Extended Abstract Track

Learning Useful Representations of Recurrent Neural Network Weight Matrices

Abstract

Recurrent Neural Networks (RNNs) are general-purpose parallel-sequential computers. The program of an RNN is its weight matrix. Its direct analysis, however, tends to be challenging. Is it possible to learn useful representations of RNN weights that facilitate downstream tasks? While the "Mechanistic Approach" directly 'looks inside' the RNN to predict its behavior, the "Functionalist Approach" analyzes its overall functionality—specifically, its input-output mapping. Our two novel Functionalist Approaches extract information from RNN weights by 'interrogating' the RNN through probing inputs. Our novel theoretical framework for the Functionalist Approach demonstrates conditions under which it can generate rich representations for determining the behavior of RNNs. RNN weight representations generated by Mechanistic and Functionalist approaches are compared by evaluating them in two downstream tasks. Our results show the superiority of Functionalist methods.

Keywords: Recurrent Neural Networks, Representation Learning

1. Introduction

For decades, researchers have developed techniques for learning representations in deep neural networks (NNs). This expertise has significantly advanced the field by enabling models to convert data into useful formats for solving problems. In particular, Recurrent NNs (RNNs) have been widely adopted due to their computational universality (18). Low-dimensional representations of the programs of RNNs (their weight matrices) are of great interest as they can speed up the search for solutions to given problems. For instance, compressed RNN representations have been used to evolve RNN parameters (10) for controlling a car from video input (9). However, such representations have employed Fourier-type transforms, e.g., the coefficient of the Discrete Cosine Transform (DCT) (19), and did not use the capabilities of NNs to learn such representations. Recent work has seen a rise of representation learning techniques for NN weights using powerful neural networks as encoders (20; 17; 1; 4). However, there is a lack of methods for learning representations of RNNs. This paper introduces novel techniques for learning them, using powerful NNs which may be RNNs themselves. Just like representation learning in other fields, such as Computer Vision, facilitates solving specific tasks, such techniques can facilitate learning, searching, and planning with RNNs.

2. Self-supervised Learning of Function Representations

We consider a Recurrent Neural Network (RNN), $f_\theta : \mathbb{R}^X \times \mathbb{R}^H \to \mathbb{R}^Y \times \mathbb{R}^H; (x, h_o) \mapsto (y, h_n)$, parametrized by $\theta \in \Theta$, which maps an input x and hidden state h_o to an output y and a new hidden state h_n. The RNN interacts with a potentially stochastic environment, $\mathcal{E}$, that maps an RNN's output y to a new input x. The environment may have its own hidden state η. By sequentially interacting with the environment, the RNN produces a rollout defined by:

$$\begin{cases} x_t, \eta_t = \mathcal{E}(y_{t-1}, \eta_{t-1}) \\ y_t, h_t = f_\theta(x_t, h_{t-1}), \end{cases}$$

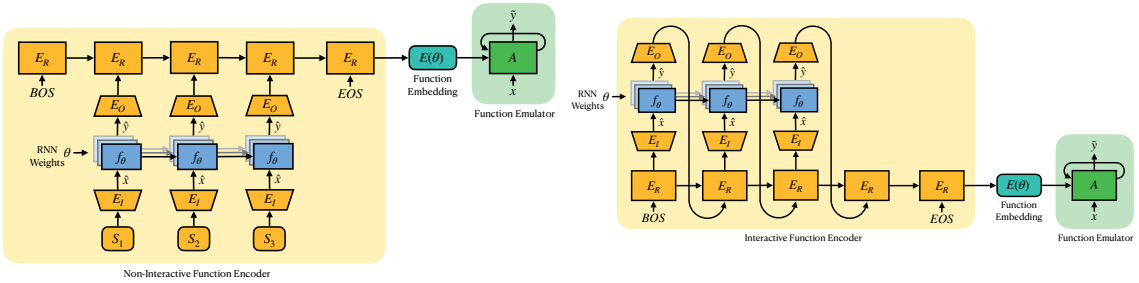

Figure 1: Left: Non-Interactive Encoder. Right: Interactive Encoder.

with fixed initial states y_0, η_0 and h_0. For instance, f_θ might be an autoregressive generative model, with $\mathcal{E}$ acting as a stochastic environment that receives a probability distribution over some language tokens, y_t—the output of f—, and produces a representation (e.g., a one-hot vector) of the new input token x_{t+1}. When the environment is stochastic, numerous rollouts can be generated for any $\theta \in \Theta$. A rollout sequence of a function f_θ in environment $\mathcal{E}$ has the form $S_\theta = (x_1, y_1, x_2, y_2, \dots)$.

