Learning Useful Representations of Recurrent Neural Network Weight Matrices

Vincent Herrmann [1] Francesco Faccio [1 2] Jürgen Schmidhuber [1 2]

Abstract

Recurrent Neural Networks (RNNs) are general-purpose parallel-sequential computers. The program of an RNN is its weight matrix. How to learn useful representations of RNN weights that facilitate RNN analysis as well as downstream tasks? While the *mechanistic approach* directly looks at some RNN's weights to predict its behavior, the *functionalist approach* analyzes its overall functionality—specifically, its input-output mapping. We consider several mechanistic approaches for RNN weights and adapt the permutation equivariant Deep Weight Space layer for RNNs. Our two novel functionalist approaches extract information from RNN weights by 'interrogating' the RNN through probing inputs. We develop a theoretical framework that demonstrates conditions under which the functionalist approach can generate rich representations that help determine RNN behavior. We release the first two 'model zoo' datasets for RNN weight representation learning. One consists of generative models of a class of formal languages, and the other one of classifiers of sequentially processed MNIST digits. With the help of an emulation-based self-supervised learning technique we compare and evaluate the different RNN weight encoding techniques on multiple downstream applications. On the most challenging one, namely predicting which exact task the RNN was trained on, functionalist approaches show clear superiority.

1. Introduction

For decades, researchers have developed techniques for learning representations of complex objects such as images, text, audio and video with deep neural networks (NNs). This expertise has significantly advanced the field by enabling models to convert data into formats useful for solving problems. In particular, recurrent NNs (RNNs) have been widely adopted due to their computational universality (Siegelmann & Sontag, 1991). 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 representations of RNN weight matrices have been used to evolve RNN parameters (Koutník et al., 2010) for controlling a car from raw video input (Koutník et al., 2013), using Fourier-type transforms, e.g., the coefficient of the Discrete Cosine Transform (DCT) (Srivastava et al., 2012), without using the capabilities of NNs to learn such representations. Recent work has seen a rise of representation learning techniques for NN weights using NNs as encoders (Eilertsen et al., 2020; Unterthiner et al., 2020; Schürholt et al., 2021; Dupont et al., 2022; Faccio et al., 2022b). However, there is a lack of methods for learning representations of RNNs.

This paper introduces novel techniques for learning RNN representations using powerful NNs, which may be RNNs themselves. Just like representation learning in other fields, such as computer vision, facilitates solutions of specific tasks, such techniques can facilitate learning, searching, and planning with RNNs. We show that by employing general RNN weight encoder architectures and self-supervised learning methods, it is possible to learn representations that capture diverse functionalities of RNNs. We differentiate between encoders that treat the weights as input data (mechanistic) and those that engage only with the function defined by the weights (functionalist). Within the functionalist approach, the *non-interactive probing* method uses learnable but fixed probing sequences as input to the RNN and observes the corresponding outputs. In contrast, *interactive probing* adapts the probing sequences dynamically based on the input RNN in order to extract the most relevant information. We provide empirical and theoretical evidence of the effectiveness of interactive probing for complex tasks, despite occasional training stability issues. For simpler tasks or when interactive properties are not required, non-interactive probing or mechanistic encoders might be more suitable.

[1]The Swiss AI Lab IDSIA, USI & SUPSI [2]AI Initiative, KAUST. Correspondence to: Vincent Herrmann <vincent.herrmann@idsia.ch>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024. Copyright 2024 by the author(s).

Our contributions are summarized as follows:

(1) We introduce the challenge of learning useful representations of RNN weights and propose six neural network architectures for processing these weights. We define the difference between mechanistic and functionalist approaches, adapt Deep Weight Space Nets (DWSNets, Navon et al. (2023); Zhou et al. (2023)) to RNNs, and introduce novel probing architectures, including the concept of interactive probing.

(2) We develop a theoretical framework for analyzing the efficiency of interactive and non-interactive probing encoders. We prove that interactive probing encoders can be exponentially more efficient for certain problems.

(3) We create and release two comprehensive RNN "model zoo" datasets. Each dataset consists of the weights of thousands of LSTMs (Hochreiter & Schmidhuber, 1997), trained on hundreds of different but related tasks. One dataset focuses on formal languages, while the other on tiled sequential MNIST. [1]

(4) We conduct empirical analyses and comparisons across the different encoder architectures using these datasets, showing which encoders are more effective.

2. Related Work

The concept of learning representations for the weights of feedforward NNs, also sometimes called hyper-representations, has been explored in studies by Eilertsen et al. (2020) and Schürholt et al. (2021). Various methods for processing NN weights have been proposed. Schürholt et al. (2021); Eilertsen et al. (2020); Faccio et al. (2020); Herrmann et al. (2022) suggest flattening the weights and using them as input data for simple encoders or predictors. Unterthiner et al. (2020) and Tang et al. (2022) use permutation-invariant layers to extract high-level weight features. Navon et al. (2023) and Zhou et al. (2023) develop weight processing layers that are equivariant to neuron permutation, leading to the creation of DWSNet architectures, which can universally approximate functions of the weight space. Other approaches include probing NNs with learnable inputs and analyzing the network based on the generated outputs, as proposed by Schmidhuber (2015); Harb et al. (2020) and Faccio et al. (2022a;b). For processing implicit neural representations, Dupont et al. (2022) employ normalizing flows and diffusion models, while Xu et al. (2022) use higher-order spatial derivatives. All mentioned works, except for Schmidhuber (2015) and Herrmann et al. (2022), focus solely on the processing of feedforward (including convolutional) NNs.

Emulation as an objective for learning representations was proposed by Raileanu et al. (2020), but their focus was on policy trajectories rather than on NN weights. For self-supervised representation learning of NN weights, reconstruction-based approaches have been explored (Schürholt et al., 2021; Dupont et al., 2022). Ramesh & Chaudhari (2021) employ populations of models trained on diverse tasks for continual learning. Both Eilertsen et al. (2020) and Schürholt et al. (2022) have released datasets of trained convolutional NNs. To our knowledge, there are no public datasets of diverse trained RNNs.

3. RNN Weight Encoders

When using NN weights as input for another network, two main challenges arise. First, the number of weights can quickly become very large. Second, the weight space exhibits symmetries, particularly with respect to the permutation of hidden neurons (Hecht-Nielsen, 1990). Rearranging the order of hidden neurons in a network does not change the computation it performs[2]. An effective NN weight encoder should recognize these symmetries.

We consider an RNN, $f_\theta : \mathbb{R}^X \times \mathbb{R}^H \to \mathbb{R}^Y \times \mathbb{R}^H; (x, h_{t-1}) \mapsto (y, h_t)$, $t \in 1, 2, \ldots$, parametrized by $\theta \in \Theta$, which maps an input x and hidden state h_{t-1} to an output y and a new hidden state h_t. In the following, we assume that all RNNs are multi-layer LSTMs[3]. An RNN weight encoder is a function $E_\phi : \Theta \to \mathbb{R}^Z; \theta \mapsto z$, mapping RNN weights θ to Z-dimensional representation vectors $z = E_\phi(\theta)$. The encoder's parameters are $\phi \in \Phi$.

We differentiate between two approaches for encoding RNN weights: (1) *Mechanistic* encoders "look" at the weights θ directly, treating them as typical input data. (2) *Functionalist* encoders, instead, interact with the function f_θ without direct access to the weights themselves. These encoders still map RNN weights to representations, but focus on a functional interpretation of the RNN. We discuss and compare six encoder architectures for representing RNN weights, as depicted in Figure 1.

Layer-Wise Statistics This approach, successfully used in previous studies (Unterthiner et al., 2020) to predict properties of CNNs, involves creating, for each weight matrix, a vector consisting of mean, standard deviations, and five quantiles $(0, 0.25, 0.5, 0.75, 1)$. For LSTMs, each layer yields twelve distinct vectors, for each of the four gates corresponding to the input-to-hidden weights, hidden-to-hidden weights, and the bias vector. These vectors are then concatenated and given to a multi-layer perceptron (MLP).

[1] `https://github.com/vincentherrmann/rnn-weights-representation-learning`

[2] With piece-wise linear activation functions like ReLU, there is also invariance to certain types of weight scaling. However, this is not a focus here as RNNs usually use different activation functions. There is another symmetry with respect to sign flips.

[3] Generalizing our framework to other RNN architectures should be straightforward.

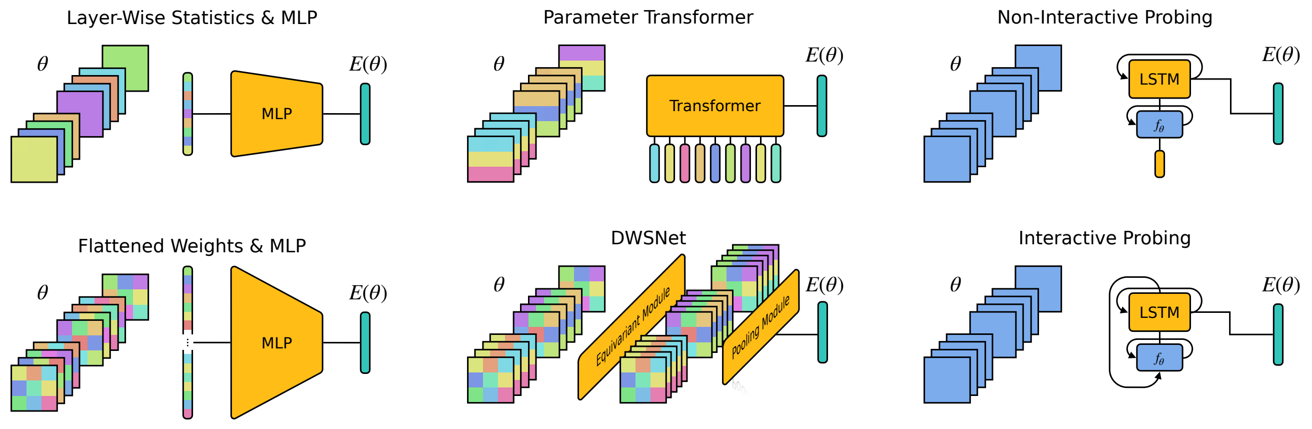

Figure 1: RNN weight encoder architectures taking weights θ as input and producing a representation $E(\theta)$. The two groups of four weight matrices symbolize the four gates of two LSTM layers. The last matrix represents the output projection.

The architecture is inherently invariant to permutations of hidden neurons. It efficiently scales with RNN size due to its reliance on high-level features. However, this invariance extends to many transformations beyond those that preserve the RNN functionality. Consequently, RNNs with identical layer-wise statistics may behave differently, preventing the encoder's ability to approximate all functions of the weight space.

