal{W}_{k-1}) \times p(r_k \mid x_{<k}; \mathcal{W}_{k-1}) \Big). \tag{4.7}$$

While computing the full marginal $p(x) = \sum_y p(x, y)$ is computationally infeasible due to the large space of possible parses, we approximate this by marginalizing over a smaller subset with beam search. Crucially, since our model predicts words and structural actions in *parallel* rather than sequentially, we do not need to use complex word-synchronous decoding procedures [Stern et al., 2017b] that introduce additional hyperparameters.

4.3.4 Implementation Details

FLOPs and memory overhead. Consider query and key matrices $Q \in \mathbb{R}^{n_d \times d}$, $K \in \mathbb{R}^{n_s \times d}$ where n_d and n_s refer to destination (hidden states attending) and source (hidden states being attended to). Let $S \in \mathbb{R}^{n_d \times n_s}$ be the (lower-triangular) matrix denoting pre-computed stack tape values for every prefix. For each Pushdown Layer, we use S to index into depth embeddings to obtain $D \in \mathbb{R}^{n_d \times n_s \times d}$, which is added to K to obtain $K_D \in \mathbb{R}^{n_d \times n_s \times d}$. Unlike standard self-attention which multiplies Q and K directly, Pushdown Layers multiply Q (a 2D tensor) with K_D (a 3D tensor). This is done by casting Q into a 3D tensor $\in \mathbb{R}^{n_d \times 1 \times d}$ and performing a batched matrix multiplication with K_D, leading to the same number of operations as standard self-attention [2]. However, since Pushdown Layers require storing 3D tensors for keys, this increases memory requirements from $O(n_d \cdot n_s + n_s \cdot d + n_d \cdot d)$ to $O(n_d \cdot n_s + n_s \cdot n_d \cdot d + n_d \cdot d)$.

[2] We note that standard self-attention is faster in practice due to better GPU memory bandwidth management,

Attending to hidden states with old memory. Pushdown Transformers build parse trees incrementally from left-to-right, and so, depth values of prefix tokens change as new tokens are predicted. Thus, a token at position i builds its representation based on attending to $x_{\leq i}$ with a stack tape that may soon become "stale" due to future transition operations that reduce tokens in $x_{\leq i}$ with new tokens. As an example, suppose we have the incremental parse [[*The dog*] [*in* [*the park*]]]. Here, the representation for *in* attends to representations of *The*, *dog* and *in* with depths [1, 1, 0] while the representation for *park* attends to these representations with *updated* depths [2, 2, 2].

4.4 Experiments

4.4.1 Warm-up: Dyck Languages

We train 6 layer LMs with Pushdown Layers (Pushdown-LM) as well as standard LMs on 100k strings sampled from $\text{DYCK}_{20,10}$, the language of well-nested brackets with 20 bracket types and max-nesting depth of 10. To ensure that improvements are not merely due to multi-task learning with an attachment head, base-LM is also trained with an attachment loss in a standard multi-task learning setup. To test generalization, models are provided an input prefix from a separate DYCK language, and evaluated on choosing the correct closing bracket. Specifically, we test generalization to DYCK strings with deeper nesting of 15–50, and DYCK strings with longer-range dependencies than seen at training time (measured as the distance to the matching bracket that needs to be closed). From Table 4.1, we find that Pushdown-LM obtains over 25% accuracy point improvement over standard language models at generalizing to deeper structure, as well as large improvements at generalizing to longer-range dependencies.

4.4.2 Sentence-Level Language Modeling

Setup. Next, we train 16-layer Pushdown Transformer LMs on the BLLIP-LG dataset of Charniak et al. [2000], with training splits from Hu et al. [2020], and the same pre-processing as Qian et al. [2021]. We use the same hyperparameters (model size, dropout, learning rate schedulers) as Sartran et al. [2022]. To measure syntactic generalization, we evaluate on BLIMP [Warstadt et al., 2020] and the SG test suites [Hu et al., 2020]. In BLIMP, models are provided with a grammatical and ungrammatical sentence, and evaluated on assigning a higher probability to the grammatical sentence. SG test suites consist of an extensive set of hand-crafted test cases, covering 6 fine-grained syntactic phenomena. Each test case involves satisfying a specific inequality constraint among surprisal

	Base-LM	Pushdown-LM
Long-Range Dependencies		
DYCK (50)	90.0	**96.5**
DYCK (100)	81.0	**88.0**
DYCK (200)	40.6	**61.2**
DYCK (300)	14.1	**42.9**
Deeper Embedded Structure		
Depth Gen.	40.6	**68.3**

Table 4.1: Evaluating LMs at closing Dyck prefixes with longer dependencies (dep. length in brackets) and deeper structure. We find significant improvements from using Pushdown Layers over standard self-attention.

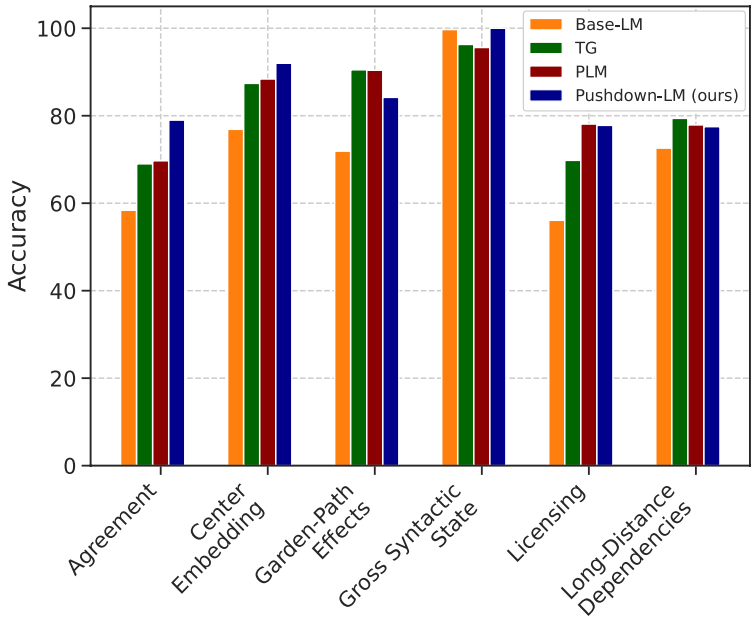


Figure 4.3: Comparing Pushdown-LMs with baseline Transformer LMs and other syntactic LMs. While Pushdown-LMs are comparable with Transformer Grammars (TG; Sartran et al., 2022) across all examples in SG test suites (Table 4.2), they outperform TGs on 4 out of 6 tests, including the recursive center embedding tests.

Model	BLIMP ↑	SG test suites ↑	PPL ↓
Models that add structural tokens to inputs			
PLM	75.1	80.2	29.8‡
PLM-Mask	75.3	78.3	49.1‡
TG	–	**82.5***	30.3‡
Models that do not add extra tokens to inputs			
Base-LM	70.1	69.5	20.1
Pushdown-LM (ours)	**75.6**	**82.3***	**19.9**

Table 4.2: Syntactic Generalization on BLIMP and SG test suites. All results for PLM-Mask are taken from Qian et al. [2021] and results for PLM and TGs are taken from Sartran et al. [2022]. $*$ denotes differences that are not significant. PPL results marked with ‡ are taken from prior work and not comparable due to differences in tokenization.

values of various continuations given prefixes, where these inequalities are grounded in theories of incremental language processing—for instance, assigning a higher surprisal to the last verb in *The painting that the artist deteriorated painted* vs. *The painting that the artist painted deteriorated*. For BLIMP, we obtain $p(x)$ by approximate marginalization via beam search. Since surprisal values $-\log p(x_t \mid x_{<t})$ in SG test suites are meant to reflect incremental sentence processing, we perform marginalization based on the beam state at time step t. We fix the beam size at 300.

Results. We present results on SG test suites in Figure 4.3. As baselines, we compare against a standard 16 layer Transformer LM and prior structured models (TG, PLM) from Sartran et al. [2022]. As expected, all models with an explicit notion of structure have much better syntactic generalization across all test suites. Next, we note that Pushdown-LM, a 16 layer Transformer LM with all self-attention blocks replaced with Pushdown Layers, outperforms prior approaches—Pushdown-LM beats TG on 4/6 tests and PLM on 3/6 tests with similar performance on licensing. Next, we present results (averaged across 3 seeds) on BLIMP as well as aggregate SG test suite results and perplexity on the BLLIP test set in Table 4.2. Here, we note that Pushdown-LM achieves better syntactic generalization than prior structured models (including the PLM-mask model from [Qian et al., 2021]) on BLIMP. Finally, we find that Pushdown-LM achieves slight gains in perplexity compared to Base-LM.

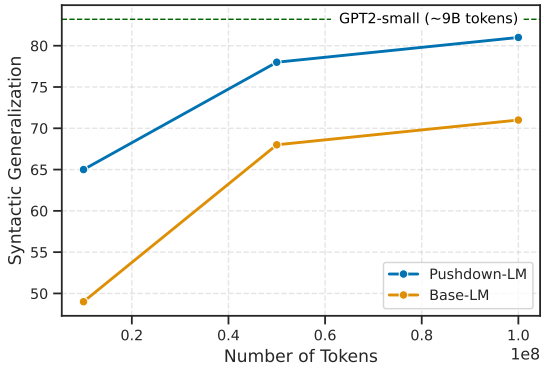


Figure 4.4: Comparing a standard GPT-2 small architecture (Base-LM) with a model where the last 6 self-attention blocks use Pushdown Layers, trained on various amounts of tokens from WIKITREES. We find that Pushdown Layers greatly improve sample efficiency of syntactic generalization. For reference, we also include GPT2-small, which is trained on over 9 billion tokens.

4.4.3 Language Modeling with WIKITREES

Can Pushdown Layers continue to offer improvements on larger-scale language modeling? We construct WIKITREES, a dataset of over 100 million tokens extracted from Wikipedia Articles (WikiText-103; Merity et al. [2017]), parsed automatically using a state-of-the-art neural constituency parser [Kitaev et al., 2019]. Typically, LMs trained on web-scale data are given multi-sentence contexts with large window sizes as inputs, and to adapt this to Pushdown-LMs we make a small number of modifications (see Murty et al. [2023b] for details).

Sample-Efficient Generalization. To measure sample efficiency in Pushdown Transformers, we train LMs on [10M, 50M, 100M] tokens from WIKITREES. To ensure stable training under low data regimes, we train a 12 layer GPT2 using the exact configuration and tokenization scheme as GPT2-small [Radford et al., 2019], and additionally use dropout to prevent overfitting. For these experiments, we compare Base-LM with an LM where the final 6 self-attention blocks are Pushdown Layers (Pushdown-LM). To measure syntactic generalization, we compute aggregate performance on the SG test suites. From results in Figure 4.4, we find that Pushdown-LMs exhibit drastically more sample-efficient syntactic generalization—for instance, syntactic generalization of Pushdown-LM trained on 10M tokens requires over 40M tokens for the Base-LM to surpass.

Finetuning for text classification. Can Pushdown Layers offer improvements on language understanding tasks, beyond syntactic generalization? To answer this, we perform staged finetuning

of GPT2-medium with Pushdown Layers. Specifically, we finetune GPT-2 medium with the final 12 self-attention blocks replaced with Pushdown Layers (Pushdown-GPT2), as a language model on WIKITREES. We use this model to obtain parses on 4 text classification tasks: RTE, SST5, MRPC and STS-B from GLUE [Wang et al., 2019a], and use these parses to pre-compute the stack tape for every token. Then, in a second finetuning step, Pushdown-GPT2 is trained to perform text classification over these datasets by reducing each task into language modeling via prompting (See Murty et al. [2023b] for prompt details). As a comparison, we also perform the same staged finetuning for the standard GPT2-medium architecture. We report averaged results across 3 seeds in Table 4.3. We find that Pushdown Layers offer improvements on 3 out of 4 text classification tasks.

Model	RTE	SST5	MRPC	STS-B
GPT2	72.2	**54.8**	88.4	89.6/89.8
Pushdown-GPT2	**72.9**	54.5	**89.3**	**89.8/90.1**

Table 4.3: Finetuning models on various semantic text classification/regression tasks. We report accuracy for RTE and SST5, F1 for MRPC, and Spearman/Pearson Correlation for STS-B.

4.5 Analysis

For all analyses, we use the 16 layer Pushdown-LM trained on BLLIP-LG from Section 4.4.2.

Parsing. Since Pushdown-LM is a syntactic language model, we obtain parses via beam search (beam size = 300) to approximately recover the most likely parse $y^* = \arg\max_y p(x, y)$ under our model. However, since this parse is (a) unlabeled and (b) binarized, we perform an *unlabeled F1 evaluation* (using `EVALB`; Collins, 1997) over *binarized* ground-truth parses from the PTB test set. We also remove instances consisting of unknown words for our model, since our model is trained without any UNK tokens, giving us 2335 out of 2416 sentences. We compare our model against Kitaev et al. [2019], the parser that was used to annotate training data for Pushdown-LM. We also present unlabeled F1 on the auto-parsed BLLIP-LG test set. From results in Table 4.4, we note that our model achieves a very competitive unlabeled F1 score of 95.3, outperforming the official implementation of Kitaev et al. [2019][3]. We also find that our model obtains a high F1 score of 97.3 on the BLLIP-LG test set.

[3]We use the `benepar_en_large` model from https://github.com/nikitakit/self-attentive-parser which reports a score of 96.29 on the full PTB test set, while we obtain 95.66 (labeled F1, using the standard `EVALB` script).

Model	PTB	BLLIP-LG
Pushdown-LM	**95.3**	97.3
[Kitaev et al., 2019]	94.7	-

Table 4.4: Unlabeled F1 scores against binarized ground-truth parses from the PTB and BLLIP test sets. We filter all examples from the PTB test set with unknown words, giving us 2335 out of 2416 sentences. Annotations on BLLIP-LG are obtained using Kitaev et al. [2019].

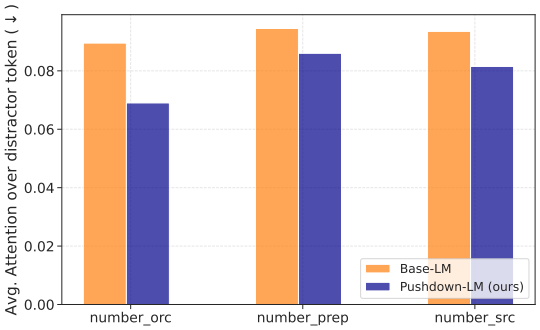
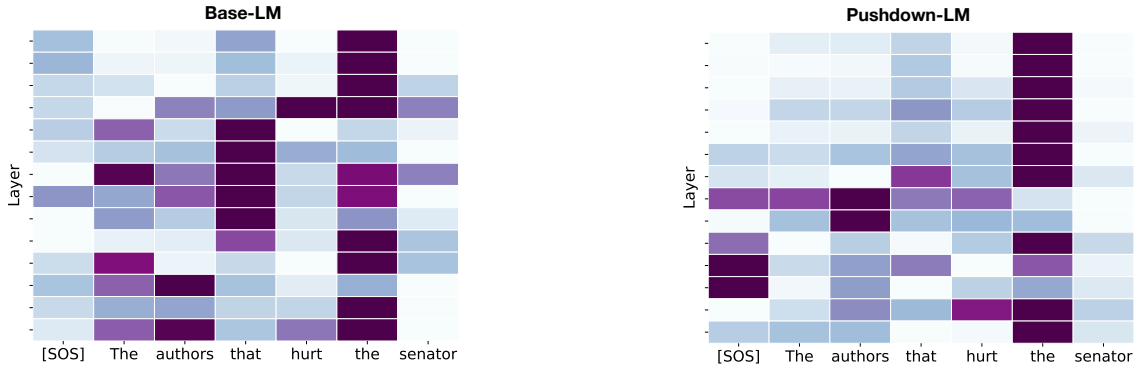


Figure 4.5: For the three subject-verb agreement tasks from [Marvin and Linzen, 2018], we compute average attention over the distractor noun when the verb is being predicted, for both the Base-LM and Pushdown-LM (ours). Across all variants, we find that our model consistently pulls attention away from distractor nouns.



(a) Base LM (b) Pushdown LM

Figure 4.6: Given a prefix containing a main noun and a distractor noun, Pushdown-LM pulls attention away from the distractor (here *senator*), helping the model predict the verb with the correct number. These attention maps average across all the instances in the number_src test of SG test suites, and we show the attention over all prefix tokens when the main verb is predicted.

Case Study: Analyzing attention patterns on subject-verb agreement tasks. We consider the 3 Subject-Verb agreement tasks [Marvin and Linzen, 2018] from the SG test suites. On these tasks, models are presented with a prefix consisting of a main subject and a distractor embedded subject, where these items conflict in number. The objective is to assign a higher logprob to the verb that agrees with the main subject rather than the distractor subject. For instance, for prefix *The author that hurt the senators*, the model must assign a higher probability to *is* than *are*.

From Figure 4.3, we find that Pushdown-LM significantly outperforms other models with close to 80% accuracy while Base-LM achieves less than 60% accuracy. To understand how Pushdown Layers modulate attention on these examples, we obtain attention scores over all prefix tokens (averaged across all layers). We present the average attention assigned to the distractor token for both Pushdown-LM and Base-LM in Figure 4.5 where we observe that Pushdown-LM pulls attention away from the distractor noun, allowing it to predict the correct verb. Finally, we plot some (averaged) attention heatmaps in Figure 4.6.

4.6 Other Related Work

While recursive structure is fundamental to natural language, modeling such structure is difficult for self-attention. Hahn [2020] considers DYCK, the simplest formal language with recursive structure, proving that hard attention cannot recognize DYCK and soft attention cannot recognize DYCK with

low cross-entropy. In practice, we find that even simpler languages like PARITY are challenging for encoder-only Transformers [Chiang and Cholak, 2022, Bhattamishra et al., 2020]. On the other hand, Transformers with decoders have been shown to be Turing-complete [Perez et al., 2021], but these constructions rely on the impractical assumption of running the decoder for an unbounded number of steps. In practice, we find that Transformer LMs struggle with generalization beyond regular languages and tend to learn shortcuts instead [Deletang et al., 2023, Liu et al., 2023].

Given these limitations, there is significant interest in inductive biases that encourage recursive structure in Transformers. One line of work considers constraining self-attention patterns according to syntactic parses [Strubell et al., 2018, Wang et al., 2019b, Peng et al., 2019, Deshpande and Narasimhan, 2020, among others]. A second line of work adds structure to language modeling by learning joint probabilistic modeling of structure and strings [Chelba, 1997, Mirowski and Vlachos, 2015, Choe and Charniak, 2016, Dyer et al., 2016, among others]. Both of these ideas are combined in recent work of Qian et al. [2021], Sartran et al. [2022], that proposes joint string, parse Transformer language models with constrained attention patterns. While Pushdown Layers are also in this modeling tradition, we do so without operating on long transition actions, and enforce structural constraints via gradient based learning.

4.7 Conclusion

We propose Pushdown Layers, a new kind of self-attention that augments Transformer language models with a stack based memory. Pushdown Layers enable auto-regressive Transformers to softly bias attention towards a recursive syntactic computation, through an updatable stack-tape that stores token depths in an incremental parse. When trained on synthetic and natural languages, we find that Transformer LMs with Pushdown Layers achieve better generalization to deep recursive structure, as well as better and more sample-efficient syntactic generalization. When pre-trained LMs are finetuned with Pushdown Layers, we obtain improvements on some GLUE tasks.

Discussion. A long line of work has proposed neural networks augmented with memory, such as stacks [Das et al., 1992, Grefenstette et al., 2015, Joulin and Mikolov, 2015, DuSell and Chiang, 2022] or random-access memories [Kurach et al., 2015]. These augmented architectures significantly improve algorithmic generalization and the ability to learn recursive structure [Suzgun et al., 2019, Deletang et al., 2023]. Our work is the first to design a structured memory (the stack-tape) for transformers that mimics shift/reduce updates while remaining parallelizable for efficient training.

Pushdown LMs softly inject syntactic structure, allowing gradient descent to learn how to adjust attention patterns based on stack features. While there have been several efforts to incorporate syntactic inductive biases into sequence models—often RNNs—in an unsupervised manner [Bowman et al., 2016, Shen et al., 2019, Drozdov et al., 2019, Kim et al., 2019, among others], some of these methods have faced skepticism [Dyer et al., 2016]. We leave unsupervised training of Pushdown Transformers as an open question for future work.

With this chapter, we conclude the first part of the thesis. Intelligent digital agents need not only a robust compositional understanding of language but also deep knowledge about the environments in which they are situated. The next chapters present methods for endowing ungrounded LLMs with this essential grounding and environmental knowledge—acquired in a completely unsupervised way through interaction with their environments.

Part II

Grounding in Digital Environments through Environment Interaction

Chapter 5

BAGEL: Bootstrapping Agents by Guiding Exploration with Language

5.1 Introduction

In recent years, large language models (LLMs) have shown strong performance on a broad range of language understanding tasks, making them powerful tools for controlling policies in digital environments such as web browsers [Yao et al., 2022, Kim et al., 2023]. Such grounded language understanding tasks are fundamentally challenging for LMs in environments with ambiguous dynamics. For instance, even inputting a date into a text box could require either simply typing or a complex interaction using a drop-down date picker. An LM cannot know this a-priori without in-depth knowledge about the website.

One common way to provide such knowledge to LM agents is via expert demonstrations that provide information about mapping instructions to action sequences, recovering from errors, and reasoning traces [Yao et al., 2022, Sun et al., 2023, Kim et al., 2023, Sodhi et al., 2023]. Of course, collecting human demonstrations for every new environment is laborious and requires knowing possible user instructions *a priori*. Moreover, as agents scale to complex tasks with hundreds of actions, human supervision will become increasingly infeasible to obtain. Instead of relying on human demonstrations for training LM agents, could we instead use exploration and environment feedback to automatically collect a large number of *synthetic* demonstrations?

Prior work has shown the effectiveness of collecting synthetic demonstrations by retroactively labeling trajectories from embodied agents [Sumers et al., 2023]. In this scenario, the environments dynamics are assumed to be well understood by the agent; the synthetic demonstrations only serve

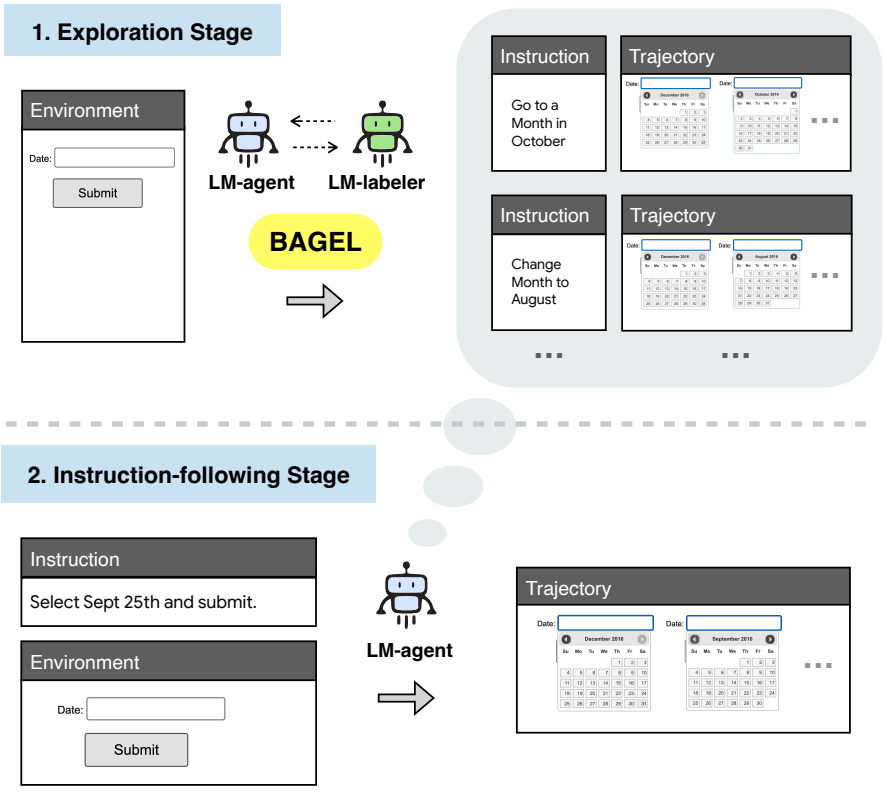


Figure 5.1: (Top) Given a seed set of explored trajectories, BAGEL constructs synthetic demonstrations via an iterative round-trip procedure between two LM components: a zero-shot LM agent that generates trajectories and an LM labeler that generates instructions for these trajectories. (Bottom) Given an instruction at test time, we retrieve synthetic demonstrations with similar instructions, to use as in-context exemplars to adapt the base agent.

to connect agent behavior with human language. However, we observe the opposite challenge with digital agents in our setting—grounding instructions is relatively easy due to the highly textual environment, but *zero-shot* digital agents typically are not exposed to any environment dynamics before they are directly used to follow instructions.

