Universal In-Context Approximation By Prompting Fully Recurrent Models

Aleksandar Petrov, Tom A. Lamb, Alasdair Paren, Philip H.S. Torr, Adel Bibi
Department of Engineering Science
University of Oxford
aleks@robots.ox.ac.uk

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.

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. 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). 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., 2023).

38th Conference on Neural Information Processing Systems (NeurIPS 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. (2024). 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., 2022; 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:

i. 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.
ii. 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).
iii. We present constructions with and without multiplicative gating. However, we observe that the constructions without these gates depend on numerically unstable conditional logic.
iv. 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.

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:

$$\boldsymbol{y}_t = g(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_t). \tag{1}$$

We will abuse the notation and refer to $(\boldsymbol{y}_1, ..., \boldsymbol{y}_t)$=$(g(\boldsymbol{x}_1), ..., g(\boldsymbol{x}_1, ..., \boldsymbol{x}_t))$ as simply $g(\boldsymbol{x}_1, ..., \boldsymbol{x}_t)$. We will also separate the input sequence into a query $(\boldsymbol{q}_1, ..., \boldsymbol{q}_n)$ and a prompt $(\boldsymbol{p}_1, ..., \boldsymbol{p}_N)$. 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]

There are various neural network architectures that fall under the general framework of Eq. (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:

$$\begin{aligned} \boldsymbol{s}_t &= \sigma(\boldsymbol{A}\boldsymbol{s}_{t-1} + \boldsymbol{B}\boldsymbol{x}_t + \boldsymbol{b}), \\ \boldsymbol{y}_t &= \phi(\boldsymbol{s}_t), \end{aligned} \qquad \text{(Classic RNN)} \tag{2}$$

[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 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.

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 (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 Eq. (2):

$$\begin{aligned} \boldsymbol{s}_t &= \boldsymbol{A}\boldsymbol{s}_{t-1} + \boldsymbol{B}\boldsymbol{x}_t + \boldsymbol{b}, \\ \boldsymbol{y}_t &= \phi(\boldsymbol{s}_t). \end{aligned} \qquad \text{(Linear RNN)} \qquad (3)$$

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 (Wang and Xue, 2023; Boyd and Chua, 1985). 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:

$$\begin{aligned} \boldsymbol{s}_t &= \boldsymbol{A}\boldsymbol{s}_{t-1} + \boldsymbol{B}\boldsymbol{x}_t + \boldsymbol{b}, \\ \boldsymbol{y}_t &= \gamma(\boldsymbol{x}_t) \odot \phi(\boldsymbol{s}_t), \end{aligned} \qquad \text{(Gated Linear RNN)} \qquad (4)$$

with γ being another MLP and $\odot$ being the element-wise multiplication operation (Hadamard product). Eq. (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 App. E. We can similarly add multiplicative gating to the classic RNN architecture:

$$\begin{aligned} \boldsymbol{s}_t &= \sigma(\boldsymbol{A}\boldsymbol{s}_{t-1} + \boldsymbol{B}\boldsymbol{x}_t + \boldsymbol{b}), \\ \boldsymbol{y}_t &= \gamma(\boldsymbol{x}_t) \odot \phi(\boldsymbol{s}_t), \end{aligned} \qquad \text{(Gated RNN)} \qquad (5)$$

Eq. (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 Apps. C and D). 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.

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., 2022; Coda-Forno et al., 2023; Bai et al., 2023). 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 $\sup_{\boldsymbol{x}\in\mathcal{D}} |f(\boldsymbol{x}){-}h(\boldsymbol{x})| {\le} \epsilon$. 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., 2019), RNNs (Schäfer and Zimmermann, 2006) 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 Eq. (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:

$$\mathcal{H}_g^{\mathcal{D}^n} = \{(\boldsymbol{q}_1, \ldots, \boldsymbol{q}_n) \mapsto g(\boldsymbol{q}_1, \ldots, \boldsymbol{q}_n, \boldsymbol{p}_1, \ldots, \boldsymbol{p}_N)[\text{-}n'{:}] \mid \forall \boldsymbol{p}_i \in \mathcal{D}, N > 0\}. \qquad (6)$$

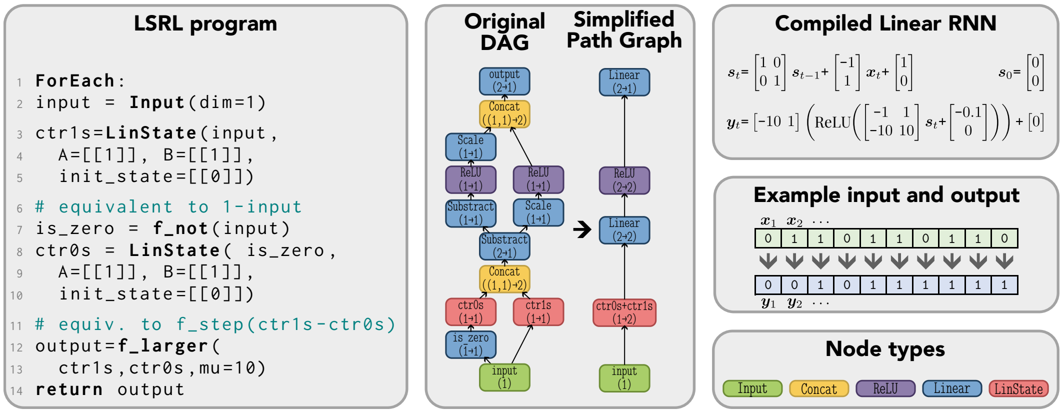


Figure 1: **Compilation of an LSRL program to a Linear RNN.** An example of a simple LSRL program that takes a sequence of 0s and 1s as an input and outputs 1 if there have been more 1s than 0s and 0 otherwise. The LSRL compiler follows the rules in App. A to simplify the computation DAG into a path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

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 Eq. (6) depend on the architecture of g, as well as its specific parameters.

We study the recurrent architectures in Eqs. (2) to (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 Eq. (6) with $D{=}[0,1]^{d_{\text{in}}}$, $n{=}n'{=}1$ and g some *fixed* model of one of the four architectures in Eqs. (2) to (5), and $\mathcal{H}^{\text{tok}}(g)$ with $D{=}\mathcal{V}$ and $n{=}n'{=}l$.

3 Linear State Recurrent Language (LSRL)

We can construct the weights for universal in-context models with the architectures in Eqs. (2) to (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 Language* (LSRL).[2] LSRL programs compile to the four architectures in Eqs. (2) to (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 Secs. 4 to 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 `Input`$(\boldsymbol{x}) = \boldsymbol{x}$ with an $\boldsymbol{x}$ of a fixed dimension. Only one `Input` can be declared in a program. Linear layers and ReLUs are also supported: `Lin`$[\boldsymbol{A}, \boldsymbol{b}](\boldsymbol{x}) := \boldsymbol{A}\boldsymbol{x} + \boldsymbol{b}$, `ReLU`$(\boldsymbol{x}) := \max(\boldsymbol{0}, \boldsymbol{x})$. The unique component of LSRL, however, is its `LinState` operation implementing the linear state update in Linear RNNs (Eq. (3)): `LinState`$[\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{b}, \boldsymbol{s}_0](\boldsymbol{x}_t) := \boldsymbol{A}\boldsymbol{s}_{t-1} + \boldsymbol{B}\boldsymbol{x}_t + \boldsymbol{b}$, where the state $\boldsymbol{s}_{t-1}$ 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

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

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

Listing 1: **LSRL program for universal approximation in-context for continuous functions.** The inputs are $\boldsymbol{q} = [\boldsymbol{q}'^\top, \boldsymbol{0}_{d_\text{out}+1}^\top]^\top$ 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 Eq. (8).

variables: $\texttt{Concat}(\boldsymbol{x}, \boldsymbol{y}) := (\boldsymbol{x}_1, ..., \boldsymbol{x}_{|\boldsymbol{x}|}, \boldsymbol{y}_1, ..., \boldsymbol{y}_{|\boldsymbol{y}|})$. 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: $\texttt{Multi}(\boldsymbol{x}) := \boldsymbol{x}[\,:\, {}^{|\boldsymbol{x}|}/_2] \odot \boldsymbol{x}[{}^{|\boldsymbol{x}|}/_2 :\,]$. Naturally, `Multi` requires that $\boldsymbol{x}$ 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).

Such a program operates over a single token $\boldsymbol{x}_t$ 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 Fig. 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 Sec. 4 we show how carefully constructing state updates and auxiliary variables can nevertheless allow to program some limited conditional behaviours.

