Universal In-context Approximation

Aleksandar Petrov

UNIVERSITY OF OXFORD

Universal In-context Approximation

A thesis submitted for
the degree of Doctor of Philosophy

Aleksandar Petrov
Reuben College
University of Oxford

Supervision
Prof. Philip H.S. Torr, FREng, FRS
Dr. Adel Bibi

Thesis committee
Dr. Rudy Bunel
Dr. João F. Henriques

Summer 2025
Oxford

Abstract

The explosive rise of large language model capabilities has shifted research and practice from task-specific training to in-context learning via prompting. This raises a fundamental question: can a model with fixed weights solve novel tasks as effectively as a fine-tuned one? This thesis investigates the theoretical capabilities and limitations of in-context learning across major sequence model architectures, introducing the formal notion of universal in-context approximation, where a single model can approximate any function from a given class by only selecting an appropriate input prompt.

Our investigation begins by exploring the limitations of prompting in transformers. We first prove a significant restriction: prompting and prefix-tuning are fundamentally incapable of changing a model's learned attention patterns over the user-provided content. This finding initially suggests that prompting is strictly less powerful than fine-tuning. However, we then demonstrate the contrary: transformers are universal in-context approximators. This thesis resolves the apparent contradiction by showing that universality is achieved not by altering attention over the content, but by leveraging the transformer's attention mechanism's ability to approximate smooth functions to arbitrary precision. Most notably, the model size is constant in the target precision.

Furthermore, we extend this inquiry beyond attention-based models to fully recurrent architectures, including RNNs, LSTMs and modern State Space Models (SSMs). As these models lack an attention mechanism, we develop a different method for proving their in-context capabilities: a compiler that translates high-level procedural programs into the parameters of recurrent models. Using this framework, we prove that these fully recurrent architectures are also universal in-context approximators, possibly even more efficiently so than the transformer.

Collectively, these results establish that, from a representational standpoint, in-context learning can be as expressive as full model retraining. This work provides a rigorous foundation for understanding the emergent capabilities of large-scale models and shows that, at least in theory, a carefully crafted prompt can go a long way.

List of Publications

This is a list of all publications that I have worked on during the course of my DPhil. The publications that have been integrated in this thesis are marked with a star (⋆).

1. Aleksandar Petrov and Marta Kwiatkowska. 2022. Robustness of unsupervised representation learning without labels. *Preprint arXiv:2210.04076*

2. Aleksandar Petrov, Francisco Eiras, Amartya Sanyal, Philip H.S. Torr, and Adel Bibi. 2023a. Certifying ensembles: A general certification theory with S-Lipschitzness. *International Conference on Machine Learning (ICML)*

3. Aleksandar Petrov, Emanuele La Malfa, Philip H.S. Torr, and Adel Bibi. 2023b. Language model tokenizers introduce unfairness between languages. In *Advances in Neural Information Processing Systems*

⋆4. Aleksandar Petrov, Philip H.S. Torr, and Adel Bibi. 2024c. When do prompting and prefix-tuning work? A theory of capabilities and limitations. In *International Conference on Learning Representations (ICLR)*

5. Francisco Eiras, Aleksandar Petrov, Philip H.S. Torr, M Pawan Kumar, and Adel Bibi. 2025. Do as I do (safely): Mitigating task-specific fine-tuning risks in large language models. In *International Conference on Learning Representations (ICLR)*

⋆6. Aleksandar Petrov, Philip H.S. Torr, and Adel Bibi. 2024b. Prompting a pretrained transformer can be a universal approximator. In *International Conference on Machine Learning (ICML)*

7. Francisco Eiras, Aleksandar Petrov, Bertie Vidgen, Christian Schroeder de Witt, Fabio Pizzati, Katherine Elkins, Supratik Mukhopadhyay, Adel Bibi, Botos Csaba, Fabro Steibel, et al. 2024. Position: Near to mid-term risks and opportunities of open-source generative AI. In *International Conference on Machine Learning (ICML)*

⋆8. Aleksandar Petrov, Tom A Lamb, Alasdair Paren, Philip H.S. Torr, and Adel Bibi. 2024a. Universal in-context approximation by prompting fully recurrent models. In *Advances in Neural Information Processing Systems*

9. Emanuele La Malfa, Aleksandar Petrov, Simon Frieder, Christoph Weinhuber, Ryan Burnell, Raza Nazar, Anthony G. Cohn, Nigel Shadbolt, and Michael Wooldridge. 2024. Language-Models-as-a-Service: Overview of a new paradigm and its challenges. *Journal of Artificial Intelligence Research*, 80

10. Aleksandar Petrov, Shruti Agarwal, Philip H.S. Torr, Adel Bibi, and John Collomosse. 2025a. On the coexistence and ensembling of watermarks. In *Advances in Neural Information Processing Systems*

11. Aleksandar Petrov, Mark Sandler, Andrey Zhmoginov, Nolan Miller, and Max Vladymyrov. 2025c. Long context in-context compression by getting to the gist of gisting. *Preprint arXiv:2504.08934*

12. Aleksandar Petrov, Pierre Fernandez, Tomáš Souček, and Hady Elsahar. 2025b. We can hide more bits: The unused watermarking capacity in theory and in practice. *Preprint arXiv:2510.12812*

Contents

Abstract iii
List of Publications v
Contents vii
Acknowledgements 1

1 Introduction 3

2 Brief Survey of Universal (In-Context) Approximation 11
2.1 Universal approximation results for the MLP . . . 13
2.2 Universal approximation results for sequence-to-sequence architectures . . . 15
2.3 Universal in-context approximation results . . . 18
2.4 In-context learning and formal languages . . . 20

3 When Do Prompting and Prefix-Tuning Work? A Theory of Capabilities and Limitations 23
3.1 Introduction . . . 23
3.2 Background . . . 25
3.2.1 The transformer architecture . . . 25
3.2.2 Context-based fine-tuning of a pretrained model . . . 26
3.3 Soft prompting has more capacity than prompting . . . 27
3.4 Prefix-tuning can only bias the output of an attention head . . . 28
3.5 The bias can elicit skills from the pretrained model . . . 31
3.6 Effects of prefix-tuning beyond the single attention layer . . . 35
3.7 Discussion and related works . . . 36
3.8 Conclusion . . . 38

4 Prompting a Pretrained Transformer Can Be a Universal Approximator 41
4.1 Introduction . . . 41
4.2 Background material . . . 43

4.2.1 Transformer architecture 43
4.2.2 Universal approximation 44
4.3 Universal approximation with a single attention head 47
4.4 Universal approximation of sequence-to-sequence functions 53
4.5 Discussion and conclusions 55

5 Universal In-Context Approximation by Prompting Fully Recurrent Models 59
5.1 Introduction 59
5.2 Preliminaries 61
5.3 Linear State Recurrent Language (LSRL) 64
5.4 Universal in-context approximation with Linear RNNs 68
5.4.1 Approximating continuous functions in $\mathcal{C}^{\text{vec}}$ 68
5.4.2 Approximating functions over token sequences in $\mathcal{C}^{\text{tok}}$ 70
5.5 Stable universal in-context approximation with Gated Linear RNNs 71
5.6 Universal in-context approximation with non-linear (Gated) RNNs 74
5.7 Discussion and conclusions 74

6 Conclusions and Outlook 79

Bibliography 83

A Appendices for When Do Prompting and Prefix-Tuning Work? A Theory of Capabilities and Limitations 101
A.1 Constructing transformers that utilize the capacity of the embedding space . 101
A.1.1 Unconditional generation for a single virtual token 101
A.1.2 Conditional generation for a single virtual token ($n_X = n_Y = 1$) 103
A.1.3 Conditional generation for longer responses ($n_X = 1, n_Y > 1$) 104
A.1.4 Conditional generation for longer user inputs ($n_X > 1, n_Y = 1$) 105
A.2 Attention distribution over the prefix 107
A.3 Expressivity of prefix-tuning across deeper models 108
A.4 Extended results 111
A.4.1 Pretraining as in Section 3.5 111
A.4.2 Extended pretraining 113
A.5 Further experiments with memorization 115

B Appendices for Prompting a Pretrained Transformer Can Be a Universal Approximator 117
B.1 Background on analysis on the sphere 117
B.2 A Jackson-type bound for universal approximation on the unit hypersphere . 120
B.3 A Jackson-type bound for approximation with a split attention head 135
B.4 Additional results 138

C Appendices for Universal In-Context Approximation by Prompting Fully Recurrent Models 139
C.1 Computation graph debranching rules 139
C.2 Error bound on the approximation scheme for continuous functions 142
C.3 Gated RNNs are GRU models 143
C.3.1 Representing the state update as a GRU layer 143
C.3.2 Representing each MLP layer as a GRU layer 144
C.3.3 Representing the multiplicative gating with a single GRU layer 144
C.3.4 Composing the operations to model a single Gated RNN layer 144
C.4 Gated RNNs are LSTMs 145
C.4.1 Representing the state update as an LSTM layer 146
C.4.2 Representing each MLP layer as an LSTM layer 146
C.4.3 Representing the multiplicative gating with an LSTM layer 147
C.4.4 Composing the operations to model a single Gated RNN layer 147
C.5 Gated Linear RNNs are Hawk/Griffin Models 147
C.5.1 Representing the state update using a recurrent block 149
C.5.2 Representing the identity function using a recurrent block 150
C.5.3 Representing each MLP layer as a gated MLP block 151
C.5.4 Representing the identify function using a gated MLP block 151
C.5.5 Representing multiplicative gating with a gated MLP block 151
C.5.6 Composing the operations to model a single gated linear-RNN layer . 152
C.6 Definitions for some helper functions in LSRL 153
C.6.1 `f_not` 153
C.6.2 `f_and` 153
C.6.3 `f_or` 154
C.6.4 `f_smaller` 154
C.6.5 `f_larger` 154
C.6.6 `f_relu_identity` 154
C.6.7 `f_modulo_counter` 155

Acknowledgements

They say it takes a village to raise a child. It turns out, it also takes a village (or a few) to make a doctor (of philosophy). Being able to invest four years to do a doctorate is a massive privilege made possible only by the immense support of others. And I have been extraordinarily fortunate to have so many people championing and guiding me along the way.

I would not have even applied for PhDs was it not for Joe Wan's unwavering encouragement and his immeasurable help during that eventful Christmas period of 2020. Jojo, thank you for being such a reliable pillar of support in so many ways ever since. I am also profoundly grateful to Gianmarco Bernasconi, Rohit Suri, Mayank Mittal and Carmen Scheidemann for keeping me sane during those trying times.

Marta and I may not have always seen eye to eye, but she gave me a chance when I thought no one would. This DPhil would not have begun without her. For that I am eternally grateful. Yet, the greatest source of joy in that first phase of my doctorate came from my labmates. I could always rely on Gabriel Santos to challenge any thought I might have, no matter how self-evident it might have appeared at first; you always beat me at debate, a feat few can achieve. While his stay with us was way too short, Jon Vadillo brought immense positivity and many fun memories! And without a sliver of doubt, I would not have survived Oxford without relying on Emanuele La Malfa for frequent venting, high-grade gossip, extra-spicy discussions and the occasional research idea.

I shall be forever indebted to my supervisors, Phil Torr and Adel Bibi, for adopting me into TVG, for their unwavering advocacy and support, and for opening so many doors. They let me chase crazy ideas, enabled me to discover my research voice, and nurtured my growth as a researcher. This thesis is a testament to the freedom, trust, and support they vested in me. It was Francisco Eiras who initially introduced me to the TVG family, later becoming my co-conspirator in the lab; we may not have saved the world, but we certainly had a lot of fun trying. I am deeply grateful to all TVG members for the wonderful conversations, collaborative efforts, spirited discussions, and good times, especially to Alasdair Paren, Cornelius Emde, Tim Franzmeyer, Tom Lamb, Botos Csaba, Dulhan Jayalath, Fabio Pizzati,

Sumeet Motwani, Ameya Prabhu, Chen Lin, Ashkan Khakzar, Francesco Pinto, Samuele Marro, Tuan Nguyen, Kalyan Ramakrishnan, Amartya Sanyal and Christian Schroeder de Witt.

Several individuals were instrumental in making this DPhil journey smooth. Wendy Poole and Katie Bourke consistently went above and beyond to help me through every administrative hurdle. I am grateful to Caroline Mawson, Kirren Mahmood, and Lionel Tarassenko for their vision in founding Reuben College and giving me a chance to shape it as part of its first-ever cohort of students. My thanks also go to Kostas Margellos and Ani Calinescu for their guidance, assistance, and encouragement throughout these four years.

England might not be a tropical paradise but a few people were my ray of sunshine, even on the murkiest of days. I am especially grateful to Philip Atkins for reliably being there for me and his readiness to hang in fun and not so fun times. If the gods play dice, then they threw only sixes when they brought me together with Jen Semler, Kaitlyn Purdie, Thea Guy, Ambre Bertrand and Tommy Escott: they quickly made Oxford feel like home and I will deeply miss the days of us living together. Joe Curtis and Max Flecha–Hirsh brought the fun in our first year, no one throws a better forest rave. So many good memories also with Lars Holdijk, Peer Nagy and Cillian Gartlan, and a special shout-out to Juuso Haavisto for being a regular lunch companion. And I am immensely thankful to Ilina, Vassya, Terry and Nic, as well as Nora, Teo and Elizabeth —the newest additions to my family in England— for reminding me that there exists life outside academia.

I am also very grateful to John Collomosse for making me feel so welcome during my internship at Adobe and for the opportunity to visit California for a few months. Shruti Agarwal's warm welcome to San Jose, coupled with all the fun activities and her mentorship, made my stay truly enjoyable. I was always grateful to rely on Vishal Asnani and Sina Alemohammad for shenanigans while our experiments ran. It was a joy to briefly live in almost the same place again with Carter Fang; thank you for always being up for an adventure and for your "Baseball 101" course! I am also profoundly thankful to Max Vladymyrov for inviting me to Seattle for a stint at Google and for the relentless fighting to set me up for success, to Andrey Zhmoginov and Chen Sun for all the fun conversations, and to Mark Sandler and Nolan Miller for their patience with my endless questions about why things do and don't work. Most of all, Reo Kawano, thank you for bearing with me, for always being so accommodating, for the countless activities, trips and fun times and, especially, for being my support and my home during my American days.

Finally, I cannot express how deeply grateful I am for the unwavering support of my parents, Vesela and Petar. They taught me the importance of education, the joy of quality work and the pride of doing the right thing. Despite the many curveballs I kept throwing at them, they always encouraged me to pursue my interests and goals and supported me along the way, even when not fully understanding my pursuits. And to my grandfather, Anton, thank you for instilling in me the engineering bug and for being the example of balancing professional, social and family life that I strive to emulate.

Chapter 1

Introduction

Since I started my DPhil four years ago, the field of machine learning has undergone dramatic changes, not least in terminology—the "AI" label once resisted by researchers is now embraced as the field entered public discourse. Problems we used to focus on, such as image classification, sentiment analysis, and speech recognition, are now largely considered solved, with entirely new challenges taking center stage. Text, image and video generation, instruction following, and zero-shot behavior, once niche areas of inquiry, have since emerged as core research priorities.

The position these systems occupy in society has fundamentally shifted, regarding both their practical utility and their presence in public consciousness. Previously, most people's AI interactions were confined to recreational applications like generating images of avocado chairs or enhancing photos with fun filters. Although machine learning has long operated behind the scenes in recommender systems, advertising, and autonomous vehicles, it functioned as a tool rather than the product itself. In contrast, today, AI is defining the zeitgeist. It underpins economic policies, dominates investment landscapes, becomes necessary for professional relevance, and is even emerging in geopolitical security discussions and trade tensions.

The notion of what safe AI is has also been turned on its head: it used to be about systems enacting harms on individuals and communities, now the focus is on humanity-level extinction events. The trend appears to be that AI will be regulated as a double-use technology with export and import controls, akin to military, aerospace and nuclear technology. And all of this happened in the span of less than three years.

The turning point was November 30, 2022, the day OpenAI released ChatGPT. ChatGPT was far from the first chatbot released (ELIZA by Weizenbaum, 1966 probably being the first, with Siri and Alexa as more recent examples), nor was it the first highly capable one, with Google's LaMDA being a notable predecessor (Thoppilan et al., 2022). However, ChatGPT

was the first powerful model that was publicly released in a way that was markedly easy to access and use. Powerful here means "useful": it was the first time most people interacted with a system that can perform some tasks more efficiently than they could themselves. It also was a general purpose tool in a way that few prior models were. Fancy sentiment analysis? Just ask it. Need a draft of a contract? Just ask it. Want something translated? Just ask it. Don't understand a step of a proof? Just ask it.

The "ChatGPT moment" quickly sent ripples throughout the research community before spreading to the rest of society. Industry immediately saw the importance and in no time a number of other proprietary models such as Gemini by Google and Claude by Anthropic were released. The open-source community quickly followed with models that quickly started to compete with the closed-source ones: LLaMA by Meta (Touvron et al., 2023), Gemma by Google (Team Gemma et al., 2024), the family of models by Mistral, and, more recently, the reasoning models by DeepSeek (Liu et al., 2024a; Guo et al., 2025).

Part of the reason the advent of general-purpose chatbots ushered in such sweeping change was their transformation of how we interact with machine learning models. Previously, solving each new task required collecting data and training a dedicated model. While fine-tuning a pretrained model offered an alternative, it still demanded task-specific data, expertise in setting up training pipelines, and access to hardware. However, large commercial (and subsequently open-source) models could reach good performance on new tasks in zero-shot fashion (requiring no training data) or through in-context learning (where training examples are provided as input, eliminating model retraining). All you needed was to acquire some API credits and send your requests in natural text, receiving responses back without having to think about GPUs, memory constraints, servers or infrastructure. This was a welcome change as the state-of-the-art models had become too large for most individuals and academic research labs to train or even fine-tune. Even if they could, it would likely be much less resource-efficient than the carefully optimized pipelines of the commercial providers.

Consequently, large segments of the research community shifted from developing task-specific datasets and architectures to devising creative strategies to coax chatbots to solve their problems. Prompting evolved into an engineering discipline, and virtually overnight, the "prompt engineer" role emerged. I may have even encountered several "Chief Prompt Engineers" on Twitter (presently X).

Throughout 2023 and 2024, a multitude of start-ups, businesses, and academic research projects were centered on prompt engineering as their core technique. The premise was that skillfully designed sophisticated prompts could enable these base models to solve formerly unthinkable tasks. Whole businesses —even some successful ones— popped up by carefully crafting system prompts and customization wrappers around the APIs of commercial chatbots.

A question lingered though: what would become of the mathematics, architectural innovations, and efforts to ensure robustness if our primary focus shifted solely to prompting

models? This prospect felt intellectually unstimulating at the time.[1] Beyond these aesthetic concerns, the prompting-centric view appeared to suggest a perpetual engine, capable of conjuring new functionalities ex nihilo. This notion fueled the discussions that preoccupied us for a while about whether large pretrained language models exhibit "emergence" of sorts—the capacity to solve problems very different from their training data. In contrast to the perceived boundless potential of prompting, those of us who still believed in classical machine learning principles held the conviction that there were inherent limits. There is no free lunch. You must be limited by your training data. You can extrapolate and compose what you learned but you cannot teach an old dog new tricks.

As a lover of impossibility results, that is what I set out to show: that there is a fundamental limit to what you can achieve by prompting a pretrained transformer. I was blissfully unaware of what a roller-coaster experience seeking this result would be and that I would eventually prove myself wrong. I had some initial success showing that prompting cannot change the relative attention pattern over the content and can only bias the outputs of an attention layer in a fixed direction. Yet, these results fell short of establishing a rigorous proof of prompting's limitation so I dug deeper and started searching for concrete counterexamples. And I failed spectacularly. It turns out prompting is, in fact, astonishingly powerful. Reality, as usual, refused to pick a side—turning out more nuanced and complex than both the fans and critics of prompting would have liked.

But how does one even begin to formally study the limits of prompting and in-context learning? The first major hurdle is formalizing precisely what we want the model to be able to express. One approach, rooted in classical deep learning theory, is to frame the question through the notion of *universal approximation*. A model architecture is considered to be a universal approximator for a class of functions $\mathcal{C}$ (called a *concept class*) if for every $\epsilon > 0$ and every $f \in \mathcal{C}$ there exist model dimensions and corresponding weights such that the model with these weights approximates f with error at most ϵ at all inputs. The set of all models that can be realized by a given architecture is called the *hypothesis class*. Formally, the question can be framed as whether the hypothesis class is *dense* in the concept class—a topic we will return to in Section 2.1. Typically, we endow the elements of the hypothesis class with some notion of *size* (like width or depth) and study the scaling properties of the architecture, specifically how decreasing the maximum error ϵ governs the model size growth. Universal approximation is the lens through which we decided to tackle the problem.

That said, the universal approximation setup, as classically defined, does not seamlessly extend to the context of in-context learning. When prompting a language model we do not modify its weights. Instead of changing the model's weights, we adjust a part of its input: the prompt. As a result, it is not clear what the appropriate hypothesis class is, or how we should define its notion of size. For instance, should we allow the model's size and parameters to depend on the target precision ϵ? While others have adopted this

[1] This feeling has evolved over the course of the work presented in this thesis. I now find myself much more convinced of the power of prompting, as well as zero-shot and in-context learning, than I once was.

approach (Wang et al., 2023), we found it offered limited insight—it effectively allows the model to memorize all possible mappings within the target precision, which is precisely the strategy used in that work. Instead, we believe that the model parameters should be fixed and independent of ϵ, so that a single model can approximate every function in the concept class, for any $\epsilon > 0$. That leaves us with just one lever to pull: the prompt. Therefore, the hypothesis class we consider is indexed by the set of all prompts. The natural scaling dimension in this setup is hence not the size of the model, but the length of the prompt. In other words, smaller ϵ would demand longer prompts, rather than larger models. We formally introduced this notion under the name "universal *in-context* approximation".

That said, what happens with the model parameters? Even if we use the same parameters for all target functions, we still have to decide what those parameters should be. We thus define an architecture to be a "universal *in-context* approximator" for a concept class $\mathcal{C}$ if there exist model parameters such that for all $f \in \mathcal{C}$ and $\epsilon > 0$, there exists a prompt such that the model from this architecture with these parameters when prefixed with this prompt will approximate f with error at most ϵ. This problem formulation is applicable to any architecture that operates over sequences. In this thesis, we will consider transformers, RNNs, LSTMs and SSMs.

We began by taking a careful look into how attention layers in transformers actually work, specifically, how much of their behavior can be changed by a prompt, as opposed to directly modifying their key, query, and value matrices. This exploration became the basis of our paper *When Do Prompting and Prefix-Tuning Work? A Theory of Capabilities and Limitations*, which appears here as Chapter 3.

To frame the problem, we compared the expressive power of different ways of feeding information into a sequence model. While most users interact with models via prompting—that is, providing a sequence of discrete tokens—it is also possible to use soft prompts (continuous embeddings passed to the first layer), or prefix-tuning, which injects continuous vectors into all layers. We compared the expressivity of these three approaches in Section 3.3 and found that prefix-tuning is strictly more expressive. This means that any limitations of prefix-tuning automatically apply to prompting as well.

Then, we showed that prefix-tuning has a severe limitation: it cannot change how an attention head distributes attention over the non-prefix elements of the sequence! The presence of a prefix or a prompt cannot change which positions the model attends the most and which the least. In fact, we showed that the only effect of a prefix is to add a fixed bias vector to the outputs of the attention layer.

So, while prompting and prefix-tuning can nudge the model's behavior, they are fundamentally limited in what they can teach it, especially when comparing to full fine-tuning or even LoRA which can reshape attention patterns in a way prompting simply cannot. Although we found that adding a bias can still help with compositional generalization, prefix-tuning nevertheless fell short on learning entirely new tasks that LoRA, with an equivalent number of parameters, could master easily. Both our theoretical analysis and

empirical results pointed in the same direction: prompting is inherently limited and you cannot expect to teach a model completely novel behaviors just by crafting a clever prompt.

While it was certainly interesting that the presence of a prompt cannot change the attention over the content tokens, this alone does not immediately rule out the possibility of transformers being universal in-context approximators. Despite remaining sceptical, the possibility lingered: could that bias, subtle as it seems, somehow be enough? After all, even a small bias at the first layer might have increasingly complex and nonlinear effects deeper in the model.

Seeking more definite proof, I set out to find a pair of functions that cannot be expressed in-context by the same transformer, regardless of how it is prompted. But after a number of frustrating dead ends, doubt crept in: what if such a pair did not exist? What if transformers in fact are universal in-context approximators?

Trying to prove the reverse conjecture led to surprisingly quick progress. If attention could act like a nearest-neighbour classifier in-context, maybe we could use that mechanism to approximate arbitrary functions. And it turns out we can, at least, if we restrict ourselves to the domain of the hypersphere. There, the dot product —which is at the center of the attention mechanism— acts like a distance metric.

With that we showed that prefix-tuning a single attention head is sufficient to approximate any smooth continuous function on the hypersphere S^m to any desired precision ϵ. We also derived bounds the prompt length as function of ϵ and the smoothness properties of the target function. Using the Kolmogorov–Arnold representation theorem, we extended this result to general sequence-to-sequence functions, achievable with transformers with depth linear in the sequence length and, importantly, independent of ϵ. That made us the first to show that transformers can indeed be universal in-context approximators—where the required model size depends on the length of the input sequences (though only linearly) but not on the desired precision ϵ or the prompt length.

This resulted in our *Prompting a Pretrained Transformer Can Be a Universal Approximator* paper presented here as Chapter 4. Importantly, this result does not contradict our earlier work. The key is that this construction does not rely on changing how the model attends across the user input, but instead, on how the user input attends to the prompt.

Of course, the world neither started nor ended with the transformer architecture. Although the transformer remained the dominant architecture in 2024, the research community was eager to discover what might come next. Among the most promising candidates were State Space Models (SSMs) —essentially Linear RNNs— which were attracting significant interest, and with good reasons. They were proposed as a response to one of the main critiques of attention: its quadratic complexity in sequence length. SSMs promised the best of both worlds: parallelizable training, like transformers, combined with the linear-time inference of RNNs. At the same time, there was considerable debate whether compressing all past information in a fixed size state would mean that SSMs are less capable than trans-

formers in processing long sequences. We were curious whether our earlier results could be extended to these new architectures and whether these perceived limitations would impact whether they can too be universal in-context approximators. But this was not a straightforward task.

Our universal in-context approximation results relied heavily on two properties: the presence of attention, and the ability to losslessly extract inputs from prompts of arbitrary length. Neither of these holds in SSMs, which are more akin to recurrent models. Therefore, we needed to get back to the drawing board, eventually culminating in our *Universal In-Context Approximation By Prompting Fully Recurrent Models* paper, presented here as Chapter 5.

The first observations was that Linear RNNs, the common core of SSMs and the various flavours of RNNs, can be decomposed into just two basic operations: linear state updates that propagate information through time, and MLPs that operate independently on each time step. Furthermore, any computation that can be expressed as a directed acyclic graph consisting of these two operations can also be realized by a Linear RNN. That, together with the complexity of manually figuring out what the model parameters should be, led me to instead develop LSRL: a programming language which compiles to model parameters for Linear RNNs. All that was left, was to write LSRL programs that act as universal in-context approximators. Writing these programs required several technical tricks but it turned out that such programs do exist. As such, Linear RNNs and their variants like LSTMs, GRUs and SSMs turned out to be universal in-context approximators as well.

We further investigated the role of multiplicative gating that many of the more advanced SSM architectures use. Our findings suggest that it plays a crucial role in ensuring the numerical stability of the conditional operations required by our programs. This helped dispel earlier doubts about whether such architectures are inherently less expressive than transformers—they are not, at least when it comes to representational capacity. In fact, our results indicate that recurrent models may even be asymptotically more efficient than transformers at universal in-context approximation.

A number of important questions remain open, and resolving them is crucial for understanding the broader consequences of architectural design choices, particularly when deciding which architecture to use in practice. For one, we had to adopt a number of simplifying assumptions. For instance, we ignored numerical precision issues: in reality the minimum approximation error we can reach is limited by the use of floating point arithmetic. This issue was particularly pronounced in our work on recurrent models. We also ignored the effect of structural components typically found in models: residual connections, normalization layers, positional encodings and various activation functions. For our transformer architecture results, we relied on the ability to scale our inputs arbitrarily in norm, something that is hindered in practical deployments by normalization.

More broadly, the results in this thesis focus on *constructing* models to realize universal in-context approximation. But in practice, models are not hand-crafted; they are *learned*. Therefore, it remains an open question to assess whether a model can learn to be universal

in-context approximator by gradient descent, and whether models pretrained on large scale internet data do in fact exhibit this property.

Despite the limitations of our analysis, we did show that in-context learning can be as expressive as training or fine-tuning a model, something that was not at all obvious when we started our work on these problems. Knowing the representational boundaries and resource requirements of different architectures provides a valuable lens for real-world applications. As training gets more expensive and inference scales to billions of users, the need for efficient and performant model at scale further stresses the need to ground our architectures in sound theory.

Chapter 2

Brief Survey of Universal (In-Context) Approximation

Before the deep learning boom, back when GPU-powered convolutional neural networks began dominating competitions (Ciregan et al., 2012; Krizhevsky et al., 2012; Simonyan and Zisserman, 2015), the question of whether neural networks were "the answer" was far more contested than it is today. Back then, deep learning was just one of many contenders in the quest toward artificial intelligence, alongside expert systems (Myers, 1986), SVMs (Cortes and Vapnik, 1995), probabilistic graphical models (Koller and Friedman, 2009) and Gaussian Processes (Rasmussen and Williams, 2006). Compared to these more principled and mathematically grounded approaches, deep learning was the least justified method. With little performance guarantees and little theoretical underpinnings why deep learning should work, many were sceptical. In fact, the community had a laundry list of concerns, from them being overparameterized, backpropagation failing at toy problems (Curry and Morgan, 1997), not being interpretable, and not being able to provide guarantees on their performance and robustness; deep learning was not without its criticism. Back in 2018, as I took my first steps into deep learning, many still regarded it as a temporary fad that would pass once people realized that you cannot just SGD your way into intelligence and believed we would ultimately get back to doing AI the proper serious way. Surely, you can put it on a self-driving car demo, but no one would really trust it on real roads, with actual people and unpredictable traffic... right?

In this sort of environment, deep learning had to prove itself, not just by winning competitions and topping benchmarks, but also by providing a theoretical justification for its existence and empirical successes. One promising angle came from demonstrating that, despite being composed of deceptively simple operations (linear transformations and element-wise non-linearities) neural networks are capable of expressing arbitrarily complex functions,

provided they have sufficient capacity. The seminal works by Cybenko (1989) and Hornik et al. (1989), later refined by Leshno et al. (1993), demonstrated that even a single hidden layer network with a non-polynomial activation function can approximate any continuous function on a compact domain to arbitrary precision. These results helped explain why neural networks could succeed across diverse domains and offered compelling mathematical assurance that the deep learning paradigm was, in fact, theoretically sound.

The times have certainly changed. Few sceptics of deep learning remain and our theoretical understanding has significantly advanced, though it is still far from where we would like it to be. While deep learning no longer needs to defend itself from external doubt, the challenges now come from within. Now that we know that deep learning works well, the question is which network architecture is "the best".

In the realm of natural language processing in particular, numerous architectures have been proposed and pitted against one another. From vanilla RNNs, to LSTMs, to the Transformer that sparked the current explosion in interest in generative models, and even a return to fundamentals with Linear RNNs and state space models (SSMs). Each new architecture sought to address the shortcomings of its predecessors: LSTMs introduced gating mechanisms to mitigate the vanishing and exploding gradients of RNNs; Transformers enabled parallel training, unlike the sequential nature of LSTMs; and Linear RNNs or SSMs tackled the quadratic inference scaling of Transformers while preserving training efficiency. And yet, each brought new issues as well.

As a result, analysing the properties of various neural architectures and understanding their respective strengths and limitations has become critical for selecting the appropriate model for a given task. Ideally, such analysis not only guides model selection but also inspires the development of new methods. Thus, the question of whether these architectures possess universal approximation capabilities, a necessary though not a sufficient condition for their utility, has resurfaced.

Once models (predominantly transformer-based ones) were scaled up sufficiently, they began to exhibit few-shot (Brown et al., 2020) and even zero-shot (Wei et al., 2021) capabilities. With that, a whole new mode of learning came to be: in-context learning. Rather than optimizing model parameters either through training from scratch or fine-tuning a pretrained model, in-context learning leaves the parameters fixed and instead modifies the model's input. In-context learning, with prompting as its most popular form, proved to be extremely successful, especially considering that it required no parameter updates.

And yet there has been disagreement as to how much one can push themselves away from the original base model. If the model had never seen some sort of data or a task, a skill or knowledge, would it be able to learn it by just seeing a description and/or examples in-context? There has been empirical evidence in both directions. Bubeck et al. (2023) had a comprehensive study of various abilities of GPT-4 (Achiam et al., 2023) and came to the conclusion that just prompting that model often leads to close to human performance, though scepticism quickly followed (Marcus, 2023). Tanzer et al. (2024) showed that large commercial models get close to human performance when translating a low-resource lan-

guage (Kalamang) that has only a single grammar book (Visser, 2022) parts of which were provided in-context. However, it was later shown that the model does not make an effective use of the grammar knowledge but rather leverages the parallel examples (Aycock et al., 2025). While counting only as anecdotal evidence, many of us have been surprised when a chatbot has managed to solve a problem that we are certain no one had ever looked at before. Still, many of the claims of emergent skills of the largest models were disputed as possible data contamination issues (Martínez, 2024; Rogers and Luccioni, 2024; Sainz et al., 2024; Udandarao et al., 2024; Elazar et al., 2024) or due to improper evaluation (Schaeffer et al., 2023; Lu et al., 2024). Therefore, understanding the limits of what these state of the art models can do when prompted appropriately remained an open question. Our efforts in studying this open question culminated in the present thesis.

This chapter offers a short overview of universal approximation results for MLPs, the first architecture that was extensively studied, in Section 2.1. This is followed by reviewing the results for sequence-to-sequence models that are more relevant to language models in Section 2.2. Then, Section 2.3 reviews other works on universal *in-context* approximation, the area of the contributions of this thesis. Finally, in Section 2.4, we briefly summarize some works comparing the expressivity of in-context learning with various formal languages, as an alternative way to study the power of prompting.

2.1 Universal approximation results for the MLP

Why do we care about universal approximation of neural networks in the first place? The key motivation is that it is a necessary condition for deep learning to work. Universal approximation properties of a given architecture do not imply that a training pipeline will find a good model efficiently. However, a lack of universal approximation properties makes it impossible to approximate certain functions, making it unlikely that said architecture would work well in practice.

Let $\mathcal{X}$ and $\mathcal{Y}$ be normed vector spaces. We consider a family of *target functions* which is a subset $\mathcal{C}$ of all mappings $\mathcal{X} \to \mathcal{Y}$, i.e., $\mathcal{C} \subseteq \mathcal{Y}^{\mathcal{X}}$, with $\mathcal{C}$ is often referred to as a *concept space*. These are the relationships we wish to learn by some simpler candidate functions. Let us denote this set of candidates by $\mathcal{H} \subset \mathcal{Y}^{\mathcal{X}}$, called *hypothesis space*. The problem of universal approximation is concerned with how well functions in $\mathcal{H}$ approximate functions in $\mathcal{C}$. There are two main ways to measure how well functions in $\mathcal{H}$ represent functions in $\mathcal{C}$: density results and approximation rate results (Carothers, 1998). Density results show that, given an ϵ, one can find a hypothesis $h \in \mathcal{H}$ approximating any $f \in \mathcal{C}$ with error at most ϵ. Approximation rate results, also called *Jackson-type* results, are stronger as they offer a measure of complexity for h to reach a desired precision ϵ. Classically, a Jackson-type result would provide a minimum width or depth necessary for a neural network to reach a desired precision ϵ. In the context of the present thesis, the notion of complexity that we care about is the length N of the prefix or prompt.

Formally, we can define these two notions of universal approximation as follows:

Definition 2.1 (Universal Approximation (Density-Type))**.** We say that $\mathcal{H}$ is a universal approximator for $\mathcal{C}$ over a compact set $S \subseteq \mathcal{X}$ if for every $f \in \mathcal{C}$ and every $\epsilon > 0$ there exists an $h \in \mathcal{H}$ such that $\sup_{x \in S} \|f(x) - h(x)\| \leq \epsilon$. One typically says that *$\mathcal{H}$ is dense in $\mathcal{C}$*.

Definition 2.2 (Approximation Rate (Jackson-Type))**.** Fix a hypothesis space $\mathcal{H}$. Let $\{\mathcal{H}^N : N \in \mathbb{N}_+\}$ be a collection of subsets of $\mathcal{H}$ such that $\mathcal{H}^N \subset \mathcal{H}^{N+1}$ and $\bigcup_{N \in \mathbb{N}_+} \mathcal{H}^N = \mathcal{H}$. Here, N is a measure of the complexity of the approximation candidates, and $\mathcal{H}^N$ is the subset of hypotheses with complexity at most N. Then, the approximation rate estimate for $f \in \mathcal{C}$ over a compact $S \subseteq \mathcal{X}$ is a bound $Z_{\mathcal{H}}$:

$$N \geq Z_{\mathcal{H}}(f, \epsilon) \implies \inf_{h \in \mathcal{H}^N} \sup_{x \in S} \|f(x) - h(x)\| \leq \epsilon.$$

$Z_{\mathcal{H}}$ gives an upper bound to the minimum hypothesis complexity necessary to reach the target precision ϵ and typically depends on the smoothness of f.

In the late 80s and early 90s, the density question (Definition 2.1) for $\mathcal{C}$ being a set of univariate functions with various continuity properties was extensively studied for the single hidden layer perceptron (Gallant and White, 1988; Irie and Miyake, 1988; Carroll and Dickinson, 1989; Cybenko, 1989; Funahashi, 1989; Hornik et al., 1989). The single layer perceptron with m inputs and N hidden neurons corresponds to:

$$\mathcal{H}^N_{P,d_{\text{in}}} = \left\{ \boldsymbol{x} \mapsto \sum_{i=1}^{N} c_i \sigma(\boldsymbol{w}_i^\top \boldsymbol{x} - b_i) \;\middle|\; \text{for } c_i, b_i \in \mathbb{R}, \boldsymbol{w}_i \in \mathbb{R}^{d_{\text{in}}} \right\}, \tag{2.1}$$

with σ being some sigmoidal function, i.e., $\lim_{t \to -\infty} \sigma(t) = 0$ and $\lim_{t \to \infty} \sigma(t) = 1$. Naturally, the hypothesis class of all perceptrons with d_{in} inputs is $\mathcal{H}_{P,d_{\text{in}}} = \bigcup_{N \in \mathbb{N}_+} \mathcal{H}^N_{P,d_{\text{in}}}$. This line of research culminated in 1993 with results by Leshno et al. (1993) and Hornik (1993) showing that the activation function not being a polynomial is a necessary and sufficient condition of universal approximation:

Theorem 2.3 (Non-polynomial activation is necessary and sufficient)**.** *Let σ be an activation function. Then, the concept class of single hidden layer MLPs ($\mathcal{H}_{P,m}$) is dense in the compact continuous functions over $\mathbb{R}^n$, if and only if σ is not a polynomial.*

Seeing that if the activation is polynomial density fails is easy: linear combinations of polynomials of degree n is of degree n and will always have irreducible error when approximating polynomials of a higher degree. Approximation rate results (Jackson-Type bounds, Definition 2.2) have also been extensively studied under different conditions, establishing the approximation error ϵ for N hidden layer parameters at $\mathcal{O}(\epsilon^{-d_{\text{in}}})$ (DeVore et al., 1989; Mhaskar, 1996; Maiorov and Pinkus, 1999).

With the advent of *deep* neural networks in the 2010s, the question arose whether increasing the depth offers something more than a shallow network. Telgarsky (2015) showed that depth does indeed help for parameter-efficient learning, namely that there exist functions

that require exponential number of parameters if approximated with a shallow network but a linear number of parameters for a deep network. In a follow-up, Telgarsky (2016) also showed that the benefits of depth never stop, i.e., one can always construct a function that a deeper network can approximate with constant layer width which a less deep network would require an exponential width. Yarotsky (2017) showed that $f(x) = x^2$ on the segment $[0, 1]$ can be approximated by a ReLU network with depth and number of parameters $\mathcal{O}(\ln(1/\epsilon))$, when a shallow model would require $\mathcal{O}(1/\epsilon)$ parameters. This is especially important, because x^2 is a good way to think about the ability of a model to approximate multiplication of its inputs (via $xy = 1/2((x+y)^2 - x^2 - y^2)$), something that will be relevant when considering the benefits of multiplicative gating in SSMs in Chapter 5. Pushing the depth-width trade-off to its limit, Park et al. (2021) showed that universal approximation is possible with a deep network of width just $\max\{d_{\text{in}} + 1, d_{\text{out}}\}$, with d_{out} being the output dimension of the target function. Overall, the question of universal approximation of MLPs has been extensively studied for various concept classes and scaling dimensions. For further reading on the classical results, the lecture notes by Telgarsky (2021) and the survey by Pinkus (1999) are highly recommended.

2.2 Universal approximation results for sequence-to-sequence architectures

As neural networks progressed beyond simple toy problems to tackle various domains, the community developed an increasingly diverse range of concept classes and model architectures. For example, CNNs (Zhou, 2020) and more generally sparse models (Bölcskei et al., 2019) were shown to be universal approximators under certain conditions. However, within the context of large language models, we are primarily concerned with sequence-to-sequence models.

Recurrent Neural Networks (RNNs), one of the first architectures proposed for sequence modelling can be thought of as a discrete-time dynamical system:

$$\begin{aligned}
\boldsymbol{h}_{t+1} &= \sigma(\boldsymbol{A}\boldsymbol{h}_t + \boldsymbol{B}\boldsymbol{x}_t + \boldsymbol{b}), \\
\mathcal{H}_{\text{RNN}}^{d_{\text{hid}}} &= \left\{\boldsymbol{x}, t \mapsto \boldsymbol{C}\boldsymbol{h}_t \;\middle|\; \boldsymbol{A} \in \mathbb{R}^{d_{\text{hid}} \times d_{\text{hid}}}, \boldsymbol{B} \in \mathbb{R}^{d_{\text{hid}} \times d_{\text{in}}}, \boldsymbol{b} \in \mathbb{R}^{d_{\text{hid}}}, \boldsymbol{C} \in \mathbb{R}^{d_{\text{out}} \times d_{\text{hid}}}\right\}, \\
\mathcal{H}_{\text{RNN}} &= \bigcup_{d_{\text{hid}} \in \mathbb{N}_+} \mathcal{H}_{\text{RNN}}^{d_{\text{hid}}}.
\end{aligned} \tag{2.2}$$

Deeper versions of RNNs can also be constructed by stacking such layers one after the other.

While RNNs were the sequence model of choice for a while, the non-linearity in the state update can make them difficult to train due to vanishing and exploding gradients (Bengio et al., 1994). To address these issues, various techniques to introduce residual-like connections to stabilise the training were proposed. The most notable models that were proposed are Gated Recurrent Units (GRUs, Cho et al. 2014) and Long Short-Term Memory models (LSTMs, Hochreiter and Schmidhuber 1997). While LSTMs remained the dominant

flavour of RNNs for a while, they require sequential training which renders them slow and difficult to train at scale.

As a result, the transformer architecture (Vaswani et al., 2017), built around the attention mechanism (Bahdanau et al., 2015), quickly picked up interest due its capacity for parallel sequence processing during training. This parallelization enabled straightforward scaling, ultimately producing the large models that demonstrated significant utility. The attention mechanism computes keys, queries and value projections of all inputs $\boldsymbol{x}_1, \dots, \boldsymbol{x}_N$ by multiplying them with corresponding $\boldsymbol{W}_K$, $\boldsymbol{W}_Q$ and $\boldsymbol{W}_V$ matrices. To calculate how position i mixes information across the inputs $j = 1, \dots, N$, attention weights α^i_j are computed as the softmax-normalized dot products $\boldsymbol{x}_i^\top \boldsymbol{W}_Q^\top \boldsymbol{W}_K \boldsymbol{x}_j$. Then the attention layer output for element i is simply the attention-weighted linear combination of the value projections: $\alpha^i_j \boldsymbol{W}_V \boldsymbol{x}_j$. A transformer typically then applies an MLP independently to each element before feeding it to the next attention layer. Residual connections (Srivastava et al., 2015; He et al., 2016) and normalization (Ba et al., 2016; Zhang and Sennrich, 2019) are also usually used.

Due to the cost of computing the attention weights α^i_j, which scales quadratically in the sequence length, the inference of transformers, unlike RNNs, does not scale well with longer sequences. As an attempt to alleviate the quadratic inference cost, Linear RNNs (also referred to as State Space Models, SSMs), which drop the non-linearity from the state update in Equation (2.2) were proposed:

$$\begin{aligned} \boldsymbol{h}_t &= \boldsymbol{A}\boldsymbol{h}_{t-1} + \boldsymbol{B}\boldsymbol{x}_t + \boldsymbol{b}, \\ \boldsymbol{y}_t &= \text{MLP}(\boldsymbol{h}_t). \end{aligned} \tag{2.3}$$

As they admit a convolutional representation, they can be trained in parallel like a transformer (Gu et al., 2021; Fu et al., 2023a), while maintaining the linear complexity inference of an RNN. The fully linear state updates do not affect the expressivity of the models, as non-linear activations are nevertheless present in the MLPs between the linear state update layers (Boyd and Chua, 1985; Wang and Xue, 2023). The state-of-the-art Linear RNN models also utilise some form of multiplicative gating (Gu and Dao, 2023; De et al., 2024; Botev et al., 2024).

When considering sequence-to-sequence models, two types of concepts classes naturally arise. First, there is the class of functions defined over inputs of fixed length over some input domain, either vocabulary (finite discrete tokens) or continuous vectors. This makes more sense when studying language as we are generally interested in language models that eventually terminate, i.e., produce an end-of-sequence (`EOS`) token (Du et al., 2022).

An alternative approach, more aligned with the control theory and dynamical systems literature, is to look at sequences of unbounded length. Processing general unbounded sequences demands correspondingly unbounded model memory capacity, an impractical requirement for non-attention architectures like RNNs, LSTMs, GRUs, and SSMs, as their state dimensions would necessarily grow with sequence length. Therefore, some memory decay property of the concept class is typically considered, i.e., that inputs far in the past have

minimal impact on the current values (Boyd and Chua, 1985; Hanson and Raginsky, 2020; Wang and Xue, 2023). While such decay might be sensible for modelling physical systems, it aligns poorly with how we use LLMs in practice: instructions provided in the beginning of the prompt are typically relevant throughout its processing. While transformers, in principle, can have unbounded memory due to being able to attend to all prior keys, this also has its limits in practice (Liu et al., 2024b; Barbero et al., 2024). Accordingly, given their better alignment with the objectives of language modelling, the present thesis focuses exclusively on results for the first kind: bounded input lengths.

Among the earliest investigations of sequence-to-sequence models was the analysis of RNNs with Schäfer and Zimmermann (2007) showing that they can universally approximate dynamical systems over bounded number of temporal steps. Consequently, they can universally approximate general continuous sequence-to-sequence functions as such functions can be expressed as dynamical systems via the Kolmogorov–Arnold theorem (Hamdouche and Sala, 2021). For the deep but narrow regime of RNNs, corresponding more closely to the architectures used in practice, Song et al. (2023) show that width of $d_{\text{in}}+d_{\text{out}}+3$ and ReLU activations are sufficient to universally approximate continuous sequence-to-sequence functions.

The universal approximation abilities of transformer were also studied. Yun et al. (2020) pioneered this direction with proving universality by embedding the input sequences with *contextual maps*: each input element is mapped to a unique scalar that depends on the entire sequence. Once that is done, an element-wise application of an MLP can transform these contextualized embeddings into the target outputs. The contextual map embedding requires depth $\mathcal{O}(\delta^{-d_{\text{in}}})$ that is exponential in the input dimension and grows as the function becomes less smooth or the target precision increases (which leads to a smaller discretization step δ). They only offer density results which are not particularly constructive but Jiang and Li (2024) provide Jackson bounds instead. Furuya et al. (2025) have also studied representing contextual maps with transformers. However, it has also been shown that δ might scale poorly even for smooth functions and that transformers might struggle to learn universal approximation abilities in practice (Nath et al., 2024).

SSMs (Linear RNNs) have similarly been examined from the perspective of universal approximation. Wang and Xue (2023) and Orvieto et al. (2024) showed that the lack of non-linearities on the state update —something that, intuitively, should be very limited in expressivity— deep state-space models with nonlinear activation *between layers* can in fact approximate continuous sequence-to-sequence relationships. Nishikawa and Suzuki (2024) demonstrated that SSMs with gated convolution can achieve similar approximation error rates like the transformer architecture for concept classes with specific smoothness properties. Meanwhile, Cirone et al. (2024) studied SSMs with more general gating operations and show that they can be more expressive than the purely linear ones.

Universal approximation
Changing the model parameters
to approximate any target function

For every function f and precision $\varepsilon > 0$ there are model parameters θ

$$\left|\text{model}_\theta(x) - f(x)\right| \leq \epsilon$$
$$\forall x \in X, \epsilon > 0$$

Universal *in-context* approximation
Keeping the model parameters fixed
and changing only a part of the input (the prompt)

A single set of model parameters θ for all functions f and precisions $\varepsilon > 0$

$$\left|\text{model}_\theta([\text{prompt}(f), x]) - f(x)\right| \leq \epsilon$$
$$\forall x \in X, \epsilon > 0, \;\; \forall f \in \mathcal{C}$$

Figure 2.1: **Comparing universal approximation and universal in-context approximation.** In universal approximation, the model parameters are indexed by the target function. In contrast, in universal *in-context* approximation they are fixed and the prompt is used to index the target function; the model parameters are universal for all functions in the concept class.

2.3 Universal in-context approximation results

As previously established, large pretrained language models demonstrate zero-shot and in-context learning capabilities. These abilities, combined with the continuously increasing scale of state-of-the-art models and the growing challenges of running them on typical hardware, have established prompting and few-shot learning as the dominant paradigms for specializing deep learning models for specific tasks. This poses a very different regime than what the classic approximation theory results study.

In the traditional setting, the hypothesis class $\mathcal{H}$ is indexed through the model parameters. In other words, the goal is to show that there exist model parameters for which the model approximates the target function. Conversely, in this new prompting and in-context learning settings the model parameters are fixed. Instead, we can only control the prefix of the input, i.e., the prompt.

One typically specifies the desired behavior by choosing the first inputs (tokens) to specify the target mapping from the following input tokens (queries) to the output tokens. For example, in the input *"Translate English to Bulgarian: Neural networks are weird!"*, the expression *"Translate English to Bulgarian"* would be the prompt while *"Neural networks are weird!"* would be the query. Therefore, our hypothesis class is associated with fixed model parameters and is indexed by *prompts*, which we can vary to approximate functions from the concept class. Note that in this setting the choice of prompt will depend on the target function and the target precision, however the model parameters are universal and fixed *for all functions in the concept class*. We can, however, specify what the model parameters should be.

Studying scaling in the in-context setting is also different than in the traditional setting. Whereas classical approaches require scaling architectural dimensions —typically

layer width (for MLPs), hidden dimensionality (for RNNs and attention mechanisms), or model depth— the in-context setting precludes such modifications due to the immutability of the parameters. Consequently, the sole mechanism for enhancing approximation quality through increased parameter allocation involves increasing the prompt length, intuitively accomplished by supplying additional demonstrations or more detailed function specifications. This mimics practice: more detailed prompts result in more specific and closely controlled behavior. The proof strategies for demonstrating universal *in-context* approximation are thus also very different than in the classic universal approximation setup. See Figure 2.1 for a summary of the differences between the two settings.

For the transformer architecture, Wang et al. (2023) were the first to pose the question of universal in-context approximation. Their proof technique relied on the observation that every prompt defines a function from the query to the output and hence the transformer needs to approximate the higher-order function `prompt` → (`query` → `output`). By applying defunctionalization (also known as uncurrying) one can view this higher-order function as the first order function with signature (`prompt` × `query`) → `output`. With that observation, one can directly apply the results by Yun et al. (2020), as discussed in Section 2.2, to show universal approximation for the uncurried function and thus universal in-context approximation. Although conceptually elegant, this approach suffers from several limitations. First, it necessitates the discretization and enumeration of all possible functions for in-context approximation, which is infeasible. Moreover, the discretization must be performed prior to determining the model parameters and locks the approximation quality in at a given precision level. As a consequence, the model parameters are not truly fixed; rather, they depend on the precision. Furthermore, as they resort to (Yun et al., 2020), the depth of the model also needs to scale with the target precision. Hence, the model parameters cannot be considered fixed. Consequently, this construction fails to satisfy the criteria we established for universal in-context approximation. Furthermore, they only offer density results and lack explicit constructions.

In contrast, we demonstrated that universal in-context approximation can be achieved far more efficiently: with fixed model parameters, a model depth that grows linearly with input length, and precision improvements attained simply by extending the prompt length, without altering the model architecture or parameters (Petrov et al., 2024b). This makes our work the first to show that a *fixed* transformer model can be a universal in-context approximator. Our proofs are based on the observation that, due to the dot product in the attention mechanism, attention heads are especially suited to approximate smooth continuous functions on the hypersphere to any desired precision. Our results are constructive and we provide explicit weights for the model parameters and for the formatting of the prompt and query. Furthermore, we offer Jackson bounds: given smoothness properties of the target function and the desired precision, we offer an upper bound to the necessary prompt length.

Li et al. (2024) study a more constrained setting, focusing on in-context learning of nearest neighbours with transformers. They further explore the learning dynamics, demonstrat-

ing that such models are trainable using standard gradient descent. Their construction resembles ours, thus it is likely that their learning dynamics results also translate to our setting.

Kratsios and Furuya (2025) demonstrate that MLPs are also capable of in-context learning, although their depth scales linearly with the hidden dimension and exponentially with the context size, potentially relying on memorization of all possible mappings, much like the construction by Wang et al. (2023). In contrast, our depth is constant in the dimension size, and linear in the context length and does not rely on memorization. Kratsios and Furuya (2025) also employ a model size that is dependent on the target precision, thus it does not satisfy the notion of a single, frozen architecture. That is not surprising as an MLP has a fixed input size. Nevertheless, it is still interesting that even an MLP can use part of its input to completely determine how the rest of its input should be processed.

As far as RNNs, their various flavours such as GRUs and LSTMs, and their linear counterparts are concerned, to the best of our knowledge our work (Petrov et al., 2024a) is the only to study their universal in-context approximation abilities. We showed that all these recurrent architectures can also be universal in-context approximators despite lacking attention mechanisms. The prompt lengths needed are $\mathcal{O}(\epsilon^{-d_{\text{in}}})$, which is as good as one may hope for. Still, a number of open questions remain, especially regarding the role of multiplicative gating.

2.4 In-context learning and formal languages

In addition to their universal approximation characteristics described in the preceding sections, autoregressive models provide an alternative framework for analysis. Because of their sequential behavior, these models resemble computing machines and may be studied through the lens of computational complexity theory. This insight has not gone unnoticed; in recent years, there has been extensive contributions on identifying the formal languages that various transformer configurations are capable of expressing.[1] Whereas universal approximation typically investigates an architecture's ability to approximate continuous functions over finite sequences in a single forward pass, the study of formal language recognition and generation centers on potentially infinite sequences composed of discrete vocabulary elements produced in an autoregressive manner. We present a concise summary of key results in this area, even though they are orthogonal to the primary scope of this thesis. For a more comprehensive review, we recommend (Strobl et al., 2024).

There are several different frameworks of formal models through which the computational abilities of transformers have been studied. Finite automata and Turing machines are a

[1]The interest in autoregressive models expressing formal languages neither started nor ended with the transformer architecture. RNNs with a specific activation function were shown to be Turing complete by Siegelmann and Sontag (1995) and further studied by Svete et al. (2024) and Nowak et al. (2023). SSMs, as their linear variant, have also been recently studied in the context of formal languages (Sarrof et al., 2024; Terzic et al., 2025). Still, most interest has been in transformers, likely due to their dominant role in practice.

popular choice. These models are closely linked to the Chomsky hierarchy, which classifies languages as regular, context-free, context-sensitive, and recursively enumerable. Finite-state automata are capable of recognizing regular languages. Non-deterministic pushdown automata recognize context-free languages. Linear-bounded non-deterministic Turing machines are associated with context-sensitive languages. At the top of the hierarchy, Turing machines recognize recursively enumerable languages. Bhattamishra et al. (2020), Pérez et al. (2021) and Merrill and Sabharwal (2024) have shown that transformers, operating in the autoregressive regime, can recognize recursively enumerable languages.

An alternative perspective comes from circuit complexity. A circuit C is a directed acyclic graphs with n binary inputs, where each vertex corresponding to a logic gate (e.g., `NOT`, `AND`, `OR`) and with the circuit producing a single binary output. To study them across different input lengths one considers *circuit families*, where each input length n corresponds to its own circuit C_n, with certain regularities as to the computational cost of constructing C_n given its input length n. Circuits have mostly been used to study lower and upper bounds of transformers in the feed-forward-only regime (no autoregression), e.g., by Hahn (2020), Hao et al. (2022), Strobl (2023) and Chiang et al. (2023). These multiple perspectives on formal languages, along with their intersections, produce a diverse array of capabilities against which different versions of the transformer architecture can be assessed. For a diagrammatic overview of these connections, refer to Figure 1 in Strobl et al. (2024).

Unsurprisingly, in the absence of intermediate steps such as chain-of-thought reasoning or autoregressive sampling, the expressive power of the model is constrained by its size (Hao et al., 2022; Merrill and Sabharwal, 2023b). However, when autoregressivity is allowed, the situation changes significantly, especially with respect to Turing completeness and the ability to recognize any recursively enumerable language. Several results support this. For instance, Pérez et al. (2021) focuses on encoder-decoder architecture where the input is provided to the encoder while the decoder simulates the Turing machine. They show that a $T(n)$-time-bounded Turing machine can be simulated by a transformer using $\mathcal{O}(\log T(n))$ precision and $\mathcal{O}(T(n))$ intermediate steps. A similar result was shown by Bhattamishra et al. (2020) via reduction to an RNN and form (Siegelmann and Sontag, 1995). Given that state-of-the-art transformers primarily rely on decoder-only designs, it is more practical to focus on these when studying computational power. To this end, Merrill and Sabharwal (2024) show that a transformer decoder with $\mathcal{O}(\log(n + T(n)))$ precision and $\mathcal{O}(T(n))$ intermediate steps can simulate a Turing machine for $T(n)$ steps. Looking more generally at language models as distributions over strings from an alphabet, corresponding to the autoregressive sampling procedure from a transformer architecture, Nowak et al. (2024) show that language models (and transformers) can represent the same distributions over strings as probabilistic Turing machines.

The above-mentioned works aiming to show equivalence between transformers and formal models rely heavily on the models acting in a discrete fashion, i.e., being able to accurately select activations at specific previous locations, akin to a computer reading an address in memory. To enable this behavior, these works often replace the standard softmax

activation in the attention mechanism with various hard-max alternatives. While at first approximating the softmax with a hardmax might appear to be just a negligible technical assumption, it yields models with substantially different properties from those of transformers used in practice. This divergence arises from the softmax function's inability to enforce hard selection, particularly when the context length extends beyond the sequences seen during training (Veličković et al., 2024). In other words, softmax is bound to disperse attention for long enough sequences, making it impossible to approximate hardmax or even simple functions like mean pooling (Petrov et al., 2025c). Further complications, plaguing also the classic universal approximation results, stem from the failure of attention in the constant precision regime and the corresponding need for at least $\mathcal{O}(\log n)$ (n being the sequence length) precision (Merrill and Sabharwal, 2023a). The vast array of computational classes under comparison, the many architectural variants of the transformer (feed-forward vs autoregressive, encoder-decoder vs decoder-only, various choices for positional embeddings), and the simplifying assumptions employed, all contribute to the difficulty of drawing concrete conclusions about what transformers can or cannot achieve in practice.

