Learning Universal Predictors

Jordi Grau-Moya [*1] **Tim Genewein** [*1] **Marcus Hutter** [*1] **Laurent Orseau** [*1] **Grégoire Déletang** [1] **Elliot Catt** [1] **Anian Ruoss** [1] **Li Kevin Wenliang** [1] **Christopher Mattern** [1] **Matthew Aitchison** [1] **Joel Veness** [1]

Abstract

Meta-learning has emerged as a powerful approach to train neural networks to learn new tasks quickly from limited data by pre-training them on a broad set of tasks. But, what are the limits of meta-learning? In this work, we explore the potential of amortizing the most powerful universal predictor, namely Solomonoff Induction (SI), into neural networks via leveraging (memory-based) meta-learning to its limits. We use Universal Turing Machines (UTMs) to generate training data used to expose networks to a broad range of patterns. We provide theoretical analysis of the UTM data generation processes and meta-training protocols. We conduct comprehensive experiments with neural architectures (e.g. LSTMs, Transformers) and algorithmic data generators of varying complexity and universality. Our results suggest that UTM data is a valuable resource for meta-learning, and that it can be used to train neural networks capable of learning universal prediction strategies.

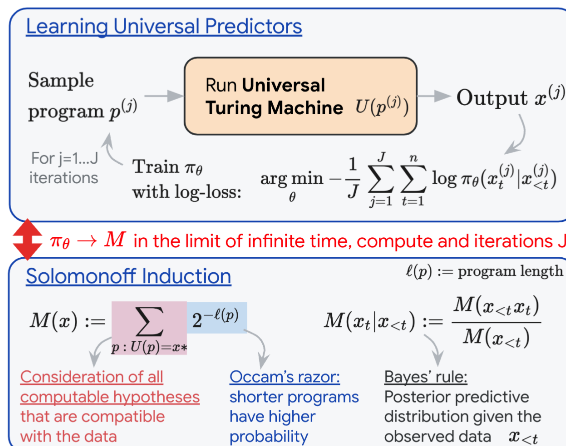


Figure 1: Summary of our meta-learning methodology.

1. Introduction

Meta-learning has emerged as a powerful approach to enable AI systems to learn new tasks quickly from limited data (Hospedales et al., 2021). By training a model on a diverse set of tasks, meta-learning encourages the discovery of representations and learning strategies that generalize to new, unseen tasks. Memory-based meta-learning (Santoro et al., 2016)—a type of meta-learning that relies on memory updates, instead of a two-level optimization procedure (Finn et al., 2017)—has been shown to be a promising way of teaching neural networks to implicitly implement Bayesian inference (Ortega et al., 2019; Mikulik et al., 2020; Genewein et al., 2023) tailored to a prior task-distribution. A key challenge in meta-learning is to design task distributions that are sufficiently broad, exposing the model to a rich variety of structures and patterns. Such broad exposure could lead to "universal" representations, enabling the system to tackle a wide range of problems and bringing us closer to the goal of artificial general intelligence (AGI).

Solomonoff Induction [1] (SI) offers a compelling theoretical foundation for constructing such an ideal universal prediction system (Solomonoff, 1964a;b) [2]. At its core, SI elegantly integrates three fundamental principles (see Figure 1). *Consideration of all computable hypotheses:* Unlike traditional approaches, SI explores the entire space of computable hypotheses (i.e. generated by a computer program) as potential explanations for observed data. *Occam's Razor:* SI assigns higher prior probabilities to simpler hypotheses with shorter descriptions. *Bayesian Updating*: With new data, SI employs Bayes' rule to refine its belief about each hypothesis. The theoretical strength of SI lies in its ability to rapidly converge on the true data-generating process, if computable (Li & Vitanyi, 1992; Hutter, 2004; Sunehag &

*Equal contribution [1]Google DeepMind, London, UK. Correspondence to: Jordi Grau-Moya <jordigrau@google.com>.

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

[1]SI arguably solved the century-old induction problem (Rathmanner & Hutter, 2011), is the basis of the Hutter prize (Hutter, 2006/2020) and has been praised by the father of AI, Marvin Minsky: "the most important discovery since Gödel".

[2]For an introduction see (Hutter et al., 2007; Hutter, 2017) and see (Hutter, 2007) for technical details.

Hutter, 2013; Li et al., 2019). Yet, a significant barrier is its practical incomputability. The exhaustive exploration of algorithmic hypotheses demands immense computational resources. To address this, approximations of SI were developed e.g. the Speed Prior (Schmidhuber, 2002; Filan et al., 2016) and the Context Tree Weighting algorithm (Willems et al., 1995; Willems, 1998; Veness et al., 2012).

To understand the power of SI, imagine a program that generates an infinite stream of data x, e.g., a fluid dynamics simulation or an AI movie generator. Let's say the length of the shortest possible version of this program—i.e. its Kolmogorov complexity (Li et al., 2019)—is N bits long. This could be approximated by removing all unnecessary elements of the program and using compression to further reduce the size. Now, feeding the data stream x to SI and letting it predict each bit, something remarkable happens: After making fewer than N prediction errors, SI predicts the future data perfectly! This occurs because SI effectively learns the underlying rules of the data-generating program. With each incorrect prediction, it eliminates a range of possible explanations, allowing it to quickly find the correct program behind the data.

In this paper we explore the potential of amortizing Solomonoff Induction into neural networks via memory-based meta-learning (see Figure 1). A key challenge is finding neural architectures and training data distributions that guide networks towards learning SI in the limit. While neural networks are theoretically capable of universal computation (Chen et al., 2017; Stogin et al., 2020; Mali et al., 2023), practical training methods (e.g., stochastic gradient descent) can limit this ability (Deletang et al., 2022). Here we simply use off-the-shelf architectures like Transformers (Vaswani et al., 2017) and LSTMs (Hochreiter & Schmidhuber, 1997), while focusing on designing a suitable data training protocol. To address this, we generate data from Universal Turing Machines (UTMs), which are fully general computers. Training on this "universal data" exposes the network to a broad space of computable patterns that guide the network towards learning universal inductive strategies.