Our method, termed BAGEL (**B**ootstrapping **A**gents by **G**uiding **E**xploration with **L**anguage), uses an iterative procedure to relabel a seed set of trajectories obtained from unconditioned exploration (Figure 5.1). Intuitively, BAGEL operates by progressively shifting the distribution of trajectories towards those that can be well-described via natural language, using two noisy LM components: an LM *labeler* takes a trajectory and relabels it with a synthetic instruction, and a zero-shot LM policy maps the instruction back into a refined trajectory (Figure 5.2). By performing these round

trips iteratively, BAGEL converts trajectories from random exploration into meaningful trajectories that are executable, without requiring a trained base agent or significant information about possible instructions. While both the re-labeling and instruction-following processes are imperfect, round-trips between these components work in harmony to reduce any noise. Once an instruction, trajectory pair reaches a threshold score under a *demonstration filter* (another prompted LM), the generated synthetic demonstration is added into a buffer. BAGEL demonstrations can be used for both in-context learning or finetuning, and serve as a drop-in replacement for expert demonstrations. Here, we follow the former strategy along with a simple retrieval augmented generation procedure—given a user instruction at test time, we retrieve the most relevant demonstrations based on instruction embeddings, and feed that into the agent's prompt to serve as in-context exemplars.

While BAGEL shares some similarities with Hindsight Experience Replay (HER, Andrychowicz et al., 2017), a popular method for retroactive relabeling of unsuccessful trajectories, there are important technical differences: Instead of relabeling trajectories based on only the final observation, our relabeling function operates on the entire transition history from the trajectory and uses language models to *iteratively* enforce a language prior over the distribution of trajectories (Section 5.3.3). Moreover, while HER is used in offline Q-learning settings, we use BAGEL primarily as a data generation method.

We experiment with BAGEL on two domains, by using a prompted LM (similar to ReAct, Yao et al., 2022) as our base policy and find significant improvements with *no human supervision*. In MiniWoB++ [Shi et al., 2017, Liu et al., 2018], an agent follows instructions on diverse web-interfaces ranging from booking flights to replying to emails, given an HTML state, by issuing a sequence of mouse and keyboard operations to interact with DOM objects. Using BAGEL for test-time adaptation, we find an improvement of over 13% compared to the base LM policy. Next, we evaluate on ToolQA [Zhuang et al., 2023], a collection of question answering tasks over 8 domains, where answering each question requires chaining together multiple tools such as SQL, text retrievers, graph tools, python interpreters and calculators. Here, we find an improvement of 2% over the base LM policy. Further analysis reveals the various positive effects of conditioning on our synthetic demonstration beyond improved accuracy, including up to 13$\times$ reduction in execution failures due to better understanding of environment dynamics. By carefully using LM priors to shape random exploration, our method serves as a tool for automated discovery of use cases in complex environments.

5.2 Background

Given a natural language instruction g, our agent interacts with the environment by taking a sequence of actions $\{a_1, a_2, \ldots, a_T\}$, where each a_t is issued in response to an environment observation o_t. The entire interaction with the environment is captured as a *trajectory* $\tau = \{o_1, a_1, o_2, \ldots, o_T, a_T, o_{T+1}\}$.

We define an agent as a *language conditioned policy* $\pi(a_t \mid \tau_{<t}, g)$ where $\tau_{<t} = \{o_1, a_1, o_2, \ldots, o_t\}$ refers to the trajectory until time-step t. Such policies are typically trained via imitation learning and optional RL finetuning, where a large set of expert curated instruction-trajectory pairs are required for imitation learning, and a suitably shaped reward signal is needed for RL finetuning [Branavan et al., 2009, Chaplot et al., 2018, Misra et al., 2017]. For our setup, both observations and actions can be expressed as natural language strings. The agent policy π can then be cast into an autoregressive LM that assigns probabilities to action strings given string descriptions of the previous actions and observations. Thus, recent work focuses on directly using LLMs as policies, by using prompts along with in-context human demonstrations [Yao et al., 2022, Shinn et al., 2023, Sun et al., 2023, Kim et al., 2023, among others].

Executing Action Strings. Similar to prior work that uses LMs to generate action strings [Huang et al., 2022, Logeswaran et al., 2022], we assume access to an environment-specific *low-level controller* that maps action strings to a low-level command (e.g. a web-driver action or an API call), which can be directly executed to change the environment.

5.3 BAGEL

BAGEL generates synthetic demonstrations via exploration, as illustrated in Figure 5.2. First, we describe the various model components in §5.3.1, and then describe the overall procedure in §5.3.2.

5.3.1 Model Components

In order to generate synthetic demonstrations, we model different aspects of the joint distribution over instructions and trajectories. Every component is implemented by the same underlying LM, but with different prompts. Every component is also implicitly dependent on a given environment, although this is omitted in the notation for simplicity. All prompts used can be found in Appendix B.

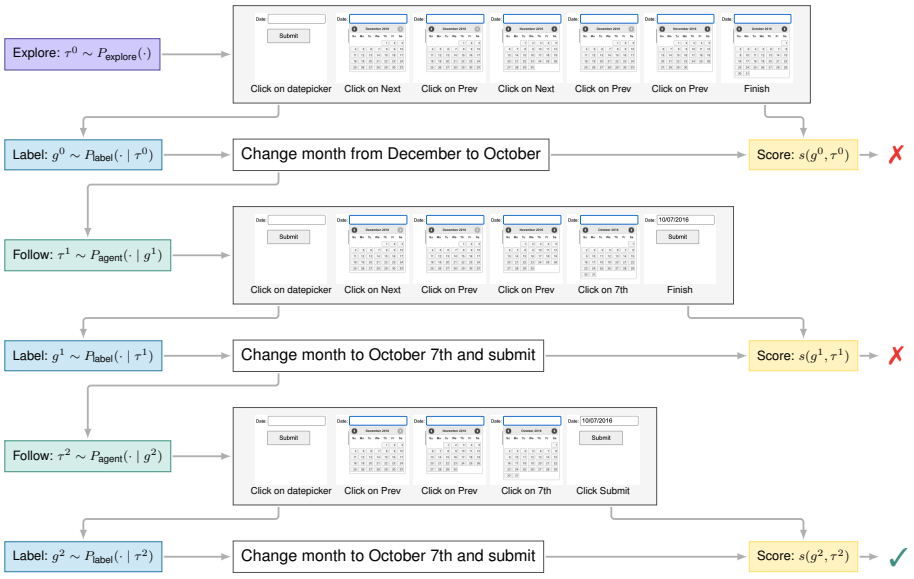


Figure 5.2: BAGEL generates synthetic demonstrations by exploring the environment. Shown here is an example from the MiniWob++ *choose-date* task. First, we generate an initial trajectory by sampling actions without conditioning on any natural language instruction. Then, we alternate between generating an instruction given a trajectory, and generating a trajectory given an instruction. The process aims to converge towards a trajectory that accurately satisfies a natural language instruction, and aims to recover from errors in labeling or instruction following from earlier rounds (see example). Once an instruction and trajectory pair satisfies a filtering criteria, it is added to the set of synthetic demonstrations. Alternatively, TreeProjections can be initialized by first sampling an instruction, as described in §5.3.2.

Exploration Policy. The exploration policy, $\pi_{\text{explore}}(a_t \mid \tau_{<t})$, selects an action without conditioning on any instruction. The prompt used is similar to that of ReAct [Yao et al., 2022]. We can sample from the resulting distribution over trajectories, $p_{\text{explore}}(\tau)$, by sampling actions from π_{explore} until the episode completes or a "finish" action is generated. We can increase the entropy of π_{explore} with a configurable temperature parameter.

Trajectory Labeler. The trajectory labeler, $p_{\text{label}}(g \mid \tau)$, is prompted to generate an instruction, g, that corresponds to a given trajectory, τ.

Instruction Following Policy. Unlike the exploration policy, the instruction following policy, $\pi_{\text{agent}}(a_t \mid \tau_{<t}, g)$, selects actions conditioned on an instruction, g. We sample from the resulting distribution over trajectories, $p_{\text{agent}}(\tau \mid g)$, by choosing actions according to π_{agent} until the episode

completes or a "finish" action is generated. This component is also implemented using a ReAct based prompt.

Demonstration Filter. Given a synthetic demonstration (g, τ), the demonstration filter makes a binary judgement $s(g, \tau) \in \{0, 1\}$, based on how well τ corresponds to the instruction g.

Instruction Generator Finally, as an alternative to the exploration policy (see §5.3.2) we can instead use an instructor generator to initialize exploration. This model defines a distribution over instructions, $p_{\text{instruct}}(g)$, based on a prompt that elicits plausible instructions based on the initial observation from the environment, and the action space.

5.3.2 Generating Demonstrations

Initial Exploration We consider and compare two different variations of BAGEL: *trajectory-first* and *instruction-first* exploration. For trajectory-first exploration, we first sample a trajectory $\tau^0 \sim p_{\text{explore}}(\cdot)$ with the exploration policy. For instruction-first exploration, we first sample an instruction $g^0 \sim p_{\text{instruct}}(\cdot)$ with the instruction generator.

Iterative Refinement Trajectories sampled from p_{explore} may not correspond to any reasonable instruction, and, similarly, there may be no feasible trajectory that satisfies instructions sampled from p_{instruct}. Our iterative re-labeling procedure aims to find an instruction and trajectory pair where the trajectory satisfies the instruction, without sacrificing the diversity of the initial exploration. The process alternates between sampling instructions and trajectories:

$$g^t \sim p_{\text{label}}(\cdot \mid \tau^t). \tag{5.1}$$

$$\tau^{t+1} \sim p_{\text{agent}}(\cdot \mid g^t). \tag{5.2}$$

We perform these iterative updates until we find a pair where $s(g^t, \tau^t) = 1$ or a maximum number of steps is reached. If we are successful, the demonstration (g^t, τ^t) is added to the set of synthetic demonstrations, $\mathcal{M}$. The overall procedure is repeated to collect multiple demonstrations.

5.3.3 Discussion

Guiding Trajectory Distribution with LM Components. To better understand how the LM labeler and policy shape the distribution of trajectories, we consider how this distribution evolves

over the course of multiple iterations. Let $p_k(\tau)$ be the distribution over trajectories and $p_k(g)$ be the distribution over instructions, after k iterations. For $k > 0$:

$$p_k(\tau) = \sum_{g'} p_{\text{agent}}(\tau \mid g') \cdot p_{k-1}(g') \tag{5.3}$$

$$p_{k-1}(g') = \sum_{\tau'} p_{\text{label}}(\tau' | g') \cdot p_{k-1}(\tau'). \tag{5.4}$$

Combining these, we obtain:

$$p_k(\tau) = \sum_{\tau', g'} p_{k-1}(\tau') \cdot \underbrace{p_{\text{label}}(g' \mid \tau') \cdot p_{\text{agent}}(\tau \mid g')}_{\text{environment and LM constraints}}. \tag{5.5}$$

Thus, we shape the distribution of trajectories from the previous marginal p_{k-1} based on the criteria that they can be assigned a concrete string g', and are executable in the environment. These soft constraints work together to ensure that (1) trajectories can be described in terms of some feasible instruction in the environment, and (2) the trajectories themselves correspond to valid environment dynamics.

Connection to Hindsight Experience Replay. Hindsight Experience Replay (HER, Andrychowicz et al., 2017) is a popular approach for training language conditioned policies. Given some goal g, HER converts an unsuccessful trajectory τ into positive examples by replacing g with some *hindsight goal* g'. That is, HER uses a *relabeling function* to map τ to a new goal g', resulting in a positive demonstration (g', τ), that is used to update the policy.

Since the original implementation of HER considers settings where the goal space is the raw environment observation space, applying HER to natural language instruction-following requires access to a learnt relabeling function to map observations to language instructions. Such relabeling functions typically map only the *final* observation o_T to the instruction via pre-trained captioning models [Xiao et al., 2022, Cideron et al., 2020, Sumers et al., 2023] that operate on trajectories from *trained* agents. In BAGEL, we use the full trajectory for relabeling and use an iterative relabeling procedure to reduce noise from zero-shot components.

5.4 Inference

We use synthetic demonstrations from BAGEL to adapt LM agents via retrieval augmented generation, and leave finetuning for future work. Concretely, given a test instruction g_{test}, we retrieve top-k most relevant demonstrations in the demonstration set $\mathcal{M}$, pre-pending these to the context window of our agent as in-context examples. More concretely, we use dual encoder retrieval, similar to Lee et al. [2019], using a T5-XXL [Raffel et al., 2020] embedding model. We first compute a vector embedding $f_\theta(g)$ for each instruction $g \in \mathcal{M}$, and then find the top-k demonstrations based on scores $f_\theta(g)^\top f_\theta(g_{\text{test}})$. More details can be found in Murty et al. [2024].

5.5 Datasets

Our experiments are based on two environments, MiniWoB++ [Shi et al., 2017, Liu et al., 2018] and ToolQA [Zhuang et al., 2023].

5.5.1 MiniWoB++

MiniWoB++ is a collection of tasks consisting of web interfaces with a shared action space of mouse and keyboard actions. In our setup, actions are specified in natural language (*Type Bob in the name text box*, *Click on the datepicker*, *Clear text on Destination*). The low-level controller that maps action strings into a Selenium API call is implemented via a separate zero-shot prompted LM (see Appendix B.4 for details). Each task consists of a script to generate variations of the task with a templated instruction, where each variation is controlled via a random seed.

Evaluation. We follow Shaw et al. [2023] for evaluating agents on MiniWoB++, by mapping the raw MiniWoB++ reward from [-1, 1] to [0, 1]. For each web interface, we report the mean score over 50 random seeds. Starting with the set of 55 MiniWoB++ tasks used in prior work on applying LM agents to this domain [Gur et al., 2023, Kim et al., 2023, Sun et al., 2023], we evaluate on the hardest 10 tasks where the zero-shot agent has an average reward of less than 0.95, to perform a more targeted evaluation of BAGEL to domains that are hard for zero-shot agents.

5.5.2 ToolQA

ToolQA is a tool augmented question-answering environment over 8 domains, where questions can be answered by chaining calls to multiple tools including text retrievers, databases, SQL interpreter,

calculator etc. Each tool can be called according to a set of pre-defined methods (see Appendix B for the full action space for the policy and corresponding tool methods). The observation space is the string output from the most recent tool call (the first observation is hard-coded as a "System prompt"). Each action corresponds to a specific tool call expressed in language (*Load the Airbnb Database*, *Calculate 3+7*), and the low-level controller is implemented by post-processing strings into tool methods. The episode terminates when the policy chooses the *Finish with Answer* action e.g. *Finish with Answer: 300*, where *300* is taken as the predicted answer.

Evaluation. Following prior work on question-answering [Rajpurkar et al., 2016, 2018, Joshi et al., 2017], we compute the F1 score of the final (free-form) model output from the *Finish with Answer* tool call against ground-truth answers.

5.6 Experimental Setup

5.6.1 Baselines and Ablations

Zero-shot. As our first baseline, we use the zero-shot policy π_{base} directly at test time.

Non-iterative Ablations. Similar in spirit to Sumers et al. [2023], in BAGEL *(trajectory-first, no itrs)*, explored trajectories τ^0 are labeled using p_{label} and resulting demonstrations (g, τ^0) are included in $\mathcal{M}$ if the score $s(g, \tau) = 1$. Similarly, in BAGEL *(instruction first, no itrs)*, synthetic instructions sampled from the instruction generator (see §5.3.1) are converted into trajectories using p_{agent}, and the resulting demonstration (g^0, τ) is added to $\mathcal{M}$, if $s(g^0, \tau) = 1$. This baseline captures a simple way to use LMs to construct synthetic demonstrations via a sample-then-filter approach: prompt an LM to generate possible instructions given the first observation from the environment, create trajectories based on these, and filter based on another criterion. In general, we expect exploration using the instruction generator to work poorly in settings where the LM cannot predict potential instructions from just the first observation (e.g. it might hard to generate candidate instructions solely from the landing page of the website without further interaction).

5.6.2 Implementation Details

We evaluate all baselines and variants of BAGEL on MiniWoB++ and ToolQA. For MiniWoB++, we start with sampling 60 trajectories in the exploration phase for trajectory-first variants of BAGEL,

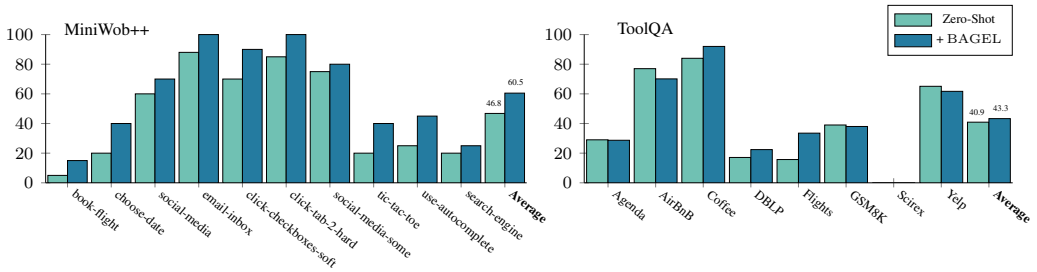


Figure 5.3: Results across MiniWoB++ and ToolQA, broken down by domain. We compare using demonstrations obtained via BAGEL (blue) with a zero-shot ReAct baseline (**green**) with no synthetic demonstrations. For MiniWob++, we use the Trajectory-First variant for exploration, and for ToolQA, we use Instruction-First. We report mean reward for MiniWob++ and F1 score for ToolQA. Overall, using BAGEL demonstrations leads to improvements on both datasets.

and sample 60 synthetic goals for instruction-first variants. For ToolQA, we sample 200 trajectories for BAGEL (trajectory-first), and 200 synthetic goals for BAGEL (instruction-first).

We use an instruction tuned PaLM-2 [Anil et al., 2023] as the base LM for all our experiments, and sample with a fixed temperature of 1.0. We set the max episode length T to 15 for all datasets and models. We also set T_{iter} to 5, when performing multiple iterations in BAGEL [1].

In addition to using ReAct prompting, we use a simple "re-sampling" procedure to recover from issuing syntactically incorrect actions—if an action causes the environment to return an Exception (such as incorrectly invoking a tool, or typing on an element that cannot be typed on), we sample another action from the agent with the Exception message appended to its context. We keep re-sampling until it chooses a syntactically correct action, or terminate the episode if the agent is unable to fix an erroneous action in $m = 5$ steps.

5.7 Main Results

Figure 5.3 compares the zero-shot baseline with agents augmented with BAGEL demonstrations. We find that using synthetic demonstrations as in-context exemplars, retrieved based on instruction relevance, lead to significant boosts in performance compared to the zero-shot agent. For the best variant of BAGEL, we find improvements of over 13% points on MiniWoB++, and over 2% on ToolQA. For MiniWoB++, our improvements are particularly strong (20% absolute) on *choose-date*, *tic-tac-toe*, and *use-autocomplete*. Solving these tasks successfully requires learning environment dynamics (e.g. Figure 5.1) which is enabled by BAGEL demonstrations. We isolate the source of

[1] While tuning T_{iter} on a dev set may lead to better results, we choose use a fixed value to remain truly zero-shot.

Dataset	Zero-Shot	instruction-first		trajectory-first	
		No-itrs	Full	No-itrs	Full
MiniWoB++	46.8	52.0	56.0	53.0	**61.0**
ToolQA	40.9	38.8	**43.3**	40.9	42.2

Table 5.1: Ablations showing the effect of multiple rounds of re-labeling in BAGEL. Multiple iterations improve performance for both instruction-first and trajectory-first variants.

these improvements from synthetic in-context exemplars in §5.8.1. Furthermore, trajectory-first exploration significantly outperforms instruction-first on MiniWoB++, which we hypothesize is due to the LM prior being misaligned with the distribution over possible instructions on MiniWoB++.

Finally, Table 5.1 shows that iterative re-labeling *always* improves performance over non-iterative baselines. Multiple iterations of round trips improves average reward by 4-8% on MiniWoB++ and 1.3-4.5% on ToolQA.

5.8 Analysis

To understand how BAGEL demonstrations improve agent performance, we first look at confounders from in-context learning (§5.8.1), and then study the impact of synthetic demonstrations on execution failures (§5.8.2). Next, we analyze the correctness (§5.8.3) and diversity (§5.8.4) of BAGEL's demonstrations to identify areas for further improvements.

5.8.1 In-context Learning with Synthetic Demonstrations

In-context exemplars can provide a range of useful learning signal to LM agents, ranging from simply providing examples of valid action trajectories or relevant natural language instructions in isolation, to providing rich information about the conditional $p(\tau \mid g)$ (how to map relevant instructions into action sequences). Indeed, for some text classification tasks, Min et al. [2022] find that improvements from in-context learning may be explained in terms of the former i.e. examples of the label space and input text. To better understand how synthetic demonstrations help in our setting, we report results from two ablations. First, we provide the model with randomly chosen demonstrations instead of using the retriever (**Random**). Next, we shuffle demonstrations so that trajectories are paired with randomly chosen instruction within the set of retrieved examples (**Shuffled**).

Method	Accuracy
Zero-shot	40.9
Random	38.0
Shuffled	41.4
Ours	**42.2**

Table 5.2: Ablations showing the effect of various sources of information in synthetic demonstrations to agent performance.

Task	Zero-Shot (↓)	+BAGEL (↓)
choose-date	1.3	**0.1**
book-flight	3.0	**0.6**
ToolQA (average)	3.0	**1.9**

Table 5.3: Average number of execution failures for tasks in MiniWoB++ and ToolQA. We find that using synthetic demonstrations reduces execution failures.

Results. Table 5.2 reports results of these ablations. First, **Shuffled** improves performance over the zero-shot baseline, suggesting that some of the improvements come from providing examples of valid action trajectories in the domain in line with findings in Min et al. [2022]. **Ours** records a further improvement of 0.8% over **Shuffled**, which suggests that the agent is able to use signal about the conditional to improve decision making.

5.8.2 Synthetic demonstrations reduce execution failures

As mentioned in §5.6.2, in our implementation, LM agents recover from execution failures using a re-sampling procedure—when the agent generates an invalid action (such as attempting to Type on a checkbox element or calling a tool with incorrect syntax), we re-prompt it with the error message produced by the environment, until it produces a valid action. Of course, such re-sampling can be costly at inference time due to multiple calls to the LM. Table 5.3 reports the average execution failures for tasks with re-sampling on MiniWoB++ and ToolQA. We note a considerable reduction in average re-sampling with BAGEL, due to a better understanding of environment dynamics, in turn leading to faster inference.

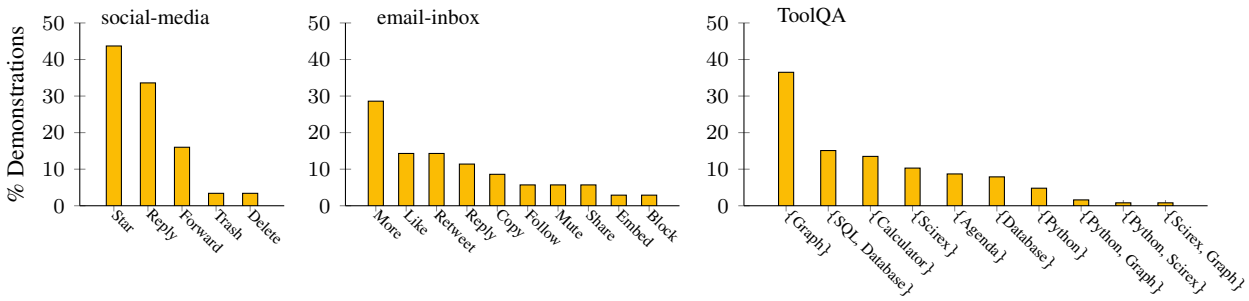


Figure 5.4: Distribution of demonstrations over semantic categories for MiniWob++ environments, *social-media* and `email-inbox`, and ToolQA. While BAGEL prefers certain modes, overall we find that these demonstrations cover a diverse range of actions.