Chapter 3

When Do Prompting and Prefix-Tuning Work? A Theory of Capabilities and Limitations

Abstract

Context-based fine-tuning methods, including prompting, in-context learning, soft prompting (also known as prompt tuning), and prefix-tuning, have gained popularity due to their ability to often match the performance of full fine-tuning with a fraction of the parameters. Despite their empirical successes, there is little theoretical understanding of how these techniques influence the internal computation of the model and their expressiveness limitations. We show that despite the continuous embedding space being more expressive than the discrete token space, soft-prompting and prefix-tuning are potentially less expressive than full fine-tuning, even with the same number of learnable parameters. Concretely, context-based fine-tuning cannot change the relative attention pattern over the content and can only bias the outputs of an attention layer in a fixed direction. This suggests that while techniques like prompting, in-context learning, soft prompting, and prefix-tuning can effectively elicit skills present in the pretrained model, they may not be able to learn novel tasks that require new attention patterns.

3.1 Introduction

Language model advances are largely driven by larger models and more training data (Kaplan et al., 2020; Rae et al., 2021). Training cutting-edge models is out of reach for most academic researchers, small enterprises, and individuals, and it has become common to

instead fine-tune open-source pretrained models (Devlin et al., 2019; Min et al., 2021). Yet, due to escalating computational demands, even fine-tuning of the larger models has become prohibitively expensive (Lialin et al., 2023).

As a result, there is an acute need for more efficient fine-tuning methods, either by sparsely modifying the parameters of the model or modifying its input context. Examples of the first type include adapter modules which introduce a few trainable layers to modify the behaviour of the frozen pretrained network (Rebuffi et al., 2017; Houlsby et al., 2019; Hu et al., 2023). One can also use low-rank updates, which also results in a reduced number of trainable parameters (Hu et al., 2021).

Context-based fine-tuning has been motivated by the success of few-shot and zero-shot learning (Wei et al., 2021; Kojima et al., 2022). The most popular context-based approach is prompting, where generation is conditioned on either human-crafted or automatically optimized tokens (Shin et al., 2020; Liu et al., 2023). In-context learning —prompting via providing input-label pairs— is another widely used technique (Brown et al., 2020). Given the challenges of discrete optimization over tokens, there is growing interest in methods that optimize real-valued embeddings (Lester et al., 2021). It is widely believed that these *soft prompts* offer greater expressiveness due to the expansive nature of continuous space. Furthermore, beyond only optimizing input embeddings, one can optimize the inputs of every attention layer (Li and Liang, 2021). This technique, *prefix-tuning*, has proven to be very successful and competitive to full fine-tuning (Liu et al., 2022).

While context-based fine-tuning approaches have witnessed impressive empirical successes and widespread adoption, we have little theoretical understanding of how they work. In this work, we analyse the influence of prompts and prefixes on the internal computations of a pretrained model and delineate their limitations. Specifically, we address the following questions:

1. **Soft prompting and prefix-tuning are motivated by the embedding space being larger than the token space. However, can a transformer utilize the additional capacity?** We show that with a careful choice of transformer weights, controlling a single embedding can generate any of the V^N completions of N tokens, while controlling a token can produce only V completions, with V being the vocabulary size. Thus, a transformer can indeed exploit the embedding space.

2. **Since prefix-tuning is more expressive than prompting, is it as expressive as full fine-tuning?** Despite the expressiveness of continuous space, prefix-tuning has structural limitations. A prefix cannot change the relative attention over the content tokens and can only bias the output of the attention block in a constant direction. In contrast, full fine-tuning can learn new attention patterns and arbitrarily modify attention block outputs, making it strictly more powerful.

3. **If context-based fine-tuning methods suffer from such structural limitations, how come they have high empirical performance?** We show that the prefix-induced bias can steer

the model towards a pretraining task. Prefix-tuning can also combine skills picked up during pretraining to solve some new tasks similar to pretraining tasks. However, it may not learn a completely new task. This is not simply because of the small number of learnable parameters: fine-tuning the same number of parameters can be sufficient to learn the novel task. Hence, context-based fine-tuning can elicit or combine pretrained model skills but cannot learn completely new behaviors.

3.2 Background

3.2.1 The transformer architecture

We outline a simplified decoder-only transformer architecture (Vaswani et al., 2017). Assume that the model has vocabulary size V (also referred to as *number of tokens*). The input is a sequence $(\mathtt{x}_1, \ldots, \mathtt{x}_p)$, $\mathtt{x}_i \in \{1, \ldots, V\}$, $\forall i$. Each token is mapped to a d_e-dimensional vector that is the $\mathtt{x}_i$-th column of an embedding matrix $\boldsymbol{E} \in \mathbb{R}^{d_e \times V}$. The attention mechanism is position-invariant, so typically position encodings are added. For a model with maximum input length N (*context size*), we use a one-hot position encoding $\boldsymbol{e}_N(i)$ concatenated with the embedding. Therefore, the embedding for the i-th position provided to the first attention block would be $\boldsymbol{x}_i = [\boldsymbol{E}_{:,\mathtt{x}_i}^\top, \boldsymbol{e}_N^\top(i)]^\top$.

A transformer consists of alternating attention blocks which operate across the whole sequence and Multi-Layer Perceptrons (MLPs) operating on each individual element. Each attention block consists of H heads. Each head h is parameterized by query, key, and value matrices $\boldsymbol{W}_Q^h, \boldsymbol{W}_K^h \in \mathbb{R}^{k \times d_{\text{in}}}, \boldsymbol{W}_V^h \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$.[1] The attention matrix $\boldsymbol{A}^h \in \mathbb{R}^{p \times p}$ for head h then has elements

$$\boldsymbol{A}_{ij}^h = \frac{\exp\left({}^{T}\!/\!\sqrt{k}(\boldsymbol{W}_Q^h \boldsymbol{x}_i)^\top (\boldsymbol{W}_K^h \boldsymbol{x}_j)\right)}{\sum_{r=1}^p \exp\left({}^{T}\!/\!\sqrt{k}(\boldsymbol{W}_Q^h \boldsymbol{x}_i)^\top (\boldsymbol{W}_K^h \boldsymbol{x}_r)\right)}, \tag{3.1}$$

where $p \leq N$ is the current length of the input and $T > 0$ is an inverse temperature parameter.[2] Equation (3.1) is the softmax function, hence with high enough T, it will result in approximately one-hot encoding of the maximum j. The output of the attention block $\mathcal{A}$ with H heads is then $(\boldsymbol{t}_1, \ldots, \boldsymbol{t}_p)$, where each position i is the sum of the attention-weighted values across all heads:

$$\mathcal{A}[(\boldsymbol{W}_Q^1, \ldots, \boldsymbol{W}_Q^H), (\boldsymbol{W}_K^1, \ldots, \boldsymbol{W}_K^H), (\boldsymbol{W}_V^1, \ldots, \boldsymbol{W}_V^H)](\boldsymbol{x}_1, \ldots, \boldsymbol{x}_p) = (\boldsymbol{t}_1, \ldots, \boldsymbol{t}_p),$$
$$\boldsymbol{t}_i = \textstyle\sum_{h=1}^H \sum_{j=1}^p \boldsymbol{A}_{ij}^h \boldsymbol{W}_V^h \boldsymbol{x}_j. \tag{3.2}$$

A transformer then applies an MLP to each output of an attention block before passing them to next attention block. We will consider linear layers $\mathcal{L}[\boldsymbol{M}, \boldsymbol{b}](\boldsymbol{x}) = \boldsymbol{M}\boldsymbol{x} + \boldsymbol{b}$ and ReLU-activated linear layers $\hat{\mathcal{L}}[\boldsymbol{M}, \boldsymbol{b}](\boldsymbol{x}) = \text{ReLU}(\boldsymbol{M}\boldsymbol{x} + \boldsymbol{b})$. When we compose attention

[1] For the first block, d_{in} must be $d_e + N$ but may be different for the deeper blocks.

[2] A causal model has $A_{ij} = 0$ for $j > i$. This does not affect our results so we will skip the masking step.

blocks and linear or softmax layers, we will implicitly assume that the linear layer is applied to all positions of the sequence. Furthermore, we will use the *then* operator $⨾$ for left-to-right function composition. Therefore, a transformer model predicting confidences over the vocabulary can, for example, be represented as:

$$(\boldsymbol{y}_1, \ldots, \boldsymbol{y}_p) = \left(\mathcal{A}_1 ⨾ \hat{\mathcal{L}}_{1,1} ⨾ \mathcal{L}_{1,2} ⨾ \mathcal{A}_2 ⨾ \hat{\mathcal{L}}_{2,1} ⨾ \mathcal{L}_{2,2} ⨾ \text{softmax}\right) \left(\begin{bmatrix} \boldsymbol{E}_{:,\mathtt{x}_1} \\ \boldsymbol{e}_N(1) \end{bmatrix}, \ldots, \begin{bmatrix} \boldsymbol{E}_{:,\mathtt{x}_p} \\ \boldsymbol{e}_N(p) \end{bmatrix}\right), \quad (3.3)$$

where the output dimension of the last layer has to be V. The next token for a deterministic transformer is selected to be the last element's largest logit: $\mathtt{x}_{p+1} = \arg\max_{u \in 1,\ldots,V} \boldsymbol{y}_{p,u}$. Given an input $(\mathtt{x}_1, \ldots, \mathtt{x}_p)$, the model then autoregressively extends this sequence one token at a time, following Equation (3.3) either until the sequence reaches a length N or until a special termination token is sampled.

A transformer has no separation between the system prompt $\mathtt{S}$, user provided input $\mathtt{X}$ and the autoregressively response $\mathtt{Y}$. Thus, a sequence conditional on user input is denoted as $(\mathtt{S}_1, ..., \mathtt{S}_{n_S}, \mathtt{X}_1, ..., \mathtt{X}_{n_X}, \mathtt{Y}_1, ..., \mathtt{Y}_{n_Y})$ and one without user input as $(\mathtt{S}_1, ..., \mathtt{S}_{n_S}, \mathtt{Y}_1, ..., \mathtt{Y}_{n_Y})$.

3.2.2 Context-based fine-tuning of a pretrained model

We now define prompting, soft prompting and prefix-tuning with the previously introduced notation.

Prompting. The most frequently used content-based fine-tuning approach is *prompting*: prefixing the input $(\mathtt{X}_1, ..., \mathtt{X}_{n_X})$ with a token sequence $S \in \{1, ..., V\}^{n_S}$ to guide the model response: $(\mathtt{S}_1, ..., \mathtt{S}_{n_S}, \mathtt{X}_1, ..., \mathtt{X}_{n_X})$. This is how most people interact with language models such as ChatGPT.

Soft prompting. Soft prompting replaces the embeddings of the system input $\boldsymbol{E}_{:,\mathtt{S}_i}$ with learned vectors $\boldsymbol{s}_i \in \mathbb{R}^{d_e}$ called *virtual tokens* (Hambardzumyan et al., 2021; Lester et al., 2021; Qin and Eisner, 2021). Hence, the input in Equation (3.3) is modified to be:

$$\left(\begin{bmatrix} \boldsymbol{s}_1 \\ \boldsymbol{e}_N(1) \end{bmatrix}, \ldots, \begin{bmatrix} \boldsymbol{s}_{n_S} \\ \boldsymbol{e}_N(n_S) \end{bmatrix}, \begin{bmatrix} \boldsymbol{E}_{:,\mathtt{X}_1} \\ \boldsymbol{e}_N(n_S+1) \end{bmatrix}, \ldots, \begin{bmatrix} \boldsymbol{E}_{:,\mathtt{X}_{n_X}} \\ \boldsymbol{e}_N(n_S+n_X) \end{bmatrix}\right) \quad (3.4)$$

with $\boldsymbol{s}_1, \ldots, \boldsymbol{s}_{n_S}$ being chosen to maximize the likelihood of a target response $Y = (\mathtt{Y}_1, ..., \mathtt{Y}_{n_Y})$, i.e., $\arg\max_{\boldsymbol{s}_1,\ldots,\boldsymbol{s}_{n_S} \in \mathbb{R}^{d_e}} \sum_{j=1}^{n_Y} \log \boldsymbol{y}_{n_S+n_X+j,\mathtt{Y}_j}$, where $\boldsymbol{y}_{n_S+n_X+j}$ are autoregressively generated.

Prefix-tuning. Prefix-tuning applies soft prompting across the depth of the model (Li and Liang, 2021; Liu et al., 2022). The first n_S positions for all attention blocks are learnable parameters, replacing the input $(\boldsymbol{x}_1^l, \ldots, \boldsymbol{x}_{n_X}^l)$ for layer l with $(\boldsymbol{s}_1^l, \ldots, \boldsymbol{s}_{n_S}^l, \boldsymbol{x}_1^l, \ldots, \boldsymbol{x}_{n_X}^l)$, where all $\boldsymbol{s}_i^l$ constitute the prefix. Hence, prefix-tuning can be formulated as

$$\arg \max_{\{\boldsymbol{s}_i^1,\ldots,\boldsymbol{s}_i^L\}_{i=1}^{n_S}} \sum_{j=1}^{n_Y} \log \boldsymbol{y}_{n_S+n_X+j,\mathtt{Y}_j}.$$

Prefix-tuning has been successful at fine-tuning models (Vu et al., 2022; Wu and Shi, 2022; Choi and Lee, 2023; Ouyang et al., 2023; Bai et al., 2023a), leading to calls for language models provided as a service (La Malfa et al., 2024) to allow providing prefixes instead of prompts (Sun et al., 2022).

Any token-based prompt $(\mathtt{S}_1, \ldots, \mathtt{S}_{n_S})$ has a corresponding soft prompt ($\boldsymbol{s}_i = \boldsymbol{E}_{:,\mathtt{S}_i}$) but the reverse does not hold. Similarly, every soft prompt $(\boldsymbol{s}_1, \ldots, \boldsymbol{s}_{n_S})$ can be represented as a prefix by setting the deeper prefixes to be the values that the model would compute at these positions ($\boldsymbol{s}_i^l = (\mathcal{A}_1 \mathbin{⨟} \ldots \mathbin{⨟} \mathcal{L}_{l-1,-1})([\boldsymbol{s}_1^\top, \boldsymbol{e}_N^\top(1)]^\top, \ldots, [\boldsymbol{s}_l^\top, \boldsymbol{e}_N^\top(l)]^\top)$). The reverse also does not hold: there are prefixes that cannot be represented as a soft prompt. A hierarchy emerges: *prompting* < *soft prompting* < *prefix-tuning*, with prefix-tuning the most powerful of the three. Hence, we focus on examining its performance relative to full fine-tuning but our findings also apply to prompting and soft prompting.

3.3 Soft prompting has more capacity than prompting

The success of soft prompting (and prefix-tuning) is commonly attributed to the larger capacity of the continuous embeddings compared to the finite tokens. Yet, increased capacity is beneficial only if the model can utilize it. We show this is indeed the case by constructing a transformer generating exponentially more completions by varying a single virtual token than by varying a hard token.

Consider unconditional generation (representing a function with no inputs) with a single system token: $(\mathtt{Y}_1, \ldots, \mathtt{Y}_N) = f(\mathtt{S}_1) = f_{\mathtt{S}_1}$. For a deterministic autoregressive function, there are a total of V functions in this family, hence the upper bound on the number of outputs of length N that one can generate by varying the first token $\mathtt{S}_1$ is V: the first token fully determines the rest of the sequence. Generally, if one varies the first N_S tokens, there are at most V^{N_S} unique outputs. What if instead of the token $\mathtt{S}_1$ we vary a single virtual token $\boldsymbol{s}_1$: $(\mathtt{Y}_1, \ldots, \mathtt{Y}_N) = f(\boldsymbol{s}_1) = f_{\boldsymbol{s}_1}$? This family of functions is indexed by a real vector and hence is infinite: in principle, one could generate all V^N possible output sequences by only controlling $\boldsymbol{s}_1$.[3] Still, a transformer may not be able to represent a function that achieves that in practice, i.e., it is not obvious if there is a surjective map from $\{f_{\boldsymbol{s}_1} : \boldsymbol{s}_1 \in \mathbb{R}^{d_e}\}$ to $\{1, \ldots, V\}^N$. We show that, in fact, there is a transformer f for which such a surjective map exists:

Theorem 3.1 (Exponential unconditional generation capacity of a single virtual token)**.** *For any $V, N > 1$, there exists a transformer with vocabulary size V, context size N, embedding size $d_e = N$, one attention layer with two heads and a three-layer MLP such that it generates any token sequence $(\mathtt{Y}_1, \ldots, \mathtt{Y}_N) \in \{1, \ldots, V\}^N$ when conditioned on the single virtual token $\boldsymbol{s}_1 = ((\mathtt{Y}_1 - 1)/V, \ldots, (\mathtt{Y}_N - 1)/V)$.*

[3] For example, for LLaMA-7B (Touvron et al., 2023) cannot have more than 32 000 unique completions when prompted with each of its 32 000 tokens. Still, we found a non-exhaustive set of 46 812 unique 10-token-long sequences by controlling the first virtual token. Hence, in practice, one can generate more outputs by soft prompting than by prompting.

However, conditional generation is more interesting: given a user input $(\mathtt{X}_1, ..., \mathtt{X}_{n_X})$, we want to generate a target response $(\mathtt{Y}_1, ..., \mathtt{Y}_{n_Y})$. Even in the simple case of one system token, the user provides one token and the model generates one token in response ($\mathtt{Y}_1 = f(\mathtt{S}_1, \mathtt{X}_1) = f_{\mathtt{S}_1}(\mathtt{X}_1)$), we cannot control the response of the model to any user input with the system token. There are V^V maps from $\mathtt{X}_1$ to $\mathtt{Y}_1$, but $\mathtt{S}_1$ can take on only V values: $|\{f_{\mathtt{S}_1} : \mathtt{S}_1 \in 1, ..., V\}| = V < V^V$. Hence, tokens cannot be used to specify an arbitrary map from user input to model output. However, a single virtual token can specify any of the V^V maps, i.e., there exists a transformer $f_{\boldsymbol{s}_1}(\mathtt{X}_2)$ for which there is a surjective map from $\{f_{\boldsymbol{s}_1} : \boldsymbol{s}_1 \in \mathbb{R}^{d_e}\}$ to $\{1, ..., V\}^{\{1,...,V\}}$.

Theorem 3.2 (Conditional generation capacity for a single virtual token ($n_X = n_Y = 1$)). *For any $V > 1$, there exists a transformer with vocabulary size V, context size $N = 2$, embedding size $d_e = V$, one attention layer with two heads and a three-layer MLP that reproduces any map $m : [1, ..., V] \to [1, ..., V]$ from a user input token to a model response token when conditioned on a single virtual token $\boldsymbol{s}_1 = ({m(1)}/{V}, ..., {m(V)}/{V})$. That is, by selecting $\boldsymbol{s}_1$ we control the model response to any user input.*

Theorem 3.2 builds on Theorem 3.1 by showing that soft prompting is also more expressive for governing the *conditional* behavior of a transformer model. This also holds for longer responses $n_Y > 1$ by increasing the length of the soft prompt, or longer user inputs $n_X > 1$, by increasing the depth of the model. We provide proofs in Appendix A.1, as well as working Python implementations.

This section showed that soft prompting, and by implication, prefix-tuning, possess greater expressiveness than prompting. As we can fully determine the map from user input to model response using virtual tokens, our findings may appear to suggest that soft prompting is as powerful as full fine-tuning. However, this is not at all the case. There are structural constraints on the capabilities of soft prompting and prefix-tuning; they cannot facilitate the learning of an entirely new task. The following section elucidates this discrepancy and reconciles these seemingly contradictory results.

3.4 Prefix-tuning can only bias the output of an attention head

We just saw that soft prompting and prefix-tuning can fully control the conditional behavior of a transformer. However, that assumed a specific design for the network weights. Given a fixed pretrained model, as opposed to a manually crafted one, can prefix-tuning be considered equally powerful to full fine-tuning? In this section, we show that, for an arbitrary pretrained model, a prefix S cannot change the relative attention over the content X, Y and can only bias the attention block outputs in a subspace of rank n_S, the prefix length, making it less powerful than full fine-tuning.

While full fine-tuning can alter the attention pattern of an attention head, prefix-tuning cannot. Recall the attention A_{ij} position i gives to position j for a trained transformer (Equation (3.1)):

$$A_{ij} = \frac{\exp\left(T/\sqrt{k}\; x_i^\top W_Q^\top W_K x_j\right)}{\sum_{r=1}^{p} \exp\left(T/\sqrt{k}\; x_i^\top W_Q^\top W_K x_r\right)} = \frac{\exp\left(T/\sqrt{k}\; x_i^\top H x_j\right)}{\sum_{r=1}^{p} \exp\left(T/\sqrt{k}\; x_i^\top H x_r\right)}, \tag{3.5}$$

where $W_Q^\top W_K = H$. Full fine-tuning can enact arbitrary changes to W_Q and W_K and hence, assuming the input does not change (e.g., at the first attention layer), we get the following attention:

$$A_{ij}^{\text{ft}} = \frac{\exp\left(T/\sqrt{k}\; x_i^\top H x_j + T/\sqrt{k}\; x_i^\top \Delta H x_j\right)}{\sum_{r=1}^{p} \exp\left(T/\sqrt{k}\; x_i^\top H x_r + T/\sqrt{k}\; x_i^\top \Delta H x_r\right)},$$

where the changes to W_Q and W_K are folded into ΔH. It is clear that by varying ΔH full fine-tuning can change the attention patterns arbitrarily. However, let us see how attention is affected by the presence of a prefix. For now, assume we have a prefix of length one (s_1) at position 0.

$$A_{i0}^{\text{pt}} = \frac{\exp\left(T/\sqrt{k}\; x_i^\top H s_1\right)}{\exp\left(T/\sqrt{k}\; x_i^\top H s_1\right) + \sum\limits_{r=1}^{p} \exp\left(T/\sqrt{k}\; x_i^\top H x_r\right)},$$

$$A_{ij}^{\text{pt}} = \frac{\exp\left(T/\sqrt{k}\; x_i^\top H x_j\right)}{\exp\left(T/\sqrt{k}\; x_i^\top H s_1\right) + \sum\limits_{r=1}^{p} \exp\left(T/\sqrt{k}\; x_i^\top H x_r\right)} \quad \text{for } j \geq 1.$$

The numerator of A_{ij}^{pt} is the same as in Equation (3.5), i.e., the prefix does not affect it. It only adds the term $\exp(T/\sqrt{k}\; x_i^\top H s_1)$ to the denominator. Therefore, the attention position i gives to the content positions $j \geq 1$ is simply scaled down by the attention it now gives to the prefix. As a result, the order of importance of the content tokens cannot change. This becomes evident by rewriting A_{ij}^{pt} as the attention of the pretrained model scaled by the attention "stolen" by the prefix:

$$A_{ij}^{\text{pt}} = A_{ij} \sum_{r=1}^{p} A_{ir}^{\text{pt}} = A_{ij}(1 - A_{i0}^{\text{pt}}). \tag{3.6}$$

Hence, prefix-tuning cannot affect the relative attention patterns across the content, it will only scale them down. In other words, one cannot modify what an attention head attends the most to via prefix-tuning.[4]

[4] Likhosherstov et al. (2021) show that a fixed attention head can approximate any sparse attention pattern. However, they require control over all the input embeddings while we can only control the prefix ones.

Prefix-tuning only adds a bias to the attention block output. Let us see how this attention scaling down affects the output of the attention block. Following Equation (3.2), the output at position i for the pretrained ($\boldsymbol{t}_i$), the fully fine-tuned ($\boldsymbol{t}_i^{\text{ft}}$) and the prefix-tuned ($\boldsymbol{t}_i^{\text{pt}}$) models are as follows:[5]

$$\begin{aligned}
\boldsymbol{t}_i &= \textstyle\sum_{j=1}^{p} \boldsymbol{A}_{ij} \boldsymbol{W}_V \boldsymbol{x}_j, \\
\boldsymbol{t}_i^{\text{ft}} &= \textstyle\sum_{j=1}^{p} \boldsymbol{A}_{ij}^{\text{ft}} (\boldsymbol{W}_V + \Delta \boldsymbol{W}_V) \boldsymbol{x}_j, \\
\boldsymbol{t}_i^{\text{pt}} &= \boldsymbol{A}_{i0}^{\text{pt}} \boldsymbol{W}_V \boldsymbol{s}_1 + \sum_{j=1}^{p} \boldsymbol{A}_{ij}^{\text{pt}} \boldsymbol{W}_V \boldsymbol{x}_j \overset{(3.6)}{=} \boldsymbol{A}_{i0}^{\text{pt}} \boldsymbol{W}_V \boldsymbol{s}_1 + \sum_{j=1}^{p} \boldsymbol{A}_{ij} (1 - \boldsymbol{A}_{i0}^{\text{pt}}) \boldsymbol{W}_V \boldsymbol{x}_j = \boldsymbol{A}_{i0}^{\text{pt}} \boldsymbol{W}_V \boldsymbol{s}_1 + (1 - \boldsymbol{A}_{i0}^{\text{pt}}) \boldsymbol{t}_i.
\end{aligned} \tag{3.7}$$

Hence, prefix-tuning only biases the attention block value at each position i towards the constant vector $\boldsymbol{W}_V \boldsymbol{s}_1$, which is independent of the content $(\boldsymbol{x}_1, ..., \boldsymbol{x}_p)$. I.e., the prefix-tuned activation is a linear combination of the pretrained activation and the constant vector $\boldsymbol{W}_V \boldsymbol{s}_1$. The content only affects the scale $\boldsymbol{A}_{i0}^{\text{pt}}$ of the bias via the amount of attention on the prefix. In contrast, in full fine-tuning $\Delta \boldsymbol{W}_Q$, $\Delta \boldsymbol{W}_K$ and $\Delta \boldsymbol{W}_V$ allow for a content-dependent change of the attention and value computation. These results hold for suffix-tuning (placing the prefix after the input) but *not* for suffix soft-prompting. We validate that this indeed is the case when prefix-tuning real-world transformers. In Figures A.2 and A.3, we show that a prefix applied to LLaMA's first layer does not change the relative attention distribution over the content positions X and results in a bias with a constant direction.

Longer prefixes define larger subspaces for the bias but are not fully utilized in practice. In the case of a longer prefix $(\boldsymbol{s}_1, \ldots, \boldsymbol{s}_{n_S})$, the bias vector is in a subspace of dimensionality n_S: $\boldsymbol{t}_i^{\text{pt}} = \sum_{j=1}^{n_S} \boldsymbol{A}_{i,S_j}^{\text{pt}} \boldsymbol{W}_V \boldsymbol{s}_j + (1 - \sum_{j=1}^{n_S} \boldsymbol{A}_{i,S_j}^{\text{pt}}) \boldsymbol{t}_i$, where i goes over the content and j over the prefix positions. Larger prefixes thus have a larger subspace to modify the attention block output. The specific direction is determined by the relative distribution of attention across the prefix positions. However, when we examine the distribution of attention across the prefix positions for various inputs as in Appendix A.2, it appears that the prefixes do not span this subspace. Regardless of the input, the attention $\boldsymbol{A}_{i,S_j}^{\text{pt}}$ over the prefix positions remains nearly constant. Thus, prefix-tuning does not seem to make full use of the space that the vectors $\boldsymbol{W}_V \boldsymbol{s}_j$ span. We hypothesise that this is due to the two competing optimization goals for the vectors $\boldsymbol{s}_j$: at the same time they need to "grab attention" when interacting with $\boldsymbol{W}_K$ and determine the bias direction when multiplied with $\boldsymbol{W}_V$.

So, is prefix-tuning equivalent to full fine-tuning or is it less powerful than full fine-tuning? In Section 3.3, we showed that prefix-tuning, in principle, has a large capacity to influence the behavior of the model. But then, in this section, we showed that it has some severe limitations, including not being able to affect the attention pattern and only biasing

[5] He et al. (2021a) show a similar analysis but do not study the expressiveness of prefix-tuning.

the attention layer activations. These two results seem to be contradicting one another, so how do we reconcile them?

The constructions for the results in Section 3.3 (described in Appendix A.1) are simply an algorithm that extracts the completion from a lookup table encoded in the virtual tokens. The attention patterns are simply extracting the current position embedding and the virtual token and hence the attention does not depend on the actual content in the tokens. There is no need to learn a new attention pattern to learn a different map from input to output.[6] Furthermore, the virtual token designates the map precisely by acting as a bias. Therefore, the observations in these two sections do not contradict one another. Soft prompting and prefix-tuning can be on par with full fine-tuning but only in very limited circumstances: when all the knowledge is represented in the virtual token as a lookup table and the model simply extracts the relevant entry. Transformers do not behave like this in practice. Models are typically trained with token inputs rather than virtual tokens. Moreover, if we had a lookup table of the responses to each input we would not need a learning algorithm in the first place.

Therefore, the limitations from this section hold for real-world pretrained transformers. Then how come prefix-tuning has been reported to achieve high accuracy and often to be competitive to full fine-tuning? The next section aims to explain when and why prefix-tuning can work in practice.

3.5 The bias can elicit skills from the pretrained model

Pretraining potentially exposes a model to different types of completions for the same token sequence. For a string like *I didn't enjoy the movie*, the model may have seen completions such as *I found the acting to be sub par*, *This is negative sentiment* or *Je n'ai pas aimé le film*. Hence, a pretrained model could do text completion, sentiment analysis, or translation. Still, the input does not fully determine the desired completion type and the model can generate any one of them. Hence, following our results from Section 3.4, we hypothesise that prefix-tuning cannot gain *new knowledge* but can bring to the surface *latent knowledge* present in the pretrained model.[7] We test this hypothesis by constructing small transformers trained on one or few tasks. We use a minimal transformer model (Karpathy, 2020) to show that prefix-tuning struggles to learn a new task that full fine-tuning can. Then, that prefix-tuning can easily elicit a latent skill from pretraining. Finally, we show how it can even learn *some* new tasks, provided they can be solved by combining pretraining skills.

[6]In a follow-up work (Petrov et al., 2024b), we utilize this observation to show that, in fact, there exist pre-trained weights for which a transformer can be a universal approximator for sequence-to-sequence functions when prefixed. This is not in contradiction with the present results as these transformers can approximate any function without having to modify their attention mechanism.

[7]A similar hypothesis has also been proposed by Reynolds and McDonell (2021) for fine-tuning in general.

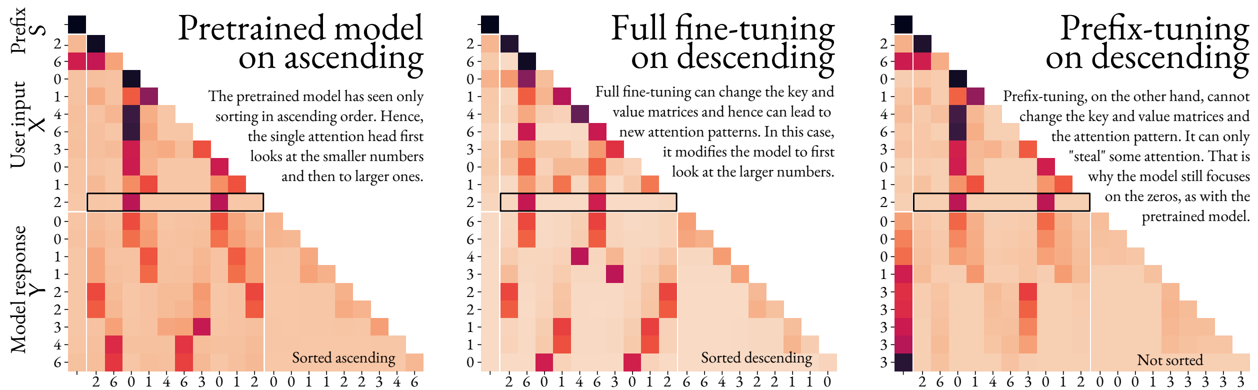


Figure 3.1: Attention patterns of a small transformer pretrained on sorting in ascending order. The model is given the prefix S and user input X and generates Y autoregressively. We have highlighted the attention when the first response $\mathtt{Y}_1$ is being generated. Full fine-tuning sorts in descending order but prefix-tuning cannot as it cannot update the learned attention. Note how the relative attention of X to X in the left and right plots is exactly the same: the prefix cannot change the attention pattern for the same inputs. The relative attention of X to X in the center plot is very different because full fine-tuning can arbitrarily change $\boldsymbol{W}_Q$ and $\boldsymbol{W}_K$.

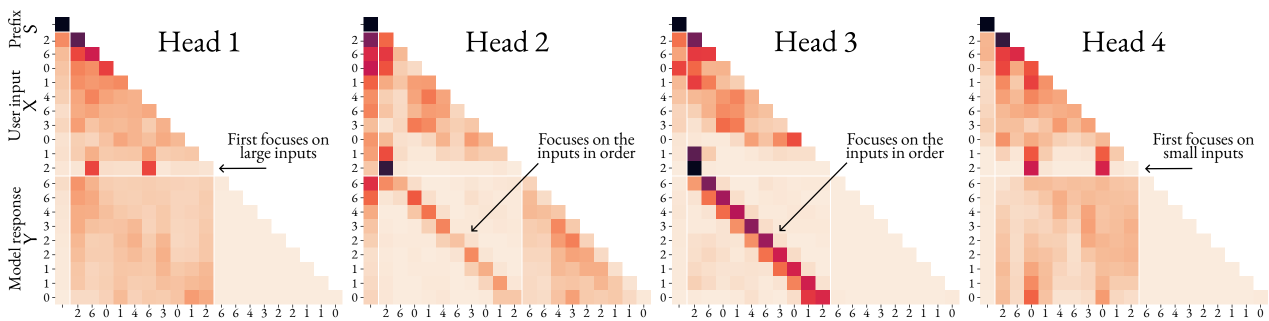


Figure 3.2: Model pretrained on the four tasks. The four attention heads specialize in the skills necessary to solve these tasks: look at the elements in order, look first at the smallest elements or first at the largest elements.

Table 3.1: A transformer pretrained on sorting in ascending order cannot be prefix-tuned to sort in descending order. 10 random seeds.

	Ascending	Descending
Pretrain on asc.	91±5%	0±0%
Full fine-tune on desc.	0±0%	85±5%
Prefix-tune on desc.	0±0%	0±0%

Table 3.2: A transformer pretrained on several tasks can be prefix-tuned for one of them. 10 random seeds.

Accuracy on:	↗	↘	+1	+2
Pretrained	25±13%	25±12%	24±11%	22±7%
Prefix-tune on ↗	95± 2%	0± 0%	0± 0%	0±0%
Prefix-tune on ↘	0± 0%	90± 3%	1± 1%	1±1%
Prefix-tune on +1	0± 0%	1± 3%	95± 6%	0±1%
Prefix-tune on +2	0± 0%	0± 0%	1± 2%	98±5%

Prefix-tuning may not learn a new task requiring a different attention pattern. To check if prefix-tuning can learn a new task, we train a 1-layer, 1-head transformer to sort numbers into ascending order and then fine-tune it to sort in descending order. During training, the model sees random sequences of 10 digits from 0 to 7 followed by their ascending sorted order. The pretrained accuracy (fully matching the sorted sequence) is 91%. Full fine-tuning on the descending task leads to 85% test accuracy, hence full fine-tuning successfully learns the new task. However, prefix-tuning with a prefix size $n_S = 1$ results in 0% accuracy, hence prefix-tuning fails to learn the new task at all.

The attention patterns in Figure 3.1 show why this is the case: the pretrained model learns to attend first to the smallest numbers and then to the larger ones. When fully fine-tuned, the attention patterns are reversed: they now first attend to the largest values. However, following Section 3.4, prefix-tuning cannot change the attention pattern over the input sequence and will still attend to the smallest values. Hence, prefix-tuning may indeed struggle to learn a new task requiring new attention patterns.

Prefix-tuning can elicit a skill from the pretrained model. The second part of our hypothesis was that prefix-tuning can elicit latent skills in the pretrained model. To test that, we pretrain a 1-layer, 4-head model with solutions sorted in ascending (↗) or descending (↘) order, or adding one (+1) or two (+2) to each element of the input sequence. Each solution is shown with 25% probability. The model has no indication of what the task is, hence, it assigns equal probability to all tasks, as shown in the first row in Table 3.2. Full fine-tuning for each task naturally results in high accuracy. However, prefix-tuning (n_S=1) can also reach accuracy above 90% for all tasks. Compared to the previous case, prefix-tuning is more successful here because the pretrained model contains the attention mechanisms for solving the four tasks, as shown in Figure 3.2.

If all a prefix does is bias the attention layer activations, how can it steer the model to collapse its distribution onto one task? This is likely due to the attention block solving all tasks in parallel and placing their solutions in different subspaces of the residual stream (intermediate representation, Elhage et al., 2021). As the MLP needs to select one solution to generate, a further indicator on the selected task (or lack of selection thereof) should also be represented. The bias induced by the prefix then acts on this "selection subspace" to

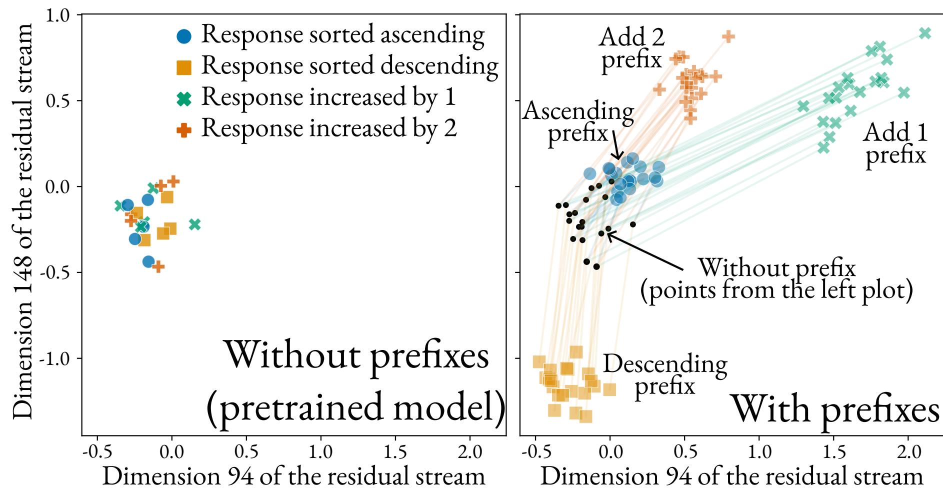


Figure 3.3: Attention block activations for ten sequences at the last input position (10) when pretrained on the four tasks. The left plot shows the pretrained activations $\boldsymbol{t}_{10}$ are not predictive of the completion. The right plot shows prefixes cluster the activations $\boldsymbol{t}_{10}^{\mathrm{pt}}$. Connecting the pretrained and prefixed activations highlights the bias. No dimensionality reduction is used; the clustering is solely due to the prefixes.

nudge the MLP to select the desired solution.

This can be clearly seen from the activations of the attention layer at the last input position ($\mathtt{X}_{n_X}$): the position where the task selection happens as the first output element fully describes the task. Figure 3.3 shows plots of randomly selected dimensions of the residual stream with and without a prefix. The attention block activations of the pretrained model (without prefix) show no correlation with the output it is about to generate, demonstrating that the choice of completion is indeed not determined by the attention block. However, the prefix-tuned activations for the same inputs are clustered as a result of the prefix-induced bias. This indicates that the bias induced by the prefix may act as a "task selector" of the subspace of the residual stream specializing in the desired task.

Prefix-tuning can combine knowledge from pretraining tasks to solve new tasks. Prefix-tuning eliciting one type of completion learned in pretraining starts to explain its practical utility. Still, prefix-tuning seems to be successful also at tasks that the pretrained model has not seen. As we showed above, a model trained to sort in one order cannot be prefix-tuned to sort in the other. Then how is it possible for prefix-tuning to learn a new task? We posit that this can happen, as long as the "skill" required to solve the new task is a combination of "skills" the pretrained model has seen.

We test this by pretraining a 40-layer 4-head model with the same four tasks. We prefix-tune ($n_S = 12$) for two new tasks: incrementing the ascending sorted sequence ($\nearrow$+1) and double histogram (mapping each element to the number of elements with the same value,

Table 3.3: Prefix tuning can learn a new task requiring only pretraining skills (↗+1) but cannot learn a completely new task (ℍ). Average accuracy over 3 seeds.

Accuracy on:	↗	↘	+1	+2	↗+1	ℍ
Pretrained	17%	23%	34%	25%	0%	0%
Prefix-tune on ↗	100%	0%	0%	0%	0%	0%
Prefix-tune on ↘	0%	100%	0%	0%	0%	0%
Prefix-tune on +1	0%	0%	100%	0%	0%	0%
Prefix-tune on +2	0%	0%	0%	100%	0%	0%
Prefix-tune on ↗+1	0%	0%	0%	0%	93%	0%
Prefix-tune on ℍ	0%	0%	0%	0%	0%	1%

e.g., 3,0,0,1↦1,2,2,1, ℍ). The pretrained model has not seen either task. Prefix-tuning results in 93% accuracy for ↗+1 which is a combination of the ↗ and +1 pretraining tasks and just 1% for the ℍ task which requires different skills: finding other instances of the same token and counting. ℍ is not a hard task: it requires 2 layers and 2 heads to be solved exactly (Weiss et al., 2021). Therefore, prefix-tuning can indeed combine skills that the model has learned in order to solve a novel task but may not learn a completely new task requiring new skills.

3.6 Effects of prefix-tuning beyond the single attention layer

Section 3.4 focused exclusively on a single attention layer. Still, even if a prefix only induces a bias on its output, this bias can exhibit complex behaviors via the subsequent MLPs and attention layers. This section shows how a prefix can change the attention pattern of the following attention layer but only in a linear fashion while full fine-tuning also has bilinear effects. Appendix A.3 further argues that the representational capacity of prefix-tuning may be limited. Therefore, prefix-tuning appears to be less expressive than full fine-tuning, even with the same number of learnable parameters.

Prefix-tuning can change the attention, albeit the one of the next layer Let us examine how the prefix of one attention layer affects the following one. Assume no MLPs, residual connections or layer norms: the output $\boldsymbol{t}_i^{(1)}$ of the first is the input $\boldsymbol{x}_i^{(2)}$ of the second. The pretrained outputs are $\boldsymbol{t}_i^{(1)} = \sum_{j=1}^{p} \boldsymbol{A}_{ij}^{(1)} \boldsymbol{W}_V^{(1)} \boldsymbol{x}_j^{(1)}$, resulting in the second layer attention $\tilde{\boldsymbol{A}}_{ij}^{(2)} = {}^{T}\!/\!\sqrt{k}\, \boldsymbol{t}_i^{(1)\top} \boldsymbol{H}^{(2)} \boldsymbol{t}_j^{(1)}$. Here $\tilde{\boldsymbol{A}}_{ij}$ is the pre-softmax attention, i.e., $\boldsymbol{A}_{ij} = {\exp \tilde{A}_{ij}}/{\sum_{r=1}^{p} \exp \tilde{A}_{ir}}$. For

prefix-tuning we then have:

$$\boldsymbol{t}_i^{\text{pt}(1)} = \boldsymbol{A}_{i0}^{\text{pt}(1)} \boldsymbol{W}_V \boldsymbol{s}_1^{(1)} + \sum_{j=1}^{p} \boldsymbol{A}_{ij}^{\text{pt}(1)} \boldsymbol{W}_V^{(1)} \boldsymbol{x}_j^{(1)} \overset{(3.7)}{=} \underbrace{\boldsymbol{A}_{i0}^{\text{pt}(1)}}_{\alpha_i} \underbrace{\boldsymbol{W}_V \boldsymbol{s}_1^{(1)}}_{\boldsymbol{\mu}} + (1 - \boldsymbol{A}_{i0}^{\text{pt}(1)}) \boldsymbol{t}_i^{(1)},$$

$$\begin{aligned}
\tilde{\boldsymbol{A}}_{ij}^{\text{pt}(2)} &= \tfrac{T}{\sqrt{k}}\, \boldsymbol{t}_i^{\text{pt}(1)\top} \boldsymbol{H}^{(2)} \boldsymbol{t}_j^{\text{pt}(1)}, \\
&= \tfrac{T}{\sqrt{k}} (\alpha_i \alpha_j \underbrace{\boldsymbol{\mu}^\top \boldsymbol{H}^{(2)} \boldsymbol{\mu}}_{\text{constant}} + \alpha_j (1\text{-}\alpha_i) \underbrace{\boldsymbol{t}_i^{(1)\top} \boldsymbol{H}^{(2)} \boldsymbol{\mu}}_{\text{depends only on } \boldsymbol{t}_i^{(1)}} + \alpha_i (1\text{-}\alpha_j) \underbrace{\boldsymbol{\mu}^\top \boldsymbol{H}^{(2)} \boldsymbol{t}_j^{(1)}}_{\text{depends only on } \boldsymbol{t}_j^{(1)}} + (1\text{-}\alpha_i)(1\text{-}\alpha_j) \underbrace{\boldsymbol{t}_i^{(1)\top} \boldsymbol{H}^{(2)} \boldsymbol{t}_j^{(1)}}_{\text{pretrained attention } \tilde{\boldsymbol{A}}_{ij}^{(2)}}).
\end{aligned}$$

The presence of $\boldsymbol{\mu}$ shows that the prefix of layer 1 can change the attention pattern of the following layer. This change is content-specific: the second and the third terms depend on the inputs, hence a simple bias can affect the attention when passed through MLPs and further attention blocks. Compare with Equation (3.6), which showed a prefix cannot change the attention of the same layer. Still, even considering this cross-layer effect, prefix-tuning is more limited in its expressiveness than full fine-tuning. While the second and the third terms are input-dependent, each depends on one input position only. The prefix does not change the bilinear dependency on both the query and key. This is something that the full fine-tuning can achieve: $\tilde{\boldsymbol{A}}_{ij}^{\text{ft}(2)} = T/\sqrt{k}\; \boldsymbol{t}_i^{\text{ft}(1)\top} (\boldsymbol{H}^{(2)} + \Delta \boldsymbol{H}^{(2)}) \boldsymbol{t}_j^{\text{ft}(1)}$.

Even if prefix-tuning could be a universal approximator, it would not be a parameter-efficient one. Prefix-tuning appears to be less parameter-efficient than other comparable approaches. We designed an experiment to this end. Our pretrained model in Section 3.5 failed to learn the double histogram task ($\mathbb{H}$). A rank-1 Low Rank Adaptation (LoRA, Hu et al., 2021) applied only to the MLPs in a 4-layer 4-head model pretrained in the exact same way results in 92% accuracy on the $\mathbb{H}$ task. The number of parameters for the LoRA fine-tuning is exactly the same as for a prefix of size 12. However, as can be expected from the results in Section 3.5, training this prefix results in 0% accuracy. Hence, prefix-tuning fails at a task that LoRA with the same number of parameters can learn.

In conclusion, while prefix-tuning, prompting, soft prompting and in-context learning have complex effects in deeper models, the interaction of the learnable parameters and the model inputs likely still results in very limited expressiveness. In particular, we demonstrated that LoRA can be used to learn a completely new task while prefix-tuning with the exact same number of parameters fails to.

3.7 Discussion and related works

Understanding fine-tuning and prefix-tuning. Prior works show that prefixes have low intrinsic dimension allowing transfer to similar tasks and initialization of prefixes for new tasks (Qin et al., 2021; Su et al., 2022; Zhong et al., 2022; Wang et al., 2022b; Zheng et al.,

2023). In this work, we offered theoretical insights into their results: this subspace is the span of the prefix-induced bias. Another line of work shows that skills can be localized in the parameter space of pretrained models (Wang et al., 2022a; Panigrahi et al., 2023). Here, we showed that it is also possible to identify subspaces of the residual stream corresponding to individual tasks and select them via prefix-tuning.

Prompting and in-context learning. Prompting and in-context learning are a special case of prefix-tuning. Therefore, the limitations and mechanisms discussed in this work apply to prompting as well: prompts cannot change the distribution of attention of the first attention layer over the content following it and can only induce a bias on the output of this layer (Section 3.4). Even considering the cross-layer effects, a prompt is strictly less expressive than full fine-tuning (Section 3.6) and prompting is unlikely to enable the model to solve a completely new task. Our theory thus explains why Kossen et al. (2023) observed that in-context examples cannot overcome pre-training skills.

While context-based fine-tuning approaches may not learn *arbitrary* new tasks, as shown in Section 3.5, they can leverage pre-trained skills. Wies et al. (2023) have PAC-learnability results that also show that when pretraining is on a mixture tasks, they can be efficiently learned via in-context learning, Moreover, transformers can learn linear models in-context by mimicking gradient descent (von Oswald et al., 2023a) or approximating matrix inversion (Akyürek et al., 2022). This is consistent with our theory: the prediction updates are enacted as biases in the attention block activations. Hence, despite the limitations discussed in this work, context-based methods can result in powerful fine-tuning if the pretrained model has "transferable skills" such as algorithmic fundamentals. Still, in-context learning will likely fail for non-algorithmic tasks, e.g., translating to a language that the model has never seen before, even if large number of translation pairs are provided in-context.

Implications for catastrophic forgetting and model alignment. The lack of expressiveness of context-based fine-tuning can be a feature: desirable properties will be maintained. Full fine-tuning can result in catastrophic forgetting (He et al., 2021b; Luo et al., 2023; Mukhoti et al., 2023). Our theory shows that context-based methods won't lose pretrained skills. Model alignment poses the reverse problem: ensuring that the model cannot pick up undesirable skills during fine-tuning. Our results show that prompting and prefix-tuning might be unable to steer the model towards new adversarial behaviors. Hence, the recent successes in adversarial prompting (Zou et al., 2023) indicate that current model alignment methods just mask the undesirable skills rather than removing them.

Implications for model interpretability. One open question for language model interpretability is whether attention is sufficient for explainability (Jain and Wallace, 2019; Wiegreffe and Pinter, 2019). Section 3.5 points toward the negative: by interfering in the *output* of the attention layer with the bias induced by a prefix, we can change the behavior of the model, without changing its attention. On the flip side, prefix-tuning can be used to

understand what “skills” a model has: if prefix-tuning for a task fails, then the model likely lacks one of the key “skills” for that task.

Limitations. The present analysis is largely limited to prefixing with prompts, soft prompts and for prefix-tuning. While our theoretical results hold for suffix-tuning, they do not necessarily apply to suffixing with prompts or soft prompts. That is because the deeper representations for prompt and soft prompt suffixes would depend on the previous positions. This does not apply to suffix-tuning as it fixes all intermediate representations. Therefore, whether suffixing is more expressive than prefixing remains an open question. Separately, while we provided evidence towards context-based fine-tuning methods being parameter inefficient learners, the formal analysis of the conditions under which they may be universal approximators remain an open question. Finally, we mostly considered simple toy problems. In practice, however, language models are pretrained with very large datasets and can pick up very complex behaviors. Hence, the extent to which the limitations we demonstrated apply to large-scale pretrained transformers also remains for future work.

3.8 Conclusion

This paper formally showed that fine-tuning techniques working in embedding space, such as soft prompting and prefix-tuning, are strictly more expressive than prompting which operates in the discrete token space. However, we then demonstrated that despite this larger expressivity, prefix-tuning suffers from structural limitations that prevent it from learning new attention patterns. As a result, it can only bias the output of the attention layer in a direction from a subspace of rank equal to the size of the prefix. We showed that this results in practical limitations by constructing minimal transformers where prefix tuning fails to solve a simple task. This result seems to be at odds with the empirical success of prefix-tuning. We provided explanations towards that. First, we showed that prefix-tuning can easily elicit a skill the pretrained model already has and can even learn a new task, if it has picked up the skills to solve it during pretraining. Second, we showed that the effect of the prefix-induced bias is more complicated and powerful when combined with downstream non-linear operations. However, it appears to be still less expressive than full fine-tuning.

Reproducibility statement

In order to facilitate the reproduction of our empirical results, validating our theoretical results, and further studying the properties of context-based fine-tuning, we release all our code and resources used in this work. Furthermore, in Appendix A.1 we offer explicit constructions of transformers with the properties discussed in Section 3.3. We also provide Python implementations of these constructions that validate their correctness.

Author contributions

This work was a collaboration with my supervisors, Dr. Adel Bibi and Prof. Philip H.S. Torr. The initial idea to look at the expressivity of prompting and prefix-tuning was mine. Adel suggested that if prefix-tuning has that much capacity then there might be an equivalence between the prefix-tuning and full fine-tuning, which precipitated the rest of the paper. I observed and proved the theoretical results, and designed and implemented the experimental evaluation. Both Adel and Phil provided guidance and advice on how to approach, structure and present the project.

Acknowledgements

This work is supported by a UKRI grant Turing AI Fellowship (EP/W002981/1), as well as the EPSRC Centre for Doctoral Training in Autonomous Intelligent Machines and Systems (EP/S024050/1). AB has received funding from the Amazon Research Awards. We also thank the Royal Academy of Engineering and FiveAI.

Chapter 4

Prompting a Pretrained Transformer Can Be a Universal Approximator

Abstract

Despite the widespread adoption of prompting, prompt tuning and prefix-tuning of transformer models, our theoretical understanding of these fine-tuning methods remains limited. A key question is whether one can arbitrarily modify the behavior of the pretrained model by prompting or prefix-tuning it. Formally, if prompting and prefix-tuning a pretrained model can universally approximate sequence-to-sequence functions. This paper answers in the affirmative and demonstrates that much smaller pretrained models than previously thought can be universal approximators when prefixed. In fact, the attention mechanism is uniquely suited for universal approximation with prefix-tuning a single attention head being sufficient to approximate any continuous function. Moreover, any sequence-to-sequence function can be approximated by prefixing a transformer with depth linear, or even constant, in the sequence length. Beyond these density-type results, we also offer Jackson-type bounds on the length of the prefix needed to approximate a function to a desired precision.

4.1 Introduction

The scale of modern transformer architectures (Vaswani et al., 2017) is ever-increasing and training competitive models from scratch, even fine-tuning them, is often prohibitively expensive (Lialin et al., 2023). To that end, there has been a proliferation of research aiming at efficient training in general and fine-tuning in particular (Rebuffi et al., 2017; Houlsby et al., 2019; Hu et al., 2021, 2023).

Motivated by the success of few- and zero-shot learning (Wei et al., 2021; Kojima et al., 2022), context-based fine-tuning methods do not change the model parameters. Instead, they modify the way the input is presented. For example, with prompting, one fine-tunes a string of tokens (*a prompt*) which is prepended to the user input (Shin et al., 2020; Liu et al., 2023). As optimizing over discrete tokens is difficult, one can optimize the real-valued embeddings instead (*soft prompting, prompt tuning*, Lester et al. 2021). A generalization to this approach is the optimization over the embeddings of every attention layer (*prefix-tuning*, Li and Liang 2021). These methods are attractive as they require a few learnable parameters and allow for different prefixes to be used for different samples in the same batch which is not possible with methods that change the model parameters. As every prompt and soft prompt can be expressed as prefix-tuning (Petrov et al., 2024c), in this paper, we will focus primarily on prefix-tuning.

While these context-based fine-tuning techniques have seen widespread adoption and are, in some cases, competitive to full fine-tuning (Liu et al., 2022), our understanding of their abilities and restrictions remains limited. How much can the behavior of a model be modified without changing any model parameter? Given a pretrained transformer and an arbitrary target function, how long should the prefix be so that the transformer approximates this function to an arbitrary precision? Differently put, can prefix-tuning of a pretrained transformer be a universal approximator? These are some of the questions we aim to address in this work.

It is well-known that fully-connected neural networks with suitable activation functions can approximate any continuous function (Cybenko, 1989; Hornik et al., 1989; Barron, 1993; Telgarsky, 2015), while Recurrent Neural Networks (RNNs) can approximate dynamical system. The attention mechanism (Bahdanau et al., 2015) has also been studied in its own right. Deora et al. (2023) derived convergence and generalization guarantees for gradient-descent training of a single-layer multi-head self-attention model, and Mahdavi et al. (2023) showed that the memorization capacity increases linearly with the number of attention heads. On the other hand, it was shown that attention layers are not expressive enough as they lose rank doubly exponentially with depth if Multi-Layer Perceptrons (MLPs) and residual connections are not present (Dong et al., 2021). However, attention layers, with a hidden size that grows only logarithmically in the sequence lengths, were shown to be good approximators for sparse attention patterns (Likhosherstov et al., 2021), except for a few tasks that require linear scaling of the size of the hidden layers in the sequence length (Sanford et al., 2023).

Considering universal approximation using encoder-only transformers, Yun et al. (2020) showed that transformers are universal approximators of sequence-to-sequence functions by demonstrating that self-attention layers can compute contextual mappings of input sequences. Jiang and Li (2024) demonstrated universality by instead leveraging the Kolmogorov-Arnold representation theorem. Moreover, Alberti et al. (2023) provided universal approximation results for architectures with non-standard attention mechanisms.

Despite this interest in theoretically understanding the approximation properties of the

transformer architecture when being trained, much less progress has been made in understanding context-based fine-tuning methods such as prompting, soft prompting, and prefix-tuning. Petrov et al. (2024c) have shown that the presence of a prefix cannot change the relative attention over the context and experimentally demonstrated that one cannot learn completely novel tasks with prefix-tuning. In the realm of in-context learning, where input-target pairs are part of the prompt (Brown et al., 2020), Xie et al. (2021) and Yadlowsky et al. (2023) show that the ability to generalize depends on the choice of pretraining tasks. However, these are not universal approximation results. The closest to our objective is the work of Wang et al. (2023). They quantize the input and output spaces allowing them to enumerate all possible sequence-to-sequence functions. All possible functions and inputs can then be hard-coded in a transformer using the constructions by Yun et al. (2020). As this approach relies on memorization, the depth of the model depends on the desired approximation precision ϵ.

In this work, we demonstrate that prefix-tuning can be a universal approximator much more efficiently than previously assumed. In particular:

1. We show that attention heads are especially suited to model functions over hyperspheres, concretely, prefix-tuning *a single attention head* is sufficient to approximate any smooth continuous function on the hypersphere S^m to any desired precision ϵ;
2. We give a bound on the required prompt length to approximate a smooth target function to a precision ϵ;
3. We demonstrate how this result can be leveraged to approximate general sequence-to-sequence functions with transformers of depth linear (or even constant) in the sequence length and independent of ϵ;
4. We discuss how prefix-tuning may result in element-wise functions which, when combined with cross-element mixing from the pretrained model, may be able to explain the success behind prefix-tuning and prompting and why it works for some tasks and not others.