Our key contributions are: *1) UTM data:* We use, for the first time, UTM data to meta-train neural networks. *2) Theoretical Analysis:* We provide a theoretical analysis of the UTM data generation process and training protocol that converges to SI in the limit. *3) Extensive Experiments:* We conduct comprehensive experiments with a variety of neural architectures (e.g. LSTMs, Transformers) and algorithmic data generators of varying complexity and universality. *4)* We open-sourced all our generators at **`https://github.com/google-deepmind/neural_networks_solomonoff_induction`**.

Our results show that increasing model size leads to improved performance, demonstrating that model scaling helps learning increasingly universal prediction strategies. We find that: Large Transformers trained on UTM data successfully transfer their learning to other tasks suggesting they acquired reusable universal patterns; On variable-order Markov sources, large LSTMs and Transformers achieve optimal performance, highlighting their ability to model Bayesian mixtures over programs necessary for SI.

2. Background

Notation. An alphabet $\mathcal{X}$ is a finite, non-empty set of symbols. A string $x_1x_2 \dots x_n \in \mathcal{X}^n$ of length n is denoted by $x_{1:n}$. The prefix $x_{1:j}$ of $x_{1:n}$, $j \leq n$, is denoted by $x_{\leq j}$ or $x_{<j+1}$. The empty string is denoted by ϵ. Our notation generalizes to out-of-bounds indices i.e. given a string $x_{1:n}$ and an integer $m > n$, we define $x_{1:m} := x_{1:n}$ and $x_{n:m} := \epsilon$. The concatenation of two strings s and r is denoted by sr. The expression $[\![A]\!]$ is 1 if A is true and 0 otherwise.

Semimeasures. A semimeasure is a probability measure P over infinite and finite sequences $\mathcal{X}^\infty \cup \mathcal{X}^*$ for some finite alphabet $\mathcal{X}$ assumed to be $\{0, 1\}$ (most statements hold for arbitrary finite $\mathcal{X}$). Let $\mu(x)$ be the probability that an (in)finite sequence *starts* with x. While proper distributions satisfy $\sum_{a\in\mathcal{X}} \mu(xa) = \mu(x)$, semimeasures exhibit *probability gaps* and satisfy $\sum_{a\in\mathcal{X}} \mu(xa) \leq \mu(x)$.

Turing Machines. A Turing Machine (TM) takes a string of symbols z as an input, and outputs a string of symbols x (after reading z and halting), i.e. $T(z) = x$. For convenience we define the output string at computation step s as $T^s(z) = x$ which may be the empty string ϵ. We adopt similar notation for Universal Turing Machines U. Monotone TMs (see Definition 1 below) are special TMs that can incrementally build the output string while incrementally reading the input program, which is a convenient practical property we exploit in our experiments.

Definition 1 (Monotonicity)**.** *A universal machine U is monotone if for all p, q, x, y with $U(p) = y$ and $U(q) = x$ we have that $\ell(x) \geq \ell(y)$ and $p \sqsubseteq q$ imply $y \sqsubseteq x$, where $p \sqsubseteq q$ means that p is a prefix string of q. See Appendix C for a more thorough description.*

Solomonoff Induction (SI). The optimal prediction over the next symbol x_{n+1} given an observed sequence $x_{1:n}$ is $\mu(x_{n+1}|x_{1:n}) = \mu(x_{1:n+1})/\mu(x_{1:n})$, assuming that μ is the true (but unknown) computable probability distribution over sequences. In contrast, SI predicts the next symbol x_{n+1} using a single universal semimeasure M widely known as the Solomonoff Universal Prior (see definition below).

Definition 2 ((Monotone) Solomonoff Prior)**.** *Let U be a universal monotone machine, then the Solomonoff prior is*

defined as

$$M(x) := \sum_{p:U(p)=x*} 2^{-\ell(p)}$$

with the sum is over all $p \in \{0,1\}^*$, *where the output* $x*$ *is any string that starts with* x and *the whole program* p *has been read by* U.

We can use M to construct the posterior predictive distribution $M(x_{n+1}|x_{1:n}) = \frac{M(x_{1:n}x_{n+1})}{M(x_{1:n})}$ (see Figure 1). This is equivalent to performing Bayesian inference on program space $M(x_{n+1}|x_{1:n}) = \sum_p P(p|x_{1:n})[\![U(p) = x_{1:n}x_{n+1}*]\!]$ (for prefix-free programs, and any continuation $*$ of the sequence), where $P(p|x_{1:n})$ is the Bayesian posterior over programs given the data using the prior $P(p) = 2^{-\ell(p)}$ and the zero-one likelihood $P(x|p) = [\![U(p) = x*]\!]$.

Solomonoff (1964a) showed that M converges fast (to the true μ) if the data is generated by *any* computable probability distribution μ:

$$\sum_{t=1}^{\infty} \sum_{x_{<t}} \mu(x_{<t}) \sum_{x \in \mathcal{X}} \left(\sqrt{M(x|x_{<t})} - \sqrt{\mu(x|x_{<t})} \right)^2 \leq K(\mu) \ln 2 < \infty$$

where $K(\mu) := \min_p\{\ell(p) : U(p) = \mu\}$ is the Kolmogorov complexity (Li et al., 2019) of the generator μ (represented as a bitstring) and $\ell(p)$ the program length. This can be seen when noticing that on the left-hand-side of the inequality we have an infinite sum and on the right we have a (small) constant [3]. The Solomonoff prior is essentially the best universal predictor given a choice of reference UTM.

There exists a normalized version of the Solomonoff prior (among others (Wood et al., 2013)) that is not a semimeasure but a proper measure i.e., properly normalized (see Definition 3 below). It has nicer properties when x contains incomputable sub-sequences (Lattimore et al., 2011) and maintains the convergence properties of the standard Solomonoff prior. This version of SI is of interest to us because it suited to be learned by neural models (that are also properly normalized) and exhibits more efficient sampling than semimeasures (due to no probability gap).