5.8.3 Correctness of Synthetic Demonstrations

One way to identify the scope for improvements in our method is to manually verify the correctness of demonstrations. We filter demonstrations which, upon execution, do not achieve the corresponding instruction. Using these filtered demonstrations improves performance further by 7% absolute on all 10 tasks from MiniWoB++.

5.8.4 Diversity of Synthetic Demonstrations

To better understand the distribution of synthetic demonstrations, we manually bucket demonstrations for *social-media* and *email-inbox* into semantic clusters— for *social-media* these clusters include {Retweet, Like, Share, ...} and for *email-inbox* we have clusters such as {Forward, Delete, Star, Reply, ...}. For ToolQA, we cluster demonstrations based on the set of tools invoked in the demonstration. We plot the number of demonstrations in each cluster in Figure 5.4. We note that while this distribution tends to be skewed towards specific modes (e.g. {graph} for ToolQA, {Star} for email-inbox), there exists a long tail that covers a broad range of possible use cases in the environment. Nevertheless, improving diversity during exploration remains a failure mode for BAGEL which we expand on next. Finally, we provide some examples of BAGEL demonstrations in Table 5.4, along with their corresponding semantic category.

5.8.5 Error Analysis

We conclude with a discussion of failure modes of our aproach using the domains *book-flight*, *search-engine*, and *SciRex* as case studies.

Instruction	Trajectory
MiniWoB++	
Find the email by Trixi and reply to them with the text "Maecenas eu massa" **{Reply}**	Move Mouse to Trixi → Click on an email-thread → Click on the reply button → Type 'Maecenas eu massa' on the textarea with id 'reply-text' → Click on the span with id 'send-reply'
Find the email by Darcy and forward it to Dionis **{Forward}**	Click on Darcy, the sender of an email thread. → Click on 'forward' button → Type Dionis on the to field → Click on the 'send' button
Retweet Gallegos's post **{Retweet}**	Move Mouse to Pretium,. Ullamcorper. → Click on retweet element with id 101
Like tweet by @leonie and share tweet by @livia **{Like, Share}**	Click on the like element with ID 41. → Click on share-113
ToolQA	
What are David's plans this weekend? **{RetrieveAgenda}**	Retrieve passages related to David's plans this weekend → Finish with answer: On the evening of September 16th, 2022, David will be attending a Blind Date Night event at The Press Lounge.
Who is affiliated with both nicolas christin and giulia fanti? **{Python, Graph}**	Load DBLP → Check neighbours of node Giulia Fanti in graph AuthorNet → Check neighbours of node Nicolas Christin in graph AuthorNet → Evaluate python code: *list1=['Wanzheng Zhu', 'Rohan Bansal', 'Zachary Weinberg', 'Nicolas Christin', 'Suma Bhat', 'Hongyu Gong']; list2=['Wanzheng Zhu', 'Rohan Bansal', 'Zachary Weinberg', 'Suma Bhat', 'Hongyu Gong', 'Giulia Fanti']; ans=set(list1) & set(list2)* → Finish with answer: {'Hongyu Gong', 'Rohan Bansal', 'Wanzheng Zhu', 'Zachary Weinberg', 'Suma Bhat'}
What are the top 5 airbnb options with price < 900, availability > 260 and at least 40 reviews **{Database, SQL}**	Load database airbnb → Filter database according to price < 900, availability 365 > 260, number of reviews > 40 → Interpret SQLite query: *SELECT FROM airbnb_data ORDER BY number_of_reviews DESC LIMIT 5* → Finish with answer: ['High-end doorman bldg in the LES', 'THE BEST DEAL ON THE HUDSON RIVER!!', 'Heart of Williamsburg, Brooklyn!', 'Beautiful & Tranquil Oasis in a Great Location', 'Sunny/Cozy 1BD']
What are the different approaches for computing graph similarity? **{RetrieveSciRex}**	Retrieve passages from ML papers related to graph similarity → Finish with answer: The different approaches to computing graph similarity are graph kernels, graph features and graph convolutional neural networks (CNNs).

Table 5.4: Example demonstrations obtained via BAGEL for MiniWoB++ (top) and ToolQA (bottom). We also provide the semantic category for these demonstrations, and report the distribution of these categories in Figure 5.4.

Handling Long-Horizon Planning. We note that *book-flight* is the most complex environment in MiniWoB++, with longer trajectories of lengths 8-20, and the zero-shot policy performs poorly on this environment (average reward of 5%). While using BAGEL demonstrations improves this to 15%, we hypothesize that further improvements would require better handling of long range plans, such as with hierarchical planning [Sodhi et al., 2023, Jiang et al., 2019].

Improving Diversity. We hypothesize that improving diversity among seed trajectories would lead to further improvements across the board. For instance, for *book-flight*, all BAGEL demonstrations correspond to booking flights in December, while the test distribution is more uniform.

Reducing Mismatch with Test Instructions. On *SciRex*, all models fail to produce even a single correct answer. Here, we find that in the absence of any knowledge about user instructions at test-time, BAGEL demonstrations tend to create questions with more descriptive answers and trajectories with generic queries (See Table 5.4 for an example) while test instructions requires retrieving specific numbers from scientific documents by querying for specific topics. Similarly, on *search-engine*, we note a modest improvement of only 5%. Here, we find that while BAGEL demonstrations cover a variety of instructions like *Search for cat and navigate to the third page of search results, Search for cars, then visit the second search result*, the model fails on test instructions like *Enter [term] then find and click the 9th search result* that requires keeping track of the number of search results per page, and navigating to the correct page. While our goal is to build fully unsupervised agents, methods that use sparse information about test-time instructions could help drive performance further.

5.9 Related Work

Instruction-Following Digital Agents. Building agents that navigate the digital world is a long standing goal of AI and language understanding [Allen et al., 2007, Branavan et al., 2009]. However, most prior work relies on expert demonstrations [Liu et al., 2018, Humphreys et al., 2022, Furuta et al., 2023] with an appropriately shaped reward [Branavan et al., 2009, Liu et al., 2018]. Here, we assume no access to demonstrations or a reward function, and use pre-trained components to bootstrap synthetic demonstrations.

LMs for Decision Making. Pre-trained LMs are increasingly being used for sequential tasks such as robotic manipulation [Ahn et al., 2022, Liang et al., 2023], instruction-following [Yao et al., 2022,

Kim et al., 2023, Sun et al., 2023, Lù et al., 2024], and tool-use [Parisi et al., 2022]. While some of these approaches finetune LMs based on human demonstrations [Nakano et al., 2021], others use human demonstrations in their prompt for in-context learning and adaptation [Yao et al., 2022, Kim et al., 2023, Sun et al., 2023]. We use no human supervision or reward and adapt LM agents purely using synthetic demonstrations. Another line of work uses LM priors in reinforcement learning to improve exploration [Mu et al., 2022, Du et al., 2023], deal with large action spaces [Yao et al., 2020], or as proxy reward functions [Kwon et al., 2023a]. In the same tradition, BAGEL bootstraps a learning signal in the form of synthetic demonstrations by combining several LM components but without using RL.

Self-training for Language Models. A recent line of work uses LM-generated data for finetuning the same LM, in settings where external verifiers may be used to filter generated data [Singh et al., 2023, Gulcehre et al., 2023]. While we also use data generated from an LM for adaptation, unlike these approaches, environment interactions form a critical part of the learning signal and we also do not use external verifiers for filtering data.

5.10 Conclusion

There is a growing interesting in grounding LMs to the real world, by building helpful assistants that execute open-ended instructions in digital environments. The complexity of such sequential tasks makes collecting expert demonstrations tedious, and so, further progress towards building such agents requires new methods for bootstrapping a learning signal with minimal human supervision. To this end, we introduce BAGEL, a method for constructing synthetic demonstrations for instruction following agents. These demonstrations are constructed by iteratively relabeling an initial seed set of trajectories or instructions, where both relabeling and exploration is driven by a language model. Experiments on two different domains show that using BAGEL demonstrations as in-context exemplars leads to considerable improvements ranging from 2-13%, as well as significant reductions in execution failures.

Discussion. BAGEL was an early attempt by us to automatically mine demonstrations for grounding a zero-shot LLM agent in an environment. One of its key ideas was to use an exploration policy to collect trajectories and then map them into instructions—an approach largely inspired by Hindsight Experience Replay [Andrychowicz et al., 2017]. These demonstrations were used to adapt

the LLM via in-context learning, leading to significant improvements in agent performance. While BAGEL achieved substantial gains, it was primarily applied to simple web interfaces in MiniWoB++. Applying such an *interaction-based* method to real websites poses a significant challenge: the exploration space can grow exponentially due to the vast number of active elements. What is needed, then, is a robust mechanism to guide this exploration. In the next chapter, I introduce `NNetNav`, our latest method for collecting demonstrations through environment interaction. The key innovation in `NNetNav` is a pruning heuristic directly inspired by the hierarchical structure of language instructions. Notably, `NNetNav` enables us to train high-performing web agents out of relatively small (sub-10B parameter) LMs that can operate effectively on real websites.

Chapter 6

NNetNav: Unsupervised Learning of Browser Agents Through Environment Interaction in the Wild

6.1 Introduction

Building grounded agents that map human language instructions to a sequence of executable actions is a long-standing goal of artificial intelligence [Winograd, 1972]. A promising new approach for building such agents is to use large language models to control policies in environments like web-browsers and computers (e.g. Chapter 5).

Unfortunately, language models struggle with such grounded instruction following out-of-the-box because LMs do not know about the myriad and ever changing interaction possibilities of different websites. For instance, on a new e-commerce website, a zero-shot LM browser agent may struggle to make a return or change order details, without expensive test-time exploration. Even simple tasks like choosing a flight can involve different UI element such as directly entering airport codes or interacting with drop-down menus, and a zero-shot agent cannot know a priori the correct thing to do. The most common solution is to provide LM browser agents with knowledge about new web interfaces via expert demonstrations, that can either be used for in-context learning [Yao et al., 2022] or supervised fine-tuning [Lai et al., 2024, Shen et al., 2024]. These demonstrations are either fully provided by human experts [Sodhi et al., 2023, Yao et al., 2022] or consist of human-generated trajectories paired with model-generated instructions [Lai et al., 2024]. However, collecting human demonstrations that

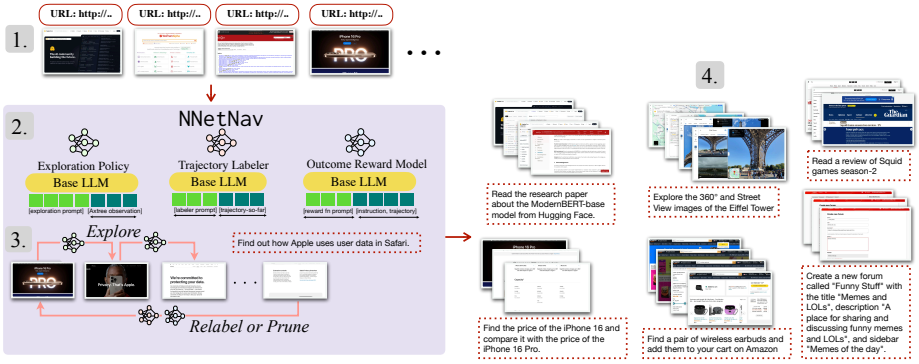


Figure 6.1: Given web URLs (1), `NNetNav` (2) uses a structured exploration strategy to interact with websites (3) and autonomously discover diverse (instruction, trajectory) demonstrations, as summarized in (4). To effectively prune exploration, the trajectory-so-far is periodically evaluated by a relabeling module and further exploration continues only if it can be assigned a meaningful language instruction. All components in `NNetNav` are implemented with the same zero-shot base LLM.

cover each possible use case for every website is an unattractively large, never-ending task. Thus, in this work, we propose a method for training LM browser agents in a *completely unsupervised way*, via synthetic demonstrations derived from interaction.

At a high level, our approach, `NNetNav` (Fig 6.2), uses a language model exploration policy to perform extended interactions with a website, and another language model trajectory labeler to annotate trajectories with instructions.[1] To effectively control the exponential space of meaningful interactions, `NNetNav` uses the hierarchical structure of language instructions as a pruning heuristic: for exploration to discover a meaningfully complex task, trajectory prefixes must correspond to meaningful sub-tasks. Thus, during an exploration episode, if a language model cannot label trajectory prefixes (at set time-steps) with a sub-task, further exploration is automatically pruned. Imposing such a structure over search not only enhances efficiency, but also results in complex and hierarchical instructions (See Table 5.4 for examples). `NNetNav` prompts the same base language model for exploration, relabeling and inferring sub-tasks.

We use `Llama-3.1-70B` [Dubey et al., 2024] to collect a large scale dataset of over 10k demonstrations (around 100k state, action transitions) from 20 websites, including 15 live, in-the-wild websites, and 5 self-hosted websites from WebArena [Zhou et al., 2023]. We classify these instructions into various intents and find a highly diverse range of internet use cases, including *flight booking*, *finding recipes*, *buying iPhones*, *searching for trails*, *commenting on github issues*, and

[1] Our code, data and trained models are available at https://nnetnav.dev/.

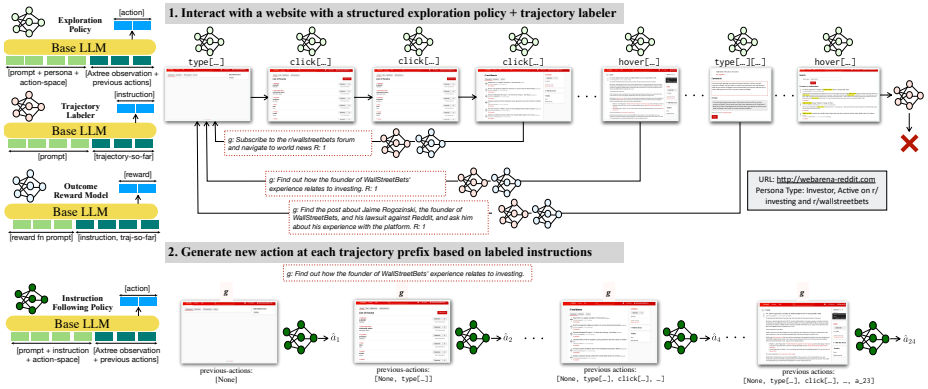


Figure 6.2: **Left:** `NNetNav` uses four components to interact with websites to create training examples, built out of zero-shot language models. **Right (Top):** An exploration episode on a website begins with sampling a persona, followed by generating persona-conditioned action sequences from the exploration policy. At fixed intervals, the trajectory labeler infers an instruction to describe the trajectory so far. If the resulting (instruction, trajectory) pair receives a low score from the ORM, the episode is pruned (indicated by a red cross). **Right (Bottom):** For each instruction, we retroactively generate a new action, given the (instruction, observation, previous actions) tuple to ensure that actions at each time-step correspond directly to the inferred instruction.

posting on Reddit. We use these demonstrations for supervised fine-tuning of `Llama-3.1-8B`. On WebArena, our model achieves a success rate of 16.3%, outperforming zero-shot `GPT-4` by 2 points and reaching state-of-the-art performance among unsupervised methods. On WebVoyager [He et al., 2024], our best model reaches a success rate of 35.2%, outperforming zero-shot `GPT-4` by 1.7 points and all known open methods on this task to the best of our knowledge. Interestingly, we find that `NNetNav` enables effective self-training—fine-tuning a smaller LM using `NNetNav` demonstrations generated by the same model yields a 4 point absolute improvement (from 1% to 5%) on WebArena. `NNetNav` opens up interesting avenues for open-ended discovery of workflows on unknown web-interfaces, without human supervision.

6.2 Background

Following instructions on a web-browser is a multi-turn sequential decision making problem. Given an instruction g, a browser agent interacts with the browser by issuing a sequence of *computer control* actions $\langle a_1, a_2, \ldots, a_T \rangle$ where each $a_i \in \mathcal{A}$ is drawn in response to an observation o_i. Executing an action causes a state transition based on some unknown environment dynamics, leading to a new observation o_{i+1}. The entire episode can be summarized as a *trajectory* $\tau :=$

$\langle o_1, a_1, o_2, a_2, \ldots o_{T-1}, a_T, o_T \rangle$. We formalize the instruction following agent as a mapping $\pi(a_t \mid o_t, \tau_{<t}; g)$ where $\tau_{<t} \coloneqq \langle o_1, a_1, \ldots a_{t-1} \rangle$ is the trajectory so far. In our case, observations are represented as either flattened DOM trees or website accessibility trees, and $\mathcal{A}$ consists of keyboard / mouse commands that operate on elements of these trees (see Appendix C for the full action space).

LLMs for Browser Control. Recent work explores using instruction-tuned large language models (LLMs) to directly parameterize the agent. These methods typically work in settings with textual observations and action spaces. At time-step t, the agent π_{LM} is provided with the following context: the instruction g, the full action space described as a string, the current observation o_t, and some representation of the trajectory-so-far $\tau_{<t}$, typically the action history. Given this information, the LLM generates an output that is parsed into an action. Typically, the LLM output contains both a reasoning step r_t (e.g. *Since my task is to buy a mug, given the current state, I should click on the buy now button*), and the chosen action command a_t (e.g. *click [1234]*).

Given expert demonstrations $\{g^i, \tau^i\}$ where $\tau^i \coloneqq \langle o_1^i, r_1^i, a_1^i, o_2^i, r_2^i, a_2^i \ldots o_T^i \rangle$, previous work adapts LM agents using demonstrations as in-context examples [Yao et al., 2022, Shinn et al., 2023, Sun et al., 2023, Kim et al., 2023, among others] or as training data for supervised fine-tuning [Furuta et al., 2023, Lai et al., 2024, Lù et al., 2024, Patel et al., 2024]. For supervised fine-tuning of π_{LM} on a dataset of demonstrations, we construct training instances $\{(g^i, \tau^i_{<t}, o^i_t), (r^i_t, a^i_t)\}$ where r^i_t, a^i_t serves as the target reasoning step and action for an intermediate context $(g^i, \tau^i_{<t})$.

Prior Methods for Synthetic Demonstrations. Since collecting human demonstrations for browser agents is time consuming and costly, recent work uses synthetic demonstrations as training data [Lai et al., 2024, Furuta et al., 2023, Murty et al., 2024]. These methods start by sampling synthetic instructions from an instruction generator (a prompted LM that takes the website landing page and an optional user persona), and then use a zero-shot browser agent to convert these instructions into trajectories. Resulting demonstrations are filtered using either the ground truth reward function [Furuta et al., 2023], or using another LM outcome reward function [Lai et al., 2024, Murty et al., 2024]. Most of these methods fine-tune smaller LMs using synthetic demonstrations from larger LMs.

Such *instruction-first* methods for data collection face several challenges. First, synthetic instructions in these demonstrations are sampled from an ungrounded LM prior that generates only plausible[2] instructions without ensuring feasibility; *e.g.*, an instruction such as *Delete the first post*

[2]We use the term *plausible* for instructions that match a website's genre or intended use. For example, searching for

on r/callofdutyfans for reddit is plausible, but not always feasible. Second, generated instructions are limited to those that reference visible features of the website; *e.g.*, given the landing page of a github-like platform, no LM prior can generate instructions like *Find information about Eric Bailey's contributions to the byteblaze project*, which require knowing about deeply embedded website-specific entities like *Eric Bailey*. Finally, these methods provide no control over the complexity of instructions, and rely entirely on the LM or bespoke prompts to generate complex instructions.

6.3 Our Approach

Instead of starting with a sampled instruction, we start by sampling an *interaction* first, and then retroactively labeling it into an instruction that is feasible by definition. `NNetNav` (Fig 6.2) is an *interaction-first* method for constructing demonstrations: An exploration policy interacts with a browser in a structured manner to sample long trajectories which are retroactively labeled into instructions (§6.3.2). We then post-process each trajectory to add post-hoc actions corresponding to the generated instructions.

6.3.1 LM Components

All components in `NNetNav` are implemented with a zero-shot instruction-tuned LLM, with different prompts (see Appendix C for prompts).

Exploration Policy. To interact with the environment, we use an exploration policy π_{explore}, implemented as -a prompted language model, similar to π_{LM}. Additionally, to simulate a diverse set of behaviors from users and improve the diversity of resulting trajectories, we seed each episode with a string description of a plausible user persona for the given website [Shanahan et al., 2023, Argyle et al., 2023]. At each time-step, π_{explore} is provided with the following context: a user persona, the list of available actions, the current observation o_t, and the action history. The output of π_{explore} is then parsed into an action.

Summarizing Trajectory changes. Actions issued by π_{explore} result in a new observation in the environment. We summarize this change as a short string description via another module Δ_{LM}, implemented using a language model. In particular, for any state o_t, action a_t and the resulting next state o_{t+1}, $\delta_t = \Delta_{\text{LM}}(o_t, a_t, o_{t+1})$ produces a string-valued description of the changes in the

clothes on a retail site or checking notifications on a social media platform. Not all plausible instructions are feasible.

observation as a result of the action. For a trajectory τ, we denote the sequence of state changes as δ_τ.

Trajectory Labeler. Given δ_τ, the trajectory labeler $\mathrm{Lf}_{\mathrm{LM}}$ produces a plausible instruction $\hat{g} = \mathrm{Lf}_{\mathrm{LM}}(\delta_\tau)$ that the agent could have followed to produce the given interaction.

Outcome Reward Model. Given $\hat{g}$ and δ_τ, the outcome reward model (ORM) assigns a reward $s_{\mathrm{LM}}(\hat{g}, \delta_\tau) \in \{0, 1\}$, based on the degree to which state changes correspond to the given instruction $\hat{g}$.

6.3.2 Sampling Demonstrations via Interactions

At specific time steps $t \in \{t_1, t_2, \ldots, t_{\text{max}}\}$, we apply a pruning heuristic to retroactively label the current trajectory. Given a partial trajectory $\tau_{<t}$ after interacting with the environment for t steps, we compute a sub-task annotation $\hat{g} = \mathrm{Lf}_{\mathrm{LM}}(\delta_{\tau_{<t}})$. If this sub-task receives no reward, i.e., $s_{\mathrm{LM}}(\hat{g}, \delta_{\tau_{<t}}) = 0$, we prune the episode and sample a new rollout. Otherwise, we store $(\hat{g}, \tau_{<t})$ as a synthetic demonstration and continue exploration. Each episode typically generates multiple such demonstrations.

Post-processing with an Agent Policy. Actions at each time-step in our our demonstration set are a result of un-directed exploration, and therefore might not be optimal for the retroactively generated instruction. Thus, we post-hoc annotate each state with a new action that directly corresponds to the generated instruction. Concretely, given every $(\hat{g}, o_i, \tau_{<t})$ tuple in our synthetic demonstration set, we use π_{LM} to output a suitable action $\hat{a}_i$ given the instruction $\hat{g}$ and current observation o_i.

BofK sampling (Optional). To further boost the quality of trajectories, we optionally use best-of-K (BofK) sampling. In particular, given `NNetNav` generated instructions, we sample K-1 additional trajectories, with π_{LM} using the same base LLM. Then, for each instruction, we use our ORM to score each of the K-1 trajectories and the original trajectory, and pair the best trajectory with the given instruction, breaking ties arbitrarily.