4.2 Background material

4.2.1 Transformer architecture

In soft prompting and prefix-tuning, the focus of this work, the sequence fed to a transformer model is split into two parts: a *prefix* sequence $P = (\boldsymbol{p}_1, \ldots, \boldsymbol{p}_N)$, which is to be learnt or hand-crafted, and an *input* sequence $X = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T)$, where $\boldsymbol{x}_i, \boldsymbol{p}_i \in \mathbb{R}^d$. A transformer operating on a sequence consists of alternating attention blocks which operate on the whole sequence and MLPs that operate on individual elements. For a sequence of length $N + T$, an attention head of dimension d is a function $\tilde{u} : \mathbb{R}^{d\times(N+T)} \to \mathbb{R}^{d\times(N+T)}$. Since we will only be interested in the output at the positions corresponding to the inputs X, we use

$u(\cdot\,;P) : \mathbb{R}^{d\times(N+T)} \to \mathbb{R}^{d\times T}$ to denote the output of $\tilde{u}$ at the locations corresponding to the input X when prefixed with P. Therefore, the k-th output of u is defined as:

$$[u(X;P)]_k = \frac{\sum_{i=1}^{N} \exp(\boldsymbol{x}_k^\top \boldsymbol{H} \boldsymbol{p}_i) \boldsymbol{W}_V \boldsymbol{p}_i + \sum_{j=1}^{T} \exp(\boldsymbol{x}_k^\top \boldsymbol{H} \boldsymbol{x}_j) \boldsymbol{W}_V \boldsymbol{x}_j}{\sum_{i=1}^{N} \exp(\boldsymbol{x}_k^\top \boldsymbol{H} \boldsymbol{p}_i) + \sum_{j=1}^{T} \exp(\boldsymbol{x}_k^\top \boldsymbol{H} \boldsymbol{x}_j)}, \tag{4.1}$$

where $\boldsymbol{W}_V$, the *value* matrix, and $\boldsymbol{H}$ are in $\mathbb{R}^{d\times d}$. $\boldsymbol{H}$ is typically split into two lower-rank matrices $\boldsymbol{H} = \boldsymbol{W}_Q^\top \boldsymbol{W}_K$, *query* and *key* matrices. Multiple attention heads can be combined into an attention block but, for simplicity, we will only consider single head attention blocks. A transformer is then constructed by alternating attention heads and MLPs.

We consider *pretrained* transformers but, in the context of this work, these are constructed rather than trained. We refer to the matrices $\boldsymbol{W}_V, \boldsymbol{W}_Q, \boldsymbol{W}_K$ along with the parameters of the MLPs as *pretrained parameters*, and they are fixed throughout and not learnt. The prefix P is the only variable that can be modified to change the behavior of the model.

4.2.2 Universal approximation

Let $\mathcal{X}$ and $\mathcal{Y}$ be normed vector spaces. We consider a family of *target functions* which is a subset $\mathcal{C}$ of all mappings $\mathcal{X} \to \mathcal{Y}$, i.e., $\mathcal{C} \subseteq \mathcal{Y}^{\mathcal{X}}$, with $\mathcal{C}$ is often referred to as a *concept space*. These are the relationships we wish to learn by some simpler candidate functions. Let us denote this set of candidates by $\mathcal{H} \subset \mathcal{Y}^{\mathcal{X}}$, called *hypothesis space*. The problem of approximation is concerned with how well functions in $\mathcal{H}$ approximate functions in $\mathcal{C}$. There are two main ways to measure how well functions in $\mathcal{H}$ represent functions in $\mathcal{C}$: density results and approximation rate results (Jiang et al., 2023). Density results show that, given an ϵ, one can find a hypothesis $h \in \mathcal{H}$ approximating any $f \in \mathcal{C}$ with error at most ϵ. Approximation rate results, also called *Jackson-type*, are stronger as they offer a measure of complexity for h to reach a desired precision ϵ. Classically, a Jackson-type result would provide a minimum width or depth necessary for a neural network to reach a desired precision ϵ. In the context of the present work, the notion of complexity that we care about is the length N of the prefix P. Formally:

Definition 4.1 (Universal Approximation (Density-Type)). We say that $\mathcal{H}$ is a universal approximator for $\mathcal{C}$ over a compact set $S \subseteq \mathcal{X}$ if for every $f \in \mathcal{C}$ and every $\epsilon > 0$ there exists an $h \in \mathcal{H}$ such that $\sup_{x\in S} \|f(x) - h(x)\| \leq \epsilon$. One typically says that $\mathcal{H}$ *is dense in* $\mathcal{C}$.

Lemma 4.2 (Transitivity). *If $\mathcal{A}$ is dense in $\mathcal{B}$ and $\mathcal{B}$ is dense in $\mathcal{C}$, then $\mathcal{A}$ is dense in $\mathcal{C}$.*

Definition 4.3 (Approximation Rate (Jackson-Type)). Fix a hypothesis space $\mathcal{H}$. Let $\{\mathcal{H}^N : N \in \mathbb{N}_+\}$ be a collection of subsets of $\mathcal{H}$ such that $\mathcal{H}^N \subset \mathcal{H}^{N+1}$ and $\bigcup_{N\in\mathbb{N}_+} \mathcal{H}^N = \mathcal{H}$. Here, N is a measure of the complexity of the approximation candidates, and $\mathcal{H}^N$ is the subset of

hypotheses with complexity at most N. Then, the approximation rate estimate for $\mathcal{C}$ over a compact $S \subseteq \mathcal{X}$ is a bound $Z_{\mathcal{H}}$:

$$N \geq Z_{\mathcal{H}}(f, \epsilon) \implies \inf_{h \in \mathcal{H}^N} \sup_{x \in S} \|f(x) - h(x)\| \leq \epsilon, \forall f \in \mathcal{C}.$$

$Z_{\mathcal{H}}$ gives an upper bound to the minimum hypothesis complexity necessary to reach the target precision ϵ and typically depends on the smoothness of f.

Lemma 4.4. *A Jackson bound for $\{\mathcal{H}^N \mid N \in \mathbb{N}_+\}$ with finite $Z_{\mathcal{H}}$ for all $f \in \mathcal{C}, \epsilon > 0$ immediately implies that $\bigcup_{N \in \mathbb{N}_+} \mathcal{H}^N = \mathcal{H}$ is dense in $\mathcal{C}$. Hence, Jackson bounds (Definition 4.3) are stronger than density results (Definition 4.1).*

The key hypothesis classes we consider in this work are the set of all prefixed attention heads and the set of prefixed transformers. This is very different from the classical universal approximation setting. The hypothesis classes in the classical universal approximation results consist of all possible parameter values of the model itself (Cybenko, 1989; Yun et al., 2020). When studying universal approximation with prefixing, the model parameters are fixed where prefixes are what can be modified.

Definition 4.5 (Prefixed Attention Heads Class)**.** This is the class of all attention heads as defined in Equation (4.1) of dimension d, input/output sequence of length T, prefix of length at most N, and *fixed* pretrained components $\boldsymbol{H}, \boldsymbol{W}_V \in \mathbb{R}^{d \times d}$:

$$\mathcal{H}^{N,T}_{-,d}(\boldsymbol{H}, \boldsymbol{W}_V) = \left\{ \begin{matrix} u : \mathbb{R}^{d \times (N'+T)} \to \mathbb{R}^{d \times T}, \\ [u]_k \text{ as in (4.1)}, \boldsymbol{p}_i \in \mathbb{R}^d, N' \leq N \end{matrix} \right\}.$$

For simplicity, we say that $\mathcal{H}^{N,T}_{-,d}$ *is dense in* $\mathcal{C}$ to imply that there exists a pair $(\boldsymbol{H}, \boldsymbol{W}_V)$ such that $\mathcal{H}^{N,T}_{-,d}(\boldsymbol{H}, \boldsymbol{W}_V)$ is dense in $\mathcal{C}$. When considering all possible prefix lengths, we drop the N: $\mathcal{H}^{T}_{-,d} = \bigcup_{N \in \mathbb{N}} \mathcal{H}^{N,T}_{-,d}$.

Definition 4.6 (Prefixed Transformers Class)**.** A transformer consists of L layers with each layer l consisting of an attention head with $\boldsymbol{H}^l$ and $\boldsymbol{W}_V^l$ followed by an MLP consisting of k_l linear layers, each parameterized as $\mathcal{L}_k^l(\boldsymbol{x}) = \boldsymbol{A}^{l,k}\boldsymbol{x} + \boldsymbol{b}^{l,k}$ interspersed with non-linear activation σ. This gives rise to the following hypothesis class when prefixed:

$$\begin{aligned} &\mathcal{H}^{N,T}_{\equiv,d}\left(\left\{\boldsymbol{H}^l, \boldsymbol{W}_V^l, \{(\boldsymbol{A}^{l,k}, \boldsymbol{b}^{l,k})\}_{k=1}^{k_l}\right\}_{l=1}^{L}\right) \\ &= \left\{ \begin{matrix} \mathcal{L}^L_{k_L} \circ \ldots \circ \sigma \circ \mathcal{L}^L_1 \circ h^L \ldots \circ h^2 \circ \mathcal{L}^1_{k_1} \circ \ldots \circ \sigma \circ \mathcal{L}^1_1 \circ h^1 \\ \text{with } h^l \in \mathcal{H}^{N',T}_{-,d}(\boldsymbol{H}^l, \boldsymbol{W}_V^l),\ l = 1, \ldots, L, N' \leq N. \end{matrix} \right\}, \end{aligned}$$

applying linear layers $\mathcal{L}$ element-wise. Again, we say $\mathcal{H}^{N,T}_{\equiv,d}$ *is dense in* $\mathcal{C}$, short for *there exist* $\left\{\boldsymbol{H}^l, \boldsymbol{W}_V^l, \{\boldsymbol{A}^{l,k}, \boldsymbol{b}^{l,k}\}_{k=1}^{k_l}\right\}_{l=1}^{L}$ *such that* $\mathcal{H}^{N,T}_{\equiv,d}\left(\left\{\boldsymbol{H}^l, \boldsymbol{W}_V^l, \{\boldsymbol{A}^{l,k}, \boldsymbol{b}^{l,k}\}_{k=1}^{k_l}\right\}_{l=1}^{L}\right)$ *is dense in* $\mathcal{C}$.

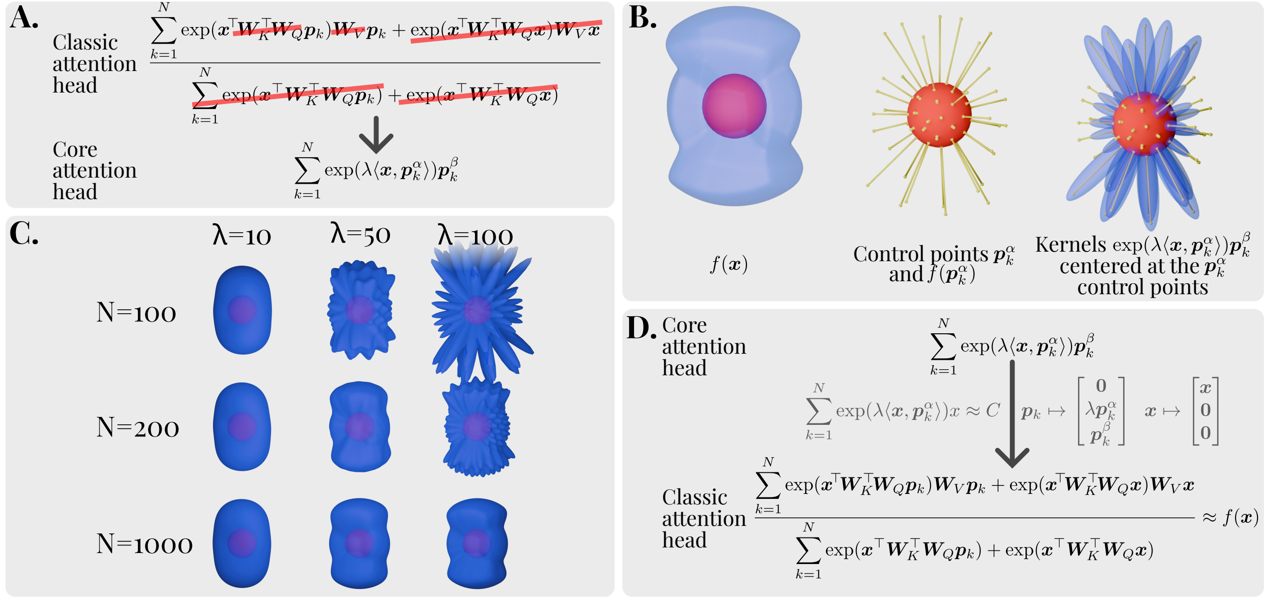


Figure 4.1: **Approximating functions on the hypersphere with a single attention head.** **A.** We simplify the classical attention head into a *core attention head*. **B.** The $\exp(\lambda\langle \boldsymbol{x}, \boldsymbol{p}_k^\alpha\rangle)\boldsymbol{p}_k^\beta$ terms act like kernels when $\boldsymbol{x}$ is restricted to a hypersphere. We can approximate a function f by placing N control points $\boldsymbol{p}_1^\alpha, \dots, \boldsymbol{p}_N^\alpha$ and centering a kernel at each of them. **C.** Increasing λ results in less smoothing, while increasing N results in more control points and hence better approximation. With large enough λ and N, we can approximate f to any desired accuracy. **D.** With the normalization term in classical attention close to a constant, and giving $\boldsymbol{x}$, $\boldsymbol{p}_k^\alpha$ and $\boldsymbol{p}_k^\beta$ orthogonal subspaces, core attention can be represented as classical attention. Hence, a classical attention head can also approximate f with arbitrary precision.

In this paper, we consider several different concept classes. For reasons that will become apparent in the following section, we focus on functions whose domain is a hypersphere $S^m = \{\boldsymbol{y} \in \mathbb{R}^{m+1} \mid \|\boldsymbol{y}\|_2 = 1\} \subset \mathbb{R}^{m+1}$. We consider both scalar and vector-valued functions on the hypersphere.

Definition 4.7 (Scalar Functions on the Hypersphere)**.** Define $C(S^m) \subset \mathbb{R}^{S^m}$ to be the space of all continuous functions defined on S^m with bounded norm, i.e.,

$$\|f\|_\infty = \sup_{\boldsymbol{x} \in S^m} |f(\boldsymbol{x})| < \infty, \; f \in C(S^m). \tag{4.2}$$

This is the concept class $\mathcal{C}_{s,m} = C(S^m) \subset \mathbb{R}^{S^m}$.

Definition 4.8 (Vector-valued Functions on the Hypersphere)**.** The class of vector-valued functions on the hypersphere is:

$$\mathcal{C}_{v,m} = \left\{ f : S^m \to \mathbb{R}^{m+1} \mid [f]_i \in C(S^m), i = 1, \dots, m+1 \right\}.$$

Transformers are typically used to learn mappings over sequences rather than individual inputs. Hence, we define two sequence-to-sequence concept classes:

Definition 4.9 (General Sequence-to-sequence Functions)**.** Given a fixed sequence length $T \in \mathbb{N}_{>0}$, we define the sequence-to-sequence function class as:

$$\mathcal{C}_{T,m} = \left\{ f : (S^m)^T \to (\mathbb{R}^{m+1})^T \mid f \text{ continuous and bounded} \right\}.$$

We will also consider the subset of element-wise functions:

Definition 4.10 (Element-wise functions)**.** Element-wise functions operate over sequences of inputs but apply the exact same function independently to all inputs:

$$\mathcal{C}_{\|,T,m} = \left\{ f \in \mathcal{C}_{T,m} \;\middle|\; \begin{array}{l} \text{there exists } g \in \mathcal{C}_{v,m}, \text{ such that} \\ f(\boldsymbol{x}_1, \dots, \boldsymbol{x}_T) = (g(\boldsymbol{x}_1), \dots, g(\boldsymbol{x}_T)) \\ \text{for all } (\boldsymbol{x}_1, \dots, \boldsymbol{x}_T) \in (S^m)^T \end{array} \right\}.$$

4.3 Universal approximation with a single attention head

In this section, we will restrict ourselves to the setting when the input sequence is of length T=1, i.e., X=$(\boldsymbol{x})$. General sequence-to-sequence functions will be discussed in Section 4.4. We will show that a single attention head can approximate any continuous function on the hypersphere, or that $\mathcal{H}^1_{-,m+1}$ is dense in $\mathcal{C}_{s,m}$. To do this, we first simplify the classical attention head in Equation (4.1), resulting in what we call a *core attention head*. Then, we show that each of the terms in the core attention act as a kernel, meaning that it can approximate any function in $\mathcal{C}_{s,m}$. Finally, we show that any core attention head can be approximated by a classical attention head, hence, $\mathcal{H}^1_{-,m+1}$ is indeed dense in $\mathcal{C}_{s,m}$. The complete pipeline is illustrated in Figure 4.1.

To illuminate the approximation abilities of the attention head mechanism we relax it a bit. That is, we allow for different values of the prefix positions when computing the attention (the exp terms in Equation (4.1)) and when computing the value (the right multiplication with $\boldsymbol{W}_V$). We will also drop the terms depending only on $\boldsymbol{x}$, set $\boldsymbol{H} = \lambda \boldsymbol{I}_d$, $\lambda > 0$, and $\boldsymbol{W}_V = \boldsymbol{I}_d$. We refer to this relaxed version as a *split attention head* with its corresponding hypothesis class:

$$h_{\Downarrow\Uparrow}(\boldsymbol{x}) = \frac{\sum_{k=1}^{N'} \exp(\lambda \langle \boldsymbol{x}, \boldsymbol{p}_k^\alpha \rangle) \boldsymbol{p}_k^\beta}{\sum_{k=1}^{N'} \exp(\lambda \langle \boldsymbol{x}, \boldsymbol{p}_k^\alpha \rangle)}. \tag{4.3}$$

Definition 4.11 (Split Attention Head Class)**.**

$$\mathcal{H}^N_{\Downarrow\Uparrow,d} = \left\{ h_{\Downarrow\Uparrow} \text{ as in (4.3)}, \boldsymbol{p}_k^\alpha, \boldsymbol{p}_k^\beta \in \mathbb{R}^d, N' \leq N, \lambda > 0 \right\}.$$

We will later show that a split head can be represented by a classical attention head. For now, let us simplify a bit further: we drop the denominator, resulting in our *core attention head*:

$$h_{\circledast}(\boldsymbol{x}) = \sum_{k=1}^{N} \exp(\lambda \langle \boldsymbol{x}, \boldsymbol{p}_k^\alpha \rangle) \boldsymbol{p}_k^\beta, \tag{4.4}$$

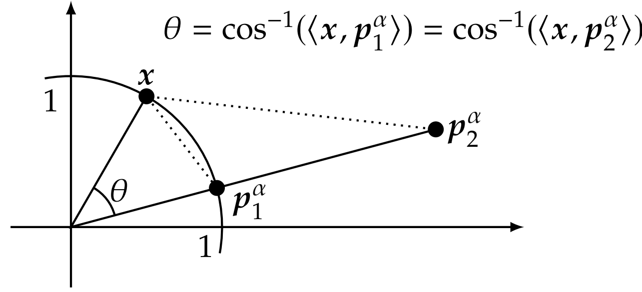


Figure 4.2: **The dot product is a measure of closeness over the hypersphere.** We want large dot product for points with lower distances. That is not the case for general $\boldsymbol{p}_1^\alpha, \boldsymbol{p}_2^\alpha \in \mathbb{R}^{m+1}$: above we show larger dot product for points which are further away, i.e., $\langle \boldsymbol{x}, \boldsymbol{p}_1^\alpha \rangle < \langle \boldsymbol{x}, \boldsymbol{p}_2^\alpha \rangle$ despite $\|\boldsymbol{x} - \boldsymbol{p}_1^\alpha\|_2 < \|\boldsymbol{x} - \boldsymbol{p}_1^\alpha\|_2$. However, if we restrict $\boldsymbol{x}$, $\boldsymbol{p}_i^\alpha$, and $\boldsymbol{p}_j^\alpha$ to the hypersphere S^m, then the dot product measures the cosine between $\boldsymbol{x}$ and $\boldsymbol{p}_i$ which is truly a measure of closeness: $\langle \boldsymbol{x}, \boldsymbol{p}_i^\alpha \rangle < \langle \boldsymbol{x}, \boldsymbol{p}_j^\alpha \rangle \iff \|\boldsymbol{x} - \boldsymbol{p}_i^\alpha\|_2 > \|\boldsymbol{x} - \boldsymbol{p}_j^\alpha\|_2$.

which gives rise to the hypothesis class:

$$\mathcal{H}_{\circledast,d}^N = \left\{ \begin{matrix} \boldsymbol{x} \mapsto \sum_{k=1}^{N'} \exp(\lambda \langle \boldsymbol{x}, \boldsymbol{p}_k^\alpha \rangle) \boldsymbol{p}_k^\beta, \text{ where} \\ \boldsymbol{p}_k^\alpha, \boldsymbol{p}_k^\beta \in \mathbb{R}^d, N' \leq N, \lambda > 0 \end{matrix} \right\}.$$

We also have their scalar-valued counterparts:

$$h_\odot(\boldsymbol{x}) = \sum_{k=1}^{N} \exp(\lambda \langle \boldsymbol{x}, \boldsymbol{p}_k^\alpha \rangle) p_k^\beta, \tag{4.5}$$

$$\mathcal{H}_{\odot,d}^N = \left\{ \begin{matrix} \boldsymbol{x} \mapsto \sum_{k=1}^{N'} \exp(\lambda \langle \boldsymbol{x}, \boldsymbol{p}_k^\alpha \rangle) p_k^\beta, \text{ where} \\ \boldsymbol{p}_k^\alpha \in \mathbb{R}^d, p_k^\beta \in \mathbb{R}, N' \leq N, \lambda > 0 \end{matrix} \right\}.$$

As the dot product is a notion of similarity, one can interpret $h_\circledast$ in Equation (4.4) and $h_\odot$ in Equation (4.5) as interpolators. The $\boldsymbol{p}_i^\alpha$ vectors act as control points, while the $\boldsymbol{p}_i^\beta$ vectors designate the output value at the location of the corresponding control point. The dot product with the input $\boldsymbol{x}$ controls how much each control point should contribute to the final result, with control points closer to $\boldsymbol{x}$ (larger dot product) contributing more.

Unfortunately, it is not generally true that higher dot product means smaller distance, hence the above interpretation fails in $\mathbb{R}^{m+1}$. To see this, consider two control points $\boldsymbol{p}_1^\alpha, \boldsymbol{p}_2^\alpha \in \mathbb{R}^{m+1}$ such that $\boldsymbol{p}_2^\alpha = t\boldsymbol{p}_1^\alpha$, with $t > 1$. Then for $\boldsymbol{x} = \boldsymbol{p}_1^\alpha$ we would have $\langle \boldsymbol{x}, \boldsymbol{p}_1^\alpha \rangle = \|\boldsymbol{p}_1^\alpha\|_2^2 < \langle \boldsymbol{x}, \boldsymbol{p}_2^\alpha \rangle = t\|\boldsymbol{p}_1^\alpha\|_2^2$; the dot product is smaller for $\boldsymbol{p}_1^\alpha$, the control point that is closer to $\boldsymbol{x}$, than for the much further away $\boldsymbol{p}_2^\alpha$ (see Figure 4.2). Therefore, the further away control point has a larger contribution than the closer point, which is at odds with the interpolation behaviour we desire. In general, the contribution of control points with larger norms will "dominate" the one of points with smaller norms. This has been observed for the attention mechanism in general by Demeter et al. (2020).

Fortunately, the domination of larger norm control points $\boldsymbol{p}_i^\alpha$ is not an issue if all control points have the same norm. In particular, if $\boldsymbol{x}$ and $\boldsymbol{p}_i^\alpha$ lie on the unit hypersphere $S^m{=}\{\boldsymbol{y}{\in}\mathbb{R}^{m+1} \mid \|\boldsymbol{y}\|_2{=}1\}$ then $\langle \boldsymbol{x}, \boldsymbol{p}_i^\alpha\rangle = \cos(\angle(\boldsymbol{x}, \boldsymbol{p}_i^\alpha))$ and it has the desired property that the closer $\boldsymbol{x}$ is to $\boldsymbol{p}_i^\alpha$, the higher their dot product. By doing this, we restrict $h_\circledast$ to be a function from the hypersphere S^m to $\mathbb{R}^{m+1}$. While this might seem artificial, modern transformer architectures do operate over hyperspheres as LayerNorm projects activations to S^m (Brody et al., 2023).

The central result of this section is that the functions in the form of Equation (4.5) can approximate any continuous function defined on the hypersphere, i.e., $\mathcal{H}_{\odot,m+1} = \bigcup_{N=1}^{\infty} \mathcal{H}_{\odot,m+1}^N$ is dense in $\mathcal{C}_{s,m}$ (Definition 4.7) and $\mathcal{H}_{\circledast,m+1} = \bigcup_{N=1}^{\infty} \mathcal{H}_{\circledast,m+1}^N$ is dense in $\mathcal{C}_{v,m}$ (Definition 4.8). Furthermore, we offer a Jackson-type approximation rate result which gives us a bound on the necessary prefix length N to achieve a desired approximation quality.

Theorem 4.12 (Jackson-type Bound for Universal Approximation on the Hypersphere)**.** *Let $f \in C(S^m)$ be a continuous function on S^m, $m \geq 8$ with modulus of continuity*

$$\omega(f;t) = \sup\left\{|f(\boldsymbol{x}) - f(\boldsymbol{y})| \;\middle|\; \boldsymbol{x}, \boldsymbol{y} \in S^m, \cos^{-1}(\langle \boldsymbol{x}, \boldsymbol{y}\rangle) \leq t\right\} \leq Lt,$$

for some $L > 0$. Then, for any $\epsilon > 0$, there exist $\boldsymbol{p}_1^\alpha, \ldots, \boldsymbol{p}_N^\alpha \in S^m$ and $p_1^\beta, \ldots, p_N^\beta \in \mathbb{R}$ such that

$$\sup_{x \in S^m} \left| f(\boldsymbol{x}) - \sum_{k=1}^{N} \exp(\lambda \langle \boldsymbol{x}, \boldsymbol{p}_k^\alpha\rangle) p_k^\beta \right| \leq \epsilon,$$

where $\lambda = \Lambda(\epsilon/2)$ with

$$\Lambda(\sigma) = \frac{(8LC_R + m\sigma + \sigma)\left(1 - \frac{\sigma^2}{8LC_H C_R + 2\sigma C_H}\right)^{\frac{\sigma}{4LC_R + \sigma}}}{\sigma\left(1 - \left(1 - \frac{\sigma^2}{8LC_H C_R + 2\sigma C_H}\right)^{\frac{2\sigma}{4LC_R + \sigma}}\right)} = \mathcal{O}\left(\frac{L^3 C_H}{\sigma^4}\right), \tag{4.6}$$

and any $N \geq N(\lambda, \epsilon)$ with

$$N(\lambda, \epsilon) = \Phi(m)\left(\frac{3\pi\left(L + \lambda\|f\|_\infty\right) c_{m+1}(\lambda)\exp(\lambda)}{\epsilon}\right)^{2(m+1)} = \mathcal{O}\left(\epsilon^{-10-14m-4m^2}\right), \tag{4.7}$$

with C_H being a constant depending on the smoothness of f (formally defined in the proof), C_R being a constant not depending on f or ϵ, $\Phi(m) = \mathcal{O}(m \log m)$ being a function that depends only on the dimension m and c_{m+1} being a normalization function.

Corollary 4.13. *$\mathcal{H}_{\odot,m+1}$ is dense in $\mathcal{C}_{s,m}$*

Proof. Theorem 4.12 holds for all $\epsilon > 0$ and Lemma 4.4. □

Theorem 4.12 is a Jackson-type result as Equation (4.7) gives the number N of control points needed to approximate f with accuracy ϵ. This corresponds to the length of the prefix sequence. Moreover, the smoother the target f is, i.e., the smaller L and C_H, the shorter the prefix length N. Thus, our construction uses only as much prefix positions as necessary.

The proof of Theorem 4.12 follows closely (Ng and Kwong, 2022). While they only provide a density result, we offer a Jackson-type bound which is non-trivial and may be of an independent interest. The idea behind the proof is as following. We first approximate f with its convolution with a kernel having the form of the terms in Equation (4.5):

$$\left(f * K_\lambda^{\mathrm{vMF}}\right)(\boldsymbol{x}) = \int_{S_m} c_{m+1}(\lambda) \exp(\lambda \langle \boldsymbol{x}, \boldsymbol{y} \rangle) f(\boldsymbol{y}) \, dw_m(\boldsymbol{y}). \tag{4.8}$$

The larger the λ is, the closer $f * K_\lambda^{\mathrm{vMF}}$ is to f and hence the smaller the approximation error (Menegatto, 1997). $\Lambda(\epsilon/2)$ gives the smallest value for λ such that this error is $\epsilon/2$. Equation (4.8) can then be approximated with sums: we partition S^m into N sets $V_1, \ldots, V_N$ small enough that f does not vary too much within each set. Each control point $\boldsymbol{p}_k^\alpha$ is placed in its corresponding V_k. Then, $\exp(\lambda \langle \boldsymbol{x}, \boldsymbol{y} \rangle) f(\boldsymbol{y})$ can be approximated with $\exp(\lambda \langle \boldsymbol{x}, \boldsymbol{p}_k^\alpha \rangle) f(\boldsymbol{p}_k^\alpha)$ when $\boldsymbol{y}$ is in the k-th set V_k. Hence, Equation (4.8) can be approximated with

$$\sum_{k=1}^{N} \exp(\lambda \langle \boldsymbol{x}, \boldsymbol{p}_k^\alpha \rangle) \, C f(\boldsymbol{p}_k^\alpha)$$

for some suitable constant C. By increasing N we can reduce the error of approximating the convolution with the sum. Equation (4.7) gives us the minimum N needed so that this error is $\epsilon/2$. Hence, we have error of at most $\epsilon/2$ from approximating f with the convolution and $\epsilon/2$ from approximating the convolution with the sum, resulting in our overall error being bounded by ϵ. The full proof is in Appendix B.2 and is illustrated in Figure B.2. The theorem can be extended to vector-valued functions in $\mathcal{C}_{v,m}$ with a multiplicative factor $1/\sqrt{m+1}$:

Corollary 4.14. *Let $f : S^m \to \mathbb{R}^{m+1}$, $m \geq 8$ be such that each component f_i satisfies the conditions in Theorem 4.12. Define $\|f\|_\infty = \max_{1 \leq i \leq m+1} \|f_i\|_\infty$. Then, for any $\epsilon > 0$, there exist $\boldsymbol{p}_1^\alpha, \ldots, \boldsymbol{p}_N^\alpha \in S^m$ and $\boldsymbol{p}_1^\beta, \ldots, \boldsymbol{p}_N^\beta \in \mathbb{R}^{m+1}$ such that*

$$\sup_{\boldsymbol{x} \in S^m} \left\| f(\boldsymbol{x}) - \sum_{k=1}^{N} \exp(\lambda \langle \boldsymbol{x}, \boldsymbol{p}_k^\alpha \rangle) \boldsymbol{p}_k^\beta \right\|_2 \leq \epsilon,$$

with $\lambda = \Lambda(\epsilon/2\sqrt{m+1})$ for any $N \geq N(\lambda, \epsilon/\sqrt{m+1})$. That is, $\mathcal{H}_{\circledast,m+1}$ is dense in $\mathcal{C}_{v,m}$ with respect to the $\|\cdot\|_2$ norm.

Thanks to Theorem 4.12 and Corollary 4.14, we know that functions in $\mathcal{C}_{v,m}$ can be approximated by core attention (Equation (4.4)). We only have to demonstrate that a core attention head can be represented as a classical attention head (Equation (4.1)). We do this by reversing the simplifications we made when constructing the core attention head.

Let's start by bringing the normalization term back, resulting in $\mathcal{H}^N_{\Downarrow,d}$, the split attention head hypothesis (Definition 4.11). Intuitively, $\sum_{k=1}^{n} \exp(\lambda\langle \boldsymbol{x}, \boldsymbol{p}_k^\alpha\rangle)$ is almost constant when the $\boldsymbol{p}_k^\alpha$ are uniformly distributed over the sphere as the distribution of distances from $\boldsymbol{x}$ to $\boldsymbol{p}_k^\alpha$ will be similar, regardless of where $\boldsymbol{x}$ lies. We can bound how far $\sum_{k=1}^{n} \exp(\lambda\langle \boldsymbol{x}, \boldsymbol{p}_k^\alpha\rangle)$ is from being a constant and adjust the approximation error to account for it. Appendix B.3 has the full proof.

Theorem 4.15. *Let $f : S^m \to \mathbb{R}^{m+1}$, $m \geq 8$ be such that each component f_i satisfies the conditions in Theorem 4.12. Then, for any $0 < \epsilon < 2\|f\|_\infty$, there exist $\boldsymbol{p}_1^\alpha, ..., \boldsymbol{p}_N^\alpha \in S^m$ such that*

$$\sup_{\boldsymbol{x}\in S^m} \left\| f(\boldsymbol{x}) - \frac{\sum_{k=1}^{N} \exp(\lambda\langle \boldsymbol{x}, \boldsymbol{p}_k^\alpha\rangle)\boldsymbol{p}_k^\beta}{\sum_{k=1}^{N} \exp(\lambda\langle \boldsymbol{x}, \boldsymbol{p}_k^\alpha\rangle)} \right\|_2 \leq \epsilon,$$

with

$$\lambda = \Lambda\left(\frac{2\epsilon L}{2L + \|f\|_\infty}\right)$$

$$\boldsymbol{p}_k^\beta = f(\boldsymbol{p}_k^\alpha),\ \forall k = 1, \dots, N,$$

for any $N \geq N(\lambda, \epsilon/\sqrt{m+1})$. That is, $\mathcal{H}_{\Downarrow,m+1}$ is dense in $\mathcal{C}_{v,m}$ with respect to the $\|\cdot\|_2$ norm.

An interesting observation is that adding the normalization term has not affected the asymptotic behavior of λ and hence also of the prefix length N. Furthermore, notice how the value $\boldsymbol{p}_i^\beta$ at the control point $\boldsymbol{p}_i^\alpha$ is simply $f(\boldsymbol{p}_k^\alpha)$, the target function evaluated at this control point.

We ultimately care about the ability of the classical attention head (Definition 4.5) to approximate functions in $\mathcal{C}_{v,m}$ by prefixing. Hence, we need to bring back the terms depending only on the input $\boldsymbol{x}$, combine $\boldsymbol{p}_k^\alpha$ and $\boldsymbol{p}_k^\beta$ parts into a single prefix $\boldsymbol{p}_k$ and bring back the $\boldsymbol{H}$ and $\boldsymbol{W}_V$ matrices. One can do this by considering an attention head with a hidden dimension $3(m+1)$ allowing us to place $\boldsymbol{x}, \boldsymbol{p}_k^\alpha$ and $\boldsymbol{p}_k^\beta$ in different subspaces of the embedding space. To do this, define a pair of embedding and projection operations:

$$\Pi : S^m \to \mathbb{R}^{3(m+1)} \qquad\qquad \Pi^{-1} : \mathbb{R}^{3(m+1)} \to \mathbb{R}^{m+1}$$

$$\boldsymbol{x} \mapsto \begin{bmatrix} \boldsymbol{I}_{m+1} \\ \boldsymbol{0}_{m+1} \\ \boldsymbol{0}_{m+1} \end{bmatrix} \boldsymbol{x} \qquad\qquad \boldsymbol{x} \mapsto \begin{bmatrix} \boldsymbol{I}_{m+1} \\ \boldsymbol{0}_{m+1} \\ \boldsymbol{0}_{m+1} \end{bmatrix}^\top \boldsymbol{x}.$$

Lemma 4.16. *$\Pi^{-1} \circ \mathcal{H}^1_{-,3(m+1)} \circ \Pi$ is dense in $\mathcal{H}_{\Downarrow,m+1}$, with the composition applied to each function in the class.*

Proof. We can prove something stronger. For all $f \in \mathcal{H}_{\Downarrow,m+1}$ there exists a $g \in \mathcal{H}^1_{-,3(m+1)}$ such that $f = \Pi^{-1} \circ g \circ \Pi$. If $f \in \mathcal{H}_{\Downarrow,m+1}$, then

$$f(\boldsymbol{x}) = \frac{\sum_{k=1}^{N} \boldsymbol{p}_k^\beta \exp(\lambda\langle \boldsymbol{x}, \boldsymbol{p}_k^\alpha\rangle)}{\sum_{k=1}^{N} \exp(\lambda\langle \boldsymbol{x}, \boldsymbol{p}_k^\alpha\rangle)},\ \forall \boldsymbol{x} \in S^m$$

for some $N, \lambda, \boldsymbol{p}_i^\alpha, \boldsymbol{p}_i^\beta$. Define:

$$\boldsymbol{p}_k = \begin{bmatrix} \mathbf{0} \\ \lambda \boldsymbol{p}_k^\alpha \\ \boldsymbol{p}_k^\beta \end{bmatrix} \in \mathbb{R}^{3(m+1)},$$
$$\boldsymbol{H} = \begin{bmatrix} M\boldsymbol{I} & \boldsymbol{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix}, \boldsymbol{W}_V = \begin{bmatrix} \mathbf{0} & \mathbf{0} & \boldsymbol{I} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix} \in \mathbb{R}^{3(m+1)\times 3(m+1)},$$

With M a negative constant tending to $-\infty$. Then:

$$g(\boldsymbol{x}) = \frac{\sum_{i=1}^{N} \exp(\boldsymbol{x}^\top \boldsymbol{H} \boldsymbol{p}_i) \boldsymbol{W}_V \boldsymbol{p}_i + \exp(\boldsymbol{x}^\top \boldsymbol{H} \boldsymbol{x}) \boldsymbol{W}_V \boldsymbol{x}}{\sum_{i=1}^{N} \exp(\boldsymbol{x}^\top \boldsymbol{H} \boldsymbol{p}_i) + \exp(\boldsymbol{x}^\top \boldsymbol{H} \boldsymbol{x})},$$

is in $\mathcal{H}^1_{-,3(m+1)}$ and $f = \Pi^{-1} \circ g \circ \Pi$. As this holds for all $f \in \mathcal{H}_{\Downarrow\Uparrow,m+1}$, it follows that $\mathcal{H}_{\Downarrow\Uparrow,m+1} \subset \Pi^{-1} \circ \mathcal{H}^1_{-,3(m+1)} \circ \Pi$. Hence, $\Pi^{-1} \circ \mathcal{H}^1_{-,3(m+1)} \circ \Pi$ is dense in $\mathcal{H}_{\Downarrow\Uparrow,m+1}$. □

Lemma 4.16 shows that every split attention head can be *exactly* represented as 3 times bigger classical attention head. Note that our choice for $\boldsymbol{H}$ and $\boldsymbol{W}_V$ is not unique. Equivalent constructions are available by multiplying each component by an invertible matrix, effectively changing the basis. Finally, the embedding and projection operations can be represented as MLPs and hence can be embedded in a transformer architecture. Now, we can provide the final result of this section, namely that the standard attention head of a transformer can approximate any vector-valued function on the hypersphere:

Theorem 4.17. *Let $f : S^m \to \mathbb{R}^{m+1}$, $m \geq 8$ be such that each component f_i satisfies the conditions in Theorem 4.12. Then, for any $0 < \epsilon \leq 2\|f\|_\infty$, there exists an attention head $h \in \mathcal{H}^{N,1}_{-,3(m+1)}$ such that*

$$\sup_{\boldsymbol{x} \in S^m} \left\| f(\boldsymbol{x}) - (\Pi^{-1} \circ h \circ \Pi)(\boldsymbol{x}) \right\|_2 \leq \epsilon, \tag{4.9}$$

for any $N \geq N(\lambda, \epsilon/\sqrt{m+1})$. That is, $\Pi^{-1} \circ \mathcal{H}^1_{-,3(m+1)} \circ \Pi$ is dense in $\mathcal{C}_{v,m}$ with respect to the $\|\cdot\|_2$ norm.

Proof. The density result follows directly from Theorem 4.15 and Lemma 4.16 and transitivity (Lemma 4.2). The Jackson bound is the same as in Theorem 4.15 as transforming the split attention head to a classical attention head is exact and does not contribute further error. □

Therefore, we have shown that a single attention head with a hidden dimension $3(m + 1)$ can approximate any continuous function $f : C(S^m) \to \mathbb{R}^{m+1}$ to an arbitrary accuracy. This is for *fixed* pretrained components, that is, $\boldsymbol{H}$ and $\boldsymbol{W}_V$ are as given in the proof of Lemma 4.16

and depend neither on the input $\boldsymbol{x}$ nor on the target function f. Therefore, the behavior of the attention head is fully controlled by the prefix. This is a Jackson-type result, with the length N of the prefix given in Theorem 4.15. To the best of our knowledge, Theorem 4.17 is the first bound on the necessary prefix length to achieve a desired accuracy of function approximation using an attention head. Most critically, Theorem 4.17 demonstrates that attention heads are more expressive than commonly thought. A *single* attention head with a very simple structure can be a universal approximator.

4.4 Universal approximation of sequence-to-sequence functions

The previous section showed how we can approximate any continuous $f : S^m \to \mathbb{R}^{m+1}$ with a single attention head. Still, one typically uses the transformer architecture for operations over sequences rather than over single inputs (the case with $T \geq 1$). We will now show how we can leverage Theorem 4.17 to model general sequence-to-sequence functions. First, we show the simpler case of functions that apply the exact same mapping to all inputs. We then show how to model general sequence-to-sequence functions using a variant of the Kolmogorov–Arnold theorem.

Element-wise functions Theorem 4.17 can be extended to element-wise functions where the exact same function is applied to each element in the input sequence, i.e., the concept class $\mathcal{C}_{\|,T,m}$ from Definition 4.10. If $f \in \mathcal{C}_{\|,T,m}$, then there exists a $g \in \mathcal{C}_{v,m}$ such that $f(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T) = (g(\boldsymbol{x}_1), \ldots, g(\boldsymbol{x}_T))$. By Theorem 4.17, there exists a prefix $\boldsymbol{p}_1, ..., \boldsymbol{p}_N$ that approximates g. As the construction in Lemma 4.16 prevents interactions between two different inputs $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$, an attention head $h^T \in \mathcal{H}^{N,T}_{-,3(m+1)}$ for a T-long input (Equation (4.1)) with the exact same prefix $\boldsymbol{p}_1, \ldots, \boldsymbol{p}_N$ approximates f:

Corollary 4.18. $\Pi^{-1} \circ \mathcal{H}^{T}_{-,3(m+1)} \circ \Pi$ *is dense in* $\mathcal{C}_{\|,T,m}$ *with respect to the* $\|\cdot\|_2$ *norm applied element-wise. That is, for every* $\epsilon > 0$*, there exists* $h^T \in \mathcal{H}^{N,T}_{-,3(m+1)}$ *such that:*

$$\sup_{\{\boldsymbol{x}_i\} \in (S^m)^T} \max_{1 \leq k \leq T} \left\| \left[f(\{\boldsymbol{x}_i\}) - \left(\Pi^{-1} \circ h^T \circ \Pi \right) (\{\boldsymbol{x}_i\}) \right]_k \right\|_2 \leq \epsilon,$$

with Π *and* Π^{-1} *applied element-wise,* $[\cdot]_k$ *selecting the k-th element, and approximate rate bound on N as in Theorem 4.17.*

General sequence-to-sequence functions Ultimately, we are interested in modeling arbitrary functions from sequences of inputs $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T)$ to sequences of outputs $(\boldsymbol{y}_1, \ldots, \boldsymbol{y}_T)$, that is, the $\mathcal{C}_{T,m}$. We will use a version of the Kolmogorov–Arnold representation Theorem. The Theorem is typically defined on functions over the unit hypercube $[0, 1]^m$. As there exists a homeomorphism between $[0, 1]^m$ and a subset of S^m (Lemma B.4.1), for simplicity,

we will ignore this technical detail. Our construction requires only $T+2$ attention layers, each with a single head.

The original Kolmogorov-Arnold representation theorem (Kolmogorov, 1957) identifies every continuous function $f:[0,1]^d \to \mathbb{R}$ with univariate functions g_q, $\psi_{p,q}$ such that:

$$f(x_1,\dots,x_d) = \sum_{q=0}^{2d} g_q\left(\sum_{p=1}^{d} \psi_{p,q}(x_p)\right).$$

In other words, multivariate functions can be represented as sums and compositions of univariate functions. As transformers are good at summing and attention heads are good at approximating functions, they can approximate functions of this form. However, g_q and $\psi_{p,q}$ are generally not well-behaved (Girosi and Poggio, 1989), so we will use the construction by Schmidt-Hieber (2021) instead.

Lemma 4.19 (Theorem 2 in (Schmidt-Hieber, 2021))**.** *For a fixed d, there exists a monotone functions $\psi:[0,1]\to C$ (the Cantor set) such that for any function $f:[0,1]^d \to \mathbb{R}$, we can find a function $g: C\to\mathbb{R}$ such that*

1. $f(x_1,\dots,x_d) = g\left(3\sum_{p=1}^{d} 3^{-p}\,\psi(x_p)\right),$ (4.10)
2. *if f is continuous, then g is also continuous,*
3. *if $|f(\boldsymbol{x})-f(\boldsymbol{y})| \le Q\|\boldsymbol{x}-\boldsymbol{y}\|_\infty$, for all $\boldsymbol{x},\boldsymbol{y}\in[0,1]^d$ and some Q, then $|g(x)-g(y)| \le 2Q, \forall x,y\in C$.*

In comparison with the original Kolmogorov–Arnold theorem, we need a single inner function ψ which does not depend on the target function f and only one outer function g. Furthermore, both ψ and g are Lipschitz. Hence, we can approximate them with our results from Section 4.3.

We need to modify Lemma 4.19 a bit to make it fit the sequence-to-sequence setting. First, flatten a sequence of T $(m+1)$-dimensional vectors into a single vector in $[0,1]^{(m+1)T}$. Second, define $\Psi_d:[0,1]^d\to\mathbb{R}^d$ to be the element-wise application of ψ: $\Psi_d(\{x_i\}_{i=1}^d) = (\psi(x_i))_{i=1}^d$. We can also define $G_i: C\to\mathbb{R}^{m+1}$, $i=1,\dots,T$ and extend Equation (4.10) for our setting:

$$\begin{aligned} f(\boldsymbol{x}_1,\dots,\boldsymbol{x}_T) &= (G_1(R),\dots,G_T(R)), \text{ with} \\ R &= 3\sum_{i=1}^{T} 3^{-(i-1)(m+1)} \sum_{p=1}^{m+1} 3^{-p}\psi(\boldsymbol{x}_{i,p}) \\ &= 3\sum_{i=1}^{T} 3^{-(i-1)(m+1)} \begin{bmatrix} 3^{-1} \\ \vdots \\ 3^{-(m+1)} \end{bmatrix}^\top \Psi_{m+1}(\boldsymbol{x}_i). \end{aligned} \tag{4.11}$$

Equation (4.11) can now be represented with a transformer with $T+2$ attention layers. Ψ_{m+1} is applied element-wise, hence, all $\Psi_{m+1}(\boldsymbol{x}_i)$ can be computed in parallel with a single

attention head (Corollary 4.18). The dot product with the $\begin{bmatrix}3^{-1} & \cdots & 3^{-(m+1)}\end{bmatrix}$ vector can be computed using a single MLP. The product with the $3^{-(i-1)(m+1)}$ scalar is a bit more challenging as it depends on the position in the sequence. However, if we concatenate position encodings to the input, another MLP can use them to compute this factor and the multiplication. The outer sum over the T inputs and the multiplication by 3 can be achieved with a single attention head. Hence, using only 2 attention layers, we have compressed the whole sequence in a single scalar R.[1]

The only thing left is to apply $G_1, \ldots, G_T$ to R to compute each of the T outputs. As each one of these is Lipschitz, we can approximate each with a single attention head using Theorem 4.17. Each G_i is different and would need its own set of prefixes, requiring T attention heads arranged in T attention layers. Using the positional encodings, each layer can compute the output for its corresponding position and pass the input unmodified for the other positions. The overall prefix size would be the longest of the prefixes necessary to approximate $\Psi_{m+1}, G_1, \ldots, G_T$. Hence, we have constructed an architecture that approximates any sequence-to-sequence function $f \in \mathcal{C}_{T,m}$ with only $T+2$ attention layers. Thus, $\mathcal{H}^T_{\equiv,d}$ is dense in $\mathcal{C}_{T,m}$.

There is also an equivalent construction with fixed depth of 3 attention layers where the last layer would need to have T attention heads. As $G_1, \ldots, G_T$ can be applied in parallel, they can be packed as separate heads in a single layer. Therefore, we can trade depth for number of layers.

One can also construct a 3-layer single-head model by defining a $G^\star$ that combines all $G_1, \ldots, G_T$:

$$G^\star(R) = \begin{cases} G_1(R) & \text{if } \pi = 1 \\ G_2(R) & \text{if } \pi = 2 \\ & \vdots \end{cases},$$

where π is the positional encoding at this token. As $G_1, \ldots, G_T$ are all Lipschitz, there exists such a $G^\star(R)$ that is also Lipschitz, although its Lipschitz constant can be arbitrarily large. By setting the single head of the third layer to approximate $G^\star(R)$, we get universal in-context approximation for sequence-to-sequence functions in three attention layers each with a single head.

4.5 Discussion and conclusions

Comparison with prior work Just like us, Wang et al. (2023) show that prefix-tuning can be a universal approximator. Their approach relies on discretizing the input space and the set of sequence-to-sequence functions to a given precision depending on ϵ, resulting in a finite number of pairs of functions and inputs, each having a unique corresponding output. Then, using the results of Yun et al. (2020), they construct a *meta-transformer* which maps

[1] Yun et al. (2020) use a similar approach but use discretization to enumerate all possible sequences and require $\mathcal{O}(\epsilon^{-m})$ attention layers. In our continuous setting, R is computed with 2 layers.

each of the function-input pairs to their corresponding output. This approach has several limitations: *i)* the model has exponential depth $\mathcal{O}(T\epsilon^{-m})$; *ii)* reducing the approximation error ϵ requires increasing the model depth; *iii)* the prefix length is fixed, hence a constant function and a highly non-smooth function would have equal prefix lengths, and *iv)* it effectively has memorized all possible functions and inputs, explaining the exponential size of their constructions. In contrast, we show that memorization is not needed: attention heads are naturally suited for universal approximation. Section 4.4 showed that $T+2$ layers are enough, we require shorter prefixes for more smooth functions and reducing the approximation error ϵ can be done by increasing the prefix length, without modifying the pretrained model.

Petrov et al. (2024c) have shown that prefix-tuning cannot change the relative attention patterns over the input tokens and hence cannot learn tasks with new attention patterns. This appears to be a limitation but von Oswald et al. (2023a) and Akyürek et al. (2022) proved that there exist attention heads that can learn any linear model, samples of which are given as a prefix. In this work, we showed the existence of a "universal" attention head ($\boldsymbol{H}$ and $\boldsymbol{W}_V$ in Lemma 4.16) that can be used to emulate any new function defined as a prefix. This reconciles the apparent contradiction between (Petrov et al., 2024c) and this work. While a prefix indeed cannot change the relative attention pattern over the input tokens, this is not needed for universal approximation. If a suitable for interpolation description of the target function is provided in-context, it suffices that the input tokens attend appropriately to this description rather than to themselves.

Prefixes have been observed to have larger norms than token embeddings (Bailey et al., 2023). Our results provide an explanation to that. While the control points $\boldsymbol{p}_k^\alpha$ are in S^m and hence have norm 1, in Lemma 4.16 we fold λ into them. Recall that the less smooth f is, the higher the concentration parameter λ has to be in order to reduce the influence of one control point on the locations far from it. Hence, the less smooth f is, the larger the norm of the prefixes.

Connection to prompting and safety implications While this work focused on prefix-tuning, the results can extend to prompting. Observe that prefix tuning (where we have a distinct prefix) can be reduced to soft prompting (where only the first layer is prefixed) by using an appropriate attention mechanism and position embeddings. Hence, if a function $f \in \mathcal{C}_{T,m}$ requires N prefixes to be approximated to precision ϵ with prefix-tuning, it would require $\mathcal{O}(TN)$ soft tokens to be approximated with soft prompting. Finally, observe that a soft token can be encoded with a sequence of hard tokens, the number of hard tokens per soft token depends on the required precision and the vocabulary size V. Hence, f could be approximated with $\mathcal{O}(\log_V(\epsilon^{-1})mTN)$ hard tokens. Therefore, our universal approximation results may translate to prompting. This raises concerns as to whether it is at all possible to prevent a transformer model from exhibiting undesirable behaviors (Zou et al., 2023; Wolf et al., 2023; Chao et al., 2023). Furthermore, this means that transformer-based agents might have the technical possibility to collude in undetectable and uninterpretable manner

(de Witt et al., 2023). Still, our results require specific form of the attention and value matrices and, hence, it is not clear whether these risk translate to real-world models.

Prefix-tuning and prompting a pretrained transformer might be less efficient than training it Typically, with neural networks one expects that the number of trainable parameters would grow as $\mathcal{O}(\epsilon^{-m})$ (Schmidt-Hieber, 2021). Indeed that is the case for universal approximation with a transformer when one learns the key, query and value matrices and the MLP parameters as shown by Yun et al. (2020). However, as Equation (4.7) shows, our construction results in the trainable parameters (prefix length in our case) growing as $\mathcal{O}(\epsilon^{-10-14m-4m^2})$. The m^2 term indicates worse asymptotic efficiency of prefix-tuning and prompting compared to training a transformer. However, our approach may not be tight. Thus, it remains an open question if a tighter Jackson bound exists or if prefix-tuning and prompting inherently require more trainable parameters to reach the same approximation accuracy as training a transformer.

Prefix-tuning and prompting may work by combining prefix-based element-wise maps with pretrained cross-element mixing The construction for general sequence-to-sequence functions in Section 4.4 is highly unlikely to occur in transformers pretrained on real data as it requires very specific parameter values. While the element-wise setting (Corollary 4.18) is more plausible, it cannot approximate general sequence-to-sequence functions. Hence, neither result explains why prefix-tuning works in practice. To this end, we hypothesise that prompting and prefix-tuning, can modify how single tokens are processed (akin to fine-tuning only MLPs), while the cross-token information mixing happens with pretrained attention patterns. Therefore, prompting and prefix-tuning can easily learn novel tasks as long as no new attention patterns are required. Our findings suggest a method for guaranteeing that a pretrained model possesses the capability to act as a token-wise universal approximator. This can be achieved by ensuring each layer of the model includes at least one attention head conforming to the structure in Lemma 4.16.

Limitations. We assume a highly specific pretrained model which is unlikely to occur in practice when pretraining with real-world data. Hence, the question of, given a real-world pretrained transformer, which is the class of functions it can approximate with prefix-tuning is still open. This is an inverse (Bernstein-type, Jiang et al. 2023) bound and is considerably more difficult to derive.

Author contributions

This work was a collaboration with my supervisors, Dr. Adel Bibi and Prof. Philip H.S. Torr. The initial idea to follow up on our previous work (Chapter 3) from a universal approximation angle was mine. I observed and proved the theoretical results. Adel helped

streamline some of the proofs. Both Adel and Phil provided guidance and advice on how to approach, structure and present the project.

Acknowledgements

We would like to thank Tom Lamb for spotting several mistakes and helping us rectify them. This work is supported by a UKRI grant Turing AI Fellowship (EP/W002981/1) and the EPSRC Centre for Doctoral Training in Autonomous Intelligent Machines and Systems (EP/S024050/1). AB has received funding from the Amazon Research Awards. We also thank the Royal Academy of Engineering and FiveAI.

Chapter 5

Universal In-Context Approximation by Prompting Fully Recurrent Models

Abstract

Zero-shot and in-context learning enable solving tasks without model fine-tuning, making them essential for developing generative model solutions. Therefore, it is crucial to understand whether a pretrained model can be prompted to approximate any function, i.e., whether it is a universal in-context approximator. While it was recently shown that transformer models do possess this property, these results rely on their attention mechanism. Hence, these findings do not apply to fully recurrent architectures like RNNs, LSTMs, and the increasingly popular SSMs. We demonstrate that RNNs, LSTMs, GRUs, Linear RNNs, and linear gated architectures such as Mamba and Hawk/Griffin can also serve as universal in-context approximators. To streamline our argument, we introduce a programming language called LSRL that compiles to these fully recurrent architectures. LSRL may be of independent interest for further studies of fully recurrent models, such as constructing interpretability benchmarks. We also study the role of multiplicative gating and observe that architectures incorporating such gating (e.g., LSTMs, GRUs, Hawk/Griffin) can implement certain operations more stably, making them more viable candidates for practical in-context universal approximation.

5.1 Introduction

Until recently, solving a task with machine learning required training or fine-tuning a model on a dataset matching the task at hand. However, large foundation models exhibit the ability to solve new tasks without being specifically fine-tuned or trained for them: often it is

sufficient to simply prompt them in the right way. This has made prompting a key method for steering a model towards a specific behaviour or task (Liu et al., 2023). Prompting has been especially successful because of *in-context learning*: the ability to modify the model's behavior with information provided within the input sequence, without changing the underlying model parameters (Brown et al., 2020). As a result, the art and skill of constructing a successful prompt (*prompt engineering*) has become extremely important (Liu and Chilton, 2022; Sahoo et al., 2024). Yet, we know little about the theoretical properties of prompting. It is not even clear if there are limits to what can be achieved with prompting or, conversely, whether it is possible to prompt your way into any behaviour or task.

This can be framed as a universal approximation question. Classically, universal approximation results show how a class of tractable functions, such as neural networks, approximates another class of concept functions, e.g., all continuous functions on a bounded domain, with arbitrary accuracy. This is often done by showing that one can choose *model parameters* that approximate the target function. However, in-context learning poses a different challenge as the model parameters are *fixed*. Instead, a part of the input (the prompt) is modified to cause the model to approximate the target function. Hence, we define universal *in-context* approximation to be the property that there exist fixed weights such that the resulting model can be prompted to approximate any function from a concept class. Understanding whether a model can be a universal *in-context* approximator is especially important as most commercial models are accessible exclusively via a prompting interface (La Malfa et al., 2024).

In-context learning has been almost exclusively studied in conjunction with the transformer architecture (Vaswani et al., 2017). This is likely because in-context abilities appear once the models are large enough (Wei et al., 2021) and most large models have been transformer-based. On the subject of universal in-context approximation, Wang et al. (2023) were first to show that a transformer possesses this property by discretising and memorising all possible functions in the model weights. Memorisation is not needed, though, and even small transformers can be universal approximators when prompted (Petrov et al., 2024b). Both results, however, critically depend on the attention mechanism of the transformer architecture (Bahdanau et al., 2015).

Still, generative models are not restricted to attention-based architectures: there are the "classic" recurrent neural networks (RNNs, Amari, 1972), long short-term memory models (LSTMs, Hochreiter and Schmidhuber, 1997) and gated recurrent units (GRUs, Cho et al., 2014). Recently, Linear RNN models (also known as state-space models or SSMs) were proposed as a scalable alternative to the transformer architecture (Orvieto et al., 2023; Fu et al., 2023a) and have started to outperform similarly-sized transformers when multiplicative gating is added (Gu and Dao, 2023; De et al., 2024; Botev et al., 2024). Furthermore, despite in-context learning being associated with the transformer, recent empirical results show in-context learning in SSMs, RNNs, LSTMs and even convolutional models (Xie et al., 2021; Akyürek et al., 2024; Lee et al., 2024).

Yet, despite their ability to be in-context learners, there is little known about the theoretical

properties of these fully recurrent architectures. As these architectures become more and more widely used, understanding their in-context approximation abilities is increasingly more important for their safety, security and alignment. We show that, in fact, many of these architectures, similarly to transformers, can be universal in-context approximators. Concretely, our contributions are as follows:

1. We develop *Linear State Recurrent Language* (LSRL): a programming language that compiles to different fully recurrent models. Programming in LSRL is akin to "thinking like a recurrent model". LSRL programs can then be implemented exactly as model weights.

2. Using LSRL, we construct Linear RNN models that can be prompted to act as any token-to-token function over finite token sequences, or to approximate any continuous function. These results also hold for RNNs, LSTMs, GRUs and Hawk/Griffin models (De et al., 2024).

3. We present constructions with and without multiplicative gating. However, we observe that the constructions without these gates depend on numerically unstable conditional logic.