Flattened Weights In this approach, all RNN weights are flattened into a single vector before being fed into an MLP. Unlike the previous architecture, this one is not invariant to hidden neuron permutations. As a result, it faces a different challenge: numerous RNNs might appear different to the encoder yet perform identical computations. This impedes generalization, as the MLP has difficulty learning these symmetries (empirically demonstrated in Appendix E.2). Another issue is the very large size of the input vector. Let N represent the number of hidden neurons in f_θ. The number of parameters in the input layer of the MLP is proportional to N^2. However, with an adequately large MLP, this architecture can approximate any function in the weight space[4]. This follows immediately from the universal approximation property of MLPs (Hornik, 1991).

Parameter Transformer Schürholt et al. (2021) introduced an attention-based architecture. This design treats the weights of individual neurons (specifically, the rows of weight matrices along with their corresponding bias values) as a sequence. These sequences are then processed by an encoder-only transformer model (Vaswani et al., 2017). A learned positional encoding ensures that the transformer has the information which weights correspond to which neuron. This also makes it not invariant to neuron permutation. The attention mechanism within the transformer enables associative retrieval of information from other neurons, which could be a beneficial inductive bias when handling NN weights. The size of both the neuron sequence and the input transformation parameters scale linearly with N. Given that transformers are known to be universal sequence-to-sequence function approximations (Yun et al., 2019), the parameter transformer can theoretically approximate any function of the weight space.

DWSNet Both Navon et al. (2023) and Zhou et al. (2023) proposed architectures for processing the weights of feedforward NNs, closely related in design. These architectures are invariant precisely to permutations of hidden neurons and are capable of universally approximating functions of the weight space. We refer to this architecture as DWSNet, following Navon et al. (2023), and extend its application to LSTM networks. The central concept of DWSNet is to construct layers that are equivariant to the hidden neuron permutation group. These layers process the weights, where each weight is represented by a feature vector. A final pooling layer across all weights ensures invariance to neuron permutation. The mechanisms for adapting DWSNets to LSTMs, along with arguments for their universality, are presented in Appendix B. Appendix E.2 validates the implementation by demonstrating that DWSNets maintain invariance only to correctly permuted weights in LSTMs.

Non-Interactive Probing In the context of Reinforcement Learning and Markov Decision Processes, policy fingerprinting has emerged as an effective method to evaluate feedforward NN policies (Harb et al., 2020; Faccio et al., 2022a;b). 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

[4]When mentioning universality of weight space functions $\Theta \to \mathbb{R}^Z$, we imply the regularity conditions of Navon et al. (2023), Proposition 6.2.

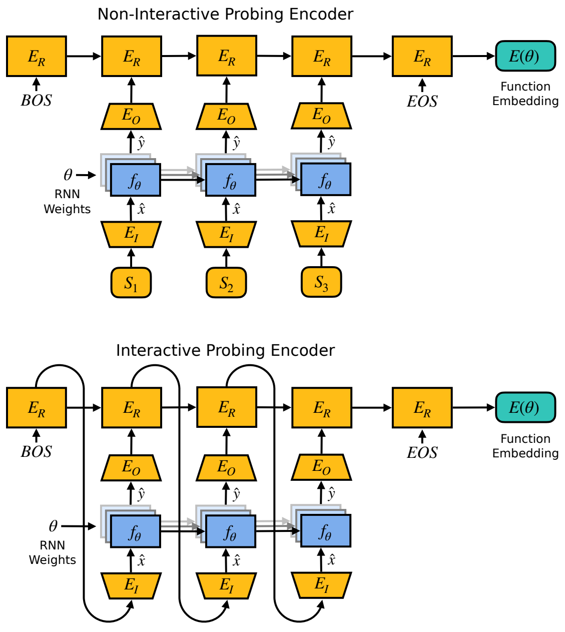

Figure 2: Comparison of non-interactive and interactive procing encoders.

is produced. This functionalist approach, which we refer to as non-interactive probing in this paper, is easily adaptable to RNNs by employing sequences of probing inputs (see Figure 2, top).

The probing sequence length, denoted as L, is a fixed hyperparameter. A non-interactive probing encoder uses a sequence of learnable embeddings, $S_i, 1 \leq i \leq L$. At each probing step i, the embedding S_i is transformed by E_I, an MLP followed by a reshaping operation that outputs a set of M probing inputs $\hat{x}_{im} \in \mathbb{R}^X \ \forall m \in \{1, \ldots, M\}$. M represents the number of parallel probing sequences. These inputs are processed as a batch by the RNN f_θ, resulting in a batch of probing outputs $\hat{y}_{im} = f_\theta(\hat{x}_{im}) \in \mathbb{R}^Y$ [5]. The probing outputs are concatenated and further transformed by another MLP E_O, resulting in a vector o_i. The sequence of vectors o_i is processed by an LSTM network E_R, which then outputs the RNN representation $E(\theta)$.

Interactive Probing The probing sequences for non-interactive 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 new probing input, $\hat{x}_i$, should depend on the previous probing outputs $\hat{y}_{<i}$. This dynamic adaptation is achieved by feeding the output of the previous step's LSTM $E_R(o_{<i})$ into E_I (Figure 2, bottom). A similar concept has been proposed by Schmidhuber (2015) for extracting algorithmic information from recurrent world models.

[5]For f_θ, the recurrence of the hidden states over sequence steps is implied.

Both types of probing encoders retain the invariance properties of f_θ. However, functionalist encoders have limitations in differentiating between weight space functions; they cannot discern mechanistic differences in functionally equivalent RNNs. Consequently, two RNNs performing exactly the same function will look identical to a probing encoder, even if they use different algorithms to compute the function. This means probing encoders are not universal in the sense that some of the mechanistic approaches are. Table 1 summarizes key properties of the different encoder architectures.

3.1. Theoretical Aspects of the Functionalist Approach

We introduce a theoretical framework for analyzing probing encoders and the distinctions between interactive and non-interactive settings. In practice, any RNN weight encoder is trained on a finite number of distinct RNNs. For each of them, it should output a unique representation. In this section, we study the related task of uniquely identifying a function from a given set by interacting with it. Rather than using RNNs, we examine total computable functions, which are functions that halt and produce an output for every input. This is a minor limitation, since in practical scenarios, RNNs are almost always given a finite runtime.

Let D represent a set of n total computable functions $\{f_i : \mathbb{N} \to \mathbb{N} | i = 1, 2, \ldots, n\}$. In other words, 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. It should be mentioned that RNNs of finite size and precision are *not* universal computers (Merrill, 2019; Delétang et al., 2022). However, the following propositions depend mainly on the relative computational ability of I_D and the functions in D. Hence we choose this simple abstract framework of distinguishing total computable functions and believe that it transfers well into realistic RNN settings. The proofs of the following propositions can be found in Appendix A.

Proposition 3.1. *Any function f_C from a set D can be identified by an interrogator through at most $|D| - 1$ interactions.*

Table 1: Properties of the different RNN weight encoder architectures. N is the number of hidden neurons in f_θ.

Encoder	Permutation Invariant	Universal Approx.	#Params	Type
Layerwise Statistics	Yes	No	const.	Mechanistic
Flattened Weights	No	Yes	$O(N^2)$	Mechanistic
Parameter Transformer	No	Yes	$O(N)$	Mechanistic
DWSNet	Yes	Yes	const.	Mechanistic
Non-Interactive Probing	Yes	No	const.	Functionalist
Interactive Probing	Yes	No	const.	Functionalist

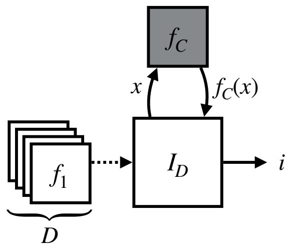

Figure 3: Interrogator I_D has access to a set D of functions and interacts with function f_C, which it has to identify.

An Interrogator is called *interactive* if the value x_j of the jth probing input depends on $f_C(x_1), \dots, f_C(x_{j-1})$, 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 3.1, 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 3.2. *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.*

Proposition 3.3. *There exist function sets for which an interactive Interrogator requires exponentially fewer probing interactions to identify a member than does a non-interactive one.*

Section 6 demonstrates that these theoretical concepts are mirrored in empirical results. In one dataset (formal languages), interactive probing significantly outperforms non-interactive probing. However, in another dataset, both methods show similar performance.

4. Self-Supervised Learning of RNN Weight Representations

We propose a general-purpose method for learning representations of RNN weights. It is based on the idea that the RNN weight representation should contain all the information necessary in order to emulate the RNN's functionality. A very similar technique is used by Raileanu et al. (2020) to learn representations of (non-recurrent) policies based on their trajectories.

The RNN f_θ 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:

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

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 at time step t—, 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 $S_\theta = (x_1, y_1, x_2, y_2, \dots)$.

To train an RNN weight encoder E_ϕ, we consider an Emulator $A_\xi : \mathbb{R}^X \times \mathbb{R}^B \times \mathbb{R}^Z \to \mathbb{R}^Y \times \mathbb{R}^B; (x, b_{t-1}, z) \mapsto (\tilde{y}, b_t)$, 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 $z = E(\theta)$.

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. 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 (see Figure 4). Put simply, the Emulator uses the representations of a set of diverse RNNs f_θ to imitate their behavior:[6]

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

In the case of continuous outputs y, the mean-squared error

[6]The recurrence of A_ξ is omitted for simplicity.

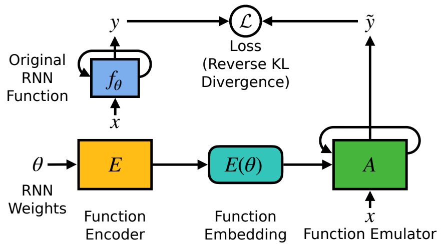

Figure 4: Emulation-based self-supervised training. The encoder E is trained to generate embeddings of θ that allow A to emulate f_θ.

provides a suitable loss function $\mathcal{L}$. For categorical outputs, we employ the reverse Kullback-Leibler divergence because of its mode-seeking behavior.

If we are interested only in the RNN weight encoder E_ϕ, the emulator A_ξ can be discarded after training. However, there might be potential applications for the emulator, for example, in the context of imitation learning (Zare et al., 2023), behavior cloning (Torabi et al., 2018), or consolidating the knowledge from many different models (Joseph & N Balasubramanian, 2020).

5. Datasets

To evaluate the methods described and foster further research, we develop and release two "model zoo" datasets for RNNs. The first dataset focuses on modeling formal languages, while the second is centered around predicting digits in a tiled Sequential MNIST format. Both datasets share a similar structure. We train 1000 LSTMs (each with two layers and a hidden state size of 32) on various tasks. The weights θ of each model are saved at 9 fixed training steps, along with 100 rollouts S_θ and additional data, such as the current performance on its task. The datasets are divided into training, validation, and out-of-distribution (OOD) test splits, with tasks in each split being non-overlapping. The tasks in the OOD set are structurally slightly different.