Encoder and Emulator Our primary objective is to propose, analyze, and train several methods for representing RNN weights. We define the Encoder $E_\phi : \Theta \to \mathbb{R}^M; \theta \mapsto z$, parametrized by $\phi \in \Phi$ as a function mapping the RNN parameters θ to a lower-dimensional representation z. To train the encoder E_ϕ, we consider an Emulator $A_\xi : \mathbb{R}^X \times \mathbb{R}^B \times \mathbb{R}^Z \to \mathbb{R}^Y \times \mathbb{R}^B; (x, b_o, z) \mapsto (\tilde{y}, b_n)$, parametrized by $\xi \in \Xi$. The Emulator is an RNN with hidden state b that learns to imitate different RNNs f_θ based on their function encoding $z = E(\theta)$.

Dataset and Training We consider a dataset $\mathcal{D} = \{(\theta_i, S_{\theta_i}) | i = 1, 2, \dots\}$ composed of tuples, each containing the parameters of a different RNN and a corresponding rollout sequence. We assume that all RNNs have the same initial state h_0 but have been trained on different tasks. Our self-supervised learning approach to training function representations is inspired by the work of (12)). The Encoder E_ϕ and the Emulator A_ξ are jointly trained by minimizing a loss function $\mathcal{L}$. This loss function measures the behavioral similarity between an RNN f_θ and the Emulator A_ξ, which is conditioned on the function representation $z = E_\phi(\theta)$ of θ as produced by the Encoder E_ϕ. Put simply, the Emulator utilizes the representations of a set of diverse RNNs f_θ to imitate their behavior:

$$\min_{\phi,\xi} \mathbb{E}_{(\theta,S)\sim\mathcal{D}} \sum_{(x_i,y_i)\in S} \mathcal{L}\big(A_\xi(x_i, b_{i-1}, E_\phi(\theta)), y_i\big). \tag{1}$$

In the case of continuous outputs y, the mean-squared error provides a suitable loss function. Conversely, for categorical outputs, we employ the inverse Kullback-Leibler divergence.

2.1. RNN Encoders

In this section, we explore various mechanistic and functionalist methods for constructing RNN encoders. These approaches will be compared in our experimental section.

Flattened Weights (Mechanistic) Flattening the weights into a single vector presents the most straightforward method for encoding an RNN. While this technique has shown efficacy on a modest scale (3; 6), it faces challenges when applied to larger parameter vectors, especially in handling weight-space symmetries such as neuron permutations.

Neural Functional (Mechanistic) Fast Weight Programmers (14; 15; 13; 8) are neural networks that can process the gradients or weights of another neural network. A recent variant thereof, called Neural Functionals(11), has been used to learn representations of neural network weights that are invariant to the permutation of hidden neurons. The architecture comprises layers that display equivariance to neuron permutations, followed by a final pooling operation that ensures the invariance property. Neural Functionals have been theoretically proven to be able to extract all information from the weights of a neural network (11). However, their implementation to date has been confined to feedforward networks, such as MLPs and CNNs.

Non-Interactive RNN Probing (Functionalist) In the context of Reinforcement Learning and Markov Decision Processes, policy fingerprinting has emerged as an effective way to evaluate feedforward neural network policies (4; 5; 2). In policy fingerprinting, a set of learnable probing inputs is given to the network. Based on the set of corresponding policy outputs, a function (policy) representation is produced. This approach can be adapted in a straightforward way for RNNs by learning whole probing sequences instead of probing inputs (see Figure 1, left). In the context of this paper, we refer to this approach as non-interactive RNN probing.

Interactive RNN Probing (Functionalist) The probing sequences for non-interactive RNN probing are static, i.e., at test time, the probing sequences do not depend on the specific RNN being evaluated. The alternative is to make the probing sequences dynamically dependent on the given RNN. Each item in the probing sequences should depend on the outputs of the given RNN to the previous items (Figure 1, right). This idea has been described previously to extract arbitrary information from a recurrent world model (16). Our theoretical framework shows that this novel approach, which we call interactive RNN probing, is more powerful than non-interactive probing in certain cases.