Compilation. Any LSRL program without `Multi` nodes can be compiled to a Linear RNN (Eq. (3)) or to a Gated Linear RNN (Eq. (4)). 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 App. A. Later, in Sec. 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.

[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 $s_{t+1} = 0 \texttt{ if } (s_t > 5) \texttt{ else } s_t$.

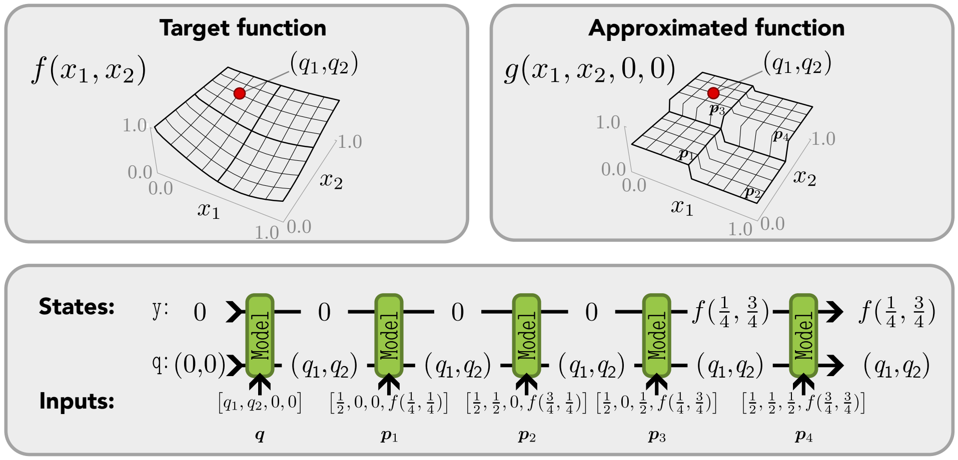


Figure 2: **Intuition behind the LSRL program for universal in-context approximation for continuous functions in Lst. 1.** Our target function f has input dimension $d_{\text{in}} = 2$ and output dimension $d_{\text{out}} = 1$. Each input dimension is split into two parts, hence $\delta = 1/2$. We illustrated 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.

Syntactic sugar. To make programming easier, we define several convenience functions. For instance, we can `Slice` variables $\boldsymbol{x}[l{:}u]$ 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: $\texttt{f\_step}[\mu](\boldsymbol{x}) := \texttt{ReLU}(\mu\boldsymbol{x}) - \mu\texttt{ReLU}(\boldsymbol{x} - 1/\mu)$, where μ is a positive constant controlling the 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 App. F 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:

$$
\begin{aligned}
\texttt{f\_ifelse}[\texttt{cond}, \texttt{t}, \texttt{f}, \lambda](\boldsymbol{x}) := {} & \texttt{ReLU}(\texttt{-}\lambda\,\texttt{cond}(\boldsymbol{x})\texttt{+f}(\boldsymbol{x})) + \texttt{ReLU}(\texttt{-}\lambda\,\texttt{f\_not}(\texttt{cond}(\boldsymbol{x}))\texttt{+t}(\boldsymbol{x})) \\
& - \texttt{ReLU}(\texttt{-}\lambda\,\texttt{cond}(\boldsymbol{x})\texttt{-f}(\boldsymbol{x})) - \texttt{ReLU}(\texttt{-}\lambda\,\texttt{f\_not}(\texttt{cond}(\boldsymbol{x}))\texttt{-t}(\boldsymbol{x})),
\end{aligned} \tag{7}
$$

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 and we will study alternatives in Sec. 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.

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, 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.

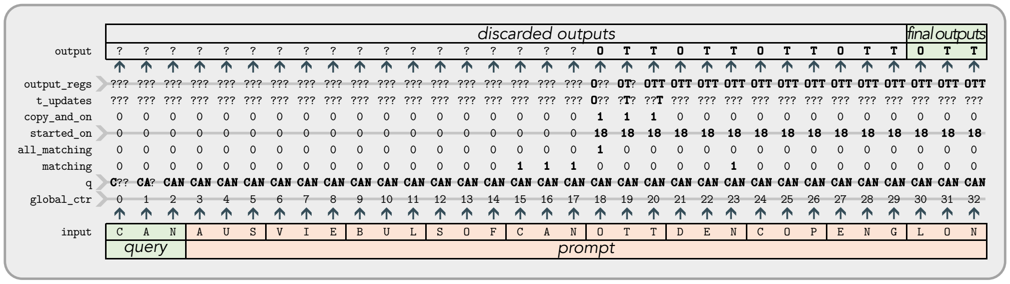


Figure 3: **Intuition behind the LSRL program for universal in-context approximation for discrete functions in Lst. 2.** Our keys and values have length n=3 and represent countries and capitals, e.g., `AUS`tria↦`VIE`nna, `BUL`garia↦`SOF`ia, and so on. The query is `CAN` for Canada and the final n outputs are `OTT` (Ottawa). We show the values of some of the variables in Lst. 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.

4 Universal In-Context Approximation with Linear RNNs

We proceed with building LSRL programs that are universal in-context approximators: one for approximating continuous functions ($\mathcal{C}^{\text{vec}}$), and one for maps between token sequences ($\mathcal{C}^{\text{tok}}$).

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

$$\boldsymbol{p} = [\delta, l_1, \ldots, l_{d_{\text{in}}}, \bar{\boldsymbol{y}}_1, \ldots \bar{\boldsymbol{y}}_{d_{\text{out}}}]^\top, \tag{8}$$

where $\bar{\boldsymbol{y}}$ is the value of f at the centre of that cell: $\bar{\boldsymbol{y}} = f(l_1 + \delta/2, \ldots, l_{d_{\text{in}}} + \delta/2)$. 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 Lst. 1 and an illustration for $d_{\text{in}} = 2, d_{\text{out}} = 1, \delta = 1/2$ is shown in Fig. 2. The prompt length required to approximate an L-Lipschitz function f (w.r.t. the ℓ_2 norm) to precision ϵ is $N = \left(2\epsilon/L\sqrt{d_{\text{in}}}\right)^{-d_{\text{in}}} = \mathcal{O}(\epsilon^{-d_{\text{in}}})$ (see App. B for the proof). Asymptotically, this is as good as one can hope without further assumptions on 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. 2024.