Definition 3 (Normalized Solomonoff Prior)**.** *For* $a \in \mathcal{X}$, *Solomonoff normalization is defined as* $M^{norm}(\epsilon) := 1$, $M^{norm}(a|x) := \frac{M(xa)}{\sum_{a \in \mathcal{X}} M(xa)} = \frac{M^{norm}(xa)}{M^{norm}(x)}$.

Algorithmic Data Generating Sources and the Chomsky Hierarchy. An algorithmic data generating source μ is simply a computable data source by, for example, a TM T fed with random inputs. There is a natural hierarchy over machines based on their memory structure known as the Chomsky hierarchy (CH) (Chomsky, 1956), which classifies sequence prediction problems—and associated automata models that solve them—by increasing complexity. There are four levels in the CH, namely, regular, context-free, context-sensitive, and recursively enumerable. Solving problems on each level requires different memory structures such as finite states, stack, finite tape and infinite tape, respectively. Note that any reasonable approximation to SI would need to sit at the top of the hierarchy.

Meta-Learning. There exists mainly two types of meta-learning i.e., model-agnostic meta-learning (MAML) (Finn et al., 2017) and—the focus in our paper—memory-based meta-learning (MBML) (Santoro et al., 2016; Ortega et al., 2019). In MAML, there are explicit outer and inner optimization loops and a diverse dataset containing data from different tasks. The outer loop optimizes (the initial parameters) across tasks and the inner loop optimizes within tasks (from the initial parameters). In MBML, memory-augmented models are trained using a single optimization loop and a diverse dataset containing data from different tasks (in our case different tasks correspond to different UTM programs). The memory is necessary to enable task adaptation at test time. The two loops are thus implicit and implemented by sampling tasks (outer loop) and data from tasks (inner loop). At test time, in contrast to MAML, there is no inner gradient optimization but automatic in-context learning driven by the memory dynamics. Note recurrent networks (e.g. RNNs, LSTMs) have explicit memory cells, whereas Transformers directly instantiate the memory in the context.

A parametric model π_θ with memory can be (memory-based) meta-trained (see Figure 1) by repeating the following steps: **1)** sample a task τ (programs in our case) from the task distribution $p(\tau)$, **2)** sample an output sequence $x_{1:n}$ from τ, and **3)** train the model π_θ with the log-loss $-\sum_{t=1}^{n} \log \pi_\theta(x_t|x_{<t})$. Ortega et al. (2019) showed that the fully trained π_θ behaves as a Bayes-optimal predictor, i.e. $\pi_\theta(x_t|x_{<t}) \approx \sum_\tau p(\tau|x_{<t})p(x_t|x_{<t},\tau)$ where $p(x_t|x_{<t},\tau)$ is the predictive distribution, and $p(\tau|x_{<t})$ the posterior. More formally, if μ is a proper measure and $D = (x^1, ..., x^J)$ are sequences cut to length n sampled from μ with empirical distribution $\hat{\mu}(x) = \frac{1}{J}\sum_{y \in D}[\![y = x]\!]$, then the log-loss

$$\begin{aligned} \text{Loss}(\theta) &:= -\frac{1}{J}\sum_{x \in D}\sum_{t=1}^{\ell(x)} \log \pi_\theta(x_t|x_{<t}) \\ &= -\frac{1}{J}\sum_{x \in D} \log \pi_\theta(x) = -\sum_{x \in \mathcal{X}^n} \hat{\mu}(x) \log p_\theta(x) \end{aligned}$$

is minimized for $\pi_\theta(x) = \hat{\mu}(x)$ if π_θ can represent $\hat{\mu}$.

[3] If instead we consider an approximation $\hat{M}$ up to accuracy ϵ_t, at time step t, then (assuming this is suitably formalized) $\sum_t \epsilon_t$ gets added to Solomonoff's bound.

3. Meta-Learning as an Approximation to Solomonoff Induction

Next we aim to provide answers to the following questions. First, *how do we generate meta-training data that allows to approximate SI?* Second, given that most architectures are trained with a limited sequence-length, *how does this affect the meta-training protocol of neural models?* Third, *can we use different program distributions (making interesting programs more likely) without losing universality?*

3.1. The Right Dataset: Estimating Solomonoff from Solomonoff Samples

Our aim here is to define a data generation process such that we obtain an approximation to M (see Figure 1) when training our model π_θ on it (assuming for now universality and essentially infinite capacity). We consider the incomputable and computable cases. All proofs can be found in the Appendix A.

Solomonoff Data Generator (Incomputable). Putting uniform random bits p on the (read-only) input tape of a monotone UTM U generates a certain distribution M of (in)finite strings x on the output tape. This is exactly Solomonoff's prior M and a semimeasure (see Section 2). Sampling from M is trivial; we just described how and coincides exactly with the standard meta-learning setup where different programs correspond to different tasks. M is equivalent to the more formal Definition 2. The following proposition shows consistency.

Proposition 4. *Let $D := (x^1, ..., x^J)$ be J (in)finite sequences sampled from a semimeasure μ (e.g. M). We can estimate μ as follows. Given*

$$\hat{\mu}_D(x) := \frac{1}{|D|} \sum_{y \in D} [\![\ell(y) \geq \ell(x) \ \wedge \ y_{1:\ell(x)} = x]\!],$$

then $\hat{\mu}_D(x) \overset{w.p.1}{\longrightarrow} \mu(x)$ for $|D| \to \infty$.

Unfortunately there are three infinities which prevent us from using M above to sample the data. There are infinitely many programs, programs may loop forever, and output strings can have infinite length. Therefore, we define the following computable version of the Solomonoff prior.

Definition 5 (Computable Solomonoff Prior)**.** *Let programs be of length $\leq L$ and stop U after s steps (denoted U^s), or if the output reaches length n. Then,*

$$M_{s,L,n}(x) := \sum_{p \in \{0,1\}^{\leq L} : U^s(p) = x*} 2^{-\ell(p)} \quad \text{if } \ell(x) \leq n \text{ and } 0 \text{ otherwise}$$

is a computable version of the Solomonoff prior and a semimeasure.