4. Nevertheless, we show that multiplicative gates lead to more compact and numerically stable models, making it more likely that universal in-context approximation properties arise in models utilising them, such as LSTMs, GRUs and the latest generation of Linear RNNs.

5.2 Preliminaries

Fully recurrent architectures. In this work, we focus exclusively on fully recurrent neural network architectures. Recurrent models operate over sequences. Concretely, consider an input sequence $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N)$ with $\boldsymbol{x}_t \in \mathcal{X}$, $\mathcal{X}$ being some input space. We will refer to the elements of the input sequence as *tokens* even if they are real-valued vectors. A recurrent model $g : \mathcal{X}^\star \to \mathcal{Y}$ maps a sequence of inputs to an output in some output space $\mathcal{Y}$. These models are always causal, namely:

$$\boldsymbol{y}_t = g(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_t). \tag{5.1}$$

We will abuse the notation and refer to $(\boldsymbol{y}_1, \ldots, \boldsymbol{y}_t) = (g(\boldsymbol{x}_1), \ldots, g(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_t))$ as simply $g(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_t)$. We will also separate the input sequence into a query $(\boldsymbol{q}_1, \ldots, \boldsymbol{q}_n)$ and a prompt $(\boldsymbol{p}_1, \ldots, \boldsymbol{p}_N)$. The prompt specifies the target function f that we approximate while the query designates the input at which we evaluate it. Contrary to the typical setting, we will place the query before the prompt.[1]

[1] That is necessitated by the limited capacity of the state variables. As the model is fixed, in order to increase the precision of the approximation, we can only increase the prompt length. If the prompt is before

There are various neural network architectures that fall under the general framework of Equation (5.1). The quintessential one is the RNN. It processes inputs one by one with only a non-linear state being passed from one time step to the other. A model g can thus be stacked RNN layers, each one being:

$$\begin{aligned} \boldsymbol{s}_t &= \sigma(\boldsymbol{A}\boldsymbol{s}_{t-1} + \boldsymbol{B}\boldsymbol{x}_t + \boldsymbol{b}), \\ \boldsymbol{y}_t &= \phi(\boldsymbol{s}_t), \end{aligned} \qquad \text{(Classic RNN)} \qquad (5.2)$$

with $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{b}$ and the initial state value $\boldsymbol{s}_0$ being model parameters, σ a non-linear activation function and ϕ a multi-layer perceptron (MLP) with `ReLU` activations. We assume that σ is always a `ReLU` to keep the analysis simpler. The non-linearity in the state update can make the model difficult to train due to vanishing and exploding gradients (Bengio et al., 1994). Therefore, Linear RNNs have been proposed as regularizing the eigenvalues of $\boldsymbol{A}$ can stabilise the training dynamics (Orvieto et al., 2023). Linear RNNs also admit a convolutional representation, making them trainable in parallel (Gu et al., 2021; Fu et al., 2023a). Linear RNNs drop the non-linearity from the state update in Equation (5.2):

$$\begin{aligned} \boldsymbol{s}_t &= \boldsymbol{A}\boldsymbol{s}_{t-1} + \boldsymbol{B}\boldsymbol{x}_t + \boldsymbol{b}, \\ \boldsymbol{y}_t &= \phi(\boldsymbol{s}_t). \end{aligned} \qquad \text{(Linear RNN)} \qquad (5.3)$$

The fully linear state updates do not affect the expressivity of the models, as non-linear activations are nevertheless present in the MLP layers ϕ between the linear state update layers (Boyd and Chua, 1985; Wang and Xue, 2023). The state-of-the-art Linear RNN models also utilise some form of multiplicative gating (Gu and Dao, 2023; De et al., 2024; Botev et al., 2024). While specific implementations can differ, we can abstract it as the following Gated Linear RNN architecture:

$$\begin{aligned} \boldsymbol{s}_t &= \boldsymbol{A}\boldsymbol{s}_{t-1} + \boldsymbol{B}\boldsymbol{x}_t + \boldsymbol{b}, \\ \boldsymbol{y}_t &= \gamma(\boldsymbol{x}_t) \odot \phi(\boldsymbol{s}_t), \end{aligned} \qquad \text{(Gated Linear RNN)} \qquad (5.4)$$

with γ being another MLP and $\odot$ being the element-wise multiplication operation (Hadamard product). Equation (5.4) encompasses a range of recently proposed models. For example, one can show that any model consisting of L stacked Gated Linear RNN layers, with γ and ϕ with k layers, can be represented as a $L(k+2)$-layer Hawk or Griffin model (De et al., 2024). The conversions are described in detail in Appendix C.5. We can similarly add multiplicative gating to the classic RNN architecture:

$$\begin{aligned} \boldsymbol{s}_t &= \sigma(\boldsymbol{A}\boldsymbol{s}_{t-1} + \boldsymbol{B}\boldsymbol{x}_t + \boldsymbol{b}), \\ \boldsymbol{y}_t &= \gamma(\boldsymbol{x}_t) \odot \phi(\boldsymbol{s}_t), \end{aligned} \qquad \text{(Gated RNN)} \qquad (5.5)$$

the query, it would have to be compressed into a fixed-size state, limiting the approximation precision even with increased prompt lengths. But if the query has a fixed size, it can be stored in a fixed-size state variable exactly.

Equation (5.5) may appear unusual but it is related to the well-known GRU (Cho et al., 2014) and LSTM (Hochreiter and Schmidhuber, 1997) architectures. Same as the case with Griffin/Hawk, any Gated RNN can be represented as a $L(k+2)$-layer GRU or LSTM model (details in Appendices C.3 and C.4). As a result, if there exists a Gated RNN model that is a universal in-context approximator (which we later show to be the case), then there also exist GRU and LSTM models with the same property.

All the models above can be boiled down to compositions of a few building blocks. Namely, linear layers, `ReLU` activations, (non-)linear state updates and multiplicative operations (in the case of gated models). These four building blocks will be the primitives of LSRL, the programming language we introduce in Section 5.3 as a tool to write programs that directly compile to these architectures. In practice, a number of additional elements might be present such as residual connections (He et al., 2016), positional embeddings (Su et al., 2024) and normalisation layers (Ba et al., 2016; Zhang and Sennrich, 2019). However, as these are not necessary for showing the in-context universal approximation abilities of the four architectures above, we will not consider them in this work.

Theoretical understanding of in-context learning. Beyond the question of universal in-context approximation, there have been attempts to theoretically understand in-context learning from various perspectives. The ability to learn linear functions and perform optimization in-context has been extensively explored in the context of linear regression (Garg et al., 2022; Akyürek et al., 2022; von Oswald et al., 2023a; Fu et al., 2023b; Zhang et al., 2023; Ahn et al., 2023), kernel regression (Han et al., 2023) and dynamical systems (Li et al., 2023). Furthermore, studies have explored how in-context learning identifies and applies the appropriate pretraining skill (Xie et al., 2021; Coda-Forno et al., 2023; Bai et al., 2023b). It has also been shown that transformers can construct internal learning objectives and optimize them during the forward pass (von Oswald et al., 2023b; Dai et al., 2023). However, these studies almost exclusively focus on the transformer architecture, and the applicability of their findings to fully recurrent models remains unclear.

Approximation theory. Let $\mathcal{X}$ and $\mathcal{Y}$ be normed vector spaces. Take a set of functions $\mathcal{C} \subseteq \mathcal{Y}^{\mathcal{X}}$ from $\mathcal{X}$ to $\mathcal{Y}$ called a *concept space*. Take also a set of nicely behaved functions $\mathcal{H} \subset \mathcal{Y}^{\mathcal{X}}$, called *hypothesis space*. $\mathcal{H}$ could be any set that we have tools to construct and analyse, e.g., all polynomials or all neural networks of a particular architectural type. Approximation theory is concerned with how well functions in $\mathcal{H}$ approximate functions in $\mathcal{C}$. We say that $\mathcal{H}$ *universally approximates* $\mathcal{C}$ over a compact domain $\mathcal{D}$ (or that $\mathcal{H}$ *is dense in* $\mathcal{C}$) if for every $f \in \mathcal{C}$ and $\epsilon > 0$ there exist a $h \in \mathcal{H}$ such that $\sup_{x \in \mathcal{D}} |f(x) - h(x)| \leq \epsilon$. There is a long history of studying the concept class of continuous functions and hypothesis classes of single hidden layer neural networks (Cybenko, 1989; Barron, 1993) or deeper models (Hornik et al., 1989; Telgarsky, 2015). The concept class of sequence-to-sequence functions has been shown to be universally approximated with the hypothesis classes of transformers (Yun et al., 2020), RNNs (Schäfer and Zimmermann, 2007) and Linear RNNs (Wang and

Xue, 2023).

The hypothesis spaces in this work are different. The model is fixed and only the prompt part of the input is changed, i.e., all learnable parameters are in the prompt. Take a recurrent model g as in Equation (5.1) with *fixed* model parameters and a query length n. The hypothesis class is all functions that result by calling g on the user query followed by the prompt and taking the last n' outputs:

$$\mathcal{H}_g^{\mathcal{D}^n} = \{(\boldsymbol{q}_1, \dots, \boldsymbol{q}_n) \mapsto g(\boldsymbol{q}_1, \dots, \boldsymbol{q}_n, \boldsymbol{p}_1, \dots, \boldsymbol{p}_N)[-n':] \mid \forall \boldsymbol{p}_i \in \mathcal{D}, N > 0\}. \tag{5.6}$$

The domain $\mathcal{D}$ of $\boldsymbol{p}_i$ and $\boldsymbol{q}_i$ can be continuous embeddings in $\mathbb{R}^d$ or discrete tokens $\mathcal{V} = \{1, ..., V\}$.

Note that each $h \in \mathcal{H}_g$ is identified by a prompt $(\boldsymbol{p}_1, ..., \boldsymbol{p}_N)$ but is a function with domain all possible queries $(\boldsymbol{q}_1, ..., \boldsymbol{q}_n)$. Therefore, finding a hypothesis $h \in \mathcal{H}_g$ that approximates a target function f is equivalent to finding the prompt of that hypothesis. The approximation properties of $\mathcal{H}_g$ in Equation (5.6) depend on the architecture of g, as well as its specific parameters. This makes it challenging to do approximation in the context window. The possibilities for interaction between the inputs are limited and the effects of the fixed model weights can be difficult to study (Petrov et al., 2024c). To the best of our knowledge, this has only been studied in the case where g is a transformer model. Wang et al. (2023) showed that in-context universal approximation is possible with a transformer by discretizing and memorising all possible functions in the model weights, while, Petrov et al. (2024b) argue that no memorisation is needed and that a transformer with $n + 2$ layers can be a universal approximator for sequence-to-sequence functions with input length n with a prompt of length $\mathcal{O}(\epsilon^{-10-14d-4d^2})$.

We study the recurrent architectures in Equations (5.2) to (5.5) and their ability to approximate continuous functions over real-valued vectors and to represent discrete maps over tokens (which corresponds to how language models are used in practice). We consider the following classes of functions. $\mathcal{C}^{\text{vec}} = (\mathbb{R}^{d_{\text{out}}})^{[0,1]^{d_{\text{in}}}}$ contains all continuous functions from the unit hypercube to $\mathbb{R}^{d_{\text{out}}}$, while $\mathcal{C}^{\text{tok}} = \{h \in (\mathcal{V}^l)^{\mathcal{V}^l} \mid h \text{ causal}\}$ all causal functions from l tokens to l tokens. The hypothesis classes are $\mathcal{H}^{\text{vec}}(g)$ corresponding to Equation (5.6) with $D = [0, 1]^{d_{\text{in}}}, n = n' = 1$ and g some *fixed* model of one of the four architectures in Equations (5.2) to (5.5), and $\mathcal{H}^{\text{tok}}(g)$ with $D = \mathcal{V}$ and $n = n' = l$.

5.3 Linear State Recurrent Language (LSRL)

We can construct the weights for universal in-context models with the architectures in Equations (5.2) to (5.5) by hand but this is labour-intensive, error-prone, difficult to interpret, and the specific weights would be architecture-dependent. Working at such a low level of abstraction can also obfuscate common mechanisms and design patterns, making it more difficult to appreciate both the capabilities and the constraints of fully recurrent architectures. Instead, we propose a new programming language: *Linear State Recurrent*

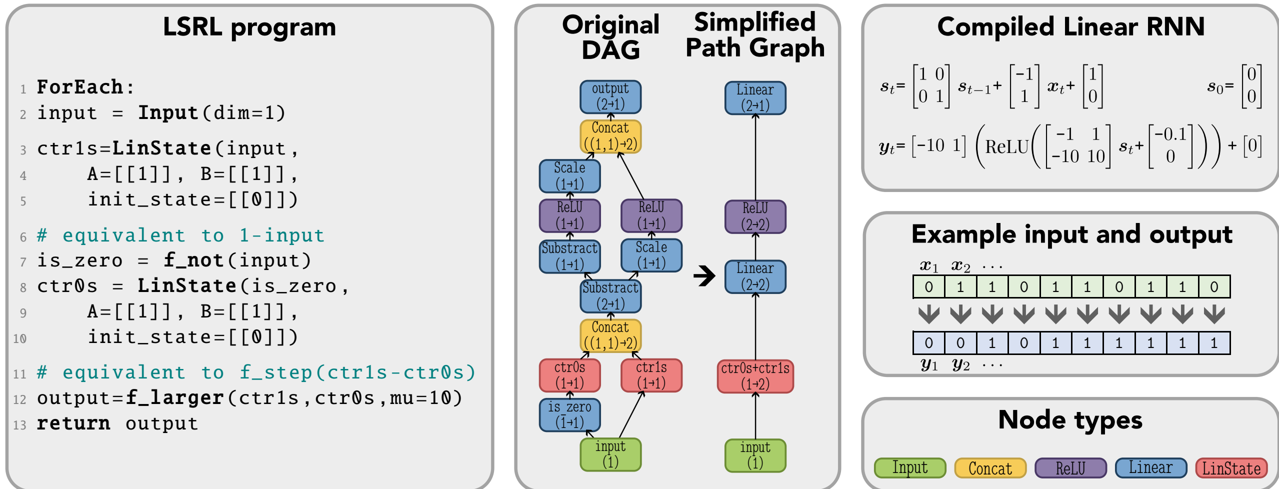


Figure 5.1: **Compilation of an LSRL program to a Linear RNN.** An example of a simple LSRL program that takes a sequence of 0s and 1s as an input and outputs 1 if there have been more 1s than 0s and 0 otherwise. The LSRL compiler follows the rules in Appendix C.1 to simplify the computation DAG into a path graph. The resulting path graph can be represented as a Linear RNN with one layer.

Language (LSRL).[2] LSRL programs compile to the four architectures in Equations (5.2) to (5.5). Conversely, any Linear RNN can be represented as an LSRL program, making LSRL a versatile tool for studying the capabilities of recurrent models. Later, in Sections 5.4 to 5.6 we make use of LSRL to develop programs that are universal approximators for $\mathcal{C}^{\text{vec}}$ and $\mathcal{C}^{\text{tok}}$, thus showing that all four architectures can be universal in-context approximators.

LSRL syntax. An LSRL program specifies how a single element is processed and how the recurrent states are updated for the next element. LSRL programs always start with an $\texttt{Input}(x) = x$ with an x of a fixed dimension. Only one `Input` can be declared in a program. Linear layers and `ReLU`s are also supported: $\texttt{Lin}[A, b](x) := Ax + b$, $\texttt{ReLU}(x) := \max(0, x)$. The unique component of LSRL, however, is its `LinState` operation implementing the linear state update in Linear RNNs (Equation (5.3)): $\texttt{LinState}[A, B, b, s_0](x_t) := As_{t-1} + Bx_t + b$, where the state s_{t-1} is the output of the call this node at step $t-1$. `LinState` is the only way information can be passed from previous tokens to the current one. We also provide a `Concat` operation that combines variables: $\texttt{Concat}(x, y) := (x_1, \ldots, x_{|x|}, y_1, \ldots, y_{|y|})$. Finally, to support gating architectures we also implement a rudimentary `Multi` operation that splits its input into two sub-arrays and returns their element-wise multiplication: $\texttt{Multi}(x) := x[\,:\,{}^{|x|}/2] \odot x[{}^{|x|}/2\,:\,]$. Naturally, `Multi` requires that x has even length. These six operations can be composed into a direct acyclic graph (DAG) with a single source node (the `Input` variable) and a single sink node (marked with a `return` statement).

[2]Our implementation of LSRL is available at https://github.com/AleksandarPetrov/LSRL

Such a program operates over a single token x_t passed to `Input`, while a recurrent model needs to operate over sequences. Thus, we wrap the program into a `ForEach` loop that passes each element individually for the DAG to output a variable denoted by a `return` clause. Each element is processed by the exact same program, with the only difference being that the state of the `LinState` variables is changing between iterations. You can see an example of a small LSRL program in Figure 5.1.

Expressiveness limitations. `ForEach` does not behave like the typical `for` loop: only the states are accessible between iterations, i.e., you cannot use the output of a linear layer at step t in any computation at step $t+1$. Furthermore, as the program is a DAG and only states of `LinState` nodes are passed between iterations, variables computed in latter operations of a previous time step are not accessible as inputs in earlier layers (with respect to the topological sorting of the computation graph). This leads to a key programming paradigm in LSRL: a `LinState` update cannot depend non-linearly on its own state. That includes it depending on a variable that depends on the `LinState` itself and conditional updates to the state. Such a dependency would break the DAG property of the program.[3] This poses serious limitations on what algorithms can be expressed in a Linear RNN and makes programming them challenging. Still, in Section 5.4 we show how carefully constructing state updates and auxiliary variables nevertheless allows to program some limited conditional behaviours.

Compilation. Any LSRL program without `Multi` nodes can be compiled to a Linear RNN (Equation (5.3)) or to a RNN (Equation (5.2)). If the program has `Multi` nodes, then it cannot be compiled to a Linear RNN as the multiplicative gating cannot be implemented exactly. However, it can be compiled to a Gated Linear RNN. To compile an LSRL program to a Linear (Gated) RNN, we first parse the program to build a computation graph. This is a DAG with a single source (the `Input` node) and a single sink (the `return` statement of the `ForEach` loop). At the same time, a Linear (Gated) RNN can be represented as a path graph (no branching) with the six basic operations as nodes. Therefore, the compilation step needs to transform this DAG into a path graph. We achieve that by iterativly collapsing the first branching point into a single node. The exact rules that achieve that are described in Appendix C.1. Later, in Section 5.6, we will show how any Linear (Gated) RNN can be converted into a *non-linear* (Gated) RNN, hence, how we can compile LSRL programs to these architectures as well.

Syntactic sugar. To make programming easier, we define several convenience functions. For instance, we can `Slice` variables $x[l{:}u]$ via sparse `Lin` layers. We can also sum variables and element-wise multiplication with scalars (implemented as `Lin` layers). For logical operations we also need step functions which can be approximated with `ReLU`s: $\texttt{f_step}[\mu](x) := \texttt{ReLU}(\mu x) - \mu\texttt{ReLU}(x - {}^1/_\mu)$, where μ is a positive constant controlling the

[3] For example, we cannot implement an operation that adds one to the state and squares it at each time step: $s_{t+1} = (s_t + 1)^2$, or an operation that performs conditional assignment scuh as $s_{t+1} = 0 \texttt{ if } (s_t > 5) \texttt{ else } s_t$.

quality of the approximation. We can also approximate bump functions (1 between l and u and 0 otherwise): $\texttt{f_bump}[\boldsymbol{l}, \boldsymbol{u}, \mu](\boldsymbol{x}) := \texttt{f_step}[\mu](\boldsymbol{x} - \boldsymbol{l}) - \texttt{f_step}[\mu](\boldsymbol{x} - \boldsymbol{u})$. Similarly, we can approximate conjunction (`f_and`), disjunction (`f_or`), negation (`f_not`), and comparison operators (`f_larger` and `f_smaller`). See Appendix C.6 for the definitions.

Critically, we need also a conditional operator that assigns a value $\texttt{t}(\boldsymbol{x})$ if a certain condition is met and another value $\texttt{f}(\boldsymbol{x})$ otherwise. One way to implement this is:

$$\begin{aligned}\texttt{f_ifelse}[\texttt{cond}, \texttt{t}, \texttt{f}, \lambda](\boldsymbol{x}) := &\ \texttt{ReLU}(\texttt{-}\lambda\, \texttt{cond}(\boldsymbol{x})\texttt{+}\texttt{f}(\boldsymbol{x})) + \texttt{ReLU}(\texttt{-}\lambda\, \texttt{f_not}(\texttt{cond}(\boldsymbol{x}))\texttt{+}\texttt{t}(\boldsymbol{x})) \\ &- \texttt{ReLU}(\texttt{-}\lambda\, \texttt{cond}(\boldsymbol{x})\texttt{-}\texttt{f}(\boldsymbol{x})) - \texttt{ReLU}(\texttt{-}\lambda\, \texttt{f_not}(\texttt{cond}(\boldsymbol{x}))\texttt{-}\texttt{t}(\boldsymbol{x})),\end{aligned} \tag{5.7}$$

where λ is a constant that is larger than any absolute value that $\texttt{t}(\boldsymbol{x})$ and $\texttt{f}(\boldsymbol{x})$ can attain. This construction, however, is not numerically stable (consider if $\texttt{cond}(\boldsymbol{x})$ is not exactly 0 but a small positive number) and we will study alternatives in Section 5.5. We provide both numerical (`SciPy.sparse`, Virtanen et al. 2020) and symbolic (`SymPy`, Meurer et al. 2017) backends with the second being crucial for programs that are not numerically stable.

Constant and dynamic variables. It is important also to distinguish between variables which can be dynamically assigned and such that must by "baked in" the model weights and be constant. Some operations can only be performed when one of the operands is a constant. For example, with a Linear RNN we cannot exactly compute the product of two variables —such as $\texttt{Lin}[\boldsymbol{A}_1, \boldsymbol{b}_1](\boldsymbol{x}) \odot \texttt{Lin}[\boldsymbol{A}_2, \boldsymbol{b}_2](\boldsymbol{x})$— but we can compute a product with a fixed vector $\texttt{Lin}[\boldsymbol{A}_1, \boldsymbol{b}_1](\boldsymbol{x}) \odot \boldsymbol{v}$. This is also why λ in Equation (5.7) cannot be dynamically computed depending on the input $\boldsymbol{x}$. This is not the case for the gated architectures, where variable product is possible, something we will leverage to construct more numerically stable conditional operators in Section 5.5.

Prior work on encoding algorithms in model weights. A similar approach to developing a programming language that compiles to model weights was already done for the transformer architecture with the RASP language (Weiss et al., 2021) and the Tracr compiler (Lindner et al., 2023). They were predominantly created as a tool for interpretability research. In a sense, RASP is to a transformer as LSRL is to a (Linear) (Gated) RNN. Hence, LSRL can be used to develop benchmarks for interpretability methods for fully-recurrent architectures. However, while RASP can only express a subset of transformer models, LSRL is isomorphic to the set of all (Gated) Linear RNNs (though not to the non-linear ones). That means that any (Gated) Linear RNN can be represented and analysed as an LSRL program and vice versa. Hence, the limitations of what you can express in LSRL are also limitations of what a Linear (Gated) RNN can do. Namely: (*i*) we cannot have exact multiplicative interactions between inputs without multiplicative gates, and (*ii*) we cannot have state variable updates depending non-linearly on their previous iterations or in any way on a variable that depends on them.

```
1  ForEach:
2  input = Input(dim=1+d_in+d_out)
3  # counter needed to know whether we are looking at the query or the prompt
4  const_1 = f_constant(input, 1)
5  counter_vector = LinState(input=const_1, A=ones(d_in,d_in), B=ones(d_in,1), init_state=zeros(d_in
       ,1))
6  # copy the query in a state (only when the counter is 1)
7  q_update = f_ifelse(cond=f_smaller(counter_vector, 1.5), t=input[: d_in], f=input[: d_in]*0)
8  q = LinState(input=q_update, A=eye(d_in), B=eye(d_in), init_state=zeros(d_in,1))
9  # the following operations will only change the output when counter > 1
10 # the step size is the first element of every prompt element
11 step_size = Linear(input=input[0], A=ones(d_in,1), b=zeros(d_in,1))
12 # using it we can compute the upper bounds of the current prompt cell
13 lb = input[1 : 1 + d_in]
14 ub = lb + step_size
15 # now check if q is in this cell (the bump should be 1 on all dimensions)
16 q_in_bump_componentwise = f_bump(q, lb, ub)
17 bump_sum = Linear(input=q_in_bump_componentwise, A=ones(1,d_in), b=zeros(1,1))
18 in_cell = f_larger(bump_sum, d_in - 0.5)
19 in_and_processing = f_and(in_cell, f_larger(counter, 0.5))
20 # if counter>1 and this cell contains q, add the value to the output state
21 update = f_ifelse(cond=f_larger(in_and_processing,0.5), t=input[-d_out:], f=input[-d_out:]*0)
22 y = LinState(input=update, A=eye(d_out), B=eye(d_out), init_state=zeros(d_out,1))
23 return y
```

Listing 1: **LSRL program for universal approximation in-context for continuous functions.** The inputs are $\boldsymbol{q} = [\boldsymbol{q}'^\top, \mathbf{0}^\top_{d_\text{out}+1}]^\top$ with $\boldsymbol{q}' \in [0,1]^{d_\text{in}}$ being the query value at which we want to evaluate the function, then followed by prompts describing the target function as in Equation (5.8).

5.4 Universal in-context approximation with Linear RNNs

Now that we are equipped with LSRL, we can proceed to building LSRL programs that are universal in-context approximators. We will describe two programs: one for approximating continuous functions ($\mathcal{C}^\text{vec}$), and one for approximating maps between token sequences ($\mathcal{C}^\text{tok}$). Formally, we construct a model g_vec of the Linear RNN architecture (Equation (5.3)) such that $\mathcal{H}^\text{vec}(g_\text{vec})$ is dense in $\mathcal{C}^\text{vec}$ and a model g_tok such that $\mathcal{H}^\text{tok}(g_\text{tok})$ is dense in $\mathcal{C}^\text{tok}$.

5.4.1 Approximating continuous functions in $\mathcal{C}^\text{vec}$

The idea behind the approximation for continuous functions is to discretise the domain into a grid and approximate the function as constant in each cell of the grid. This technique is commonly used for showing universal approximation using the step activation function (Blum and Li, 1991; Scarselli and Tsoi, 1998). However, it is not obvious how to implement this approach in-context when information across input tokens can be only combined linearly. Consider a target function $f : [0,1]^{d_\text{in}} \to [0,1]^{d_\text{out}}$ and a discretization step δ. Our approach is to describe the value of f in each of the discretization cells as a single prompt token. For the cell with lower bounds $l_1, \ldots, l_{d_\text{in}}$ and their respective upper bounds

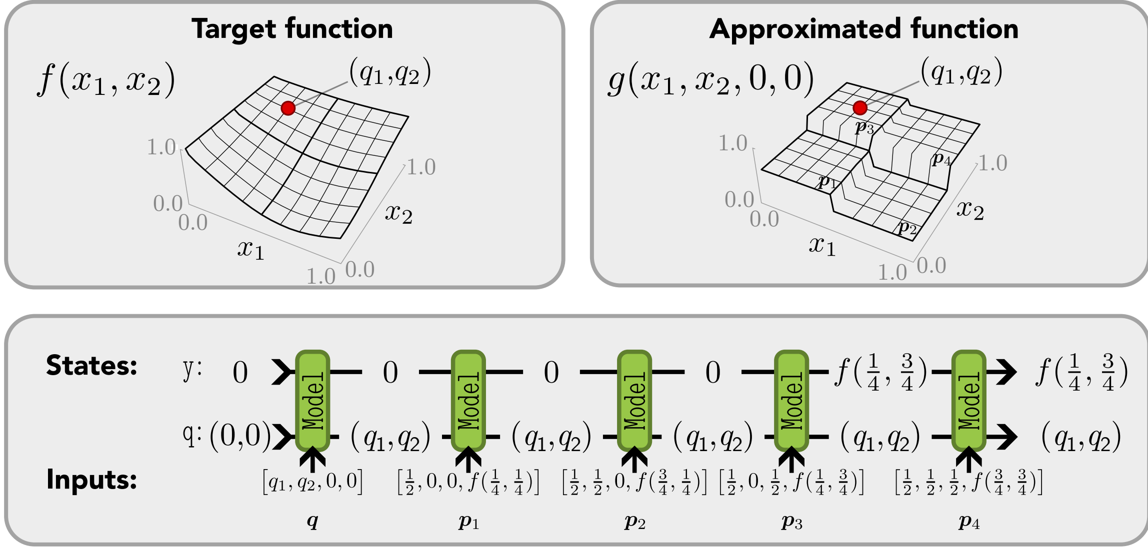


Figure 5.2: **Intuition behind the LSRL program for universal in-context approximation for continuous functions in Listing 1.** Our target function f has input dimension $d_{\text{in}} = 2$ and output dimension $d_{\text{out}} = 1$. Each input dimension is split into two parts, hence $\delta = 1/2$. We illustrated an example input sequence of length 5: one for the query and four for the prompt tokens corresponding to each of the discretisation cells. The query (q_1, q_2) falls in the cell corresponding to the third prompt token. We show how the two `LinState` variables in the program are updated after each step. Most notably, how the state holding the output y is updated after $\boldsymbol{p}_3$ is processed.

$l_1+\delta, ..., l_{d_{\text{in}}}+\delta$, the corresponding prompt token is a $(d_{\text{in}}+d_{\text{out}}+1)$-dimensional vector:

$$\boldsymbol{p} = [\delta, l_1, \ldots, l_{d_{\text{in}}}, \bar{\boldsymbol{y}}_1, \ldots \bar{\boldsymbol{y}}_{d_{\text{out}}}]^\top, \tag{5.8}$$

where $\bar{\boldsymbol{y}}$ is the value of f at the centre of that cell: $\bar{\boldsymbol{y}} = f(l_1+\delta/2, ..., l_{d_{\text{in}}}+\delta/2)$. Each prompt token describes the size of the cell (the discretisation step δ), its starting lower bound, and the value of the target function at the centre of the cell. Thus, $\lceil 1/\delta \rceil^{d_{\text{in}}}$ such tokens, one for each cell, are sufficient to describe the piece-wise constant approximation of f. A query $\boldsymbol{q}' \in [0,1]^{d_{\text{in}}}$ can fall in only one of the cells. We pad it with zeros and encode it as the first input element: $\boldsymbol{q} = [\boldsymbol{q}'^\top, \boldsymbol{0}^\top_{d_{\text{out}}+1}]^\top$, followed by the prompt. Our program will extract and save $\boldsymbol{q}'$ to a state and then process the prompt tokens one at a time until it finds the one whose cell contains $\boldsymbol{q}'$. The target function value for this cell will be added to an accumulator state. If the current cell does not contain $\boldsymbol{q}'$, then 0 is instead added.' Hence, the accumulator's final value corresponds to the value of f at the centre of the cell containing $\boldsymbol{q}'$. The full LSRL program is provided in Listing 1 and an illustration for $d_{\text{in}} = 2, d_{\text{out}} = 1, \delta = 1/2$ is shown in Figure 5.2. The prompt length required to approximate an L-Lipschitz function f (w.r.t. the ℓ_2 norm) to precision ϵ is $N = (2\epsilon/L\sqrt{d_{\text{in}}})^{-d_{\text{in}}} = \mathcal{O}(\epsilon^{-d_{\text{in}}})$ (see Appendix C.2 for the proof). Asymptotically, this is as good as one can hope without further assumptions on

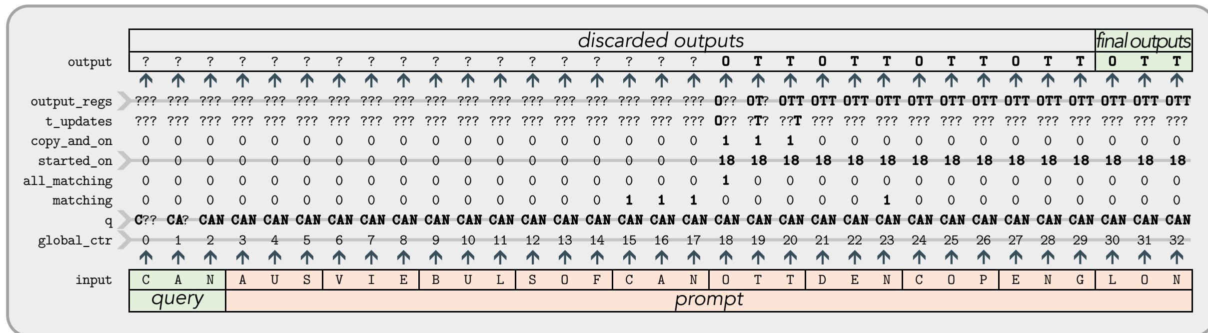


Figure 5.3: **Intuition behind the LSRL program for universal in-context approximation for discrete functions in Listing 2.** Our keys and values have length n=3 and represent countries and capitals, e.g., `AUS`tria↦`VIE`nna, `BUL`garia↦`SOF`ia, and so on. The query is `CAN` for Canada and the final n outputs are `OTT` (Ottawa). We show the values of some of the variables in Listing 2 at each step, with the `LinState` variables being marked with arrows. For cleaner presentation we are tokenizing letters as `0`↦`?`, `1`↦`A`, `2`↦`B`, etc. Vertical separators are for illustration purposes only.

the target function. This is also better than the best known result for the same problem for transformers: $\mathcal{O}(\epsilon^{-10-14d_{\text{in}}-4d_{\text{in}}^2})$ in Petrov et al. 2024b.

The LSRL program in Listing 1 also allows us to perform *streaming* universal in-context approximation for free. As the discretization step δ is not hard-coded in the model, we can first provide prompts at a coarse grid and then iteratively add prompts at increasingly finer grids, each providing a correction of the estimate of the previous one. Thus, if the computation is interrupted, or a compute budget is reached, our model will still output the best approximation of the target function for this budget.

5.4.2 Approximating functions over token sequences in $\mathcal{C}^{\text{tok}}$

Section 5.4.1 focused on continuous functions but recurrent architectures are often used to model natural language whose domain is tokens. Thus, we also look at modelling maps over a discrete domain. Any function from n tokens to n tokens taking values in $\mathcal{V} = \{1, \ldots, V\}$ can be represented as a dictionary whose keys and values are in $\mathcal{V}^n$. Therefore, a simple way to represent this function in-context is to first provide the n tokens corresponding to the query and then a sequence of $2n$ tokens corresponding to key and value pairs (see Figure 5.3 for an illustration of the setup). The model stores the query in a state and processes the key-value pairs one by one by comparing the key (the first n tokens) with the query. If they match, then the value (the next n tokens) is copied into a state that keeps it and repeatedly outputs it. This continues until the end of the prompt, at which point the last n outputted tokens will be the value corresponding to the key matching the query. This is essentially a dictionary lookup. However, as shown in Listing 2, implementing dictionary lookup in a linear recurrent model is much less straightforward than executing `dict[key]` in a

general-purpose programming language.

Listing 2 can appear daunting at first so we would like to clarify the non-trivial aspects. First, we need to count how far we are into every set of n or $2n$ tokens. This can be done with $\bmod n$ and $\bmod 2n$ operations but implementing modulo for arbitrary large inputs is not possible with `ReLU` MLPs (Ziyin et al., 2020). Therefore, we implement this with `LinState` as `f_modulo_counter` which has a unit-length state that is rotated $1/n$ or $1/2n$ revolutions per iteration, with the angle corresponding to the modulo value (Appendix C.6.7). Second, we need to do dynamic indexing to copy the i-th input in a subsequence to the i-th element of a state and vice-versa. Dynamic indexing, however, cannot be succinctly represented in a Linear RNN. We work around this with temporary variables that are non-zero only at the i-th coordinates (see Lines 16, 17, 19, 20, 32 to 35, 37 and 38). Finally, in order to compare whether all n elements in the query and the key match, we need to remember whether the previous n pairs were matching. As RNNs do not have attention, we implement this short-term memory buffer as a `LinState` with a shift matrix (Line 23).

5.5 Stable universal in-context approximation with Gated Linear RNNs

The ReLU-based conditional operator is not numerically stable. The LSRL programs in Listings 1 and 2 for approximating functions in respectively $\mathcal{C}^{\text{vec}}$ and $\mathcal{C}^{\text{tok}}$ rely on the `f_ifelse` conditional assignment operator in Equation (5.7) in order to implement different behaviours depending on whether we are processing the query or specific parts of the prompt. This operator is not numerically stable. The first term in Equation (5.7) relies on $\texttt{cond}(\boldsymbol{x})$ being exactly zero if the condition is not met. In this way, multiplying it with $-\lambda$ would be 0 and $\texttt{f}(\boldsymbol{x})$ would be returned. However, if $\texttt{cond}(\boldsymbol{x})$ is not identically 0 but has a small positive value, then $-\lambda\texttt{cond}(\boldsymbol{x})$ can "overpower" $\texttt{f}(\boldsymbol{x})$ resulting in the `ReLU` output being 0. In our experience, this is not a problem when processing inputs through the LSRL program step-by-step. However, de-branching the DAG into a path graph —which is necessary in order to uncover the equivalent Linear RNN— appears to introduce such numerical instabilities which occasionally result in wrong outputs as conditional assignments will be 0 when they should not. This problem is more prominent in Listing 2 which is longer (more debranching steps) and has more `f_ifelse` operations: it gets most tokens wrong because of that instability (see *Original, No noise* in Figure 5.4). To this end, we support LSRL with a symbolic backend (based on SymPy) that performs the debranching steps exactly. Using it, both programs always produce the correct output.

This numerical instability highlights a critical practical limitation of the universal approximation results in Section 5.4: if the models are not numerically stable, it is unlikely that they occur in practice by training models using gradient descent. This section shows how to improve the numerical stability of Equation (5.7) and obtain more realistic recurrent models that are universal approximators in-context.

Implementing `f_ifelse` with MLPs. As LSRL allows us to express arbitrary MLPs and MLPs can approximate any continuous function, it is tempting to replace Equation (5.7) with a deep MLP model. However, such implementation would also not be exact, which can cause problems when composing such logical operations. For this reason, and for compactness of the resulting compiled model, we do not consider deeper implementations of `f_ifelse`.

Removing unnecessary terms in Equation (5.7). Equation (5.7) has 4 separate `ReLU` terms. The first two handle the cases when $\mathtt{t}(x)$ and $\mathtt{f}(x)$ are positive and the second two when they are negative. Therefore, if we know that one or both of these will always be non-negative, we can drop the corresponding terms. This is especially useful for Listing 2, approximating $\mathcal{C}^{\text{tok}}$, as it exclusively uses non-negative values (counters, mode switches and token values). Additionally, if $\mathtt{f}(x)$ is always 0, then the first and third terms can be safely dropped. Similarly, the second and fourth are unnecessary if $\mathtt{f}(x) \equiv 0$. All `f_ifelse` in Listings 1 and 2 fall in this case and hence can be simplified. We will refer to this `f_ifelse` implementation that is aware of the attainable values of $\mathtt{t}(x)$ and $\mathtt{f}(x)$ as `optimized`. As it reduces the number of numerically unstable `ReLU` operations in the model, we expect that it will improve the stability of the compiled models. We experimented with adding various levels of noise to the non-zero model parameters, and, as the results in Figure 5.4 show, `optimized` is indeed more numerically robust than `original`.

Step-based implementation. We can get rid of the input sensitivity of Equation (5.7) using `f_step`:

$$
\begin{aligned}
\mathtt{f_ifelse}[\mathtt{cond}, \mathtt{t}, \mathtt{f}, \lambda](x) := {} & \mathtt{ReLU}(-\lambda + \lambda \mathtt{f_step}(1/2 - \mathtt{cond}(x)) + \mathtt{f}(x)) \\
& + \mathtt{ReLU}(-\lambda + \lambda \mathtt{f_step}(\mathtt{cond}(x) - 1/2) + \mathtt{t}(x)) \\
& - \mathtt{ReLU}(-\lambda + \lambda \mathtt{f_step}(1/2 - \mathtt{cond}(x)) - \mathtt{f}(x)) \\
& - \mathtt{ReLU}(-\lambda + \lambda \mathtt{f_step}(\mathtt{cond}(x) - 1/2) - \mathtt{t}(x)).
\end{aligned} \tag{5.9}
$$

We can also apply the optimisation strategy here. While this implementation is robust to noise in the input it appears to be more sensitive to parameter noise, as shown in Figure 5.4.

Numerically stable `f_ifelse` with multiplicative gates. Removing the unused `ReLU` terms in the original `f_ifelse` reduces the opportunities for numerical precision issues to creep in but does not solve the underlying problem. The multiplicative gating present in the Linear Gated RNN (Equation (5.4)) and Gated RNN models (Equation (5.5)) can help via implementing a numerically stable conditional operator:

$$
\mathtt{f_ifelse}[\mathtt{cond}, \mathtt{t}, \mathtt{f}](x) := \mathtt{cond}(x) \odot \mathtt{t}(x) + \mathtt{f_not}(\mathtt{cond}(x)) \odot \mathtt{f}(x), \tag{5.10}
$$

where the element-wise product is implemented in LSRL with `Concat` and `Multi`. We will refer to the implementation of `f_ifelse` in Equation (5.10) as `multiplicative`. Similarly to original implementation of `f_ifelse` in Equation (5.7), we can drop the $\mathtt{t}(x)$ and

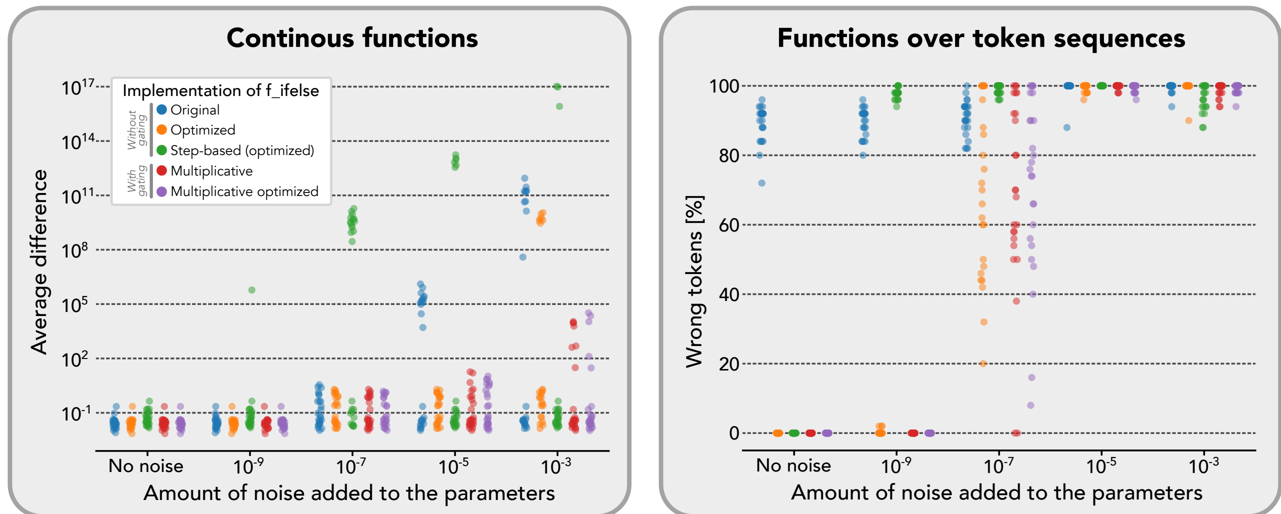


Figure 5.4: **Robustness of the various `f_ifelse` implementations to model parameter noise.** We show how the performance of the two universal approximation programs in Listings 1 and 2 deteriorates as we add Gaussian noise of various magnitudes to the non-zero weights of the resulting compiled models. As expected, the original `f_ifelse` implementation in Equation (5.7) exhibits numerical precision errors at the lowest noise magnitude. For the token sequence case, numerical precision errors are present in all samples even in the no-noise setting. Hence, the original `f_ifelse` implementation is less numerically robust while the implementations with multiplicative gating are the most robust. For Listing 1 (approximating $\mathcal{C}^{\text{vec}}$) we report the Euclidean distance between the target function value and the estimated one over 10 queries for 25 target functions. For Listing 2 we report the percentage of wrong token predictions over 5 queries for 25 dictionary maps. Lower values are better in both cases.

$\mathtt{f}(x)$ term if they are equal to zero (`multiplicative optimized`). If $\mathtt{cond}(x)$ is not exactly zero, $\mathtt{cond}(x) \odot \mathtt{t}(x)$ will result in a small error to the output but, in contrast to the original implementation, is not going to cause a discontinuity in the output of the operation. Therefore, Equation (5.10) should be more robust to numerical precision issues than Equation (5.7). Figure 5.4 shows that this is the case in practice with Listings 1 and 2 being more robust to parameter noise when using multiplicative gates compared to the `ReLU`-based implementations. Therefore, Linear Gated RNNs (Equation (5.4)) —to which models with multiplicative gates can be compiled— are more likely than Linear RNNs (Equation (5.3)) to exhibit universal approximation properties in practice.

Stability benefits of multiplicative gating in recurrent models has been previously shown in the context of deterministic finite-state automata (Omlin and Giles, 1996). Beyond more stable conditional operators, multiplicative gating also results in strictly more expressive models than models with only element-wise nonlinearities (Jayakumar et al., 2020).

5.6 Universal in-context approximation with non-linear (Gated) RNNs

Sections 5.4 and 5.5 showed how universal approximation of continuous and token-to-token functions can be implemented in LSRL and compiled to respectively Linear RNNs and Linear Gated RNNs. This section aims to address the situation with *non-linear* state updates, that is, the cases of classic and gated RNNs (Equations (5.2) and (5.5)). Concretely, we show how every *linear* (Gated) RNN can be converted to a *non-linear* (Gated) RNN. Thus, we can compile any LSRL program (including Listings 1 and 2) also to an RNN (if it has no `Multi` operations) or a Gated RNN.

The key idea is that the `ReLU` applied to the state updates in the non-linear architectures is an identity operation if its inputs are positive. Hence, we can split the states in positive and negative components, flip the sign of the negative component, pass them separately through the `ReLU`—which will act as an identity as all elements will be non-negative— and then fuse the positive and negative components back together in the $\boldsymbol{A}$ matrix at the next time step. Formally, we can convert a Linear RNN state update into a classic RNN state update as following:

$$
\begin{aligned}
\boldsymbol{s}_t &= \boldsymbol{A}\boldsymbol{s}_{t-1} + \boldsymbol{B}\boldsymbol{x}_t + \boldsymbol{b} \\
\boldsymbol{y}_t &= \phi(\boldsymbol{s}_t).
\end{aligned}
\quad\equiv\quad
\begin{aligned}
\begin{bmatrix} \boldsymbol{s}_t^+ \\ \boldsymbol{s}_t^- \end{bmatrix} &= \mathtt{ReLU}\left(\begin{bmatrix} \boldsymbol{A} & -\boldsymbol{A} \\ -\boldsymbol{A} & \boldsymbol{A} \end{bmatrix}\begin{bmatrix} \boldsymbol{s}_t^+ \\ \boldsymbol{s}_t^- \end{bmatrix} + \begin{bmatrix} \boldsymbol{B} \\ -\boldsymbol{B} \end{bmatrix}\boldsymbol{x}_t + \begin{bmatrix} \boldsymbol{b} \\ -\boldsymbol{b} \end{bmatrix}\right) \\
\boldsymbol{y}_t &= \phi\left(\begin{bmatrix} \boldsymbol{I} & -\boldsymbol{I} \end{bmatrix}\begin{bmatrix} \boldsymbol{s}_t^+ \\ \boldsymbol{s}_t^- \end{bmatrix}\right).
\end{aligned}
\tag{5.11}
$$

Another way to look at this is by recognizing that an RNN is equivalent to a Linear RNN with the exact same weights if the states are always non-negative. Hence, all we need is a trick to ensure the states are non-negative. This approach works just as well for the Gated RNNs as the gating and the state updates are independent from one another.

Using Equation (5.11) we can compile any LSRL program to an RNN (Equation (5.2)) or a Gated RNN (Equation (5.5)). This includes Listings 1 and 2. Hence, RNNs and Gated RNNs can be universal in-context approximators for continuous and token-to-token functions. As any Gated RNN can be represented as a GRU model (Appendix C.3) or an LSTM (Appendix C.4), these models are too universal in-context approximators. The same numerical stability issues discussed in Section 5.5 apply here and as a result, universal approximation capabilities are probably more likely to occur in Gated RNNs than in RNNs.

5.7 Discussion and conclusions

We developed LSRL: a programming language for specifying programs expressible with recurrent neural architectures. We then used LSRL to show that various architectures —from the humble RNN to the state-of-the-art Linear Gated RNNs— can all be universal approximators *in-context*. That is, there exist fixed models with these architectures which can

be prompted to act as any token-to-token function or approximate any continuous function to an arbitrary precision. These results do not assume infinite precision or exponential hidden state sizes. The hidden state sizes of our constructions are also independent of the target precision (in the continuous setting) or the key-value dictionary size (in the discrete setting).

Comparison with the transformer architecture. Contemporary Linear RNNs attempt to challenge the dominant role of the transformer architecture. At the same time, our understanding of their in-context abilities is significantly lacking behind that of the transformer. This work makes an important contribution to this problem: we showed that Linear SSMs are not only universal in-context approximators but are potentially require shorter prompts than transformers ($\mathcal{O}(\epsilon^{-d_{\text{in}}})$ vs $\mathcal{O}(\epsilon^{-10-14d_{\text{in}}-4d_{\text{in}}^2})$ from Petrov et al. 2024b). That approach also relies on the Kolmogorov-Arnold representation theorem (Kolmogorov, 1957) which is notoriously unlikely to be useful in practice (Girosi and Poggio, 1989). Our constructions are much simpler, especially in the token-to-token case.

Ability of models to implement algorithms. There has been a lot of attention on evaluating how well models can execute various algorithms (Giannou et al., 2023; La Malfa et al., 2024; Sanford et al., 2024). These abilities are fundamentally limited to how many basic operations a model can do in a single forward pass. Our results indicate that this might be architecture dependent and that multiplicative gating might be much more efficient for implementing logic operations at the core of many algorithms. Hence, it might be possible that more complex logic programs could be expressed with the same number of parameters if one uses an architecture with multiplicative gating.

Safety and security implications. If a model can be prompted to approximate any function, then preventing it from exhibiting undesirable behaviours (i.e., alignment) might be fundamentally impossible. On the flip side, recently, methods for improving the safety of transformers using interpretability approaches have been proposed (Conmy et al., 2023; Geiger et al., 2024). The success of interpretability techniques is difficult to assess though. To assist with that, benchmarks of models with known behaviours can be developed. RASP has already been used to evaluate *transformer* interpretability methods (Friedman et al., 2023; Zhou et al., 2024). However, interpretability tools for fully recurrent models are significantly lagging behind the ones for transformers. Therefore, we hope that LSRL can be helpful for designing interpretability benchmarks for fully recurrent models, similarly to how RASP has contributed to understanding transformer models.

Limitations. In this work we provide *constructive existence results*: that is, we show that there can exist models with various recurrent architectures that are universal in-context approximators. However, the present theory is not sufficient to analyse whether *a given model* has this property. That is a much more difficult question that would require a very

different approach. We also assume no restrictions on the A matrix in the state update equations. However, many state-of-the-art models impose structural constraints on A (e.g., it being diagonal) for the sake of fast training and inference (Gu et al., 2020, 2021; Gupta et al., 2022). It is not directly obvious whether such structural restrictions would affect the universal in-context approximation abilities of these architectures. In practice, however, the compiled matrices are very sparse and often diagonal. Therefore, it is highly likely that our results translate to models with structural restrictions.

Author contributions

This work was a collaboration with Tom A. Lamb and Alasdair Paren, as well as my supervisors, Dr. Adel Bibi and Prof. Philip H.S. Torr. The initial idea to follow up on our previous work (Chapter 4) and analyse whether universal in-context approximation capabilities extend to recurrent models was mine. I developed LSRL and its compiler, the proofs via LSRL programs for both the continuous and discrete settings and performed the experiments. Alasdair and Tom offered a lot of insight on how to approach the problem and proved that our results extend to LSTMs, GRUs and SSMs, namely the Hawk and Griffin architectures. Adel and Phil provided guidance and advice on how to approach, structure and present the project.

Acknowledgements

We would like to thank Simon Schug for pointing us to relevant works on fully recurrent models. We are also extremely grateful to Juuso Haavisto for his insight on building compilers. This work is supported by a UKRI grant Turing AI Fellowship (EP/W002981/1) and the EPSRC Centre for Doctoral Training in Autonomous Intelligent Machines and Systems (EP/S024050/1).

```
ForEach:
input = Input(dim=1)
# counter needed to know whether we are looking at the query or the prompt
const_1 = f_constant(input, 1)
global_ctr = LinState(input=const_1, A=[[1]], B=[[1]], init_state=[[-1]])
# counters mod[n] and mod[2n]
mod_n_ctr = f_modulo_counter(input, n)
mod_2n_ctr = f_modulo_counter(input, 2*n)
# which mode are we in (looking at the query, comparing query with key, or copying value to state)
is_prompt = f_larger(global_ctr, n-0.5)
is_compare_mode = f_larger(mod_2n_ctr, n-0.5)
is_copy_mode = f_and(is_prompt, f_not(is_compare_mode))
is_first_token_for_copy = f_and(is_copy_mode,f_smaller(mod_n_ctr, 0.5))
# update the state holding the query if this is one of the first n tokens
tq=f_ifelse(f_smaller(is_prompt, 0.5), t=input, f=input*0)
tqs=[f_ifelse(f_and(f_larger(mod_n_ctr,i-0.5), f_smaller(mod_n_ctr,i+0.5)),t=tq,f=tq*0) for i in
    1..n]
q = LinState(input=Concat(tqs), A=eye(n), B=eye(n), init_state=zeros(n,1)) # query
# if we are in compare mode (looking at keys), check if this token matches the corresponding one in
    the query
qs=[f_ifelse(f_and(f_larger(mod_n_ctr,i-0.5),f_smaller(mod_n_ctr,i+0.5)),t=q[i],f=q[i]*0) for i in
    1..n]
cor_q_el = Linear(input=Concat(qs), A=ones(1,n), b=zeros(1,1))
matching = f_and(f_and(f_larger(input, cor_q_el-0.5),f_smaller(input, cor_q_el+0.5)),
    is_compare_mode)
# keep a buffer of the last last n+1 match values, the +1 because we can only read the buffer after
    writing
buffer = LinState(input=matching, A=shift_matrix, B=[[0] for _ in 1..n], [[1]]), init_state=zeros(n
    +1,1))
buffer_sum = Linear(input=buffer, A=[[1 for _ in 1..n], [0]], b=zeros(1, 1))
all_matching = f_larger(buffer_sum, n-0.5)
# if all are matching and it's the first token in the value part of the (key, value) pair, then
    mark this as the iterations when we start copying to state
matching_and_first_for_copy = f_and(all_matching, is_first_token_for_copy)
t_started_on_update = f_ifelse(matching_and_first_for_copy,t=global_ctr, f=global_ctr*0)
started_on = LinState(input=t_started_on_update, A=eye(1), B=eye(1), init_state=zeros(1, 1))
# copying to state for n iterations after started_on
copy_and_on = f_and(is_copy_mode, f_smaller(global_ctr, started_on+n))
mod_n_eq_i = [ f_and(f_larger(mod_n_ctr,i-0.5), f_smaller(mod_n_ctr,i+0.5)) for i in 1..n ]
t_updates_should_update = [f_and(copy_and_on, mod_n_eq_i[i]) for i in 1..n]
t_updates = [f_ifelse(f_larger(t_updates_should_update[i], 0.5), t=input, f=input*0) for i in 1..n]
output_regs = [LinState(input=update, A=eye(1), B=eye(1), init_state=zeros(1,1)) for update in
    t_updates]
# finally, read out the value from the corresponding output register in order to output from the
    model
t_outputs = [f_ifelse(f_larger(mod_n_eq_i[i], 0.5), t=output_regs[i], f=output_regs[i]*0) for i in
    1..n]
return Linear(input=Concat(t_outputs), A=ones(1,n), b=zeros(1,1))
```

Listing 2: **LSRL program for universal in-context approximation of discrete functions.** The inputs are $q_1, ..., q_n$ (the query tokens), followed by pairs of keys and values from the map we are approximating. The last n outputs are the value corresponding to the key matching the query.

Chapter 6

Conclusions and Outlook

Roughly a year into my DPhil, amidst the whirlwind of developments in machine learning, this thesis started to take shape around a central question: what are the powers and the limitations of pretrained sequence-to-sequence models. The emergence of zero-shot capabilities in large language models —and the accompanying rise of prompting as a paradigm— raised fundamental questions about what these systems can and cannot achieve. The theoretical lens of choice for this thesis was universal approximation: a classic tool for assessing a model's expressive capacity. We had to adapt this setup for the new paradigm of in-context learning, which resulted in the introduction of the concept of *universal in-context approximation*.

Although other approaches (such as analyzing formal language recognition) do exist, there are compelling reasons we focused on universal approximation. First, establishing universal approximation properties is a canonical way to support the choice of a neural network model architecture and a first line of defense against its detractors. And yet, it was not clear how in-context learning fits with the existing universal approximation literature. After all, classic results assume learnable parameters, which prompting explicitly avoids. Deep down, we suspected that prompting had limits: that there are problems one simply cannot solve without modifying the model itself. Perhaps it was part skepticism, part existential dread, and part empirical grounding: our first paper showed that prompting cannot match the flexibility of fine-tuning, at least for a single attention layer. Given our goal of proving a limitation, showing the absence of universal approximation seemed like a definitive way to make the case. As the reader knows by now, the punchline is a twist: we did not find a limitation. Instead, we found universality across all the major sequence-to-sequence architectures.

A natural question that follows our results is: what does all this mean for real-world models? Is one's favourite model a universal in-context approximator and what would that

mean? Unfortunately, that question is considerably harder to answer. In this work, as in most of the follow-up literature and the related studies on computational classes, we take a proof by construction approach: we explicitly build model weights that exhibit the desired universal behavior. However, the weights we select are highly structured and unlikely to arise through gradient descent under typical regularization, normalization, or real-world training data. So while we prove that such weights exist, this tells us little about whether a given model's weights actually have these properties.

Answering the second question is both more compelling and considerably more difficult. In principle, one might analyze whether the weights of a real attention layer could be mapped to our constructions via some linear transformation. But this idea quickly runs into practical complications. For one, real-world transformers have multiple attention heads, often employing grouped or sparse attention mechanisms. Depending on the specific implementation, this can cause the effective key and query matrices to be rank-deficient, though not necessarily so. Then, there are all the other structural details: residual connections, normalization layers (LayerNorm or RMSNorm), BOS tokens, input embeddings, nonlinearities, and so on.

Strict universality is a high bar, and I suspect even the best pretrained models fall short of it in practice. That is, one can likely find a target function that no prompt can cause that model to approximate. But this does not mean they are not useful—far from it. These models still approximate a wide range of practically important functions. So it may be more appropriate to study whether models can approximate a narrower, more realistic set of functions. The problem, of course, is that we do not know what that function class should be. If we did, we would not be building language models in the first place. This points to a broader tension typical of deep learning theory: the gap between theoretical expressivity and practical learnability may not be easily bridged.

One further complication involves numerical precision: our results assume arbitrary-precision arithmetic, which we cannot depend on in practical deployments. Consequently, even if we baked the weights we pick into a model and run it on a computer, it will fail to reach arbitrary precision for really long inputs. This limitation is not unique to our work: it is a common caveat for most universal approximation results. Our work with recurrent models, especially Linear RNNs, revealed just how sensitive these constructions are. When multiplicative gating was not available, floating-point error during both compilation and execution forced us to design a symbolic backend to operate on exact algebraic forms instead.