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

Sec. 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 Fig. 3 for an illustration of the setup). The model stores the

```
1  ForEach:
2  input = Input(dim=1)
3  # counter needed to know whether we are looking at the query or the prompt
4  const_1 = f_constant(input, 1)
5  global_ctr = LinState(input=const_1, A=[[1]], B=[[1]], init_state=[[-1]])
6  # counters mod[n] and mod[2n]
7  mod_n_ctr = f_modulo_counter(input, n)
8  mod_2n_ctr = f_modulo_counter(input, 2*n)
9  # which mode are we in (looking at the query, comparing query with key, or copying value to state)
10 is_prompt = f_larger(global_ctr, n-0.5)
11 is_compare_mode = f_larger(mod_2n_ctr, n-0.5)
12 is_copy_mode = f_and(is_prompt, f_not(is_compare_mode))
13 is_first_token_for_copy = f_and(is_copy_mode,f_smaller(mod_n_ctr, 0.5))
14 # update the state holding the query if this is one of the first n tokens
15 tq=f_ifelse(f_smaller(is_prompt, 0.5), t=input, f=input*0)
16 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]
17 q = LinState(input=Concat(tqs), A=eye(n), B=eye(n), init_state=zeros(n,1)) # query
18 # if we are in compare mode (looking at keys), check if this token matches the corresponding one in the query
19 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]
20 cor_q_el = Linear(input=Concat(qs), A=ones(1,n), b=zeros(1,1))
21 matching = f_and(f_and(f_larger(input, cor_q_el-0.5),f_smaller(input, cor_q_el+0.5)),is_compare_mode)
22 # keep a buffer of the last last n+1 match values, the +1 because we can only read the buffer after writing
23 buffer = LinState(input=matching, A=shift_matrix, B=[[0] for _ in 1..n], [[1]]), init_state=zeros(n+1,1))
24 buffer_sum = Linear(input=buffer, A=[[1 for _ in 1..n], [0]], b=zeros(1, 1))
25 all_matching = f_larger(buffer_sum, n-0.5)
26 # 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
27 matching_and_first_for_copy = f_and(all_matching, is_first_token_for_copy)
28 t_started_on_update = f_ifelse(matching_and_first_for_copy,t=global_ctr, f=global_ctr*0)
29 started_on = LinState(input=t_started_on_update, A=eye(1), B=eye(1), init_state=zeros(1, 1))
30 # copying to state for n iterations after started_on
31 copy_and_on = f_and(is_copy_mode, f_smaller(global_ctr, started_on+n))
32 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 ]
33 t_updates_should_update = [f_and(copy_and_on, mod_n_eq_i[i]) for i in 1..n]
34 t_updates = [f_ifelse(f_larger(t_updates_should_update[i], 0.5), t=input, f=input*0) for i in 1..n]
35 output_regs = [LinState(input=update, A=eye(1), B=eye(1), init_state=zeros(1,1)) for update in t_updates]
36 # finally, read out the value from the corresponding output register in order to output from the model
37 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]
38 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 $\boldsymbol{q}_1, ..., \boldsymbol{q}_n$ (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.

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 Lst. 2, implementing dictionary lookup in a linear recurrent model is much less straightforward than executing `dict[key]` in a general-purpose programming language.

Lst. 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 (App. F.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 Stable Universal In-Context Approximation with Gated Linear RNNs

The ReLU-based conditional operator is not numerically stable. The LSRL programs in Lsts. 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 Eq. (7) in order to implement different behaviours depending on whether

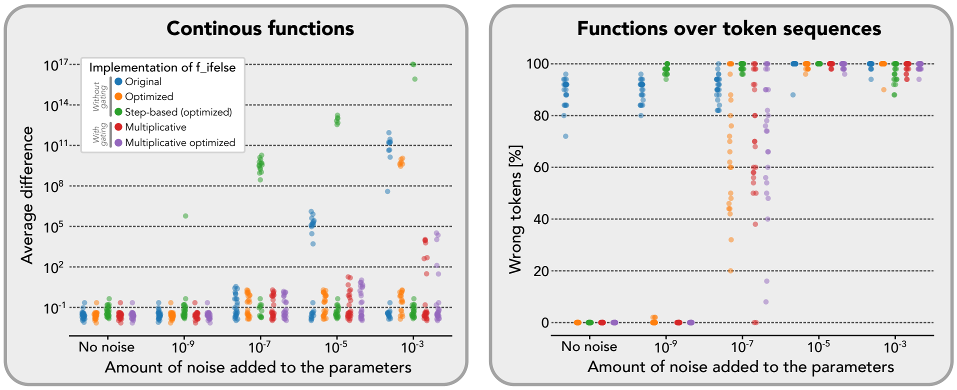


Figure 4: **Robustness of the various `f_ifelse` implementations to model parameter noise.** We show how the performance of the two universal approximation programs in Lsts. 1 and 2 deteriorates as we add Gaussian noise of various magnitudes to the non-zero weights of the resulting compiled models. As expected, the original `f_ifelse` implementation in Eq. (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 Lst. 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 Lst. 2 we report the percentage of wrong token predictions over 5 queries for 25 dictionary maps. Lower values are better in both cases.

we are processing the query or specific parts of the prompt. This operator is not numerically stable. The first term in Eq. (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 Lst. 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 Fig. 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 limitations of the universal approximation results in Sec. 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 Eq. (7) and obtain more realistic recurrent models that are universal approximators in-context.

Removing unnecessary terms in Eq. (7). Eq. (7) has 4 separate `ReLU` terms. The first two handle the cases when $\texttt{t}(\boldsymbol{x})$ and $\texttt{f}(\boldsymbol{x})$ 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. Additionally, if $\texttt{f}(\boldsymbol{x})$ is always 0, then the first and third terms can be safely dropped. Similarly, the second and fourth are unnecessary if $\texttt{f}(\boldsymbol{x}) \equiv 0$. All `f_ifelse` in Lsts. 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 $\texttt{t}(\boldsymbol{x})$ and $\texttt{f}(\boldsymbol{x})$ 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 Fig. 4 show, `optimized` is indeed more numerically robust than `original`.

Step-based implementation. We can get rid of the input sensitivity of Eq. (7) using `f_step`:

$$\begin{aligned}\texttt{f\_ifelse}[\texttt{cond},\texttt{t},\texttt{f},\lambda](\boldsymbol{x}) := {}&\texttt{ReLU(-}\lambda\texttt{+}\lambda\texttt{f\_step(}{}^1\!/\!_2\texttt{-cond(}\boldsymbol{x}\texttt{))+f(}\boldsymbol{x}\texttt{))} + \texttt{ReLU(-}\lambda\texttt{+}\lambda\texttt{f\_step(cond(}\boldsymbol{x}\texttt{)-}{}^1\!/\!_2\texttt{)+t(}\boldsymbol{x}\texttt{))}\\ &- \texttt{ReLU(-}\lambda\texttt{+}\lambda\texttt{f\_step(}{}^1\!/\!_2\texttt{-cond(}\boldsymbol{x}\texttt{))-f(}\boldsymbol{x}\texttt{))} - \texttt{ReLU(-}\lambda\texttt{+}\lambda\texttt{f\_step(cond(}\boldsymbol{x}\texttt{)-}{}^1\!/\!_2\texttt{)-t(}\boldsymbol{x}\texttt{))}.\end{aligned} \tag{9}$$

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 Fig. 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 (Eq. (4)) and Gated RNN models (Eq. (5)) can help via implementing a numerically stable conditional operator:

$$\texttt{f\_ifelse}[\texttt{cond}, \texttt{t}, \texttt{f}](\boldsymbol{x}) := \texttt{cond}(\boldsymbol{x}) \odot \texttt{t}(\boldsymbol{x}) + \texttt{f\_not}(\texttt{cond}(\boldsymbol{x})) \odot \texttt{f}(\boldsymbol{x}), \tag{10}$$

where the element-wise product is implemented in LSRL with `Concat` and `Multi`. We will refer to the implementation of `f_ifelse` in Eq. (10) as `multiplicative`. Similarly to original implementation of `f_ifelse` in Eq. (7), we can drop the $\texttt{t}(\boldsymbol{x})$ and $\texttt{f}(\boldsymbol{x})$ term if they are equal to zero (`multiplicative optimized`). If $\texttt{cond}(\boldsymbol{x})$ is not exactly zero, $\texttt{cond}(\boldsymbol{x}) \odot \texttt{t}(\boldsymbol{x})$ 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, Eq. (10) should be more robust to numerical precision issues than Eq. (7). Fig. 4 shows that this is the case in practice with Lsts. 1 and 2 being more robust to parameter noise when using multiplicative gates compared to the `ReLU`-based implementations. Therefore, Linear Gated RNNs (Eq. (4)) —to which models with multiplicative gates can be compiled— are more likely than Linear RNNs (Eq. (3)) to exhibit universal approximation properties in practice.

6 Universal In-context Approximation with Non-linear (Gated) RNNs

Secs. 4 and 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 (Eqs. (2) and (5)). Concretely, we show how every *linear* (Gated) RNN can be converted to a *non-linear* (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:

$$\begin{aligned} \boldsymbol{s}_t &= \boldsymbol{A}\boldsymbol{s}_{t-1} + \boldsymbol{B}\boldsymbol{x}_t + \boldsymbol{b} \\ \boldsymbol{y}_t &= \phi(\boldsymbol{s}_t). \end{aligned} \quad \equiv \quad \begin{aligned} \begin{bmatrix} \boldsymbol{s}_t^+ \\ \boldsymbol{s}_t^- \end{bmatrix} &= \texttt{ReLU}\left(\begin{bmatrix} \boldsymbol{A} & -\boldsymbol{A} \\ -\boldsymbol{A} & \boldsymbol{A} \end{bmatrix} \begin{bmatrix} \boldsymbol{s}_t^+ \\ \boldsymbol{s}_t^- \end{bmatrix} + \begin{bmatrix} \boldsymbol{B} \\ -\boldsymbol{B} \end{bmatrix} \boldsymbol{x}_t + \begin{bmatrix} \boldsymbol{b} \\ -\boldsymbol{b} \end{bmatrix} \right) \\ \boldsymbol{y}_t &= \phi\left([\boldsymbol{I} \quad -\boldsymbol{I}] \begin{bmatrix} \boldsymbol{s}_t^+ \\ \boldsymbol{s}_t^- \end{bmatrix} \right). \end{aligned} \tag{11}$$

Using Eq. (11) we can compile any LSRL program to an RNN (Eq. (2)) or a Gated RNN (Eq. (5)). This includes Lsts. 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 (App. C) or an LSTM (App. D), these models are too universal in-context approximators.

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*.

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 impossible. Therefore, it is important to further study the safety and security implications of these properties.

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 $\boldsymbol{A}$ matrix in the state update equations. However, many state-of-the-art models impose structural constraints on $\boldsymbol{A}$ (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.

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). Adel Bibi acknowledges funding from the KAUST Office of Sponsored Research (OSR-CRG2021-4648) and support from Google Cloud through the Google Gemma 2 Academic Program GCP Credit Award.

References

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 *The Eleventh 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*.

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.

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

Yu Bai, Fan Chen, Huan Wang, Caiming Xiong, and Song Mei. 2023. Transformers as statisticians: Provable in-context learning with in-context algorithm selection. In *Advances in neural information processing systems*.

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.

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

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.

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. In *Advances in Neural Information Processing Systems*.

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)*.

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.

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*.

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*.

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*.

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*.

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

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*.

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*.

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*.

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

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

Emanuele La Malfa, Aleksandar Petrov, Simon Frieder, Christoph Weinhuber, Ryan Burnell, Raza Nazar, Anthony G. Cohn, Nigel Shadbolt, and Michael Wooldridge. 2023. Language Models as a Service: Overview of a new paradigm and its challenges. *arXiv preprint arXiv:2309.16573*.

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*.

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*.

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*.

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.

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*.

Aleksandar Petrov, Philip HS Torr, and Adel Bibi. 2024. Prompting a pretrained transformer can be a universal approximator. In *International Conference on Machine Learning*.

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.

Anton Maximilian Schäfer and Hans Georg Zimmermann. 2006. Recurrent neural networks are universal approximators. In *Artificial Neural Networks–ICANN 2006: 16th International Conference, Athens, Greece, September 10-14, 2006. Proceedings, Part I 16*, pages 632–640. Springer.

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

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

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*.

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.

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*.

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*.

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

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*.

Sang Michael Xie, Aditi Raghunathan, Percy Liang, and Tengyu Ma. 2022. An explanation of in-context learning as implicit Bayesian inference. In *International Conference on Learning Representations*.

Chulhee Yun, Srinadh Bhojanapalli, Ankit Singh Rawat, Sashank Reddi, and Sanjiv Kumar. 2019. Are transformers universal approximators of sequence-to-sequence functions? In *International Conference on Learning Representations*.

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

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*.

A 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.

The debranching process goes through the following cases in order. And iterates 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*.

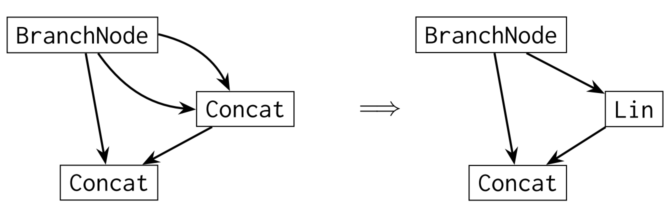
Case 1A: If all subsequent nodes are `Multi`. As all `Multi` nodes that have the same input (the branching node) they must all be producing the exact same output. Hence, only one can be kept. This removes one branch.

Case 1B: If 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 multiplicatin 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 `Slices`. 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 `Slices` 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 `Slices` 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 $\boldsymbol{b} = \mathbf{0}$ 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 `Slices`. We can combine them into a single `Lin` node and then add `Slices` to extract the relevant subspaces for the subsequent nodes. These `Slices` 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 Eq. (11), by splitting the positive and negative components and treating them separately. See App. F.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 Sec. 2.

B Error Bound on the Approximation Scheme for Continuous Functions

In Sec. 4.1 we outlined a strategy to perform universal in-context approximation for continuous functions with Linear RNNs. The full program is in Lst. 1 and an illustration of the scheme is presented in Fig. 2. In Sec. 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 Lst. 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 Gated RNNs are GRU models

A GRU layer (Cho et al., 2014) with input $\boldsymbol{a}_t \in \mathbb{R}^{d_{\texttt{in}}}$ 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:

$$\boldsymbol{z}_t = \texttt{Sigmoid}(\boldsymbol{W}_z\boldsymbol{a}_t + \boldsymbol{U}_z\boldsymbol{h}_{t-1} + \boldsymbol{b}_z), \qquad \text{(update gate vector)} \tag{12}$$

$$\boldsymbol{r}_t = \texttt{Sigmoid}(\boldsymbol{W}_r\boldsymbol{a}_t + \boldsymbol{U}_r\boldsymbol{h}_{t-1} + \boldsymbol{b}_r), \qquad \text{(reset gate vector)} \tag{13}$$

$$\hat{\boldsymbol{h}}_t = \texttt{tanh}(\boldsymbol{W}_h\boldsymbol{a}_t + \boldsymbol{U}_h(\boldsymbol{r}_t \odot \boldsymbol{h}_{t-1}) + \boldsymbol{b}_h), \qquad \text{(candidate activation vector)} \tag{14}$$

$$\boldsymbol{h}_t = (1 - \boldsymbol{z}_t) \odot \boldsymbol{h}_{t-1} + \boldsymbol{z}_t \odot \hat{\boldsymbol{h}}_t, \qquad \text{(output vector)} \tag{15}$$