5.1. Formal Languages

The models are auto-regressive language models, trained on different formal languages using teacher forcing and the standard cross-entropy language modelling objective. Let a, b, c and d be the four tokens of a language. We define a language as $L_{m_b,m_c,m_d} := \left\{a^n b^{n+m_b} c^{n+m_c} d^{n+m_d} | n \geq - \min\{0, m_b, m_c, m_d\}\right\}$. This means that, in a string from such a language, the number of appearances of the tokens, relative to each other, is determined by m_b, m_c and m_d. For example, the strings from the language $L_{1,-1,2}$ are $\{\{abbddd\}, \{aabbbcdddd\}, \{aaabbbbccddddd\}, \ldots\}$. Each model is trained on one language from the set $G_L := \{L_{m_b,m_c,m_d} | m_b, m_c, m_d \in \{-3, -2, \ldots, 2\}\}$. Note that these languages are essentially ones used also in the proof of proposition 3.3. In G_L, there are a total of $6^3 = 216$ different languages.

All models are trained on sequences of length 42, including one begin-of-sequence (BOS) and one end-of-sequence (EOS) token. The maximum value of n is 10. If a language string is shorter than 42, it is padded at the end with EOS tokens. The OOD test set contains the RNNs trained on languages where the sum of the absolute values of m_b, m_c and m_d is the smallest.

5.2. Tiled Sequential MNIST

The models of this dataset are trained to classify MNIST digits presented in a sequential format. Unlike the typical pixel-wise sequence, each digit is represented as a sequence of 49 4×4 tiles (plus BOS and EOS tokens). This approach improves computational efficiency for both RNN training and weight representation experiments. After each tile of the sequence, the model predicts the digit. The loss is the mean cross-entropy of all predictions in the sequence. However, the accuracy of each model is assessed based on the final prediction, i.e., when the model has seen the entire digit. The dataset's task involves rotating MNIST digits. Each model is exposed to the entire MNIST dataset, with images rotated by a unique random angle. For the training and validation sets, the rotations range from 0 to 311 degrees, while for the OOD test set, they range from 312 to 360 degrees.

6. Experiments and Results

Our empirical investigation involves two experimental phases. In the first phase, we apply the emulation-based representation method described in Section 6.1 to learn representations for RNNs from Formal Languages and the Sequential MNIST dataset. The second phase is dedicated to predicting properties of the RNNs. These predictions are either based on representations learned in the first phase or derived from fully supervised models trained from scratch. We conduct the main experiments using 15 different random seeds for each model. The outcomes are presented as bootstrapped means with 95% confidence intervals. For RNNs trained on the Formal Languages dataset, performance is measured by the proportion of correctly generated strings (i.e., strings belonging to the language on which it was trained). For the Sequential MNIST dataset, we assess performance using standard digit classification validation accuracy. For the Flattened Weights and the Parameter Transformer encoders, the training data is augmented by randomly permuting the neurons of the input RNN.

Table 2: Self-supervised validation losses.

Encoder	Formal Languages	Sequential MNIST
Layer-Wise Statistics	0.051 (0.050,0.053)	0.039 (0.038,0.039)
Flattened Weights	0.045 (0.045,0.046)	0.024 (0.024,0.024)
Parameter Transformer	0.043 (0.042,0.044)	0.067 (0.067,0.067)
DWSNet	0.046 (0.046,0.046)	0.024 (0.023,0.025)
Non-Interactive Probing	0.023 (0.019,0.029)	**0.017** (0.016,0.017)
Interactive Probing	**0.015** (0.008,0.022)	**0.017** (0.017,0.018)

6.1. Representation Learning

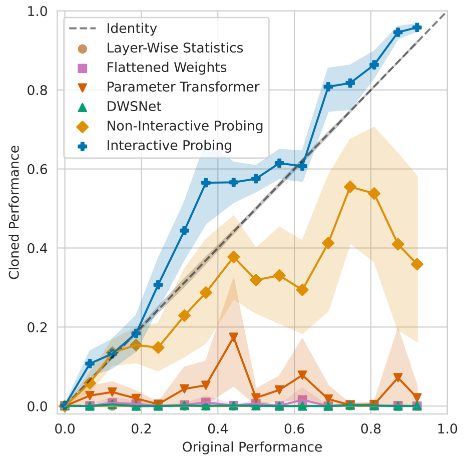

Figure 5: f_θ's original performance on formal language generation vs. the performance of A_ξ's emulation based on $E_\phi(\theta)$ (validation data).

The six different encoder architectures are trained according to Objective 1. The hyperparameters of these encoders are selected to ensure a comparable number of parameters across all models. Each encoder generates a 16-dimensional representation z. An LSTM with two layers functions as the emulator A_ξ. The conditioning of A_ξ on an RNN f_θ is implemented by incorporating a linear projection of the corresponding representation z to the BOS token of the input sequence of A_ξ. More details and hyperparameters can be found in Appendix D.

Table 2 displays the emulation losses for the different encoder types on both the Formal Languages and the Sequential MNIST validation datasets. For Formal Languages, the interactive probing method outperforms the others. In the case of Sequential MNIST, the performance of both probing encoders is quite similar and superior to that of the mechanistic encoders. Figure 5 shows the emulation effectiveness of various RNNs from the Formal Languages dataset. We compute 16 equally spaced target performance values between the best and the worst performances in the datasets. For each target performance, we select the 15 RNNs with performances closest to each target. The x-position of each point represents the mean of the original performances, and the y-positions represent the mean of the emulated performances (the shaded areas give the 95% confidence intervals). The variance of the original performances for each point is relatively low, as can be seen from the shaded area around the identity line. For this dataset, only the interactive probing encoder yields representations that enable A_ξ to effectively emulate the original RNN. Figure 11 in the Appendix shows the analogous results also for the Formal Languages training and test set, Figure 12 for the Sequential MNIST datasets.

Figure 6 (top) examines the structure of the embedding space created by the interactive probing encoder for the Formal Languages validation set. It visually demonstrates that our method successfully learns coherent representation spaces of RNN weights. Each point represents an RNN, reduced from Z to two dimensions using principal component analysis. Each language forms its own cluster, with each cluster exhibiting a gradient representing the generation accuracy of the RNNs. We highlight one cluster, corresponding to the language $L_{-2,2,-2}$. In contrast, t-SNE dimensionality reduction (bottom) fails to represent such structures in RNN weights. In t-SNE reductions, the weights of the RNN models for a single language are scattered across the entire space. Similarly, Figure 7 shows the embedding space for the Sequential MNIST dataset, where the task involves rotating MNIST digits. This rotation, a continuous scalar, is evident in the embedding space, alongside the classification accuracy of each model. In the t-SNE plot, although small clusters are noticeable for the nine snapshots of each run, the overall embedding space lacks a coherent structure. Visualizations of embedding spaces for all encoder architectures and OOD test sets are presented in Appendix E.4.

6.2. Downstream Property Prediction

To assess the effectiveness of representations learned through self-supervised learning, we evaluate them in predicting various properties of RNNs. We train an MLP as a supervised prediction model using these representations, obtained from a fixed, pre-trained encoder E_ϕ. The properties to be predicted are stored as metadata for each RNN within the datasets. For RNNs from the Formal Languages dataset, the properties are task and accuracy. For RNNs from the Sequential MNIST dataset, the properties are task (i.e., the rotation of the digits), accuracy, training step, and generalization gap. The formal language task is represented as a three-hot vector (for values m_b, m_c, and m_d), and the predictor is trained using binary cross-entropy. For all scalar properties like accuracy, Sequential MNIST task, and generalization gap, the predictor is trained using mean squared error loss. The Sequential MNIST training step prediction is framed as a 9-way classification problem, using cross-entropy loss.

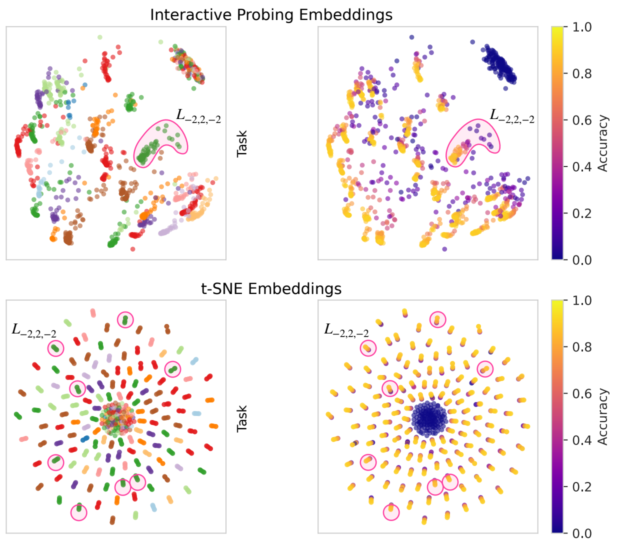

Figure 6: Visualization of embeddings of RNNs from the Formal Languages validation set.

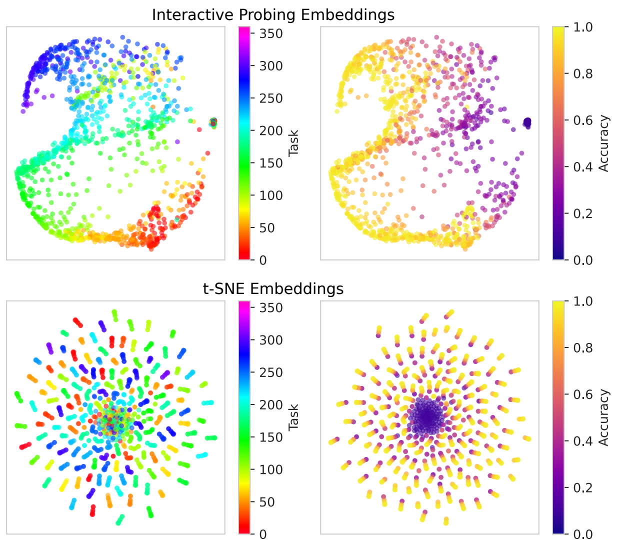

Figure 7: Visualization of embeddings of RNNs from the Sequential MNIST validation set.

For comparison, we also trained models for property prediction in a purely supervised manner. The setup is the same as for the pre-trained scenario, consisting of an RNN weight encoder followed by an MLP predictor. However, in this case, the encoder is not pre-trained but is randomly initialized and trained end-to-end with the predictor. Our primary interest lies in the generalization capabilities of the pre-trained E_ϕ, rather than the predictor itself. Consequently, the training data for the predictor in both supervised and pre-trained settings includes half of the OOD test set. We place particular emphasis on *task* prediction (specifically, the type of formal language or the rotation of the MNIST dataset). This is because it demands a genuine understanding of the RNN's function, which goes beyond mere high-level statistical analysis of the weights. Figure 8 (top) demonstrates that for task prediction in the Formal Languages dataset (OOD split), the pre-trained representations from an interactive probing encoder significantly outperform those from other encoders, as well as purely supervised models. For the Sequential MNIST dataset (Figure 8, bottom), both types of probing encoders surpass other architectures. In the supervised scenario, the non-interactive probing encoder excels, but the interactive version does not perform as well. We hypothesize that this is due to the limited information provided by supervised training compared to self-supervised pre-training, which does not offer sufficient feedback for the interactive encoder to develop effective probing sequences. Complete results for downstream predictions, both supervised and pre-trained, on validation and OOD test data are shown in Figures 19 and 20 in the Appendix. Although these results do not definitively favor any single encoder architecture, they can be summarized as follows: for the Formal Languages dataset, on both task and performance prediction, interactive probing generally yields the best results, particularly in the pre-trained setting. For the Sequential MNIST dataset, non-interactive probing performs best in the supervised setting, while both probing architectures are effective when pre-trained. In all other property predictions (accuracy, generalization gap, training step), DWSNet performs most consistently.