A concern raised by our findings relates to safety and security. If a deployed model is a universal in-context approximator, then in principle, it could be prompted to do anything—including tasks considered unsafe or undesirable. In that case, alignment becomes impossible: no matter the safeguards, there would always exist some prompt capable of steering the model toward unwanted behavior. However, our results depend on the assumption that the full specification of the target function is provided in-context. But if someone already has that full specification, then they do not need the model at all—they could just compute it directly. In that sense, these theoretical results are not immediately

threatening. Realistic concerns arise when prompting leverages both the information provided in-context and the knowledge encoded in the model's weights. Similar criticisms apply to computational class results. A model being Turing complete only matters if you give it the code to execute—at which point, why not just run the code elsewhere? For these results to inform real-world risks or capabilities, they must focus on the interaction between what the prompt provides and what the model already knows.

Sadly, the above argument becomes more complicated once these models are placed in agentic harnesses. Then, an attacker can use the universal approximation capabilities or its ability to execute arbitrary code in-context in order to coerce a model to produce an unsafe action that it otherwise would not. For example, while browsing the web to answer a query, the model might encounter a lengthy prompt that completely changes its behavior to anything the attacker wants. While similar to the ever increasing cases of malicious instructions hidden in webpages or documents, this type of an attack can fully change the model behaviour itself, rather than simply change the instruction it follows.

The architectures used in practice are often composed of a variety of components that function well in real-world applications but are challenging to study from a theoretical standpoint. As a result, both our work and nearly all other studies exploring the theoretical properties of these architectures necessarily rely on simplifying assumptions. For instance, many studies assume hardmax attention as a proxy for softmax attention, though there is evidence suggesting that the relationship between the two is not as straightforward as it might seem. While our approach to transformers worked with softmax attention, which might appear more realistic, it did assume that inputs could scale to arbitrary magnitudes, with less smooth functions requiring larger inputs and, consequently, higher norms. However, in practical scenarios, normalization layers would typically prevent this. Most empirical work, on the other hand, tends to focus on a limited set of synthetic tasks, which neither cover the full range of potential tasks nor reflect real-world use cases. Given all these simplifying assumptions and the often idealized evaluation setups, it remains unclear how directly such theoretical results translate to real-world models.

So does this mean that studying the theoretical properties of neural network architectures, and particularly their universal in-context approximation abilities, is fundamentally of little practical utility? Not at all. In fact, as researchers, we often make intuition-based claims that are difficult to verify empirically, especially when these claims are sensitive to specific conditions and hyperparameters. The only reliable way to validate such claims is through theoretical analysis. For instance, Linear RNNs and SSMs have received criticism for their perceived inability to capture long-term memory. However, we showed that this actually need not be a problem. The key lies in how data is represented and the fact that these models lack attention and rely on a fixed-size state does not prevent them from possessing the same universal in-context approximation abilities as transformers, perhaps even more efficiently. While theoretical results might not always provide definitive answers about the models we use in practice, they often serve to challenge our assumptions and intuitions, revealing that things are often more nuanced than we initially believed—or would like to

admit.

There are also several valuable “byproducts” from this work: thinking through how transformers or RNNs function, and how their behaviors can be implemented by construction, provides insight into how these architectures work, and, just as importantly, how they cannot work. Despite their impressive results at scale, their building blocks are simple and in fact quite limited in how they operate. Figuring out how to coax them to do the simple operations needed for universal in-context approximation was not at all easy, and, along the way, we discovered numerous ways in which they fail to work. This understanding of what these models can and cannot express has proven useful in tackling more applied problems. For example, in our work (Petrov et al., 2025c), this insight into the workings of transformer layers, and what kind of communication is possible (or impossible) between layers, was crucial in understanding why a popular context compression technique did not work well in practice. By addressing these issues, we were able to propose a much more effective approach. Beyond pure theoretical insight, there is also tooling that we can reuse for other purposes. For instance, while studying the properties of Linear RNNs we developed the LSRL programming language and compiler. This tool itself has practical applications, from building recurrent model components with specific properties to supporting interpretability research on such architectures. So, while these outcomes may not always be direct, they certainly offer valuable avenues for future work.

While this thesis did answer the questions it set out to explore, it inevitably opened up many new ones. For instance, it is evident that the true practical utility of models in real-world applications does not come solely from the model’s weights and parameters, nor do we fully specify the function we are modelling explicitly in context. Rather, it is the interaction between the two that drives practical utility. The same model might perform poorly with a naïve prompt and excellently with a carefully crafted one. Therefore, studying the interplay between in-context learning and in-weights learning will be key to understanding why current models succeed and how we can improve future ones.

It is also clear that theoretical results can only take us so far in deep learning. On the other hand, empirical results on a few synthetic tasks are similarly limited in scope and real-world relevance. What we really need are large-scale, systematic experiments across a diverse set of datasets and tasks, complemented by robust (possibly causal) analysis, to better understand how various architectures and hyperparameters translate into performance on unseen tasks. Additionally, the growing importance of test-time compute and the emergence of new scaling paradigms raise new questions about how we should measure performance.

After so many years of intense interest and research in deep learning, a surprising number of unknowns remain about what constitutes good models and effective training paradigms. In many ways, it is not even clear what questions we should be asking: What makes a model truly “good” to begin with? What is becoming increasingly evident, however, is that we need to think more deeply about what we need to understand in order to derive useful insights—and what is the right balance of theoretical understanding and empirical evidence we need in order to get there.

Bibliography

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*.

Kwangjun Ahn, Xiang Cheng, Hadi Daneshmand, and Suvrit Sra. 2023. Transformers learn to implement preconditioned gradient descent for in-context learning. In *Advances in Neural Information Processing Systems*.

Ekin Akyürek, Dale Schuurmans, Jacob Andreas, Tengyu Ma, and Denny Zhou. 2022. What learning algorithm is in-context learning? Investigations with linear models. In *International Conference on Learning Representations*.

Ekin Akyürek, Bailin Wang, Yoon Kim, and Jacob Andreas. 2024. In-context language learning: Arhitectures and algorithms. *arXiv preprint arXiv:2401.12973*.

Silas Alberti, Niclas Dern, Laura Thesing, and Gitta Kutyniok. 2023. Sumformer: Universal approximation for efficient transformers. In *Proceedings of 2nd Annual Workshop on Topology, Algebra, and Geometry in Machine Learning (TAG-ML)*.

Shun-ichi Amari. 1972. Learning patterns and pattern sequences by self-organizing nets of threshold elements. *IEEE Transactions on Computers*, C-21(11):1197–1206.

Donald E Amos. 1974. Computation of modified Bessel functions and their ratios. *Mathematics of Computation*, 28(125):239–251.

Kendall Atkinson and Weimin Han. 2012. *Spherical Harmonics and Approximations on the Unit Sphere: An Introduction*.

Seth Aycock, David Stap, Di Wu, Christof Monz, and Khalil Sima'an. 2025. Can LLMs really learn to translate a low-resource language from one grammar book? In *International Conference on Learning Representations*.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016. Layer normalization. *arXiv preprint arXiv:1607.06450*.

Yogesh J Bagul and Satish K Panchal. 2018. Certain inequalities of Kober and Lazarević type. *Research Group in Mathematical Inequalities and Applications Research Report Collection*, 21(8).

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *International Conference on Learning Representations*.

Jiaqi Bai, Zhao Yan, Jian Yang, Xinnian Liang, Hongcheng Guo, and Zhoujun Li. 2023a. KnowPrefix-Tuning: A two-stage prefix-tuning framework for knowledge-grounded dialogue generation. *arXiv preprint arXiv:2306.15430*.

Yu Bai, Fan Chen, Huan Wang, Caiming Xiong, and Song Mei. 2023b. Transformers as statisticians: Provable in-context learning with in-context algorithm selection. In *Advances in Neural Information Processing Systems*.

Luke Bailey, Gustaf Ahdritz, Anat Kleiman, Siddharth Swaroop, Finale Doshi-Velez, and Weiwei Pan. 2023. Soft prompting might be a bug, not a feature. In *Workshop on Challenges in Deployable Generative AI at International Conference on Machine Learning*.

Federico Barbero, Andrea Banino, Steven Kapturowski, Dharshan Kumaran, João Madeira Araújo, Oleksandr Vitvitskyi, Razvan Pascanu, and Petar Veličković. 2024. Transformers need glasses! Information over-squashing in language tasks. In *Advances in Neural Information Processing Systems*.

Alex Barnett. 2021. Lower bounds on the modified Bessel function of the first kind. Mathematics Stack Exchange.

Andrew R Barron. 1993. Universal approximation bounds for superpositions of a sigmoidal function. *IEEE Transactions on Information Theory*, 39(3):930–945.

Iz Beltagy, Matthew E Peters, and Arman Cohan. 2020. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*.

Yoshua Bengio, Patrice Simard, and Paolo Frasconi. 1994. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166.

Satwik Bhattamishra, Arkil Patel, and Navin Goyal. 2020. On the computational power of transformers and its implications in sequence modeling. In *Proceedings of the 24th Conference on Computational Natural Language Learning*.

Edward K Blum and Leong Kwan Li. 1991. Approximation theory and feedforward networks. *Neural networks*, 4(4):511–515.

Helmut Bölcskei, Philipp Grohs, Gitta Kutyniok, and Philipp Petersen. 2019. Optimal approximation with sparsely connected deep neural networks. *SIAM Journal on Mathematics of Data Science*, 1(1):8–45.

Aleksandar Botev, Soham De, Samuel L Smith, Anushan Fernando, George-Cristian Muraru, Ruba Haroun, Leonard Berrada, Razvan Pascanu, Pier Giuseppe Sessa, Robert Dadashi, et al. 2024. RecurrentGemma: Moving past transformers for efficient open language models. *arXiv preprint arXiv:2404.07839*.

Stephen Boyd and Leon Chua. 1985. Fading memory and the problem of approximating nonlinear operators with Volterra series. *IEEE Transactions on Circuits and Systems*, 32(11):1150–1161.

Shaked Brody, Uri Alon, and Eran Yahav. 2023. On the expressivity role of LayerNorm in transformers' attention. *arXiv preprint arXiv:2305.02582*.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in Neural Information Processing Systems*.

Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, Harsha Nori, Hamid Palangi, Marco Tulio Ribeiro, and Yi Zhang. 2023. Sparks of artificial general intelligence: Early experiments with GPT-4. *Preprint arXiv:2303.12712*.

Neal L Carothers. 1998. A short course on approximation theory. *Bowling Green State University, Bowling Green, OH*, 38.

S.M. Carroll and B. W. Dickinson. 1989. Construction of neural nets using the Radon transform. In *International 1989 Joint Conference on Neural Networks*, pages 607–611.

Patrick Chao, Alexander Robey, Edgar Dobriban, Hamed Hassani, George J. Pappas, and Eric Wong. 2023. Jailbreaking black box large language models in twenty queries. *arXiv preprint arXiv:2310.08419*.

David Chiang, Peter Cholak, and Anand Pillay. 2023. Tighter bounds on the expressivity of transformer encoders. In *International Conference on Machine Learning*.

Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

YunSeok Choi and Jee-Hyong Lee. 2023. CodePrompt: Task-agnostic prefix tuning for program and language generation. In *Findings of the Association for Computational Linguistics: ACL 2023*.

Dan Ciregan, Ueli Meier, and Jürgen Schmidhuber. 2012. Multi-column deep neural networks for image classification. In *2012 IEEE conference on computer vision and pattern recognition*, pages 3642–3649. IEEE.

Nicola Muca Cirone, Antonio Orvieto, Benjamin Walker, Cristopher Salvi, and Terry Lyons. 2024. Theoretical foundations of deep selective state-space models. In *Advances in Neural Information Processing Systems*.

Julian Coda-Forno, Marcel Binz, Zeynep Akata, Matt Botvinick, Jane Wang, and Eric Schulz. 2023. Meta-in-context learning in large language models. In *Advances in Neural Information Processing Systems*, pages 65189–65201.

Arthur Conmy, Augustine Mavor-Parker, Aengus Lynch, Stefan Heimersheim, and Adrià Garriga-Alonso. 2023. Towards automated circuit discovery for mechanistic interpretability. *Advances in Neural Information Processing Systems*.

Corinna Cortes and Vladimir Vapnik. 1995. Support-vector networks. *Machine Learning*, 20:273–297.

Ana Santos Costa, Montserrat Comesaña, and Ana Paula Soares. 2022. PHOR-in-One: A multilingual lexical database with PHonological, ORthographic and PHonographic word similarity estimates in four languages. *Behavior Research Methods*.

Bruce Curry and P Morgan. 1997. Neural networks: A need for caution. *Omega*, 25(1):123–133.

George Cybenko. 1989. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314.

Damai Dai, Yutao Sun, Li Dong, Yaru Hao, Shuming Ma, Zhifang Sui, and Furu Wei. 2023. Why can GPT learn in-context? Language models secretly perform gradient descent as meta-optimizers. In *Findings of the Association for Computational Linguistics: ACL 2023*.

Feng Dai and Yuan Xu. 2013. *Approximation Theory and Harmonic Analysis on Spheres and Balls*.

Soham De, Samuel L Smith, Anushan Fernando, Aleksandar Botev, George Cristian-Muraru, Albert Gu, Ruba Haroun, Leonard Berrada, Yutian Chen, Srivatsan Srinivasan, et al. 2024. Griffin: Mixing gated linear recurrences with local attention for efficient language models. *arXiv preprint arXiv:2402.19427*.

Christian Schroeder de Witt, Samuel Sokota, J. Zico Kolter, Jakob Foerster, and Martin Strohmeier. 2023. Perfectly secure steganography using minimum entropy coupling. In *International Conference on Learning Representations*.

David Demeter, Gregory Kimmel, and Doug Downey. 2020. Stolen probability: A structural weakness of neural language models. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*.

Puneesh Deora, Rouzbeh Ghaderi, Hossein Taheri, and Christos Thrampoulidis. 2023. On the optimization and generalization of multi-head attention. *arXiv preprint arXiv:2310.12680*.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. 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)*.

Ronald A DeVore, Ralph Howard, and Charles Micchelli. 1989. Optimal nonlinear approximation. *Manuscripta Mathematica*, 63:469–478.

Yihe Dong, Jean-Baptiste Cordonnier, and Andreas Loukas. 2021. Attention is not all you need: Pure attention loses rank doubly exponentially with depth. In *International Conference on Machine Learning*.

Li Du, Lucas Torroba Hennigen, Tiago Pimentel, Clara Meister, Jason Eisner, and Ryan Cotterell. 2022. A measure-theoretic characterization of tight language models. *arXiv preprint arXiv:2212.10502*.

Francisco Eiras, Aleksandar Petrov, Philip H.S. Torr, M Pawan Kumar, and Adel Bibi. 2025. Do as I do (safely): Mitigating task-specific fine-tuning risks in large language models. In *International Conference on Learning Representations (ICLR)*.

Francisco Eiras, Aleksandar Petrov, Bertie Vidgen, Christian Schroeder de Witt, Fabio Pizzati, Katherine Elkins, Supratik Mukhopadhyay, Adel Bibi, Botos Csaba, Fabro Steibel, et al. 2024. Position: Near to mid-term risks and opportunities of open-source generative AI. In *International Conference on Machine Learning (ICML)*.

Yanai Elazar, Akshita Bhagia, Ian Helgi Magnusson, Abhilasha Ravichander, Dustin Schwenk, Alane Suhr, Evan Pete Walsh, Dirk Groeneveld, Luca Soldaini, Sameer Singh, Hannaneh Hajishirzi, Noah A. Smith, and Jesse Dodge. 2024. What's in my big data? In *The Twelfth International Conference on Learning Representations*.

Nelson Elhage, Neel Nanda, Catherine Olsson, Tom Henighan, Nicholas Joseph, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, Tom Conerly, Nova DasSarma, Dawn Drain, Deep Ganguli, Zac Hatfield-Dodds, Danny Hernandez, Andy Jones, Jackson Kernion, Liane Lovitt, Kamal Ndousse, Dario Amodei, Tom Brown, Jack Clark, Jared Kaplan, Sam McCandlish, and Chris Olah. 2021. A mathematical framework for transformer circuits. *Transformer Circuits Thread*.

Ricardo Estrada. 2014. On radial functions and distributions and their Fourier transforms. *Journal of Fourier Analysis and Applications*, 20(2):301–320.

Uriel Feige and Gideon Schechtman. 2002. On the optimality of the random hyperplane rounding technique for MAX CUT. *Random Structures & Algorithms*, 20(3):403–440.

Dan Friedman, Alexander Wettig, and Danqi Chen. 2023. Learning transformer programs. In *Advances in Neural Information Processing Systems*.

Daniel Y Fu, Tri Dao, Khaled Kamal Saab, Armin W Thomas, Atri Rudra, and Christopher Re. 2023a. Hungry Hungry Hippos: Towards language modeling with state space models. In *International Conference on Learning Representations*.

Deqing Fu, Tian-Qi Chen, Robin Jia, and Vatsal Sharan. 2023b. Transformers learn higher-order optimization methods for in-context learning: A study with linear models. *arXiv preprint arXiv:2310.17086*.

Ken-Ichi Funahashi. 1989. On the approximate realization of continuous mappings by neural networks. *Neural Networks*, 2(3):183–192.

Paul Funk. 1915. Beiträge zur Theorie der Kugelfunktionen. *Mathematische Annalen*, 77:136–152.

Takashi Furuya, Maarten V. de Hoop, and Gabriel Peyre. 2025. Transformers are universal in-context learners. In *International Conference on Learning Representations*.

A. Ronald Gallant and Halbert White. 1988. There exists a neural network that does not make avoidable mistakes. In *IEEE 1988 International Conference on Neural Networks*, pages 657–664.

Shivam Garg, Dimitris Tsipras, Percy S Liang, and Gregory Valiant. 2022. What can transformers learn in-context? A case study of simple function classes. In *Advances in Neural Information Processing Systems*.

Atticus Geiger, Zhengxuan Wu, Christopher Potts, Thomas Icard, and Noah Goodman. 2024. Finding alignments between interpretable causal variables and distributed neural representations. In *Causal Learning and Reasoning*.

Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. 2000. Learning to forget: Continual prediction with LSTM. *Neural computation*, 12(10):2451–2471.

Angeliki Giannou, Shashank Rajput, Jy-Yong Sohn, Kangwook Lee, Jason D. Lee, and Dimitris Papailiopoulos. 2023. Looped transformers as programmable computers. In *International Conference on Machine Learning*.

Federico Girosi and Tomaso Poggio. 1989. Representation properties of networks: Kolmogorov's theorem is irrelevant. *Neural Computation*, 1(4):465–469.

Albert Gu and Tri Dao. 2023. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*.

Albert Gu, Tri Dao, Stefano Ermon, Atri Rudra, and Christopher Ré. 2020. HiPPO: Recurrent memory with optimal polynomial projections. In *Advances in Neural Information Processing Systems*.

Albert Gu, Karan Goel, and Christopher Re. 2021. Efficiently modeling long sequences with structured state spaces. In *International Conference on Learning Representations*.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Ankit Gupta, Albert Gu, and Jonathan Berant. 2022. Diagonal state spaces are as effective as structured state spaces. In *Advances in Neural Information Processing Systems*.

Michael Hahn. 2020. Theoretical limitations of self-attention in neural sequence models. *Transactions of the Association for Computational Linguistics*.

Karen Hambardzumyan, Hrant Khachatrian, and Jonathan May. 2021. WARP: Word-level Adversarial ReProgramming. 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)*.

Mohamed Hamdouche and Alex Sala. 2021. Dynamic Kolmogorov approach to RNN universality.

Chi Han, Ziqi Wang, Han Zhao, and Heng Ji. 2023. In-context learning of large language models explained as kernel regression. *arXiv preprint arXiv:2305.12766*.

Joshua Hanson and Maxim Raginsky. 2020. Universal simulation of stable dynamical systems by recurrent neural nets. In *Learning for Dynamics and Control*.

Yiding Hao, Dana Angluin, and Robert Frank. 2022. Formal language recognition by hard attention transformers: Perspectives from circuit complexity. *Transactions of the Association for Computational Linguistics*.

Mohamad H Hassoun. 1995. *Fundamentals of artificial neural networks*. MIT press.

Junxian He, Chunting Zhou, Xuezhe Ma, Taylor Berg-Kirkpatrick, and Graham Neubig. 2021a. Towards a unified view of parameter-efficient transfer learning. In *International Conference on Learning Representations*.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and pattern Recognition*.

Tianxing He, Jun Liu, Kyunghyun Cho, Myle Ott, Bing Liu, James Glass, and Fuchun Peng. 2021b. Analyzing the forgetting problem in pretrain-finetuning of open-domain dialogue response models. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*.

E Hecke. 1917. Über orthogonal-invariante Integralgleichungen. *Mathematische Annalen*, 78:398–404.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Computation*, 9(8):1735–1780.

Kurt Hornik. 1993. Some new results on neural network approximation. *Neural Networks*, 6(8):1069–1072.

Kurt Hornik, Maxwell Stinchcombe, and Halbert White. 1989. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366.

Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-efficient transfer learning for NLP. In *International Conference on Machine Learning*.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

Zhiqiang Hu, Yihuai Lan, Lei Wang, Wanyu Xu, Ee-Peng Lim, Roy Ka-Wei Lee, Lidong Bing, and Soujanya Poria. 2023. LLM-Adapters: An adapter family for parameter-efficient fine-tuning of large language models. *arXiv preprint arXiv:2304.01933*.

Bunpei Irie and Sei Miyake. 1988. Capabilities of three-layered perceptrons. In *IEEE 1988 International Conference on Neural Networks*, pages 641–648.

Sarthak Jain and Byron C. Wallace. 2019. Attention is not explanation. 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)*.

Siddhant M Jayakumar, Wojciech M Czarnecki, Jacob Menick, Jonathan Schwarz, Jack Rae, Simon Osindero, Yee Whye Teh, Tim Harley, and Razvan Pascanu. 2020. Multiplicative interactions and where to find them. In *International Conference on Learning Representations*.

Haotian Jiang and Qianxiao Li. 2024. Approximation rate of the transformer architecture for sequence modeling. In *Advances in Neural Information Processing Systems*.

Haotian Jiang, Qianxiao Li, Zhong Li, and Shida Wang. 2023. A brief survey on the approximation theory for sequence modelling. *arXiv preprint arXiv:2302.13752*.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*.

Andrej Karpathy. 2020. minGPT GitHub Repository.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in Neural Information Processing Systems*.

Daphne Koller and Nir Friedman. 2009. *Probabilistic graphical models: Principles and techniques*. MIT Press.

Andrei Nikolaevich Kolmogorov. 1957. On the representation of continuous functions of many variables by superposition of continuous functions of one variable and addition. In *Doklady Akademii Nauk*, volume 114, pages 953–956. Russian Academy of Sciences.

Jannik Kossen, Tom Rainforth, and Yarin Gal. 2023. In-context learning in large language models learns label relationships but is not conventional learning. *arXiv preprint arXiv:2307.12375*.

Anastasis Kratsios and Takashi Furuya. 2025. Is in-context universality enough? MLPs are also universal in-context. *arXiv preprint arXiv:2502.03327*.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc.

Emanuele La Malfa, Aleksandar Petrov, Simon Frieder, Christoph Weinhuber, Ryan Burnell, Raza Nazar, Anthony G. Cohn, Nigel Shadbolt, and Michael Wooldridge. 2024. Language-Models-as-a-Service: Overview of a new paradigm and its challenges. *Journal of Artificial Intelligence Research*, 80.

Emanuele La Malfa, Christoph Weinhuber, Orazio Torre, Fangru Lin, Anthony Cohn, Nigel Shadbolt, and Michael Wooldridge. 2024. Code simulation challenges for large language models. *arXiv preprint arXiv:2401.09074*.

Ivan Lee, Nan Jiang, and Taylor Berg-Kirkpatrick. 2024. Exploring the relationship between model architecture and in-context learning ability. In *International Conference on Learning Representations*.

Moshe Leshno, Vladimir Ya. Lin, Allan Pinkus, and Shimon Schocken. 1993. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Networks*, 6(6):861–867.

Brian Lester, Rami Al-Rfou, and Noah Constant. 2021. The power of scale for parameter-efficient prompt tuning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*.

Shengqiao Li. 2010. Concise formulas for the area and volume of a hyperspherical cap. *Asian Journal of Mathematics & Statistics*, 4(1):66–70.

Xiang Lisa Li and Percy Liang. 2021. Prefix-Tuning: Optimizing continuous prompts for generation. 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).

Yingcong Li, Muhammed Emrullah Ildiz, Dimitris Papailiopoulos, and Samet Oymak. 2023. Transformers as algorithms: Generalization and stability in in-context learning. In *International Conference on Machine Learning*.

Zihao Li, Yuan Cao, Cheng Gao, Yihan He, Han Liu, Jason Klusowski, Jianqing Fan, and Mengdi Wang. 2024. One-layer transformer provably learns one-nearest neighbor in context. In *Advances in Neural Information Processing Systems*.

Vladislav Lialin, Vijeta Deshpande, and Anna Rumshisky. 2023. Scaling down to scale up: A guide to parameter-efficient fine-tuning. *arXiv preprint arXiv:2303.15647*.

Valerii Likhosherstov, Krzysztof Choromanski, and Adrian Weller. 2021. On the expressive power of self-attention matrices. *arXiv preprint arXiv:2106.03764*.

David Lindner, János Kramár, Sebastian Farquhar, Matthew Rahtz, Tom McGrath, and Vladimir Mikulik. 2023. Tracr: Compiled transformers as a laboratory for interpretability. In *Advances in Neural Information Processing Systems*.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024a. DeepSeek-V3 technical report. *arXiv preprint arXiv:2412.19437*.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024b. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*.

Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys*.

Vivian Liu and Lydia B Chilton. 2022. Design guidelines for prompt engineering text-to-image generative models. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*.

Xiao Liu, Kaixuan Ji, Yicheng Fu, Weng Tam, Zhengxiao Du, Zhilin Yang, and Jie Tang. 2022. P-Tuning: Prompt tuning can be comparable to fine-tuning across scales and tasks. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*.

Sheng Lu, Irina Bigoulaeva, Rachneet Sachdeva, Harish Tayyar Madabushi, and Iryna Gurevych. 2024. Are emergent abilities in large language models just in-context learning? In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*.

Yun Luo, Zhen Yang, Fandong Meng, Yafu Li, Jie Zhou, and Yue Zhang. 2023. An empirical study of catastrophic forgetting in large language models during continual fine-tuning. *arXiv preprint arXiv:2308.08747*.

Sadegh Mahdavi, Renjie Liao, and Christos Thrampoulidis. 2023. Memorization capacity of multi-head attention in transformers. *arXiv preprint arXiv:2306.02010*.

Vitaly Maiorov and Allan Pinkus. 1999. Lower bounds for approximation by MLP neural networks. *Neurocomputing*, 25(1-3):81–91.

Gary Marcus. 2023. The sparks of AGI? Or the end of science? *BLOG@CACM*.

Eric Martínez. 2024. Re-evaluating GPT-4's bar exam performance. *Artificial Intelligence and Law*.

Valdir Antônio Menegatto. 1997. Approximation by spherical convolution. *Numerical Functional Analysis and Optimization*, 18(9-10):995–1012.

William Merrill and Ashish Sabharwal. 2023a. A logic for expressing log-precision transformers. In *Advances in Neural Information Processing Systems*.

William Merrill and Ashish Sabharwal. 2023b. The parallelism tradeoff: Limitations of log-precision transformers. *Transactions of the Association for Computational Linguistics*.

William Merrill and Ashish Sabharwal. 2024. The expressive power of transformers with chain of thought. In *The Twelfth International Conference on Learning Representations*.

Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, Amit Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. 2017. SymPy: Symbolic computing in Python. *PeerJ Computer Science*, 3.

Hrushikesh N Mhaskar. 1996. Neural networks for optimal approximation of smooth and analytic functions. *Neural Computation*, 8(1):164–177.

Bonan Min, Hayley Ross, Elior Sulem, Amir Pouran Ben Veyseh, Thien Huu Nguyen, Oscar Sainz, Eneko Agirre, Ilana Heintz, and Dan Roth. 2021. Recent advances in natural language processing via large pre-trained language models: A survey. *ACM Computing Surveys*.

Jishnu Mukhoti, Yarin Gal, Philip H. S. Torr, and Puneet K. Dokania. 2023. Fine-tuning can cripple your foundation model; preserving features may be the solution. *arXiv preprint arXiv:2308.13320*.

Ware Myers. 1986. Introduction to expert systems. *IEEE Intelligent Systems*, 1(1):100–109.

Linyong Nan, Dragomir Radev, Rui Zhang, Amrit Rau, Abhinand Sivaprasad, Chiachun Hsieh, Xiangru Tang, Aadit Vyas, Neha Verma, Pranav Krishna, Yangxiaokang Liu, Nadia Irwanto, Jessica Pan, Faiaz Rahman, Ahmad Zaidi, Mutethia Mutuma, Yasin Tarabar, Ankit Gupta, Tao Yu, Yi Chern Tan, Xi Victoria Lin, Caiming Xiong, Richard Socher, and Nazneen Fatema Rajani. 2021. DART: Open-domain structured data record to text generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*.

Swaroop Nath, Harshad Khadilkar, and Pushpak Bhattacharyya. 2024. Transformers are expressive, but are they expressive enough for regression? *arXiv preprint arXiv:2402.15478*.

Tin Lok James Ng and Kwok-Kun Kwong. 2022. Universal approximation on the hypersphere. *Communications in Statistics – Theory and Methods*, 51(24):8694–8704.

Naoki Nishikawa and Taiji Suzuki. 2024. State space models are comparable to transformers in estimating functions with dynamic smoothness. *arXiv preprint arXiv:2405.19036*.

Franz Nowak, Anej Svete, Alexandra Butoi, and Ryan Cotterell. 2024. On the representational capacity of neural language models with chain-of-thought reasoning. *arXiv preprint arXiv:2406.14197*.

Franz Nowak, Anej Svete, Li Du, and Ryan Cotterell. 2023. On the representational capacity of recurrent neural language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*.

Christian W Omlin and C Lee Giles. 1996. Constructing deterministic finite-state automata in recurrent neural networks. *Journal of the ACM (JACM)*, 43(6):937–972.

Antonio Orvieto, Soham De, Caglar Gulcehre, Razvan Pascanu, and Samuel L Smith. 2024. Universality of linear recurrences followed by non-linear projections: Finite-width guarantees and benefits of complex eigenvalues. In *Proceedings of the 41st International Conference on Machine Learning*.

Antonio Orvieto, Samuel L Smith, Albert Gu, Anushan Fernando, Caglar Gulcehre, Razvan Pascanu, and Soham De. 2023. Resurrecting recurrent neural networks for long sequences. In *International Conference on Machine Learning*.

Yawen Ouyang, Yongchang Cao, Yuan Gao, Zhen Wu, Jianbing Zhang, and Xinyu Dai. 2023. On prefix-tuning for lightweight out-of-distribution detection. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*.

Abhishek Panigrahi, Nikunj Saunshi, Haoyu Zhao, and Sanjeev Arora. 2023. Task-specific skill localization in fine-tuned language models. In *International Conference on Machine Learning*.

Sejun Park, Chulhee Yun, Jaeho Lee, and Jinwoo Shin. 2021. Minimum width for universal approximation. In *International Conference on Learning Representations*.

Aleksandar Petrov, Shruti Agarwal, Philip H.S. Torr, Adel Bibi, and John Collomosse. 2025a. On the coexistence and ensembling of watermarks. In *Advances in Neural Information Processing Systems*.

Aleksandar Petrov, Francisco Eiras, Amartya Sanyal, Philip H.S. Torr, and Adel Bibi. 2023a. Certifying ensembles: A general certification theory with S-Lipschitzness. *International Conference on Machine Learning (ICML)*.

Aleksandar Petrov, Pierre Fernandez, Tomáš Souček, and Hady Elsahar. 2025b. We can hide more bits: The unused watermarking capacity in theory and in practice. *Preprint arXiv:2510.12812*.

Aleksandar Petrov and Marta Kwiatkowska. 2022. Robustness of unsupervised representation learning without labels. *Preprint arXiv:2210.04076*.

Aleksandar Petrov, Emanuele La Malfa, Philip H.S. Torr, and Adel Bibi. 2023b. Language model tokenizers introduce unfairness between languages. In *Advances in Neural Information Processing Systems*.

Aleksandar Petrov, Tom A Lamb, Alasdair Paren, Philip H.S. Torr, and Adel Bibi. 2024a. Universal in-context approximation by prompting fully recurrent models. In *Advances in Neural Information Processing Systems*.

Aleksandar Petrov, Mark Sandler, Andrey Zhmoginov, Nolan Miller, and Max Vladymyrov. 2025c. Long context in-context compression by getting to the gist of gisting. *Preprint arXiv:2504.08934*.

Aleksandar Petrov, Philip H.S. Torr, and Adel Bibi. 2024b. Prompting a pretrained transformer can be a universal approximator. In *International Conference on Machine Learning (ICML)*.

Aleksandar Petrov, Philip H.S. Torr, and Adel Bibi. 2024c. When do prompting and prefix-tuning work? A theory of capabilities and limitations. In *International Conference on Learning Representations (ICLR)*.

Allan Pinkus. 1999. Approximation theory of the MLP model in neural networks. *Acta Numerica*, 8:143–195.

Jorge Pérez, Pablo Barceló, and Javier Marinkovic. 2021. Attention is Turing-complete. *Journal of Machine Learning Research*.

Guanghui Qin and Jason Eisner. 2021. Learning how to ask: Querying LMs with mixtures of soft prompts. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*.

Yujia Qin, Xiaozhi Wang, Yusheng Su, Yankai Lin, Ning Ding, Jing Yi, Weize Chen, Zhiyuan Liu, Juanzi Li, Lei Hou, et al. 2021. Exploring universal intrinsic task subspace via prompt tuning. *arXiv preprint arXiv:2110.07867*.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners.

Jack W. Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides, Sarah Henderson, Roman Ring, et al. 2021. Scaling language models: Methods, analysis & insights from training Gopher.

David L Ragozin. 1971. Constructive polynomial approximation on spheres and projective spaces. *Transactions of the American Mathematical Society*, 162:157–170.

Carl Rasmussen and Christopher Williams. 2006. *Gaussian Processes for Machine Learning*. MIT Press.

Sylvestre-Alvise Rebuffi, Hakan Bilen, and Andrea Vedaldi. 2017. Learning multiple visual domains with residual adapters. In *Advances in Neural Information Processing Systems*.

Laria Reynolds and Kyle McDonell. 2021. Prompt programming for large language models: Beyond the few-shot paradigm. In *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*.

Anna Rogers and Sasha Luccioni. 2024. Position: Key claims in LLM research have a long tail of footnotes. In *Forty-first International Conference on Machine Learning*.

C. A. Rogers. 1963. Covering a sphere with spheres. *Mathematika*, 10(2):157–164.

Pranab Sahoo, Ayush Kumar Singh, Sriparna Saha, Vinija Jain, Samrat Mondal, and Aman Chadha. 2024. A systematic survey of prompt engineering in large language models: Techniques and applications. *arXiv preprint arXiv:2402.07927*.

Oscar Sainz, Iker García-Ferrero, Alon Jacovi, Jon Ander Campos, Yanai Elazar, Eneko Agirre, Yoav Goldberg, Wei-Lin Chen, Jenny Chim, Leshem Choshen, et al. 2024. Data contamination report from the 2024 CONDA shared task. *arXiv preprint arXiv:2407.21530*.

Clayton Sanford, Daniel Hsu, and Matus Telgarsky. 2023. Representational strengths and limitations of transformers. *arXiv preprint arXiv:2306.02896*.

Clayton Sanford, Daniel Hsu, and Matus Telgarsky. 2024. Transformers, parallel computation, and logarithmic depth. *arXiv preprint arXiv:2402.09268*.

Elvis Saravia, Hsien-Chi Toby Liu, Yen-Hao Huang, Junlin Wu, and Yi-Shin Chen. 2018. CARER: Contextualized affect representations for emotion recognition. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*.

Yash Sarrof, Yana Veitsman, and Michael Hahn. 2024. The expressive capacity of state space models: A formal language perspective. *arXiv preprint arXiv:2405.17394*.

Franco Scarselli and Ah Chung Tsoi. 1998. Universal approximation using feedforward neural networks: A survey of some existing methods, and some new results. *Neural networks*, 11(1):15–37.

Rylan Schaeffer, Brando Miranda, and Sanmi Koyejo. 2023. Are emergent abilities of large language models a mirage? In *Advances in Neural Information Processing Systems*.

Anton Maximilian Schäfer and Hans-Georg Zimmermann. 2007. Recurrent neural networks are universal approximators. *International Journal of Neural Systems*, 17(04):253–263.

Johannes Schmidt-Hieber. 2021. The Kolmogorov–Arnold representation theorem revisited. *Neural Networks*, 137:119–126.

Noam Shazeer. 2019. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*.

Taylor Shin, Yasaman Razeghi, Robert L. Logan IV, Eric Wallace, and Sameer Singh. 2020. AutoPrompt: Eliciting knowledge from language models with automatically generated prompts. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Hava T. Siegelmann and Eduardo D. Sontag. 1995. On the computational power of neural nets. *Journal of Computer and System Sciences*, 50(1):132–150.

Karen Simonyan and Andrew Zisserman. 2015. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations*.

Mirko Solazzi and Aurelio Uncini. 2004. Regularising neural networks using flexible multivariate activation function. *Neural Networks*, 17(2):247–260.

Chang hoon Song, Geonho Hwang, Jun ho Lee, and Myungjoo Kang. 2023. Minimal width for universal property of deep RNN. *Journal of Machine Learning Research*.

Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. 2015. Highway networks. *arXiv preprint arXiv:1505.00387*.

Lena Strobl. 2023. Average-hard attention transformers are constant-depth uniform threshold circuits. *arXiv preprint arXiv:2308.03212*.

Lena Strobl, William Merrill, Gail Weiss, David Chiang, and Dana Angluin. 2024. What formal languages can transformers express? A survey. *Transactions of the Association for Computational Linguistics*.

Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. 2024. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568.

Yusheng Su, Xiaozhi Wang, Yujia Qin, Chi-Min Chan, Yankai Lin, Huadong Wang, Kaiyue Wen, Zhiyuan Liu, Peng Li, Juanzi Li, Lei Hou, Maosong Sun, and Jie Zhou. 2022. On transferability of prompt tuning for natural language processing. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*.

Tianxiang Sun, Yunfan Shao, Hong Qian, Xuanjing Huang, and Xipeng Qiu. 2022. Black-box tuning for language-model-as-a-service. In *International Conference on Machine Learning*.

Anej Svete, Robin Shing Moon Chan, and Ryan Cotterell. 2024. On efficiently representing regular languages as RNNs. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*.

Garrett Tanzer, Mirac Suzgun, Eline Visser, Dan Jurafsky, and Luke Melas-Kyriazi. 2024. A benchmark for learning to translate a new language from one grammar book. In *The Twelfth International Conference on Learning Representations*.

Team Gemma, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, et al. 2024. Gemma: Open models based on Gemini research and technology. *arXiv preprint arXiv:2403.08295*.

Matus Telgarsky. 2015. Representation benefits of deep feedforward networks. *arXiv preprint arXiv:1509.08101*.

Matus Telgarsky. 2016. Benefits of depth in neural networks. In *Conference on Learning Theory*, pages 1517–1539.

Matus Telgarsky. 2021. Deep learning theory lecture notes.

Aleksandar Terzic, Michael Hersche, Giacomo Camposampiero, Thomas Hofmann, Abu Sebastian, and Abbas Rahimi. 2025. On the expressiveness and length generalization of selective state-space models on regular languages. In *AAAI Conference on Artificial Intelligence*.

Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, et al. 2022. LaMDA: Language models for dialog applications. *arXiv preprint arXiv:2201.08239*.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. LLaMA: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Vishaal Udandarao, Ameya Prabhu, Adhiraj Ghosh, Yash Sharma, Philip H.S. Torr, Adel Bibi, Samuel Albanie, and Matthias Bethge. 2024. No "zero-shot"' without exponential data: Pretraining concept frequency determines multimodal model performance. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*.

Petar Veličković, Christos Perivolaropoulos, Federico Barbero, and Razvan Pascanu. 2024. softmax is not enough (for sharp out-of-distribution). *arXiv preprint arXiv:2410.01104*.

Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. 2020. SciPy 1.0: Fundamental algorithms for scientific computing in Python. *Nature Methods*, 17:261–272.

Eline Visser. 2022. *A grammar of Kalamang*. Language Science Press.

Johannes von Oswald, Eyvind Niklasson, Ettore Randazzo, João Sacramento, Alexander Mordvintsev, Andrey Zhmoginov, and Max Vladymyrov. 2023a. Transformers learn in-context by gradient descent. In *International Conference on Machine Learning*.

Johannes von Oswald, Eyvind Niklasson, Maximilian Schlegel, Seijin Kobayashi, Nicolas Zucchet, Nino Scherrer, Nolan Miller, Mark Sandler, Max Vladymyrov, Razvan Pascanu, and João Sacramento. 2023b. Uncovering mesa-optimization algorithms in transformers. *arXiv preprint arXiv:2309.05858*.

Tu Vu, Brian Lester, Noah Constant, Rami Al-Rfou', and Daniel Cer. 2022. SPoT: Better frozen model adaptation through soft prompt transfer. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*.

Shida Wang and Beichen Xue. 2023. State-space models with layer-wise nonlinearity are universal approximators with exponential decaying memory. In *Advances in Neural Information Processing Systems*.

Xiaozhi Wang, Kaiyue Wen, Zhengyan Zhang, Lei Hou, Zhiyuan Liu, and Juanzi Li. 2022a. Finding skill neurons in pre-trained transformer-based language models. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*.

Yihan Wang, Jatin Chauhan, Wei Wang, and Cho-Jui Hsieh. 2023. Universality and limitations of prompt tuning. In *Advances in Neural Information Processing Systems*.

Zhen Wang, Rameswar Panda, Leonid Karlinsky, Rogerio Feris, Huan Sun, and Yoon Kim. 2022b. Multitask prompt tuning enables parameter-efficient transfer learning. In *International Conference on Learning Representations*.

Jason Wei, Maarten Bosma, Vincent Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. 2021. Finetuned language models are zero-shot learners. In *International Conference on Learning Representations*.

Gail Weiss, Yoav Goldberg, and Eran Yahav. 2021. Thinking like transformers. In *International Conference on Machine Learning*.

Joseph Weizenbaum. 1966. ELIZA - A computer program for the study of natural language communication between man and machine. *Communications of the ACM*, 9(1):36–45.

Sarah Wiegreffe and Yuval Pinter. 2019. Attention is not not explanation. 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)*.

Noam Wies, Yoav Levine, and Amnon Shashua. 2023. The learnability of in-context learning. *arXiv preprint arXiv:2303.07895*.

Yotam Wolf, Noam Wies, Oshri Avnery, Yoav Levine, and Amnon Shashua. 2023. Fundamental limitations of alignment in large language models. *arXiv preprint arXiv:2304.11082*.

Hui Wu and Xiaodong Shi. 2022. Adversarial soft prompt tuning for cross-domain sentiment analysis. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*.

Sang Michael Xie, Aditi Raghunathan, Percy Liang, and Tengyu Ma. 2021. An explanation of in-context learning as implicit Bayesian inference. In *International Conference on Learning Representations*.

Steve Yadlowsky, Lyric Doshi, and Nilesh Tripuraneni. 2023. Pretraining data mixtures enable narrow model selection capabilities in transformer models. *arXiv preprint arXiv:2311.00871*.

Dmitry Yarotsky. 2017. Error bounds for approximations with deep ReLU networks. *Neural Networks*, 94:103–114.

Chulhee Yun, Srinadh Bhojanapalli, Ankit Singh Rawat, Sashank Reddi, and Sanjiv Kumar. 2020. Are transformers universal approximators of sequence-to-sequence functions? In *International Conference on Learning Representations*.

Biao Zhang and Rico Sennrich. 2019. Root mean square layer normalization. In *Advances in Neural Information Processing Systems*.

Ruiqi Zhang, Spencer Frei, and Peter L Bartlett. 2023. Trained transformers learn linear models in-context. *arXiv preprint arXiv:2306.09927*.

Yuanhang Zheng, Zhixing Tan, Peng Li, and Yang Liu. 2023. Black-box prompt tuning with subspace learning. *arXiv preprint arXiv:2305.03518*.

Qihuang Zhong, Liang Ding, Juhua Liu, Bo Du, and Dacheng Tao. 2022. PANDA: Prompt transfer meets knowledge distillation for efficient model adaptation. *arXiv preprint arXiv:2208.10160*.

Ding-Xuan Zhou. 2020. Universality of deep convolutional neural networks. *Applied and Computational Harmonic Analysis*, 48(2):787–794.

Hattie Zhou, Arwen Bradley, Etai Littwin, Noam Razin, Omid Saremi, Josh Susskind, Samy Bengio, and Preetum Nakkiran. 2024. What algorithms can transformers learn? A study in length generalization. In *International Conference on Learning Representations*.

Liu Ziyin, Tilman Hartwig, and Masahito Ueda. 2020. Neural networks fail to learn periodic functions and how to fix it. In *Advances in Neural Information Processing Systems*.

Andy Zou, Zifan Wang, J Zico Kolter, and Matt Fredrikson. 2023. Universal and transferable adversarial attacks on aligned language models. *arXiv preprint arXiv:2307.15043*.

Appendix A

Appendices for When Do Prompting and Prefix-Tuning Work? A Theory of Capabilities and Limitations

A.1 Constructing transformers that utilize the capacity of the embedding space

A.1.1 Unconditional generation for a single virtual token

This section provides an explicit construction of a transformer with the properties described in Theorem 3.1. The goal is to construct a transformer that, by varying the choice of the virtual token, can generate any sequence of N tokens.

First, we need to specify how we encode the target sequence $(\mathtt{Y}_1, \dots, \mathtt{Y}_N)$ into the virtual token s_1. We chose the size of the embedding (and hence of s_1) to be N. This way, each element of s_1 can represent one position of the target sequence. We then represent the token value by discretizing each element of s_1 into V levels:

$$s_1 = \left((\mathtt{Y}_1 - 1)/V, \dots, (\mathtt{Y}_N - 1)/V \right).$$

Note that this means that each element of s_1 is in $[0, 1)$.

When predicting the token for the $i + 1$ position, the transformer needs to pick the i-th element of s_1, and then decode the corresponding value as a one-hot encoding representing the $\mathtt{Y}_i$-th token.

We extract the i-th element of s_1 using one attention block of two heads. The `fst` head always looks at the first position which is our virtual token s_1. For that purpose we create

an attention head that always has $A^{\text{fst}}_{ij} = 1$ if $j = 1$ and $A^{\text{fst}}_{ij} = 0$ otherwise together with a value matrix W^{fst}_V that extracts the embedding. This is achieved with

$$W^{\text{fst}}_Q = [\mathbf{0}_N, \mathbf{1}_N], \qquad W^{\text{fst}}_K = [\mathbf{0}_N, 1, \mathbf{1}_{N-1}], \qquad W^{\text{fst}}_V = [I_N, \mathbf{0}_{N\times N}], \tag{A.1}$$

and a sufficiently high inverse temperature parameter T.

The pos head instead extracts the one-hot encoding of the current position. This can be done with an attention head that always attends only to the current position and a value matrix W^{pos}_V that extracts the position embedding as a one-hot vector:

$$W^{\text{pos}}_Q = [\mathbf{0}_{N\times N}, I_N], \qquad W^{\text{pos}}_K = [\mathbf{0}_{N\times N}, I_N], \qquad W^{\text{pos}}_V = [\mathbf{0}_{N\times N}, I_N]. \tag{A.2}$$

When the outputs of these two attention heads are summed, then only the element of s_1 that corresponds to the current position will be larger than 1. From Equation (3.2) the output at the i-th position of the attention block is:

$$t_i = \sum_{j=1}^{p} A^{\text{fst}}_{ij} x_j + \sum_{j=1}^{p} A^{\text{pos}}_{ij} e_N(j) = s_1 + e_N(i),$$

where $x_1 = s_1$ and $x_j = E_{:,\mathtt{Y}_{j-1}}$ for $j > 1$.

We can extract the value of s_1 corresponding to the current position by substracting 1 from the hidden state and apply ReLU: $\hat{\mathcal{L}}_{\text{ex}} = \hat{\mathcal{L}}[I_N, -\mathbf{1}_N]$. Now, we are left with only one non-zero entry and that's the one corresponding to the next token. We can retain only the non-zero entry if we just sum all the entries of the hidden state with $\hat{\mathcal{L}}_{\text{sum}} = \hat{\mathcal{L}}[\mathbf{1}_N^\top, 0]$.

The final step is to map this scalar to a V-dimensional vector which has its maximum value at index $\mathtt{Y}_i$. This task is equivalent to designing V linear functions, each attaining its maximum at one of $0, {}^1/_V, \ldots, {}^{(V-1)}/_V$. To construct this, we use the property of convex functions that their tangent is always under the plot of the function. Therefore, given a convex function $\gamma(x)$, we construct the i-th linear function to be simply the tangent of γ at ${}^{i-1}/_V$. If we take $\gamma(x) = (x - {}^1/_2)^2$, this results in the following linear layer:

$$\mathcal{L}_{\text{proj}} = \mathcal{L}\left[\left[\frac{2(1-1)}{V} - 1, \ldots, \frac{2(V-1)}{V} - 1\right]^T, \left[\frac{1}{4} - \frac{(1-1)^2}{V^2}, \ldots, \frac{1}{4} - \frac{(V-1)^2}{V^2}\right]^\top\right]. \tag{A.3}$$

Figure A.1 shows the predictors for each individual token id.

With just two attention heads and three linear layers, the transformer

$$\mathcal{A}[(W^{\text{fst}}_Q, W^{\text{pos}}_Q), (W^{\text{fst}}_K, W^{\text{pos}}_K), (W^{\text{fst}}_V, W^{\text{pos}}_V)] \mathbin{⨾} \hat{\mathcal{L}}_{\text{ex}} \mathbin{⨾} \hat{\mathcal{L}}_{\text{sum}} \mathbin{⨾} \mathcal{L}_{\text{proj}} \mathbin{⨾} \text{softmax}$$

achieves the upper bound of V^N unique outputs by controlling a single virtual token at its input. Note that for this construction, the choice of embedding matrix $E \in \mathbb{R}^{N\times V}$ does not matter. The same transformer architecture can generate only V unique outputs if we only control the first token instead. Therefore, it is indeed the case that the embedding space has exponentially more capacity for control than the token space. You can see this transformer implemented and running in practice in Section 2 of this notebook.

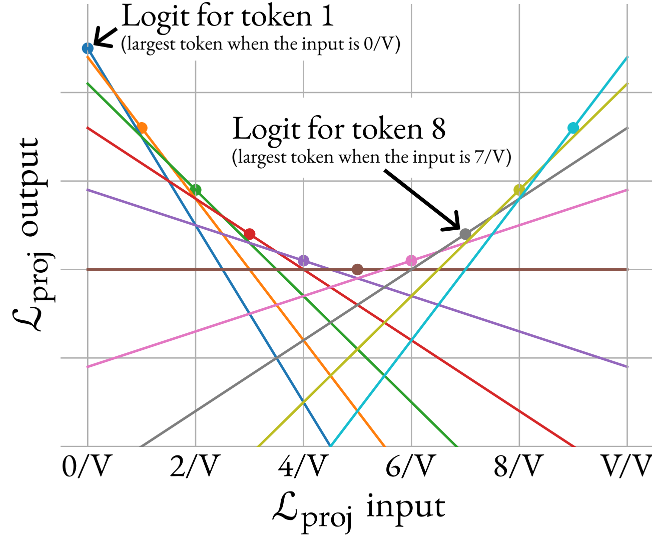


Figure A.1: Illustration of the predictors for each token in the $\mathcal{L}_{\text{proj}}$ linear layer for $V = 10$. The layer is constructed in such a way that the i-th token has the highest confidence when the input is $^{i-1}/_{V}$.

A.1.2 Conditional generation for a single virtual token ($n_X = n_Y = 1$)

This section provides an explicit construction of a transformer with the properties described in Theorem 3.2. The goal is to construct a transformer that, by varying the choice of the virtual token, can cause the model to act as any map $m : [1, \ldots, V] \to [1, \ldots, V]$. In other words, by selecting the virtual token, we can fully control how the model will respond to any token the user may provide.

First, we need to specify how the map m will be encoded in the virtual token $\boldsymbol{s}_1$. We choose the embedding size d_e to be V. Now, we can use the same encoding scheme as before, but now each element in $\boldsymbol{s}_1$ corresponds to a different user token, rather than to a position in the generated sequence:

$$\boldsymbol{s}_1 = \left(^{m(1)}/_{V}, \ldots, ^{m(V)}/_{V}\right).$$

Therefore, the first element of $\boldsymbol{s}_1$ designates the response if the user provides token 1, the second element is the response to the token 2, and so on.

Extracting the $\mathtt{Y}_i$-th value from $\boldsymbol{s}_1$ and decoding it can be done in a very similar way as for the unconditional case. The only difference is that instead of looking at the user input position, we look at its value. Take $\boldsymbol{E} = \boldsymbol{I}_V$ and $N = 2$.

Hence we have the following `val` head (only differing in the $\boldsymbol{W}_V$ matrix from Equation (A.2)):

$$\boldsymbol{W}_Q^{\text{val}} = [\boldsymbol{0}_{2\times V}, \boldsymbol{I}_2], \qquad \boldsymbol{W}_K^{\text{val}} = [\boldsymbol{0}_{2\times V}, \boldsymbol{I}_2], \qquad \boldsymbol{W}_V^{\text{val}} = [\boldsymbol{I}_V, \boldsymbol{0}_{V\times 2}].$$

We also need embedding of the first token, so we have a modified version of Equation (A.1):

$$W_Q^{\text{fst}} = [\mathbf{0}_V, 1, 1], \qquad W_K^{\text{fst}} = [\mathbf{0}_V, 1, 0], \qquad W_V^{\text{fst}} = [I_V, \mathbf{0}_{V\times 2}].$$

And hence the output of this attention block at the second position would be:

$$t_2 = \sum_{j=1}^{2} A_{ij}^{\text{fst}} x_j + \sum_{j=1}^{2} A_{ij}^{\text{val}} W_V^{\text{fst}} x_j = s_1 + e_V(\mathrm{Y}_1).$$

Similarly to the unconditional case, only the entry of t_2 corresponding to the user token will have a value above 1 and that value would be $1 + m(x_1)/V$.

We can now extract the one-hot representation of the target token using the same approach as before, just adjusting for the different hidden state size: $\hat{\mathcal{L}}_{\text{ex}} = \hat{\mathcal{L}}[I_V, -\mathbf{1}_V]$, $\hat{\mathcal{L}}_{\text{sum}} = \hat{\mathcal{L}}[\mathbf{1}_V^\top, 0]$, and the same projection had as before (Equation (A.3)). The final transformer is then: $\mathcal{A}[(W_Q^{\text{fst}}, W_Q^{\text{val}}), (W_K^{\text{fst}}, W_K^{\text{val}}), (W_V^{\text{fst}}, W_V^{\text{val}})] \fatsemi \hat{\mathcal{L}}_{\text{ex}} \fatsemi \hat{\mathcal{L}}_{\text{sum}} \fatsemi \mathcal{L}_{\text{proj}} \fatsemi \text{softmax}$. You can see this transformer implemented and running in practice in Section 3 here.

A.1.3 Conditional generation for longer responses ($n_X = 1, n_Y > 1$)

We can obtain longer responses via a simple extension. If the response length is N_0, then we can encode the map $m : [1, \dots, V] \to [1, \dots, V]^{N_0}$ in N_0 virtual tokens, each corresponding to one of the target positions:

$$s_i = \left({}^{m(1)_i}/_V, \dots, {}^{m(V)_i}/_V\right) \text{ for } i = 1, \dots, N_0.$$

For this model we would then have $N = 2N_0$ and $d_e = V$.

First, we need a head that always looks at the token provided by the user, which will be at position $N_o + 1$:

$$W_Q^{\text{user}} = [\mathbf{0}_V, \mathbf{1}_N], \qquad W_K^{\text{user}} = [\mathbf{0}_{(V+N_o)}, 1, \mathbf{0}_{(N_o-1)}], \qquad W_V^{\text{user}} = [I_V, \mathbf{0}_{V\times N}].$$

In order to consume the map at the right location, we need to also look at the embedding of the token N_o positions before the one we are trying to generate:

$$W_Q^{\text{back}} = \begin{bmatrix} \mathbf{0}_{N_0\times(N_0+V)} & I_{N_0} \\ \mathbf{0}_{N_0\times(N_0+V)} & \mathbf{0}_{N_0\times N_0} \end{bmatrix}, \qquad W_K^{\text{back}} = [\mathbf{0}_{N\times V}, I_N], \qquad W_V^{\text{back}} = [I_V, \mathbf{0}_{V\times N}].$$

From here on, the decoding is exactly the same as in the $n_X = n_Y = 1$ case. The final transformer is then: $\mathcal{A}[(W_Q^{\text{user}}, W_Q^{\text{back}}), (W_K^{\text{user}}, W_K^{\text{back}}), (W_V^{\text{user}}, W_V^{\text{back}})] \fatsemi \hat{\mathcal{L}}_{\text{ex}} \fatsemi \hat{\mathcal{L}}_{\text{sum}} \fatsemi \mathcal{L}_{\text{proj}} \fatsemi$ softmax. You can see this transformer implemented and running in practice in Section 4 here.

A.1.4 Conditional generation for longer user inputs ($n_X > 1, n_Y = 1$)

Finally, we consider the case when the user input $\mathtt{X}$ is longer. This is a bit more complicated because we need to search through a domain of size V^V. We will only consider the case with $n_X = 2$ where we would need two attention layers. A similar approach can be used to construct deeper models for $n_X > 2$. Finally, combining the strategy in the previous section for longer responses with the strategy in this section for longer user inputs allows us to construct transformers that map from arbitrary length user strings to arbitrary length responses.

In order to encode a map $m : [1, \dots, V]^2 \to [1, \dots, V]$ into a single virtual token we would need a more involved construction than before. Similarly to how we discretized each element of the virtual token $\boldsymbol{s}_1$ in V levels before, we are going to now discretize it into V^V levels. Each one of these levels would be one of the V^V possible maps from the *second* user token to the response. The first user token would be used to select the corresponding element of $\boldsymbol{s}_1$. Then this scalar will be "unpacked" into a new vector of V elements using the first attention block. Then, the second user token will select an element from this unpacked vector, which will correspond to the target token.

We construct the virtual token as follows:

$$\boldsymbol{s}_1 = \left[\sum_{i=1}^{V} m_1(i) \times \frac{V^{i-1}}{V^V}, \dots, \sum_{i=1}^{V} m_V(i) \times \frac{V^{i-1}}{V^V} \right],$$

where $m_f(x) = m(f, x)$ is a map from the second user token to the response when the first token is fixed to be f.