In this section, we show a conversion of a single Gated RNN layer (Eq. (5)) to $k+2$ GRU layers. Here, k is the number of layers in the γ and h MLPs in Eq. (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.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, Eq. (14) reduces to:

$$\hat{\boldsymbol{h}}_t = \sigma(\boldsymbol{W}_h\boldsymbol{a}_t + \boldsymbol{U}_h\boldsymbol{h}_{t-1} + \boldsymbol{b}_h), \tag{16}$$

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 Eq. (15) becoming:

$$\boldsymbol{h}_t = \sigma\left(\begin{bmatrix}\boldsymbol{0} & \boldsymbol{B}\\ \boldsymbol{0} & \boldsymbol{I}\end{bmatrix}\begin{bmatrix}\boldsymbol{0}\\ \boldsymbol{x}_t\end{bmatrix} + \begin{bmatrix}\boldsymbol{A} & \boldsymbol{0}\\ \boldsymbol{0} & \boldsymbol{0}\end{bmatrix}\begin{bmatrix}\boldsymbol{s}_{t-1}\\ \boldsymbol{0}\end{bmatrix} + \begin{bmatrix}\boldsymbol{b}\\ -\boldsymbol{k}_{lb}\end{bmatrix}\right) = \begin{bmatrix}\sigma(\boldsymbol{A}\boldsymbol{s}_{t-1} + \boldsymbol{B}\boldsymbol{x}_t + \boldsymbol{b})\\ \sigma(\boldsymbol{x}_t - \boldsymbol{k}_{lb})\end{bmatrix} = \begin{bmatrix}\sigma(\boldsymbol{s}_t)\\ \boldsymbol{x}_t - \boldsymbol{k}_{lb}\end{bmatrix}. \tag{17}$$

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 (11) rather than subtracting $\boldsymbol{k}$ in this layer and adding in the next. However, we omit this approach for clarity of presentation.

C.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 (Eq. (15)) for layer i is:

$$\boldsymbol{h}_t = \sigma\left(\begin{bmatrix} \boldsymbol{W}_{\phi_i} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{W}_{\gamma_i} \end{bmatrix} \begin{bmatrix} \boldsymbol{a}_{1,t} \\ \boldsymbol{a}_{2,t} \end{bmatrix} + \begin{bmatrix} \boldsymbol{b}_{\phi_i} \\ \boldsymbol{b}_{\gamma_i} \end{bmatrix}\right) = \begin{bmatrix} \phi_i(\boldsymbol{a}_{1,t}) \\ \gamma_i(\boldsymbol{a}_{1,t}) \end{bmatrix}. \tag{18}$$

Here, ϕ_i and γ_i are the i-th layers (including the `ReLU`) of respectively ϕ and γ in Eq. (5).

C.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 Eq. (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 $\boldsymbol{h}_t$ (Eq. (15)) of this GRU layer becomes:

$$\boldsymbol{h}_t = \sigma\left(\begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \boldsymbol{I} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \boldsymbol{a}_{1,t} \\ \boldsymbol{a}_{2,t} \end{bmatrix}\right) \odot \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \boldsymbol{a}_{1,t} \\ \boldsymbol{a}_{2,t} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \sigma(\boldsymbol{a}_{1,t}) \odot \boldsymbol{a}_{2,t} \end{bmatrix}. \tag{19}$$

If $\boldsymbol{a}_t$ is the output of a GRU layer constructed as in Eq. (18) (as is in our case), then it must be non-negative. This is due to the `ReLU` application in Eq. (18). Hence, the application of another `ReLU` to $\boldsymbol{a}_{1,t}$ in Eq. (19) can be safely removed as `ReLU` is idempotent and Eq. (19) simplifies to

$$\boldsymbol{h}_t = \begin{bmatrix} \mathbf{0} \\ \boldsymbol{a}_{1,t} \odot \boldsymbol{a}_{2,t} \end{bmatrix}. \tag{20}$$

Thus, this construction computes element-wise multiplication of $\boldsymbol{a}_{1,t}$ and $\boldsymbol{a}_{2,t}$.

C.4 Composing the operations to model a single Gated RNN layer

In order to represent Eq. (5), we use one GRU layer for the recurrence (as described in App. C.1), followed by k GRU layers modelling a pair of the k MLP layers of ϕ and γ (App. C.2), completed with a single mixing layer (App. C.3). This stack of $k + 2$ layers models exactly the Gated RNN layer (Eq. (5)):

$$\begin{aligned} \boldsymbol{s}_t &= \sigma\left(\boldsymbol{A} \begin{bmatrix} \mathbf{0} \\ \boldsymbol{s}_{t-1} \end{bmatrix} + \boldsymbol{B} \begin{bmatrix} \boldsymbol{x}_t \\ \mathbf{0} \end{bmatrix} + \boldsymbol{b}\right) \\ \boldsymbol{y}_t &= \begin{bmatrix} \mathbf{0} \\ \gamma(\boldsymbol{x}_t) \odot \phi(\boldsymbol{s}_t) \end{bmatrix}, \end{aligned}$$

With this, we have shown that any Gated RNN (Eq. (5)) can be expressed as a GRU-based model. Hence, the two universal approximation programs in Lsts. 1 and 2 can be implemented also in GRU-based models. Thus, the GRU architecture can also be a universal in-context approximator.

D Gated RNNs are LSTMs

A single LSTM layer (Hochreiter and Schmidhuber, 1997; Gers et al., 2000) with input $\boldsymbol{a}_t \in \mathbb{R}^{d_{\text{in}}}$, hidden state $\boldsymbol{h}_{t-1} \in \mathbb{R}^{d_{\text{hidden}}}$, candidate memory cell $\tilde{\boldsymbol{c}}_t \in \mathbb{R}^{d_{\text{hidden}}}$, memory cell $\boldsymbol{c}_t \in \mathbb{R}^{d_{\text{hidden}}}$ and layer output $\boldsymbol{h}_t \in \mathbb{R}^{d_{\texttt{hidden}}}$ can be expressed as:

$$\boldsymbol{f}_t = \texttt{Sigmoid}(\boldsymbol{W}_f \boldsymbol{a}_t + \boldsymbol{U}_f \boldsymbol{h}_{t-1} + \boldsymbol{b}_f), \quad \text{(forget gate vector)} \tag{21}$$
$$\boldsymbol{i}_t = \texttt{Sigmoid}(\boldsymbol{W}_i \boldsymbol{a}_t + \boldsymbol{U}_i \boldsymbol{h}_{t-1} + \boldsymbol{b}_i), \quad \text{(input gate vector)} \tag{22}$$
$$\boldsymbol{o}_t = \texttt{Sigmoid}(\boldsymbol{W}_o \boldsymbol{a}_t + \boldsymbol{U}_o \boldsymbol{h}_{t-1} + \boldsymbol{b}_o), \quad \text{(output gate vector)} \tag{23}$$
$$\tilde{\boldsymbol{c}}_t = \texttt{tanh}(\boldsymbol{W}_c \boldsymbol{a}_t + \boldsymbol{U}_c \boldsymbol{h}_{t-1} + \boldsymbol{b}_c), \quad \text{(candidate cell vector)} \tag{24}$$
$$\boldsymbol{c}_t = \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \tilde{\boldsymbol{c}}_t, \quad \text{(memory cell vector)} \tag{25}$$
$$\boldsymbol{h}_t = \boldsymbol{o}_t \odot \texttt{tanh}(\boldsymbol{c}_t), \quad \text{(output vector)} \tag{26}$$

where $\boldsymbol{h}_0 = \mathbf{0}$ and $\boldsymbol{c}_0 = \mathbf{0}$.

In a way analogous to App. C, we show that a single layer of a gated RNN (Eq. (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 $d_{\texttt{in}} = d_{\texttt{hidden}}$. The set up follows the same structure as in App. C. 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 $i \in [d_{\texttt{in}}]$ for all $\boldsymbol{x}_t \in \mathcal{X}$. In other words, we have $(\boldsymbol{x}_t - \boldsymbol{k}_{lb})_i \geq 0$ for for $i \in 1, \ldots, d_{\texttt{in}}$. We will make use of this fact several times when dealing with `ReLU` activations.

D.1 Representing the state update as an LSTM layer

We first represent the non-linear state update in Eq. (5) using a single layer of an LSTM. In particular, we set $\boldsymbol{W}_f = \mathbf{0}$, $\boldsymbol{U}_f = \mathbf{0}$ and $\boldsymbol{b}_f = \mathbf{0}$ so that $\boldsymbol{f}_t = \mathbf{0}$. We also set $\boldsymbol{W}_i = \mathbf{0}$, $\boldsymbol{U}_i = \mathbf{0}$, $\boldsymbol{b}_i = \mathbf{1}$ and $\boldsymbol{W}_c = \mathbf{0}$, $\boldsymbol{U}_c = \mathbf{0}$, $\boldsymbol{b}_c = \mathbf{1}$. This results in $\boldsymbol{i}_t = \mathbf{1}$ and $\tilde{\boldsymbol{c}}_t = \mathbf{1}$. We see from this that the LSTM layer with these weight settings reduces to

$$\boldsymbol{h}_t = \boldsymbol{o}_t = \sigma(\boldsymbol{W}_o \boldsymbol{a}_t + \boldsymbol{U}_o \boldsymbol{h}_{t-1} + \boldsymbol{b}_o). \tag{27}$$

We now set $\boldsymbol{a}_t = \begin{bmatrix} \mathbf{0} \\ \boldsymbol{x}_t \end{bmatrix}$, where $\boldsymbol{x}_t \in \mathbb{R}^{d_{\texttt{in}}/2}$, $\boldsymbol{h}_{t-1} = \begin{bmatrix} \boldsymbol{s}_{t-1} \\ \mathbf{0} \end{bmatrix}$, where $\boldsymbol{s}_{t-1} \in \mathbb{R}^{d_{\texttt{hidden}}/2}$, $\boldsymbol{W}_o = \begin{bmatrix} \mathbf{0} & \boldsymbol{B} \\ \mathbf{0} & \boldsymbol{I} \end{bmatrix}$, $\boldsymbol{U}_o = \begin{bmatrix} \boldsymbol{A} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}$, $\boldsymbol{b}_o = \begin{bmatrix} \boldsymbol{b} \\ -\boldsymbol{k}_{lb} \end{bmatrix}$ so that