6.4 Main Experiments

6.4.1 Collecting Demonstrations in the Wild

We apply `NNetNav` on 20 websites to collect a dataset of over 10,000 demonstrations. We consider 15 live websites (same set as He et al. 2024): Allrecipes, Amazon, Apple, ArXiv, BBC News, Booking, Cambridge Dictionary, Coursera, ESPN, GitHub, Google Flights, Google Map, Google Search, Huggingface, and Wolfram Alpha, and 5 self-hosted websites from WebArena (WA; Zhou et al., 2023).

We use instruct-tuned `Llama-3.1-70b` as the base LLM for all components in `NNetNav`, with t_{max} set to 40, running `NNetNav` pruning every 4 time-steps at {4, 8, 12, 16, ..., 40}. Additionally, we perform BofK sampling with $K = 3$, using π_{LM} (with the same `Llama-3.1-70b` base model). While we only consider text based browser agents in this work, we release both accessibility tree strings as well as browser screenshots at each time step, to support future work on multi-modal browser agents.

Difficulty	**`NNetNav` (WA)**	**`NNetNav` (Live)**
Easy	498	1448
Medium	2532	2369
Hard	1164	1204
Very-Hard	501	556
Total	4695	5577

Table 6.1: We report the breakdown of `NNetNav` demonstrations into categories defined based on the number of actions in the trajectory.

Diversity and Complexity. To evaluate diversity in resulting instructions, we cluster them by intent for each website. We obtain these intents through a two-step procedure—we input instructions for each website into `GPT-4o`, prompting it to identify common intents, and then classify each instruction into one of these intents in a second forward pass. On average, we identify 21 intents per website for self-hosted websites and 25 for live websites. Analyzing the distribution of these intents, we observe an average perplexity (PPL) of 13.5 for self-hosted sites and 16.2 for live websites. Higher perplexity suggests a more evenly distributed set of intents, indicating substantial diversity in the collected demonstrations. We provide a visual representation of this distribution as a sunburst plot in Appendix C.

Agent	#Params	WebArena SR	WebVoyager SR	Human Supervision Used?
		Using Closed Models		
GPT-4 [Zhou et al., 2023, He et al., 2024]	Unknown	14.1	33.5	✗
GPT-4-AWM [Wang et al., 2024b]	Unknown	35.5	-	✗
GPT-4 + LLama-70b [Shen et al., 2024]	Unknown	50.0	-	✓
		Using Open Models		
`Llama-3.1-8b`	8B	1.0	4.4	✗
Lai et al. [2024]	7B	2.5	-	✗
Ou et al. [2024]	7B	6.3	-	✗
Patel et al. [2024]	72B	9.4	-	✗
LLaVa-7B PAE + Claude [Zhou et al., 2024]	7B	-	22.3	✗
LLaVa-34B PAE + Claude [Zhou et al., 2024]	34B	-	33.0	✗
Qwen2.5-7B-AgentTrek [Xu et al., 2024]	7B	10.5	-	✗
Qwen2.5-32B-AgentTrek [Xu et al., 2024]	32B	16.3	-	✗
`Llama8B-NNetNav-WA` **(Ours)**	8B	**16.3**	28.1	✗
`Llama8B-NNetNav-Live` **(Ours)**	8B	9.5	**35.2**	✗
`Llama8B-NNetNav-All` **(Ours)**	8B	14.9	34.1	✗

Table 6.2: We present average success rate (SR) on browser tasks from WebArena and WebVoyager for various approaches, along with key details such as model size, the use of open LLMs and human supervision. For Lai et al. [2024], we report results from the setting that does not use human supervision. Zero-shot `GPT-4` results are sourced from Zhou et al. [2023] and He et al. [2024]. The last three rows report the performance of our fine-tuned `Llama-3.1-8b` agents, which achieve state-of-the-art results, outperforming zero-shot `GPT-4` and outperforming or matching prior open-model approaches with significantly fewer parameters, across both benchmarks.

To analyze the complexity of demonstrations, we categorize each demonstration into one of four levels based on the number of action sequences: *easy* (fewer than 5 actions), *medium* (5 to 10 actions), *hard* (10 to 20 actions), and *very hard* (over 20 actions). Table 6.1 presents the distribution of demonstrations across these categories, showing a substantial number of complex demonstrations.

6.4.2 Finetuning: Details and Results

We perform supervised fine-tuning of the smaller instruct-tuned `Llama-3.1-8B` with `NNetNav` demonstrations. To measure transfer between knowledge learned from live websites and self-hosted WebArena websites, we fine-tune on: only WebArena websites (`Llama8B-NNetNav-WA`), only live websites (`Llama8B-NNetNav-Live`), and all websites together (`Llama8B-NNetNav-All`).

As described in Section 2, each demonstration expands into multiple training instances, resulting in a total of 100k training examples for the full dataset. We fine-tune for 2 epochs with a batch size of 128, truncating the max sequence length to 20000, with a learning rate of 2e-5, that is warmed with a linear scheduler over 500 gradient updates (more details can be found in Murty et al. [2025]).

We use open-instruct [Wang et al., 2023] for fine-tuning, and set up local inference servers using VLLM [Kwon et al., 2023b]. During inference, we sample with a temperature of 0.01 and perform nucleus sampling [Holtzman et al., 2019] with top-p set to 0.9.

Benchmarks. We evaluate models on 812 tasks from WebArena [Zhou et al., 2023] and 557 tasks from WebVoyager [He et al., 2024], omitting tasks in Google Flights and Booking, as they are no longer feasible (following Zhou et al., 2024). For WebArena, we report averaged success rate (SR) across all tasks based on the provided evaluator that measures functional correctness. For WebVoyager, we use the author-provided script that uses GPT-4V to judge success based on instructions and browser screenshots at each time step. We report the average across all websites.

Results. We report our results in Table 6.2, where we present prior results from using closed models (typically `GPT-4o`) as well as with open models. On WebArena, both `Llama8B-NNetNav-WA` and `Llama8B-NNetNav-All` outperform zero-shot `GPT-4o`, with our best model achieving state-of-the-art performance among unsupervised methods. On WebVoyager, `Llama8B-NNetNav-Live` and `Llama8B-NNetNav-All` surpass zero-shot `GPT-4o`, establishing a new state-of-the-art among open-source methods. Notably, they outperform the previous best OSS result from Zhou et al. [2024], which relied on a significantly larger 34B-parameter vision-language model (VLM) and a closed-model verifier. Interestingly, we find that `Llama8B-NNetNav-WA`, which is trained exclusively on WebArena websites, exhibits poor transfer to live websites. We analyze cross-website transfer next.

6.4.3 Cross-Website Transfer

We present per-website success rates of our fine-tuned models across all 18 websites in Table 6.3. For WebArena websites, by comparing columns 2 and 3, we find that 3 out of 5 websites benefit from incorporating in-domain data. By comparing columns 1 and 3, we observe an average performance drop of 1.8 points, with the most significant decrease on the *Maps* domain. This decline is likely due to the semantic search capabilities in *Google Maps*, which are absent in WebArena *Maps*, necessitating more complex query formulation. For live websites, fine-tuning on in-domain live website data improves performance on 10 out of 13 domains, as indicated by comparing columns 1 and 3. The effect of incorporating out-of-domain WebArena data, however, is mixed. While it results in negative transfer for 7 websites and positive transfer for 6, the overall average performance decreases by 1.3 points. Notable gains are observed in *ESPN*, *Apple*, and *GitHub*, suggesting potential synergies when fine-tuning on closely related domains.

Website	NNetNav (WA)	NNetNav (Live)	NNetNav (Live+WA)
	Self-hosted Websites (WebArena)		
Reddit	**26.3**	9.6	25.4
Gitlab	**18.4**	5.6	16.8
Maps	**15.6**	14.8	10.9
CMS	**11.5**	5.5	9.9
Shopping	**13.0**	9.9	**13.0**
	Live Websites (WebVoyager)		
Allrecipes	26.7	**37.8**	29.5
Amazon	24.4	**43.9**	34.1
Apple	32.6	27.9	**34.9**
ArXiV	27.9	**46.5**	44.2
BBC News	33.3	**42.9**	28.6
Cambridge Dictionary	46.5	**58.1**	48.8
Coursera	**47.6**	45.2	42.9
ESPN	20.5	22.7	**27.3**
GitHub	12.2	17.1	**19.5**
Google Maps	34.1	**46.3**	43.9
Google Search	0.0	2.7	**6.2**
Huggingface	**30.2**	18.6	**30.2**
Wolfram Alpha	26.1	43.5	**45.7**

Table 6.3: Per-website success rates on all websites, using a `Llama-3.1-8b` agent fine-tuned on (1) the WebArena subset of `NNetNav`, (2) the live website subset of `NNetNav`, and (3) all demonstrations. On WebArena, incorporating in-domain data improves performance on 3 out of 5 websites (comparing columns 2 and 3). For live websites, incorporating in-domain data improves performance for 10 out of 13 websites (comparing columns 1 and 3). These results highlight the importance of scalable methods to enable training on diverse websites.

Overall, fine-tuning with in-domain website data improves performance on 13 out of 18 websites. These findings underscore the importance of learning from unsupervised interaction on real websites, as relying solely on human-labeled trajectories from a limited set of simulated websites may be insufficient for developing generalist web agents.

6.4.4 Scaling Trends for Multi-domain Training

Table 6.2 suggests that incorporating data from live websites can hurt performance on WebArena. Is this a fundamental consequence of conflicting task information in multi-task learning—an issue that is well documented (e.g., see [])—or is it simply due to the model sizes used? To investigate this, we fine-tune a series of Qwen-2.5-Instruct [Yang et al., 2024] models with sizes {0.5B, 1.5B, 3B, 7B, 14B, 32B}. These models are trained under two settings: (1) WebArena-only, where the model is fine-tuned solely on the WebArena subset of the `NNetNav` data, and (2) WebArena+OpenWeb, where the model is fine-tuned on the full `NNetNav` dataset, including both WebArena and live website data.

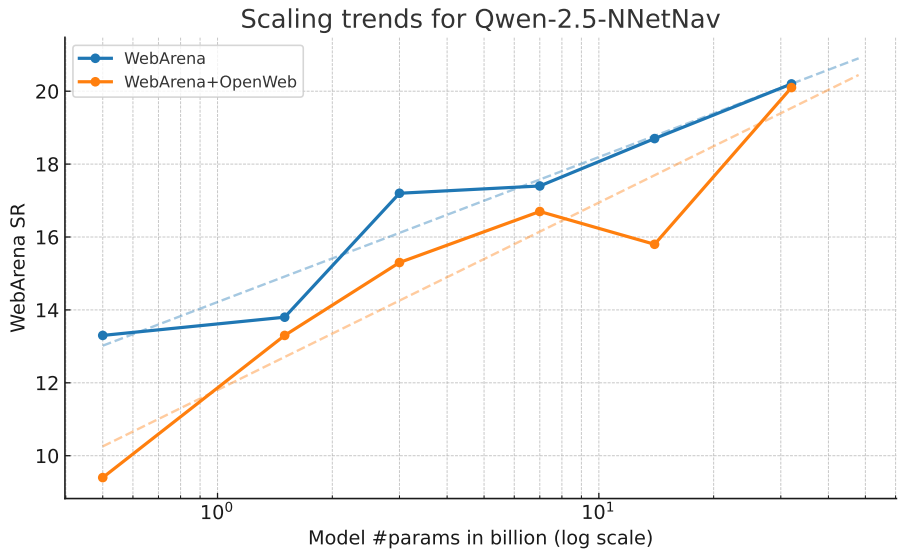


Figure 6.3: Scaling trends for Qwen-2.5-Instruct models fine-tuned on WebArena and WebArena+OpenWeb data. We plot the average WebArena success rate (SR) against model size (log scale). As model size increases, both settings show improved performance, with the gap between them narrowing. The slope for the full dataset (WebArena+OpenWeb) is steeper, suggesting faster improvement with scale.

We use a validation set for early stopping and train for a maximum of 3 epochs. Further hyperparameter details are provided in Murty et al. [2025].

Results. Figure 6.3 shows the results, with the x-axis on a log scale representing model size and the y-axis showing the average WebArena success rate (SR). We also include linear regressions to highlight the trend lines. As expected, performance improves consistently with model size—WebArena SRs rise from around 13.5% for the smallest model to over 20% for the largest. Notably, the performance gap between the WebArena-only and full-dataset settings narrows as model size increases. Moreover, the slope for the full-dataset models is steeper, indicating faster performance gains with scale. These results suggest that (a) the impact of conflicting gradients from live website data diminishes with scale, and (b) incorporating live data leads to faster improvements in WebArena performance as model size grows, although this is not definitive.

6.4.5 Error Analysis

We analyze failure modes of our best models by manually annotating failed trajectories and categorizing them into fine-grained failure types. Specifically, we select 4 to 6 failure cases per website, prioritizing trajectories that exhibit distinct error patterns. Each failed trajectory is manually annotated with free-form natural language comments describing the nature of the failure *e.g. correctly navigates to cornell (maintainer of arxiv), but couldn't navigate to the section where there is info about undergraduate enrollment*. To categorize these failure cases, we use `GPT-4` to cluster similar annotations into distinct failure attributes. We then manually refine these attributes by merging redundant categories. Next, we prompt `GPT-4` to assign a score of {-1, 0, 1} to each trajectory based on these fine-grained attributes, where a score of 0 indicates *not applicable*, 1 indicates *positive reward* and -1 indicates *negative reward*.

Figure 6.4 presents the average reward per attribute for WebArena and WebVoyager. We observe that while our agents exhibit strong performance in element interaction and search functionality, they struggle with navigation efficiency, and sometimes execute redundant steps before reaching the target. Additionally, extracting information from structured data (*e.g.*, tables on ESPN.com) remains a significant challenge. These findings suggest that future improvements should focus on heuristics to minimize unnecessary actions and enhancing the model's ability to parse and retrieve structured web content.

6.5 Controlled Experiments

We conduct controlled experiments on a smaller scale to compare `NNetNav` with baselines. In addition to evaluating on WebArena, we also consider MiniWoB++ [Shi et al., 2017, Liu et al., 2018]. MiniWoB++ is a dataset of synthetic web-interfaces with a shared action space. Tasks on MiniWoB++ range from clicking on buttons to complex tasks like making a booking on a website. We use a subset of 8 complex tasks from MiniWoB++ as a toy benchmark to evaluate our method. We use the `bid`-based action space from BrowserGym [Drouin et al., 2024], consisting of 12 actions, and a DOM based observation space. Due to its synthetic nature, MiniWoB++ comes with an automatic reward function. We report the mean reward over 20 random seeds for each task, similar to Drouin et al. [2024].

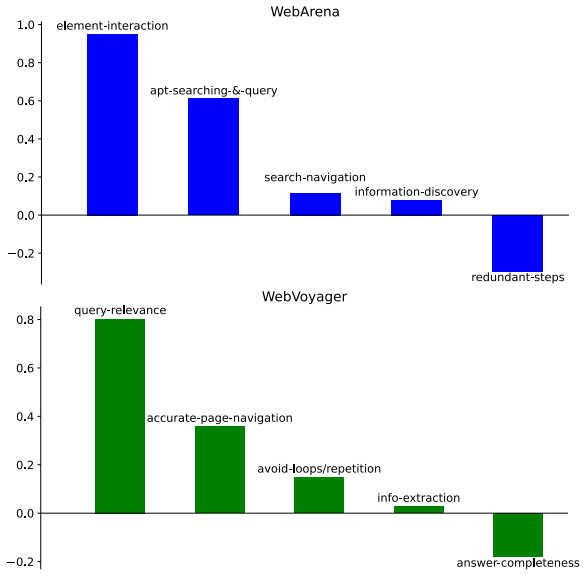


Figure 6.4: We score trajectories produced by our best models along 5 fine-grained attributes that were obtained by categorizing manually writing comments on model failures, for both WebArena and WebVoyager. We find that models can accurately interact with web-elements and conduct accurate searching and querying, however avoiding loops and redundant steps remains a challenge, as well as discovering information on web-sites by issuing multiple queries and piecing together information.

6.5.1 Experimental Settings

As before, we evaluate a `Llama-3.1-8b` based browser agent under the following settings:

1. **Zero-Shot:** A baseline zero-shot agent, prompted using chain-of-thought prompting [Wei et al., 2022]. Next, we consider various fine-tuned models.

2. **SFT (Instruction-First):** Supervised fine-tuning of the `Llama-3.1-8b` agent using data collected via instruction-first sampling. Here, we use the same reward model for filtering demonstrations as `NNetNav`, and also sample the same number of demonstrations for fair comparison.

3. **SFT (`NNetNav`):** Supervised fine-tuning of the `Llama-3.1-8b` agent with demonstrations collected via `NNetNav`.

4. **SFT (`NNetNav` + Distil.):** Ablation, where we only retain instructions found via `NNetNav` and re-generate trajectories by prompting the same large LM as an agent. We use this setting to isolate performance improvements attributable to `NNetNav` trajectories.

For these small scale experiments, we use `gpt-4o-mini-2024-07-18` as the base LLM for both `NNetNav` and instruction-first methods. For Instruction-first data collection, we sample 50 instructions per website for WebArena, and 80 instructions per interface in MiniWoB++, and prompt the instruction generator with the landing page as well as a persona (to improve diversity). For `NNetNav`, we use our exploration policy to generate 50 episodes per website for WebArena, and 80 episodes per interface for MiniWoB++. We set T_{max} to 40 for WebArena, and 20 for MiniWoB++. For both MiniWoB++ and WebArena, we apply the pruning function every 4 time-steps. We use 16 persona types per website for WebArena, and 10 persona types per web-interface for MiniWoB++.

Model Setting	**MiniWoB++**	**WebArena**
Zero-Shot	0.28	1.0
SFT (Instruction-First)	0.28	4.2
SFT (`NNetNav`)	**0.48**	**7.2**
SFT (`NNetNav` + Distil.)	0.36	6.0

Table 6.4: Controlled evaluation of `NNetNav` with instruction-first methods. We present results for MiniWoB++ and WebArena, averaged across domain, reporting mean reward for MiniWoB++ and task success rate (SR) for WebArena. Fine-tuning with `NNetNav` leads to the largest improvements: from 28% to 48% on MiniWoB++; from 1% to 7.2% on WebArena.

6.5.2 Results

`NNetNav` outperforms instruction-first methods. We report results from all settings in Table 6.4. Fine-tuning `Llama-3.1-8b` using synthetic demonstrations generated by `NNetNav` yields significant improvements: an increase of 20 points on MiniWoB++ and over 6 points on WebArena. Notably, `NNetNav` outperforms instruction-first methods by a substantial margin, with gains of 12 points on MiniWoB++ and 1.2 points on WebArena. Interestingly, SFT (`NNetNav`) outperforms SFT (`NNetNav` + Distil.) on both MiniWoB++ and WebArena. This difference likely stems from the distinct procedures used to generate trajectories. In `NNetNav`, the model first acts, and the corresponding instruction is inferred afterward through a hindsight procedure. In contrast, `NNetNav` + Distil. provides the instruction upfront, sampling the trajectory later.

Computational savings from `NNetNav` pruning. We visualize overall improvements in exploration efficiency in Fig 6.5. Each horizontal line depicts the fraction of interaction episodes that

Domain	Zero-Shot	Self-Train (NNetNav)
Shopping	3.8	**15.4**
CMS	0.0	0.0
Reddit	0.0	0.0
Gitlab	0.0	0.0
Maps	0.0	**7.1**
Avg.	1.0	**5.3**

Table 6.5: We generate NNetNav demonstrations using Llama-3.1-8b, which we use for supervised fine-tuning of an agent based on the same LM, and find significant improvements on WebArena from 1% to 5.3%.

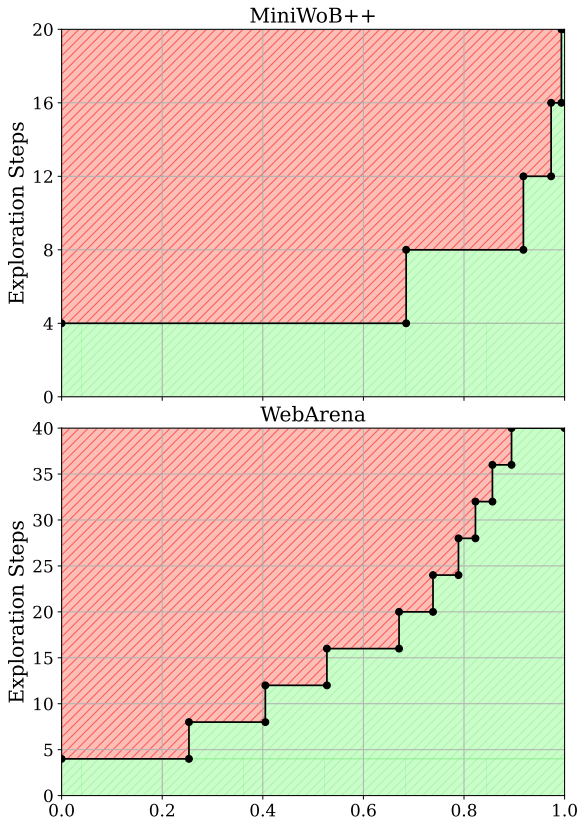


Figure 6.5: Horizontal lines indicate fraction of episodes terminating at corresponding y-axis exploration step. The red shaded area represents prevented actions, showing significant savings on both datasets.

terminate at a specific time-step (indicated by the y-axis), with the red shaded area depicting additional actions that were prevented from early pruning. We find clear evidence of computational

savings. In particular, over 60% of all exploration episodes were pruned after 16 actions for WebArena. For MiniWoB++, 65% of episodes were pruned after just 4 actions in MiniWoB++, which we identify as interactions where these first actions resulted in execution failures that our pruning heuristic successfully identified.

Self-training with `NNetNav`. Can `NNetNav` demonstrations from an LM be used for improving the *same* LM agent? To answer this, we collect another set of `NNetNav` demonstrations on WebArena, using `Llama-3.1-8b` as the base LM for data collection. Given the limitations of this smaller model, we anticipate fewer meaningful interactions. To compensate, we increase the number of episodes to 200 episodes per website, resulting in 302 demonstrations which we use for fine-tuning the same `Llama-3.1-8b` agent. From results in Table 6.5, we find improvements of 4.3 points on WebArena.

6.6 Related Work

Language Conditioned Digital Assistants. Mapping instructions to actions in digital environments has been a long-standing goal in natural language understanding [Allen et al., 2007, Branavan et al., 2009]. Most pre-LLM approaches for this rely on expert demonstrations for behavioral cloning [Chen and Mooney, 2011, Humphreys et al., 2022], along with appropriately shaped reward functions [Branavan et al., 2009, Liu et al., 2018, Misra et al., 2017, among others]. Here, learning is driven purely by synthetic demonstrations derived via (language model) exploration of websites.

Linguistic Priors for Exploration. Several prior works have used natural language priors to inform exploration for sequential decision making. Harrison et al. [2017] use a *trained model* of associations between language and state/action pairs to guide exploration during policy learning. Mu et al. [2022] use language annotations of states to train a goal generator module that provides intrinsic rewards for achieving generated goals. Similarly, Du et al. [2023] constrain exploration towards goals generated by a pre-trained LLM at each intermediate state of an agent. In constrast, `NNetNav` biases exploration through two news ways of using language priors. First, we use natural language as a way to filter meaningful interactions. Second, we use it as a pruning heuristic to navigate the potentially exponential search space of these interactions.

Training Data for LLM browser agents. LLMs have shown strong performance over a wide range of language understanding tasks, and are increasingly being used to interpret language in

grounded contexts such as browsers [Yao et al., 2022, Lai et al., 2024, Wang et al., 2024b, Patel et al., 2024, Lù et al., 2024, among others]. Many of these approaches rely on human demonstrations, either for in-context learning [Yao et al., 2022, Sodhi et al., 2023, Kim et al., 2023] or for finetuning [Lù et al., 2024, Shen et al., 2024]. Since human demonstrations are costly, recent work trains LLM agents through synthetic demonstrations generated using instruction-first methods [Lai et al., 2024, Patel et al., 2024]. One exception is Murty et al. [2024], which introduces an interaction-first method for generating synthetic demonstrations for in-context learning. Despite its novelty, their approach does not scale well to real websites due to the lack of a mechanism for effective exploration in environments with many possible interactions. In contrast, `NNetNav` also follows an interaction-first approach but improves efficiency by leveraging linguistically motivated pruning to navigate the space of meaningful interactions.