An additional change from the previous constructions is that we are going to divide the residual stream into two sections. This is in line with the theory that different parts of the residual stream specialize for different communications needs by different attention heads (Elhage et al., 2021). We will use the first half of the residual stream to extract and "unpack" the correct mapping from second token to target token, while the second half of the residual stream will be used to copy the second token value so that the second attention layer can use it to extract the target. As usual, the embedding matrix will be the identity matrix: $\boldsymbol{E} = \boldsymbol{I}_V$. Finally, for convenience, we will also use a dummy zero virtual token that we will attend to when we want to not attend to anything. This results in context size $N = 4$ with the input being

$$\left(\begin{bmatrix} 0_V \\ \boldsymbol{e}_N(1) \end{bmatrix}, \begin{bmatrix} \boldsymbol{s}_1 \\ \boldsymbol{e}_N(2) \end{bmatrix}, \begin{bmatrix} \boldsymbol{E}_{:,\mathtt{X}_1} \\ \boldsymbol{e}_N(3) \end{bmatrix}, \begin{bmatrix} \boldsymbol{E}_{:,\mathtt{X}_2} \\ \boldsymbol{e}_N(4) \end{bmatrix} \right) = \left(\begin{bmatrix} \boldsymbol{0}_V \\ \boldsymbol{e}_N(1) \end{bmatrix}, \begin{bmatrix} \boldsymbol{s}_1 \\ \boldsymbol{e}_N(2) \end{bmatrix}, \begin{bmatrix} \boldsymbol{e}_V(\mathtt{X}_1) \\ \boldsymbol{e}_N(3) \end{bmatrix}, \begin{bmatrix} \boldsymbol{e}_V(\mathtt{X}_2) \\ \boldsymbol{e}_N(4) \end{bmatrix} \right).$$

We want the output at the last position to be the target $m(\mathtt{X}_1, \mathtt{X}_2)$, that is:

$$\arg\max_{u \in 1, \dots, V} y_{4,u} = m(\mathtt{X}_1, \mathtt{X}_2) \text{ for any } m, \mathtt{X}_1, \mathtt{X}_2.$$

The first attention block will have three attention heads.

As before, we want to extract the value of s_1 that corresponds to the first token the user provided ($\mathtt{X}_1$) and place it in the first half of the residual stream. We want only the third position to do that, while the rest of the positions keep the first half of their residual stream with zeros. Hence we have the following `fst` head:

$$\boldsymbol{W}_Q^{\text{fst}} = \left[\begin{array}{c|cccc} \boldsymbol{0}_{2\times V} & 1 & 1 & 0 & 1 \\ & 0 & 0 & 1 & 0 \end{array}\right], \quad \boldsymbol{W}_K^{\text{fst}} = \left[\begin{array}{c|cccc} \boldsymbol{0}_{2\times V} & 1 & 0 & 0 & 0 \\ & 0 & 1 & 0 & 0 \end{array}\right], \quad \boldsymbol{W}_V^{\text{fst}} = \begin{bmatrix} \boldsymbol{I}_V & \boldsymbol{0}_{V\times N} \\ \boldsymbol{0}_{V\times V} & \boldsymbol{0}_{V\times N} \end{bmatrix}.$$

The `user1` head extracts the value of the first user-provided token ($\mathtt{X}_1$) and also places it in the first half of the residual stream:

$$\boldsymbol{W}_Q^{\text{user1}} = \left[\begin{array}{c|cccc} \boldsymbol{0}_{2\times V} & 1 & 1 & 0 & 1 \\ & 0 & 0 & 1 & 0 \end{array}\right], \boldsymbol{W}_K^{\text{user1}} = \left[\begin{array}{c|cccc} \boldsymbol{0}_{2\times V} & 1 & 0 & 0 & 0 \\ & 0 & 0 & 1 & 0 \end{array}\right], \boldsymbol{W}_V^{\text{user1}} = \begin{bmatrix} \boldsymbol{I}_V & \boldsymbol{0}_{V\times N} \\ \boldsymbol{0}_{V\times V} & \boldsymbol{0}_{V\times N} \end{bmatrix}.$$

And the `user2` head does the same for the value of the second user-provided token ($\mathtt{X}_2$), placing it in the second half of the residual stream:

$$\boldsymbol{W}_Q^{\text{user2}} = \left[\begin{array}{c|cccc} \boldsymbol{0}_{2\times V} & 1 & 1 & 1 & 0 \\ & 0 & 0 & 0 & 1 \end{array}\right], \boldsymbol{W}_K^{\text{user2}} = \left[\begin{array}{c|cccc} \boldsymbol{0}_{2\times V} & 1 & 0 & 0 & 0 \\ & 0 & 0 & 0 & 1 \end{array}\right], \boldsymbol{W}_V^{\text{user2}} = \begin{bmatrix} \boldsymbol{0}_{V\times V} & \boldsymbol{0}_{V\times N} \\ 2\boldsymbol{I}_V & \boldsymbol{0}_{V\times N} \end{bmatrix},$$

where the factor 2 is there because, as usual, the first linear layer will subtract 1 from everything in order to extract the value selected by the first token.

This linear layer looks as usual: $\hat{\mathcal{L}}_{\text{ex2}} = \hat{\mathcal{L}}[\boldsymbol{I}_{2V}, -\boldsymbol{1}_{2V}]$. The result is that the first V elements will be 0 except one which designates which map from second user token to output we should use, and the second V elements have a one hot-encoding of the second user token. Constructing an MLP that unpacks the mapping can become quite involved so we do not provide an explicit form for it. But from the universal approximation theorems and the finiteness of the domain and range, we know that such an MLP should exist. We thus designate by `unpack` the MLP that decodes the first half of the residual stream to:

$$\left(\frac{m_{\mathtt{X}_1}(1)}{V}, \ldots, \frac{m_{\mathtt{X}_1}(V)}{V}\right)$$

and keeps the second half unchanged.

And now, by using two attention heads, the second attention block extracts the value of the above vector at the position designated by the second token, in a fashion not dissimilar to all the previous cases:

$$\boldsymbol{W}_Q^{\text{emb}} = [\boldsymbol{0}_V^\top, \boldsymbol{1}_V^\top], \qquad \boldsymbol{W}_K^{\text{emb}} = [\boldsymbol{1}_V^\top, \boldsymbol{0}_V^\top], \qquad \boldsymbol{W}_V^{\text{emb}} = \begin{bmatrix} \boldsymbol{I}_V & \boldsymbol{0}_{V\times V} \end{bmatrix},$$

$$\boldsymbol{W}_Q^{\text{user2}'} = [\boldsymbol{0}_V^\top, \boldsymbol{1}_V^\top], \qquad \boldsymbol{W}_K^{\text{user2}'} = [\boldsymbol{0}_V^\top, \boldsymbol{1}_V^\top], \qquad \boldsymbol{W}_V^{\text{user2}'} = \begin{bmatrix} \boldsymbol{0}_{V\times V} & \boldsymbol{I}_V \end{bmatrix},$$

And finally, with $\hat{\mathcal{L}}_{\text{ex}} = \hat{\mathcal{L}}[\boldsymbol{I}_V, -\boldsymbol{1}_V]$, $\hat{\mathcal{L}}_{\text{sum}} = \hat{\mathcal{L}}[\boldsymbol{1}_V^\top, 0]$, and the same projection had as before (Equation (A.3)), we get the target token. The final transformer is then:

$$\mathcal{A}[(\boldsymbol{W}_Q^{\text{fst}}, \boldsymbol{W}_Q^{\text{user1}}, \boldsymbol{W}_Q^{\text{user2}}), (\boldsymbol{W}_K^{\text{fst}}, \boldsymbol{W}_K^{\text{user1}}, \boldsymbol{W}_K^{\text{user2}}), (\boldsymbol{W}_V^{\text{fst}}, \boldsymbol{W}_V^{\text{user1}}, \boldsymbol{W}_V^{\text{user2}})] \mathbin{⨟} \hat{\mathcal{L}}_{\text{ex2}} \mathbin{⨟} \texttt{unpack}$$
$$\mathbin{⨟} \mathcal{A}[(\boldsymbol{W}_Q^{\text{emb}}, \boldsymbol{W}_Q^{\text{user2}'}), (\boldsymbol{W}_K^{\text{emb}}, \boldsymbol{W}_K^{\text{user2}'}), (\boldsymbol{W}_V^{\text{emb}}, \boldsymbol{W}_V^{\text{user2}'})] \mathbin{⨟} \hat{\mathcal{L}}_{\text{ex}} \mathbin{⨟} \hat{\mathcal{L}}_{\text{sum}} \mathbin{⨟} \mathcal{L}_{\text{proj}} \mathbin{⨟} \text{softmax}.$$

You can see this transformer implemented and running in practice in Section 5 here.

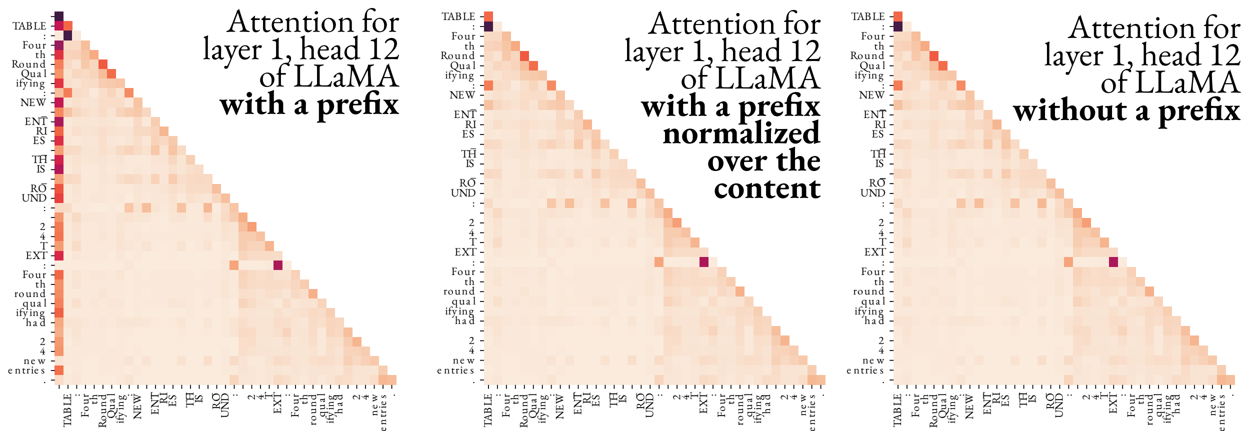


Figure A.2: The attention of the twelfth head of the first layer of LLaMA (Touvron et al., 2023). The left plot shows the attention with a prefix of length one. The second plot shows the same attention but normalized such that the attenion over the non-prefix positions sums to 1. The right plot shows the attention of the pre-trained model (without prefix). The center and the right plots are the same, illustrating that the presence of the prefix indeed only scales down the attention over the content (non-prefix positions) but does not change its relative distribution, providing empirical validation of Equation (3.6). The test sequence is `TABLE: Fourth Round Qualifying :  NEW_ENTRIES_THIS_ROUND : 24 TEXT: Fourth round qualifying had 24 new entries.` from the DART table-to-test dataset (Nan et al., 2021).

A.2 Attention distribution over the prefix

As discussed in Section 3.4, longer prefixes define a subspace from which the bias for the attention block is selected. For a prefix of size n_S, that means that this subspace is n_S-dimensional. Each of prefix position j defines a basis vector $\boldsymbol{W}_V \boldsymbol{s}_j$ for this subspace, while the attention $\boldsymbol{A}^{\text{pt}}_{i,S_j}$ on this position determines how much of this basis component contributes to the bias.

In ordered to span the whole subspace and make full use of the capacity of the prefix, $\boldsymbol{A}^{\text{pt}}_{i,S_j}$ should vary between 0 and 1 for different inputs. However, we observe that this does not happen in practice. Figure A.4 shows the ranges of attention the different prefix positions take for the GPT-2 model (Radford et al., 2019). For layer 1, for example, the attention each prefix positions gets is almost constant hence, the effective subspace is collapsed and there is a single bias vector that's applied to the attention layer output, regardless of the user input X.

Some other layers show slightly higher variation. For example, layer 3 has three prefix positions with large variations. Therefore, the effective bias subspace is 3-dimensional and the user input X governs which bias vector from this subspace will be selected.

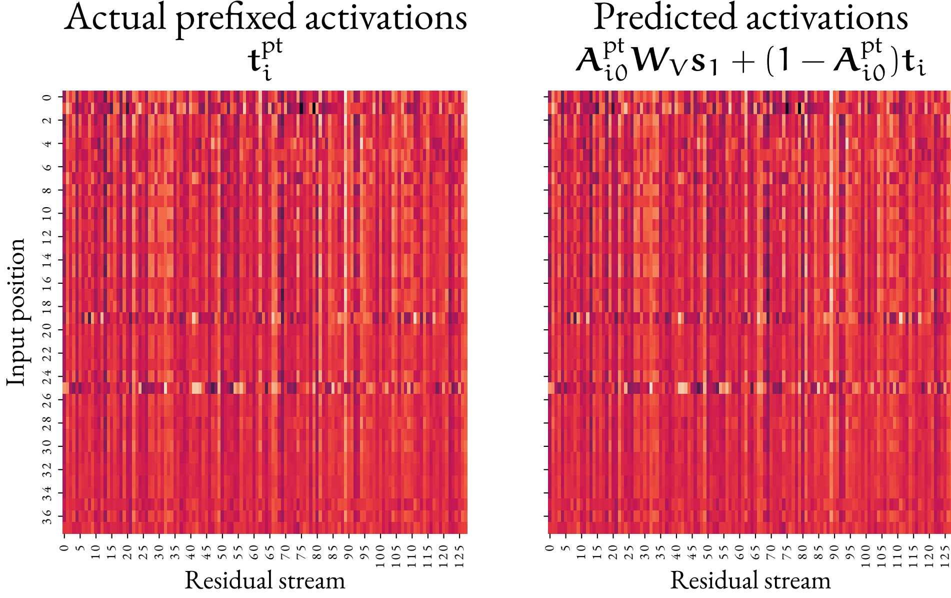


Figure A.3: The activations of the twelfth head of the first layer of LLaMA (Touvron et al., 2023). The left plot shows the activations in the presence of the prefix. The right plot shows the activations t_i of the pretrained model, scaled by one minus the attention that the prefix would take and then biased in the direction $W_V s_1$. The two plots are the same, illustrating that our theory, Equation (3.7) in particular, also holds for real-world large transformer models. The test sequence is the same as in Figure A.2.

A.3 Expressivity of prefix-tuning across deeper models

In Section 3.6, we considered the effect of the presence of the prefix in the first attention layer on the attention of the second. However, the effects become more complex as one adds more attention attention layers. We also ignored the MLPs between, but in practice they can play an important role. Here, instead, we analyse prefix-tuning as learning a neural network. We argue that while the resulting architecture includes both linear operation and non-linear activations, the structure is unlikely to learn efficiently.

For simplicity, we will consider two inputs x_1 and x_2 and a single prefix s. The output of the attention head, parameterized by s is then:

$$\begin{aligned}
\mathcal{A}_s(x_1, x_2) &= \langle y_1, y_2 \rangle \\
y_1 &= \frac{\exp(x_1{}^\top H s) W_V s + \exp(x_1{}^\top H x_1) W_V x_1 + \exp(x_1{}^\top H x_2) W_V x_2}{C_1} \\
y_2 &= \frac{\exp(x_2{}^\top H s) W_V s + \exp(x_2{}^\top H x_1) W_V x_1 + \exp(x_2{}^\top H x_2) W_V x_2,}{C_2}
\end{aligned} \tag{A.4}$$

where we have omitted the $T/\sqrt{k}$ factors and have folded the softmax normalization

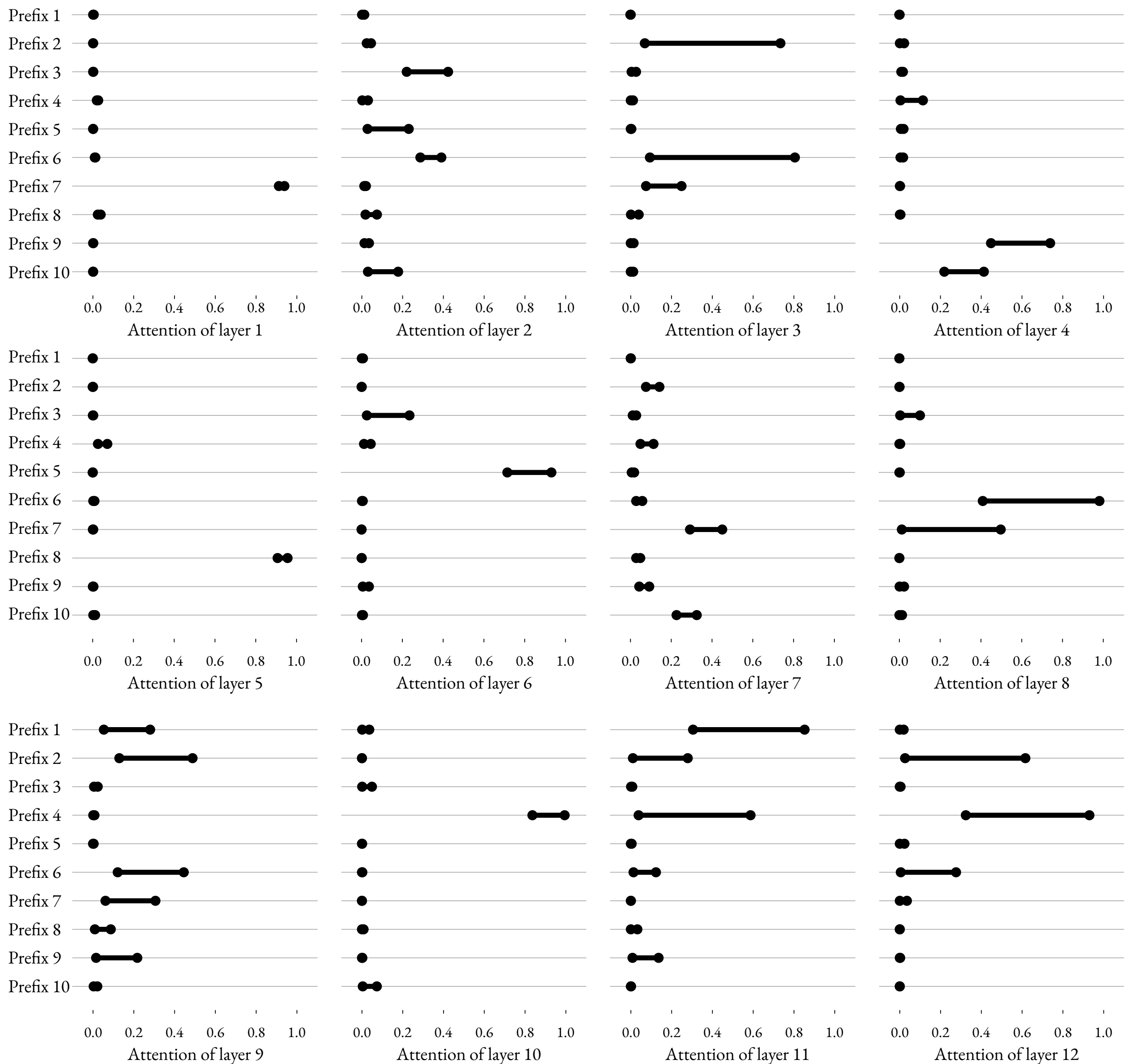


Figure A.4: The range of attention (1st to 99th percentile) for a single GPT-2 (Radford et al., 2019) prefix trained on the Emotion dataset (Saravia et al., 2018). The prefix is of size 10 ($n_S = 10$). This is the attention of the last user input token (n_X) because this is the position at which the class prediction is done. For illustration purposes, we have normalized the attention so that the attention over the 10 prefix positions sums to 1. The range of attention over the 10 positions for each layer are shown.

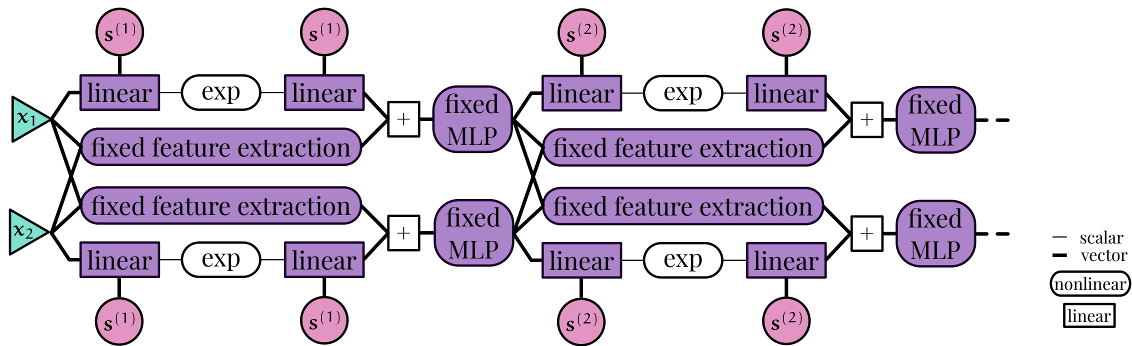


Figure A.5: Prefix-tuning as a neural network architecture. While linearities and non-linearities are present, the only learnable parameters $\boldsymbol{s}^{(1)}, \boldsymbol{s}^{(2)}, \ldots$ have limited interaction with the inputs $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$. The interaction of the prefix parameters with each input is only via the scalar attention, shown here with a light connection. The mixing of information between the inputs happens via residual connections with the pretrained fixed feature extraction and hence is not learnable. The MLP is also fixed and hence only acts as a multivariate activation function. This limited interaction explains why prefix-tuning struggles to learn new tasks even in deeper models.

into C_1 and C_2. The layer inputs, pretrained parameters and the learnable parameters are correspondingly highlighted. The attention head is clearly a non-linear operation. However, the learnable parameter $\boldsymbol{s}$ participates only in the left term. It interacts with only one of the inputs at a time and only by computing a single scalar value $\boldsymbol{x}_i^\top \boldsymbol{H}\boldsymbol{s}$. As we discussed above, the interaction between $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ is non-trainable and can be thought as a hard-coded feature extraction. Each of the outputs is then passed to a pretrained MLP which can be thought of as an activation function. This would be a *multivariate* activation function, which while unusual in the contemporary practice has been studied before (Solazzi and Uncini, 2004). Figure A.5 illustrates the computation graph of the resulting neural network and shows that the only learnable interaction between the inputs is indirect. Therefore, prefix-tuning can be considered as learning a neural network where the only interaction between the inputs happens via non-learnable residual connections. Nevertheless, the alternating linear and nonlinear operations are reminiscent of the standard neural network architecture and their universal approximation properties (Hassoun, 1995). That begs the question if the prefix-tuning architecture can be a universal approximator and whether it would be a parameter-efficient one.

An example of prefix-tuning failing to be a universal approximator. While we leave the formal analysis of the representational capacity of prefix-tuning as future work, we provide an example of pretrained parameters for which the architecture is *not* a universal approximator. As can be seen in Figure A.5, all information must pass through the non-learnable MLPs. Thus, if the MLPs destroy all input information, there is no value for the prefixes $\boldsymbol{s}^{(1)}, \boldsymbol{s}^{(2)}, \ldots$ that can change that fact. The MLPs can destroy all information,

if, e.g., one of their linear layers has a zero weight matrix. While this is unlikely to be learned with typical pretraining, this demonstrates that if prefix-tuning could be a universal approximator, that would pose specific requirements on the pretrained MLPs and it is not clear whether real-world pretraining would satisfy these requirements.

A.4 Extended results

In this appendix we present further experiments in the context of Section 3.5. We consider different prefix lengths ($n_S \in \{10, 50, 100\}$) and different model sizes (4, 16 and 32 layers). We consider two extensions, the first one maintains the pretraining step as in Section 3.5, the other one extends the set of pretraining tasks with 4 additional tasks. Both pretraining and prefix-tuning are done for 100 000 iterations with the prefix-tuning accuracy reported every 10 000 iterations.

A.4.1 Pretraining as in Section 3.5

The first setting has the same pretraining tasks as in Section 3.5, namely sorting in ascending (↗) or descending (↘) order, and adding one (+1) or two (+2) to each element of the input sequence. We evaluate by prefix-tuning on the same four tasks plus incrementing the ascending sorted sequence (↗+1), double histogram ($\mathbb{H}$), element-wise modulo operation (with respect to the first element of the sequence), and FilterAtLeastNTimes which puts zeros at the positions of elements whose value appears at less than N times in the sequence with N being the first element of the sequence:

Pretraining tasks:	Sort in ascending order (↗) Sort in descending order (↘) Add 1(+1) Add 2 (+2)
Prefix-tuning tasks:	Sort in ascending order (↗) Sort in descending order (↘) Add 1(+1) Add 2 (+2) Sort ascending and add 1 (↗+1) Modulo the first element (Modulo) Double Histogram ($\mathbb{H}$) Filter the elements that are at least as large as the first element (FilterAtLeastNTimes)

The results are plotted in Figure A.6. As expected, the prefix-tuned accuracy on the pretraining tasks is close to 100%. Interestingly, prefix length 50 for the largest (32-layer) architecture appears to be an exception and does not learn the +1, ↗ and ↘.

As also observed in Section 3.5, regardless of the prefix length and the model size, prefix-tuning a model pretrained with these four tasks cannot learn the double histogram

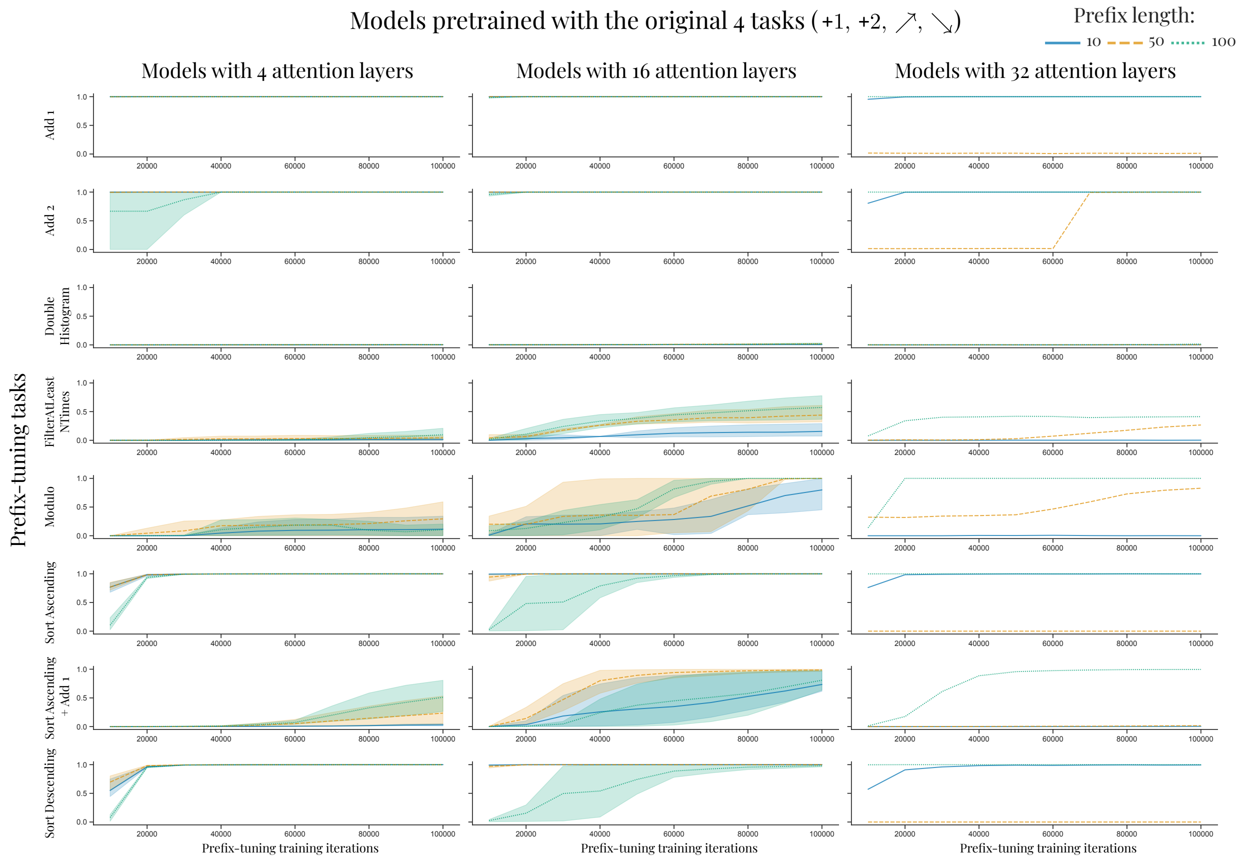


Figure A.6: Extended result with pretraining as in Section 3.5. The pretraining and the prefix-tuning tasks are described in Appendix A.4.1. Each prefix was trained for 100 000 iterations and we report accuracy at every 10 000 iterations. Each experiment is performed with 3 random seeds, except the 32 layer case which, due to the computational costs involved, was performed only with one seed.

task (ℍ). FilterAtLeastNTimes also appears to be challenging but the 16-layer and 32-layer experiments reach about 50% accuracy for the longest prefix size $n_S = 100$. This is curious as FilterAtLeastNTimes is related to the double histogram task: it can be considered as thresholding the double histogram output. FilterAtLeastNTimes, similarly to double histogram, is therefore not compositional in the pretraining task. It is surprising then that FilterAtLeastNTimes would achieve higher accuracy than double histogram. This hints that, perhaps, compositionality does not fully explain why prefix-tuning works for some and not other downstream tasks.

The results in Figure A.6 also hint that bigger is not always better. For example, the 4-layer model prefix-tuned for the Modulo task performs better with a prefix size 50 than the larger prefix size 100. A similar effect can be observed with the 16-layer model prefix-tuned for the ↗+1 task. Larger models are also not necessarily more conducive to successful

prefix-tuning: for many of the cases the 32-layer models perform worse when prefix-tuned than the 16-layer models.

A.4.2 Extended pretraining

The second setting extends the set of pretraining tasks. On top of the original three pretraining tasks $\nearrow$, $\searrow$,+1 (without +2) we also pretrain on element-wise modulo operation (with respect to the first element of the sequence), element-wise less than (less than the first element in the sequence), element-wise divisible (by the first element of the sequence), and inverse binary (element-wise negation). For the inverse binary task, the input is restricted to be binary. We evaluate by prefix-tuning on the same seven tasks, as well as 11 additional ones as listed in the following table:

Pretraining tasks:	Sort in ascending order ($\nearrow$)
	Sort in descending order ($\searrow$)
	Add 1 (+1)
	Modulo the first element (Modulo)
	Filter the elements that are less than the first element (LessThan)
	Filter the elements that are divisible by first element (Divisible)
	Element-wise negation (InverseBinary)
Prefix-tuning tasks:	Sort in ascending order ($\nearrow$)
	Sort in descending order ($\searrow$)
	Add 1 (+1)
	Add 2 (+2)
	Add 3 (+3)
	Modulo the first element (Modulo)
	Filter the elements that are less than the first element (LessThan)
	Filter the elements that are not less than the first element (MoreThanEqual)
	Filter the elements that are divisible by first element (Divisible)
	Filter the elements that are not divisible by first element (NotDivisible)
	Element-wise negation (InverseBinary)
	Double Histogram ($\mathbb{H}$)
	Filter the elements that are at least as large as the first element (FilterAtLeastNTimes)
	Sort ascending, followed by add 1 ($\nearrow$+1)
	Add 1, followed by LessThan (+1 + LessThan)
	LessThan, followed by Add 1 (LessThan +1)
	LessThan, followed by sort ascending (LessThan + $\nearrow$)
	Divisible, followed by Add 1 (Divisible +1)

The results are shown in Figure A.7. Even though the model did not see the +2 and +3 tasks, it appears that prefix-tuning can generalize from the +1 task. The +1 + LessThan task is another example of successful prefix-tuning for a task that is compositional in pretraining tasks. Similarly, for Divisible +1, LessThan +1, LessThan + $\nearrow$, MoreThanEqual, NotDivisible,

Similarly to the case in Appendix A.4.1, we observe several instances in which the largest, 32-layer, model performs worse, when prefix-tuned, than the smaller models, as well as cases where shorter prefixes perform better than longer ones.

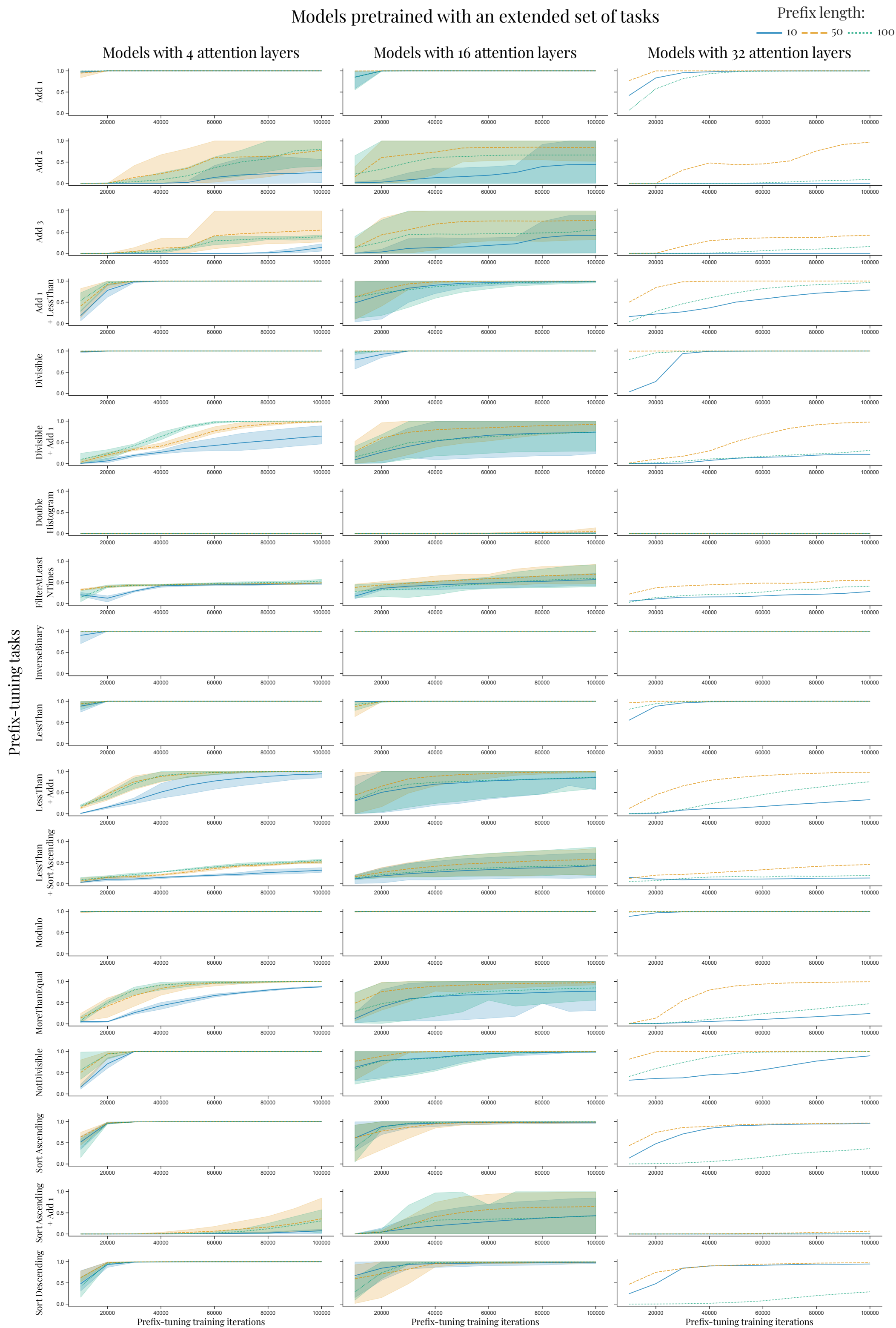


Figure A.7: Extended result with pretraining with additional tasks. The pretraining and the prefix-tuning tasks are described in Appendix A.4.2. Each prefix was trained for 100 000 iterations and we report accuracy at every 10 000 iterations. Each experiment is performed with 3 random seeds, except the 32 layer case which, due to the computational costs involved, was performed only with one seed.

A.5 Further experiments with memorization

Our experiments in Section 3.5 focused on algorithmic tasks such as sorting, incrementing and counting. However, learning a natural language also includes a substantial memorization component. For example, learning a new language requires learning its vocabulary. Hence, if a novel task requires memorization of novel concepts, the fine-tuning method should be able to memorize them. To this end, we evaluate and compare the abilities of prefix-tuning and LoRA to learn to memorize a large number of new words and show that, for the same amount of learnable parameters, LoRA can learn to translate to a new language while prefix-tuning cannot. These findings further strengthen our results from Section 3.5.

We use the PHOR-in-One dataset (Costa et al., 2022) that contains 4921 unique translations of English words into German, Spanish and Portuguese. The dataset is preprocessed to remove all accents in order to ensure that fine-tuning does not require characters that the model has not seen during pre-training (e.g., *éçü* would become *ecu*). Character-level tokenization is used, with the additional `<TR>` token that separates the source and target words and a `<PAD>` token that we use to ensure all training sequences are of the same length.

We then pre-train a 4-layer 4-head transformer to translate the English words to German. As seen in Table A.1, this model successfully memorizes 99.3% of the English-German pairs. We then want to fine-tune this model to instead translate the English words to Spanish. Spanish is linguistically closer to English than German, so, in a way, the fine-tuning task is simpler than the pre-training task.

We do prefix-tuning on English-Spanish pairs with a prefix of size $n_S = 66$ but it achieves only 0.18% accuracy (about 10 words which are the same in English and Spanish, e.g., *instrumental*, *orbital*, *solar*). However, rank-4 LoRA is able to achieve 94.8% accuracy with half the training iterations. Both fine-tuning methods have similar numbers of learnable parameters (see Table A.1). Therefore, this is further evidence that prefix-tuning fails to learn a new task, while LoRA with the same number of learnable parameters can.

Note that there is no train-test split for the word pairs. We are evaluating the accuracy on the training set as this experiment is measuring *memorization* rather than *generalization*.

Table A.1: Further experiments on memorization of word pairs form (Costa et al., 2022) in different languages. The model is pre-trained on the English-to-German word pairs to almost perfect accuracy. Prefix-tuning fails to modify it to memorize English-Spanish word pairs while LoRA with the same number of parameters and half the training iterations reaches 94% percent accuracy. 5 random seeds.

	Pre-training on English to German	Prefix-tune (n_S=66) on English to Spanish	Rank-4 LoRA on English to Spanish
Accuracy	99.3 ± 0.1%	0.18 ± 0.01%	94.8 ± 1.1%
Learnable parameters	3.19M	67 584	66 780
Training iterations	50 000	100 000	50 000

Appendix B

Appendices for Prompting a Pretrained Transformer Can Be a Universal Approximator

B.1 Background on analysis on the sphere

As mentioned in the main text, the investigation of the properties of attention heads naturally leads to analysing functions over the hypersphere. To this end, our results require some basic facts about the analysis on the hypersphere. We will review them in this appendix. For a comprehensive reference, we recommend (Atkinson and Han, 2012) and (Dai and Xu, 2013).

Define $\mathbb{P}_k(\mathbb{R}^{m+1})$ to be the space of polynomials of degree at most k. The restriction of a polynomial $p \in \mathbb{P}_k(\mathbb{R}^{m+1})$ to the unit hypersphere $S^m = \{x \in \mathbb{R}^{m+1} \mid \|x\|_2 = 1\}$ is called a *spherical polynomial*. We can thus define the space of spherical polynomials:

$$\mathbb{P}_k(S^m) = \{p|_{S^m} \text{ for } p \in \mathbb{P}_k(\mathbb{R}^{m+1})\}.$$

Define by $\mathbb{H}_k(\mathbb{R}^{m+1})$ the space of polynomials of degree k that are homogeneous:

$$\mathbb{H}_k(\mathbb{R}^{m+1}) = \operatorname{span}\left\{(x_1, \dots, x_{m+1}) \mapsto x_1^{\alpha_1} \times \cdots \times x_{m+1}^{\alpha_{m+1}} \mid \sum_{i=1}^{m+1} \alpha_i = k\right\}.$$

Its restriction to the sphere $\mathbb{H}_k(S^m)$ is defined analogously to $\mathbb{P}_k(S^m)$. Finally, we can define the space $\mathbb{Y}(\mathbb{R}^{m+1})$ of harmonic homogeneous polynomials:

$$\mathbb{Y}_k(\mathbb{R}^{m+1}) = \left\{p \in \mathbb{H}_k(\mathbb{R}^{m+1}) \mid \frac{\partial^2}{\partial \boldsymbol{x}^2} p(\boldsymbol{x}) = 0,\ \forall \boldsymbol{x} \in \mathbb{R}^{m+1}\right\}.$$

$\mathbb{Y}_k(S^m)$ which is the restriction of $\mathbb{Y}_k(\mathbb{R}^{m+1})$ to S^m is the set of *spherical harmonics* of degree k. Spherical harmonics are a higher-dimensional extension of Fourier series.

Notably, even though

$$\mathbb{Y}_k(S^m) \subset \mathbb{H}_k(S^m) \subset \mathbb{P}_k(S^m),$$

the restriction of any polynomial on S^m is a sum of spherical harmonics:

$$\mathbb{P}_k(S^m) = \mathbb{Y}_0(S^m) \oplus \cdots \oplus \mathbb{Y}_k(S^m),$$

with $\oplus$ being the direct sum (Atkinson and Han, 2012, Corollary 2.19).

We define $C(S^m)$ to be the space of all continuous functions defined on S^m with the uniform norm

$$\|f\|_\infty = \sup_{x \in S^m} |f(x)|, \ f \in C(S^m). \tag{B.1}$$

Similarly, $\mathcal{L}_p(S^m), 1 \leq p < \infty$ is the space of all functions defined on S^m which are integrable with respect to the standard surface measure dw_m. The norm in this space is:

$$\|f\|_p = \left(\frac{1}{w_m} \int_{S^m} |f(x)|^p \, dw_m(x) \right)^{1/p}, \ f \in \mathcal{L}_p(S^m), \tag{B.2}$$

with the surface area being

$$w_m = \int_{S^m} dw_m = \frac{2\pi^{(m+1)/2}}{\Gamma\big((m+1)/2\big)}. \tag{B.3}$$

We will use V_m to denote any of these two spaces and $\|\cdot\|_m$ the corresponding norm.

A key property of spherical harmonics is that sums of spherical harmonics can uniformly approximate the functions in $C(S^m)$. In other words, the span of $\bigcup_{k=0}^{\infty} \mathbb{Y}_k(S^m)$ is dense in $C(S^m)$ with respect to the uniform norm $\|\cdot\|_\infty$. Hence, any $f \in C(S^m)$ can be expressed as a series of spherical harmonics:

$$f(x) = \sum_{k=0}^{\infty} Y_k^m(x), \ \text{with } Y_k^m \in \mathbb{Y}_k(S^m), \ \forall k.$$

We will also make a heavy use of the concept of spherical convolutions. Define the space of kernels $\mathcal{L}^{1,m}$ to consist of all measurable functions K on $[-1, 1]$ with norm

$$\|K\|_{1,m} = \frac{w_{m-1}}{w_m} \int_{-1}^{1} |K(t)|(1-t^2)^{(m-2)/2} \, dt \ < \infty.$$

Definition B.1.1 (Spherical convolution)**.** The spherical convolution $K * f$ of a kernel K in $\mathcal{L}^{1,m}$ with a function $f \in V_m$ is defined by:

$$(K * f)(x) = \frac{1}{w_m} \int_{S^m} K(\langle x, y \rangle) f(y) \, dw_m(y), \ x \in S^m.$$

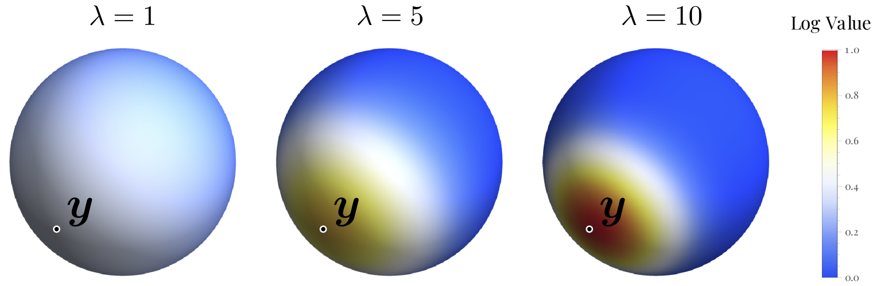


Figure B.1: Plots of the von Mises-Fisher kernel $K_\lambda^{\mathrm{vMF}}(\langle \boldsymbol{x}, \boldsymbol{y} \rangle)$ for $\lambda = 1, 5, 10$ and fixed $\boldsymbol{y}$ in three dimensions ($m = 2$). The larger λ is, the more concentrated the kernel is around $\boldsymbol{y}$.

Spherical convolutions map functions $f \in V_m$ to functions in V_m. Furthermore, the spherical harmonics are eigenfunctions of the function generated by a kernel in $\mathcal{L}^{1,m}$:

Lemma B.1.2 (Funk and Hecke's formula (Funk, 1915; Hecke, 1917; Estrada, 2014)).

$$K * Y_k^m = a_k^m(K) Y_k^m, \text{ when } K \in \mathcal{L}^{1,m},\ Y_k^m \in \mathbb{Y}_k(S^m), k = 0, 1, \ldots,$$

where $a_k^m(K)$ are the coefficients in the series expansion in terms of Gegenbauer polynomials associated with the kernel K:

$$a_k^m(K) = \frac{w_{m-1}}{w_m} \int_{-1}^{1} K(t) \frac{Q_k^{(m-1)/2}(t)}{Q_k^{(m-1)/2}(1)} (1 - t^2)^{(m-2)/2}\, dt, k = 0, 1, \ldots. \tag{B.4}$$

Here, $Q_k^{(m-1)/2}$ is the Gegenbauer polynomial of degree k.

Note also that with a change of variables we have:

$$\int_{S^m} K(\langle \boldsymbol{x}, \boldsymbol{y} \rangle)\, dw_m(\boldsymbol{y}) = w_{m-1} \int_{-1}^{1} K(t)(1 - t^2)^{(m-2)/2}\, dt. \tag{B.5}$$

Ideally, we would like a kernel that acts as an identity for the convolution operation. In this case, we would have $\|K * f - f\|_\infty = 0, f \in \mathcal{C}(S^m)$ which would be rather convenient. However, there is no such kernel in the spherical setting (Menegatto, 1997). The next best thing is to construct a sequence of kernels $\{K_n\} \in \mathcal{L}^{1,m}$ such that $\|K_n * f - f\|_m \to 0$ as $n \to \infty$ for all $f \in V_m$. This sequence of kernels is called an *approximate identity*. The specific such sequence of kernels we will use is based on the von Mises-Fisher distribution as this gives us the $\exp(\langle \boldsymbol{x}, \boldsymbol{y} \rangle)$ form that we also observe in the transformer attention mechanism.

Definition B.1.3 (von Mises-Fisher kernels, (Ng and Kwong, 2022)). We define the sequence of von Mises-Fisher kernels as:

$$K_\lambda^{\mathrm{vMF}}(t) = c_{m+1}(\lambda) \exp(\lambda t),\ t \in [-1, 1],$$

where

$$c_{m+1}(\lambda) = \frac{w_m \lambda^{\frac{m+1}{2}-1}}{(2\pi)^{\frac{m+1}{2}} I_{\frac{m+1}{2}-1}(\lambda)},$$

with I_v being the modified Bessel function at order v.

Note that a von Mises-Fisher kernel can also be expressed in terms of points on S^m. In particular, for a fixed $\boldsymbol{y} \in S^m$ we have $K_\lambda^{\text{vMF}}(\langle \boldsymbol{x}, \boldsymbol{y} \rangle), \boldsymbol{x} \in S^m$. The parameter λ is a "peakiness" parameter: the large λ is, the closer $K_\lambda^{\text{vMF}}(\langle \boldsymbol{x}, \boldsymbol{y} \rangle)$ approximates the delta function centered at $\boldsymbol{y}$, as can be seen in Figure B.1. It is easy to check that $\|K_\lambda^{\text{vMF}}\|_{1,m} = 1, \ \forall \lambda > 1, m > 1$ and hence the sequence is in $\mathcal{L}^{1,m}$, meaning they are valid kernels. Ng and Kwong (2022, Lemma 4.2) show that $\{K_\lambda^{\text{vMF}}\}$ is indeed an approximate identity, i.e., $\|K_\lambda^{\text{vMF}} * f - f\|_m \to 0$ as $\lambda \to \infty$ for all $f \in V_m$.[1] As we want a Jackson-type result however, we will need to upper bound the error $\|K_\lambda^{\text{vMF}} * f - f\|_m$ as a function of λ, that is a non-asymptotic result on the quality of the approximation by spherical convolutions with K_λ^{vMF}. We do that in Lemma B.2.5.

B.2 A Jackson-type bound for universal approximation on the unit hypersphere

The overarching goal in this section is to provide a Jackson-type (Definition 4.3) bound for approximating functions $f : S^m \to \mathbb{R}^{m+1}$ on the hypersphere $S^m = \{\boldsymbol{x} \in \mathbb{R}^{m+1} \mid \|\boldsymbol{x}\|_2 = 1\}$ by functions of the form

$$h(\boldsymbol{x}) = \sum_{k=1}^{N} \xi_k \exp(\lambda \langle \boldsymbol{x}, \boldsymbol{b}_k \rangle) \tag{B.6}$$

To this end, we will leverage results from approximation on the hypersphere using spherical convolutions by Menegatto (1997) and recent results on the universal approximation on the hypersphere by Ng and Kwong (2022). While these two works inspire the general proof strategy, they only offer uniform convergence (i.e., density-type results, Definition 4.1). Instead, we offer a non-asymptotic analysis and develop the first approximation rate results on the sphere for functions of the form of Equation (B.6), i.e., Jackson-type results (Definition 4.3).

The high-level idea of the proof is to split the goal into approximating f with the convolution $f * K_\lambda^{\text{vMF}}$ and approximating the convolution $f * K_\lambda^{\text{vMF}}$ with a sum of terms that have

[1] The w_m term in the normalization constant $c_{m+1}(\lambda)$ is not in (Ng and Kwong, 2022). However, without it K_λ^{vMF} are not an approximate identity.

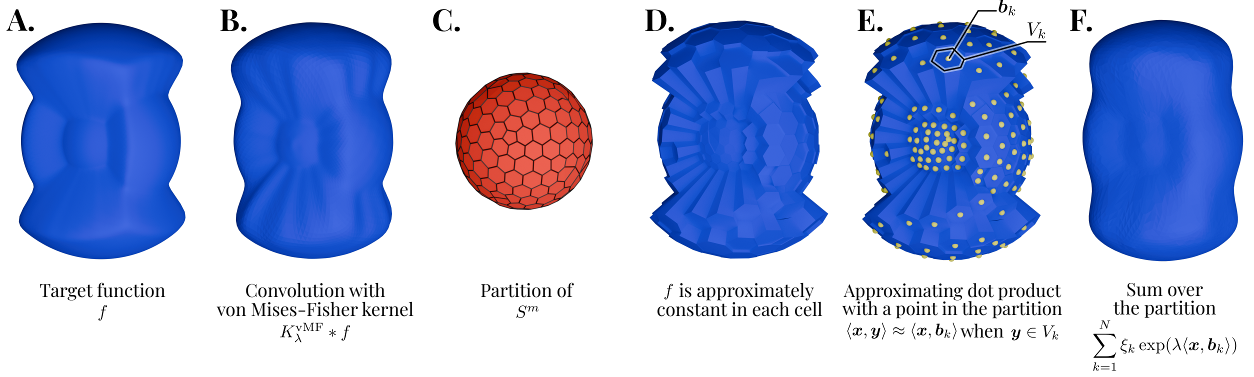


Figure B.2: **Intuition behind the proof of our Jackson-type bound for universal approximation on the hypersphere. A.** We want to approximate a function f over the hypersphere S^m. This illustration is in three-dimensional space, so $m = 2$. **B.** In order to get the $\exp(\lambda\langle \cdot, \boldsymbol{y}\rangle)$ form that we want, we convolve f with the $K_\lambda^{\text{vMF}}(t) = c_{m+1}(\lambda)\exp(\lambda t)$ kernel. **C.** We partition S^m into N cells $V_1, \ldots, V_N$. **D.** Our choice of N is such that f does not vary too much in each cell and hence can be approximated by a function that is constant in each V_k. **E.** As each cell is small, the dot product of $\boldsymbol{x}$ with any point in the cell V_k can be approximated by the dot product of $\boldsymbol{x}$ with a fixed point $\boldsymbol{b}_k \in V_k$. **F.** This allows us to approximate the integral in the convolution K_λ^{vMF} with a finite sum.

the $\exp(t\langle \boldsymbol{x}, \boldsymbol{b}_k\rangle)$ structure resembling the kernel K_λ^{vMF} (Definition B.1.3):

$$\sup_{\boldsymbol{x}\in S^m} \left\| f(\boldsymbol{x}) - \sum_{k=1}^{N} \xi_k \exp(\lambda\langle \boldsymbol{x}, \boldsymbol{b}_k\rangle) \right\| \leq \underbrace{\left\| f - f * K_\lambda^{\text{vMF}} \right\|_\infty}_{\text{Eq. B.8 / Eq. B.2.5}} + \underbrace{\left\| f * K_\lambda^{\text{vMF}} - \sum_{k=1}^{N} \xi_k \exp(\lambda\langle \cdot, \boldsymbol{b}_k\rangle) \right\|_\infty}_{\text{Lemma B.2.7}}. \tag{B.7}$$

This is also illustrated in Figure B.2.

Let's focus on the first term in Equation (B.7). It can be further decomposed into three terms by introducing $W_q \in \mathbb{P}_q(S^m)$, the best approximation of f with a spherical polynomial of degree q:

$$\|K_\lambda^{\text{vMF}} * f - f\|_\infty \leq \underbrace{\|K_\lambda^{\text{vMF}} * f - K_\lambda^{\text{vMF}} * W_q\|_\infty}_{\text{Lemma B.2.2}} + \underbrace{\|K_\lambda^{\text{vMF}} * W_q - W_q\|_\infty}_{\text{Lemma B.2.4}} + \underbrace{\|W_q - f\|_\infty}_{\text{Lemma B.2.1}}. \tag{B.8}$$

There are a number of Jackson-type results for how well finite sums of spherical polynomials approximate functions $f \in V_m$ (the last term in Equation (B.8)). In particular, they are interested in bounding

$$\min_{W_q\in\mathbb{P}_q(S^m)} \|f - W_q\|_p, \; 1 \leq p \leq \infty. \tag{B.9}$$

We will use a simple bound by Ragozin (1971):

Lemma B.2.1 (Ragozin bound). *For $f \in C(S^m)$ and $q \in \mathbb{N}_{>0}$ it holds that:*

$$\min_{W_q \in \mathbb{P}_q(S^m)} \|f - W_q\|_\infty \leq C_R\, \omega\left(f; \frac{1}{q}\right), \tag{B.10}$$

for some constant C_R that does not depend on f or q and ω being the first modulus of continuity of f defined as:

$$\omega(f;t) = \sup\{|f(\boldsymbol{x}) - f(\boldsymbol{y})| \mid \boldsymbol{x}, \boldsymbol{y} \in S^m, \cos^{-1}(\boldsymbol{x}^\top \boldsymbol{y}) \leq t\}.$$

We recommend Atkinson and Han (2012, Chapter 4) and Dai and Xu (2013, Chapter 4) for an overview of the various bounds proposed for Equation (B.9) depending on the continuity properties of f and its derivatives. In particular, the above bound could be improved with a term $1/n^k$ if f has k continuous derivates (Ragozin, 1971).

We can upper-bound the first term in Equation (B.8) by recalling that the norm of the kernel K_λ^{vMF} is 1:

Lemma B.2.2.

$$\|K_\lambda^{vMF} * f - K_\lambda^{vMF} * W_q\|_m \leq \|f - W_q\|_m.$$

Hence:

$$\|K_\lambda^{vMF} * f - K_\lambda^{vMF} * W_q\|_\infty \leq \|f - W_q\|_\infty \leq C_R\, \omega\left(f; \frac{1}{q}\right).$$

Proof. Convolution is linear so $\|K_\lambda^{\text{vMF}} * f - K_\lambda^{\text{vMF}} * W_q\|_m = \|K_\lambda^{\text{vMF}} * (f - W_q)\|_m$. Using the Hölder inequality (Dai and Xu, 2013, Theorem 2.1.2) we get $\|K_\lambda^{\text{vMF}} * f - K_\lambda^{\text{vMF}} * W_q\|_m \leq \|K_\lambda^{\text{vMF}}\|_{1,m} \|f - W_q\|_m$. As $\|K_\lambda^{\text{vMF}}\|_{1,m} = 1$ for all $\lambda > 0, m > 1$, we obtain the inequality in the lemma. For the uniform norm, we also use the Ragozin bound from Lemma B.2.1. □

Only the second term in Equation (B.8) is left. However, before we tackle it, we will need a helper lemma that bounds the eigenvalues of the von Mises-Fisher kernel (Equation (B.4)):

Lemma B.2.3 (Bounds on the eigenvalues $a_k^m(K_\lambda^{\text{vMF}})$). *The eigenvalues a_k^m, as defined in Equation (B.4), for the sequence of von Mises-Fisher kernels (Definition B.1.3) are bounded from below and above as:*

$$0 < \left(\frac{\lambda}{\left(\frac{m-1}{2} + k\right) + \sqrt{\lambda^2 + \left(\frac{m-1}{2} + k\right)^2}} \right)^k \leq a_k^m(K_\lambda^{vMF}) \leq 1$$

Proof. We have

$$a_k^m(K_\lambda^{\text{vMF}}) = \frac{w_{m-1}}{w_m} \int_{-1}^{1} K_\lambda^{\text{vMF}}(t) \frac{Q_k^{(m-1)/2}(t)}{Q_k^{(m-1)/2}(1)} (1 - t^2)^{(m-2)/2} dt$$

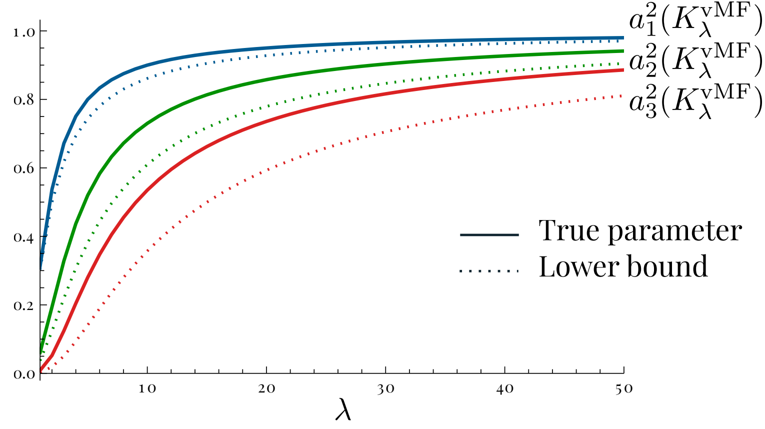


Figure B.3: The coefficients a_k^m for the von Mises-Fisher kernels K_λ^{vMF} for $m = 2$ and $k \in \{1, 2, 3\}$ as well as the lower bound from Lemma B.2.3.

$$
\begin{aligned}
&= \frac{w_{m-1}}{w_m} \int_{-1}^{1} c_{m+1}(\lambda) \exp(\lambda t) \frac{Q_k^{(m-1)/2}(t)}{Q_k^{(m-1)/2}(1)} (1 - t^2)^{(m-2)/2} dt \\
&\stackrel{\star}{=} \frac{I_{\frac{m-1}{2}+k}(\lambda)}{I_{\frac{m-1}{2}}(\lambda)},
\end{aligned}
$$

with $\star$ solved using Mathematica. From here we can see that $a_0^m(K_\lambda^{\text{vMF}}) = 1$ for all $m > 1, \lambda > 1$. Furthermore, for $v > 1$ and $\lambda > 0$ the modified Bessel function of the first kind $I_v(\lambda)$ is monotonically decreasing as v increases. Therefore, $a_k^m(K_\lambda^{\text{vMF}}) \leq a_0^m(K_\lambda^{\text{vMF}}) = 1$, which gives us the upper bound in the lemma.