We can sample $D^J := (x^1, ..., x^J)$ from $M_{s,L,n}$ in the same trivial way as described above for M, but now the involved computation is finite. Note that all sampled strings have length $\leq n$, since $M_{s,L,n}(x) := 0$ for $\ell(x) > n$. Consistency of meta-training data is shown next.

Proposition 6. *Let now $D^J := (x^1, ..., x^J)$ be samples from the measure $M_{s,L,n}$. Given*

$$\hat{M}_{D^J}(x) = \frac{1}{J} \sum_{y \in D^J} [\![\ell(y) \geq \ell(x) \ \wedge \ y_{1:\ell(x)} = x]\!]$$

then, $\hat{M}_{D^J}(x) \ \longrightarrow \ M_{s,L,n}(x)$ for $J \to \infty$.

We get better approximations to the Solomonoff prior M as we increase the UTM steps s, the program length L, the sequence length n, and the number of samples J. More formally, since

$$M(x) = \lim_{s,L,n \to \infty} M_{s,L,n}(x) = \sup_{s,L,n} M_{s,L,n}(x),$$

we have that $\hat{M}_{D^J} \to M$ for $s, L, n, J \to \infty$. Note that D^J depends on s, L, n, but this can easily be avoided by choosing $s(j), L(j), n(j)$ to be any functions tending to infinity, and sampling x^j from $M_{s(j),L(j),n(j)}(x)$ for $j = 1, 2, 3, ...$

Remark 7. *Although $M_{s,L,n}$ is computable, it still suffers from two inconveniences. First, sampling from it is inefficient because it is a semimeasure and exhibits a probability gap. Second, we need to differentiate whether programs halt or end up in a infinite non-printing loop (to fill the probability gap with "absorbing" tokens when training). We can bypass these inconveniences by estimating the normalized and computable Solomonoff prior combining Definitions 3 and 5.*

We can estimate the (computable) normalized Solomonoff prior, $M^{norm}_{s,L,n}(x)$, by the following.

Proposition 8. *Using the definitions from Proposition 6 we have that*

$$\hat{M}^{norm}_{s,L,n}(x_t | x_{<t}) \ = \ \frac{\sum_{y \in D^J} [\![\ell(y) \geq t \ \wedge \ y_{1:t} = x_{1:t}]\!]}{\sum_{y \in D^J} [\![\ell(y) \geq t \ \wedge \ y_{<t} = x_{<t}]\!]} \overset{J \to \infty}{\longrightarrow} \ M^{norm}_{s,L,n}(x_t | x_{<t})$$

Then, we can take the product over $t = 1, ..., n$ to obtain $\hat{M}^{norm}_{s,L,n}(x) \to M^{norm}_{s,L,n}(x) \to M^{norm}(x)$.

Summary. Propositions 4, 6 and 8 state that the data generated by the Solomonoff Data Generator and their respective variants (computable and normalized computable) are statistically consistent, and that meta-training on this data would make an estimator converge to their respective Solomonoff version (under realizability and learnability assumptions).

3.2. Training Models on Solomonoff Data using Fixed-Sequence Lengths

Most neural models (especially Transformers) require training sequences of fixed length n. Due to this, we require a slight modifications to the loss function for shorter-than-n sequences to maintain convergence to SI. We drop s, L, n from $M^{\cdots}_{s,L,n}$ since what follows holds for infinite as well as finite values. We focus on describing the training protocol that converges to the normalized version of Solomonoff, M^{norm}. We refer readers interested in the standard unnormalized version (M) to the Appendix B.

Normalized Solomonoff M^{norm} with Neural Networks. To converge to M^{norm}, we pad the x^j in D^J to length n with arbitrary symbols from $\mathcal{X}$, and cut the log-loss short at $\ell(x^j)$. When doing so, the log-loss takes the form (see Appendix B.1 for derivation that uses Proposition 8):

$$\text{Loss}(\theta) \;=\; -\sum_{t=1}^{n}\sum_{x_{<t}}\Bigg(\Big(\sum_{x_t}\hat{M}_{D^J}(x_{1:t})\Big) \Big(\sum_{x_t}\hat{M}^{norm}(x_t|x_{<t})\log\pi_\theta(x_t|x_{<t})\Big)\Bigg) \quad (1)$$

In this form, it is easy to see how the last bracket, and hence the loss, is minimized for $\pi_\theta(x_t|x_{<t}) = \hat{M}^{norm}(x_t|x_{<t})$, as desired. By the chain rule this implies that the neural model $\pi_\theta(x)$ converges to $\hat{M}^{norm}(x)$. Note that $\text{Loss}(\theta)$ does *not* depend on the padding of x^j, so any padding leads to the same gradient and same solution.

Under the (unrealistic) assumptions that the neural model has the capacity to represent $\hat{M}^{\cdots}$, and the learning algorithm can find the representation, this (tautologically) implies that the neural model distribution π_θ converges to $\hat{\mu} = \hat{M}^{\cdots}$. Similarly, if the neural model is trained on x^j sampled from $M^{\cdots}_{s(j),L(j),n}(x)$ for $j = 1, 2, 3, ...$, it converges to $M^{\cdots}_{\infty,\infty,n}$. For a neural model with context length n increasing over time, even $\hat{M}^{\cdots} \to M^{\cdots}_{\infty,\infty,\infty}$ could be possible. Though theoretically possible, there are many practical challenges that need to be surmounted to achieve this, one of them being how to efficiently sample programs.

3.3. Solomonoff from Non-Uniform Samples

For practical purposes, sampling from non-uniform (possibly learned) distribution over programs can be advantageous for efficiency. For our BrainPhoque UTM (that we use in our experiments later) it increases the yield of 'interesting' programs by a factor of 137 (see Appendix Table 4). Below we show this can be done without any concerns on losing universality.