3. Experiments

We empirically analyze various approaches to learning representations of RNNs, with a specific focus on LSTM (7) weights. Each LSTM in our dataset serves as an autoregressive generative model of a specific formal language. The set of formal languages is identical to the one constructed for the proof of Proposition 4. Each LSTM is trained on strings from a particular language using the standard language modelling objective. We split the dataset into training, validation, and out-of-distribution (OOD) test parts. The OOD split includes only tasks in which the relative frequencies of each token appearance are small (i.e., all tokens appear approximately the same number of times). The validation set is utilized for early stopping during training. All shown results are derived from the test set. The experiments employ the four types of function encoders described in Section 2.1. The encoders' hyperparameters are selected to ensure a comparable number of parameters among them. The training details remain consistent across all runs (further details are available

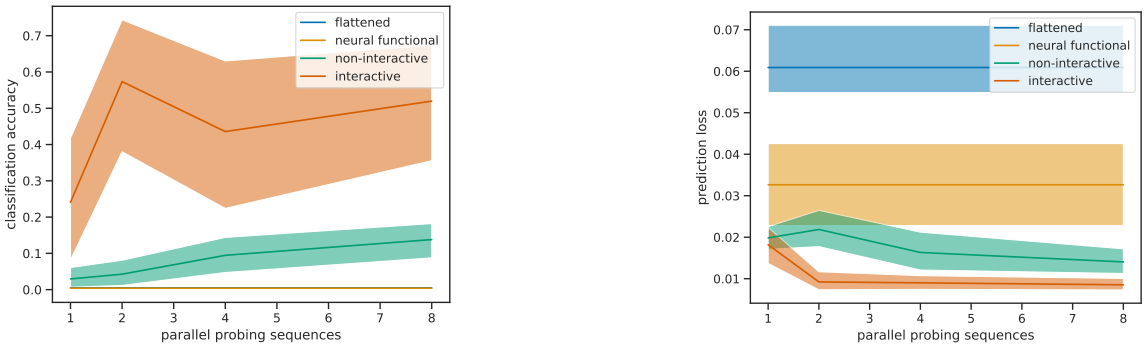

Figure 2: Left: Accuracy of a language classifier, trained using the generated function encodings. Right: Loss of a performance predictor, also trained on the generated function encodings, depicted on the test set. Plots are presented as a function of the number of parallel probing sequences (only relevant for interactive and non-interactive probing encoders). Both graphs display the mean and bootstrapped 95% confidence intervals, aggregated across 15 seeds.

in Appendix B). All encoders are trained end-to-end together with an LSTM emulator to minimize the loss defined in Equation 1, utilizing the reverse Kullback-Leibler divergence as the loss function $\mathcal{L}$. The objective is to ensure that the LSTM weight encodings z serve as generally useful representations. We verify this by training models for two downstream tasks using the fixed representations provided by the encoder E. The first task involves classifying the language on which an LSTM f_θ was trained, given its encoding $E(\theta)$. This classification is inherently challenging, considering the dataset contains a total of 216 different languages, and some networks are nearly untrained. The second task aims to predict the performance of f_θ, defined as the percentage of strings generated by f_θ belonging to the language that f_θ was trained on. We present the results for these tasks in Figure 2. A visualization of the learned embedding spaces can be found in Figure 8 in the Appendix. From the results, it is evident that the interactive probing encoder yields the most useful representations for both tasks. Having multiple probing sequences in parallel benefits both interactive and non-interactive encoders. The representations derived from the flattened weights and the neural functional encoder appear to contain no useful information for the language classifier. In predicting accuracy, representations from neural functionals outperform those based on flattened weights but fall short when compared to functionalist representations.

4. Conclusion and Future Work

We identified two classes of methods for learning RNN weight representations. Firstly, we adapted the Mechanistic Neural Functional approach to RNNs and, secondly, presented two novel Functionalist methods, theoretically demonstrating when their representations can be utilized to identify RNNs. Functionalist methods outperformed Mechanistic ones, learning more useful RNN weight representations for two downstream tasks. Future work will explore the combination of both approaches and evaluate their performance on more challenging problems.

References

[1] E. Dupont, H. Kim, S. Eslami, D. Rezende, and D. Rosenbaum. From data to functa: Your data point is a function and you can treat it like one. *arXiv preprint arXiv:2201.12204*, 2022.

[2] F. Faccio, V. Herrmann, A. Ramesh, L. Kirsch, and J. Schmidhuber. Goal-conditioned generators of deep policies. *arXiv preprint arXiv:2207.01570*, 2022.

[3] F. Faccio, L. Kirsch, and J. Schmidhuber. Parameter-based value functions. *Preprint arXiv:2006.09226*, 2020.

[4] F. Faccio, A. Ramesh, V. Herrmann, J. Harb, and J. Schmidhuber. General policy evaluation and improvement by learning to identify few but crucial states. *arXiv preprint arXiv:2207.01566*, 2022.

[5] J. Harb, T. Schaul, D. Precup, and P.-L. Bacon. Policy evaluation networks. *arXiv preprint arXiv:2002.11833*, 2020.

[6] V. Herrmann, L. Kirsch, and J. Schmidhuber. Learning one abstract bit at a time through self-invented experiments encoded as neural networks. *arXiv preprint arXiv:2212.14374*, 2022.