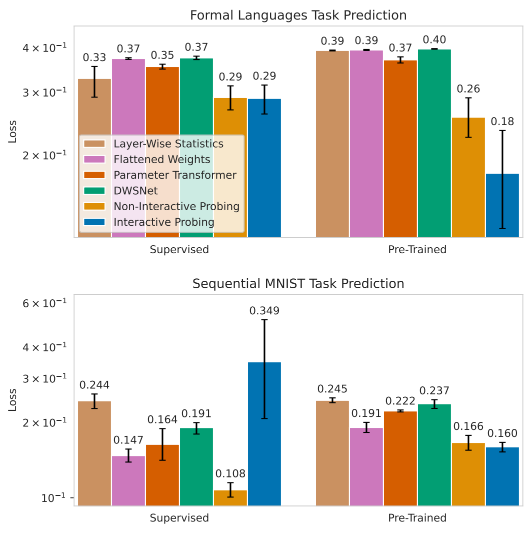

Figure 8: Downstream performance of pre-trained and purely supervised model on task prediction for the Formal Languages and Sequential MNIST OOD test set.

7. Discussion

To effectively learn useful representations of RNN weights, three key components are necessary. First, datasets comprising a variety of trained RNNs are essential. The two datasets presented in this study, based on formal languages and tiled Sequential MNIST, are designed to support rapid experimental cycles and evaluations. Despite their scale, they offer sufficient complexity and challenge to analyze various approaches to the representation learning problem. These datasets address distinct facets of potential RNN applications: generative modeling of formal languages, which suits RNNs due to its precise, algorithmic nature, and digit classification, a less precise perceptual task.

Second, a pre-training method is required. We suggest a method rooted in the understanding that only those representations of RNN weights that can accurately emulate the original RNN possess meaningful information about the network. However, this is a necessary but not sufficient condition for effective representations. If the RNNs in the dataset used to train the encoder differ only superficially (and not algorithmically), then the derived representations would only reflect these superficial properties. Therefore, the task family from the training data must be complex enough to necessitate extracting complex information from the RNNs, but still be learnable by some method. Our experiments demonstrate that only an interactive probing encoder can adequately capture the formal language task family. The Sequential MNIST task family (digit rotation) is simpler to learn, although the embedding visualizations suggest that Layer-wise Statistics and Parameter Transformer approaches fall short (see Figure 22).

Third, an encoder architecture capable of handling the complex structure of RNN weights as input is crucial. Except for Layer-Wise statistics, mechanistic encoders can approximate any function within the weight space. Nonetheless, in practice, functionalist approaches outperform them both in self-supervised training objectives and task prediction. Our findings confirm the theoretical prediction that interactive probing is more efficient for certain datasets (Proposition 3.3), as evidenced by the results on the Formal Languages datasets. However, interactive probing may suffer from training stability issues in some instances. This issue might be mitigated in the future through different training methods, loss functions, or regularizations. When the interactive property is unnecessary or too complex to train, such as with the Sequential MNIST dataset, non-interactive probing may yield better results. Additionally, functionalist encoders offer the advantage of being agnostic to the precise architecture of f_θ. In more straightforward tasks like supervised property prediction, mechanistic encoders like or Layer-Wise Statics or DWSNet excel.

To date, our work is limited to relatively small RNNs. All approaches presented here can in principle be adapted in a straightforward way to other architectures, such as state space models (Gu et al., 2021), deep equilibrium models (Bai et al., 2019), or transformers (Vaswani et al., 2017). For scaling to larger models, we see the greatest potential in functionalist probing approaches. These are, in principle, agnostic to the architecture and size of the RNN (or any general sequence model), provided it is differentiable. By employing policy gradient methods, we could even process models that are impossible or prohibitively expensive to differentiate.

8. Conclusion and Future Work

We have proposed a framework that uses self-supervised learning to derive useful representations of RNN weights. Through this framework, we have trained and evaluated various weight encoder architectures. Notably, our newly proposed interactive probing approach is the only method capable of learning suitable representations for formal language tasks. This finding corroborates our theoretical results, demonstrating that interactive probing can, in certain situations, outperform non-interactive probing.

This work establishes a foundation for numerous future applications. For instance, the techniques introduced here can be applied in the context of reinforcement learning within partially observable environments, where they can facilitate policy representation and improvement (see e.g., Raileanu et al. (2020); Faccio et al. (2020; 2022b)), as well as exploration and skill discovery (Herrmann et al., 2022). Additionally, RNN weight representations could be beneficial in meta-learning and few-shot learning scenarios. Another field where our work might be useful is the mechanistic interpretability of sequence models (e.g., (Weiss et al., 2018; Olsson et al., 2022)). Typically, it involves a lot of labor-intensive reverse engineering. Training models to create meaningful representations of RNN (or general sequence model) weights may assist or even partially automate these efforts. Our functionalist probing approaches have the potential to extract information not only from populations of small models but also from large foundational models. This can be achieved by conditioning either the encoder or the foundational model itself on a specific task, a concept that aligns with discussions in previous research (Schmidhuber, 2015; Zeng et al., 2022; Zhuge et al., 2023).

Acknowledgements

We thank Louis Kirsch, Piotr Piekos, Aditya Ramesh, and Wenyi Wang for insightful discussions. This work was supported by the ERC Advanced Grant (no: 742870). We also thank NVIDIA Corporation for donating a DGX-1 as part of the Pioneers of AI Research Award.

Impact Statement

This paper introduces research aimed at advancing the field of Machine Learning. While acknowledging numerous potential societal consequences, we believe that none need to be specifically emphasized in here.

References

Bai, S., Kolter, J. Z., and Koltun, V. Deep equilibrium models. *Advances in neural information processing systems*, 32, 2019.

Delétang, G., Ruoss, A., Grau-Moya, J., Genewein, T., Wenliang, L. K., Catt, E., Cundy, C., Hutter, M., Legg, S., Veness, J., et al. Neural networks and the chomsky hierarchy. *arXiv preprint arXiv:2207.02098*, 2022.

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

Eilertsen, G., Jönsson, D., Ropinski, T., Unger, J., and Ynnerman, A. Classifying the classifier: dissecting the weight space of neural networks. *arXiv preprint arXiv:2002.05688*, 2020.

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

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

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

Gu, A., Goel, K., and Ré, C. Efficiently modeling long sequences with structured state spaces. *arXiv preprint arXiv:2111.00396*, 2021.

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

Hecht-Nielsen, R. On the algebraic structure of feedforward network weight spaces. In *Advanced Neural Computers*, pp. 129–135. Elsevier, 1990.

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

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

Hornik, K. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

Joseph, K. and N Balasubramanian, V. Meta-consolidation for continual learning. *Advances in Neural Information Processing Systems*, 33:14374–14386, 2020.

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

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

Loshchilov, I. and Hutter, F. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

Merrill, W. Sequential neural networks as automata. *arXiv preprint arXiv:1906.01615*, 2019.

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

Olsson, C., Elhage, N., Nanda, N., Joseph, N., DasSarma, N., Henighan, T., Mann, B., Askell, A., Bai, Y., Chen, A., et al. In-context learning and induction heads. *arXiv preprint arXiv:2209.11895*, 2022.

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

Ramesh, R. and Chaudhari, P. Model zoo: A growing" brain" that learns continually. *arXiv preprint arXiv:2106.03027*, 2021.

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

Schürholt, K., Kostadinov, D., and Borth, D. Self-supervised representation learning on neural network weights for model characteristic prediction. *Advances in Neural Information Processing Systems*, 34:16481–16493, 2021.

Schürholt, K., Taskiran, D., Knyazev, B., Giró-i Nieto, X., and Borth, D. Model zoos: A dataset of diverse populations of neural network models. *Advances in Neural Information Processing Systems*, 35:38134–38148, 2022.

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

Srivastava, R. K., Schmidhuber, J., and Gomez, F. Generalized compressed network search. In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference companion*, GECCO Companion '12, pp. 647–648, New York, NY, USA, 2012. ACM, ACM. ISBN 978-1-4503-1178-6. doi: 10.1145/2330784.2330902. URL http://doi.acm.org/10.1145/2330784.2330902.

Tang, H., Meng, Z., Hao, J., Chen, C., Graves, D., Li, D., Yu, C., Mao, H., Liu, W., Yang, Y., et al. What about inputting policy in value function: Policy representation and policy-extended value function approximator. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pp. 8441–8449, 2022.

Torabi, F., Warnell, G., and Stone, P. Behavioral cloning from observation. *arXiv preprint arXiv:1805.01954*, 2018.

Unterthiner, T., Keysers, D., Gelly, S., Bousquet, O., and Tolstikhin, I. O. Predicting neural network accuracy from weights. *ArXiv*, abs/2002.11448, 2020. URL https://api.semanticscholar.org/CorpusID:211506753.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. In *Advances in neural information processing systems*, pp. 5998–6008, 2017.

Weiss, G., Goldberg, Y., and Yahav, E. Extracting automata from recurrent neural networks using queries and counterexamples. In *International Conference on Machine Learning*, pp. 5247–5256. PMLR, 2018.

Xu, D., Wang, P., Jiang, Y., Fan, Z., and Wang, Z. Signal processing for implicit neural representations. *Advances in Neural Information Processing Systems*, 35:13404–13418, 2022.

Yun, C., Bhojanapalli, S., Rawat, A. S., Reddi, S. J., and Kumar, S. Are transformers universal approximators of sequence-to-sequence functions? *arXiv preprint arXiv:1912.10077*, 2019.

Zare, M., Kebria, P. M., Khosravi, A., and Nahavandi, S. A survey of imitation learning: Algorithms, recent developments, and challenges. *arXiv preprint arXiv:2309.02473*, 2023.

Zeng, A., Attarian, M., Ichter, B., Choromanski, K., Wong, A., Welker, S., Tombari, F., Purohit, A., Ryoo, M., Sindhwani, V., et al. Socratic models: Composing zero-shot multimodal reasoning with language. *arXiv preprint arXiv:2204.00598*, 2022.

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

Zhuge, M., Liu, H., Faccio, F., Ashley, D. R., Csordás, R., Gopalakrishnan, A., Hamdi, A., Hammoud, H. A. A. K., Herrmann, V., Irie, K., et al. Mindstorms in natural language-based societies of mind. *arXiv preprint arXiv:2305.17066*, 2023.