$$\boldsymbol{h}_t = \sigma\left(\begin{bmatrix} \mathbf{0} & \boldsymbol{B} \\ \mathbf{0} & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \mathbf{0} \\ \boldsymbol{x}_t \end{bmatrix} + \begin{bmatrix} \boldsymbol{A} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \boldsymbol{s}_{t-1} \\ \mathbf{0} \end{bmatrix} + \begin{bmatrix} \boldsymbol{b} \\ -\boldsymbol{k}_{lb} \end{bmatrix}\right) = \begin{bmatrix} \sigma(\boldsymbol{A}\boldsymbol{s}_{t-1} + \boldsymbol{B}\boldsymbol{x}_t + \boldsymbol{b}) \\ \sigma(\boldsymbol{x}_t - \boldsymbol{k}_{lb}) \end{bmatrix} = \begin{bmatrix} \boldsymbol{s}_t \\ \boldsymbol{x}_t - \boldsymbol{k}_{lb} \end{bmatrix}. \tag{28}$$

D.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 $\boldsymbol{W}_f = \mathbf{0}$, $\boldsymbol{U}_f = \mathbf{0}$, $\boldsymbol{b}_f = \mathbf{0}$ and $\boldsymbol{W}_i = \mathbf{0}$, $\boldsymbol{U}_i = \mathbf{0}$, $\boldsymbol{b}_i = \mathbf{1}$ and $\boldsymbol{W}_c = \mathbf{0}$, $\boldsymbol{U}_c = \mathbf{0}$, $\boldsymbol{b}_c = \mathbf{1}$ as before. We make a change for these LSTM layers by setting $\boldsymbol{W}_o = \begin{bmatrix} \boldsymbol{W}_{\phi_i} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{W}_{\gamma_i} \end{bmatrix}$, $\boldsymbol{U}_o = \mathbf{0}$ 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:

$$\boldsymbol{h}_t = \sigma\left(\begin{bmatrix} \boldsymbol{W}_{\phi_i} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{W}_{\gamma_i} \end{bmatrix} \begin{bmatrix} \boldsymbol{a}_{1,t} \\ \boldsymbol{a}_{2,t} \end{bmatrix} + \begin{bmatrix} \boldsymbol{b}_{\phi_i} \\ \boldsymbol{b}_{\gamma_i} \end{bmatrix}\right) = \begin{bmatrix} \phi_i(\boldsymbol{a}_{1,t}) \\ \gamma_i(\boldsymbol{a}_{2,t}) \end{bmatrix}. \tag{29}$$

Here, ϕ_i and γ_i again refer to the i-th layers (including the `ReLU`) of respectively ϕ and γ in Eq. (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 $\mathbf{1}$ 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

$$\boldsymbol{o}_t = \sigma\left(\begin{bmatrix} \boldsymbol{I} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{W}_{\gamma_i} \end{bmatrix} \begin{bmatrix} \phi(\boldsymbol{s}_t) \\ \boldsymbol{a}_{2,t} \end{bmatrix} + \begin{bmatrix} \mathbf{0} \\ \boldsymbol{b}_{\phi_i} \end{bmatrix}\right) = \begin{bmatrix} \phi(\boldsymbol{s}_t) \\ \gamma_i(\boldsymbol{a}_{2,t}) \end{bmatrix}. \tag{30}$$

D.3 Representing the multiplicative gating with an LSTM layer

Finally, we model the element-wise multiplication of the outputs of ϕ and γ in Eq. (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:

$$\boldsymbol{i}_t = \sigma\left(\begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \boldsymbol{I} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \boldsymbol{a}_{1,t} \\ \boldsymbol{a}_{2,t} \end{bmatrix} + \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \end{bmatrix}\right) = \begin{bmatrix} \mathbf{0} \\ \boldsymbol{a}_{1,t} \end{bmatrix} \tag{31}$$

and

$$\tilde{\boldsymbol{c}} = \sigma\left(\begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \boldsymbol{a}_{1,t} \\ \boldsymbol{a}_{2,t} \end{bmatrix} + \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \end{bmatrix}\right) = \begin{bmatrix} \mathbf{0} \\ \boldsymbol{a}_{2,t} \end{bmatrix}. \tag{32}$$

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

$$\boldsymbol{y}_t = \sigma(\boldsymbol{c}_t) = \begin{bmatrix} \sigma(\mathbf{0} \odot \mathbf{0}) \\ \sigma(\boldsymbol{a}_{1,t} \odot \boldsymbol{a}_{2,t}) \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \sigma(\boldsymbol{a}_{1,t} \odot \boldsymbol{a}_{2,t}) \end{bmatrix}. \tag{33}$$

D.4 Composing the operations to model a single Gated RNN layer

To model the gated RNN described in Eq. (5), we again follow the same lines as described in App. C. In particular, we use one LSTM layer for the recurrent state updated as described in App. D.1. We then stack k LSTM layers as described in App. D.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 App. D.3 in order to match the output of the gated RNN in Eq. (5). Now, since we are working with $\sigma = \text{ReLU}$, 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. (33) leaves the the components invariant. Therefore, we output is

$$\boldsymbol{y}_t = \begin{bmatrix} \mathbf{0} \\ \gamma(\boldsymbol{x}_t) \odot \phi(\boldsymbol{s}_t) \end{bmatrix}, \tag{34}$$

as required.

Hence, we have shown that a single layer of a gated RNN as described by Eq. (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 Lsts. 1 and 2 can also be implemented for LSTMs. Hence, LSTM models are also universal approximators in the sense described in Sec. 4.

E 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 $\boldsymbol{a}_t \in \mathbb{R}^{d_{\mathtt{in}}}$, inputs to the blocks of dimensions $d_{\mathtt{in}}$ and outputs from each block of dimensions $d_{\mathtt{in}}$. Within blocks, all vectors have dimensionality $d_{\mathtt{hidden}} = Ed_{\mathtt{in}}$, 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 $\boldsymbol{a}_t$. The one-dimensional temporal convolution can be written as:

$$\boldsymbol{a}'_t = \boldsymbol{W}_a \boldsymbol{a}_t, \tag{35}$$

$$\boldsymbol{g}_t = \texttt{GeLU}(\boldsymbol{W}_g \boldsymbol{a}_t + \boldsymbol{b}_g), \tag{36}$$

$$\boldsymbol{M}_t = \left[\boldsymbol{a}'_{t-(d_{\text{conv}}-1)}, \ldots, \boldsymbol{a}'_{t-2}, \boldsymbol{a}'_{t-1}, \boldsymbol{a}'_t\right], \tag{37}$$

$$\boldsymbol{z}_t = \sum_{i=0}^{d_{\text{conv}}-1} \boldsymbol{W}_M[i]\boldsymbol{M}_t[t-i] \; + \boldsymbol{b}_{\text{conv}} \qquad \text{(convolution with window size } d_{\text{conv}}\text{)}, \tag{38}$$

where $\boldsymbol{b}_{\text{conv}}$ is a bias vector and $\boldsymbol{W}_M = \left[\tilde{\boldsymbol{B}}, \tilde{\boldsymbol{A}}\tilde{\boldsymbol{B}}, \tilde{\boldsymbol{A}}^2\tilde{\boldsymbol{B}}, \cdots, \tilde{\boldsymbol{A}}^t\tilde{\boldsymbol{B}}, \cdots\right]$ 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 $\boldsymbol{z}_t$ from the one-dimensional convolution and with recurrent state $\boldsymbol{h}_t \in \mathbb{R}^{d_{\texttt{model}}}$:

$$\boldsymbol{r}_t = \texttt{Sigmoid}(\boldsymbol{W}_r \boldsymbol{z}_t + \boldsymbol{b}_r), \tag{39}$$

$$\boldsymbol{i}_t = \texttt{Sigmoid}(\boldsymbol{W}_i \boldsymbol{z}_t + \boldsymbol{b}_i), \tag{40}$$

$$a = \texttt{Sigmoid}(\Lambda), \qquad (\Lambda \text{ a learnable parameter}) \tag{41}$$

$$\boldsymbol{a}_t = a^{c\boldsymbol{r}_t}, \qquad (c = 8 \text{ fixed scalar constant}) \tag{42}$$

$$\boldsymbol{h}_t = \boldsymbol{a}_t \odot \boldsymbol{h}_{t-1} + \sqrt{1 - \boldsymbol{a}_t^2} \odot (\boldsymbol{i}_t \odot \boldsymbol{z}_t). \tag{43}$$

Now consider the second branch of the recurrent block. This performs a linear transformation followed by a non-linear activation:

$$\boldsymbol{g}_t = \texttt{GeLU}(\boldsymbol{W}_g \boldsymbol{a}_t + \boldsymbol{b}_g). \tag{44}$$

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:

$$\boldsymbol{h}'_t = \boldsymbol{g}_t \odot \boldsymbol{h}_t, \tag{45}$$

$$\boldsymbol{o}_t = \boldsymbol{W}_o \boldsymbol{h}'_t + \boldsymbol{b}_o. \tag{46}$$

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

$$\boldsymbol{e}_t = \boldsymbol{W}_e \boldsymbol{o}_t + \boldsymbol{b}_e, \tag{47}$$

and the second performs a single layer MLP transformation as

$$\boldsymbol{f}_t = \texttt{GeLU}(\boldsymbol{W}_f \boldsymbol{o}_t + \boldsymbol{b}_f). \tag{48}$$

These are then combined through a Hadamard product and linear transformation as

$$\boldsymbol{e}'_t = \boldsymbol{e}_t \odot \boldsymbol{f}_t, \tag{49}$$

$$\boldsymbol{m}_t = \boldsymbol{W}_m \boldsymbol{e}'_t + \boldsymbol{b}_m. \tag{50}$$

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.[4] 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.

[4]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.

E.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 $\boldsymbol{a}_t = \begin{bmatrix} \mathbf{0} \\ \boldsymbol{x}_t \end{bmatrix}$. First, we set $\boldsymbol{W}_a = \boldsymbol{I}$ so that $\boldsymbol{a}'_t = \boldsymbol{a}_t$. Next we choose matrices $\tilde{\boldsymbol{A}} = \begin{bmatrix} \mathbf{0} & \boldsymbol{A} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}$ and $\tilde{\boldsymbol{B}} = \begin{bmatrix} \mathbf{0} & \boldsymbol{B} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}$ which we then use, 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 $\boldsymbol{b}_{\text{conv}} = \begin{bmatrix} \mathbf{0} \\ \mathbf{1} \end{bmatrix}$ gives