[7] S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 1997.

[8] K. Irie, I. Schlag, R. Csordás, and J. Schmidhuber. Going beyond linear transformers with recurrent fast weight programmers. *Advances in Neural Information Processing Systems*, 34:7703–7717, 2021.

[9] J. Koutník, G. Cuccu, J. Schmidhuber, and F. Gomez. Evolving large-scale neural networks for vision-based reinforcement learning. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 1061–1068, 2013.

[10] J. Koutník, F. Gomez, and J. Schmidhuber. Evolving neural networks in compressed weight space. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, pages 619–626, 2010.

[11] A. Navon, A. Shamsian, I. Achituve, E. Fetaya, G. Chechik, and H. Maron. Equivariant architectures for learning in deep weight spaces. *arXiv preprint arXiv:2301.12780*, 2023.

[12] R. Raileanu, M. Goldstein, A. Szlam, and R. Fergus. Fast adaptation via policy-dynamics value functions. *arXiv preprint arXiv:2007.02879*, 2020.

[13] I. Schlag, K. Irie, and J. Schmidhuber. Linear transformers are secretly fast weight programmers. In *International Conference on Machine Learning*, pages 9355–9366. PMLR, 2021.

[14] J. Schmidhuber. Learning to control fast-weight memories: An alternative to dynamic recurrent networks. *Neural Computation*, 4(1):131–139, 1992.

[15] J. Schmidhuber. A 'self-referential'weight matrix. In *International Conference on Artificial Neural Networks*, pages 446–450. Springer, 1993.

[16] J. Schmidhuber. On learning to think: Algorithmic information theory for novel combinations of reinforcement learning controllers and recurrent neural world models. *Preprint arXiv:1511.09249*, 2015.

[17] K. Schürholt, D. Kostadinov, and D. Borth. Hyper-representations: Self-supervised representation learning on neural network weights for model characteristic prediction. 2021.

[18] H. T. Siegelmann and E. D. Sontag. Turing computability with neural nets. *Applied Mathematics Letters*, 4(6):77–80, 1991.

[19] R. K. Srivastava, J. Schmidhuber, and F. Gomez. Generalized compressed network search. In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference companion*, GECCO Companion '12, page 647–648, New York, NY, USA, 2012. ACM, ACM.

[20] T. Unterthiner, D. Keysers, S. Gelly, O. Bousquet, and I. O. Tolstikhin. Predicting neural network accuracy from weights. *ArXiv*, abs/2002.11448, 2020.

[21] A. Zhou, K. Yang, K. Burns, Y. Jiang, S. Sokota, J. Z. Kolter, and C. Finn. Permutation equivariant neural functionals. *arXiv preprint arXiv:2302.14040*, 2023.

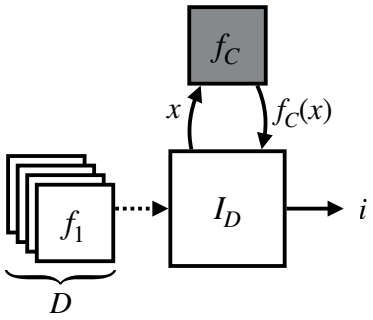

Figure 3: The Interrogator I_D has access to the set of functions D and can interact with one function f_C, which it has to identify.

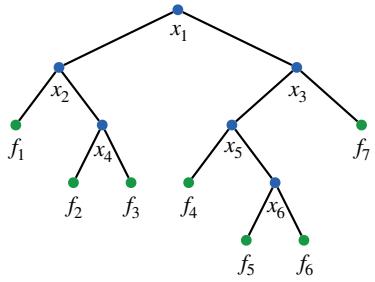

Figure 4: A binary tree constructed as described in the proof of Theorem 2. Giving the inputs x_j corresponding to all branching nodes to a function allows to uniquely identify it.

Appendix A. A Theoretical Framework for the Functionalist Approach

Developing an abstract theoretical model based on the functionalist view of RNN weights provides fundamental insights into the potential and limitations of this approach. The functionalist perspective emphasizes the overall functionality, disregarding the specific underlying mechanisms of RNNs. Therefore, our abstraction adopts the models of total Turing machines as a model of computation. Practically, the function encoder is trained using a dataset of functions. In contrast, the encoder maintains perpetual direct access to the dataset in our theoretical framework. Note that our framework does not incorporate the concept of generalization to unseen functions or networks. We detail our theoretical functionalist framework and explore the interactive and non-interactive approach.