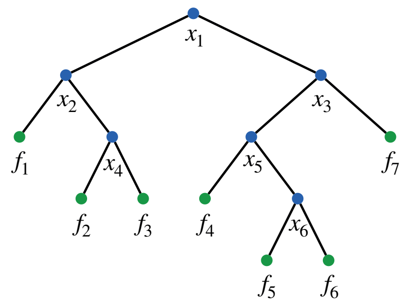

Figure 9: A binary tree constructed as described in the proof of Theorem 3.1. Giving the inputs x_j corresponding to all branching nodes to a function allows to uniquely identify it.

A. Theoretical results

Lemma A.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 A.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 A.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 9) 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.

Proposition A.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 $1, \ldots, L - 1$. □

Proposition A.4. *There exist function sets 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

$$L_{m_2,m_3,\ldots,m_k} := \{a_1^n a_2^{n+m_2} a_3^{n+m_3} \ldots a_k^{n+m_k} | n \in \mathbb{N}\}, \tag{2}$$

with $m_2, \ldots, m_k \in \mathbb{N}$ and $a_1, \ldots, a_k$ being the letters or 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 $L_{2,1}$ is the string $a_1 a_2 a_2 a_2 a_3 a_3$.

Let $G_L := \{L_{m_2,\ldots,m_k} | m_2, \ldots, m_k \in \{1, \ldots, M\}\}$, i.e., a set of such languages with different parameters m_i. G_L contains $M^{(k-1)}$ languages. To each language $L_{m_2,\ldots,m_k}$, we can assign a unique generative function $g_{m_2,\ldots,m_k}$. 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 prefix of string from the language, it returns the empty string ϵ. For example, $g_{2,1}(a_1) = (a_1, a_2)$, $g_{2,1}(a_1 a_2 a_2) = (a_2)$, and $g_{2,1}(a_1 a_2 a_2 a_3) = \epsilon$. Our function set D_L is a set of such generative functions, $D_L := \{g_{m_2,\ldots,m_k} | m_2, \ldots, m_k \in \{1, \ldots, M\}\}$.

For an interactive Interrogator, there is a simple strategy to identify a given function $g_C \in D_L$ using $M \cdot (k-1)$ interactions: The first input is the string $a_1 a_2$. 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 to be 1 from the first input string. This is repeated until ϵ is returned, which is after a maximum of $M \cdot (k-1)$ calls to g_C. The last probing input string will be of the form $a_1 a_2^{r_2} \ldots a_k^{r_k}$, from which the language can easily be inferred to be $L_{r_2-1,\ldots,r_k-1}$.

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, exponentially many calls to g_C are needed to identify it. Assuming $n = 1$, there are M^{k-2} unique prefixes for the first token a_k. Each of these prefixes is only allowed in M languages $L_{m_2,\ldots,m_{k-1},\cdot}$, namely the ones with fixed $m_1, \ldots, m_{k-1}$. 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, $M^{k-2}(M-1)$ different inputs have to be given to f_C. Only then it is guaranteed that the only informative string about the unknown value of m_k, namely $a_1 a_2^{m_2} \ldots a_k^{m_k}$, is among the probing inputs. It follows that to determine all values $m_2 \ldots, m_k$ and identify the exact language of g_C, a total of $\sum_{b=2}^{k-2} M^b(M-1) = M^{k-1} - M^2$ inputs are needed.

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 $O(M^k)$. □

B. DWSNet Details

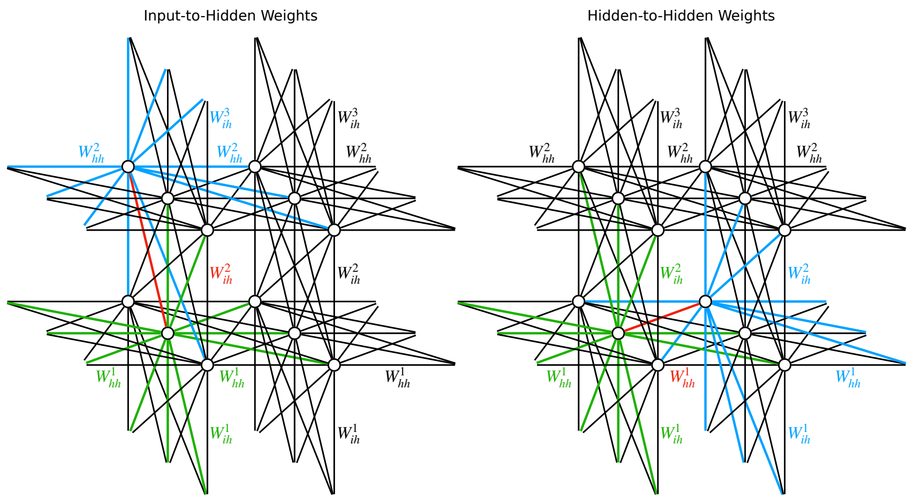

Figure 10: All connections from and to a single weight of an RNN in an input-to-hidden weight matrix (left) or hidden-to-hidden weight matrix (right). They must all be taken into account to ensure the equivariance of DWSNets for RNN weights.

The implementation of our permutation equivariant weight space layer for RNNs is based on the work of Zhou et al. (2023). Unlike feedforward networks, RNNs feature two distinct types of weight matrices: those mapping inputs to hidden states (W_{ii}^l for layer l) and those mapping hidden states to other hidden states (W_{hh}^l). To ensure the equivariance of weight space feature maps, it is crucial to consider all incoming and outgoing connections for both types of matrices (as illustrated in Figure 10). We define an equivariant linear function H_i for the input to hidden weights as:

$$
\begin{aligned}
H_i(W_{ih})^l_{jk} =& \left(\sum_s a^{l,s}_{ih} (W^l_{ih})_{*,*} + b^{l,s}_{ih} (W^l_{hh})_{*,*} \right) \\
&+ c^{l,l}_{ih} (W^l_{ih})_{*,k} + c^{l,l-1}_{ih} (W^{l-1}_{ih})_{k,*} + d^{l,l}_{ih} (W^l_{ih})_{j,*} + d^{l,l+1}_{ih} (W^{l+1}_{ih})_{*,j} \\
&+ e^{l,l}_{ih} (W_{hh})^l_{*,k} + e^{l,l-1}_{ih} (W_{hh})^{l-1}_{k,*} + f^{l,l}_{ih} (W^l_{hh})_{j,*} + f^{l,l-1}_{ih} (W^{l-1}_{hh})_{*,j} \\
&+ g^l_{ih} (W^l_{ih})_{jk}.
\end{aligned}
$$

The learnable parameters are $a_{ih}, b_{ih}, c_{ih}, d_{ih}, e_{ih}, f_{ih}$ and g_{ih}. For hidden to hidden weights, the equivariant linear function H_h is:

$$
\begin{aligned}
H_h(W_{hh})^l_{jk} =& \left(\sum_s a^{l,s}_{hh} (W^l_{ih})_{*,*} + b^{l,s}_{hh} (W^l_{hh})_{*,*} \right) \\
&+ c^{l,l}_{hh} (W^l_{hh})_{*,k} + c^{l,l-1}_{hh} (W^{l-1}_{hh})_{k,*} + d^{l,l}_{hh} (W^l_{hh})_{j,*} + d^{l,l+1}_{hh} (W^{l+1}_{ih})_{*,j} \\
&+ e^{l,l}_{hh} (W_{ih})^l_{*,k} + e^{l,l+1}_{hh} (W_{ih})^{l+1}_{k,*} + f^{l,l}_{hh} (W^l_{hh})_{j,*} + f^{l,l+1}_{hh} (W^{l+1}_{hh})_{*,j} \\
&+ g^l_{hh} (W^l_{hh})_{jk},
\end{aligned}
$$

with learnable parameters $a_{hh}, b_{hh}, c_{hh}, d_{hh}, e_{hh}, f_{hh}$ and g_{hh}. In LSTMs, the weight matrices for the input, output, forget, and cell gates are treated as four channels of the weight space input feature map.

Given that an RNN is characterized by the mapping from input and hidden states to output and updated hidden states by a single RNN cell, the universality proofs by Navon et al. (2023) are applicable. Consequently, under similar regularity conditions, our adapted DWSNets can approximate any function within the weight space.

C. Dataset Details

The process for creating the Formal Languages and Sequential MNIST RNN weight datasets is consistent across all instances. In each of the 1000 training iterations, a new LSTM network is initialized, and a task is chosen at random. This model undergoes training for a total of 20000 steps using the AdamW optimizer (Loshchilov & Hutter, 2017), with a weight decay of 10^{-4} and a batch size of 32. We implement a piece-wise linear learning rate schedule that adjusts the learning rate between $[0.01, 0.003, 0.0003]$ at steps $[0, 10000, 20000]$. The model weights are saved as a datapoint at the training steps $[0, 100, 200, 500, 1000, 2000, 5000, 10000, 20000]$, along with 100 rollout sequences and various attributes (such as the task, training step, and performance metrics, including Sequential MNIST validation and training loss).

D. Experiment Details

All experiments, both pre-training and downstream property prediction tasks, are run for 100k training steps with an early stopping criterion based on validation loss performance. All MLPs use ReLU activation functions within their hidden layers. The hyperparameters for the self-supervised pre-training phase are detailed in Table 3, the additional settings for the prediction phase are listed in Table 4. Tables 5-10 report the hyperparameters for six distinct RNN weight encoder architectures, in addition to specifying the total encoder model sizes for both the Formal Languages and Sequential MNIST studies. The probing encoders use a probing sequence length of 22 for Formal Languages and 51 for Sequential MNIST.

Table 3: Pre-Training Hyperparameters

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 4: Additional Downstream Hyperparameters

Hyperparameter	**Value**
Predictor MLP Size	128
Predictor MLP #Layers	2

Table 5: Layer-Wise Statistics Hyperparameters

Hyperparameter	**Value**
MLP Hidden Size	768
MLP #Layers	3
#Parameters Formal Languages	*1248016*
#Parameters Sequential MNIST	*1248016*

Table 6: Flattened Weights Hyperparameters

Hyperparameter	**Value**
MLP Hidden Size	128
MLP #Layers	3
#Parameters Formal Languages	*1797264*
#Parameters Sequential MNIST	*1978000*

Table 7: Parameter Transformer Hyperparameters

Hyperparameter	**Value**
Size	128
Transformer MLP Hidden Size	512
#Layers	6
#Heads	2
#Parameters Formal Languages	*1276688*
#Parameters Sequential MNIST	*1278480*

Table 8: DWSNet Hyperparameters

Hyperparameter	**Value**
# Channels	48
# Layers	4
#Parameters Formal Languages	*1279736*
#Parameters Sequential MNIST	*1282400*

Table 9: Non-Interactive Probing Hyperparameters

Hyperparameter	**Value**
E_R Hidden Size	256
E_R #Layers	2
E_I & E_O Hidden Size	128
E_I & E_O #Layers	1
#Parameters Formal Languages	*1241718*
#Parameters Sequential MNIST	*1254016*

Table 10: Interactive Probing Hyperparameters

Hyperparameter	**Value**
E_R Hidden Size	256
E_R #Layers	2
E_I & E_O Hidden Size	128
E_I & E_O #Layers	1
#Parameters Formal Languages	*1235830*
#Parameters Sequential MNIST	*1240960*

E. Additional Results

Figure 11 shows the emulation performance of different emulators A_ξ on the Formal Languages datasets. We observe overfitting on the training data of the non-interactive probing and parameter transformer encoders. Note, however, that this occurs despite the use of early stopping, meaning the models evaluated are those with the lowest validation loss.