Let Q be a probability measure on $\mathcal{X}^\infty$, with shorthand $Q(q) := Q(\Gamma_q)$, the Q-probability that a sequence starts with q, where $\Gamma_q := \{\omega \in \mathcal{X}^\infty : q \sqsubseteq \omega\} = q\mathcal{X}^\infty$. We define the *generalized Solomonoff semimeasure* as

$$M_T^Q(x) \;:=\; \sum_{q:T(q)=x*} Q(q)$$

with special case $M_U(x) \;:= \sum_{q:U(q)=x*} 2^{-\ell(q)}$ for a universal TM $T = U$ and unbiased coin flips $Q(q) = 2^{-\ell(q)}$. M_U is strongly universal in the sense that it is a Bayesian mixture over all lower semi-computable semimeasures (Wood et al., 2011). Next, we show that under very mild conditions on Q, M_U^Q is also universal. This finding is similar to (Sterkenburg, 2017), but our proof is shorter and more self-contained.

Theorem 9 (Universality of generalized Solomonoff semimeasures)**.** *$M_U^Q(x)$ is strongly universal, provided Q is a computable measure and $Q(q) > 0\ \forall q \in \mathcal{X}^*$ and $Q(q_{1:n}) \to 0$ for $n \to \infty$. More precisely, for all universal monotone TM U and all Q with the above properties, there exists a universal MTM V (as constructed in the proof) s.th. $M_U^Q(x) = M_V(x)\ \forall x$. Proof in Appendix C.*

Note on the assumptions above. We assumed an infinite number of data points and universality (and learnablity) of the approximator, which are difficult to obtain in practice and diminish the relevance of inductive biases of neural models. For finite data, however, inductive biases are important for strong generalization. We leave out of the scope of the paper the theoretical work on the effect of the inductive bias and universality of neural models and simply provide experimental evidence of neural network performance in the next section.

4. Experimental Methodology

We aim to evaluate various neural architectures and sizes trained on UTM and two other types of algorithmically generated data for comparison and analysis.

Variable-order Markov Sources (VOMS). A k-Markov model assigns probabilities to a string of characters by, at any step t, only using the last k characters to output the next character probabilities. A VOMS is a Markov model where the value of k is variable and it is obtained using a tree of non-uniform depth. A tree here is equivalent to a program that generates data. We sample trees and meta-train on the generated data. We consider *binary* VOMS where a Bayes-optimal predictor exists: the Context Tree Weighting (CTW) predictor (Willems et al., 1995; 1997), to which we compare our models to. CTW is only universal w.r.t. n-Markov sources, and not w.r.t. all computable functions like SI. See Appendix D.2 for more intuition on VOMS, how we generate the data and how to compute the CTW Bayes-optimal predictor.

Chomsky Hierarchy (CH) Tasks. We take the 15 algorithmic tasks (e.g. arithmetic, reversing strings) from (Deletang

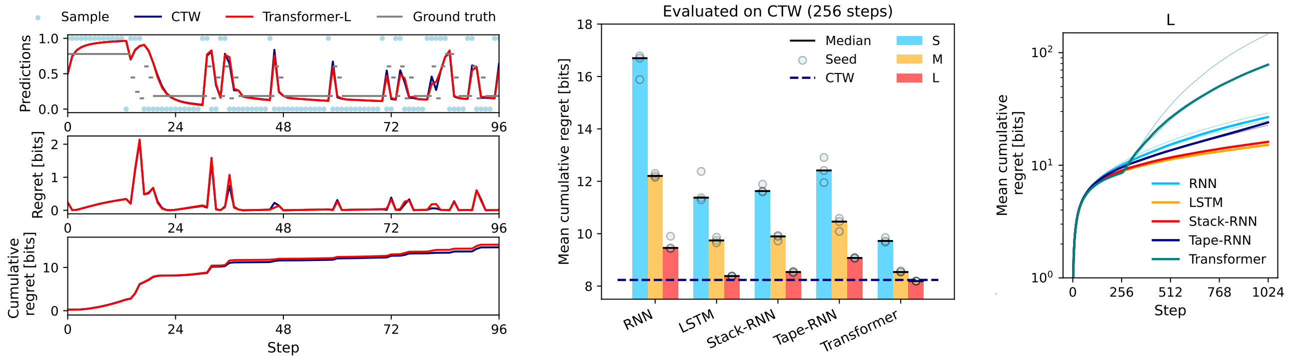


Figure 2: Evaluation on VOMS data. **Left:** Example sequence and highly overlapped predictions of Transformer-L (red) and Bayes-optimal CTW predictor (blue). Lower panels show instantaneous and cumulative regret w.r.t. the ground-truth. **Middle:** Mean cumulative regret over 6k sequences (length 256, max. CTW tree depth 24, in-distribution) for different networks (3 seeds) and sizes (S, M, L). Larger models perform better for all architectures, and the Transformer-L and LSTM-L match the optimal CTW predictor. **Right:** Length generalization (1024 steps). LSTMs generalize to longer length, whereas Transformers do not.

et al., 2022) lying on different levels of the Chomsky hierarchy (see Appendix D.3 for a description of all tasks). These tasks are useful for comparison and for assessing the algorithmic power of our models. In contrast to (Deletang et al., 2022), in which they train on *individual* tasks, we are interested in meta-training on all tasks *simultaneously*. We make sure that all tasks use the same alphabet $\mathcal{X}$ (expanding the alphabet of tasks with smaller alphabets). We do not consider transduction as in (Deletang et al., 2022) but sequence prediction, thus we concatenate inputs and outputs with additional delimiter tokens i.e. for $\{(x_i \in \mathcal{X}, y_i \in \mathcal{X})\}_{i=1}^{I}$ and delimiters ',' and ';', we construct sequences of the form $z := (x_1, y_1; x_2, y_2; \dots x_n, y_n; \dots)$. We evaluate our models using the regret (and accuracy) *only* on the output symbols, masking the inputs because they are usually random and non-informative of task performance. Denoting $\mathcal{O}_z$ the set of outputs time-indices, we compute accuracy for trajectory z as $A(z) := \frac{1}{|\mathcal{O}_z|}\sum_{t\in\mathcal{O}_z}[\![\arg\max_y \pi_\theta(y|z_{<t}) = z_t]\!]$. See Appendix D.3 for details.