Let D represent a set of n total computable functions $\{f_i : \mathbb{N} \to \mathbb{N} | i = 1, 2, \ldots, n\}$. Here, D comprises n Turing machines that halt on every input, with no pair being functionally equivalent. Let I_D denote another Turing machine, which we call the *Interrogator*. I_D has access to the function set D (e.g., the corresponding Turing numbers might be written somewhere on its tape). Moreover, I_D is given access to one function $f_C \in D$ as a black box. I_D can interact with f_C by providing an input $x \in \mathbb{N}$ and subsequently reading the corresponding output $f_C(x)$. The task of I_D is to identify which member of D corresponds to function f_C, while minimizing interactions with f_C. Specifically, I_D must return $i \in \{1, \ldots, n\}$ such that $f_C = f_i$. This setup is depicted in Figure 3.

Lemma 1 *Given any subset $G \subseteq D, |G| \geq 2$, there exists an input x that can be computed for which $f_a(x) \neq f_b(x)$ with $f_a, f_b \in G$.*

Proof This follows immediately from the fact that all functions in G are total computable and functionally distinct. ■

Proposition 2 *Any function f_C from a set D can be identified by an interrogator through at most $|D| - 1$ interactions.*

Proof According to Lemma 1, it is possible to split any set $G \subseteq D, |G| \geq 2$ into two nonempty, non-overlapping subsets: $G_a := \{f \in G | f(x_j) = f_a(x_j)\}$ and $G_b := \{f \in G | f(x_j) \neq f_a(x_j)\}$ for some $f_a \in G$ and $x_j \in \mathbb{N}$. Any resulting subset that has at least two members can be split again using the same procedure with a different probing input x_{j+1}. Starting from the full set D, it is possible to construct a binary tree (see Figure 4) where the leaves are subsets of D containing exactly one uniquely identified function. The branching (i.e., non-leaf) nodes correspond to the splitting operation, which involves observing the output of a specific probing input x_j.

The Interrogator can identify a given function $f_C \in D$ by providing it with all inputs x_j corresponding to the branching nodes in the binary tree and observing the outputs. Since any binary tree with n leaves has exactly $n-1$ branching nodes, any function $f_C \in D$ can be identified using $|D|-1$ interactions. ■

Of course, there are 'easy' function sets in the sense that their members can be identified using much fewer interactions. Consider, for example, the set $\{n \mapsto i \; \forall n | 1 \leq i \leq L\}$. Here, only one (any) probing input is necessary, since the identity of the function can be directly read from the output.

An Interrogator is called *interactive* if the value x_j of the jth probing input depends on $f_C(x_1), \ldots, f_C(x_{j-1})$, i.e., the outputs corresponding to the previous probing inputs. This implies that the probing inputs generally depend on the specific function f_C given to I. Conversely, a *non-interactive* Interrogator can only provide a fixed set of probing inputs to f_C, and their values do not depend on the outputs of f_C. In the proof of Proposition 2, the probing inputs given to f_C do not dynamically depend on f_C. This means that the theorem holds for non-interactive Interrogators. A natural question arises: Can interactive Interrogators identify a function using fewer interactions? Although there are instances where they need exponentially fewer interactions, in the worst-case scenario, both methods necessitate an equivalent number of interactions:

Proposition 3 *The upper bound for probing interactions required to identify a function from a given function set D is $|D|-1$ for both interactive and non-interactive Interrogators.*

Proof It is easy to construct function sets D for which the members cannot be identified in less than $|D|-1$ interactions, even by an interactive Interrogator.

One such function set is $\{\xi_i | 1 \leq i \leq L\}$ with $\xi_i : n \mapsto \begin{cases} 0 \text{ if } n = i, \\ n \text{ else} \end{cases}$. In the worst case, there is no way around trying all inputs $i, \ldots, L-1$. ■

Proposition 4 *There exists a function set for which an interactive Interrogator requires exponentially fewer probing interactions to identify a member than does a non-interactive one.*

Proof We construct a concrete set of functions that an interactive Interrogator can identify exponentially faster than a non-interactive one. Consider the family of context-sensitive languages

$$L_{m_1,\ldots,m_k} := \{a_1^{n+m_1} a_2^{n+m_2} \ldots a_k^{n+m_k} | n \in \mathbb{N}\}, \tag{2}$$

with $m_1, \ldots, m_k \in \mathbb{N}$ and $a_1, \ldots, a_k$ being the letters/tokens of the language. The parameters m_i define the relative number of times different tokens may appear. As an example, one member of the language $L_{3,1,2}$ is the string $a_1a_1a_1a_1a_2a_2a_3a_3a_3$.