For the lower bound, we will use the following bound on the ratio of modified Bessel functions by Amos (1974, Eq. 9):

$$
\frac{I_{v+1}(x)}{I_v(x)} \geq \frac{x}{(v+1) + \sqrt{x^2 + (v+1)^2}}, \quad v \geq 0, x \geq 0.
$$

As mentioned above, $0 \leq \frac{I_{v+1}(x)}{I_v(x)} \leq 1$. Furthermore, these ratios are decreasing as v increases, i.e., $\frac{I_{v+2}(x)}{I_{v+1}(x)} \leq \frac{I_{v+1}(x)}{I_v(x)}$ for all $v \geq 0$ and $x \geq 0$ (Amos, 1974, Eq. 10). Combining these facts gives us:

$$
\frac{I_{v+k}(x)}{I_v(x)} \geq \left(\frac{I_{v+k}(x)}{I_{v+k-1}(x)} \right)^k = \left(\frac{x}{(v+k) + \sqrt{x^2 + (v+k)^2}} \right)^k. \tag{B.11}
$$

We can now give the lower bound for $a_k^m(K_n^{\text{vMF}})$ using Equation (B.11) :

$$
a_k^m(K_\lambda^{\text{vMF}}) = \frac{I_{\frac{m-1}{2}+k}(\lambda)}{I_{\frac{m-1}{2}}(\lambda)} \geq \left(\frac{\lambda}{\left(\frac{m-1}{2} + k\right) + \sqrt{\lambda^2 + \left(\frac{m-1}{2} + k\right)^2}} \right)^k.
$$

The lower bound for $m = 2$ is plotted in Figure B.3. □

We can now provide a bound for the second term in Equation (B.8):

Lemma B.2.4. *Take an $f \in C(S^m)$. Furthermore, assume that there exists a constant $C_H \geq 0$ that upper-bounds the norms of the spherical harmonics of any best polynomial approximation W_q of f:*

$$\text{for all } q \geq 1, W_q = \sum_{k=0}^{q} Y_k^m,\ Y_k^m \in \mathbb{Y}(S^m),\ \|Y_k^m\|_\infty \leq C_H,\ \forall k = 0, \dots, q,$$

where $W_q = \arg\min_{h \in \mathbb{P}_q(S^m)} \|f - h\|_\infty$. Then

$$\|K_\lambda^{vMF} * W_q - W_q\|_\infty \leq C_H\, q \left(1 - \left(\frac{\lambda}{\left(\frac{m-1}{2} + q\right) + \sqrt{\lambda^2 + \left(\frac{m-1}{2} + q\right)^2}} \right)^q \right).$$

Proof. Using that W_q is a spherical polynomial of degree q and hence can be expressed as a sum of spherical harmonics $W_q = \sum_{k=0}^{q} Y_k^m$, we get:

$$\begin{aligned}
\|K_\lambda^{\text{vMF}} * W_q - W_q\|_\infty &= \left\| K_\lambda^{\text{vMF}} * \sum_{k=0}^{q} Y_k^m(x) - \sum_{k=0}^{q} Y_k^m(x) \right\|_\infty \\
&= \left\| \sum_{k=0}^{q} \left(K_\lambda^{\text{vMF}} * Y_k^m(x) - Y_k^m(x) \right) \right\|_\infty \\
&\leq \sum_{k=0}^{q} \left\| \left(K_\lambda^{\text{vMF}} * Y_k^m(x) - Y_k^m(x) \right) \right\|_\infty && \text{(Triangle inequality)} \\
&= \sum_{k=0}^{q} \left\| \left(a_k^m(K_\lambda^{\text{vMF}}) Y_k^m(x) - Y_k^m(x) \right) \right\|_\infty && \text{(Lemma B.1.2)} \\
&= \sum_{k=0}^{q} \left\| \left(a_k^m(K_\lambda^{\text{vMF}}) - 1 \right) Y_k^m(x) \right\|_\infty \\
&= \sum_{k=0}^{q} |a_k^m(K_\lambda^{\text{vMF}}) - 1|\, \|Y_k^m(x)\|_\infty \\
&\leq \sum_{k=0}^{q} |a_k^m(K_\lambda^{\text{vMF}}) - 1|\, C_H \\
&\leq C_H \sum_{k=0}^{q} \left(1 - \left(\frac{\lambda}{\left(\frac{m-1}{2} + k\right) + \sqrt{\lambda^2 + \left(\frac{m-1}{2} + k\right)^2}} \right)^k \right) && \text{(Lemma B.2.3)}
\end{aligned}$$

$$\leq C_H \sum_{k=0}^{q} \left(1 - \left(\underbrace{\frac{\lambda}{\left(\frac{m-1}{2}+q\right)+\sqrt{\lambda^2+\left(\frac{m-1}{2}+q\right)^2}}}_{B} \right)^k \right) \qquad \text{(as } k \leq q\text{)}$$

$$= C_H \left(q+1-\sum_{k=0}^{q} B^k \right)$$

$$\leq C_H \left(q+1-(1+qB^q) \right) \qquad \text{(using that } 0<B<1\text{)}$$

$$\leq C_H \, q \left(1-B^q\right).$$

□

We can finally combine Lemmas B.2.1, B.2.2, and B.2.4 in order to provide an upper bound to Equation (B.8):

Lemma B.2.5 (Bound on $\|K_\lambda^{\mathrm{vMF}} * f - f\|_\infty$)**.** *Take an $f \in V_m$ with modulus of continuity $\omega(f;t) \leq Lt$. As in Lemma B.2.4, assume that there exists a constant $C_H \geq 0$ that upper-bounds the norms of the spherical harmonics of any best polynomial approximation W_q of f:*

$$\text{for all } q \geq 1, W_q = \sum_{k=0}^{q} Y_k^m,\ Y_k^m \in \mathbb{Y}(S^m),\ \|Y_k^m\|_\infty \leq C_H,\ \forall k=0,\ldots,q,$$

where $W_q = \arg\min_{h \in \mathbb{P}_q(S^m)} \|f-h\|_\infty$. If $\lambda \geq \Lambda(\epsilon)$, where

$$\Lambda(\epsilon) = \frac{(8LC_R + m\epsilon + \epsilon)\left(1-\frac{\epsilon^2}{8LC_HC_R+2\epsilon C_H}\right)^{\frac{\epsilon}{4LC_R+\epsilon}}}{\epsilon\left(1-\left(1-\frac{\epsilon^2}{8LC_HC_R+2\epsilon C_H}\right)^{\frac{2\epsilon}{4LC_R+\epsilon}}\right)} = \mathcal{O}\left(\frac{L^3C_HC_R^3}{\epsilon^4}\right). \tag{B.12}$$

then $\|f - K_\lambda^{vMF} * f\|_\infty \leq \epsilon$.

Proof. As we want to upper-bound Equation (B.8) with ϵ, we will split our ϵ budget over the three terms.

For the first and the third terms, using the Ragozin bound from Lemma B.2.1 we have:

$$\|f - W_q\|_\infty \leq C_R\ \omega\left(f;\frac{1}{q}\right) \leq \frac{C_R L}{q},\ q \geq 1. \tag{B.13}$$

We want to select an integer q large enough so that $\|f - W_q\|_\infty \leq \epsilon/4$. That is $q = \left\lceil \frac{4C_RL}{\epsilon} \right\rceil$. This will be how we bound the first and last terms in Equation (B.8).

Let's focus on the second term. Pick W_q to be the best approximation from the Ragozin bound. From Lemma B.2.4 we have that

$$\|K_\lambda^{\mathrm{vMF}} * W_q - W_q\|_\infty \leq C_H \, q \left(1-B^q\right),$$

where

$$B = \frac{\lambda}{\left(\frac{m-1}{2} + q\right) + \sqrt{\lambda^2 + \left(\frac{m-1}{2} + q\right)^2}}.$$

The error budget we want to allocate for the $\|K_\lambda^{\text{vMF}} * W_q - W_q\|_\infty$ term is $\epsilon/2$. Hence:

$$B \geq \left(1 - \frac{\epsilon}{2C_H q}\right)^{1/q} = D \implies \|K_n^{\text{vMF}} * W_q - W_q\|_\infty \leq \frac{\epsilon}{2}. \tag{B.14}$$

We just need to find the minimum value for λ such that Equation (B.14) holds. We have:

$$B = \frac{\lambda}{\underbrace{\left(\frac{m-1}{2} + q\right)}_{E} + \sqrt{\lambda^2 + \left(\frac{m-1}{2} + q\right)^2}} = \frac{\lambda}{E + \sqrt{\lambda^2 + E^2}}. \tag{B.15}$$

Then, combining Equations (B.14) and (B.15) we get:

$$\frac{\lambda}{E + \sqrt{\lambda^2 + E^2}} \geq D$$
$$\lambda \geq \frac{2DE}{1 - D^2}.$$

Finally, replacing D and E with the expressions in Equations (B.14) and (B.15), then upper-bounding $q = \left\lceil \frac{4C_R L}{\epsilon} \right\rceil$ as $q \geq \frac{4C_R L}{\epsilon} + 1$ and finally simplifying the expression we get our final bound for λ. If $\lambda \geq \Lambda(\epsilon)$, with

$$\Lambda(\epsilon) = \frac{(8LC_R + m\epsilon + \epsilon)\left(1 - \frac{\epsilon^2}{8LC_H C_R + 2\epsilon C_H}\right)^{\frac{\epsilon}{4LC_R + \epsilon}}}{\epsilon\left(1 - \left(1 - \frac{\epsilon^2}{8LC_H C_R + 2\epsilon C_H}\right)^{\frac{2\epsilon}{4LC_R + \epsilon}}\right)},$$

then $\|K_\lambda^{\text{vMF}} * W_q - W_q\|_\infty \leq \epsilon/2$.

Hence, for any $\lambda \geq \Lambda(\epsilon)$ we have:

$$\begin{aligned}
\|K_\lambda^{\text{vMF}} * f - f\|_\infty &\leq \|K_\lambda^{\text{vMF}} * f - K_\lambda^{\text{vMF}} * W_q\|_\infty + \|K_\lambda^{\text{vMF}} * W_q - W_q\|_\infty + \|W_q - f\|_\infty \\
&\leq \|f - W_q\|_\infty + \frac{\epsilon}{2} + \|W_q - f\|_\infty \\
&\leq \frac{\epsilon}{4} + \frac{\epsilon}{2} + \frac{\epsilon}{4} \\
&= \epsilon.
\end{aligned}$$

This concludes our bound on Equation (B.8).

Finally, to give the asymptotic behavior of $\Lambda(\epsilon)$ as $\epsilon \to 0$ we observe that the Taylor series expansion of Λ around $\epsilon = 0$ is:

$$\Lambda(\epsilon) = \frac{128L^3 C_H C_R^3}{\epsilon^4} + \frac{16L^2(m-1)C_H C_R^2}{\epsilon^3} + \mathcal{O}\left(\frac{1}{\epsilon^2}\right),$$

hence:

$$\Lambda(\epsilon) = \mathcal{O}\left(\frac{L^3 C_H C_R^3}{\epsilon^4}\right).$$

□

Lemma B.2.5 is our bound on Equation (B.8) which is also the first term of Equation (B.7). Recall that bounding Equation (B.7) is our ultimate goal. Hence, we are halfway done with our proof. Let's focus now on the second term in Equation (B.7), that is, how well we can approximate the convolution of f with the von Mises-Fisher kernel using a finite sum:

$$\left\| f * K_\lambda^{\text{vMF}} - \sum_{k=1}^{N} \xi_k \exp(\lambda\langle\cdot, \boldsymbol{b}_k\rangle) \right\|_\infty .$$

The basic idea behind bounding this term is that we can partition the hypersphere S^m into N sets ($\{V_1, \ldots, V_N\}$), each small enough so that, for a fixed $\boldsymbol{x} \in S^m$, the $K(\langle \boldsymbol{x}, \boldsymbol{y}\rangle) f(\boldsymbol{y})$ term in the convolution

$$(K * f)(\boldsymbol{x}) = \frac{1}{w_m} \int_{S^m} K(\langle \boldsymbol{x}, \boldsymbol{y}\rangle) f(\boldsymbol{y})\, dw_m(\boldsymbol{y})$$

is almost the same for all values $\boldsymbol{y} \in V_k$ in that element of the partition. Hence we can approximate the integral over the partition with estimate over a single point $\boldsymbol{b}_k$:

$$\int_{V_k} K(\langle \boldsymbol{x}, \boldsymbol{y}\rangle) f(\boldsymbol{y})\, dw_m(\boldsymbol{y}) \approx |V_k|\; K(\langle \boldsymbol{x}, \boldsymbol{b}_k\rangle)\; f(\boldsymbol{b}_k).$$

The rest of this section will make this formal.

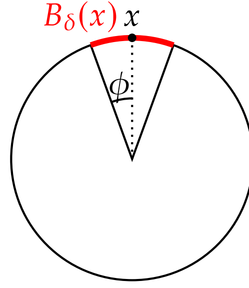
First, in order to construct our partition $\{V_1, \ldots, V_N\}$ of S^m we use a cover of S^m. Then, our partition will be such that each element V_k is a subset of the corresponding element of the cover of S^m. In this way, we can control the maximum size of the elements of the cover.

Lemma B.2.6. *Consider a cover $\{B_\delta(\boldsymbol{b}_1), \ldots, B_\delta(\boldsymbol{b}_{N_\delta^m})\}$ of S^m by N_δ^m hyperspherical caps $B_\delta(\boldsymbol{x}) = \{\boldsymbol{y} \in S^m \mid \langle \boldsymbol{x}, \boldsymbol{y}\rangle \geq 1 - \delta\}$ for $0 < \delta < 1$, centred at $\boldsymbol{b}_1, \ldots, \boldsymbol{b}_{N_\delta^m} \in S^m$. By cover we mean that $\bigcup_{i=1}^{N_\delta^m} B_\delta(\boldsymbol{b}_i) = S^m$, with N_δ^m being the smallest number of hyperspherical caps to cover S^m (its covering number). Then, for $m \geq 8$ we have:*

$$\frac{2}{I_{(\delta(2-\delta))}\left(\frac{m}{2}, \frac{1}{2}\right)} \leq N_\delta^m < \frac{\Phi(m)}{(\delta(2-\delta))^{\frac{m+1}{2}}} < \frac{\Phi(m)}{\delta^{m+1}},$$

with I being the regularized incomplete beta function and $\Phi(m) = \mathcal{O}(m \log m)$ being a function that depends only on the dimension m.

Proof. Define $\phi = \cos^{-1}(1 - \delta)$:



Naturally, the area of the caps needs to be at least as much as the area of the hypersphere for the set of caps to be a cover. This gives us our lower bound. The area of a cap with colatitude angle ϕ as above is (Li, 2010):

$$w_m^\phi = \frac{1}{2} w_m I_{\sin^2 \phi}\left(\frac{m}{2}, \frac{1}{2}\right).$$

As $\sin\phi = \sin\left(\cos^{-1}(1-\delta)\right) = \sqrt{1-(1-\delta)^2} = \sqrt{\delta(2-\delta)}$, we have our lower bound:

$$N_\delta^m \geq \frac{w_m}{w_m^\phi} = \frac{2}{I_{\sin^2\phi}\left(\frac{m}{2}, \frac{1}{2}\right)} = \frac{2}{I_{(\delta(2-\delta))}\left(\frac{m}{2}, \frac{1}{2}\right)}$$

For the upper bound, we can use the observation that if a unit *ball* is covered with *balls* of radius r, then the unit *sphere* is also covered with the same number of *caps* of radius r. From (Rogers, 1963, intermediate result from the proof of theorem 3) we have that for $m \geq 8$ and $1/r \geq m+1$, a unit ball can be covered by less than

$$e\left((m+1)\log(m+1) + (m+1)\log\log(m+1) + 5(m+1)\right)\frac{1}{r^{m+1}} = \Phi(m)\frac{1}{r^{m+1}}$$

balls of radius r. For high m, this is a pretty good approximation since most of the volume of the hypersphere lies near its surface. Our caps B_δ can fit inside balls of radius $r = \sin\phi$. Hence, we have the upper bound:

$$N_\delta^m < \frac{\Phi(m)}{\sin^{m+1}\phi} = \frac{\Phi(m)}{(\delta(2-\delta))^{\frac{m+1}{2}}}.$$

□

Now we can use a partition resulting from this covering in order to bound the error between the integral and its Riemannian sum approximation:

Lemma B.2.7 (Approximation via Riemann sums). *Let $g(x,y) : S^m \times S^m \to \mathbb{R}$ with $m \geq 8$ be a continuous function with modulus of continuity for both arguments $\omega(g(\cdot;\boldsymbol{y}),t) \leq Lt$, $\forall \boldsymbol{y} \in S^m$ and $\omega(g(\boldsymbol{x},\cdot);t) \leq Lt$, $\forall \boldsymbol{x} \in S^m$. Take any $0 < \delta < 1$. Then, there exists a partition $\{V_1, \ldots, V_{N_\delta^m}\}$ of S^m into $N_\delta^m = \lceil \Phi(m)/\delta^{m+1} \rceil$ subsets, as well as $\boldsymbol{b}_1, \ldots, \boldsymbol{b}_{N_\delta^m} \in S^m$ such that:*

$$\max_{x\in S^m} \frac{1}{w_m}\left|\int_{S^m} g(\boldsymbol{x},\boldsymbol{y})\, dw_m(\boldsymbol{y}) - \sum_{k=1}^{N_\delta^m} g(\boldsymbol{x},\boldsymbol{b}_k)\, w_m(V_k)\right| \leq 3L\cos^{-1}(1-\delta).$$

Here, $\Phi(m) = \mathcal{O}(m \log m)$ is a function that depends only on the dimension m.

Proof. This proof is a non-asymptotic version of the proof of Lemma 4.3 from Ng and Kwong (2022). First, we can use Lemma B.2.6 to construct a covering $\{B_\delta(\boldsymbol{b}_1), \ldots, B_\delta(\boldsymbol{b}_{N_\delta^m})\}$ of S^m. If we have a covering of S^m it is trivial to construct a partition of it $\{V_1, \ldots, V_{N_\delta^m}\}, \bigcup_{k=1}^{N_\delta^m} V_k = S^m$, $V_i \cap V_j = \emptyset, i \neq j$ such that $V_k \subseteq B_\delta(\boldsymbol{b}_k), \forall k$. This partition can also be selected to be such that all elements of it have the same measure $w_m(V_1) = w_m(V_i), \forall i$ (Feige and Schechtman, 2002, Lemma 21). While this is not necessary for this proof, we will use this equal measure partition in Lemma B.3.2 and Theorem B.3.3.

We can then use the triangle inequality to split the term we want to bound in three separate terms:

$$
\begin{aligned}
&\left| \int_{S^m} g(\boldsymbol{x}, \boldsymbol{y}) \, dw_m(\boldsymbol{y}) - \sum_{k=1}^{N_\delta^m} g(\boldsymbol{x}, \boldsymbol{b}_k) \, w_m(V_k) \right| \\
\leq &\left| \int_{S^m} g(\boldsymbol{x}, \boldsymbol{y}) \, dw_m(\boldsymbol{y}) - \int_{S^m} g(\boldsymbol{b}_\star, \boldsymbol{y}) \, dw_m(\boldsymbol{y}) \right| \\
+ &\left| \int_{S^m} g(\boldsymbol{b}_\star, \boldsymbol{y}) \, dw_m(\boldsymbol{y}) - \sum_{k=1}^{N_\delta^m} g(\boldsymbol{b}_\star, \boldsymbol{b}_k) \, w_m(V_k) \right| \\
+ &\left| \sum_{k=1}^{N_\delta^m} g(\boldsymbol{b}_\star, \boldsymbol{b}_k) \, w_m(V_k) - \sum_{k=1}^{N_\delta^m} g(\boldsymbol{x}, \boldsymbol{b}_k) \, w_m(V_k) \right|,
\end{aligned}
\tag{B.16}
$$

where $\boldsymbol{b}_\star$ is the center of one of the caps whose corresponding partition contains $\boldsymbol{x}$, i.e., $\boldsymbol{b}_\star = \boldsymbol{b}_i \iff \boldsymbol{x} \in V_i$. Due to $\{V_1, \ldots, V_{N_\delta^m}\}$ being a partition, $\boldsymbol{b}_\star$ is well-defined as $\boldsymbol{x}$ is in exactly one of the elements of the partition.

Observe also that the modulus of continuity gives us a Lipschitz-like bound, i.e., if $\langle \boldsymbol{x}, \boldsymbol{y} \rangle \geq 1 - \delta$ for $\boldsymbol{x}, \boldsymbol{y} \in S^m$ and $\omega(f; t) \leq Lt$, then

$$
|f(\boldsymbol{x}) - f(\boldsymbol{y})| \leq \omega(f; \cos^{-1}(\langle \boldsymbol{x}, \boldsymbol{y} \rangle)) \leq \omega(f; \cos^{-1}(1-\delta)) \leq L \cos^{-1}(1-\delta). \tag{B.17}
$$

Let's start with the first term in Equation (B.16). Using the fact that we selected $\boldsymbol{b}_\star$ to be such that $\langle \boldsymbol{b}_\star, \boldsymbol{x} \rangle \geq 1 - \delta$ and Equation (B.17), we have:

$$
\begin{aligned}
\left| \int_{S^m} g(\boldsymbol{x}, \boldsymbol{y}) \, dw_m(\boldsymbol{y}) - \int_{S^m} g(\boldsymbol{b}_\star, \boldsymbol{y}) \, dw_m(\boldsymbol{y}) \right| &= \left| \int_{S^m} (g(\boldsymbol{x}, \boldsymbol{y}) - g(\boldsymbol{b}_\star, \boldsymbol{y})) \, dw_m(\boldsymbol{y}) \right| \\
&\leq \int_{S^m} \left| g(\boldsymbol{x}, \boldsymbol{y}) - g(\boldsymbol{b}_\star, \boldsymbol{y}) \right| \, dw_m(\boldsymbol{y}) \\
&\leq \int_{S^m} L \cos^{-1}(1-\delta) \, dw_m(\boldsymbol{y}) \\
&= L \cos^{-1}(1-\delta) \int_{S^m} dw_m(\boldsymbol{y})
\end{aligned}
$$

$$= L \cos^{-1}(1-\delta)\, w_m.$$

We can similarly upper-bound the second term of Equation (B.16) using also the fact that $\{V_k\}$ is a partition of S^m:

$$\begin{aligned}
&\left| \int_{S^m} g(\boldsymbol{b}_\star, \boldsymbol{y})\, dw_m(\boldsymbol{y}) - \sum_{k=1}^{N_\delta^m} g(\boldsymbol{b}_\star, \boldsymbol{b}_k)\, w_m(V_k) \right| \\
=& \left| \sum_{k=1}^{N_\delta^m} \int_{V_k} g(\boldsymbol{b}_\star, \boldsymbol{y})\, dm_w(\boldsymbol{y}) - \sum_{k=1}^{N_\delta^m} g(\boldsymbol{b}_\star, \boldsymbol{b}_k)\, w_m(V_k) \right| \\
=& \left| \sum_{k=1}^{N_\delta^m} \int_{V_k} \left(g(\boldsymbol{b}_\star, \boldsymbol{y}) - g(\boldsymbol{b}_\star, \boldsymbol{b}_k) \right)\, dm_w(\boldsymbol{y}) \right| \\
\leq& \sum_{k=1}^{N_\delta^m} \int_{V_k} \left| g(\boldsymbol{b}_\star, \boldsymbol{y}) - g(\boldsymbol{b}_\star, \boldsymbol{b}_k) \right|\, dm_w(\boldsymbol{y}) \\
\leq& \sum_{k=1}^{N_\delta^m} \int_{V_k} L \cos^{-1}(1-\delta)\, dm_w(\boldsymbol{y}) \\
=& L \cos^{-1}(1-\delta) \sum_{k=1}^{N_\delta^m} \int_{V_k} dm_w(\boldsymbol{y}) \\
=& L \cos^{-1}(1-\delta) w_m.
\end{aligned}$$

And analogously, for the third term we get:

$$\begin{aligned}
\left| \sum_{k=1}^{N_\delta^m} g(\boldsymbol{b}_\star, \boldsymbol{b}_k)\, w_m(V_k) - \sum_{k=1}^{N_\delta^m} g(\boldsymbol{x}, \boldsymbol{b}_k)\, w_m(V_k) \right| &= \left| \sum_{k=1}^{N_\delta^m} \left(g(\boldsymbol{b}_\star, \boldsymbol{b}_k) - g(\boldsymbol{x}, \boldsymbol{b}_k) \right) w_m(V_k) \right| \\
&\leq \sum_{k=1}^{N_\delta^m} \left| g(\boldsymbol{b}_\star, \boldsymbol{b}_k) - g(\boldsymbol{x}, \boldsymbol{b}_k) \right|\, w_m(V_k) \\
&\leq L \cos^{-1}(1-\delta) \sum_{k=1}^{N_\delta^m} w_m(V_k) \\
&= L \cos^{-1}(1-\delta) w_m.
\end{aligned}$$

Finally, observing that the above bounds do not depend on the choice of $\boldsymbol{x} \in S^m$ and combining the three results we obtain our desired bound. □

By observing that we can set $g(\boldsymbol{x}, \boldsymbol{y}) = K_\lambda^{\text{vMF}}(\langle \boldsymbol{x}, \boldsymbol{y} \rangle) f(\boldsymbol{y})$, it becomes clear how Lemma B.2.7 can be used to bound the second term in Equation (B.7). For that we will also need to know what is the modulus of continuity of the von Mises-Fisher kernels K_λ^{vMF}.

Lemma B.2.8 (Modulus of continuity of K_λ^{vMF})**.** *The von Mises-Fisher kernels K_λ^{vMF} have modulus of continuity $\omega(K_\lambda^{vMF}; t) \leq c_{m+1}(\lambda)\exp(\lambda)$.*

Proof. Recall that $K_\lambda^{\text{vMF}}(u)$ is defined on $u \in [-1, 1]$. $K_\lambda^{\text{vMF}}(u)$ and its derivative are both monotonically increasing in u. Hence:

$$\omega(K_\lambda^{\text{vMF}}; t) = \sup\left\{|K_\lambda^{\text{vMF}}(\langle \boldsymbol{z}, \boldsymbol{x}\rangle) - K_\lambda^{\text{vMF}}(\langle \boldsymbol{z}, \boldsymbol{y}\rangle) \mid \boldsymbol{x}, \boldsymbol{y} \in S^m, \cos^{-1}(\langle \boldsymbol{x}, \boldsymbol{y}\rangle) \leq t\right\}.$$

Using the mean value theorem we know there exists a $\boldsymbol{d} \in S^m$ such that

$$\begin{aligned}
|K_\lambda^{\text{vMF}}(\langle \boldsymbol{z}, \boldsymbol{x}\rangle) - K_\lambda^{\text{vMF}}(\langle \boldsymbol{z}, \boldsymbol{y}\rangle)| &= |(K_\lambda^{\text{vMF}})'(\langle \boldsymbol{z}, \boldsymbol{d}\rangle)\,(\langle \boldsymbol{z}, \boldsymbol{x}\rangle - \langle \boldsymbol{z}, \boldsymbol{y}\rangle)| \\
&= (K_\lambda^{\text{vMF}})'(\langle \boldsymbol{z}, \boldsymbol{d}\rangle)\,|\langle \boldsymbol{z}, \boldsymbol{x}\rangle - \langle \boldsymbol{z}, \boldsymbol{y}\rangle| \\
&= \lambda c_{m+1}(\lambda)\exp(\lambda\langle \boldsymbol{z}, \boldsymbol{d}\rangle)\,|\langle \boldsymbol{z}, \boldsymbol{x}\rangle - \langle \boldsymbol{z}, \boldsymbol{y}\rangle| \\
&\leq \lambda c_{m+1}(\lambda)\exp(\lambda)\,|\langle \boldsymbol{z}, \boldsymbol{x}\rangle - \langle \boldsymbol{z}, \boldsymbol{y}\rangle| \\
&= \lambda c_{m+1}(\lambda)\exp(\lambda)\|\boldsymbol{z}\|_2\|\boldsymbol{x} - \boldsymbol{y}\|_2\cos(\text{angle}(\boldsymbol{z}, \boldsymbol{x} - \boldsymbol{y})) \\
&\leq \lambda c_{m+1}(\lambda)\exp(\lambda)\|\boldsymbol{x} - \boldsymbol{y}\|_2.
\end{aligned}$$

Using the law of cosines and that the angle between $\boldsymbol{x}$ and $\boldsymbol{y}$ is less than t:

$$\begin{aligned}
\|\boldsymbol{x} - \boldsymbol{y}\|_2 &= \sqrt{\|\boldsymbol{x}\|_2^2 + \|\boldsymbol{y}\|_2^2 - 2\|\boldsymbol{x}\|_2^2\|\boldsymbol{y}\|_2^2\cos(\text{angle}(\boldsymbol{x}, \boldsymbol{y}))} \\
&= \sqrt{2 - 2\cos(\text{angle}(\boldsymbol{x}, \boldsymbol{y}))} \\
&\leq \sqrt{2 - 2\cos t} \\
&\leq t.
\end{aligned}$$

Hence:

$$\omega(K_\lambda^{\text{vMF}}; t) \leq \lambda c_{m+1}(\lambda)\exp(\lambda)t.$$

□

Our final result, a bound on Equation (B.7), combines Lemma B.2.5 and Lemma B.2.7, each bounding one of the two terms in Equation (B.7).

Theorem B.2.9 (Jackson-type bound for universal approximation on the hypersphere, Theorem 4.12 in the main text)**.** *Let $f \in C(S^m)$ be a continuous function on S^m with modulus of continuity $\omega(f; t) \leq Lt$ for some $L \in \mathbb{R}_{>0}$ and $m \geq 8$. Assume that there exists a constant $C_H \geq 0$ that upper-bounds the norms of the spherical harmonics of any best polynomial approximation W_q of f:*

$$\textit{for all } q \geq 1, W_q = \sum_{k=0}^{q} Y_k^m,\ Y_k^m \in \mathbb{Y}(S^m),\ \|Y_k^m\|_\infty \leq C_H,\ \forall k = 0, \ldots, q,$$

where $W_q = \arg\min_{h \in \mathbb{P}_q(S^m)} \|f - h\|_\infty$. Then, for any $\epsilon > 0$, there exist $\xi_1, \ldots, \xi_N \in \mathbb{R}$ and $\boldsymbol{b}_1, \ldots, \boldsymbol{b}_N \in S^m$ such that

$$\sup_{\boldsymbol{x} \in S^m}\left|f(\boldsymbol{x}) - \sum_{k=1}^{N}\xi_k\exp(\lambda\langle \boldsymbol{x}, \boldsymbol{b}_k\rangle)\right| \leq \epsilon,$$

where $\lambda = \Lambda(\epsilon/2)$ (Equation (B.12)) and for any N such that

$$N \geq N(\lambda, \epsilon) = \Phi(m) \left(\frac{3\pi(L + \lambda \|f\|_\infty) c_{m+1}(\lambda) \exp(\lambda)}{\epsilon} \right)^{2(m+1)} = \mathcal{O}(\epsilon^{-10-14m-4m^2}). \tag{B.18}$$

Proof. Recall the decomposition in Equation (B.7). We will split our error budget ϵ in half. Hence, we first select λ such that approximating f with its convolution with K_λ^{vMF} results in an error at most $\lambda/2$. Then, using this λ, we find how finely we need to partition S^m in order to be able to approximate the convolution with a sum up to an error $\epsilon/2$.

Let's select how "peaky" we need the kernel K_λ^{vMF} to be, that is, how big should λ be. From Lemma B.2.5 we have that if $\lambda = \Lambda(\epsilon/2)$, then we would have $\|f - f * K_\lambda^{\text{vMF}}\|_\infty \leq \epsilon/2$.

Now, for the second term in Equation (B.7), consider Lemma B.2.7 with

$$g(\boldsymbol{x}, \boldsymbol{y}) = K_\lambda^{\text{vMF}}(\langle \boldsymbol{x}, \boldsymbol{y} \rangle) f(\boldsymbol{y}).$$

From Lemma B.2.8 we have that the modulus of continuity of K_λ^{vMF} is

$$\omega(K_\lambda^{\text{vMF}}; t) \leq t \; \lambda \; c_{m+1}(\lambda) \exp(\lambda).$$

Hence, we have modulus of continuity for $g(\boldsymbol{x}, \boldsymbol{y})$ being bounded as:

$$\begin{aligned} \omega(g; t) &\leq \|K_\lambda^{\text{vMF}}\|_\infty \omega(f; t) + \|f\|_\infty \omega(K_\lambda^{\text{vMF}}; t) \\ &\leq K_\lambda^{\text{vMF}}(1) \; Lt + \|f\|_\infty \lambda c_{m+1}(\lambda) \exp(\lambda) \; t \\ &= c_{m+1}(\lambda) \exp(\lambda) \; Lt + \|f\|_\infty \lambda c_{m+1}(\lambda) \exp(\lambda) \; t \\ &= c_{m+1}(\lambda) \exp(\lambda) \left(L + \lambda \|f\|_\infty \right) t. \end{aligned}$$

Take

$$\delta = \left(\frac{2\epsilon}{6\pi(L + \lambda \|f\|_\infty) c_{m+1}(\lambda) \exp(\lambda)} \right)^2.$$

Then, by Lemma B.2.7, there exists a partition $\{V_1, \ldots, V_N\}$ of S^m and $\boldsymbol{b}_1, \ldots, \boldsymbol{b}_N \in S^m$ for N as in the lemma such that:

$$\max_{\boldsymbol{x} \in S^m} \left| \frac{1}{w_m} \int_{S^m} K_\lambda^{\text{vMF}}(\langle \boldsymbol{x}, \boldsymbol{y} \rangle) f(\boldsymbol{y}) \, dw_m(\boldsymbol{y}) - \frac{1}{w_m} \sum_{k=1}^N K_\lambda^{\text{vMF}}(\langle \boldsymbol{x}, \boldsymbol{b}_k \rangle) f(\boldsymbol{b}_k) \, w_m(V_k) \right| \leq 3(L + \lambda \|f\|_\infty) c_{m+1} \exp(\lambda) \cos^{-1}(1 - \delta).$$

As $(2x/\pi)^2 < 1 - \cos(x)$ (Bagul and Panchal, 2018, Theorem 1), we have:

$$\delta = \left(\frac{2\epsilon}{6\pi(L + \lambda \|f\|_\infty) c_{m+1}(\lambda) \exp(\lambda)} \right)^2 < 1 - \cos\left(\frac{\epsilon}{6(L + \lambda \|f\|_\infty) c_{m+1}(\lambda) \exp(\lambda)} \right). \tag{B.19}$$

Hence:

$$\max_{\boldsymbol{x} \in S^m} \left| \frac{1}{w_m} \int_{S^m} K_\lambda^{\text{vMF}}(\langle \boldsymbol{x}, \boldsymbol{y} \rangle) f(\boldsymbol{y}) \, dw_m(\boldsymbol{y}) - \frac{1}{w_m} \sum_{k=1}^N K_\lambda^{\text{vMF}}(\langle \boldsymbol{x}, \boldsymbol{b}_k \rangle) f(\boldsymbol{b}_k) \, w_m(V_k) \right|$$

$$\leq 3(L + \lambda\|f\|_\infty)c_{m+1}\exp(\lambda)\cos^{-1}(1-\delta)$$
$$< 3(L + \lambda\|f\|_\infty)c_{m+1}\exp(\lambda)\cos^{-1}\left(\cos\left(\frac{\epsilon}{6(L+\lambda\|f\|_\infty)c_{m+1}(\lambda)\exp(\lambda)}\right)\right)$$
$$= \epsilon/2.$$

Combining the two results we have:

$$\begin{aligned}
&\left\| f - \frac{1}{w_m}\sum_{k=1}^{N} K_\lambda^{\mathrm{vMF}}(\langle \boldsymbol{x}, \boldsymbol{b}_k\rangle) f(\boldsymbol{y})\, w_m(V_k)\right\|_\infty \\
\leq &\| f - f * K_\lambda^{\mathrm{vMF}}\|_\infty + \left\| f * K_\lambda^{\mathrm{vMF}} - \frac{1}{w_m}\sum_{k=1}^{N} K_\lambda^{\mathrm{vMF}}(\langle \boldsymbol{x}, \boldsymbol{b}_k\rangle) f(\boldsymbol{y})\, w_m(V_k)\right\|_\infty \\
\leq &\epsilon/2 + \epsilon/2 \\
= &\epsilon.
\end{aligned}$$

Now, the only thing left is to show that this expression can be expressed in the form of Equation (B.6).

$$\begin{aligned}
\frac{1}{w_m}\sum_{k=1}^{N_\delta^m} K_\lambda^{\mathrm{vMF}}(\langle \boldsymbol{x}, \boldsymbol{b}_k\rangle)\, f(\boldsymbol{b}_k)\, w_m(V_k) &= \sum_{k=1}^{N_\delta^m} \frac{1}{w_m} c_{m+1}(\lambda)\, \exp(\lambda\langle \boldsymbol{x}, \boldsymbol{b}_k\rangle)\, f(\boldsymbol{b}_k)\, w_m(V_k) \\
&= \sum_{k=1}^{N_\delta^m} \xi_k \exp(\lambda\langle \boldsymbol{x}, \boldsymbol{b}_k\rangle),
\end{aligned}$$

with

$$\xi_k = c_{m+1}(\lambda) f(\boldsymbol{b}_k)\frac{w_m(V_k)}{w_m}. \tag{B.20}$$

If we have chosen a partition of equal measure this further simplifies to

$$\xi_k = \frac{c_{m+1}(\lambda)}{N} f(\boldsymbol{b}_k).$$

Hence, for this choice of Λ, N and $\boldsymbol{b}_k$ and ξ_k constructed as above, we indeed have

$$\sup_{\boldsymbol{x}\in S^m}\left| f(\boldsymbol{x}) - \sum_{k=1}^{N} \xi_k \exp(\lambda\langle \boldsymbol{x}, \boldsymbol{b}_k\rangle)\right| \leq \epsilon.$$

Finally, let's study the asymptotic growth of N as $\epsilon \to 0$. We have:

$$N(\lambda, \epsilon) = \Phi(m)\left(\frac{3\pi(L+\lambda\|f\|_\infty)c_{m+1}(\lambda)\exp(\lambda)}{\epsilon}\right)^{2(m+1)}.$$

$\Phi(m)$ is constant in ϵ so we can ignore it. Expanding c_{m+1} and dropping the terms that do not depend on ϵ gives us:

$$\mathcal{O}\left(\frac{(L+\lambda\|f\|_\infty)\lambda^{\frac{m+1}{2}-1}\exp(\lambda)}{\epsilon\, I_{\frac{m+1}{2}-1}(\lambda)}\right)^{2(m+1)}. \tag{B.21}$$

The asymptotics of the modified Bessel function of the first kind are difficult to analyse. However, as we care about an upper bound, we can simplify the expression by lower-bounding $I_\nu(\lambda)$ using Equation (B.11) and that $I_0(\lambda) \geq C\exp(\lambda)/\sqrt{\lambda}$ for $\lambda > 1/2$ (Barnett, 2021):

$$I_\nu(\lambda) \geq C\left(\frac{\sqrt{\nu^2+\lambda^2}-\nu}{\lambda}\right)^{\nu+1}\frac{\exp\left(\sqrt{\nu^2+\lambda^2}\right)}{\sqrt{\lambda}},$$

for some constant C. Plugging this in Equation (B.21), replacing λ with its asymptotic growth ϵ^{-4} and taking the Taylor series expansion at for $\epsilon \to 0$ gives us:

$$\mathcal{O}(\epsilon^{-10-14m-4m^2}).$$

□

We can easily extend Theorem B.2.9 to vector-valued functions:

Corollary B.2.10 (Corollary 4.14 in the main text). *Let $f : S^m \to \mathbb{R}^{m+1}$, $m \geq 8$ be such that each component f_i is in $C(S^m)$, $i = 1, \ldots, m+1$ and satisfies the conditions in Theorem B.2.9. Furthermore, define $\|f\|_\infty = \max_{1\leq i\leq m+1}\|f_i\|_\infty$. Then, for any $\epsilon > 0$, there exist $\boldsymbol{\xi}_1, \ldots, \boldsymbol{\xi}_N \in \mathbb{R}^{m+1}$ and $\boldsymbol{b}_1, \ldots, \boldsymbol{b}_N \in S^m$ such that*

$$\sup_{\boldsymbol{x}\in S^m}\left\| f(\boldsymbol{x}) - \sum_{k=1}^{N}\boldsymbol{\xi}_k \exp(\lambda\langle \boldsymbol{x}, \boldsymbol{b}_k\rangle)\right\|_2 \leq \epsilon,$$

with $\lambda = \Lambda(\epsilon/2\sqrt{m+1})$ for any $N \geq N(\lambda, \epsilon/\sqrt{m+1})$.

Proof. The proof is the same as for Theorem B.2.9. As the concentration parameter λ of the kernels K_λ^{vMF} depends only on the smoothness properties of the individual components and these are assumed to be the same, the same kernel choice can be used for all components f_i. Furthermore, the choice of partition is independent of the function to be approximated and depends only on the concentration parameter of the kernel. Hence, we can also use the same partition for all components f_i. We only need to take into account that:

$$\|x-y\|_2 = \sqrt{\sum_{i=1}^{m+1}|x_i-y_i|^2} \leq \sqrt{(m+1)\epsilon^2} = \sqrt{m+1}\epsilon$$

for all $x, y \in \mathbb{R}^{m+1}, |x_i - y_i| \leq \epsilon, i = 1, \ldots, m+1$ which results in the factor of $\sqrt{m+1}$. □

B.3 A Jackson-type bound for approximation with a split attention head

Lemma B.3.1. *Let $a, b : \mathbb{R}^d \to \mathbb{R}$, $c, d : \mathbb{R} \to \mathbb{R}$, $c(x), d(x) \neq 0, \forall x \in \mathbb{R}$ and $\epsilon_1, \epsilon_2 \geq 0$ be such that:*

$$\sup_{\boldsymbol{y} \in \mathbb{R}^d} \|a(\boldsymbol{y}) - b(\boldsymbol{y})\|_2 \leq \epsilon_1$$
$$\sup_{x \in \mathbb{R}} |c(x) - d(x)| = |c - d|_\infty \leq \epsilon_2.$$

Then for all $x \in \mathbb{R}$ and $\boldsymbol{y} \in \mathbb{R}^d$:

$$\left\| \frac{a(\boldsymbol{y})}{c(x)} - \frac{b(\boldsymbol{y})}{d(x)} \right\|_2 \leq \frac{\epsilon_1 |c|_\infty + \epsilon_2 \sup_{\boldsymbol{y} \in \mathbb{R}^d} \|a(\boldsymbol{y})\|_2}{|c(x)\, d(x)|}.$$

Proof. For a fixed $x \in \mathbb{R}$ and $\boldsymbol{y} \in \mathbb{R}^d$, using the triangle inequality gives us

$$\begin{aligned}
\left\| \frac{a(\boldsymbol{y})}{c(x)} - \frac{b(\boldsymbol{y})}{d(x)} \right\|_2 &= \left\| \frac{a(\boldsymbol{y})\, d(x) - b(\boldsymbol{y})\, c(x)}{c(x)\, d(x)} \right\|_2 \\
&= \frac{\left\| a(\boldsymbol{y})\, d(x) - b(\boldsymbol{y})\, c(x) \right\|_2}{|c(x)\, d(x)|} \\
&= \frac{\left\| a(\boldsymbol{y})\, d(x) - a(\boldsymbol{y})\, c(x) + a(\boldsymbol{y})\, c(x) - b(\boldsymbol{y})\, c(x) \right\|_2}{|c(x)\, d(x)|} \\
&\leq \frac{\left\| a(\boldsymbol{y})\, (d(x) - c(x)) \right\|_2 + \left\| c(x)\, (a(\boldsymbol{y}) - b(\boldsymbol{y})) \right\|_2}{|c(x)\, d(x)|} \\
&\leq \frac{\epsilon_2 \|a(\boldsymbol{y})\|_2 + \epsilon_1 |c(x)|}{|c(x)\, d(x)|} \\
&\leq \frac{\epsilon_1 |c|_\infty + \epsilon_2 \sup_{\boldsymbol{y} \in \mathbb{R}^d} \|a(\boldsymbol{y})\|_2}{|c(x)\, d(x)|}.
\end{aligned}$$

And as this holds for all $x \in \mathbb{R}$ and $\boldsymbol{y} \in \mathbb{R}^d$, the inequality in the lemma follows. □

Lemma B.3.2. *Let $f : S^m \to \mathbb{R}^{m+1}$, $m \geq 8$ satisfy the requirements in Corollary B.2.10. Then, given an $\epsilon > 0$ and taking λ, N, and $\boldsymbol{b}_1, \ldots, \boldsymbol{b}_N \in S^m$ as prescribed by the Corollary, we have that $\sum_{k=1}^N \exp(\lambda \langle \boldsymbol{x}, \boldsymbol{b}_k \rangle)$ is close to being a constant:*

$$\sup_{\boldsymbol{x} \in S^m} \left| 1 - \frac{c_{m+1}(\lambda)}{N} \sum_{k=1}^{N} \exp(\lambda \langle \boldsymbol{x}, \boldsymbol{b}_k \rangle) \right| \leq \frac{\epsilon}{2\sqrt{m+1}(L + \lambda \|f\|_\infty)}. \tag{B.22}$$

Proof. We can use Lemma B.2.7 by taking $g(\boldsymbol{x}, \boldsymbol{y}) = K_\lambda^{\text{vMF}}(\langle \boldsymbol{x}, \boldsymbol{y} \rangle)$. From Lemma B.2.8 we have that the modulus of continuity of K_λ^{vMF} is $\omega(K_\lambda^{\text{vMF}}; t) \leq t\, c_{m+1}(\lambda) \exp(\lambda)$. Observe that

using Equation (B.5) we have

$$\int_{S^m} g(x,y)dw_m(y) = \int_{S^m} K_\lambda^{\mathrm{vMF}}(\langle x,y\rangle)dw_m(y) = w_{m-1}\int_{-1}^{1} K_\lambda^{\mathrm{vMF}}(t)(1-t^2)^{(m-2)/2}dt = w_m.$$

The value for δ has to be selected as in Corollary B.2.10 (Equation (B.19)):

$$\delta = \left(\frac{2\epsilon}{6\pi\sqrt{m+1}(L+\lambda\|f\|_\infty)c_{m+1}(\lambda)\exp(\lambda)}\right)^2 < 1\text{-}\cos\left(\frac{\epsilon}{6\sqrt{m+1}(L+\lambda\|f\|_\infty)c_{m+1}(\lambda)\exp(\lambda)}\right).$$

Now, using the same partition from Lemma B.2.7, and recalling that we constructed it such that each element of the partition has the same measure $w_m(V_1) = w_m(V_i), \forall i$, we have:

$$\begin{aligned}
&\max_{x\in S^m}\left|\frac{1}{w_m}\int_{S^m} g(\boldsymbol{x},\boldsymbol{y})dw_m(\boldsymbol{y}) - \frac{1}{w_m}\sum_{k=1}^{N} K_\lambda^{\mathrm{vMF}}(\langle \boldsymbol{x},\boldsymbol{b}_k\rangle)w_m(V_k)\right|\\
&= \max_{x\in S^m}\left|1 - \frac{1}{w_m}\sum_{k=1}^{N} c_{m+1}(\lambda)\exp(\lambda\langle \boldsymbol{x},\boldsymbol{b}_k\rangle)w_m(V_k)\right|\\
&= \max_{x\in S^m}\left|1 - \frac{c_{m+1}(\lambda)}{N}\sum_{k=1}^{N}\exp(\lambda\langle \boldsymbol{x},\boldsymbol{b}_k\rangle)\right|\\
&\le 3c_{m+1}\exp(\lambda)\cos^{-1}(1-\delta)\\
&< 3c_{m+1}\exp(\lambda)\cos^{-1}\left(\cos\left(\frac{\epsilon}{6\sqrt{m+1}(L+\lambda\|f\|_\infty)c_{m+1}(\lambda)\exp(\lambda)}\right)\right)\\
&= \frac{3c_{m+1}\exp(\lambda)\epsilon}{6\sqrt{m+1}(L+\lambda\|f\|_\infty)c_{m+1}(\lambda)\exp(\lambda)}\\
&= \frac{\epsilon}{2\sqrt{m+1}(L+\lambda\|f\|_\infty)}.
\end{aligned}$$

□

Theorem B.3.3. *Let $f : S^m \to \mathbb{R}^{m+1}$, $m \ge 8$ satisfies the conditions in Corollary B.2.10. Define $\|f\|_\infty = \max_{1\le i\le m+1}\|f_i\|_\infty$. For any $\epsilon > 0$, there exist $\boldsymbol{b}_1,\dots,\boldsymbol{b}_N \in S^m$ such that f can be uniformly approximated to an error at most ϵ:*

$$\sup_{x\in S^m}\left\|f(\boldsymbol{x}) - \frac{\sum_{k=1}^{N}\xi_k\exp(\lambda\langle \boldsymbol{x},\boldsymbol{b}_k\rangle)}{\sum_{k=1}^{N}\exp(\lambda\langle \boldsymbol{x},\boldsymbol{b}_k\rangle)}\right\|_2 \le \epsilon,$$

with:

$$\lambda = \Lambda\left(\frac{2\epsilon L}{2L+\|f\|_\infty}\right) \quad \textit{with } \Lambda \textit{ from Equation (B.12)},$$

$$N \geq \Phi(m) \left(\frac{3\pi(L + \lambda\|f\|_\infty)\sqrt{m+1}\, c_{m+1}(\lambda) \exp(\lambda)}{\epsilon} \right)^{2(m+1)},$$
$$\xi_k = f(\boldsymbol{b}_k),\ \forall k = 1, \ldots, N.$$

Proof. From Corollary B.2.10 we know that $\sum_{k=1}^N \xi_k \exp(\lambda\langle \boldsymbol{x}, \boldsymbol{b}_k\rangle)$ approximates $f(\boldsymbol{x})$ and from Lemma B.3.2 we know that $\frac{c_{m+1}(\lambda)}{N} \sum_{k=1}^N \exp(\lambda\langle \boldsymbol{x}, \boldsymbol{b}_k\rangle)$ approximates 1. Using Lemma B.3.1 we can combine the two results to bound how well

$$\frac{\sum_{k=1}^N \xi_k \exp(\lambda\langle \boldsymbol{x}, \boldsymbol{b}_k\rangle)}{\frac{c_{m+1}(\lambda)}{N} \sum_{k=1}^N \exp(\lambda\langle \boldsymbol{x}, \boldsymbol{b}_k\rangle)}$$

approximates $f(\boldsymbol{x})/1 = f(\boldsymbol{x})$. The fact that $\frac{c_{m+1}(\lambda)}{N} \sum_{k=1}^N \exp(\lambda\langle \boldsymbol{x}, \boldsymbol{b}_k\rangle)$ is not identically 1 means that we will need to increase the precision of approximating the numerator by reducing ϵ in order to account for the additional error coming from the denominator. In particular, we have

$$\begin{aligned}
\sup_{\boldsymbol{x}\in S^m} \left\| f(\boldsymbol{x}) - \sum_{k=1}^N \xi_k \exp(\lambda\langle \boldsymbol{x}, \boldsymbol{b}_k\rangle) \right\|_2 &\leq \epsilon', \\
\sup_{\boldsymbol{x}\in S^m} \left| 1 - \frac{c_{m+1}(\lambda)}{N} \sum_{k=1}^N \exp(\lambda\langle \boldsymbol{x}, \boldsymbol{b}_k\rangle) \right| &\leq \frac{\epsilon'}{2\sqrt{m+1}(L + \lambda\|f\|_\infty)}, \\
\sup_{\boldsymbol{x}\in S^m} \|f(\boldsymbol{x})\|_2 &\leq \sqrt{m+1}\|f\|_\infty, \\
\sup_{\boldsymbol{x}\in S^m} \left| \frac{c_{m+1}(\lambda)}{N} \sum_{k=1}^N \exp(\lambda\langle \boldsymbol{x}, \boldsymbol{b}_k\rangle) \right| &\leq 1 + \frac{\epsilon'}{2\sqrt{m+1}(L + \lambda\|f\|_\infty)}.
\end{aligned}$$

Hence, applying Lemma B.3.1 gives us:

$$\begin{aligned}
\left\| f(\boldsymbol{x}) - \frac{\sum_{k=1}^N \xi_k \exp(\lambda\langle \boldsymbol{x}, \boldsymbol{b}_k\rangle)}{\frac{c_{m+1}(\lambda)}{N} \sum_{k=1}^N \exp(\lambda\langle \boldsymbol{x}, \boldsymbol{b}_k\rangle)} \right\|_2 &= \left\| \frac{f(\boldsymbol{x})}{1} - \frac{\sum_{k=1}^N \xi_k \exp(\lambda\langle \boldsymbol{x}, \boldsymbol{b}_k\rangle)}{\frac{c_{m+1}(\lambda)}{N} \sum_{k=1}^N \exp(\lambda\langle \boldsymbol{x}, \boldsymbol{b}_k\rangle)} \right\|_2 \\
&\leq \frac{\epsilon' + \frac{\epsilon'}{2\sqrt{m+1}(L+\lambda\|f\|_\infty)} \sqrt{m+1}\|f\|_\infty}{1 + \frac{\epsilon'}{2\sqrt{m+1}(L+\lambda\|f\|_\infty)}} \\
&= \frac{\epsilon'\sqrt{m+1}(\|f\|_\infty + 2L + 2\lambda\|f\|_\infty)}{2\sqrt{m+1}(\lambda\|f\|_\infty + L) + \epsilon'} \\
&\leq \frac{\epsilon'}{2}\left(\frac{\|f\|_\infty}{\lambda\|f\|_\infty + L} + 2\frac{\lambda\|f\|_\infty + L}{\lambda\|f\|_\infty + L} \right) \\
&\leq \frac{\epsilon'}{2}\left(2 + \frac{\|f\|_\infty}{L} \right),
\end{aligned}$$

where we used that $\epsilon', \|f\|_\infty L > 0$, which is the case in realistic scenarios. Therefore, if we want this error to be upper bounded by ϵ, we need to select

$$\epsilon' \leq \frac{2\epsilon L}{2L + \|f\|_\infty}.$$

From Corollary B.2.10 (Equation (B.12)) that can be achieved by selecting

$$\lambda = \Lambda\left(\frac{2\epsilon L}{2L + \|f\|_\infty}\right)$$

and

$$N \geq \Phi(m)\left(\frac{3\pi(L + \lambda\|f\|_\infty)\sqrt{m+1}\, c_{m+1}(\lambda)\exp(\lambda)}{\epsilon}\right)^{2(m+1)}.$$

Finally, observe that the $c_{m+1}(\lambda)/N$ factor can be folded in the $\boldsymbol{\xi}_k$ terms (Equation (B.20)):

$$\boldsymbol{\xi}_k = \frac{N}{c_{m+1}(\lambda)} f(\boldsymbol{b}_k)\, c_{m+1}(\lambda) \frac{w_m(V_k)}{w_m} = f(\boldsymbol{b}_k),$$

with $\boldsymbol{\xi}_k$ nicely reducing to be the evaluation of f at the corresponding control point $\boldsymbol{b}_k$. □

B.4 Additional results

Lemma B.4.1. *Define the stereographic projection and its inverse:*

$$\begin{aligned}\Sigma_m : \bar{S}^m &\to \mathbb{R}^m \\ (x_1, \ldots, x_{m+1}) &\mapsto \left(\frac{x_1}{1 - x_{m+1}}, \ldots, \frac{x_m}{1 - x_{m+1}}\right)\end{aligned}$$

$$\begin{aligned}\Sigma_m^{-1} : \mathbb{R}^m &\to S^m \\ (y_1, \ldots, y_m) &\mapsto \left(\frac{2y_1}{\sum_{i=1}^m y_i^2 + 1}, \ldots, \frac{2y_m}{\sum_{i=1}^m y_i^2 + 1}, \frac{\sum_{i=1}^m y_i^2 - 1}{\sum_{i=1}^m y_i^2 + 1}\right)\end{aligned}$$

with $\bar{S}^m$ the part of S^m that gets mapped to $[0,1]^m$, i.e., $\bar{S}^m = \Sigma_m^{-1}([0,1]^m)$. Σ_m and Σ_m^{-1} are continuous and inverses of each other and there exist L_m^Σ and $L_m^{\Sigma^{-1}}$ such that $\omega(\Sigma_m; t) \leq L_m^\Sigma t$ and $\omega(\Sigma_m^{-1}; t) \leq L_m^{\Sigma^{-1}} t$. Furthermore, $\Sigma_m \circ \mathcal{H}^1_{H,3(m+1)} \circ \Sigma_m^{-1}$ is dense in the set of continuous functions $[0,1]^m \to \mathbb{R}^m$.

Appendix C

Appendices for Universal In-Context Approximation by Prompting Fully Recurrent Models

C.1 Computation graph debranching rules

We convert the computation DAG resulting from the LSRL program into a path program by attending to the first node whose output is the input for multiple other nodes, i.e., the first branching node.

Preparation step. Before we even start debranching we first pre-process the graph by fusing consecutive nodes of the same type together. The specific rules are:

- If a `Lin` node is followed by a single other `Lin` node, then fuse them together. This follows directly from the classical result that composing linear functions is a linear function.
- If a `ReLU` node is followed by another `ReLU` node, we can drop one of them as `ReLU` is idempotent.
- If a `Lin` is followed by a `LinState`, we can subsume the weight matrix $\boldsymbol{A}$ of the linear node in the $\boldsymbol{B}$ matrix of the `LinState`, and the bias $\boldsymbol{b}$ of the `Lin` node in the bias $\boldsymbol{b}$ of the `LinState`.
- If all inputs of a `Concat` node are the same, then this node only duplicates the input and hence can be safely replaced with a `Lin` layer.

This step is repeated after the any debranching rule is applied.

The debranching process goes through the following cases in order. It iterates over the rules until there are no branching nodes left, in other words, until the graph has become a path graph. We will refer to the nodes whose input is the branching node as *subsequent nodes*.

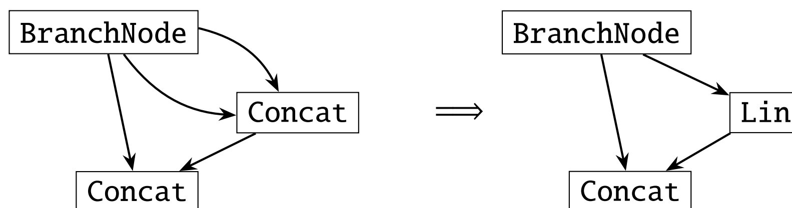
Case 1A: All subsequent nodes are `Multi`. As all `Multi` nodes that have the same input (the branching node) they must all be producing the exact same output. Hence, only one can be kept. This removes one branch.

Case 1B: The subsequent nodes are a combination of `Multi` and other nodes. We add a single `Lin` layer that acts as a bypass for the non-`Multi` nodes using the fact that multiplication by 1 is identity. This is followed by a single `Multi` layer. We then add `Slice` operators between the new `Lin` layer and the non-`Multi` nodes. This keeps the number of branches unchanged but removes the `Multi` node and the new branch can be handled by the other rules.