6.7 Conclusion

We propose `NNetNav`, a method for unsupervised interaction with websites "in-the-wild" that enables training browser agents with synthetic demonstrations. `NNetNav` flips the standard paradigm of synthetic data generation by first interacting with a website to generate trajectories and then hindsight relabeling trajectories into instructions. Real websites have a prohibitively large set of possible interactions; `NNetNav` searches over this space efficiently using a pruning function inspired by the hierarchical structure of language instructions: any complex instruction consists of language describable sub-tasks and so, if during an interaction a relabeling module cannot infer a meaningful sub-task for the trajectory-so-far, further exploration is pruned. We apply `NNetNav` to collect a diverse and complex set of 10k demonstrations from 15 live-websites and 5 self-hosted websites. We use these demonstrations for supervised finetuning of a small, `Llama-3.1-8b` model, achieving state-of-the-art results for unsupervised methods on both the WebArena and WebVoyager, surpassing zero-shot `GPT-4` by 1.7 to 2.2 points. `NNetNav` opens up the possibility of scaling up training data for generalist web agents across a broad range of web interfaces without any human intervention.

Chapter 7

Conclusion

The incredible advances in language understanding over the past six years, driven largely by the unsupervised learning revolution, have fueled widespread optimism about general-purpose agents. However, as this thesis argues, important technical challenges must be addressed to fully realize this vision. In particular, to build *reliable* agents, we need robust linguistic priors as well as methods for learning from environment interaction.

The first part of this thesis focuses on one crucial form of these linguistic priors: the ability to build meaning in a bottom-up, compositional manner, which is essential for systematic and reliable generalization. Chapter 2 demonstrates that despite having no explicit inductive bias for compositionality, transformers trained on natural language with stochastic gradient descent gradually become more tree-like. This has important implications for their *behavior*, as explored in this chapter and in Chapter 3. In Chapter 4, we go a step further by introducing stack-based memory to transformers. Pushdown Layers are designed to *expand* the range of functions transformers can express. Motivated by formal results showing that transformers cannot robustly handle arbitrarily deep recursive structures, Pushdown Layers augment the model with a stack that can "push" and "pop" memories, much like a programming language, enabling it to better handle deeply nested or recursive patterns.

The second part of this thesis turns to the challenge of autonomously learning about new environments. Learning here corresponds to gradient descent on automatically discovered demonstrations. The methods I present, BAGEL and NNetNav, propose schemes for generating these demonstrations through exploration. BAGEL (presented in Chapter 5) leverages a pre-trained LLM exploration policy to sample trajectories and a labeler component to assign language goals, iteratively refining them until an equilibrium is reached. These demonstrations are then added to a large vector database, indexed

by the vector representations of goals. At test time, an LLM agent is "grounded" in the environment via in-context learning: given a new goal, the top-k nearest demonstrations are retrieved and used as examples for the agent. Chapter 6 extends this idea further by proposing a new pruning heuristic to structure exploration in rich, complex environments like real-world websites. This approach enables us to collect over 10,000 demonstrations, which are then used to fine-tune LLM agents, achieving state-of-the-art performance on multiple browser-based instruction-following benchmarks. Together, these chapters offer a way forward in addressing the ongoing debate about language models being trained purely on form [Bender and Koller, 2020]. Demonstrations provide a clear mapping from form (language goals) to meaning (the sequence of actions that accomplishes those goals). I believe that large-scale training on interaction data could move us closer to the grounded language understanding systems we ultimately aim to build.

Despite the significant disruptions that digital agents are poised to bring—displacing traditional knowledge work, automating office tasks, and driving profound shifts in the labor market—the long-term outlook for these technologies offers a more hopeful and transformative vision. Over time, these agents have the potential to move beyond merely replacing jobs, to becoming sophisticated co-pilots that augment human capability. By acting as collaborators, rather than simple replacements, digital agents could help workers focus on the creative and strategic aspects of their roles while taking over repetitive tasks. I believe this evolution is akin to shifts in computing history. Just as the move from terminals to graphical user interfaces unlocked new ways of interacting with machines, the emergence of digital agents may fundamentally change how we engage with computing. In the future, tasks like tax filing, travel planning, software troubleshooting, HR payroll could be seamlessly handled by these agents, freeing up time and energy for more meaningful work. More ambitiously, digital agents could become true computer co-pilots for scientific exploration, offering automated support to researchers in fields as diverse as biology, computer science, and mechanical engineering. These developments promise not just to streamline routine work but to unlock entirely new forms of creativity, helping us dream bigger and build more sophisticated solutions in a rapidly evolving digital world.

Bibliography

Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, et al. Do as i can, not as i say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691*, 2022.

Kabir Ahuja, Vidhisha Balachandran, Madhur Panwar, Tianxing He, Noah A. Smith, Navin Goyal, and Yulia Tsvetkov. Learning syntax without planting trees: Understanding hierarchical generalization in transformers. *Transactions of the Association for Computational Linguistics*, 13:121–141, 2025. doi: 10.1162/tacl_a_00733. URL https://aclanthology.org/2025.tacl-1.6/.

James Allen, Nathanael Chambers, George Ferguson, Lucian Galescu, Hyuckchul Jung, Mary Swift, and William Taysom. Plow: a collaborative task learning agent. In *Proceedings of the 22nd National Conference on Artificial Intelligence - Volume 2*, AAAI'07, page 1514–1519. AAAI Press, 2007. ISBN 9781577353232.

Jacob Andreas. Measuring compositionality in representation learning. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=HJz05o0qK7.

Jacob Andreas and Dan Klein. Alignment-based compositional semantics for instruction following. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1165–1174, Lisbon, Portugal, September 2015. Association for Computational Linguistics. doi: 10.18653/v1/D15-1138. URL https://aclanthology.org/D15-1138/.

Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. Neural module networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 39–48, 2016.

Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob

McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. *Advances in neural information processing systems*, 30, 2017.

Rohan Anil, Andrew M Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, et al. Palm 2 technical report. *arXiv preprint arXiv:2305.10403*, 2023.

Stanislaw Antol, Aishwarya Agrawal, Jiasen Lu, Margaret Mitchell, Dhruv Batra, C. Lawrence Zitnick, and Devi Parikh. Vqa: Visual question answering. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 2425–2433, 2015. doi: 10.1109/ICCV.2015.279.

Lisa P Argyle, Ethan C Busby, Nancy Fulda, Joshua R Gubler, Christopher Rytting, and David Wingate. Out of one, many: Using language models to simulate human samples. *Political Analysis*, 31(3):337–351, 2023.

Yoav Artzi and Luke Zettlemoyer. Weakly supervised learning of semantic parsers for mapping instructions to actions. *Transactions of the Association for Computational Linguistics*, 1:49–62, 2013. doi: 10.1162/tacl_a_00209. URL https://aclanthology.org/Q13-1005.

Dzmitry Bahdanau, Shikhar Murty, Michael Noukhovitch, Thien Huu Nguyen, Harm de Vries, and Aaron Courville. Systematic generalization: What is required and can it be learned? In *International Conference on Learning Representations (ICLR)*, 2019.

Emily M. Bender and Alexander Koller. Climbing towards NLU: On meaning, form, and understanding in the age of data. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 5185–5198, Online, July 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.463. URL https://aclanthology.org/2020.acl-main.463/.

Satwik Bhattamishra, Kabir Ahuja, and Navin Goyal. On the Ability and Limitations of Transformers to Recognize Formal Languages. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 7096–7116, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.576. URL https://aclanthology.org/2020.emnlp-main.576.

Yonatan Bisk, Ari Holtzman, Jesse Thomason, Jacob Andreas, Yoshua Bengio, Joyce Chai, Mirella Lapata, Angeliki Lazaridou, Jonathan May, Aleksandr Nisnevich, et al. Experience grounds language. *arXiv preprint arXiv:2004.10151*, 2020.

E. Black, S. Abney, D. Flickenger, C. Gdaniec, R. Grishman, P. Harrison, D. Hindle, R. Ingria, F. Jelinek, J. Klavans, M. Liberman, M. Marcus, S. Roukos, B. Santorini, and T. Strzalkowski. A procedure for quantitatively comparing the syntactic coverage of English grammars. In *Speech and Natural Language: Proceedings of a Workshop Held at Pacific Grove, California, February 19-22, 1991*, 1991. URL https://aclanthology.org/H91-1060.

Gilles Blanchard, Gyemin Lee, and Clayton Scott. Generalizing from several related classification tasks to a new unlabeled sample. In *Advances in neural information processing systems*, pages 2178–2186, 2011.

Samuel Bowman, Jon Gauthier, Abhinav Rastogi, Raghav Gupta, Christopher D Manning, and Christopher Potts. A fast unified model for parsing and sentence understanding. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1466–1477, 2016.

S.R.K. Branavan, Harr Chen, Luke Zettlemoyer, and Regina Barzilay. Reinforcement learning for mapping instructions to actions. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, pages 82–90, Suntec, Singapore, August 2009. Association for Computational Linguistics. URL https://aclanthology.org/P09-1010.

S.R.K. Branavan, Luke Zettlemoyer, and Regina Barzilay. Reading between the lines: Learning to map high-level instructions to commands. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, pages 1268–1277, Uppsala, Sweden, July 2010. Association for Computational Linguistics. URL https://aclanthology.org/P10-1129/.

Peter F. Brown, Vincent J. Della Pietra, Peter V. deSouza, Jennifer C. Lai, and Robert L. Mercer. Class-based n-gram models of natural language. In *Computational Linguistics*, volume 18, pages 467–479. MIT Press, 1992.

Peter F. Brown, Stephen A. Della Pietra, Vincent J. Della Pietra, and Robert L. Mercer. The mathematics of statistical machine translation: Parameter estimation. *Computational Linguistics*, 19(2):263–311, 1993.

Hyungjoo Chae, Namyoung Kim, Kai Tzu-iunn Ong, Minju Gwak, Gwanwoo Song, Jihoon Kim, Sunghwan Kim, Dongha Lee, and Jinyoung Yeo. Web agents with world models: Learning and leveraging environment dynamics in web navigation. *arXiv preprint arXiv:2410.13232*, 2024.

Devendra Singh Chaplot, Kanthashree Mysore Sathyendra, Rama Kumar Pasumarthi, Dheeraj Rajagopal, and Ruslan Salakhutdinov. Gated-attention architectures for task-oriented language grounding. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

Eugene Charniak. Statistical parsing with a context-free grammar and word statistics. *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, pages 598–603, 1997.

Eugene Charniak, Don Blaheta, Niyu Ge, Keith Hall, John Hale, and Mark Johnson. Bllip 1987-89 wsj corpus release 1. *Linguistic Data Consortium, Philadelphia*, 36, 2000.

Ciprian Chelba. A structured language model. In *35th Annual Meeting of the Association for Computational Linguistics and 8th Conference of the European Chapter of the Association for Computational Linguistics*, pages 498–500, 1997.

David Chen and Raymond Mooney. Learning to interpret natural language navigation instructions from observations. *Proceedings of the AAAI Conference on Artificial Intelligence*, 25(1):859–865, Aug. 2011. doi: 10.1609/aaai.v25i1.7974. URL https://ojs.aaai.org/index.php/AAAI/article/view/7974.

Kanzhi Cheng, Qiushi Sun, Yougang Chu, Fangzhi Xu, Yantao Li, Jianbing Zhang, and Zhiyong Wu. Seeclick: Harnessing gui grounding for advanced visual gui agents. *arXiv preprint arXiv:2401.10935*, 2024.

David Chiang and Peter Cholak. Overcoming a theoretical limitation of self-attention. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7654–7664, Dublin, Ireland, May 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.acl-long.527. URL https://aclanthology.org/2022.acl-long.527.

Do Kook Choe and Eugene Charniak. Parsing as language modeling. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2331–2336, Austin, Texas, November 2016. Association for Computational Linguistics. doi: 10.18653/v1/D16-1257. URL https://aclanthology.org/D16-1257.

Noam Chomsky. *Syntactic Structures*. Mouton and Co., The Hague, 1957.

Noam Chomsky. *Aspects of the Theory of Syntax*. The MIT Press, Cambridge, 1965.

Kenneth Ward Church. A stochastic parts program and noun phrase parser for unrestricted text. In *Second Conference on Applied Natural Language Processing*, pages 136–143, Austin, Texas, USA, February 1988. Association for Computational Linguistics. doi: 10.3115/974235.974260. URL https://aclanthology.org/A88-1019/.

Geoffrey Cideron, Mathieu Seurin, Florian Strub, and Olivier Pietquin. Higher: Improving instruction following with hindsight generation for experience replay. In *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 225–232. IEEE, 2020.

Kevin Clark, Urvashi Khandelwal, Omer Levy, and Christopher D. Manning. What does BERT look at? an analysis of BERT's attention. In *Proceedings of the 2019 ACL Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 276–286, Florence, Italy, August 2019. Association for Computational Linguistics. doi: 10.18653/v1/W19-4828. URL https://aclanthology.org/W19-4828.

Michael Collins. Three generative, lexicalised models for statistical parsing. In *35th Annual Meeting of the Association for Computational Linguistics and 8th Conference of the European Chapter of the Association for Computational Linguistics*, pages 16–23, 1997.

Stephen Crain and Mineharu Nakayama. Structure dependence in grammar formation. *Language*, 63 (3):522–543, 1987.

Róbert Csordás, Kazuki Irie, and Juergen Schmidhuber. The devil is in the detail: Simple tricks improve systematic generalization of transformers. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 619–634, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.49. URL https://aclanthology.org/2021.emnlp-main.49.

Sreerupa Das, C Lee Giles, and Guo-Zheng Sun. Learning context-free grammars: Capabilities and limitations of a recurrent neural network with an external stack memory. In *Proceedings of The Fourteenth Annual Conference of Cognitive Science Society. Indiana University*, volume 14, 1992.

Gregoire Deletang, Anian Ruoss, Jordi Grau-Moya, Tim Genewein, Li Kevin Wenliang, Elliot Catt, Chris Cundy, Marcus Hutter, Shane Legg, Joel Veness, and Pedro A Ortega. Neural networks and the chomsky hierarchy. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WbxHAzkeQcn.

Ameet Deshpande and Karthik Narasimhan. Guiding attention for self-supervised learning with transformers. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 4676–4686, 2020.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Association for Computational Linguistics (ACL)*, pages 4171–4186, 2019.

Li Dong and Mirella Lapata. Language to logical form with neural attention. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 33–43, Berlin, Germany, August 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-1004. URL https://aclanthology.org/P16-1004/.

Alexandre Drouin, Maxime Gasse, Massimo Caccia, Issam H. Laradji, Manuel Del Verme, Tom Marty, David Vazquez, Nicolas Chapados, and Alexandre Lacoste. WorkArena: How capable are web agents at solving common knowledge work tasks? In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 11642–11662. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/drouin24a.html.

Andrew Drozdov, Patrick Verga, Mohit Yadav, Mohit Iyyer, and Andrew McCallum. Unsupervised latent tree induction with deep inside-outside recursive auto-encoders. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 1129–1141, 2019.

Yuqing Du, Olivia Watkins, Zihan Wang, Cédric Colas, Trevor Darrell, Pieter Abbeel, Abhishek Gupta, and Jacob Andreas. Guiding pretraining in reinforcement learning with large language models. *arXiv preprint arXiv:2302.06692*, 2023.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Brian DuSell and David Chiang. Learning hierarchical structures with differentiable nondeterministic stacks. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=5LXw_QplBiF.

Chris Dyer, Adhiguna Kuncoro, Miguel Ballesteros, and Noah A Smith. Recurrent neural network grammars. In *North American Association for Computational Linguistics (NAACL)*, 2016.

Catherine Finegan-Dollak, Jonathan K. Kummerfeld, Li Zhang, Karthik Ramanathan, Sesh Sadasivam, Rui Zhang, and Dragomir Radev. Improving text-to-SQL evaluation methodology. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 351–360, Melbourne, Australia, July 2018. Association for Computational Linguistics. doi: 10.18653/v1/P18-1033. URL https://aclanthology.org/P18-1033.

Hiroki Furuta, Ofir Nachum, Kuang-Huei Lee, Yutaka Matsuo, Shixiang Shane Gu, and Izzeddin Gur. Multimodal web navigation with instruction-finetuned foundation models. *arXiv preprint arXiv:2305.11854*, 2023.

Yaroslav Ganin, Evgeniya Ustinova, Hana Ajakan, Pascal Germain, Hugo Larochelle, Francois Laviolette, Mario March, and Victor Lempitsky. Domain-adversarial training of neural networks. *Journal of Machine Learning Research (JMLR)*, 17, 2016.

Joshua T. Goodman. A bit of progress in language modeling. In *Proceedings of the 2001 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, volume 1, pages 401–404. IEEE, 2001.

Edward Grefenstette, Karl Moritz Hermann, Mustafa Suleyman, and Phil Blunsom. Learning to transduce with unbounded memory. *Advances in neural information processing systems*, 28, 2015.

Caglar Gulcehre, Tom Le Paine, Srivatsan Srinivasan, Ksenia Konyushkova, Lotte Weerts, Abhishek Sharma, Aditya Siddhant, Alex Ahern, Miaosen Wang, Chenjie Gu, et al. Reinforced self-training (rest) for language modeling. *arXiv preprint arXiv:2308.08998*, 2023.

Izzeddin Gur, Ofir Nachum, Yingjie Miao, Mustafa Safdari, Austin Huang, Aakanksha Chowdhery, Sharan Narang, Noah Fiedel, and Aleksandra Faust. Understanding HTML with large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 2803–2821, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-emnlp.185. URL https://aclanthology.org/2023.findings-emnlp.185.

Michael Hahn. Theoretical Limitations of Self-Attention in Neural Sequence Models. *Transactions of the Association for Computational Linguistics*, 8:156–171, 01 2020. ISSN 2307-387X. doi: 10.1162/tacl_a_00306. URL https://doi.org/10.1162/tacl_a_00306.

John Hale, Chris Dyer, Adhiguna Kuncoro, and Jonathan Brennan. Finding syntax in human encephalography with beam search. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2727–2736, Melbourne, Australia, July 2018. Association for Computational Linguistics. doi: 10.18653/v1/P18-1254. URL https://aclanthology.org/P18-1254.

Brent Harrison, Upol Ehsan, and Mark O Riedl. Guiding reinforcement learning exploration using natural language. *arXiv preprint arXiv:1707.08616*, 2017.

Marc D. Hauser, Noam Chomsky, and W. Tecumseh Fitch. The faculty of language: What is it, who has it, and how did it evolve? *Science*, 298(5598):1569–1579, 2002. doi: 10.1126/science.298.5598.1569. URL https://www.science.org/doi/abs/10.1126/science.298.5598.1569.

Hongliang He, Wenlin Yao, Kaixin Ma, Wenhao Yu, Yong Dai, Hongming Zhang, Zhenzhong Lan, and Dong Yu. Webvoyager: Building an end-to-end web agent with large multimodal models. *arXiv preprint arXiv:2401.13919*, 2024.

John Hewitt and Christopher D. Manning. A structural probe for finding syntax in word representations. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4129–4138, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1419. URL https://aclanthology.org/N19-1419.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. In *Proceedings of the 36th Conference on Neural Information Processing Systems (NeurIPS 2022)*, 2022.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. *arXiv preprint arXiv:1904.09751*, 2019.

Jennifer Hu, Jon Gauthier, Peng Qian, Ethan Wilcox, and Roger Levy. A systematic assessment of syntactic generalization in neural language models. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 1725–1744, Online, July 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.158. URL https://aclanthology.org/2020.acl-main.158.

Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *International Conference on Machine Learning*, pages 9118–9147. PMLR, 2022.

Drew A Hudson and Christopher D Manning. Gqa: A new dataset for real-world visual reasoning and compositional question answering. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 6700–6709, 2019.

Peter C Humphreys, David Raposo, Tobias Pohlen, Gregory Thornton, Rachita Chhaparia, Alistair Muldal, Josh Abramson, Petko Georgiev, Adam Santoro, and Timothy Lillicrap. A data-driven approach for learning to control computers. In *International Conference on Machine Learning*, pages 9466–9482. PMLR, 2022.

Dieuwke Hupkes, Verna Dankers, Mathijs Mul, and Elia Bruni. The compositionality of neural networks: integrating symbolism and connectionism. *CoRR*, abs/1908.08351, 2019. URL http://arxiv.org/abs/1908.08351.

Frederick Jelinek. Self-organized language modeling for speech recognition. In *Readings in speech recognition*, pages 450–506. Morgan Kaufmann, 1990.

Frederick Jelinek, John D. Lafferty, and Robert L. Mercer. Basic methods of probabilistic context free grammars. In *Speech Recognition and Understanding. Recent Advances*, volume F 75 of *NATO ASI Series*, Berlin, 1992. Springer Verlag. URL https://api.semanticscholar.org/CorpusID:62304080.

Robin Jia and Percy Liang. Data recombination for neural semantic parsing. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12–22, Berlin, Germany, August 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-1002. URL https://aclanthology.org/P16-1002/.

Yiding Jiang, Shixiang Shane Gu, Kevin P Murphy, and Chelsea Finn. Language as an abstraction for hierarchical deep reinforcement learning. *Advances in Neural Information Processing Systems*, 32, 2019.

Justin Johnson, Bharath Hariharan, Laurens Van Der Maaten, Li Fei-Fei, C Lawrence Zitnick, and Ross Girshick. Clevr: A diagnostic dataset for compositional language and elementary visual reasoning. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2901–2910, 2017.

Mandar Joshi, Eunsol Choi, Daniel Weld, and Luke Zettlemoyer. TriviaQA: A large scale distantly supervised challenge dataset for reading comprehension. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1601–1611, Vancouver, Canada, July 2017. Association for Computational Linguistics. doi: 10.18653/v1/P17-1147. URL https://aclanthology.org/P17-1147.

Armand Joulin and Tomas Mikolov. Inferring algorithmic patterns with stack-augmented recurrent nets. *Advances in neural information processing systems*, 28, 2015.

Geunwoo Kim, Pierre Baldi, and Stephen McAleer. Language models can solve computer tasks. *arXiv preprint arXiv:2303.17491*, 2023.

Najoung Kim and Tal Linzen. COGS: A compositional generalization challenge based on semantic interpretation. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 9087–9105, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.731. URL https://aclanthology.org/2020.emnlp-main.731.

Yoon Kim, Alexander M Rush, Lei Yu, Adhiguna Kuncoro, Chris Dyer, and Gábor Melis. Unsupervised recurrent neural network grammars. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 1105–1117, 2019.

Nikita Kitaev, Steven Cao, and Dan Klein. Multilingual constituency parsing with self-attention and pre-training. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 3499–3505, 2019.

Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. Neural random-access machines. *arXiv preprint arXiv:1511.06392*, 2015.

Minae Kwon, Sang Michael Xie, Kalesha Bullard, and Dorsa Sadigh. Reward design with language models. In *The Eleventh International Conference on Learning Representations*, 2023a. URL https://openreview.net/forum?id=10uNUgI5Kl.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023b.

Hanyu Lai, Xiao Liu, Iat Long Iong, Shuntian Yao, Yuxuan Chen, Pengbo Shen, Hao Yu, Hanchen Zhang, Xiaohan Zhang, Yuxiao Dong, and Jie Tang. Autowebglm: Bootstrap and reinforce a large language model-based web navigating agent, 2024.

Brenden Lake and Marco Baroni. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks. In *International Conference on Machine Learning (ICML)*, 2018.

Yair Lakretz, Théo Desbordes, Dieuwke Hupkes, and Stanislas Dehaene. Can transformers process recursive nested constructions, like humans? In *Proceedings of the 29th International Conference on Computational Linguistics*, pages 3226–3232, Gyeongju, Republic of Korea, October 2022. International Committee on Computational Linguistics. URL https://aclanthology.org/2022.coling-1.285.

Kenton Lee, Mike Lewis, and Luke Zettlemoyer. Global neural CCG parsing with optimality guarantees. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2366–2376, Austin, Texas, November 2016. Association for Computational Linguistics. doi: 10.18653/v1/D16-1262. URL https://aclanthology.org/D16-1262/.