Let $G_L := \{L_{m_1,\ldots,m_k} | m_1, \ldots, m_k \in \{1, \ldots, M\}\}$, i.e., a set of such languages with different parameters m_i. G_L contains M^k languages. To each language $L_{m_1,\ldots,m_k}$, we can assign a unique generative function $g_{m_1,\ldots,m_k}$. This function, given a partial string from the language, returns a list of the allowed tokens for the next step. If the input string is not a partial string of the language, it returns the empty string ϵ. For example, $g_{3,1,2}(a_1a_1a_1) = (a_1, a_2)$, $g_{3,1,2}(a_1a_1a_1a_1a_2) = (a_2)$, and $g_{3,1,2}(a_1a_1a_2a_2) = \epsilon$. Our function set D_L is a set of such generative functions, $D_L := \{g_{m_1,\ldots,m_k} | m_1, \ldots, m_k \in \{1, \ldots, M\}\}$.

For an interactive Interrogator, there is a simple strategy to identify a given function $g_C \in D_L$ using $M \cdot k$ interactions: The first input is the string $a_1^M a_2$. From there on, the Interrogator acts as an autoregressive generative model—it appends the allowed token returned by g_C to the string and uses it as the new input. Only one valid token will be returned by g_C for all probing input strings that are generated using this approach since the n is determined from the first input string. This is repeated until ϵ is returned, which is after a maximum of $(M-1) \cdot k$ calls to g_C. The last probing input string will be $a_1^{r_1} \ldots a_k^{r_k}$ with $r_1 = M$, from which the language can be inferred in the following way: Let $n := \min\{r_1, \ldots, r_k\}$. The language g_C is generating is thus $L_{r_1-n,\ldots,r_k-n}$.

The non-interactive Interrogator cannot use this strategy, since every probing input except the first depends on g_C's output for the previous probing input. We can show that in the non-interactive setting, $(M-1)^k$ calls to g_C are needed to identify it. Assuming $n = 0$, there are M^{k-1} unique prefixes for the first token a_k. Each of these prefixes is only allowed in M languages $L_{m_1,\ldots,m_{k-1},\cdot}$, namely the ones with specific $m_1, \ldots, m_{k-1}$. Remember that g_C returns ϵ whenever it is given a substring that is not part of its language. That means, to determine m_k, $M^{k-1}(M-1)$ different inputs have to be given to f_C. It follows that in total, $\sum_{b=2}^{k-1} M^b(M-1) = M^k - M^2$ inputs are needed to identify the exact language of g_C.

In short, to identify a function from the set D_L described above, an interactive Interrogator needs $O(Mk)$ probing inputs, whereas a non-interactive one needs $O(M^k)$. ■

Appendix B. Implementation Details

Flattened Weights For the flattened weights Encoder, all parameters θ of the RNN to be encoded are flattened into a vector. This weight vector is given as input to a multi-layer perceptron (MLP) with ReLU nonlinearities, which outputs the RNN encoding z.

Neural Functional For the neural functional (NF) Encoder, we adapt the equivariant NF-layer (21) for LSTMs. To preserve both equivariance to neuron permutation and functional universality, the appropriate row- and column-wise feature extractors have to be added for input-to-hidden and hidden-to-hidden weights, considering rollouts across time and depth of the network.

Non-interactive RNN Probing A diagram of the non-interactive probing encoder is shown in Figure 1 (left). RNN probing Encoders have three main components: the core LSTM E_R, an input projection MLP E_I and an output projection MLP E_O.

For the the non-interactive Encoder, a learnable latent probing sequence $(S_1, S_2, \ldots, S_l)$ with a fixed length l is given to E_I. $E_I(S_i)$ is interpreted as either one a several parallel probing inputs $\hat{x}_i$ and given to f_θ. The resulting probing outputs $\hat{y}_i := f_\theta(\hat{x}_i)$ are given to E_O (in the case of multiple parallel probing outputs, the values $\hat{y}_i$ are concatenated). The sequence of probing output projections $(E_O(\hat{y}_1), \ldots, E_O(\hat{y}_l))$ is given as input to E_R, preceded by a begin-of-sequence (BOS) and followed by an end-of-sequence (EOS) token. E_R's output after the EOS token is transformed with a learned linear projection into the RNN representation z.

Interactive RNN Probing As can be seen in Figure 1 (right), the interactive probing encoder differs from the non-interactive one in one crucial aspect: Instead of having a learned but static latent probing sequence, the probing inputs at each step are based on the output of E_R from the current step, which in turn depends on the probing outputs of the previous step. This means that the interactive probing Encoder can dynamically adapt the probing sequences to the particular RNN f_θ that is being encoded.

Emulator The Emulator A_ξ is an LSTM network. The conditioning on the function representation z is done by adding a learned linear projection of z to the embedding of the begin-of-sequence token.