Figure 12 shows the emulation performance on the Sequential MNIST datasets. There we can observe an interesting effect: in the Parameter Transformer setup, A_ξ always classifies the digits with high accuracy, even when conditioned on low-performing RNNs. We suggest the following explanation: from Figures 20, 22 and 23 it is clear that for Sequential MNIST, the parameter transformer fails to learn informative representations. What does the emulator A_ξ learn in this case? For Sequential MNIST, the target distributions y typically assign a high probability to the correct digit. Therefore, the emulator is incentivized to learn a rotation-invariant MNIST classifier that will perform well on any input, regardless of the conditioning. However, there is an additional nuance: the reverse KL divergence loss function prefers high entropy output distributions from the emulator. This means that while the emulator's performance in terms of classification loss would be poor (since high entropy distributions generally result in high losses), its performance in terms of classification accuracy can still be good even with a high entropy distribution, as long as the correct digits have a probability that is at least slightly higher than the other ones.

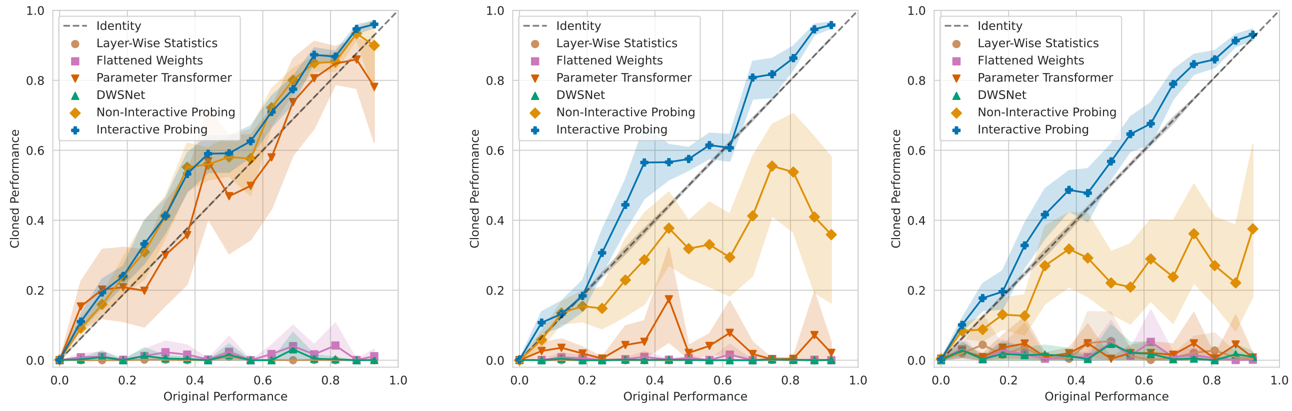

Figure 11: Formal Languages original performance vs. emulated performance. Training (left), validation (middle), and OOD test data (right).

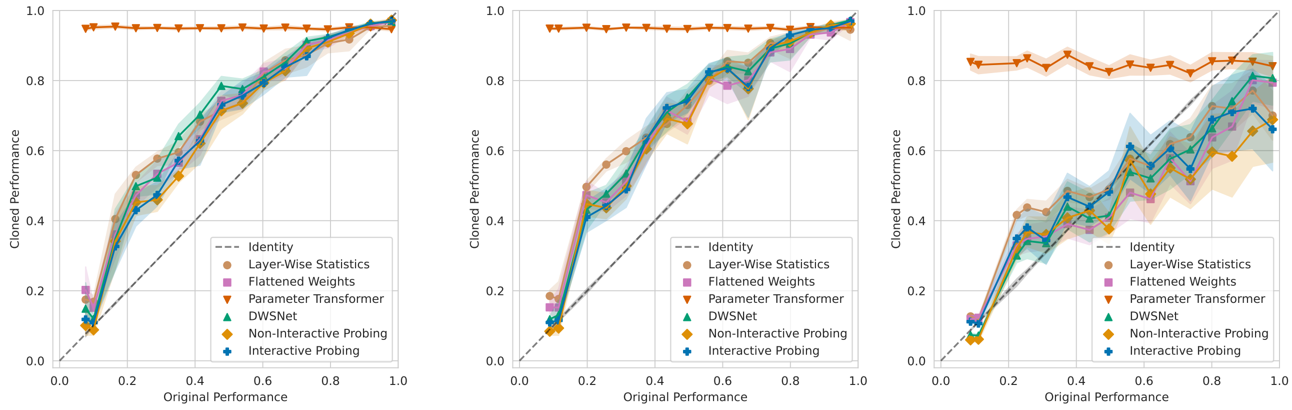

Figure 12: Sequential MNIST original performance vs. emulated performance. Training (left), validation (middle), and OOD test data (right)

E.1. Probing Sequence Number and Length

The probing encoders presented in this work can utilize a fixed number of parallel probing sequences. If not specified otherwise, 8 probing sequences are used. Figures 13 and 14 demonstrate that using more probing sequences correlates with lower loss. Interactive probing proves highly effective for RNNs applied to the Formal Languages dataset, though it faces challenges with training stability. In principle, one probing sequence is sufficient to solve the task, as evidenced by the top-performing model among 15 seeds achieving very low loss regardless of the number of probing sequences. Nonetheless, training stability improves with an increase in probing sequences.

For RNNs analyzing the Sequential MNIST dataset, the optimal length for probing sequences is 51 (49 plus BOS and EOS), matching the number of tiles representing a full digit. In the case of Formal Languages, a probing sequence length of 22 is sufficient for interactive probing to identify all languages of the dataset. Figure 15 illustrates that shorter probing sequences yield poorer results, while longer ones do not increase performance and can even harm training stability. Therefore, the length of the probing sequence should be as long as necessary but no longer.

Figure 16 displays the probing sequences generated by different Interactive Probing encoders for a specific RNN that produces strings from languages $L_{-3,-1,1}$ (the encoders for this figure learn only one probing sequence). A length of 7 is notably insufficient, failing to learn any meaningful probing sequence. However, at sequence lengths of 12, 22, and 42, the encoder successfully learns to probe actual strings of varying lengths from $L_{-3,-1,1}$. Note that while there is a string of length less than 12 for this particular language, this does not apply to all languages in the dataset.

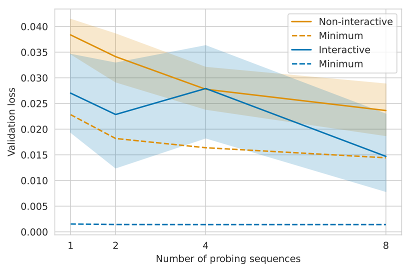

Figure 13: Formal Languages validation loss vs. number of probing sequences

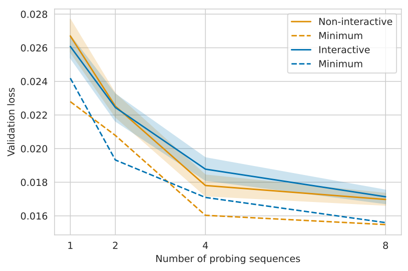

Figure 14: Sequential MNIST validation loss vs. number of probing sequences

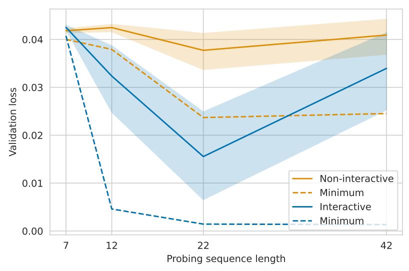

Figure 15: Formal Languages validation loss vs. length of probing sequences

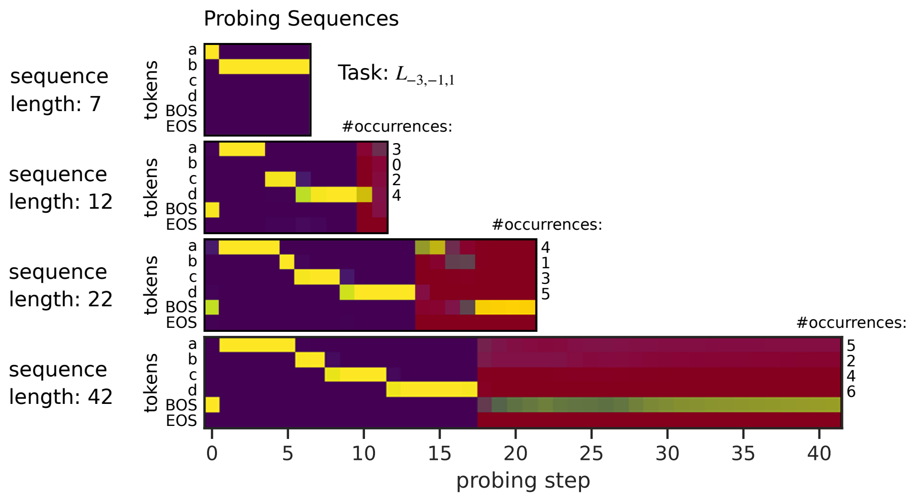

Figure 16: Learned interactive probing sequences for a formal language

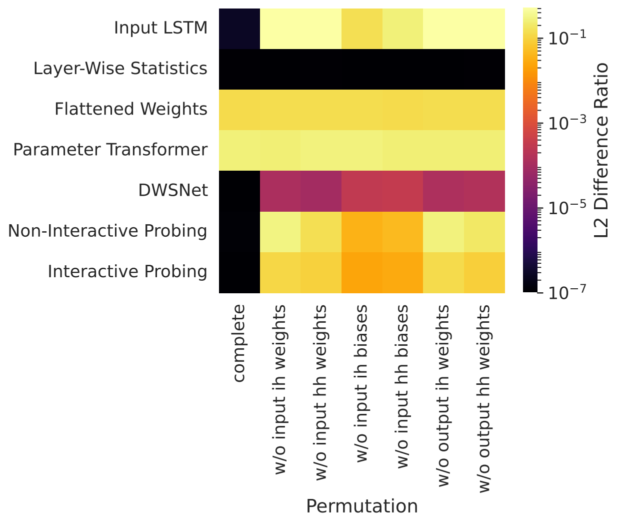

Figure 17: Various encoder architectures, with respect to either complete or incomplete permutation of hidden neurons, are analyzed. A black cell indicates that the output remains unchanged, whereas a bright cell signifies that the output changes to a similar extent as it would have with completely different weights. These results pertain to untrained encoders.