Kenton Lee, Ming-Wei Chang, and Kristina Toutanova. Latent retrieval for weakly supervised open domain question answering. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 6086–6096, 2019.

Kaixin Li, Ziyang Meng, Hongzhan Lin, Ziyang Luo, Yuchen Tian, Jing Ma, Zhiyong Huang, and Tat-Seng Chua. Screenspot-pro: Gui grounding for professional high-resolution computer use. *arXiv preprint arXiv:2504.07981*, 2025.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 9493–9500. IEEE, 2023.

Percy Liang, Michael I Jordan, and Dan Klein. Learning dependency-based compositional semantics. *Computational Linguistics*, 39(2):389–446, 2013.

Tal Linzen, Emmanuel Dupoux, and Yoav Goldberg. Assessing the ability of LSTMs to learn syntax-sensitive dependencies. *Transactions of the Association for Computational Linguistics*, 4: 521–535, 2016.

Bingbin Liu, Jordan T. Ash, Surbhi Goel, Akshay Krishnamurthy, and Cyril Zhang. Transformers learn shortcuts to automata. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=De4FYqjFueZ.

Evan Zheran Liu, Kelvin Guu, Panupong Pasupat, Tianlin Shi, and Percy Liang. Reinforcement learning on web interfaces using workflow-guided exploration. In *International Conference on Learning Representations*, 2018.

Ziming Liu, Ouail Kitouni, Niklas S Nolte, Eric Michaud, Max Tegmark, and Mike Williams. Towards understanding grokking: An effective theory of representation learning. *Advances in Neural Information Processing Systems*, 35:34651–34663, 2022a.

Ziming Liu, Eric J Michaud, and Max Tegmark. Omnigrok: Grokking beyond algorithmic data. In *The Eleventh International Conference on Learning Representations*, 2022b. URL https://openreview.net/pdf?id=zDiHoIWa0q1.

Lajanugen Logeswaran, Yao Fu, Moontae Lee, and Honglak Lee. Few-shot subgoal planning with language models. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 5493–5506, 2022.

Xing Han Lù, Zdeněk Kasner, and Siva Reddy. Weblinx: Real-world website navigation with multi-turn dialogue. *arXiv preprint arXiv:2402.05930*, 2024.

Arjun Majumdar, Ayush Shrivastava, Stefan Lee, Peter Anderson, Devi Parikh, and Dhruv Batra. Improving vision-and-language navigation with image-text pairs from the web. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part VI 16*, pages 259–274. Springer, 2020.

Christopher D. Manning, Kevin Clark, John Hewitt, Urvashi Khandelwal, and Omer Levy. Emergent linguistic structure in artificial neural networks trained by self-supervision. *Proceedings of the National Academy of Sciences*, 117:30046 – 30054, 2020.

Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330, 1993. URL https://aclanthology.org/J93-2004/.

Rebecca Marvin and Tal Linzen. Targeted syntactic evaluation of language models. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1192–1202, Brussels, Belgium, October-November 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1151. URL https://aclanthology.org/D18-1151.

R. Thomas McCoy, Tal Linzen, Ewan Dunbar, and Paul Smolensky. Rnns implicitly implement tensor product representations. In *International Conference on Learning Representations (ICLR)*, 2019.

R. Thomas McCoy, Robert Frank, and Tal Linzen. Does syntax need to grow on trees? Sources of hierarchical inductive bias in sequence-to-sequence networks. *Transactions of the Association for Computational Linguistics*, 2020.

Hongyuan Mei, Mohit Bansal, and Matthew Walter. Listen, attend, and walk: Neural mapping of navigational instructions to action sequences. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=Byj72udxe.

William Merrill, Vivek Ramanujan, Yoav Goldberg, Roy Schwartz, and Noah A. Smith. Effects of parameter norm growth during transformer training: Inductive bias from gradient descent. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 1766–1781, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.133. URL https://aclanthology.org/2021.emnlp-main.133.

William Merrill, Ashish Sabharwal, and Noah A. Smith. Saturated transformers are constant-depth threshold circuits. *Transactions of the Association for Computational Linguistics*, 10:843–856, 2022. doi: 10.1162/tacl_a_00493. URL https://aclanthology.org/2022.tacl-1.49.

Sewon Min, Xinxi Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. Rethinking the role of demonstrations: What makes in-context learning work? *arXiv preprint arXiv:2202.12837*, 2022.

Piotr Mirowski and Andreas Vlachos. Dependency recurrent neural language models for sentence completion. In *Proceedings of the 53rd Annual Meeting of the Association for Computational*

Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers), pages 511–517, 2015.

Dipendra Misra, John Langford, and Yoav Artzi. Mapping instructions and visual observations to actions with reinforcement learning. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1004–1015, Copenhagen, Denmark, September 2017. Association for Computational Linguistics. doi: 10.18653/v1/D17-1106. URL https://aclanthology.org/D17-1106/.

Dipendra Kumar Misra and Yoav Artzi. Neural shift-reduce CCG semantic parsing. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1775–1786, Austin, Texas, November 2016. Association for Computational Linguistics. doi: 10.18653/v1/D16-1183. URL https://aclanthology.org/D16-1183/.

Richard Montague. Universal grammar. *Theoria*, 36(3):373–398, 1970.

Jesse Mu, Victor Zhong, Roberta Raileanu, Minqi Jiang, Noah Goodman, Tim Rocktäschel, and Edward Grefenstette. Improving intrinsic exploration with language abstractions. *Advances in Neural Information Processing Systems*, 35:33947–33960, 2022.

Krikamol Muandet, David Balduzzi, and Bernhard Schölkopf. Domain generalization via invariant feature representation. In *ICML*, pages 10–18, 2013. URL http://proceedings.mlr.press/v28/muandet13.html.

Aaron Mueller, Robert Frank, Tal Linzen, Luheng Wang, and Sebastian Schuster. Coloring the blank slate: Pre-training imparts a hierarchical inductive bias to sequence-to-sequence models. In *Findings of the Association for Computational Linguistics: ACL 2022*, pages 1352–1368, Dublin, Ireland, May 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.findings-acl.106. URL https://aclanthology.org/2022.findings-acl.106.

Shikhar Murty, Pratyusha Sharma, Jacob Andreas, and Christopher Manning. Grokking of hierarchical structure in vanilla transformers. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 439–448, Toronto, Canada, July 2023a. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-short.38. URL https://aclanthology.org/2023.acl-short.38/.

Shikhar Murty, Pratyusha Sharma, Jacob Andreas, and Christopher Manning. Pushdown layers: Encoding recursive structure in transformer language models. In *Proceedings of the 2023 Conference*

on Empirical Methods in Natural Language Processing, pages 3233–3247, Singapore, December 2023b. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.195. URL https://aclanthology.org/2023.emnlp-main.195/.

Shikhar Murty, Pratyusha Sharma, Jacob Andreas, and Christopher D Manning. Characterizing intrinsic compositionality in transformers with tree projections. In *The Eleventh International Conference on Learning Representations*, 2023c. URL https://openreview.net/forum?id=sAOOeI878Ns.

Shikhar Murty, Christopher D Manning, Peter Shaw, Mandar Joshi, and Kenton Lee. BAGEL: Bootstrapping agents by guiding exploration with language. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 36894–36910. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/murty24a.html.

Shikhar Murty, Hao Zhu, Dzmitry Bahdanau, and Christopher D Manning. Nnetnav: Unsupervised learning of browser agents through environment interaction in the wild, 2025. *URL https://arxiv.org/abs/2410.02907*, 2025.

Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, et al. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*, 2021.

Ananjan Nandi, Christopher D Manning, and Shikhar Murty. Sneaking syntax into transformer language models with tree regularization. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 8006–8024, Albuquerque, New Mexico, April 2025. Association for Computational Linguistics. ISBN 979-8-89176-189-6. URL https://aclanthology.org/2025.naacl-long.407/.

Tianyue Ou, Frank F Xu, Aman Madaan, Jiarui Liu, Robert Lo, Abishek Sridhar, Sudipta Sengupta, Dan Roth, Graham Neubig, and Shuyan Zhou. Synatra: Turning indirect knowledge into direct demonstrations for digital agents at scale. *arXiv preprint arXiv:2409.15637*, 2024.

Christophe Pallier, Anne-Dominique Devauchelle, and Stanislas Dehaene. Cortical representation of the constituent structure of sentences. *Proceedings of the National Academy of Sciences*, 108(6):

2522–2527, 2011. doi: 10.1073/pnas.1018711108. URL https://www.pnas.org/doi/abs/10.1073/pnas.1018711108.

Aaron Parisi, Yao Zhao, and Noah Fiedel. Talm: Tool augmented language models. *arXiv preprint arXiv:2205.12255*, 2022.

Ajay Patel, Markus Hofmarcher, Claudiu Leoveanu-Condrei, Marius-Constantin Dinu, Chris Callison-Burch, and Sepp Hochreiter. Large language models can self-improve at web agent tasks. *arXiv preprint arXiv:2405.20309*, 2024.

Arkil Patel, Satwik Bhattamishra, Phil Blunsom, and Navin Goyal. Revisiting the compositional generalization abilities of neural sequence models. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 424–434, Dublin, Ireland, May 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.acl-short.46. URL https://aclanthology.org/2022.acl-short.46.

Hao Peng, Roy Schwartz, and Noah A. Smith. PaLM: A hybrid parser and language model. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3644–3651, Hong Kong, China, November 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-1376. URL https://aclanthology.org/D19-1376.

Jorge Perez, Pablo Barcelo, and Javier Marinkovic. Attention is turing-complete. *Journal of Machine Learning Research*, 22(75):1–35, 2021. URL http://jmlr.org/papers/v22/20-302.html.

Jackson Petty and Robert Frank. Transformers generalize linearly. *arXiv preprint arXiv:2109.12036*, 2021.

Alethea Power, Yuri Burda, Harri Edwards, Igor Babuschkin, and Vedant Misra. Grokking: Generalization beyond overfitting on small algorithmic datasets. *arXiv preprint arXiv:2201.02177*, 2022.

Peng Qian, Tahira Naseem, Roger Levy, and Ramón Fernandez Astudillo. Structural guidance for transformer language models. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 3735–3745, Online, August

2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.289. URL https://aclanthology.org/2021.acl-long.289.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. 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)*, pages 1139–1149, Vancouver, Canada, July 2017. Association for Computational Linguistics. doi: 10.18653/v1/P17-1105. URL https://aclanthology.org/P17-1105/.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners, 2019. URL https://d4mucfpksywv.cloudfront.net/better-language-models/language_models_are_unsupervised_multitask_learners.pdf.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21:1–67, 2020.

Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. SQuAD: 100,000+ questions for machine comprehension of text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2383–2392, Austin, Texas, November 2016. Association for Computational Linguistics. doi: 10.18653/v1/D16-1264. URL https://aclanthology.org/D16-1264.

Pranav Rajpurkar, Robin Jia, and Percy Liang. Know what you don't know: Unanswerable questions for SQuAD. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 784–789, Melbourne, Australia, July 2018. Association for Computational Linguistics. doi: 10.18653/v1/P18-2124. URL https://aclanthology.org/P18-2124.

Marco Tulio Ribeiro, Tongshuang Wu, Carlos Guestrin, and Sameer Singh. Beyond accuracy: Behavioral testing of NLP models with CheckList. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4902–4912, Online, July 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.442. URL https://aclanthology.org/2020.acl-main.442.

Laura Ruis, Jacob Andreas, Marco Baroni, Diane Bouchacourt, and Brenden M Lake. A benchmark for systematic generalization in grounded language understanding. *Advances in neural information processing systems*, 33:19861–19872, 2020.

Laurent Sartran, Samuel Barrett, Adhiguna Kuncoro, Miloš Stanojević, Phil Blunsom, and Chris Dyer. Transformer Grammars: Augmenting Transformer Language Models with Syntactic Inductive Biases at Scale. *Transactions of the Association for Computational Linguistics*, 10:1423–1439, 12 2022. ISSN 2307-387X. doi: 10.1162/tacl_a_00526. URL https://doi.org/10.1162/tacl_a_00526.

Murray Shanahan, Kyle McDonell, and Laria Reynolds. Role play with large language models. *Nature*, 623(7987):493–498, 2023.

Peter Shaw, Mandar Joshi, James Cohan, Jonathan Berant, Panupong Pasupat, Hexiang Hu, Urvashi Khandelwal, Kenton Lee, and Kristina Toutanova. From pixels to ui actions: Learning to follow instructions via graphical user interfaces. *arXiv preprint arXiv:2306.00245*, 2023.

Junhong Shen, Atishay Jain, Zedian Xiao, Ishan Amlekar, Mouad Hadji, Aaron Podolny, and Ameet Talwalkar. Scribeagent: Towards specialized web agents using production-scale workflow data. *arXiv preprint arXiv:2411.15004*, 2024.

Yikang Shen, Shawn Tan, Alessandro Sordoni, and Aaron Courville. Ordered neurons: Integrating tree structures into recurrent neural networks. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=B1l6qiR5F7.

Tianlin Shi, Andrej Karpathy, Linxi Fan, Jonathan Hernandez, and Percy Liang. World of bits: An open-domain platform for web-based agents. In *International Conference on Machine Learning*, pages 3135–3144. PMLR, 2017.

Noah Shinn, Beck Labash, and Ashwin Gopinath. Reflexion: an autonomous agent with dynamic memory and self-reflection. *arXiv preprint arXiv:2303.11366*, 2023.

Avi Singh, John D Co-Reyes, Rishabh Agarwal, Ankesh Anand, Piyush Patil, Peter J Liu, James Harrison, Jaehoon Lee, Kelvin Xu, Aaron Parisi, et al. Beyond human data: Scaling self-training for problem-solving with language models. *arXiv preprint arXiv:2312.06585*, 2023.

Paul Smolensky. Tensor product variable binding and the representation of symbolic structures in connectionist systems. *Artificial Intelligence*, 46(1):159–216, 1990. ISSN 0004-3702. doi:

https://doi.org/10.1016/0004-3702(90)90007-M. URL https://www.sciencedirect.com/science/article/pii/000437029090007M.

Richard Socher, Alex Perelygin, Jean Y Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2013.

Paloma Sodhi, SRK Branavan, and Ryan McDonald. Heap: Hierarchical policies for web actions using llms. *arXiv preprint arXiv:2310.03720*, 2023.

Daniel Soudry, Elad Hoffer, Mor Shpigel Nacson, Suriya Gunasekar, and Nathan Srebro. The implicit bias of gradient descent on separable data. *The Journal of Machine Learning Research*, 19(1): 2822–2878, 2018.

Mark Steedman. *Surface Structure and Interpretation*. MIT Press, 1996.

Mitchell Stern, Jacob Andreas, and Dan Klein. A minimal span-based neural constituency parser. In *Association for Computational Linguistics (ACL)*, pages 818–827, 2017a.

Mitchell Stern, Daniel Fried, and Dan Klein. Effective inference for generative neural parsing. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1695–1700, Copenhagen, Denmark, September 2017b. Association for Computational Linguistics. doi: 10.18653/v1/D17-1178. URL https://aclanthology.org/D17-1178.

Emma Strubell, Patrick Verga, Daniel Andor, David Weiss, and Andrew McCallum. Linguistically-informed self-attention for semantic role labeling. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 5027–5038, Brussels, Belgium, October-November 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1548. URL https://aclanthology.org/D18-1548.

Theodore Sumers, Kenneth Marino, Arun Ahuja, Rob Fergus, and Ishita Dasgupta. Distilling internet-scale vision-language models into embodied agents. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 32797–32818. PMLR, 23–29 Jul 2023. URL https://proceedings.mlr.press/v202/sumers23a.html.

Haotian Sun, Yuchen Zhuang, Lingkai Kong, Bo Dai, and Chao Zhang. Adaplanner: Adaptive planning from feedback with language models. *arXiv preprint arXiv:2305.16653*, 2023.

Mirac Suzgun, Sebastian Gehrmann, Yonatan Belinkov, and Stuart M Shieber. Memory-augmented recurrent neural networks can learn generalized dyck languages. *arXiv preprint arXiv:1911.03329*, 2019.

Kai Shen Tai, Richard Socher, and Christopher D. Manning. Improved semantic representations from tree-structured long short-term memory networks. In *Association for Computational Linguistics (ACL)*, 2015a.

Kai Sheng Tai, Richard Socher, and Christopher D. Manning. Improved semantic representations from tree-structured long short-term memory networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1556–1566, Beijing, China, July 2015b. Association for Computational Linguistics. doi: 10.3115/v1/P15-1150. URL https://aclanthology.org/P15-1150.

Stefanie Tellex, Thomas Kollar, Steven Dickerson, Matthew Walter, Ashis Banerjee, Seth Teller, and Nicholas Roy. Understanding natural language commands for robotic navigation and mobile manipulation. *Proceedings of the AAAI Conference on Artificial Intelligence*, 25(1):1507–1514, Aug. 2011. doi: 10.1609/aaai.v25i1.7979. URL https://ojs.aaai.org/index.php/AAAI/article/view/7979.

Ke M Tran, Arianna Bisazza, and Christof Monz. The importance of being recurrent for modeling hierarchical structure. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 4731–4736, 2018.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017a.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, 2017b. URL https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

Oriol Vinyals, Łukasz Kaiser, Terry Koo, Slav Petrov, Ilya Sutskever, and Geoffrey Hinton. Grammar as a foreign language. In *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015. URL https://proceedings.neurips.cc/paper_files/paper/2015/file/277281aada22045c03945dcb2ca6f2ec-Paper.pdf.

Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. GLUE: A multi-task benchmark and analysis platform for natural language understanding. In *International Conference on Learning Representations*, 2019a. URL https://openreview.net/forum?id=rJ4km2R5t7.

Boshi Wang, Xiang Yue, Yu Su, and Huan Sun. Grokked transformers are implicit reasoners: A mechanistic journey to the edge of generalization, 2024a. URL https://arxiv.org/pdf/2405.15071.

Yaushian Wang, Hung-Yi Lee, and Yun-Nung Chen. Tree transformer: Integrating tree structures into self-attention. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 1061–1070, Hong Kong, China, November 2019b. Association for Computational Linguistics. doi: 10.18653/v1/D19-1098. URL https://aclanthology.org/D19-1098.

Yizhong Wang, Hamish Ivison, Pradeep Dasigi, Jack Hessel, Tushar Khot, Khyathi Raghavi Chandu, David Wadden, Kelsey MacMillan, Noah A. Smith, Iz Beltagy, and Hannaneh Hajishirzi. How far can camels go? exploring the state of instruction tuning on open resources, 2023.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. *arXiv preprint arXiv:2409.07429*, 2024b.

Alex Warstadt, Alicia Parrish, Haokun Liu, Anhad Mohananey, Wei Peng, Sheng-Fu Wang, and Samuel R. Bowman. BLiMP: The benchmark of linguistic minimal pairs for English. *Transactions of the Association for Computational Linguistics*, 8:377–392, 2020. doi: 10.1162/tacl_a_00321. URL https://aclanthology.org/2020.tacl-1.25.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pages 24824–24837. Curran Associates, Inc., 2022.

Joseph Weizenbaum. Eliza—a computer program for the study of natural language communication between man and machine. *Commun. ACM*, 9(1):36–45, January 1966. ISSN 0001-0782. doi: 10.1145/365153.365168. URL https://doi.org/10.1145/365153.365168.

Terry Winograd. Understanding natural language. *Cognitive Psychology*, 3(1):1–191, 1972. ISSN 0010-0285. doi: https://doi.org/10.1016/0010-0285(72)90002-3. URL https://www.sciencedirect.com/science/article/pii/0010028572900023.

Ludwig Wittgenstein. *Philosophical Investigations*. Wiley-Blackwell, Oxford, 4 edition, 1953. Originally published in 1953.

Yuk Wah Wong and Raymond Mooney. Learning for semantic parsing with statistical machine translation. In *Proceedings of the Human Language Technology Conference of the NAACL, Main Conference*, pages 439–446, New York City, USA, June 2006. Association for Computational Linguistics. URL https://aclanthology.org/N06-1056/.

Zhengxuan Wu, Elisa Kreiss, Desmond C Ong, and Christopher Potts. Reascan: Compositional reasoning in language grounding. *arXiv preprint arXiv:2109.08994*, 2021.

Ted Xiao, Harris Chan, Pierre Sermanet, Ayzaan Wahid, Anthony Brohan, Karol Hausman, Sergey Levine, and Jonathan Tompson. Robotic skill acquisition via instruction augmentation with vision-language models. *arXiv preprint arXiv:2211.11736*, 2022.

Yiheng Xu, Dunjie Lu, Zhennan Shen, Junli Wang, Zekun Wang, Yuchen Mao, Caiming Xiong, and Tao Yu. Agenttrek: Agent trajectory synthesis via guiding replay with web tutorials. *arXiv preprint arXiv:2412.09605*, 2024. URL https://arxiv.org/abs/2412.09605.

An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, et al. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*, 2024.

Shunyu Yao, Rohan Rao, Matthew Hausknecht, and Karthik Narasimhan. Keep CALM and explore: Language models for action generation in text-based games. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 8736–8754, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.704. URL https://aclanthology.org/2020.emnlp-main.704/.

Shunyu Yao, Binghui Peng, Christos Papadimitriou, and Karthik Narasimhan. Self-attention networks can process bounded hierarchical languages. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 3770–3785, Online, August

2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.292. URL https://aclanthology.org/2021.acl-long.292.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2022.

Aditya Yedetore and Najoung Kim. Semantic training signals promote hierarchical syntactic generalization in transformers. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 4059–4073, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.235. URL https://aclanthology.org/2024.emnlp-main.235/.

Lang Yu and Allyson Ettinger. Assessing phrasal representation and composition in transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 4896–4907, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.397. URL https://aclanthology.org/2020.emnlp-main.397.

M. Zelle and R. J. Mooney. Learning to parse database queries using inductive logic programming. In *Association for the Advancement of Artificial Intelligence (AAAI)*, pages 1050–1055, 1996.

Luke S. Zettlemoyer and Michael Collins. Learning to map sentences to logical form: structured classification with probabilistic categorial grammars. In *Proceedings of the Twenty-First Conference on Uncertainty in Artificial Intelligence*, UAI'05, page 658–666, Arlington, Virginia, USA, 2005. AUAI Press. ISBN 0974903914.

Shizhuo Dylan Zhang, Curt Tigges, Stella Biderman, Maxim Raginsky, and Talia Ringer. Can transformers learn to solve problems recursively? *arXiv preprint arXiv:2305.14699*, 2023.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Yonatan Bisk, Daniel Fried, Uri Alon, et al. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*, 2023.

Yifei Zhou, Qianlan Yang, Kaixiang Lin, Min Bai, Xiong Zhou, Yu-Xiong Wang, Sergey Levine, and Erran Li. Proposer-agent-evaluator (pae): Autonomous skill discovery for foundation model internet agents. *arXiv preprint arXiv:2412.13194*, 2024.

Yuchen Zhuang, Yue Yu, Kuan Wang, Haotian Sun, and Chao Zhang. Toolqa: A dataset for llm question answering with external tools. *arXiv preprint arXiv:2306.13304*, 2023.