Universal Turing Machine Data. Following Sections 3.1 and 3.2, we generate random programs (encoding any structured sequence generation process) and run them in our UTM to generate the outputs. A program could, in principle, generate the image of a cow, a chess program, or the books of Shakespeare, but of course, these programs are extremely unlikely to be sampled (see Figure 6 in the Appendix for exemplary outputs). As a choice of UTM, we constructed a variant of the BrainF*ck UTM (Müller, 1993), which we call BrainPhoque, mainly to help with the sampling process and to ensure that all sampled programs are valid. We set output symbols alphabet size to $|\mathcal{X}| = 17$, equal to the Chomsky tasks, to enable task-transfer evaluation. BrainPhoque has a single working tape and a write-only output tape. It has 7 instructions to move the working tape pointer (WTP), de/increment the value under the WTP (the *datum*), perform jumps and append the datum to the output. We skip imbalanced brackets to make all programs valid. While it slightly changes the program distribution, this is not an issue according to Theorem 9: each valid program has a non-zero probability to be sampled. Programs are generated and run at the same time, as described in Sections 3.1 and 3.2, for $s = 1000$ steps with 200 memory cells, with a maximum output length of $n = 256$ symbols. Ideally, we should use SI as the optimal baseline comparison but since it is uncomputable and intractable, we calculate a (rather loose, but non-trivial) upper bound on the log-loss by using the prior probability of shortened programs (removing unnecessary brackets or self-canceling instructions) that generate the outputs. See Appendix E for a full description of BrainPhoque and our sampling procedure.

Neural Predictors. Our neural models π_θ sequentially observe symbols $x_{<t}$ from the data generating source and predict the next-symbol probabilities $\pi_\theta(\cdot|x_{<t})$. We train our models using the log-loss $\text{Loss}(\theta) := -\frac{1}{n}\sum_{t=1}^{n}\log \pi_\theta(x_t|x_{<t})$, therefore maximizing lossless compression of input sequences (Delétang et al., 2023). We use stochastic gradient descent with the ADAM optimizer (Kingma & Ba, 2014). We train for 500K iterations with batch size 128, sequence length 256, and learning rate 10^{-4}. On the UTM data source, we cut the log-loss to approximate the normalized version of SI (see Section 3.2). We evaluate the following architectures: RNNs, LSTMs, Stack-RNNs, Tape-RNNs and Transformers. We note that Stack-RNNs (Joulin & Mikolov, 2015) and Tape-RNNs (Deletang et al., 2022) are RNNs augmented with a stack and tape memory, respectively, which stores and

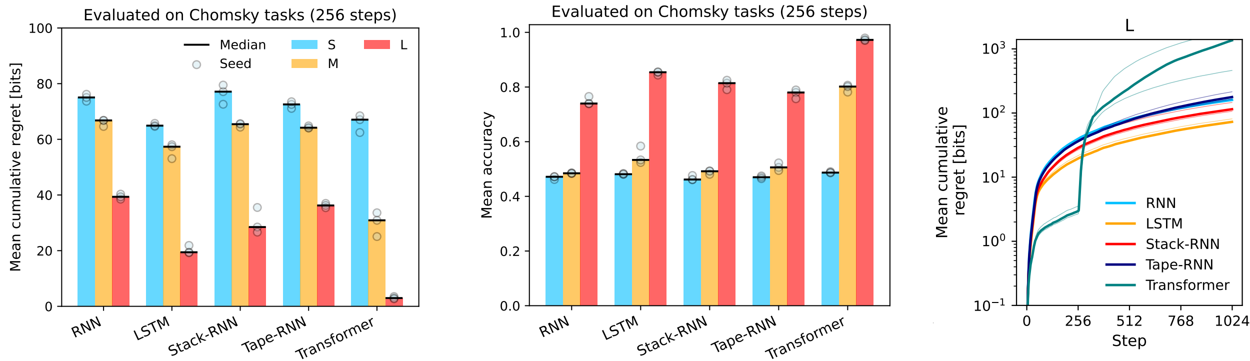


Figure 3: Evaluation on 6k sequences from the **Chomsky hierarchy tasks** (400 per task). As the model size increases, cumulative regret (**Left**) and accuracy (**Middle**) improve across all architectures. Overall, the Transformer-L achieves the best performance by a margin. **Right:** Length generalization (1024 steps). Detailed results per task are in Figure 8 on the Appendix.

manipulate symbols. This external memory should help networks to predict better, as showed in Deletang et al. (2022). We consider three model sizes (S, M and L) for each architecture by increasing the width and depth simultaneously. We train 3 parameter initialization seeds per model variation. See Appendix D.1 for all architecture details.

Evaluation Procedure. Our main evaluation metric is the *expected instantaneous regret*,

$$R_{\pi\mu}(t) := \mathbb{E}_{x_t \sim \mu}\left[\log \mu(x_t \mid x_{<t}) - \log \pi(x_t \mid x_{<t})\right]$$

(at time t), and *cumulative expected regret*, $R^T_{\pi\mu} := \sum_{t=1}^{T} R_{\pi\mu}(t)$, where π is the model and μ the ground-truth source. The lower the regret the better. We evaluate our neural models on 6k sequences of length 256, which we refer as *in-distribution* (same length as used for training) and of length 1024, referred as *out-of-distribution*.

5. Results

Variable-order Markov Source (VOMS) Results. In Figure 2 (Left) we show an example trajectory from VOMS data-source of length 256 with the true samples (blue dots), ground truth (gray), Transformer-L (red) and CTW (blue) predictions. As we can see, the predictions of the CTW predictor and the Transformer-L are overlapping, suggesting that the Transformer is implementing a Bayesian mixture over programs/trees like the CTW does, which is necessary to perform SI. In the second and third panels the instantaneous regret and the cumulative regret also overlap. Figure 2 (Middle) shows the cumulative regret of all neural predictors evaluated in-distribution. First, we observe that as model size increases (from S, M, to L) the cumulative regret decreases. The best model is the Transformer-L achieving optimal performance, whereas the worst models are the RNNs and the Tape-RNNs. The latter model likely could not successfully leverage its external memory. Note how not only Transformers, but also LSTM-L, achieve close to optimal performance. On the Right we show the out-of-distribution performance showing how transformers fail on length-generalization, whereas LSTMs perform the best. To better understand where our models struggle, we show in the Appendix F, Figures 7c and 7d, the cumulative regret averaged across trajectories from different CTW tree depths and context lengths. Models perform uniformly for all tree-depths and struggle on mid-sized context-lengths.