Appendix C. Experimental Details

LSTM Dataset The set of languages is $\{L_{r,r+o_1,r+o_2,r+o_3} | o_1, o_2, o_3 \in \{-3, \ldots, 2\}$ and $r = -\min\{o_1, o_2, o_3\}\}$, with L defined in Equation 2. This set contains $6^3 = 216$ uniquely identifiable languages. The training data for each LSTM are strings from a particular language of length ≤ 40, with an additional begin-of-sequence and end-of-sequence token.

The LSTMs trained for the dataset have two layers with a hidden size of 32, resulting in a total of 13766 parameters. In total, 1000 such networks are trained, each on one of the 216 possible languages. For each LSTM, 10 snapshots (at steps 0, 100, 200, 500, 1000, 2000, 5000, 10000 and 20000) are saved during training. A snapshot consists of the LSTM's current weights and 100 sequences, also of length 40, generated by it.

Hyperparameters Table 1 shows the hyperparameters shared by all four encoder types in the experiments. Hyperparameters specific to probing, flattened and neural functional encoders are shown in Tables 2, 3 and 4, respectively.

Hyperparameter	Value
A hidden size	256
A #layers	2
z size	16
batch size	64
optimizer	AdamW
learning rate	0.0001
weight decay	0.01
gradient clipping	0.1

Table 1: General hyperparameters

Hyperparameter	Value
E_R hidden size	256
E_R #layers	2
E_I hidden size	128
E_I #layers	1
E_O hidden size	128
E_O #layers	1
probing sequence length	22

Table 2: Hyperparameters for probing (interactive and non-interactive) encoders

Hyperparameter	Value
hidden size	128
#layers	3

Table 3: Hyperparameters for flattened weights encoders

Hyperparameter	Value
#channels	4
#layers	4

Table 4: Hyperparameters for neural functional encoders

Additional Results Figure 5 shows the test losses of the different Emulators (as defined in Equation 1). The relative performance of the different encoders types is similar as for the downstream tasks shown Figure 2.

We also investigate the results for different lengths of the probing sequence for the interactive and non-interactive probing encoders. This is shown in Figure 7.

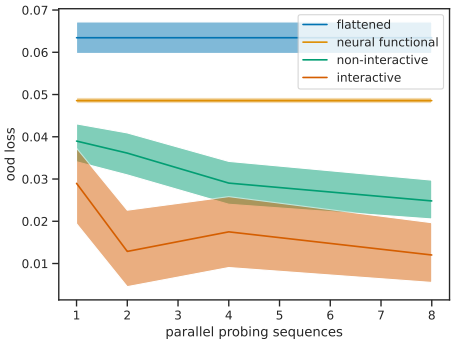

Figure 5: Loss of the emulator on the test. Plotted as a function of the number of parallel probing sequences. Mean and bootstrapped 95% confidence intervals across 15 seeds.

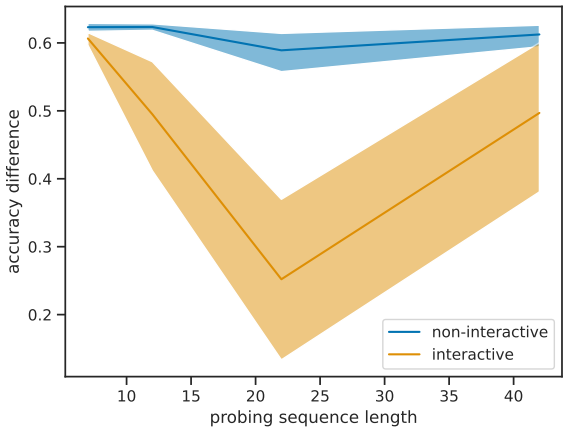

Figure 6: Difference in generation accuracy of the emulated function compared to the original one. Validation set. Plotted as a function of the length of the probing sequence. Mean and bootstrapped 95% confidence intervals across 15 seeds.

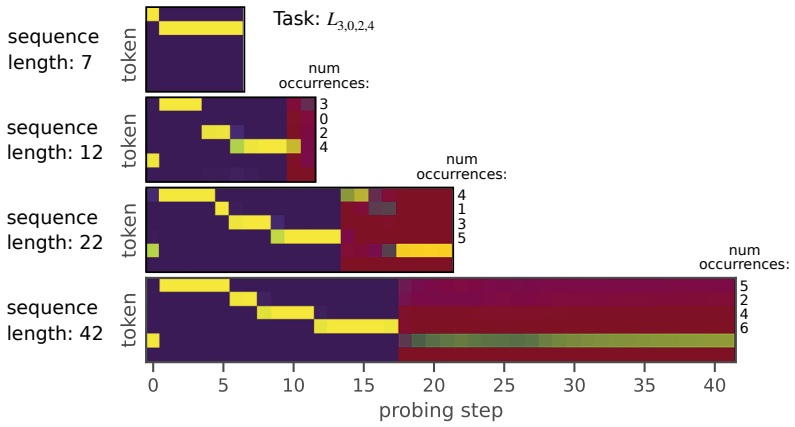

Figure 7: Probing sequences generated for $g_{2,0,1,3}$ by the best performing interactive function encoder with different sequence lengths. For the sequence lengths 12, 22 and 42, the encoder produces a insightful probing sequence, i.e. probing sequences that belong to the corresponding language.