Appendix A

Formal Results on Tree Projections

Lemma 1. $\mathcal{L}(f, g_{\phi^*}, T) \leq \sum_{S \in \mathcal{D}} \mathrm{SCI}(S, T(S))$

Proof. Let $l(f, g_\phi, S, T) \triangleq \sum_{s \in T(S)} d(g_\phi(s, T(s)), \boldsymbol{v}_s^S)$ for any $S \in \mathcal{D}$, where g is a tree-structured network indexed by $\phi \in \mathbb{R}^p$. The overall error of g_ϕ on $\mathcal{D}$ is

$$\mathcal{L}(f, g_\phi, T) = \sum_{S \in \mathcal{D}} l(f, g_\phi, S, T). \tag{A.1}$$

Let $\phi^* \triangleq \arg\min_\phi \mathcal{L}(f, g_\phi, T)$. Next, consider $\hat{\phi} \in \mathbb{R}^p$ such that $g_{\hat{\phi}}(s, T(s)) = \tilde{\boldsymbol{v}}_s$ for all $s \in \mathcal{D}$. Such a $\hat{\phi}$ always exists for large enough p, since there exists a unique $\tilde{\boldsymbol{v}}_s$ for any p given $\mathcal{D}$ and f. Clearly, $l(f, g_{\hat{\phi}}, S, T) = \sum_{s \in T(S)} d(\boldsymbol{v}_s^S, \tilde{\boldsymbol{v}}_s)$. By definition, we have

$$\begin{aligned} \mathcal{L}(f, g_{\phi^*}, T) &\leq \mathcal{L}(f, g_{\hat{\phi}}, T) && \text{(A.2)} \\ &= \sum_{S \in \mathcal{D}} \sum_{s \in T(S)} d(\boldsymbol{v}_s^S, \tilde{\boldsymbol{v}}_s) = \sum_{S \in \mathcal{D}} \mathrm{SCI}(S, T(S)). && \text{(A.3)} \end{aligned}$$

□

Theorem 1. $\min_{\phi, T} \mathcal{L}(f, g_\phi, T) \leq \sum_{S \in \mathcal{D}} \min_{T(S)} \mathrm{SCI}(S, T(S))$. *In other words, the best tree structured approximation to f has an error upper bounded by cumulative* SCI *scores.*

Proof. We have

$$\min_{\phi, T} \mathcal{L}(f, g_\phi, T) = \min_T \min_\phi \mathcal{L}(f, g_\phi, T) \tag{A.4}$$

For any given T, we have $\min_\phi \mathcal{L}(f, g_\phi, T) \leq \sum_{S\in\mathcal{D}} \text{SCI}(S, T(S))$. Thus minimizing both sides with respect to T, we have

$$\min_T \min_\phi \mathcal{L}(f, g_\phi, T) \leq \min_T \sum_{S\in\mathcal{D}} \text{SCI}(S, T(S)) \tag{A.5}$$

$$= \sum_{S\in\mathcal{D}} \min_{T(S)} \text{SCI}(S, T(S)) \tag{A.6}$$

□

Under Assumption 1 and Theorem 1, we have the proof for Corollary 1.1 which we present next.

Corollary 1.1. *Under Assumption 1,* $\min_{\phi,T} \mathcal{L}(f, g_\phi, T) = \sum_{S\in\mathcal{D}} \min_{T(S)} \text{SCI}(S, T(S))$. *Moreover,* $T_{proj}(S) = \arg\min_{T(S)} \text{SCI}(S, T(S))$ *for any* $S \in \mathcal{D}$.

Proof. Let s_T be the collection of all spans that occur as a constituent for some $T(S)$ where $S \in \mathcal{D}$. We have

$$\mathcal{L}(f, g_\phi, T) = \sum_{S\in\mathcal{D}} \sum_{s\in T(S)} d(g_\phi(s, T(s)), \boldsymbol{v}_s^S) \tag{A.7}$$

$$= \sum_{s\in s_T} \sum_{S\in S_s} d(g_\phi(s, T(s)), \boldsymbol{v}_s^S). \tag{A.8}$$

Now, using Assumption 1, we note that

$$\sum_{S\in S_s} d(g_\phi(s, T(s)), \boldsymbol{v}_s^S) \geq \min_{\boldsymbol{v}} \sum_{S\in S_s} d(\boldsymbol{v}, \boldsymbol{v}_s^S) = \sum_{S\in S_s} d(\tilde{\boldsymbol{v}}_s, \boldsymbol{v}_s^S). \tag{A.9}$$

Combining Equation A.9 and Lemma 1, we have

$$\min_\phi \mathcal{L}(f, g_\phi, T) = \sum_{S\in\mathcal{D}} \text{SCI}(S, T(S)) \tag{A.10}$$

Now, we have

$$T_{\text{proj}} = \arg\min_T \left[\min_\phi \mathcal{L}(f, g_\phi, T)\right] = \arg\min_T \sum_{S\in\mathcal{D}} \text{SCI}(S, T(S)) \tag{A.11}$$

Thus, $T_{\text{proj}}(S) = \arg\min_{T(S)} \text{SCI}(S, T(S))$ □

Next, we consider specific examples of distance metric d, and what Assumption 1 implies for context-free vectors $\tilde{\boldsymbol{v}}_s$.

Example A.0.1. *Suppose d is the euclidean L_2 distance i.e., $d(\boldsymbol{x}, \boldsymbol{y}) = \|\boldsymbol{x} - \boldsymbol{y}\|$. Then, Assumption 1 requires that $\tilde{\boldsymbol{v}}_s = \frac{1}{|Ss|} \sum_{S \in S_s} \boldsymbol{v}_s^S$*

Proof Sketch. We have $\boldsymbol{v}_s^* = \arg\min_v \sum_{S \in S_s} d(\boldsymbol{v}_s^S, \boldsymbol{v}) = \arg\min_v \sum_{S \in S_s} \|\boldsymbol{v} - \boldsymbol{v}_s^S\|$. Setting derivatives with respect to $\boldsymbol{v}$ to 0, we have $\boldsymbol{v}_s^* = \frac{1}{|S_s|} \sum_{S \in S_s} \boldsymbol{v}_s^S$ □

Example A.0.2. *Let d be the cosine distance of $\boldsymbol{x}$ and $\boldsymbol{y}$ i.e., $d(\boldsymbol{x}, \boldsymbol{y}) = 1 - \frac{\boldsymbol{x}^\top \boldsymbol{y}}{\|\boldsymbol{x}\|\|\boldsymbol{y}\|}$. Then, Assumption 1 requires that $\tilde{\boldsymbol{v}}_s = \frac{1}{|S_s|} \sum_{S \in S_s} \frac{\boldsymbol{v}_s^S}{\|\boldsymbol{v}_s^S\|}$*

Proof Sketch. We have

$$\boldsymbol{v}_s^* = \arg\min_{\boldsymbol{v}} \sum_{S \in S_s} d(\boldsymbol{v}_s^S, \boldsymbol{v}) = \arg\max_{\boldsymbol{v}} \sum_{S \in S_s} \frac{\boldsymbol{v}^\top \boldsymbol{v}_s^S}{\|\boldsymbol{v}\|\|\boldsymbol{v}_s^S\|} = \arg\max_{\boldsymbol{v}} \frac{\boldsymbol{v}^\top}{\|\boldsymbol{v}\|} \Big(\sum_{S \in S_s} \frac{\boldsymbol{v}_s^S}{\|\boldsymbol{v}_s^S\|} \Big). \quad \text{(A.12)}$$

Thus, $\boldsymbol{v}_s^* = k \sum_{S \in S_s} \frac{\boldsymbol{v}_s^S}{\|\boldsymbol{v}_s^S\|}$ for any $k > 0$ □

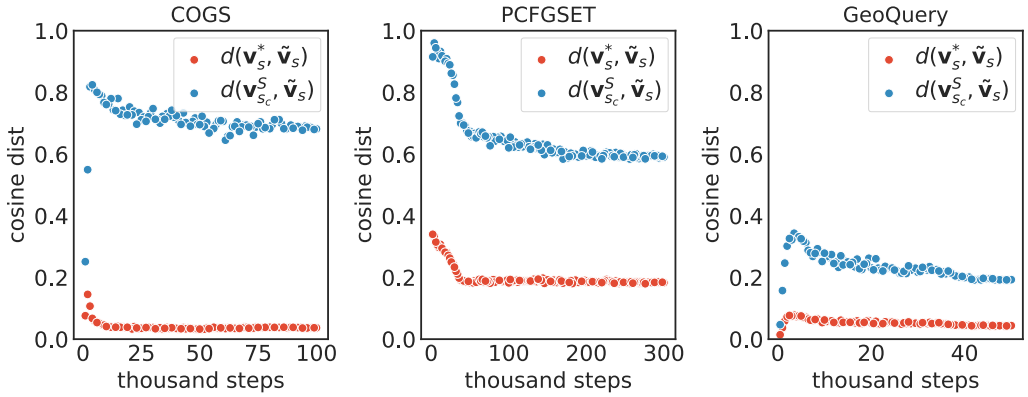


Figure A.1: We plot $d(\boldsymbol{v}_s^*, \tilde{\boldsymbol{v}}_s)$ for randomly sampled spans at various points during training. As a control, we also plot $d(\boldsymbol{v}_{s_c}^S, \tilde{\boldsymbol{v}}_s)$ for a random span s_c. We observe that for COGS and GeoQuery, the distance between the optimal $\boldsymbol{v}_s^*$ and $\tilde{\boldsymbol{v}}_s$ eventually becomes less than 0.05. We conclude that the conditions of Assumption 1 approximately hold true for 2/3 datasets.

Appendix B

BAGEL: Prompts used for various LM components

B.1 Details of the Retriever used in BAGEL

We use a T5-XXL model to embed each word in the instruction, and mean pool across word embeddings to obtain an instruction vector. Given a test-time instruction, to retrieve relevant demonstrations, we compute cosine similarities between the test instruction embedding and instruction embeddings for each demonstration in our buffer, and return the top 3 demonstrations with the highest cosine similarities.

B.2 Re-sampling action strings

When executing an action string in the environment results in an exception from the low-level controller, we pass the exception message to the LM policy, and re-sample till the model outputs a valid action, or the LM exceeds the max number of tries $m = 5$. Here is an example prompt we use for this re-sampling procedure (the prompt is appended to the LM policy).

Listing B.1: Re-sampling during Execution Failure

```
Executing Action: {error_action}...
resulted in error: {error_message}. Think about what could have caused the error, and
then choose a new action.

Thought: [[thought_pred]]
```

```
Now, output a different action based on your thought. End your output with a newline.

Action: [[action]]
```

B.3 Prompts

B.3.1 MiniWoB++

We start by presenting all prompts for MiniWoB++. The action space for MiniWob++ is:

Listing B.2: Action Space

```
- Click on *description*: This action will click on element that matches *description*
e.g. Click on the red button, Click on the first result in the autocomplete
- Move Mouse to *description*: This action will hover mouse over web element that
matches *description* e.g. Move mouse to the menu bar.
- Type char *char* on *description*: This action will type a single character *char*
into the web element matching *description* e.g. Type char B on the first name field.
Use this if you want to type in a word character by character, to view or narrow search
 results.
- Type *text* on *description*: This action will type *text* into the web element
matching *description*. Use this to type in all the words in *text*' all at once.
- Clear text on *description*: This action will clear all previously typed text in web
element matching *description*
```

This is then directly used for various prompts as `{inventory_str}`.

Listing B.3: Exploration Policy

```
You are a web-agent that can interact with the given webpage by taking actions. You can
 take the following kinds of actions:
{inventory_str}

Your objective is to discover diverse and interesting tasks (that a human might give to
 an agent) by interacting with the webpage through these actions. You've executed the
following actions, and observed the following webpage states (described briefly in
language).

**Previous observations and actions**
```

```
{prev_observations_and_actions}

After taking these actions, you observe the current web-page HTML:
{webpage_html}

Start by thinking about what action you should take next.
Thought: [[pred]]

Now, act by taking an action based in the inventory (or output Finish if you are done).
Action: [[pred]]
```

Listing B.4: Instruction Generator

```
**Objective**
You are a web-agent that can accomplish useful tasks on a website. You are given the
landing page of the website as follows:
{init_html}

To accomplish tasks, you can break it down into a sequence of sub-tasks from a task
inventory:
{inventory_str}

Propose a new task that can be performed on this website. Ensure that your tasks are
concrete and use features / contents of the given website.

Start by thinking about what new task you will generate.
Thought: [[pred]]
Answer: [[pred]]
```

Listing B.5: Trajectory Relabeler

```
A web-agent is given a precise instruction from a human, which it carries out through a
 sequence of sub-tasks, where each sub-task (such as clicking on elements / typing on
elements / scrolling etc.) changes the HTML state of the webpage.
You are given the initial webpage (as HTML), the final webpage after all sub-tasks are
carried out, as well as a summary of changes that each sub-task made to the starting
HTML state.

Initial Webpage:
{init_webpage}

Final Webpage:
{final_webpage}
```

```
Sub-tasks attempted by the web agent:
{subgoal_str}

Summary of changes made to HTML:
{observation_changes}

Your objective to guess the instruction that was given to the agent. Ensure that your
instructions are concrete and such that every sub-task meaningfully contributes to
fulfiling the instruction. Start by providing your reasoning. Use the following format
for your answer:
Reasoning: your reasoning
Answer: your answer

**Output**
Reasoning: [[pred]]
Answer: [[pred]]
```

Listing B.6: Instruction Following Policy

```
You are a web-agent on an HTML page capable of executing the following kinds of sub-
tasks:
{inventory_str}

You are also given some examples of how to perform instructions on the website by
converting them into sub-tasks (along with the change each sub-task caused on the
website).
{exemplars}

You are given the following instruction: {instruction}.
To perform this instruction, you've executed the following sub-tasks, and observed the
following webpage states (described briefly in language).

**Previous observations and actions**
{prev_observations_and_actions}

After taking these actions, you observe the current web-page HTML:
{webpage_html}

Webpage Description: [[pred]]

First, think about which inventory item you should pick as your next action.
Thought: [[pred]]
```

```
Now, output next action (output *finished* if the instruction has been accomplished) by
 choosing an item from your inventory
Action: [[pred]]
```

Listing B.7: Demonstration Filter

```
You are given an initial web-page from a website (as HTML). To accomplish some task, a
web-agent then interacts with the website, leading to a final webpage.
Given the task, the initial webpage and the final webpage, your objective is to judge
how well the web-agent carried out this task by giving it a score from 1 to 5.
Only give a score of 5 if the task is perfectly accomplished and the final webpage has
no errors.

Task:
{goal_str}

Initial Webpage:
{init_webpage}

Final Webpage:
{final_webpage}

Start by thinking about what the web-agent was trying to accomplish, and describe how
well it was done.
Thought: [[pred]]
Answer: [[pred]]
```

B.3.2 ToolQA

Next, we present all prompts for ToolQA below. The list of methods for various tools in ToolQA is:

Listing B.8: ToolQA methods

```
(1) Calculate[formula], which calculates the formula and returns the result.
(2) RetrieveAgenda[keyword], which retrieves the agenda related to keyword.
(3) RetrieveScirex[keyword], which retrieves machine learning papers' paragraphs
related to keyword.
(4) LoadDB[DBName], which loads the database DBName and returns the database. The
DBName can be one of the following: flights/coffee/airbnb/yelp.
(5) FilterDB[condition], which filters the database DBName by the column column_name
the relation (e.g., =, >, etc.) and the value value, and returns the filtered database.
```

```
(6) GetValue[column_name], which returns the value of the column column_name in the
database DBName.
(7) LoadGraph[GraphName], which loads the graph GraphName and returns the graph. The
GraphName can be one of the following: PaperNet/AuthorNet.
(8) NeighbourCheck[GraphName, Node], which lists the neighbours of the node Node in the
 graph GraphName and returns the neighbours.
(9) NodeCheck[GraphName, Node], which returns the detailed attribute information of
Node.
(10) EdgeCheck[GraphName, Node1, Node2], which returns the detailed attribute
information of the edge between Node1 and Node2.
(11) SQLInterpreter[SQL], which interprets the SQL query SQL and returns the result.
(12) PythonInterpreter[Python], which interprets the Python code Python.
```

and the action space for the LM policy is:

Listing B.9: Action Space

```
(1) Calculate *formula*, which calculates an arithmetic formula (such as 2+3, 2 * 4 etc
) and returns the result.
(2) Retrieve passages related to *phrase*, which retrieves information relevant to the
supplied phrase. This retriever operates on documents containing information about
people's schedules.
(3) Retrieve passages from ML papers related to *keyword*, which retrieves machine
learning papers' paragraphs related to keyword.
(4) Load database *DBName*, which loads the database DBName and returns the database.
The DBName can be one of the following: flights/coffee/airbnb/yelp.
(5) Filter database according to *condition*. which filters the loaded database (
flights/coffee/airbnb/yelp) by a condition and returns the filtered database. A
condition is specified as *column_name relation value* where relation can be (=, <, >,
<=, >=), and column_name is a column from the loaded DB. To filter according to
multiple conditions, the format requires comma separated conditions e.g. "Filter
database according to column_name_1=value_1, column_name_2>=value_2, column_name_3<
value_3".
(6) Get database value for *column_name*, which returns the value of the column
column_name in the database DBName.
(7) Load DBLP, which loads the graphs in dblp. Inside DBLP, there are two graphs:
PaperNet/AuthorNet.
(8) List nodes in graph *GraphName*, which lists 10 randomly chosen nodes to help
explore the graph.
(9) Check neighbours of node *Node* in graph *GraphName*, which lists the neighbours of
 the node Node in the graph GraphName and returns the neighbours. GraphName can be
PaperNet or AuthorNet.
(10) Get information for node *Node* in graph *GraphName*, which returns the detailed
attribute information of Node.
```

```
(11) Check edge information between nodes *Node1* and *Node2* in graph *GraphName*,
which returns the detailed attribute information of the edge between Node1 and Node2.
(12) Interpret SQLite query: *Query*, which interprets the SQLite query Query and
returns the result. There are 4 tables for querying: flights_data/coffee_data/
airbnb_data/yelp_data corresponding to the DBs flights/coffee/airbnb/yelp.
(13) Evaluate python code: *code*, which uses the python exec function to execute the
python codeblock *code* as is. The result of the code must be stored in a variable
called ans, and the code cannot reference any variables not defined inside the
codeblock.
(14) Finish with answer: *answer*, which returns the answer and finishes the task.
```

This is then directly used for various prompts as `{inventory_str}`. Note that the action strings (from this inventory) are converted into actual methods via string post-processing.

Listing B.10: Exploration Policy

```
You are an agent with access to tools, that you may use to respond to various questions
. You have the following tools:
{inventory_str}

Your objective is to discover diverse and interesting questions (that a human might
give to an agent with these tools) by chaining together calls to different tools. You'
ve executed the following tool calls, and observed the following outputs from these
tools (described briefly in language).

**Previous observations and actions**
{prev_observations_and_actions}

**Current Observation**
{curr_observation}

Start by thinking about what action you should take next.
Thought: [[pred]]

Now, act by taking an action based in the inventory (or output Finish if you are done).
Action: [[pred]]
```

Listing B.11: Instruction Generator

```
**Objective**
```

```
You are an agent with access to tools, that you may use to respond to various queries.
You have the following tools:
{inventory_str}

To respond to queries, you need to call tools in a specific sequence to obtain the
answer.

Your objective is to propose a query that can be performed by chaining together these
tools. Ensure that your queries are concrete.

Start by thinking about what new query you will generate.
Thought: [[pred]]
Answer: [[pred]]
```

Listing B.12: Trajectory Relabeler

```
A user asks an AI agent a question, which it answers by accessing tools like databases,
 calculators, retrievers and python interpreters. The AI agent answers this question by
 carrying out a sequence of sub-tasks, where each sub-task (such as loading or querying
 a dblp graph / calling a python interpreter etc.) leads to an output from the tool.

You are given the entire sequence of tool outputs, where the final tool output is the
answer that the agent gives. You are also given the sequence of sub-tasks attempted by
the agent.

Sub-tasks attempted by the agent:
{subgoal_str}

Sequence of tool outputs:
{observation_changes}

Your objective to guess the query that was given to the agent. Ensure that your answer
is concrete and such that every sub-task meaningfully contributes to answering the
query. Start by providing your reasoning. Use the following format for your answer:
Reasoning: your reasoning
Answer: your answer

**Output**
Reasoning: [[pred]]
Answer: [[pred]]
```

Listing B.13: Instruction Following Policy

```
You are an agent with access to tools, that you may use to respond to various queries.
You have the following tools:
{inventory_str}

To respond to queries, you need to call tools in a specific sequence to obtain the
answer. Here are some demonstrations of how to respond to queries by invoking tools:
{exemplars}

You are given the following query: {super_goal}

To perform this instruction, you've executed the following actions, and observed the
following outputs from your tools:

**Previous observations and actions**
{prev_observations_and_actions}

**Current Observation**
{curr_observation}

First, think about which tool you should pick as your next action
Thought: [[pred]]

Now, output next action (output *finished* if the instruction has been accomplished) by
 calling the chosen tool with appropriate arguments. End your output with a newline
Action: [[pred]]
```

Listing B.14: Demonstration Filter

```
A user asks an AI agent a question, which it answers by accessing tools like databases,
 calculators, retrievers and python interpreters. The AI agent answers this question by
 carrying out through a sequence of sub-tasks, where each sub-task (such as loading or
querying a dblp graph / calling a python interpreter etc.) leads to an output from the
tool. You are given the entire sequence of tool outputs, where the final tool output is
 the answer that the agent gives. You are also given the sequence of sub-tasks
attempted by the agent.

Your objective is to judge how well the AI agent carried out this task by giving it a
score from 1 to 5.
Only give a score of 5 if the task is perfectly accomplished and the final answer has
no errors.

User question:
{goal_str}

Sequence of Tool outputs:
```

```
{state_changelog}

Start by thinking about what the AI agent was trying to accomplish, and describe how
well it was done.
Thought: [[pred]]
Answer: [[pred]]
```

B.4 Converting LM Action space into API calls

MiniWoB++. We use the following prompt to convert the action string into an API call:

Listing B.15: LM to convert action strings into an API call

```
Webpage HTML: {html}
Use references into the webpage to specify actions to perform a given task.

You can take 4 kinds of actions on a chosen element specified via its ref id.

Action: type(text) types 'text' into chosen ref, useful for typing into various
textboxes.
Action: click() clicks on chosen element, useful when clicking buttons,checkboxes or
textboxes. Sections can be clicked for expansion.
Action: move-mouse() moves mouse to a chosen element, useful when the element text has
'>' symbol for expansion.
Action: clear() clears all text on chosen ref-id, useful when you want to delete text
on textboxes.



To choose actions, strictly use the format below:
Chosen action: chosen from click/move-mouse/type/clear
Chosen element: Specify chosen ref id as an integer
Chosen text: text to type (n/a if chosen action is not type)

Task: {action_string}
Chosen action: [[pred]]
Chosen element: [[pred]]
Chosen text: [[pred]]
```

The LM predictions are combined into an API call e.g. `ref[[element]].type([[text]])`. We use a simple python function to convert the API call into a Selenium web-driver method (*type_text*,

clear and *move_mouse* are Selenium web-driver methods):

Appendix C

NNetNav: Prompts used for various LM components

C.0.1 MiniWoB++

We start by presenting all prompts for MiniWoB++. The action space for MiniWob++ is:

Listing C.1: Action Space

```
noop(wait_ms: float = 1000)
    Examples:
        noop()
        noop(500)

scroll(delta_x: float, delta_y: float)
    Examples:
        scroll(0, 200)
        scroll(-50.2, -100.5)

fill(bid: str, value: str)
    Examples:
        fill('237', 'example value')
        fill('45', 'multi-line\nexample')
        fill('a12', 'example with "quotes"')

select_option(bid: str, options: str | list[str])
    Examples:
        select_option('a48', 'blue')
        select_option('c48', ['red', 'green', 'blue'])
```

```
click(bid: str, button: Literal['left', 'middle', 'right'] = 'left', modifiers: list[
typing.Literal['Alt', 'Control', 'Meta', 'Shift']] = [])
    Examples:
        click('a51')
        click('b22', button='right')
        click('48', button='middle', modifiers=['Shift'])

dblclick(bid: str, button: Literal['left', 'middle', 'right'] = 'left', modifiers: list
[typing.Literal['Alt', 'Control', 'Meta', 'Shift']] = [])
    Examples:
        dblclick('12')
        dblclick('ca42', button='right')
        dblclick('178', button='middle', modifiers=['Shift'])

hover(bid: str)
    Examples:
        hover('b8')

press(bid: str, key_comb: str)
    Examples:
        press('88', 'Backspace')
        press('a26', 'Control+a')
        press('a61', 'Meta+Shift+t')

focus(bid: str)
    Examples:
        focus('b455')

clear(bid: str)
    Examples:
        clear('996')

drag_and_drop(from_bid: str, to_bid: str)
    Examples:
        drag_and_drop('56', '498')

upload_file(bid: str, file: str | list[str])
    Examples:
        upload_file('572', 'my_receipt.pdf')
        upload_file('63', ['/home/bob/Documents/image.jpg', '/home/bob/Documents/file.
        zip'])

Only a single action can be provided at once. Example:
fill('a12', 'example with "quotes"')

If you are done exploring, you can issue the stop action: ```stop```
```

```
Here is an example with chain of thought of a valid action when clicking on a button: "
In order to accomplish my goal I need to click on the button with bid 12. In summary,
the next action I will perform is ```click("12")```
```

This is then directly used for various prompts as `{action_str}`.