Chomsky Hierarchy Results. In Figure 3 (Left) we show the in-distribution performance of all our models trained on the Chomsky hierarchy tasks by means of cumulative regret and accuracy. Overall, the Transformer-L achieves the best performance by a margin. This suggests that our models, specially Transformers, have the capability of algorithmic reasoning to some extent. On the Right we show the length-generalization capabilities of models, showing how Transformers fail to generalize to longer lengths. In the Appendix (Figure 8) we show the results for each task individually.

Universal Turing Machine Results. Figure 4 (Left) shows the mean cumulative regret on the UTM task with the (loose) Solomonoff Upper Bound (UB) as a non-trivial baseline (see Section 4 for its description). In the Middle we show how all models achieve fairly good accuracy. This shows how our models are capable of learning a broad set of patterns present in the data (see example UTM trajectories in appendix Figure 6). In general, larger architectures attain lower cumulative regret and all models beat the Solomonoff upper bound. Performing better than the bound is non-trivial since the upper-bound is computed using the underlying program that generated the outputs whereas the neural models do not have this information. In Figure 10 (in the Appendix) we show the cumulative regret against program length and,

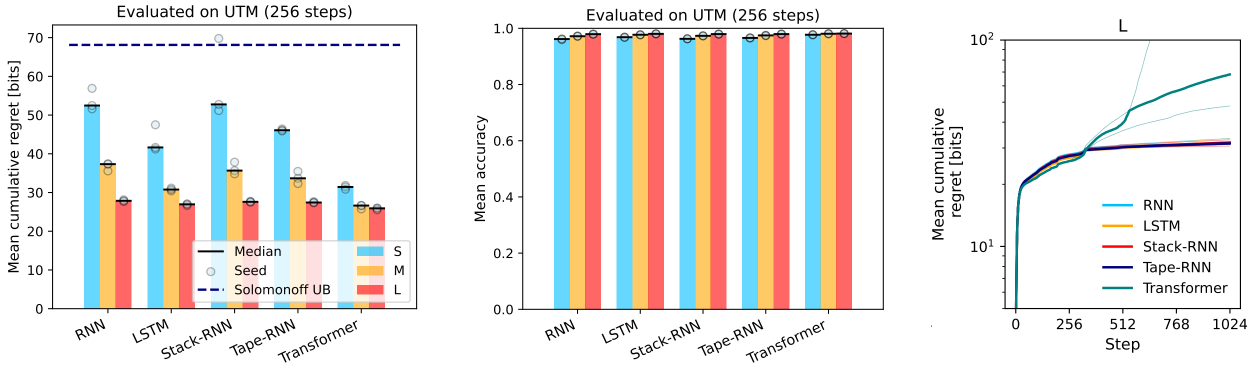


Figure 4: Evaluation on the **UTM data generator** with 6k sequences. **Left:** The larger the architecture the lower the cumulative regret. We see better performance than the non-trivial baseline Solomonoff Upper Bound (UB). **Middle:** The mean accuracy on UTM data shows the models can quickly learn UTM patterns (see Appendix Fig. 9 for additional figures). **Right:** Length generalization (1024 steps). Detailed results per program length are in Figure 10.

as expected, observe that the longer the underlying program of a sequence the higher the cumulative regret of our models, suggesting a strong correlation between program length and prediction difficulty. Remarkably, in Figure 5 we see that the Transformer networks trained on UTM data exhibit the most transfer to the Chomsky tasks and, LSTMs transfer the most to the VOMS task (compare to the 'naive' random predictor). For the VOMS, we re-trained the LSTM and Transformer models with the BrainPhoque UTM setting the alphabet size to 2 matching our VOMS task to enable comparison. All transfer results suggest that UTM data contains enough transferable patterns for these tasks.

6. Discussion

Large Language Models (LLMs) and Solomonoff Induction. The last few years the ML community has witnessed the training of enormous models on massive quantities of diverse data (Kenton & Toutanova, 2019; Hoffmann et al., 2022). This trend is in line with the premise of our paper, i.e. to achieve increasingly universal models one needs large architectures and large quantities of diverse data. LLMs have been shown to have impressive in-context learning capabilities (Kenton & Toutanova, 2019; Chowdhery et al., 2022). LLMs pretrained on long-range coherent documents can learn new tasks from a few examples by inferring a shared latent concept (Xie et al., 2022; Wang et al., 2023). They can do so because in-context learning does implicit Bayesian inference (in line with our CTW experiments) and builds world representations and algorithms (Li et al., 2023a;b) (necessary to perform SI). In fact, one could argue that the impressive in-context generalization capabilities of LLMs is a sign of a rough approximation of Solomonoff induction. The advantage of pre-trained LLMs compared to our method (training on universal data) is that LLM data (books, code, online conversations etc.) is generated by humans, and thus very well aligned with the tasks we (humans) want to solve; whereas our UTMs do not necessarily assign high probability to human tasks.

Learning the UTM. Theorem 9 of our paper (and (Sterkenburg, 2017)) opens the path for modifying/learning the program distribution of a UTM while maintaining the universality property. This is of practical importance since we would prefer distributions that assign high probability to programs relevant for human tasks. Similarly, the aim of (Sunehag & Hutter, 2014) is to directly learn a UTM aligned to problems of interest. A good UTM or program distribution would contribute to having better synthetic data generation used to improve our models. This would be equivalent to data-augmentation technique so successfully used in the machine learning field (Perez & Wang, 2017; Lemley et al., 2017; Kataoka et al., 2020). In future work, equipped with our Theorem 9, we plan study optimizations to the sampling process from UTMs to produce more human-aligned outputs.