Case 2: All subsequent nodes are `LinState`. `LinState` nodes can be fused into a single `LinState` node by combining their states and update matrices. As each `LinState` may have different subsequent nodes itself, we add `Slice` nodes to extract the respective subspaces of the state. This keeps the number of branches unchanged but puts the graph into Case 5A.

Case 3: All subsequent nodes are `ReLU`. We can replace them by a single `ReLU` node. This removes one branch.

Case 4: All subsequent nodes are `Concat`. One complication is that `Concat` nodes can depend on other `Concat` nodes. So, we will restrict ourselves at this step by only treating the `Concat` nodes that depend only on the branch node directly by replacing them with a single `Lin` node. The rest will be handled by the `Lin` and `Concat` case (Case 10) or the only `Lin` case (Cases 5A and 5B). See the following example:



Hence, this operation either reduces the number of branches by one or will be followed by a case that reduces the number of branches.

Case 5A: Only `Lin` nodes and they are all `Slice`s. This is one of the more challenging cases. While the `Slice` nodes are simply `Lin` nodes with special structure, we cannot treat them like standard `Lin` nodes (see Case 5B). While we can merge them into a single `Lin` node, we will then need further `Slice`s to extract the relevant subspaces for the subsequent nodes. Therefore, we would be simply replacing `Slice` nodes with `Slice` nodes. Instead, we use the observation that `Slice` nodes can be fused with subsequent `Lin` and `LinState` nodes and can be pushed after `ReLU` and `Concat` nodes. Therefore we treat each subsequent node differently, depending on its type:

- If there are `Multi` nodes after any of the `Slice` nodes, they can all be fused into a single `Lin` node followed by a single `Multi` node.
- If there are `Lin` or `LinState` nodes after any of the `Slice` nodes, the `Slice`s can be fused with the $\boldsymbol{A}$ matrix of the `Lin` nodes and the $\boldsymbol{B}$ matrix of the `LinState` nodes. This uses the fact that composing linear functions results in a linear function.
- If there is a `ReLU` after a `Slice` node, their position can be switched without changing the nodes. That is because `ReLU` commutes with linear operations with $\boldsymbol{b} = \boldsymbol{0}$ and $\boldsymbol{A}$ with non-negative eigenvalues as is the case for `Slice` nodes.
- If there is a `Concat` node after a `Slice` node, we can similarly push the `Slice` as a new `Lin` node after the `Concat`.

This step does not reduce the number of branching nodes but prepares the graph for a removal, with the specific case depending on the remaining nodes.

Case 5B: Only `Lin` nodes and they are not all `Slice`s. We can combine them into a single `Lin` node and then add `Slice`s to extract the relevant subspaces for the subsequent nodes. These `Slice`s can then be pushed into the next operations using Case 5A.

Case 6: Both `LinState` nodes and other nodes. If both `LinState` nodes and other nodes are present, we can pass through the other variables with dummy `LinState` variables using zero matrices for $\boldsymbol{A}$ and identities for $\boldsymbol{B}$. Then, Case 2 can be used to fuse all the `LinState` variables together.

Case 7A: Only `Lin` and `ReLU` nodes where all `Lin` nodes are followed by only one node which is a `ReLU`. If we add `Lin` bypasses to the `ReLU`s we will have only `Lin` nodes left. Each one of them would be followed by a `ReLU`. Hence, Case 5B can be first applied, followed by Case 3.

Case 7B: Only `Lin` and `ReLU` nodes where some `Lin` nodes are not followed by only one node which is a `ReLU`. In this case we cannot apply the above strategy. Instead, we fuse the `ReLU`s by placing `ReLU`-based bypasses before the `Lin` nodes. We do this in a similar

spirit to Equation (5.11), by splitting the positive and negative components and treating them separately. See Appendix C.6.6 for the LSRL implementation. Our DAG will then be in Case 7A first, then Case 5B, and, finally, in Case 3.

Case 8: Only `Lin` and `Concat` nodes. We add `Lin` bypasses for the `Concat` nodes which can then be merged using Case 5B and then Case 5A.

Case 9: Only `ReLU` and `Concat` nodes. Same strategy as for Case 8 but with `ReLU` bypasses.

Case 10: Only `Lin`, `ReLU` or `Concat` nodes. We introduce `ReLU` bypasses to all `Concat` nodes and to the `Lin` branches which are not immediately followed by a `ReLU`. This will be followed by applying Case 5B and then Case 3.

The above 13 cases cover all possible branching configurations. After repeated application, they reduce any DAG corresponding to an LSRL program to a path graph that can be compiled to one of the recurrent models in Section 5.2.

C.2 Error bound on the approximation scheme for continuous functions

In Section 5.4.1 we outlined a strategy to perform universal in-context approximation for continuous functions with Linear RNNs. The full program is in Listing 1 and an illustration of the scheme is presented in Figure 5.2. In Section 5.4.1 we claimed that the prompt length required to approximate an L-Lipschitz function f (w.r.t. the ℓ_2 norm) to precision ϵ is $N = (2\epsilon/L\sqrt{d_{\text{in}}})^{-d_{\text{in}}} = \mathcal{O}(\epsilon^{-d_{\text{in}}})$. The present appendix offers the formal proof of this claim.

The program in Listing 1 approximates the value of a function $\boldsymbol{y} = f(\boldsymbol{q})$ with the value $\bar{\boldsymbol{y}}$ at the centre $\boldsymbol{c}$ of the cell that contains $\boldsymbol{q}$. Therefore, the error of our approximation is the maximum difference between $f(\boldsymbol{q})$ and $f(\boldsymbol{c})$: $\|f(\boldsymbol{q}) - f(\boldsymbol{c})\|_2$. First, as the length of each side of the cell is δ, that means that $\|\boldsymbol{q} - \boldsymbol{c}\|_\infty \leq \delta/2$. Thus, $\|\boldsymbol{q} - \boldsymbol{c}\|_2 \leq \sqrt{d_{\text{in}}}\delta/2$. Therefore, thanks to f being L-Lipschitz we get:

$$\|f(\boldsymbol{q}) - f(\boldsymbol{c})\|_2 \leq \frac{\delta L \sqrt{d_{\text{in}}}}{2}.$$

If we want to upper bound this approximation error by ϵ, we need to have δ small enough:

$$\delta \leq \frac{2\epsilon}{L\sqrt{d_{\text{in}}}}.$$

Finally, as the number of cells we need to cover the whole domain is $N = (1/\delta)^{d_{\text{in}}}$, this corresponds to us needing sufficiently long prompt:

$$N \geq \left(\frac{1}{\delta}\right)^{d_{\text{in}}} \geq \left(\frac{L\sqrt{d_{\text{in}}}}{2\epsilon}\right)^{d_{\text{in}}}.$$

Therefore, if we want our approximation to have error at most ϵ anywhere in the domain, we need a prompt of length at least $(L\sqrt{d_{\text{in}}}/2\epsilon)^{d_{\text{in}}}$.

C.3 Gated RNNs are GRU models

A GRU layer (Cho et al., 2014) with input $\boldsymbol{a}_t \in \mathbb{R}^{d_{\text{in}}}$ and hidden state $\boldsymbol{h}_{t-1} \in \mathbb{R}^{d_{\texttt{hidden}}}$, and output $\boldsymbol{h}_t \in \mathbb{R}^{d_{\texttt{hidden}}}$ can be described as follows:

$$\boldsymbol{z}_t = \texttt{Sigmoid}(\boldsymbol{W}_z \boldsymbol{a}_t + \boldsymbol{U}_z \boldsymbol{h}_{t-1} + \boldsymbol{b}_z), \quad \text{(update gate vector)} \tag{C.1}$$

$$\boldsymbol{r}_t = \texttt{Sigmoid}(\boldsymbol{W}_r \boldsymbol{a}_t + \boldsymbol{U}_r \boldsymbol{h}_{t-1} + \boldsymbol{b}_r), \quad \text{(reset gate vector)} \tag{C.2}$$

$$\hat{\boldsymbol{h}}_t = \texttt{tanh}(\boldsymbol{W}_h \boldsymbol{a}_t + \boldsymbol{U}_h(\boldsymbol{r}_t \odot \boldsymbol{h}_{t-1}) + \boldsymbol{b}_h), \quad \text{(candidate activation vector)} \tag{C.3}$$

$$\boldsymbol{h}_t = (1 - \boldsymbol{z}_t) \odot \boldsymbol{h}_{t-1} + \boldsymbol{z}_t \odot \hat{\boldsymbol{h}}_t, \quad \text{(output vector)} \tag{C.4}$$

In this section, we show a conversion of a single Gated RNN layer (Equation (5.5)) to $k+2$ GRU layers. Here, k is the number of layers in the γ and h MLPs in Equation (5.5). We first show that a single GRU layer can be used to compute the updated state $\boldsymbol{s}_t$ and the output of the first layer of γ when applied to $\boldsymbol{x}_t$. Then, every pair of single layers of $\gamma(\boldsymbol{x}_t)$ and $\phi(\boldsymbol{s}_t)$ can be represented as an individual GRU layer. Finally, a single layer can be used to compute the element-wise multiplication $\gamma(\boldsymbol{x}_t) \odot \phi(\boldsymbol{s}_t)$. For simplicity, we assume the `Sigmoid` and `tanh` nonlinearities are replaced by `ReLU`s. If not, they can each be approximated with MLPs and hence also with additional GRU layers. Additionally, for convenience we will assume $d_{\texttt{in}} = d_{\texttt{hidden}}$.

C.3.1 Representing the state update as a GRU layer

For this layer we set $\boldsymbol{b}_z = \boldsymbol{1}$, $\boldsymbol{W}_z = \boldsymbol{0}$, $\boldsymbol{U}_z = \boldsymbol{0}$ giving $\boldsymbol{z}_t = \boldsymbol{1}$. Similarly, we set $\boldsymbol{b}_r = \boldsymbol{1}$, $\boldsymbol{W}_r = \boldsymbol{0}$, $\boldsymbol{U}_r = \boldsymbol{0}$ giving $\boldsymbol{r}_t = \boldsymbol{1}$. Thus, Equation (C.3) reduces to:

$$\hat{\boldsymbol{h}}_t = \sigma(\boldsymbol{W}_h \boldsymbol{a}_t + \boldsymbol{U}_h \boldsymbol{h}_{t-1} + \boldsymbol{b}_h), \tag{C.5}$$

Setting $\boldsymbol{a}_t = \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{x}_t \end{bmatrix}$, where $\boldsymbol{x}_t \in \mathbb{R}^{d_{\text{in}}/2}$, $\boldsymbol{h}_{t-1} = \begin{bmatrix} \boldsymbol{s}_{t-1} \\ \boldsymbol{0} \end{bmatrix}$, where $\boldsymbol{s}_{t-1} \in \mathbb{R}^{d_{\texttt{hidden}}/2}$, $\boldsymbol{W}_h = \begin{bmatrix} \boldsymbol{0} & \boldsymbol{B} \\ \boldsymbol{0} & \boldsymbol{I} \end{bmatrix}$, $\boldsymbol{U}_h = \begin{bmatrix} \boldsymbol{A} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} \end{bmatrix}$, $\boldsymbol{b}_h = \begin{bmatrix} \boldsymbol{b} \\ -\boldsymbol{k}_{lb} \end{bmatrix}$, where $\boldsymbol{k}_{lb}$ is a vector where every element in $\boldsymbol{k}$ is a lower bound on $\boldsymbol{x}_t$. results in Equation (C.4) becoming:

$$\boldsymbol{h}_t = \sigma\left(\begin{bmatrix} \boldsymbol{0} & \boldsymbol{B} \\ \boldsymbol{0} & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{x}_t \end{bmatrix} + \begin{bmatrix} \boldsymbol{A} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} \end{bmatrix} \begin{bmatrix} \boldsymbol{s}_{t-1} \\ \boldsymbol{0} \end{bmatrix} + \begin{bmatrix} \boldsymbol{b} \\ -\boldsymbol{k}_{lb} \end{bmatrix} \right) = \begin{bmatrix} \sigma(\boldsymbol{A}\boldsymbol{s}_{t-1} + \boldsymbol{B}\boldsymbol{x}_t + \boldsymbol{b}) \\ \sigma(\boldsymbol{x}_t - \boldsymbol{k}_{lb}) \end{bmatrix} = \begin{bmatrix} \sigma(\boldsymbol{s}_t) \\ \boldsymbol{x}_t - \boldsymbol{k}_{lb} \end{bmatrix}. \tag{C.6}$$

Note: if we do not want to assume a compact domain for $\boldsymbol{x}_t$, it would be possible to use the same trick as in Equation (5.11) rather than subtracting $\boldsymbol{k}$ in this layer and adding in the next. However, we omit this approach for clarity of presentation.

C.3.2 Representing each MLP layer as a GRU layer

In these layers, similarly to the recurrent layer, we set $\boldsymbol{b}_z = \mathbf{1}$, $\boldsymbol{W}_z = \mathbf{0}$, $\boldsymbol{U}_z = \mathbf{0}$ giving $\boldsymbol{z}_t = \mathbf{1}$. In the same way, we set $\boldsymbol{b}_r = \mathbf{1}$, $\boldsymbol{W}_r = \mathbf{0}$, $\boldsymbol{U}_r = \mathbf{0}$ giving $\boldsymbol{r}_t = \mathbf{1}$. Here, however, we set $\boldsymbol{W}_h = \begin{bmatrix} \boldsymbol{W}_{h_i} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{W}_{\gamma_i} \end{bmatrix}$, $\boldsymbol{U}_h = \mathbf{0}$ and $\boldsymbol{b}_h = \begin{bmatrix} \boldsymbol{b}_{h_i} \\ \boldsymbol{b}_{\gamma_i} \end{bmatrix}$, except for the first of such layer where $\boldsymbol{b}_h = \begin{bmatrix} \boldsymbol{b}_{h_i} \\ \boldsymbol{b}_{\gamma_i} + \boldsymbol{W}_{\gamma_i} \boldsymbol{k}_{lb} \end{bmatrix}$. Thus, for an input $\boldsymbol{a}_t = \begin{bmatrix} \boldsymbol{a}_{1,t} \\ \boldsymbol{a}_{2,t} \end{bmatrix}$ the layer output (Equation (C.4)) for layer i is:

$$\boldsymbol{h}_t = \sigma\left(\begin{bmatrix} \boldsymbol{W}_{\phi_i} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{W}_{\gamma_i} \end{bmatrix} \begin{bmatrix} \boldsymbol{a}_{1,t} \\ \boldsymbol{a}_{2,t} \end{bmatrix} + \begin{bmatrix} \boldsymbol{b}_{\phi_i} \\ \boldsymbol{b}_{\gamma_i} \end{bmatrix} \right) = \begin{bmatrix} \phi_i(\boldsymbol{a}_{1,t}) \\ \gamma_i(\boldsymbol{a}_{1,t}) \end{bmatrix}. \tag{C.7}$$

Here, ϕ_i and γ_i are the i-th layers (including the `ReLU`) of respectively ϕ and γ in Equation (5.5).

C.3.3 Representing the multiplicative gating with a single GRU layer

The only thing left is to model the element-wise multiplication of the outputs of ϕ and γ in Equation (5.5). We do this using a GRU layer with $\boldsymbol{b}_z = \mathbf{0}$, $\boldsymbol{W}_z = \mathbf{0}$, $\boldsymbol{U}_z = \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{I} \end{bmatrix}$. We set $\boldsymbol{b}_r = \mathbf{0}$, $\boldsymbol{W}_r = \mathbf{0}$, $\boldsymbol{U}_r = \mathbf{0}$ giving $\boldsymbol{r}_t = \mathbf{0}$. We also set $\boldsymbol{b}_h = \mathbf{0}$, $\boldsymbol{W}_h = \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \boldsymbol{I} & \mathbf{0} \end{bmatrix}$, $\boldsymbol{U}_h = \mathbf{0}$. Thus, for an input $\boldsymbol{a}_t = \begin{bmatrix} \boldsymbol{a}_{1,t} \\ \boldsymbol{a}_{2,t} \end{bmatrix}$, the output $\boldsymbol{h}_t$ (Equation (C.4)) of this GRU layer becomes:

$$\boldsymbol{h}_t = \sigma\left(\begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \boldsymbol{I} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \boldsymbol{a}_{1,t} \\ \boldsymbol{a}_{2,t} \end{bmatrix} \right) \odot \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \boldsymbol{a}_{1,t} \\ \boldsymbol{a}_{2,t} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \sigma(\boldsymbol{a}_{1,t}) \odot \boldsymbol{a}_{2,t} \end{bmatrix}. \tag{C.8}$$

If $\boldsymbol{a}_t$ is the output of a GRU layer constructed as in Equation (C.7) (as is in our case), then it must be non-negative. This is due to the `ReLU` application in Equation (C.7). Hence, the application of another `ReLU` to $\boldsymbol{a}_{1,t}$ in Equation (C.8) can be safely removed as `ReLU` is idempotent and Equation (C.8) simplifies to

$$\boldsymbol{h}_t = \begin{bmatrix} \mathbf{0} \\ \boldsymbol{a}_{1,t} \odot \boldsymbol{a}_{2,t} \end{bmatrix}. \tag{C.9}$$

Thus, this construction computes element-wise multiplication of $\boldsymbol{a}_{1,t}$ and $\boldsymbol{a}_{2,t}$.

C.3.4 Composing the operations to model a single Gated RNN layer

In order to represent Equation (5.5), we use one GRU layer for the recurrence (as described in Appendix C.3.1), followed by k GRU layers modelling a pair of the k MLP layers of ϕ and γ (Appendix C.3.2), completed with a single mixing layer (Appendix C.3.3). This stack of

$k+2$ layers models exactly the Gated RNN layer (Equation (5.5)):

$$\begin{aligned} \boldsymbol{s}_t &= \sigma\left(\boldsymbol{A}\begin{bmatrix}\boldsymbol{0}\\ \boldsymbol{s}_{t-1}\end{bmatrix} + \boldsymbol{B}\begin{bmatrix}\boldsymbol{x}_t\\ \boldsymbol{0}\end{bmatrix} + \boldsymbol{b}\right)\\ \boldsymbol{y}_t &= \begin{bmatrix}\boldsymbol{0}\\ \gamma(\boldsymbol{x}_t)\odot\phi(\boldsymbol{s}_t)\end{bmatrix}, \end{aligned}$$

With this, we have shown that any Gated RNN (Equation (5.5)) can be expressed as a GRU-based model. Hence, the two universal approximation programs in Listings 1 and 2 can be implemented also in GRU-based models. Thus, the GRU architecture can also be a universal in-context approximator.

C.4 Gated RNNs are LSTMs

A single LSTM layer (Hochreiter and Schmidhuber, 1997; Gers et al., 2000) with input $\boldsymbol{a}_t \in \mathbb{R}^{d_{\mathtt{in}}}$, hidden state $\boldsymbol{h}_{t-1} \in \mathbb{R}^{d_{\mathtt{hidden}}}$, candidate memory cell $\tilde{\boldsymbol{c}}_t \in \mathbb{R}^{d_{\mathtt{hidden}}}$, memory cell $\boldsymbol{c}_t \in \mathbb{R}^{d_{\mathtt{hidden}}}$ and layer output $\boldsymbol{h}_t \in \mathbb{R}^{d_{\mathtt{hidden}}}$ can be expressed as:

$$\begin{aligned} \boldsymbol{f}_t &= \mathtt{Sigmoid}(\boldsymbol{W}_f\boldsymbol{a}_t + \boldsymbol{U}_f\boldsymbol{h}_{t-1} + \boldsymbol{b}_f), && \text{(forget gate vector)} && \text{(C.10)}\\ \boldsymbol{i}_t &= \mathtt{Sigmoid}(\boldsymbol{W}_i\boldsymbol{a}_t + \boldsymbol{U}_i\boldsymbol{h}_{t-1} + \boldsymbol{b}_i), && \text{(input gate vector)} && \text{(C.11)}\\ \boldsymbol{o}_t &= \mathtt{Sigmoid}(\boldsymbol{W}_o\boldsymbol{a}_t + \boldsymbol{U}_o\boldsymbol{h}_{t-1} + \boldsymbol{b}_o), && \text{(output gate vector)} && \text{(C.12)}\\ \tilde{\boldsymbol{c}}_t &= \mathtt{tanh}(\boldsymbol{W}_c\boldsymbol{a}_t + \boldsymbol{U}_c\boldsymbol{h}_{t-1} + \boldsymbol{b}_c), && \text{(candidate cell vector)} && \text{(C.13)}\\ \boldsymbol{c}_t &= \boldsymbol{f}_t\odot\boldsymbol{c}_{t-1} + \boldsymbol{i}_t\odot\tilde{\boldsymbol{c}}_t, && \text{(memory cell vector)} && \text{(C.14)}\\ \boldsymbol{h}_t &= \boldsymbol{o}_t\odot\mathtt{tanh}(\boldsymbol{c}_t), && \text{(output vector)} && \text{(C.15)} \end{aligned}$$

where $\boldsymbol{h}_0 = \boldsymbol{0}$ and $\boldsymbol{c}_0 = \boldsymbol{0}$.

In a way analogous to Appendix C.3, we show that a single layer of a gated RNN (Equation (5.5)) can be expressed using $k+2$ LSTM layers, where k is the maximum depth of either of the MLP networks ϕ or γ. We again follow the setup of replacing all `Sigmoid` and `tanh` activation functions with `ReLU` activations which we denote σ and we again assume that $d_{\mathtt{in}} = d_{\mathtt{hidden}}$. The set up follows the same structure as in Appendix C.3. First, we show that the non-linear state update computing $\boldsymbol{s}_t$ can be expressed as a single LSTM layer. We then show that we can represent the layers in MLP networks $\gamma(\boldsymbol{x}_t)$ and $\phi(\boldsymbol{s}_t)$ using single LSTM layers. Finally, a single layer can compute the Hadamard product between $\gamma(\boldsymbol{x}_t)$ and $\phi(\boldsymbol{s}_t)$. Therefore, any Gated RNN with `ReLU` activations can be expressed as a LSTM with `ReLU` activations.

For clarity of the exposition, we once again assume that our inputs belong to a compact domain $\mathcal{X}$ of real vectors. This implies that the set is bounded and, in particular, that we can find a vector $\boldsymbol{k}_{lb}$ such that $\boldsymbol{k}_{lb,i} \leq (\boldsymbol{x}_t)_i$ for $i \in [d_{\mathtt{in}}]$ for all $\boldsymbol{x}_t \in \mathcal{X}$. In other words, we have $(\boldsymbol{x}_t - \boldsymbol{k}_{lb})_i \geq 0$ for for $i \in 1,\ldots,d_{\mathtt{in}}$. We will make use of this fact several times when dealing with `ReLU` activations.

C.4.1 Representing the state update as an LSTM layer

We first represent the non-linear state update in Eq. (5.5) using a single layer of an LSTM. In particular, we set $\boldsymbol{W}_f = \boldsymbol{0}$, $\boldsymbol{U}_f = \boldsymbol{0}$ and $\boldsymbol{b}_f = \boldsymbol{0}$ so that $\boldsymbol{f}_t = \boldsymbol{0}$. We also set $\boldsymbol{W}_i = \boldsymbol{0}$, $\boldsymbol{U}_i = \boldsymbol{0}$, $\boldsymbol{b}_i = \boldsymbol{1}$ and $\boldsymbol{W}_c = \boldsymbol{0}$, $\boldsymbol{U}_c = \boldsymbol{0}$, $\boldsymbol{b}_c = \boldsymbol{1}$. This results in $\boldsymbol{i}_t = \boldsymbol{1}$ and $\tilde{\boldsymbol{c}}_t = \boldsymbol{1}$. We see from this that the LSTM layer with these weight settings reduces to

$$\boldsymbol{h}_t = \boldsymbol{o}_t = \sigma(\boldsymbol{W}_o \boldsymbol{a}_t + \boldsymbol{U}_o \boldsymbol{h}_{t-1} + \boldsymbol{b}_o). \tag{C.16}$$

We now set $\boldsymbol{a}_t = \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{x}_t \end{bmatrix}$, where $\boldsymbol{x}_t \in \mathbb{R}^{d_{\text{in}}/2}$, $\boldsymbol{h}_{t-1} = \begin{bmatrix} \boldsymbol{s}_{t-1} \\ \boldsymbol{0} \end{bmatrix}$, where $\boldsymbol{s}_{t-1} \in \mathbb{R}^{d_{\texttt{hidden}}/2}$, $\boldsymbol{W}_o = \begin{bmatrix} \boldsymbol{0} & \boldsymbol{B} \\ \boldsymbol{0} & \boldsymbol{I} \end{bmatrix}$, $\boldsymbol{U}_o = \begin{bmatrix} \boldsymbol{A} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} \end{bmatrix}$, $\boldsymbol{b}_o = \begin{bmatrix} \boldsymbol{b} \\ -\boldsymbol{k}_{lb} \end{bmatrix}$ so that

$$\boldsymbol{h}_t = \sigma\left(\begin{bmatrix} \boldsymbol{0} & \boldsymbol{B} \\ \boldsymbol{0} & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{x}_t \end{bmatrix} + \begin{bmatrix} \boldsymbol{A} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} \end{bmatrix} \begin{bmatrix} \boldsymbol{s}_{t-1} \\ \boldsymbol{0} \end{bmatrix} + \begin{bmatrix} \boldsymbol{b} \\ -\boldsymbol{k}_{lb} \end{bmatrix} \right) = \begin{bmatrix} \sigma(\boldsymbol{A}\boldsymbol{s}_{t-1} + \boldsymbol{B}\boldsymbol{x}_t + \boldsymbol{b}) \\ \sigma(\boldsymbol{x}_t - \boldsymbol{k}_{lb}) \end{bmatrix} = \begin{bmatrix} \boldsymbol{s}_t \\ \boldsymbol{x}_t - \boldsymbol{k}_{lb} \end{bmatrix}. \tag{C.17}$$

C.4.2 Representing each MLP layer as an LSTM layer

Now we want to use an LSTM layers to model the MLP layers of both γ and ϕ simultaneously. We set $\boldsymbol{W}_f = \boldsymbol{0}$, $\boldsymbol{U}_f = \boldsymbol{0}$, $\boldsymbol{b}_f = \boldsymbol{0}$ and $\boldsymbol{W}_i = \boldsymbol{0}$, $\boldsymbol{U}_i = \boldsymbol{0}$, $\boldsymbol{b}_i = \boldsymbol{1}$ and $\boldsymbol{W}_c = \boldsymbol{0}$, $\boldsymbol{U}_c = \boldsymbol{0}$, $\boldsymbol{b}_c = \boldsymbol{1}$ as before. We make a change for these LSTM layers by setting $\boldsymbol{W}_o = \begin{bmatrix} \boldsymbol{W}_{\phi_i} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{W}_{\gamma_i} \end{bmatrix}$, $\boldsymbol{U}_o = \boldsymbol{0}$ and $\boldsymbol{b}_o = \begin{bmatrix} \boldsymbol{b}_{\phi_i} \\ \boldsymbol{b}_{\gamma_i} \end{bmatrix}$, except for the first layer where $\boldsymbol{b}_\phi = \begin{bmatrix} \boldsymbol{b}_{\phi_1} \\ \boldsymbol{b}_{\gamma_1} + \boldsymbol{W}_{\gamma_1}\boldsymbol{k} \end{bmatrix}$. Thus, for an input $\boldsymbol{a}_t = \begin{bmatrix} \boldsymbol{a}_{1,t} \\ \boldsymbol{a}_{2,t} \end{bmatrix}$ the layer output is:

$$\boldsymbol{h}_t = \sigma\left(\begin{bmatrix} \boldsymbol{W}_{\phi_i} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{W}_{\gamma_i} \end{bmatrix} \begin{bmatrix} \boldsymbol{a}_{1,t} \\ \boldsymbol{a}_{2,t} \end{bmatrix} + \begin{bmatrix} \boldsymbol{b}_{\phi_i} \\ \boldsymbol{b}_{\gamma_i} \end{bmatrix} \right) = \begin{bmatrix} \phi_i(\boldsymbol{a}_{1,t}) \\ \gamma_i(\boldsymbol{a}_{2,t}) \end{bmatrix}. \tag{C.18}$$

Here, ϕ_i and γ_i again refer to the i-th layers (including the `ReLU`) of respectively ϕ and γ in Equation (5.5).

Note that, without a loss of generality, if we have that ϕ has m layers whereas γ has k with $m < k$, then we can also model this by simply adding additional layers to model additional layers for γ whilst simply passing on ϕ unchanged. Specifically, we set set the weights to ensure that $\boldsymbol{f}_t = 0$ and that $\boldsymbol{i}_t$ and $\tilde{\boldsymbol{c}}_t$ are $\boldsymbol{1}$ so that $\boldsymbol{h}_t = \boldsymbol{o}_t$. The input to this layer for $i > k$ is then given as $\boldsymbol{a}_t = \begin{bmatrix} \phi(\boldsymbol{s}_t) \\ \boldsymbol{a}_{2,t} \end{bmatrix}$. The we set the weights to compute $\boldsymbol{o}_t$ as

$$\boldsymbol{o}_t = \sigma\left(\begin{bmatrix} \boldsymbol{I} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{W}_{\gamma_i} \end{bmatrix} \begin{bmatrix} \phi(\boldsymbol{s}_t) \\ \boldsymbol{a}_{2,t} \end{bmatrix} + \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{b}_{\phi_i} \end{bmatrix} \right) = \begin{bmatrix} \phi(\boldsymbol{s}_t) \\ \gamma_i(\boldsymbol{a}_{2,t}) \end{bmatrix}. \tag{C.19}$$

C.4.3 Representing the multiplicative gating with an LSTM layer

Finally, we model the element-wise multiplication of the outputs of ϕ and γ in Equation (5.5). To do this we set the weights of the input gate and candidate cell vectors for the final layers of of γ and ϕ to be as follows:

$$\boldsymbol{i}_t = \sigma\left(\begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \boldsymbol{I} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \boldsymbol{a}_{1,t} \\ \boldsymbol{a}_{2,t} \end{bmatrix} + \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \end{bmatrix}\right) = \begin{bmatrix} \mathbf{0} \\ \boldsymbol{a}_{1,t} \end{bmatrix} \tag{C.20}$$

and

$$\tilde{\boldsymbol{c}} = \sigma\left(\begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \boldsymbol{a}_{1,t} \\ \boldsymbol{a}_{2,t} \end{bmatrix} + \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \end{bmatrix}\right) = \begin{bmatrix} \mathbf{0} \\ \boldsymbol{a}_{2,t} \end{bmatrix}. \tag{C.21}$$

Then by setting $\boldsymbol{W}_f = \mathbf{0}$, $\boldsymbol{U}_f = \mathbf{0}$, $\boldsymbol{b}_f = \mathbf{0}$ and $\boldsymbol{W}_o = \mathbf{0}$, $\boldsymbol{U}_o = \mathbf{0}$, $\boldsymbol{b}_o = \mathbf{1}$ to force $\boldsymbol{f}_t = \mathbf{0}$ and $\boldsymbol{o}_t = \mathbf{1}$, we get

$$\boldsymbol{y}_t = \sigma(\boldsymbol{c}_t) = \begin{bmatrix} \sigma(\mathbf{0} \odot \mathbf{0}) \\ \sigma(\boldsymbol{a}_{1,t} \odot \boldsymbol{a}_{2,t}) \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \sigma(\boldsymbol{a}_{1,t} \odot \boldsymbol{a}_{2,t}) \end{bmatrix}. \tag{C.22}$$

C.4.4 Composing the operations to model a single Gated RNN layer

To model the gated RNN described in Eq. (5.5), we again follow the same lines as described in Appendix C.3. In particular, we use one LSTM layer for the recurrent state updated as described in Appendix C.4.1. We then stack k LSTM layers as described in Appendix C.4.2 to model the k MLP layers of ϕ and γ in parallel. We then use one final layer to both give the final MLP layer of ϕ and γ and to compute their Hadamard product as set out in Appendix C.4.3 in order to match the output of the gated RNN in Eq. (5.5). Now, since we are working with σ = ReLU, both $\gamma(\boldsymbol{x}_t)$ and $\phi(\boldsymbol{s}_t)$ are positive and therefore so is their product. Hence, applying σ to the product components in Eq. (C.22) leaves the the components invariant. Therefore, we output is

$$\boldsymbol{y}_t = \begin{bmatrix} \mathbf{0} \\ \gamma(\boldsymbol{x}_t) \odot \phi(\boldsymbol{s}_t) \end{bmatrix}, \tag{C.23}$$

as required.

Hence, we have shown that a single layer of a gated RNN as described by Eq. (5.5) can be represented using $k+2$ LSTM layers where k is the maximum depth of ϕ and γ. Therefore, once again, the two universal approximation programs in Listings 1 and 2 can also be implemented for LSTMs. Hence, LSTM models are also universal approximators in the sense described in Section 5.4.

C.5 Gated Linear RNNs are Hawk/Griffin Models

A single residual block of a Hawk/Griffin model (De et al., 2024) consists of two components, a recurrent block for temporal mixing which makes use of a one-dimensional temporal

convolution, as well as real-gated linear recurrent unit (RG-LRU) and a gated MLP block. Specifically, we consider an input $a_t \in \mathbb{R}^{d_{\texttt{in}}}$, inputs to the blocks of dimensions $d_{\texttt{in}}$ and outputs from each block of dimensions $d_{\texttt{in}}$. Within blocks, all vectors have dimensionality $d_{\texttt{hidden}} = Ed_{\texttt{in}}$, where E is denotes an expansion factor. Below, we formally describe the form of the recurrent and gated MLP blocks which are the two main components making up the residual blocks used for Hawk and Griffin.

Recurrent block. The recurrent block consists of two branches. The first applies a one-dimensional temporal convolution followed by a RG-LRU. The second branch simply performs a linear transformation followed by a non-linearity, i.e. applies a single layer of an MLP.

Consider the first branch of the recurrent block with an input a_t. The one-dimensional temporal convolution can be written as:

$$a'_t = W_a a_t, \tag{C.24}$$
$$g_t = \texttt{GeLU}(W_g a_t + b_g), \tag{C.25}$$
$$M_t = \left[a'_{t-(d_{\text{conv}}-1)}, \ldots, a'_{t-2}, a'_{t-1}, a'_t\right], \tag{C.26}$$
$$z_t = \sum_{i=0}^{d_{\text{conv}}-1} W_M[i] M_t[t-i] \; + b_{\text{conv}} \qquad \text{(convolution with window size } d_{\text{conv}}), \tag{C.27}$$

where b_{conv} is a bias vector and $W_M = \left[\tilde{B}, \tilde{A}\tilde{B}, \tilde{A}^2\tilde{B}, \cdots, \tilde{A}^t\tilde{B}, \cdots\right]$ is the convolutional kernel for the one-dimensional temporal convolution.

The output of this convolution is then fed into a RG-LRU. We can write this down concretely using as an input z_t from the one-dimensional convolution and with recurrent state $h_t \in \mathbb{R}^{d_{\texttt{model}}}$:

$$r_t = \texttt{Sigmoid}(W_r z_t + b_r), \tag{C.28}$$
$$i_t = \texttt{Sigmoid}(W_i z_t + b_i), \tag{C.29}$$
$$a = \texttt{Sigmoid}(\Lambda), \qquad (\Lambda \text{ a learnable parameter}) \tag{C.30}$$
$$a_t = a^{c r_t}, \qquad (c = 8 \text{ fixed scalar constant}) \tag{C.31}$$
$$h_t = a_t \odot h_{t-1} + \sqrt{1 - a_t^2} \odot (i_t \odot z_t). \tag{C.32}$$

Now consider the second branch of the recurrent block. This performs a linear transformation followed by a non-linear activation:

$$g_t = \texttt{GeLU}(W_g a_t + b_g). \tag{C.33}$$

To get the final output of the recurrent block, we multiply the components of the vectors computed from each branch within the recurrent block and then perform a non-linear transformation:

$$h'_t = g_t \odot h_t, \tag{C.34}$$

$$\boldsymbol{o}_t = \boldsymbol{W}_o \boldsymbol{h}'_t + \boldsymbol{b}_o. \tag{C.35}$$

Gated MLP block. After passing through the recurrent block, we pass the output $\boldsymbol{o}_t$ into a gated MLP block. Again we have two branches, the first where we linearly transform the input to this block

$$\boldsymbol{e}_t = \boldsymbol{W}_e \boldsymbol{o}_t + \boldsymbol{b}_e, \tag{C.36}$$

and the second performs a single layer MLP transformation as

$$\boldsymbol{f}_t = \texttt{GeLU}(\boldsymbol{W}_f \boldsymbol{o}_t + \boldsymbol{b}_f). \tag{C.37}$$

These are then combined through a Hadamard product and linear transformation as

$$\boldsymbol{e}'_t = \boldsymbol{e}_t \odot \boldsymbol{f}_t, \tag{C.38}$$

$$\boldsymbol{m}_t = \boldsymbol{W}_m \boldsymbol{e}'_t + \boldsymbol{b}_m. \tag{C.39}$$

We then have that the vector $\boldsymbol{m}_t$ acts as the output of the residual block given the input $\boldsymbol{a}_t$.

Distinction between the Griffin and Hawk models. Hawk is the more simple of the two architectures proposed in (De et al., 2024). Here, residual blocks using the recurrent block described above are simply stacked on top of each other to form the Hawk architecture. Griffin, on the other hand, mixes recurrent blocks and local attention. In particular, two residual blocks with recurrent blocks are followed by one residual block using local MQA attention (Beltagy et al., 2020; Shazeer, 2019).

Simplifying Assumptions. We again follow the setup of replacing all `Sigmoid` and `tanh` activation functions with `ReLU` activations which we denote σ. Furthermore, we assume for simplicity that $d_{\texttt{in}} = d_{\texttt{hidden}}$ by choosing $E = 1$. Moreover, the Hawk and Griffin architecture contains residual connections and normalising layers which we omit.[1] We again assume compactness of the input domain $\mathcal{X}$ and denote a vector of finite values $\boldsymbol{k}_{lb}$, such that $\boldsymbol{k}_{lb,i} \leq (\boldsymbol{x}_t)_i$ for $i \in [d_{\texttt{in}}]$ and all $\boldsymbol{x}_t \in \mathcal{X}$, just as before. Finally, we assume that $d_{\text{conv}} = T$ where T is the maximum sequence length.

C.5.1 Representing the state update using a recurrent block

Starting with the input to the Hawk model, which we denote $\boldsymbol{a}_t$, we define this to be a function of the input to the Gated RNN $\boldsymbol{x}_t$ as $\boldsymbol{a}_t = \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{x}_t \end{bmatrix}$. First, we set $\boldsymbol{W}_a = \boldsymbol{I}$ so that $\boldsymbol{a}'_t = \boldsymbol{a}_t$. Next we choose matrices $\tilde{\boldsymbol{A}} = \begin{bmatrix} \boldsymbol{0} & \boldsymbol{A} \\ \boldsymbol{0} & \boldsymbol{0} \end{bmatrix}$ and $\tilde{\boldsymbol{B}} = \begin{bmatrix} \boldsymbol{0} & \boldsymbol{B} \\ \boldsymbol{0} & \boldsymbol{0} \end{bmatrix}$ which we then use,

[1] We will force a lot of our recurrent blocks to implement the identity function. So instead of this, we could implement the 0 function in the recurrent block and use a residual connection between the residual block input and the output of the recurrent block to achieve the same identity function. However, for clarity we ignore residual connections in our derivations.

with a convolutional window size of $d_{\text{conv}} = T$ to form the convolutional kernel $\boldsymbol{W}_M = \left[\tilde{\boldsymbol{B}}, \tilde{\boldsymbol{A}}\tilde{\boldsymbol{B}}, \tilde{\boldsymbol{A}}^2\tilde{\boldsymbol{B}}, \cdots, \tilde{\boldsymbol{A}}^t\tilde{\boldsymbol{B}}, \cdots\right]$. Setting the convolutional bias as $\boldsymbol{b}_{\text{conv}} = \begin{bmatrix}\boldsymbol{0}\\\boldsymbol{1}\end{bmatrix}$ gives

$$\begin{aligned}\boldsymbol{z}_t &= \sum_{i=0}^{t-1} \boldsymbol{W}_M[i]\boldsymbol{M}_t[t-i] \quad + \boldsymbol{b}_{\text{conv}}, &\text{(C.40)}\\ &= \tilde{\boldsymbol{B}}\boldsymbol{a}_t + \tilde{\boldsymbol{A}}\tilde{\boldsymbol{B}}\boldsymbol{a}_{t-1} + \cdots + \tilde{\boldsymbol{A}}^{t-1}\tilde{\boldsymbol{B}}\boldsymbol{a}_1 + \begin{bmatrix}\boldsymbol{0}\\\boldsymbol{1}\end{bmatrix} &\text{(C.41)}\\ &= \begin{bmatrix}\boldsymbol{s}_t\\\boldsymbol{1}\end{bmatrix}. &\text{(C.42)}\end{aligned}$$

Now, we pass $\boldsymbol{z}_t$ through the RG-LRU. We set $\Lambda = 0$ so that $\boldsymbol{a}_t = 0$. We also define $\boldsymbol{W}_i = \boldsymbol{0}$ and $\boldsymbol{b}_i = \boldsymbol{1}$ so that $\boldsymbol{i}_t = \boldsymbol{1}$. This gives us $\boldsymbol{h}_t = \boldsymbol{z}_t$, so that we pass the output of the one-dimensional convolution through he RG-LRU.

Next, let's focus on the second branch. Making use of the lower bound $\boldsymbol{k}_{lb}$ on the domain $\mathcal{X}$, we set $\boldsymbol{W}_g = \boldsymbol{I}$ and $\boldsymbol{b}_g = \begin{bmatrix}\boldsymbol{1}\\-\boldsymbol{k}_{lb}\end{bmatrix}$ so that

$$\boldsymbol{g}_t = \sigma\left(\boldsymbol{I}\begin{bmatrix}\boldsymbol{0}\\\boldsymbol{x}_t\end{bmatrix} + \begin{bmatrix}\boldsymbol{1}\\-\boldsymbol{k}_{lb}\end{bmatrix}\right) = \begin{bmatrix}\sigma(\boldsymbol{1})\\\sigma(\boldsymbol{x}_t - \boldsymbol{k}_{lb})\end{bmatrix} = \begin{bmatrix}\boldsymbol{1}\\\boldsymbol{x}_t - \boldsymbol{k}_{lb}\end{bmatrix}, \tag{C.43}$$

where we used that $(\boldsymbol{x}_t - \boldsymbol{k}_{lb})_i \geq \boldsymbol{0}$ for every i. Combining the two branches gives

$$\boldsymbol{h}'_t = \begin{bmatrix}\boldsymbol{1}\\\boldsymbol{x}_t - \boldsymbol{k}_{lb}\end{bmatrix} \odot \begin{bmatrix}\boldsymbol{s}_t\\\boldsymbol{1}\end{bmatrix} = \begin{bmatrix}\boldsymbol{s}_t\\\boldsymbol{x}_t - \boldsymbol{k}_{lb}\end{bmatrix}. \tag{C.44}$$

We finally get the output of the recurrent block by defining $\boldsymbol{W}_o = \boldsymbol{I}$ and $\boldsymbol{b}_0 = \begin{bmatrix}\boldsymbol{0}\\\boldsymbol{k}_{lb}\end{bmatrix}$ so that

$$\boldsymbol{o}_t = \begin{bmatrix}\boldsymbol{s}_t\\\boldsymbol{x}_t\end{bmatrix}. \tag{C.45}$$

C.5.2 Representing the identity function using a recurrent block

We now show that we can pass an input unchanged through a recurrent block. Assume that the input to the recurrent block is $\boldsymbol{a}_t = \begin{bmatrix}\boldsymbol{a}_{1,t}\\\boldsymbol{a}_{2,t}\end{bmatrix}$ with $\boldsymbol{W}_a = \boldsymbol{I}$ so that $\boldsymbol{a}'_t = \boldsymbol{a}_t$. Then we define matrices $\tilde{\boldsymbol{A}} = \boldsymbol{0}$ and $\tilde{\boldsymbol{B}} = \boldsymbol{I}$ which we then use to form the convolutional kernel $\boldsymbol{W}_M = \left[\tilde{\boldsymbol{B}}, \tilde{\boldsymbol{A}}\tilde{\boldsymbol{B}}, \tilde{\boldsymbol{A}}^2\tilde{\boldsymbol{B}}, \cdots, \tilde{\boldsymbol{A}}^t\tilde{\boldsymbol{B}}, \cdots\right]$. Finally, setting the convolutional bias as $\boldsymbol{b}_{\text{conv}} = \boldsymbol{0}$ results in $\boldsymbol{z}_t = \boldsymbol{a}_t$. From here, we can again set $\Lambda = 0$, $\boldsymbol{W}_i = \boldsymbol{0}$ and $\boldsymbol{b}_i = \boldsymbol{1}$ so that $\boldsymbol{h}_t = \boldsymbol{z}_t$. Looking at the second branch and setting $\boldsymbol{W}_g = 0$ and $\boldsymbol{b}_g = \boldsymbol{1}$ so that $\boldsymbol{h}'_t = \boldsymbol{h}_t$. Finally, we can simply output the input to the recurrent block by setting $\boldsymbol{W}_o = \boldsymbol{I}$ and $\boldsymbol{b}_o = \boldsymbol{0}$ so that $\boldsymbol{o}_t = \boldsymbol{h}_t$ which means that $\boldsymbol{o}_t = \boldsymbol{a}_t$.

C.5.3 Representing each MLP layer as a gated MLP block

We can represent the MLP layers of the networks $\phi(\boldsymbol{s}_t)$ and $\gamma(\boldsymbol{x}_t)$ as described in Eq. (5.4) using Gated MLP blocks. We again denote the i-th layer of ϕ and γ as ϕ_i and γ_i. Assume that the input to the gated MLP block is $\boldsymbol{a}_t = \begin{bmatrix} \boldsymbol{a}_{1,t} \\ \boldsymbol{a}_{2,t} \end{bmatrix}$. Then, on the first purely linear branch, let us define $\boldsymbol{W}_e = \boldsymbol{I}$ and $\boldsymbol{b}_e = \boldsymbol{1}$ so that $\boldsymbol{e}_t = \boldsymbol{1}$. On the second non-linear branch, we can define $\boldsymbol{W}_f = \begin{bmatrix} \boldsymbol{W}_{\phi_i} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{W}_{\gamma_i} \end{bmatrix}$ and $\boldsymbol{b}_f = \begin{bmatrix} \boldsymbol{b}_{\phi_i} \\ \boldsymbol{b}_{\gamma_i} \end{bmatrix}$. This results in

$$\boldsymbol{f}_t = \sigma\left(\begin{bmatrix} \boldsymbol{W}_{\phi_i} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{W}_{\gamma_i} \end{bmatrix} \begin{bmatrix} \boldsymbol{a}_{1,t} \\ \boldsymbol{a}_{2,t} \end{bmatrix} + \begin{bmatrix} \boldsymbol{b}_{\phi_i} \\ \boldsymbol{b}_{\gamma_i} \end{bmatrix}\right) = \begin{bmatrix} \phi_i(\boldsymbol{a}_{1,t}) \\ \gamma_i(\boldsymbol{a}_{2,t}) \end{bmatrix}. \tag{C.46}$$

Due to our setting of $\boldsymbol{e}_t$, we get $\boldsymbol{e}'_t = \boldsymbol{f}_t$. Further, defining $\boldsymbol{W}_m = \boldsymbol{I}$ and $\boldsymbol{b}_m = \boldsymbol{0}$ makes the output of the MLP block be

$$\boldsymbol{m}_t = \begin{bmatrix} \phi_i(\boldsymbol{a}_{1,t}) \\ \gamma_i(\boldsymbol{a}_{2,t}) \end{bmatrix}. \tag{C.47}$$

Emulating the layers of only one the two networks. Suppose without loss of generality (WLOG) that ϕ has m layers and γ has n layers where $m < n$. Suppose also that our input to the MLP block is $\boldsymbol{a}_t = \begin{bmatrix} \phi(\boldsymbol{x}_t) \\ \boldsymbol{a}_{2,t} \end{bmatrix}$. Again, on the first purely linear branch, let us define $\boldsymbol{W}_e = \boldsymbol{I}$ and $\boldsymbol{b}_e = \boldsymbol{1}$ so that $\boldsymbol{e}_t = \boldsymbol{1}$. Now we modify the weights on the second non-linear branch by defining $\boldsymbol{W}_f = \begin{bmatrix} \boldsymbol{I} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{W}_{\gamma_i} \end{bmatrix}$ and $\boldsymbol{b}_f = \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{b}_{\gamma_i} \end{bmatrix}$. This gives us

$$\boldsymbol{f}_t = \sigma\left(\begin{bmatrix} \boldsymbol{I} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{W}_{\gamma_i} \end{bmatrix} \begin{bmatrix} \phi(\boldsymbol{x}_t) \\ \boldsymbol{a}_{2,t} \end{bmatrix} + \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{b}_{\gamma_i} \end{bmatrix}\right) = \begin{bmatrix} \sigma(\phi(\boldsymbol{x}_t)) \\ \gamma_i(\boldsymbol{a}_{2,t}) \end{bmatrix} = \begin{bmatrix} \phi(\boldsymbol{x}_t) \\ \gamma_i(\boldsymbol{a}_{2,t}) \end{bmatrix}, \tag{C.48}$$

where we have used that since $\phi(\boldsymbol{x}_t)$ is a ReLU network whose final activation is a ReLU, we have that $\phi(\boldsymbol{x}_t) = \sigma(\phi(\boldsymbol{x}_t))$. Hence, if our networks have different depths and we have fully emulated one of the networks, we can continue to emulate the remaining layers of the other network while keeping the fully emulated network fixed and unchanged.

C.5.4 Representing the identify function using a gated MLP block

In this section we show that we can represent an identity function using a gated MLP block. This can be simply done by setting $\boldsymbol{W}_f = \boldsymbol{0}, \boldsymbol{b}_f = \boldsymbol{1}, \boldsymbol{W}_e = \boldsymbol{I}, \boldsymbol{b}_e = \boldsymbol{0}, \boldsymbol{W}_m = \boldsymbol{I}$ and $\boldsymbol{b}_m = \boldsymbol{0}$. This then gives us that for an input $\boldsymbol{a}_t = \begin{bmatrix} \boldsymbol{a}_{1,t} \\ \boldsymbol{a}_{2,t} \end{bmatrix}$ to the gated MLP block, the output of the gated MLP block is $\boldsymbol{m}_t = \boldsymbol{a}_t$. Thus, we pass the input through the gated MLP unchanged.

C.5.5 Representing multiplicative gating with a gated MLP block

The final thing we need to do is to compute an element-wise product of two vectors in order to match the output in Eq. (5.4). In other words, to match the $\phi(\boldsymbol{x}_t) \odot \gamma(\boldsymbol{s}_t)$ operation.

Again, assume that the input to the gated MLP block is $\boldsymbol{a}_t = \begin{bmatrix} \boldsymbol{a}_{1,t} \\ \boldsymbol{a}_{2,t} \end{bmatrix}$. Working with the first linear branch, we define $\boldsymbol{W}_e = \begin{bmatrix} \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{I} & \boldsymbol{0} \end{bmatrix}$ and $\boldsymbol{b}_e = \boldsymbol{0}$, so that

$$\boldsymbol{e}_t = \begin{bmatrix} \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{I} & \boldsymbol{0} \end{bmatrix} \begin{bmatrix} \boldsymbol{a}_{1,t} \\ \boldsymbol{a}_{2,t} \end{bmatrix} + \boldsymbol{0} = \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{a}_{1,t} \end{bmatrix}. \tag{C.49}$$

Next, we define $\boldsymbol{W}_f = \boldsymbol{I}$ and $\boldsymbol{b}_e = \boldsymbol{0}$ so that

$$\boldsymbol{f}_t = \begin{bmatrix} \sigma(\boldsymbol{a}_{1,t}) \\ \sigma(\boldsymbol{a}_{2,t}) \end{bmatrix}. \tag{C.50}$$

Setting $\boldsymbol{W}_m = \boldsymbol{I}$ and $\boldsymbol{b}_m = \boldsymbol{0}$ gives the output of the gated MLP as

$$\boldsymbol{m}_t = \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{a}_{1,t} \odot \sigma(\boldsymbol{a}_{2,t}) \end{bmatrix}. \tag{C.51}$$

C.5.6 Composing the operations to model a single gated linear-RNN layer

Now that we have all the individual layers, we can combine them so that we can use a Hawk model to emulate a single Gated RNN layer.

First we start by taking the input of the form $\boldsymbol{a}_t = \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{x}_t \end{bmatrix}$. We use a residual block that consists of a recurrent block computing the state update as descried in Appendix C.5.1 and then a gated MLP block that computes the identity function as demonstrated in Appendix C.5.4. This gives an output from this first recurrent block as $\boldsymbol{o}_t = \begin{bmatrix} \boldsymbol{s}_t \\ \boldsymbol{x}_t \end{bmatrix}$.

Next, we emulate the MLP layers of the networks ϕ and γ in parallel. Suppose WLOG that ϕ and γ have m and n MLP layers respectively, where $m \leq n$. We stack m residual blocks using recurrent blocks that implement the identity function as described in Appendix C.5.2 followed by MLP blocks that apply the MLP layers of ϕ and γ as described in Appendix C.5.3. Stacking m such residual blocks results in the output $\boldsymbol{m}_t = \begin{bmatrix} \gamma_m(\boldsymbol{s}_t) \\ \phi(\boldsymbol{x}_t) \end{bmatrix}$, where we can fully emulate the shallower network $\phi(\boldsymbol{x}_t)$.

Now, for the remaining $k - m$ layers for the network $\gamma(\boldsymbol{x}_t)$, we stack residual blocks with recurrent blocks implementing the identity function as described in Appendix C.5.2 and MLP blocks that leave $\phi(\boldsymbol{x}_t)$ unchanged whilst applying the additional layers needed to emulate $\gamma(\boldsymbol{s}_t)$ as described at the end of Appendix C.5.3. After stacking $k - m$ additional residual layers in this fashion, the output of the final residual block will now be $\boldsymbol{m}_t = \begin{bmatrix} \gamma(\boldsymbol{s}_t) \\ \phi(\boldsymbol{x}_t) \end{bmatrix}$, which fully reconstructs the MLP networks γ and ϕ.

Finally, we utilise a residual block with a recurrent block that implements the identity function as described in Appendix C.5.2 followed by a gated MLP block that applies multiplicative gating as described in Appendix C.5.5. This then gives as an output of this final residual block $\boldsymbol{m}_t = \begin{bmatrix} \mathbf{0} \\ \gamma(\boldsymbol{s}_t) \odot \sigma(\phi(\boldsymbol{x}_t)) \end{bmatrix}$. Since $\phi(\boldsymbol{x}_t)$ is a MLP network with the final activation function being a ReLU activation, we have that $\sigma(\phi(\boldsymbol{x}_t)) = \phi(\boldsymbol{x}_t)$, giving the required final output from the stacked block of residual blocks as

$$\boldsymbol{m}_t = \begin{bmatrix} \mathbf{0} \\ \gamma(\boldsymbol{s}_t) \odot \phi(\boldsymbol{x}_t) \end{bmatrix}. \tag{C.52}$$

Hence, we have shown that a single layer of a gated RNN as described by Eq. (5.5) can be represented using $k+2$ Hawk residual blocks where k is the maximum depth of ϕ and γ. Once again, the two universal approximation programs in Listings 1 and 2 can also be applied to Hawk models as they can represent Gated Linear RNNs. Therefore, Hawk models are also universal approximators in the sense described in Section 5.4.

Gated Linear-RNNs are Griffin models too. The above argument extends to the Griffin architecture which uses stacks of two residual blocks with recurrent blocks followed by a residual block with attention. The only thing that changes is that for every third residual block, which in our argument will be used to compute the MLP layers of ϕ and γ in parallel, the recurrent block is now replaced with a local MQA block.

We can set the key query and values matrices to implement the identity function which is to act input to the block. Hence, as a corollary of the above argument, we can also show that the universal approximation programs in Listings 1 and 2 can also be implemented as Griffin models. Therefore, Griffin models can also be universal approximators in the sense described in Section 5.4.

C.6 Definitions for some helper functions in LSRL

C.6.1 `f_not`

This is a convenience function that creates a NOT function block. It assumes that x is 0 or 1. Works with scalar and vector-valued inputs. With vector-valued inputs, it acts element-wise.

```
1 not_x = 1 - x
```

C.6.2 `f_and`

This is a convenience function that creates an AND function block. It assumes that x and y are 0 or 1. Works with scalar and vector-valued inputs. With vector-valued inputs, it acts element-wise. `mu` is the approximation parameter μ for `f_step` as described in Section 5.3.

```
1 and_x_y = ReLU(f_step(x, mu) + f_step(y, mu) - 1)
```

C.6.3 f_or

This is a convenience function that creates an OR function block. It assumes that x and y are 0 or 1. Works with scalar and vector-valued inputs. With vector-valued inputs, it acts element-wise. mu is the approximation parameter μ for f_step as described in Section 5.3.

```
or_x_y = f_step(x + y, mu=mu)
```

C.6.4 f_smaller

This is a convenience function that a less than comparison block. Works with scalar and vector-valued inputs. With vector-valued inputs, it acts element-wise. mu is the approximation parameter μ for f_step as described in Section 5.3.

```
smaller_x_y = f_step(y - x, mu=mu)
```

C.6.5 f_larger

This is a convenience function that a more than comparison block. Works with scalar and vector-valued inputs. With vector-valued inputs, it acts element-wise. mu is the approximation parameter μ for f_step as described in Section 5.3.

```
larger_x_y = f_step(x - y, mu=mu)
```

C.6.6 f_relu_identity

Identity operation using ReLUs. This is useful for debranching when some of the branches have ReLUs but the other don't. We can add this as a bypass for the ones that do not and can then merge the ReLUs together (see Appendix C.1 for details).

```
positive_part = ReLU(x)
negative_part = ReLU(
    Linear(
        input=x,
        A=-1 * eye(x.dim),
        b=zeros(x.dim, 1),
    )
)
both = Concat([positive_part, negative_part])
relu_identity = Linear(
    input=both,
    A=hstack(eye(x.dim), -1 * eye(x.dim)),
    b=zeros(x.dim, 1),
)
```

C.6.7 f_modulo_counter

Computes the x mod divisor where x is a counter starting from zero. The idea is that we rotate a unit vector so that it makes a full revolution every divisor rotations. dummy_input can be any variable, we use it only to construct a constant.

```
angle = 2 * pi / divisor
R = [[cos(angle), sin(angle)], [sin(angle), cos(angle)]]
unit_vector = [[1], [0]]
# we first rotate, then output so if we want the first output to be 0 we need to have the
    init_state one step before that
init_state = R.inv() @ unit_vector
# this rotates a 2D vector 1/divisor revolutions at a time
cycler = LinState(
    input=dummy_input,
    A=R,
    B=zeros(2, dummy_input.dim),
    init_state=init_state,
)
# we now need to extract the position of the cycler
extractor_matrix = vstack(*[(R^i * unit_vector).T) for i in range(divisor)])
indicator = Linear(
    input=cycler,
    A=extractor_matrix,
    b=zeros(divisor, 1)
)
# the dot product with the row of extractor_matrix corresponding to the current position of the
    cycler is 1
# the dot product with the second highest is cos(angle)
# thus, we can threshold at 1-cos(angle/2) to get a one hot encoding of the current position of the
    cycler
one_hot = f_larger(indicator, cos(angle / 2))
# and to get an integer value we need one final linear layer
mod_value = Linear(
    one_hot,
    A=[[i for i in range(divisor)]],
    b=zeros(1, 1)
)
```