$$\boldsymbol{z}_t = \sum_{i=0}^{t-1} \boldsymbol{W}_M[i]\boldsymbol{M}_t[t-i] \quad + \boldsymbol{b}_{\text{conv}}, \tag{51}$$

$$= \tilde{\boldsymbol{B}}\boldsymbol{a}_t + \tilde{\boldsymbol{A}}\tilde{\boldsymbol{B}}\boldsymbol{a}_{t-1} + \cdots + \tilde{\boldsymbol{A}}^{t-1}\tilde{\boldsymbol{B}}\boldsymbol{a}_1 + \begin{bmatrix} \mathbf{0} \\ \mathbf{1} \end{bmatrix} \tag{52}$$

$$= \begin{bmatrix} \boldsymbol{s}_t \\ \mathbf{1} \end{bmatrix}. \tag{53}$$

Now, we pass $\boldsymbol{z}_t$ through the RG-LRU. We set $\Lambda = 0$ so that $\boldsymbol{a}_t = 0$. We also define $\boldsymbol{W}_i = \mathbf{0}$ and $\boldsymbol{b}_i = \mathbf{1}$ so that $\boldsymbol{i}_t = \mathbf{1}$. 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 $\boldsymbol{b}_g = \begin{bmatrix} \mathbf{1} \\ -\boldsymbol{k}_{lb} \end{bmatrix}$ so that

$$\boldsymbol{g}_t = \sigma\left(\boldsymbol{I}\begin{bmatrix} \mathbf{0} \\ \boldsymbol{x}_t \end{bmatrix} + \begin{bmatrix} \mathbf{1} \\ -\boldsymbol{k}_{lb} \end{bmatrix}\right) = \begin{bmatrix} \sigma(\mathbf{1}) \\ \sigma(\boldsymbol{x}_t - \boldsymbol{k}_{lb}) \end{bmatrix} = \begin{bmatrix} \mathbf{1} \\ \boldsymbol{x}_t - \boldsymbol{k}_{lb} \end{bmatrix}, \tag{54}$$

where we used that $(\boldsymbol{x}_t - \boldsymbol{k}_{lb})_i \geq \mathbf{0}$ for every i. Combining the two branches gives

$$\boldsymbol{h}'_t = \begin{bmatrix} \mathbf{1} \\ \boldsymbol{x}_t - \boldsymbol{k}_{lb} \end{bmatrix} \odot \begin{bmatrix} \boldsymbol{s}_t \\ \mathbf{1} \end{bmatrix} = \begin{bmatrix} \boldsymbol{s}_t \\ \boldsymbol{x}_t - \boldsymbol{k}_{lb} \end{bmatrix}. \tag{55}$$

We finally get the output of the recurrent block by defining $\boldsymbol{W}_o = \boldsymbol{I}$ and $\boldsymbol{b}_0 = \begin{bmatrix} \mathbf{0} \\ \boldsymbol{k}_{lb} \end{bmatrix}$ so that

$$\boldsymbol{o}_t = \begin{bmatrix} \boldsymbol{s}_t \\ \boldsymbol{x}_t \end{bmatrix}. \tag{56}$$

E.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 $\tilde{\boldsymbol{A}} = \mathbf{0}$ 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 $\boldsymbol{b}_{\text{conv}} = \mathbf{0}$ results in $\boldsymbol{z}_t = \boldsymbol{a}_t$. From here, we can again set $\Lambda = 0$, $\boldsymbol{W}_i = \mathbf{0}$ and $\boldsymbol{b}_i = \mathbf{1}$ so that $\boldsymbol{h}_t = \boldsymbol{z}_t$. Looking at the second branch and setting $\boldsymbol{W}_g = 0$ and $\boldsymbol{b}_g = \mathbf{1}$ 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 $\boldsymbol{b}_o = \mathbf{0}$ so that $\boldsymbol{o}_t = \boldsymbol{h}_t$ which means that $\boldsymbol{o}_t = \boldsymbol{a}_t$.

E.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. (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

$\boldsymbol{b}_e = \mathbf{1}$ so that $\boldsymbol{e}_t = \mathbf{1}$. On the second non-linear branch, we can define $\boldsymbol{W}_f = \begin{bmatrix} \boldsymbol{W}_{\phi_i} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{W}_{\gamma_i} \end{bmatrix}$ and $\boldsymbol{b}_f = \begin{bmatrix} \boldsymbol{b}_{\phi_i} \\ \boldsymbol{b}_{\gamma_i} \end{bmatrix}$. This results in

$$\boldsymbol{f}_t = \sigma\left(\begin{bmatrix} \boldsymbol{W}_{\phi_i} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{W}_{\gamma_i} \end{bmatrix}\begin{bmatrix} \boldsymbol{a}_{1,t} \\ \boldsymbol{a}_{2,t} \end{bmatrix} + \begin{bmatrix} \boldsymbol{b}_{\phi_i} \\ \boldsymbol{b}_{\gamma_i} \end{bmatrix}\right) = \begin{bmatrix} \phi_i(\boldsymbol{a}_{1,t}) \\ \gamma_i(\boldsymbol{a}_{2,t}) \end{bmatrix}. \tag{57}$$

Due to our setting of $\boldsymbol{e}_t$, we get $\boldsymbol{e}'_t = \boldsymbol{f}_t$. Further, defining $\boldsymbol{W}_m = \boldsymbol{I}$ and $\boldsymbol{b}_m = \mathbf{0}$ makes the output of the MLP block be

$$\boldsymbol{m}_t = \begin{bmatrix} \phi_i(\boldsymbol{a}_{1,t}) \\ \gamma_i(\boldsymbol{a}_{2,t}) \end{bmatrix}. \tag{58}$$

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 $\boldsymbol{b}_e = \mathbf{1}$ so that $\boldsymbol{e}_t = \mathbf{1}$. Now we modify the weights on the second non-linear branch by defining $\boldsymbol{W}_f = \begin{bmatrix} \boldsymbol{I} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{W}_{\gamma_i} \end{bmatrix}$ and $\boldsymbol{b}_f = \begin{bmatrix} \mathbf{0} \\ \boldsymbol{b}_{\gamma_i} \end{bmatrix}$. This gives us

$$\boldsymbol{f}_t = \sigma\left(\begin{bmatrix} \boldsymbol{I} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{W}_{\gamma_i} \end{bmatrix}\begin{bmatrix} \phi(\boldsymbol{x}_t) \\ \boldsymbol{a}_{2,t} \end{bmatrix} + \begin{bmatrix} \mathbf{0} \\ \boldsymbol{b}_{\gamma_i} \end{bmatrix}\right) = \begin{bmatrix} \sigma(\phi(\boldsymbol{x}_t)) \\ \gamma_i(\boldsymbol{a}_{2,t}) \end{bmatrix} = \begin{bmatrix} \phi(\boldsymbol{x}_t) \\ \gamma_i(\boldsymbol{a}_{2,t}) \end{bmatrix}, \tag{59}$$

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.

E.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 $\boldsymbol{W}_f = \mathbf{0}, \boldsymbol{b}_f = \mathbf{1}, \boldsymbol{W}_e = \boldsymbol{I}, \boldsymbol{b}_e = \mathbf{0}, \boldsymbol{W}_m = \boldsymbol{I}$ and $\boldsymbol{b}_m = \mathbf{0}$. 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.

E.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. (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 $\boldsymbol{W}_e = \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \boldsymbol{I} & \mathbf{0} \end{bmatrix}$ and $\boldsymbol{b}_e = \mathbf{0}$, so that

$$\boldsymbol{e}_t = \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \boldsymbol{I} & \mathbf{0} \end{bmatrix}\begin{bmatrix} \boldsymbol{a}_{1,t} \\ \boldsymbol{a}_{2,t} \end{bmatrix} + \mathbf{0} = \begin{bmatrix} \mathbf{0} \\ \boldsymbol{a}_{1,t} \end{bmatrix}. \tag{60}$$

Next, we define $\boldsymbol{W}_f = \boldsymbol{I}$ and $\boldsymbol{b}_e = \mathbf{0}$ so that

$$\boldsymbol{f}_t = \begin{bmatrix} \sigma(\boldsymbol{a}_{1,t}) \\ \sigma(\boldsymbol{a}_{2,t}) \end{bmatrix}. \tag{61}$$

Setting $\boldsymbol{W}_m = \boldsymbol{I}$ and $\boldsymbol{b}_m = \mathbf{0}$ gives the output of the gated MLP as

$$\boldsymbol{m}_t = \begin{bmatrix} \mathbf{0} \\ \boldsymbol{a}_{1,t} \odot \sigma(\boldsymbol{a}_{2,t}) \end{bmatrix}. \tag{62}$$

E.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 $\boldsymbol{a}_t = \begin{bmatrix} \mathbf{0} \\ \boldsymbol{x}_t \end{bmatrix}$. We use a residual block that consists of a recurrent block computing the state update as descried in App. E.1 and then a gated MLP block that computes the identity function as demonstrated in App. E.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 App. E.2 followed by MLP blocks that apply the MLP layers of ϕ and γ as described in App. E.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 App. E.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 App. E.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 App. E.2 followed by a gated MLP block that applies multiplicative gating as described in App. E.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

$$\boldsymbol{m}_t = \begin{bmatrix} \mathbf{0} \\ \gamma(\boldsymbol{s}_t) \odot \phi(\boldsymbol{x}_t) \end{bmatrix}. \tag{63}$$

Hence, we have shown that a single layer of a gated RNN as described by Eq. (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 Lsts. 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 Sec. 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 Lsts. 1 and 2 can also be implemented as Griffin models. Therefore, Griffin models can also be universal approximators in the sense described in Sec. 4.

F Definitions for some helper functions in LSRL

F.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.