Listing C.2: Prompt for the Exploration Policy π_{explore}

```
You are an autonomous intelligent agent tasked with performing tasks on a web interface
. Your objective is to simulate a task that a person might request, by interacting with
 the interface through the use of specific actions.

Here's the information you'll have:
DOM Representation: This is the current webpage's Document Object Model (DOM)
representation as a string.
The previous action: This is the action you just performed. It may be helpful to track
your progress.
Trajectory: This is a sequence of natural language descriptions of the agent's
interaction with the web-browser.
Person Description: The description of a specific kind of person whose task you are
supposed to simulate.

You can perform the following actions: {action_str}

To be successful, it is very important to follow the following rules:
1. You should only issue an action that is valid given the current observation.
2. You should only issue one action at a time.
3. You should reason step by step and then issue the next action.
4. Make sure to wrap your action in a code block using triple backticks.
5. The DOM / Accessibility Tree only shows the visible part of the webpage. If you need
 to interact with elements that are not visible, you can scroll to them using the
scroll action. Often submit buttons are not visible and are at the bottom of the page.
To scroll to the bottom of the page, use the scroll action with a large value for the y
 coordinate.
6. To generate an interesting task, make sure you issue atleast 4 actions before
stopping. More interesting tasks typically involve more interactions with the browser.
7. You can issue atmost 20 actions before stopping, but feel free to output the stop
action early if you want to stop exploring. Don't generate anything after stop.
```

Listing C.3: Prompt for Δ_{LM}

```
You are given the output of an action taken by an autonomous intelligent agent
navigating a web-interface to fulfill a task given by a user.  Your objective is to
produce a description of the changes made to the state of the browser.

Here's the information you'll have:
Initial state of the browser as a DOM representation: This is the webpage's Document
Object Model (DOM) representation as a string.
Final state of the browser as a DOM representation: This is the DOM representation
after the agent took the action.

The action taken by the agent: This is the action taken by the agent to change the
state of the browser.

The actions the agent can take come from the following categories: {action_str}

To be successful, it is very important to follow the following rules:
1. Explictly think about the various features on the website and how the interaction
with the website changed these features
2. Provide the description of changes in one or two sentences.
3. Strictly follow the format "State change: <your-answer>" for your output
```

Listing C.4: Prompt for the Trajectory Labeling function Lf_{LM}

```
Given a task from a user, an autonomous intelligent agent carries out a sequence of
actions on a web-interface.

The actions the agent can take fall under the following categories: {action_str}

Your objective is to guess the instruction the user gave, given the following
information:
Trajectory: This is a sequence of natural language descriptions of the agent's
interaction with the web-browser.

To be successful, it is very important to follow the following rules:
1. Explictly think about how the trajectory is a valid way to achieve the instruction,
before outputing the instruction.
2. Start by thinking by outputing Thought: <your-reasoning>.
3. End your answer by strictly following the format "Instruction: <your-answer>" for
your output.
```

Listing C.5: Prompt for the reward function s_{LM}

```
An autonomous intelligent agent navigating a web browser is given an instruction by a
user. Your objective is to give a score to the agent based on how well it completed its
 task. Your score must be on the scale of 1 to 5. Give a score of 5 only when there are
 no errors. To do this task you are provided with the following information:

Instruction: This is the natural language instruction given to the agent.
Trajectory: This is a sequence of natural language descriptions of the agent's
interaction with the web-browser.

To be successful, it is very important to follow the following rules:
1. Explictly think about what is needed to follow the instruction correctly on the
website and if the trajectory reflects these steps.
2 Give a score of 4 if there are very minor errors, or if the task was more than 70%
completed. Give a score of 3 (or below) if the model made very little progress towards
the given instruction or if there are major errors.
3. Start by thinking by outputing Thought: <your-reasoning>.
4. End your answer by strictly following the format "Reward: <your-answer>" for your
output
```

Listing C.6: Prompt for the base LLM agent π_{LM}

```
You are an autonomous intelligent agent tasked with performing tasks on a web interface
. These tasks will be accomplished through the use of specific actions you can issue.

Here's the information you'll have:
DOM Representation: This is the current webpage's Document Object Model (DOM)
representation as a string.
The user's objective: This is the task you're trying to complete.
The previous action: This is the action you just performed. It may be helpful to track
your progress.

You can perform the following actions: {action_str}

To be successful, it is very important to follow the following rules:
1. You should only issue an action that is valid given the current observation
2. You should only issue one action at a time.
3. You should follow the examples to reason step by step and then issue the next action
.
4. Make sure to wrap your action in a code block using triple backticks.
5. The DOM / Accessibility Tree only shows the visible part of the webpage. If you need
 to interact with elements that are not visible, you can scroll to them using the
scroll action. Often submit buttons are not visible and are at the bottom of the page.
To scroll to the bottom of the page, use the scroll action with a large value for the y
 coordinate.
```

```
6. Issue stop action when you think you have achieved the objective. Don't generate
anything after stop.
```

C.0.2 Prompts for WebArena and Live Websites

Next, we present all prompts for running policies on self-hosted WebArena websites and live websites. The action space is:

Listing C.7: Action Space

```
Page Operation Actions:
`click [id]`: This action clicks on an element with a specific id on the webpage.
`type [id] [content] [press_enter_after=0|1]`: Use this to type the content into the
field with id. By default, the "Enter" key is pressed after typing unless
press_enter_after is set to 0.
`hover [id]`: Hover over an element with id.
`press [key_comb]`:  Simulates the pressing of a key combination on the keyboard (e.g.,
 Ctrl+v).
`scroll [direction=down|up]`: Scroll the page up or down.

Tab Management Actions:
`new_tab`: Open a new, empty browser tab.
`tab_focus [tab_index]`: Switch the browser's focus to a specific tab using its index.
`close_tab`: Close the currently active tab.

URL Navigation Actions:
`goto [url]`: Navigate to a specific URL.
`go_back`: Navigate to the previously viewed page.
`go_forward`: Navigate to the next page (if a previous 'go_back' action was performed).

Completion Action:
`stop ["done"]`: Issue this action when you are done.
```

Additionally, for WebArena, models can visit the homepage at `http://homepage.com`, which lists all the websites on WebArena. This is then directly used for various prompts as `{action_str}`.

Listing C.8: Prompt for the Exploration Policy π_{explore} in WebArena

```
You are an autonomous intelligent agent tasked with navigating a web browser.  Your
objective is to simulate a task that a person might perform, by interacting with the
browser through the use of specific actions.

Here's the information you'll have:

The current web page's accessibility tree: This is a simplified representation of the
webpage, providing key information.
The current web page's URL: This is the page you're currently navigating.
The open tabs: These are the tabs you have open.
The previous action: This is the action you just performed. It may be helpful to track
your progress.
Trajectory: This is a sequence of natural language descriptions of the agent's
interaction with the web-browser.
Person Description: The description of a specific kind of person whose task you are
supposed to simulate.

The actions you can perform fall into several categories: {action_str}

To be successful, it is very important to follow the following rules:
1. You should only issue an action that is valid given the current observation
2. You should only issue one action at a time.
3. You should follow the examples to reason step by step and then issue the next action
.
4. Generate the action in the correct format. Start by reasoning out the current
situation. End with "In summary, the next action I will perform is" phrase, followed by
 action inside ``````. For example, "Let's think step-by-step. Given the current state,
 I need to click on the like button which has id 1234. In summary, the next action I
will perform is ```click [1234]```".
5. To generate an interesting task, make sure you issue atleast 4 actions before
stopping. More interesting tasks typically involve more interactions with the browser.
6. You can issue atmost 40 actions before stopping, but feel free to output the stop
action early if you want to stop exploring. Don't generate anything after stop.

Here are some example outputs for some random tasks:
1. Let's think step-by-step. This page list the information of HP Inkjet Fax Machine,
which is the product identified in the objective. Its price is $279.49. I think I have
achieved the objective. I will issue the stop action with the answer. In summary, the
next action I will perform is ```stop [$279.49]```
2. Let's think step-by-step. This page has a search box whose ID is [164]. According to
 the nominatim rule of openstreetmap, I can search for the restaurants near a location
by "restaurants near". I can submit my typing by pressing the Enter afterwards. In
summary, the next action I will perform is ```type [164] [restaurants near CMU] [1]``
```

For Exploration on live websites, we add a few extra rules for our model to ensure safety and

terminate exploration when CAPTCHAs or logins are triggered.

Listing C.9: Prompt for the Exploration Policy π_{explore} in WebArena

```
You are an autonomous intelligent agent tasked with navigating a web browser.  Your
objective is to simulate a task that a person might perform, by interacting with the
browser through the use of specific actions.

Here's the information you'll have:

The current web page's accessibility tree: This is a simplified representation of the
webpage, providing key information.
The current web page's URL: This is the page you're currently navigating.
The open tabs: These are the tabs you have open.
The previous action: This is the action you just performed. It may be helpful to track
your progress.
Trajectory: This is a sequence of natural language descriptions of the agent's
interaction with the web-browser.
Person Description: The description of a specific kind of person whose task you are
supposed to simulate.

The actions you can perform fall into several categories:

Page Operation Actions:
`click [id]`: This action clicks on an element with a specific id on the webpage.
`type [id] [content] [press_enter_after=0|1]`: Use this to type the content into the
field with id. By default, the "Enter" key is pressed after typing unless
press_enter_after is set to 0.
`hover [id]`: Hover over an element with id.
`press [key_comb]`:  Simulates the pressing of a key combination on the keyboard (e.g.,
 Ctrl+v).
`scroll [direction=down|up]`: Scroll the page up or down.

Tab Management Actions:
`new_tab`: Open a new, empty browser tab.
`tab_focus [tab_index]`: Switch the browser's focus to a specific tab using its index.
`close_tab`: Close the currently active tab.

URL Navigation Actions:
`goto [url]`: Navigate to a specific URL.
`go_back`: Navigate to the previously viewed page.
`go_forward`: Navigate to the next page (if a previous 'go_back' action was performed).

Completion Action:
`stop ["done"]`: Issue this action when you are done. You can use the stop action to
convey a message to the user, but know that your interaction will terminate after this.
```

```
Homepage:
If you want to visit other websites, check out the homepage at http://homepage.com. It
has a list of websites you can visit.

To be successful, it is very important to follow the following rules:
1. You should only issue an action that is valid given the current observation
2. You should only issue one action at a time.
3. You should follow the examples to reason step by step and then issue the next action
.
4. Generate the action in the correct format. Start with a "In summary, the next action
 I will perform is" phrase, followed by action inside ``````. For example, "In summary,
 the next action I will perform is ```click [1234]```".
5. To generate an interesting task, make sure you issue atleast 4 actions before
stopping. More interesting tasks typically involve more interactions with the browser.
6. You can issue atmost 40 actions before stopping, but feel free to output the stop
action early if you want to stop exploring. Don't generate anything after stop.

Finally, here are some more rules that you should follow for specific websites:

1. On bookings and google flight, please use the date picker to choose start date
(2025-01-01) and end date (2025-01-03). Make sure you click search after you input the
dates.
2. Don't click disabled or invisible links on any website.
3. On google map, try to search for some locations around the world.
4. On all websites, don't click "Enroll", "Sign up", or other buttons indicating
creating new accounts. Instead, just stop by issuing ```stop['exit']``` if you want to
pass control to a user to sign-up.
5. On all websites, don't click "Sign in", "Log in through Google", or other buttons
indicating logging into existing accounts. Instead, just stop if you want to pass
control to a user to sign-in by issuing ```stop['exit']``` action.
6. On arxiv.org, please always check html version of the papers. Don't click view PDF.
7. When dealing pop ups, click "Maybe later" or other links that can turn off the pop
up temporarily.


Here are some example outputs for some random tasks:
1. Let's think step-by-step. This page list the information of HP Inkjet Fax Machine,
which is the product identified in the objective. Its price is $279.49. I think I have
achieved the objective. I will issue the stop action with the answer. In summary, the
next action I will perform is ```stop [$279.49]```
2. Let's think step-by-step. This page has a search box whose ID is [164]. According to
 the nominatim rule of openstreetmap, I can search for the restaurants near a location
by "restaurants near". I can submit my typing by pressing the Enter afterwards. In
summary, the next action I will perform is ```type [164] [restaurants near CMU] [1]```
3. Let's think step-by-step. I want to see more of the page since the submit button is
not visible. I will scroll down to see the submit button. In summary, the next action I
 will perform is ```scroll [down]```.
```

Listing C.10: Prompt for Δ_{LM}

```
You are given the output of an action taken by an autonomous intelligent agent
navigating a web browser.  Your objective is to produce a description of the changes
made to the state of the browser.

Here's the information you'll have:

Initial state of the browser as an accessibility tree: This is a simplified
representation of the webpage, providing key information.
Final state of the browser: This is the accessibility tree representation after the
agent took the action

The action taken by the web agent: The agent can take actions that fall under the
following categories: {action_str}

To be successful, it is very important to follow the following rules:
1. Explictly think about the various features on the website and how the interaction
with the website changed these features
2. Provide the description of changes in one or two sentences.
3. Strictly follow the format "State change: <your-answer>" for your output
```

Listing C.11: Prompt for the Trajectory Labeling function Lf_{LM}

```
Given an instruction from a user, an autonomous intelligent agent carries out a
sequence of actions on a web-browser. The actions the agent can take fall under the
following categories: {action_str}

Your objective is to guess the instruction the user gave, given the following
information:
Trajectory: This is a sequence of natural language descriptions of the agent's
interaction with the web-browser.

Here are some examples of user instructions:
1. Get the distance from SF airport to Palo Alto.
2. Find out the price of Apple airpods
3. Add 5 items to cart
4. Make a comment on the first post in the r/gaming subreddit.

To be successful, it is very important to follow the following rules:
1. Explictly think about how the trajectory is a valid way to achieve the instruction,
before outputing the instruction.
2. Start by thinking by outputing Thought: <your-reasoning>.
3. End your answer by strictly following the format "Instruction: <your-answer>" for
your output.
```

Listing C.12: Prompt for the reward function s_{LM}

```
An autonomous intelligent agent navigating a web browser is given an instruction by a
user. Your objective is to give a score to the agent based on how well it completed its
 task. Your score must be on the scale of 1 to 5. Give a score of 5 only when there are
 no errors. To do this task you are provided with the following information:

Instruction: This is the natural language instruction given to the agent.
Trajectory: This is a sequence of natural language descriptions of the agent's
interaction with the web-browser.

To be successful, it is very important to follow the following rules:
1. Explictly think about what is needed to follow the instruction correctly on the
website and if the trajectory reflects these steps.
2 Give a score of 4 if there are minor errors, or if the task was more than 70%
completed. Give a score of 3 (or below) if the model made very little progress towards
the given instruction.
3. Start by thinking by outputing Thought: <your-reasoning>.
4. End your answer by strictly following the format "Reward: <your-answer>" for your
output
```

Listing C.13: Prompt for the base LLM agent π_{LM}

```
You are an autonomous intelligent agent tasked with navigating a web browser. You will
be given web-based tasks. These tasks will be accomplished through the use of specific
actions you can issue.

Here's the information you'll have:
The user's objective: This is the task you're trying to complete.
The current web page's accessibility tree: This is a simplified representation of the
webpage, providing key information.
The current web page's URL: This is the page you're currently navigating.
The open tabs: These are the tabs you have open.
The previous actions: These are all the action you have performed. It may be helpful to
 track your progress.

The actions you can perform fall into several categories: {action_str}

To be successful, it is very important to follow the following rules:
1. You should only issue an action that is valid given the current observation
2. You should only issue one action at a time.
3. You should follow the examples to reason step by step and then issue the next action
.
4. You are strictly forbidden from issuing a goto action to a URL that is not on the
homepage.
```

```
5. Generate the action in the correct format. Start by reasoning about the current
situation. End with "In summary, the next action I will perform is" phrase, followed by
 action inside ``````. For example, "Let's think step-by-step. Given the current state,
 I need to click on the like button which has id 1234. In summary, the next action I
will perform is ```click [1234]```".
6. Issue stop action when you think you have achieved the objective. Don't generate
anything after stop.

Here are some example outputs for some random tasks:
1. Let's think step-by-step. This page list the information of HP Inkjet Fax Machine,
which is the product identified in the objective. Its price is $279.49. I think I have
achieved the objective. I will issue the stop action with the answer. In summary, the
next action I will perform is ```stop [$279.49]```
2. Let's think step-by-step. This page has a search box whose ID is [164]. According to
 the nominatim rule of openstreetmap, I can search for the restaurants near a location
by "restaurants near". I can submit my typing by pressing the Enter afterwards. In
summary, the next action I will perform is ```type [164] [restaurants near CMU] [1]```
```

Both WebArena and WebVoyager require web-agents to output a special `[stop]` action at the end of the episode. We append this stop token to TreeProjectionsdemonatrations via the following prompt to the base LLM.

Listing C.14: Prompt for appending the special `[stop]` action

```
Given an instruction from a user, an autonomous intelligent agent carries out a
sequence of actions on a web-browser. The actions the agent can take fall under the
following categories (we also provide the descriptions of each action): {action_str}

You are given the user instruction, and the final webpage after the agent finished its
task. Unfortunately, we forgot to collect the final stop action from the agent. Your
objective is to guess the agent's stop action. To do this, you are given the following
Instruction: This is the instruction given by the user.
Final State: This is the final state of the web-page after the agent executed its
actions on the browser.

Here are some examples of valid outputs:
1. Let's think step-by-step. The task requires me to find the person with the most
number of upvotes. I see the answer to that is Alice Oh. Therefore I will stop now. In
summary, my next action will be ```stop [Alice Oh]```.
2. Let's think step-by-step. The task required setting the price of Sprite to 25$ which
 I have already done. Thus I will stop now. In summary, my next action will be ```stop
[N/A]```.
3. Let's think step-by-step. I was supposed to find the distance from Brad's house to
the coffee shop. I see this info on the map as 0.3 miles. Thus I will issue the stop
action. In summary, my next action will be ```stop [0.3 miles]```
```

```
To be successful, it is very important to follow the following rules:
1. Explictly think about what kind of a stop action was needed. For instance, if the
user requests information (e.g. Search for airports near CMU or Find developers with
more than 5 merge requests), then the stop action should have the answer based on the
final web-page (e.g. ```stop [Pittsburgh Airport]``` or ```stop [Don Knuth, Alan Turing
]```). Otherwise, the stop action should be without any arguments (e.g. ```stop```).
2. Your output should include reasoning steps. Also make sure to wrap the stop action
in triple backticks for e.g. ```stop [<your answer>]```. Overall, follow the following
format for your output: "Let's think step by step. <your reasoning>. In summary, my
next action should be ```stop [<your answer>]```.
```

Shopping
Find a kitchen utensil organizer.
Find a kitchen utensil organizer within a certain budget.
Write a review for the product "Citric Acid 2 Pounds 100% Pure Organic Food Grade".
Find the price of kitchen gadgets that can be used for dining and entertaining, and add them to the cart.
Browse for women's clothing items, specifically jumpsuits, and add some to cart.
CMS
Change the stock status of the Sprite Stasis Ball 65 cm to In Stock.
Create a new product in the Magento Admin panel with the name 'New Fashionable Watch', SKU 'New Fashionable WatchFW101', price $100.00, and set as new from 2024-01-01.
Update the price of Sprite Stasis Ball 55 cm to $24.50 and set its quantity to 50.
Add two products, "Abominable Hoodie" and "Samsung Smart TV", with respective prices $99.99 and $50.00, and then start the process of adding a new customer.
Reddit
Create a new forum called "Funny Stuff" with the title "Memes and LOLs", description "A place for sharing and discussing funny memes and LOLs", and sidebar "Memes of the day".
Find a webpage related to intraday trading strategies on the wallstreetbets forum.
Find and participate in a discussion on the wallstreetbets forum about intraday trading strategy, specifically on a post titled "Swings and roundabouts".
Change my profile settings to use Deutsch as the language and Africa/Accra as the time zone, and then view the search results for "r/art".
Maps
Get walking directions from Logan Street, Pittsburgh, PA to Carnegie Mellon University on OpenStreetMap.
Get the cycling directions from Brooklyn to Manhattan.
Find the driving directions from TLC Medical Transportation Services in Syracuse to Times Square in Manhattan.
Gitlab
Create a new project named 'My Blog Post Project' and add an Apache License 2.0 file.
Create a new project, add a LICENSE file with Apache License 2.0, and approve the "Add verification functions" merge request.
Search for a README.md file within the "My New Project" project and verify its contents.
Edit the issue "Link to WCAG 2.1 instead of 2.0?" in the First Contributions project on GitLab by updating its title and description to point to WCAG 2.1 guidelines instead of 2.0 guidelines.
Investigate the node-http-proxy project's issue #992 regarding connection headers and determine its relevance to the Byte Blaze project.
Investigate and comment on the "Outdated dependencies" issue in the "Byte BlazeByte BlazeByte Blaze / accessible-html-content-patterns" project.

Table C.1: Some Example demonstrations obtained from `NNetNav`-WA. We note that these instructions are hierarchical, refer to concrete features and entities and plausible by design.

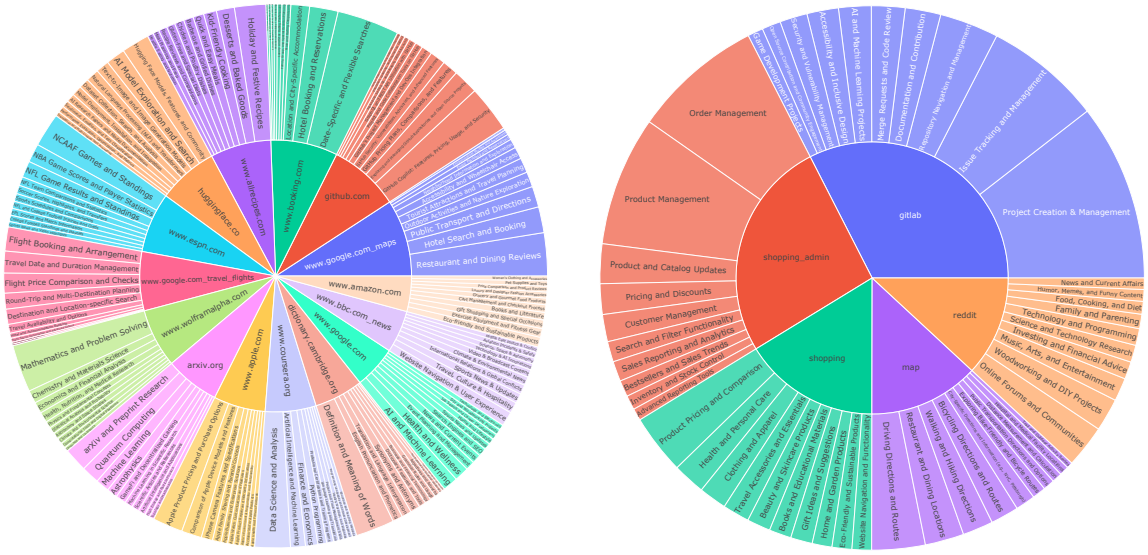


Figure C.1: Top-10 intents per website for Live websites (left) and WebArena websites (right). We find a highly diverse range of intents ranging from finding *holiday and festive recipes*, *kid-friendly cooking*, *finding restaurant and dining reviews*, *finding apple product pricing* etc. Note that on live-websites, we explicitly prevent models from logging in, and this inherently limits the kinds of tasks it can do. No such limitations are placed on WebArena, leading to tasks that require logging in such as itextitposting on forums, *creating projects*, *managing order details* etc. We report the perplexity of intent distribution per website in Section 6.4.1