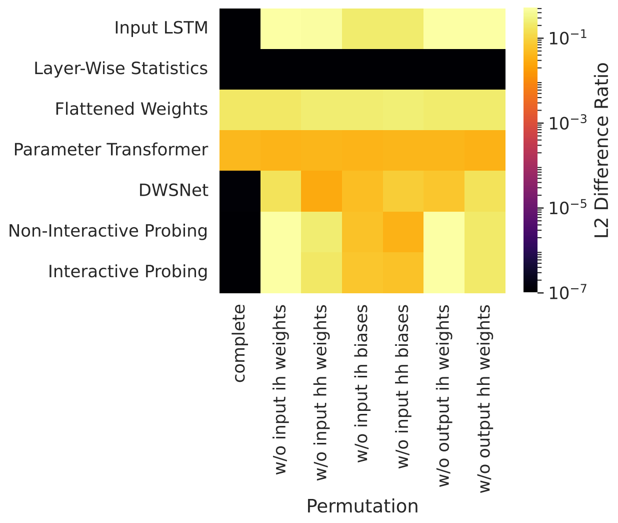

Figure 18: Invariance properties of the input LSTM f_θ and of fully trained encoders. Although the general invariance properties of the architectures remain unchanged, we observe that DWSNet becomes more sensitive to incorrect or incomplete permutations.

E.2. Hidden Neuron Permutation Invariance

Figures 17 and 18 illustrate the invariance of different encoder architectures towards correct (first column) and incorrect (other columns) hidden neuron permutations. Ideally, invariance should be present only in the first column. However, the Layerwise Statistics encoder shows invariance to both correct and incorrect permutations. In contrast, the Flattened Weights and Parameter Transformer encoders exhibit no invariance, neither before (Figure 17) nor after training (Figure 18). During training, DWSNet becomes less invariant to wrong and incomplete permutations. The methodology of the plot is explained below.

An LSTM consists of four gates: input gate, output gate, forget gate, and cell gate. These gates process the input data (from the previous layer) and the incoming hidden state. For a correct hidden neuron permutation in an LSTM, which preserves accuracy, all of the following elements must undergo the same permutation:

- the rows of the four input-to-hidden weight matrices
- the rows of the four hidden-to-hidden weight matrices
- the four input-to-hidden bias vectors
- the four hidden-to-hidden bias vectors
- the columns of the four input-to-hidden weight matrices of the next layer
- the columns of the four hidden-to-hidden weight matrices

In Figure 18, the top row labeled 'Input LSTM' shows $\frac{||f_{\hat{\theta}}(x)-f_\theta(x)||_2}{||f_\psi(x)-f_\theta(x)||_2}$, where $\theta \in \Theta$ represents the original weights, $\tilde{\theta}$ is the (correct or incorrect, depending on the column in the figure) permutation of θ and $\psi \in \Theta$ is an entirely different set of weights. The other rows show $\frac{||E(\hat{\theta})-E(\theta)||_2}{||E(\psi)-E(\theta)||_2}$ for different encoders E. A bright cell indicates that the permutation significantly changes the result. A black cell means that the result for $\hat{\theta}$ is unchanged compared to the original result for θ. The encoders used in Figure 18 have been trained on the Formal Languages dataset, which also provides θ and ψ. Layerwise statistics features are invariant to all neuron permutations. The Flattened Weights Encoder and the Parameter Transformer show no signs of permutation invariance, even after training. DWSNet is, due to its construction, invariant only to complete permutations. During training it gets more sensitive to wrong and incomplete permutation. The probing encoders naturally inherit the invariance properties directly from the RNN f.

E.3. Downstream Performance

Table 11 shows the Pearson correlation coefficients between pre-training validation loss, as defined in Equation 1, and the downstream prediction losses for different properties, across all encoder architectures and random seeds. This analysis offers insight into the alignment between the pre-training objective and downstream applications. There is a high correlation with formal language task prediction and with Sequential MNIST performance prediction as well as in-distribution task prediction. A lower correlation does not necessarily imply that pre-training is less effective; it may simply indicate that moderate emulation performance is sufficient for good downstream prediction results. Figures 19 and 20 provide the complete results of downstream predictions for Formal Languages and Sequential MNIST data.

Table 11: Correlation of pre-training validation loss with downstream prediction performance

Downstream Prediction	Validation	OOD Test
Formal Languages		
Performance	0.509	0.505
Task	0.985	0.987
Sequential MNIST		
Performance	0.883	0.872
Task	0.864	0.521
Generalization Gap	0.776	0.642
Training Step	0.762	0.486

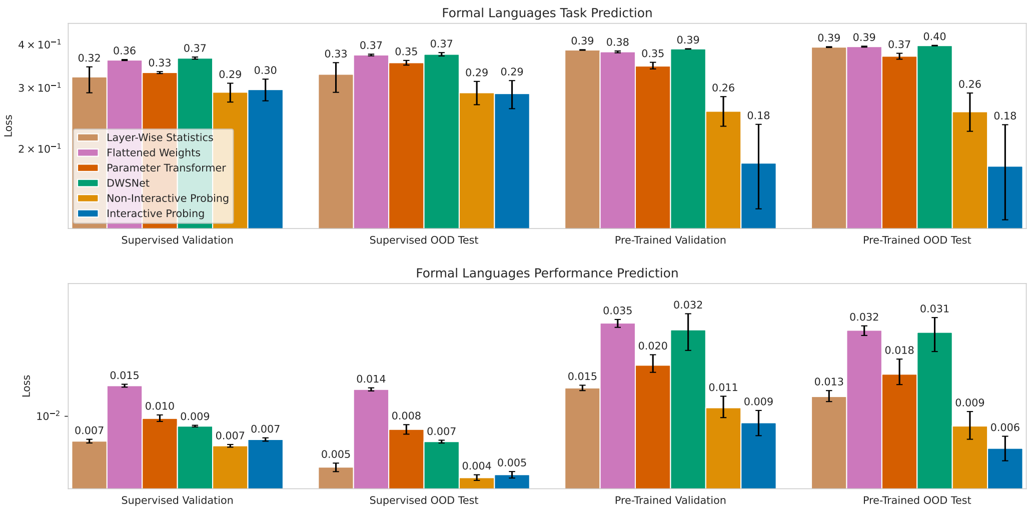

Figure 19: Formal Languages downstream performance on task and performance prediction.

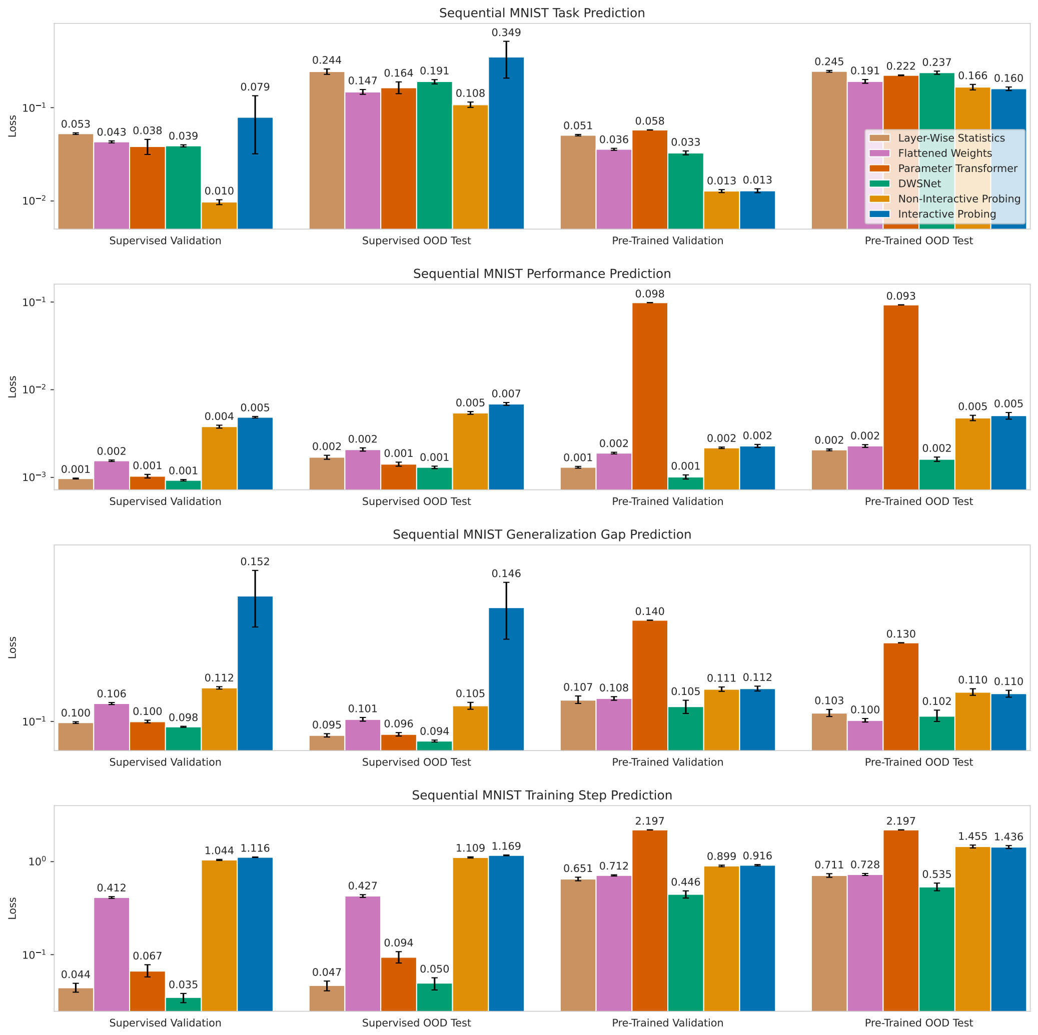

Figure 20: Sequential MNIST downstream performance on task prediction, performance prediction, generalization gap, and training step prediction.