```
not_x = 1 - x
```

F.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 Sec. 3.

```
and_x_y = ReLU(f_step(x, mu) + f_step(y, mu) - 1)
```

F.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 Sec. 3.

```
or_x_y = f_step(x + y, mu=mu)
```

F.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 Sec. 3.

```
smaller_x_y = f_step(y - x, mu=mu)
```

F.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 Sec. 3.

```
larger_x_y = f_step(x - y, mu=mu)
```

F.6 `f_relu_identity`

Identity operation using `ReLU`s. This is useful for debranching when some of the branches have `ReLU`s but the other don't. We can add this as a bypass for the ones that do not and can then merge the `ReLU`s together (see App. A 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),
)
```

F.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)
)
```

NeurIPS Paper Checklist

i. **Claims**

Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

Answer: [Yes]

Justification: The abstract and the introduction clearly state all the contributions of the paper and clearly differentiate the theoretical results which hold in general and the empirical phenomena that we observe, which may not generalize to all settings.

Guidelines:

- The answer NA means that the abstract and introduction do not include the claims made in the paper.
- The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
- The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
- It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

ii. **Limitations**

Question: Does the paper discuss the limitations of the work performed by the authors?

Answer: [Yes]

Justification: The paper discusses the limitations of the present work. There is a dedicated *Limitations* section in Sec. 7 that addresses the fact that we only provide constructive existence results but not necessary and sufficient conditions for universal in-context approximation to arise. We also highlight that our results might not hold to models with structural constraints on their parameters. Moreover, we have a dedicated section (Sec. 5) which addresses some of the limitations of constructing universal in-context approximators with fully recurrent architectures in practice. This section proposes solutions and demonstrates that they result in more numerically stable models which are more likely to occur in practice.

Guidelines:

- The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
- The authors are encouraged to create a separate "Limitations" section in their paper.
- The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
- The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
- The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
- The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
- If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
- While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best

judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

iii. **Theory Assumptions and Proofs**

Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [Yes]

Justification: The paper has two main theoretical results: the constructions of universal in-context approximators for continuous and for discrete functions. Both results are presented as LSRL programs which compile to the architectures considered in this work. Furthermore, these programs have been implemented in Python, their correctness has been tested and they are available in the supplementary materials.

Guidelines:

- The answer NA means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

iv. **Experimental Result Reproducibility**

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: There are two experimental aspects to this work. First, there is the implementation of LSRL and the two universal approximation programs in Lsts. 1 and 2. The most critical aspect of implementing LSRL is the debranching algorithm which is described in detail in App. A. Additionally, the two programs are described in full in their corresponding listings. We also provide Python implementation for the LSRL compiler and the two programs.

Second, there is the study of how affected by parameter noise are the different implementations of the conditional assignment operator `f_ifelse` which was presented in Sec. 6. The details of this experiment are described in Fig. 4 and we also provide the code with which we did the experiment and our plots.

Guidelines:

- The answer NA means that the paper does not include experiments.
- If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.

- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

v. **Open access to data and code**

Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: We are providing code that includes an implementation of LSRL, the two universal in-context approximation programs in Lsts. 1 and 2 and everything needed to reproduce the experiments in this work.

Guidelines:

- The answer NA means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

vi. **Experimental Setting/Details**

Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

Answer: [NA]

Justification: The experiments in our work are based on *constructed* models rather than *trained* models. Therefore, considerations such as dataset, optimizers and hyperparameters do not apply.

Guidelines:

- The answer NA means that the paper does not include experiments.

- The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
- The full details can be provided either with the code, in appendix, or as supplemental material.

vii. **Experiment Statistical Significance**

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [No]

Justification: For this work, uncertainty quantification could only make sense in the context of Fig. 4. However, the behaviour we observe, especially for the continuous case, is bimodal. As bimodal distributions cannot be properly captured with error bars we decided against using them. Furthermore, we are studying whether a phenomenon occurs, rather than quantifying it. Therefore, we decided to instead use a strip plot instead as it explicitly shows all our results unabridged, clearly indicates the bimodal nature of the results, and distinctly showcases the noise robustness trends of the different approaches we consider.

Guidelines:

- The answer NA means that the paper does not include experiments.
- The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

viii. **Experiments Compute Resources**

Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

Answer: [No]

Justification: Our experiments were ran on a single machine and using only CPU compute. Therefore, the compute required is negligible for the contemporary machine learning standards.

Guidelines:

- The answer NA means that the paper does not include experiments.
- The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
- The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
- The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

ix. **Code Of Ethics**

Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics https://neurips.cc/public/EthicsGuidelines?

Answer: [Yes]

Justification: This is a theoretical work with no human participants, datasets, or potential societal impact or harmful consequences. Therefore, the present work has no moral or ethical relevance or implications.

Guidelines:

- The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
- If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
- The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

x. **Broader Impacts**

Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

Answer: [NA]

Justification: As mentioned above, this is a theoretical work which establishes theoretical properties of mathematical objects that are already used in practice. However, we do discuss the implications of our findings, namely that if models are universal in-context approximators, then it might be difficult to ensure that they are aligned and cannot be misused. Nevertheless, we only show that this is a property already present in existing models, and hence our work does not introduce new attack or misuse vectors. On the contrary, we hope that us highlighting this issues will help the community to develop safer and more secure generative AI systems.

Guidelines:

- The answer NA means that there is no societal impact of the work performed.
- If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

xi. **Safeguards**

Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

Answer: [NA]

Justification: We release no data or models.

Guidelines:

- The answer NA means that the paper poses no such risks.
- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

xii. **Licenses for existing assets**

Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

Answer: [NA]

Justification: The paper does not use existing assets beyond common Python libraries.

Guidelines:

- The answer NA means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, paperswithcode.com/datasets has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

xiii. **New Assets**

Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

Answer: [Yes]

Justification: The only new asset arising from this work is the LSRL code base which we have ensured to be well-documented, accompanied by unit and integration tests, and with illustrative Jupyter notebooks.

Guidelines:

- The answer NA means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

xiv. **Crowdsourcing and Research with Human Subjects**

Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [NA]

Justification: There were no human participants involved in any part of this work.

Guidelines:

- The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

xv. **Institutional Review Board (IRB) Approvals or Equivalent for Research with Human Subjects**

Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer: [NA]

Justification: This is a purely theoretical work and as such no IRB approval or equivalent was necessary or appropriate.

Guidelines:

- The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.