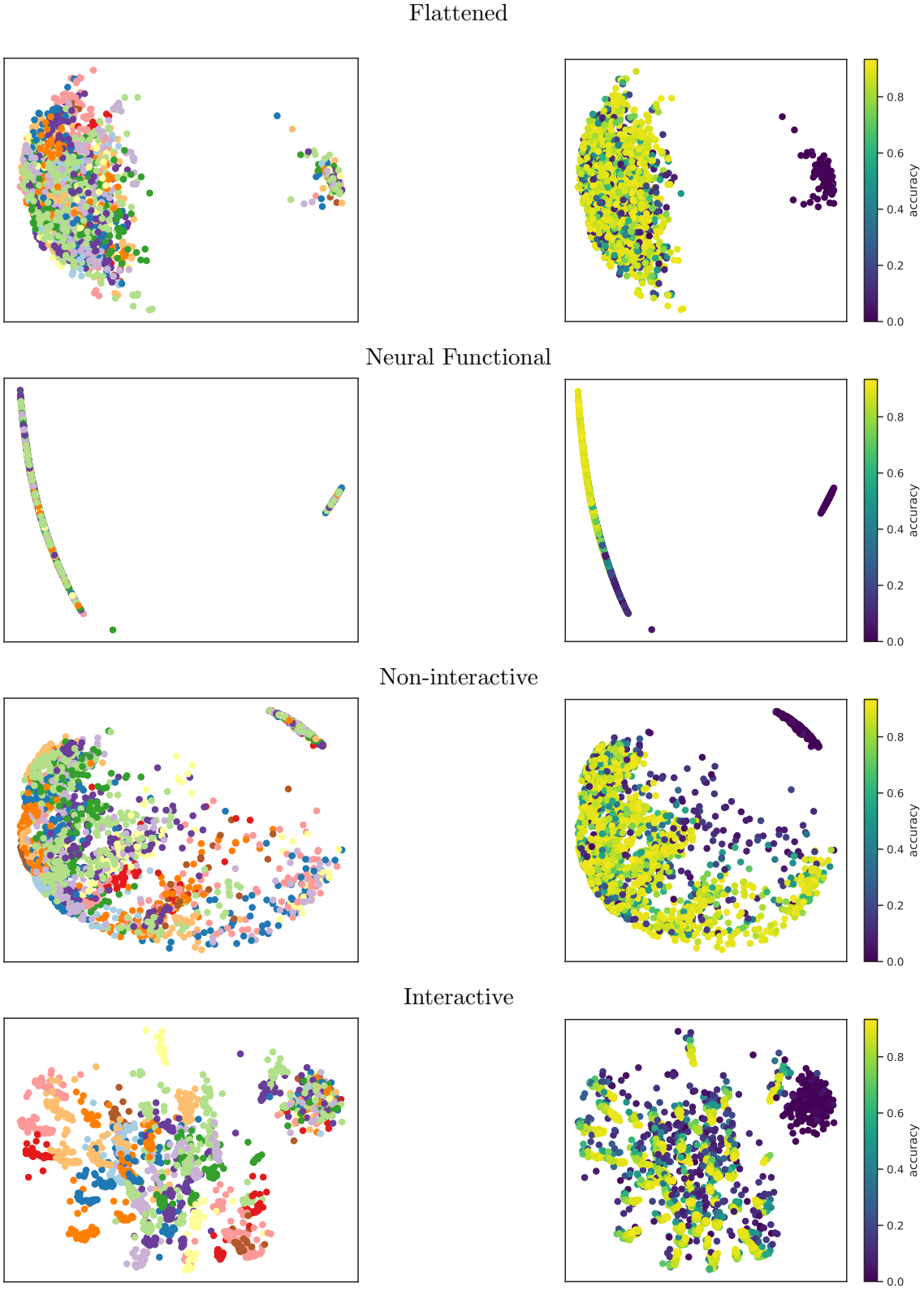

Figure 8: PCA of the function encodings generated by different encoders. Left column: Colored by language, Right column: Colored by perfomance (accuracy).

Extended Abstract Track