E.4. Learned Embedding Spaces

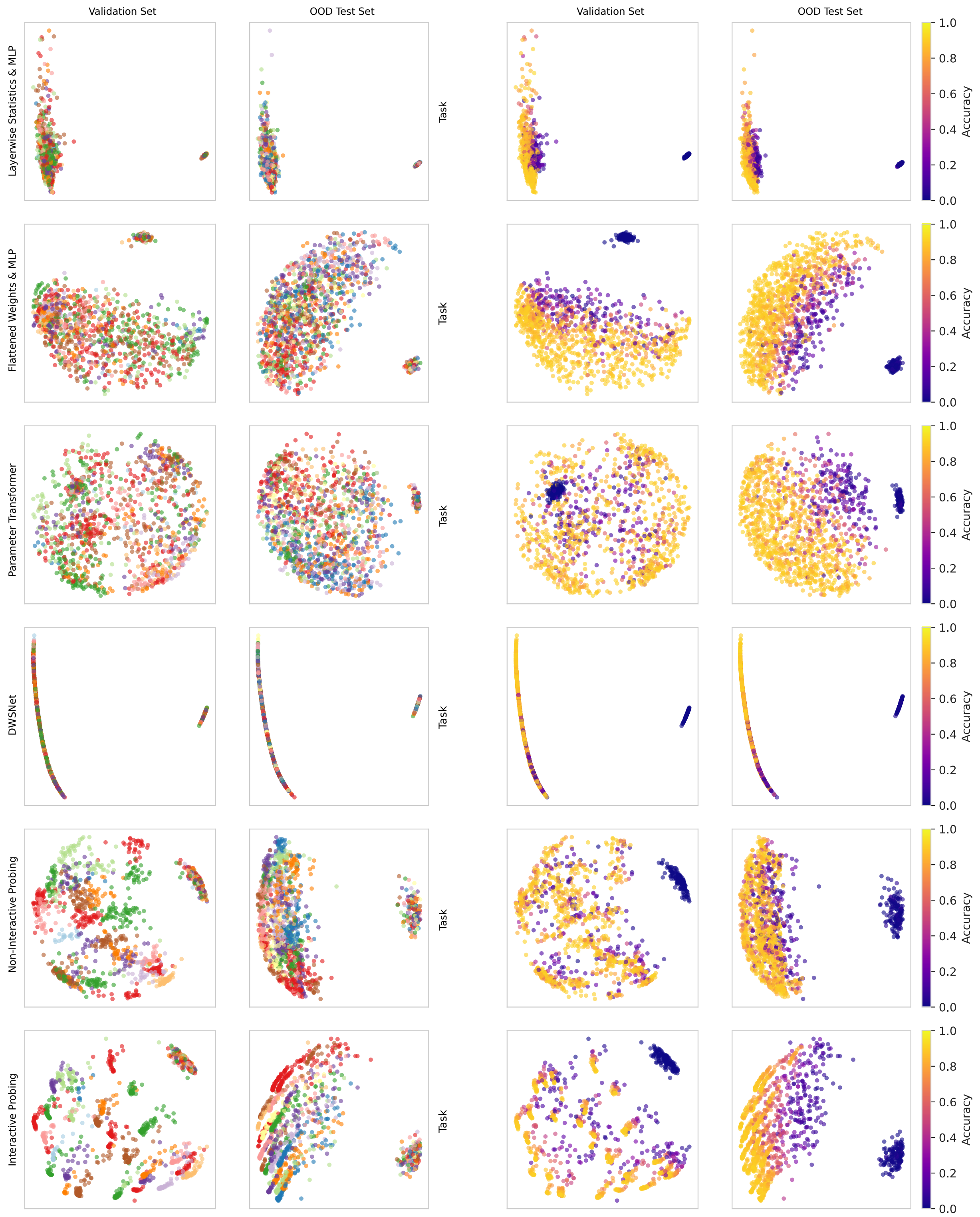

Figure 21: Visualization of the embedding spaces for Formal Languages.

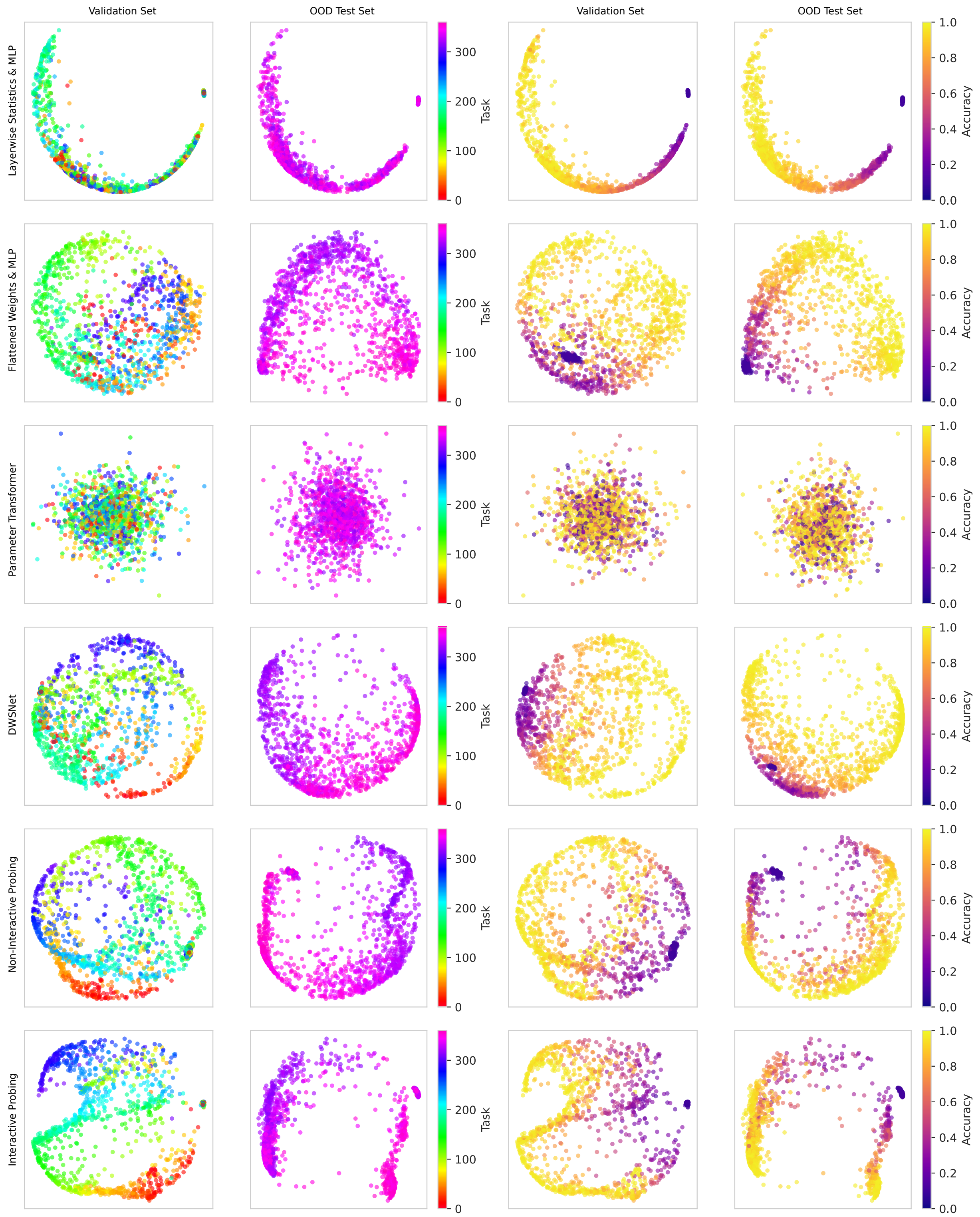

Figure 22: Visualization of the embedding spaces for Sequential MNIST (colored by task and return).

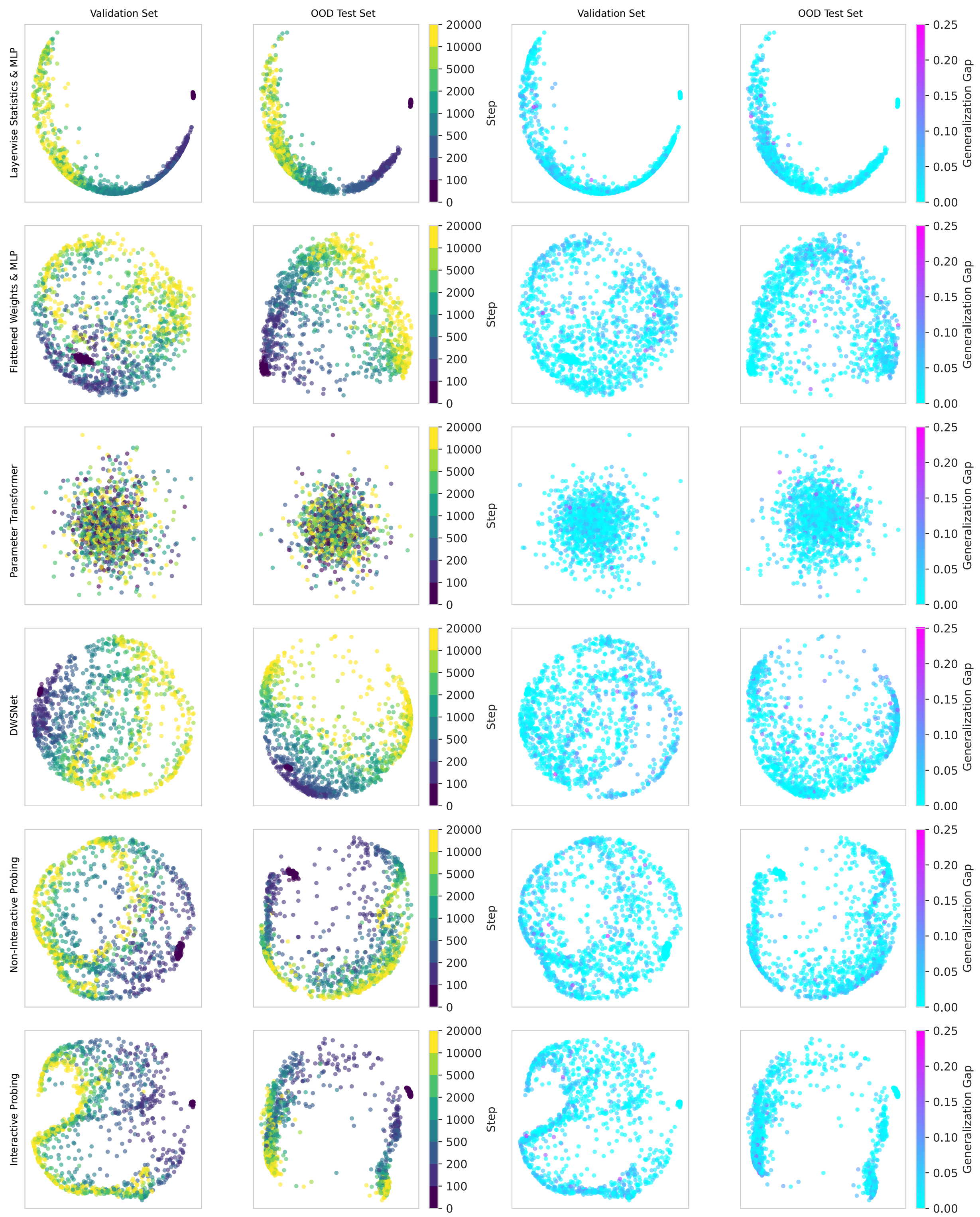

Figure 23: Visualization of the embedding spaces for Sequential MNIST (colored by training step and generalization gap).