Increasingly Universal Architectures. The output of the UTM $U^s(p)$ (using program p) requires at maximum s computational steps. Approximating $M_{s,L,n}$ would naively require wide networks (to represent many programs in parallel) of s-depth and context length n. Thus bigger networks would better approximate stronger SI approximations. If computational patterns can be reused, depth could be smaller than s. Transformers seem to exhibit reusable "shortcuts" thereby representing all automata of length T in $O(\log T)$-depth (Liu et al., 2023). An alternative way to increase the amount of serial computations is with chain-of-thought (Wei et al., 2022) (see Hahn & Goyal (2023) for theoretical results). When data is limited, inductive biases are important for generalization. Luckily it seems neural networks have

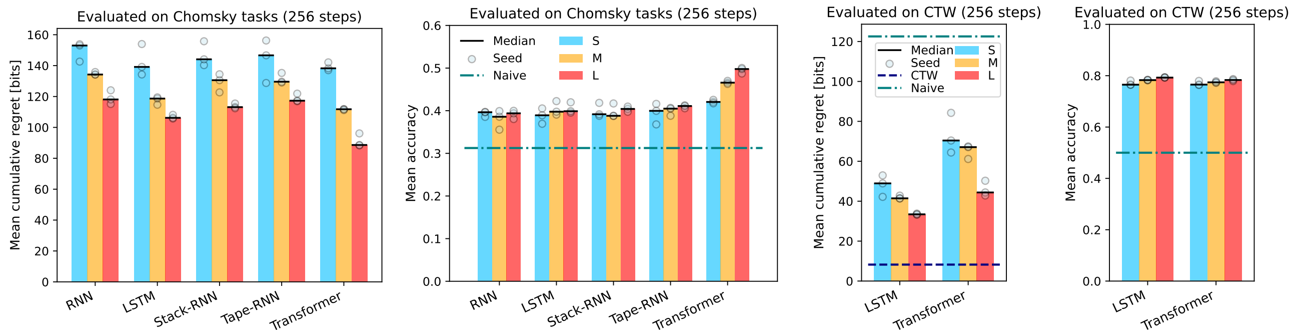


Figure 5: **Transfer learning** from *UTM-trained models* on 3k trajectories. Mean cumulative regret (**Left**) and accuracy (**Middle-Left**) of neural models trained on UTM data evaluated against the tasks of the Chosmky hierarchy. We observe a small increase in accuracy (transfer) from the Transformer models. Transfer to CTW is shown in the right two panels: **Middle-Right:** mean cumulative regret, **Right:** mean accuracy; 'Naive' is a random uniform predictor.

an implicit inductive bias towards simple functions at initialization (Dingle et al., 2018; Valle-Perez et al., 2018; Mingard et al., 2023) compatible with Kolmogorov complexity, which is greatly convenient when trying to approximate SI in the finite-data regime.

Length Generalization. We observed that our Transformer models usually perform poorly outside the sequence-length range they have been trained on (e.g. cf. Figures 2 and 12 in the Appendix). Our findings are in concordance with the quite-active area of research on length-generalization of transformers (Anil et al., 2022; Zhou et al., 2024; Ruoss et al., 2023). The main failure point (among others) seems to arise due to the positional encodings. While positional encodings help in-distribution, they do not seem to generalize well to longer sequences. In contrast, LSTMs do not present such drawbacks due to the no-use of positional encodings and because the input is processed sequentially.

Algorithmic Reasoning. In the Algorithmic Reasoning (AR) literature (Veličković et al., 2022; Ibarz et al., 2022) the aim is to train neural networks such that they perform algorithmic tasks like sorting, searching, etc. Thus, AR aims at a similar goal as we do; We want our networks to implement and execute algorithms to solve algorithmic tasks. To do this, AR typically relies on more sophisticated graph neural networks (Zhou et al., 2020), while we take more standard architectures like Transformers and LSTMs. We frame the problem as a sequence prediction problem that is universal in generality, whereas in AR typically focuses on classification. While having similar aims as AR, we believe we can provide a much deeper foundational point of view to the problem. We actually respond to the question: *"If we train an algorithmic reasoner on all possible algorithms, will it predict well?"* and our answer is: *it will approximate Solomonoff Induction which is the best universal predictor.*

Limitations. Given the empirical nature of our results, we cannot guarantee that our neural networks mimic SI's universality. Solomonoff Induction is uncomputable/undecidable and one would need infinite time to exactly match it in the limit. However, our theoretical results establish that good approximations are obtainable, in principle, via meta-training; whereas our empirical results show that is possible to make practical progress in that direction, though many questions remain open, e.g., how to construct efficient relevant universal datasets for meta-learning, and how to obtain easily-trainable universal architectures.

7. Conclusions

We aimed at using meta-learning as driving force to approximate Solomonoff Induction. For this we had to carefully specify the data generation process and the training loss so that the convergence (to various versions of SI) is attained in the limit. Our experiments on the three different algorithmic data-sources tell that: neural models can implement algorithms and Bayesian mixtures, and that larger models attain increased performance. Remarkably, networks trained on the UTM data exhibit transfer to the other domains suggesting they learned a broad set of transferable patterns. We believe that we can improve future sequence models by scaling our approach using UTM data and mixing it with existing large datasets.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

References

Anil, C., Wu, Y., Andreassen, A. J., Lewkowycz, A., Misra, V., Ramasesh, V. V., Slone, A., Gur-Ari, G., Dyer, E., and Neyshabur, B. Exploring length generalization in large language models. In Oh, A. H., Agarwal, A., Belgrave, D., and Cho, K. (eds.), *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=zSkYVeX7bC4.

Böhm, C. On a family of turing machines and the related programming language. *ICC bulletin*, 3:185–194, 1964.

Catt, E., Quarel, D., and Hutter, M. *An Introduction to Universal Artificial Intelligence*. Chapman & Hall/CRC Artificial Intelligence and Robotics Series. Taylor and Francis, 2024. ISBN 9781032607153. URL http://www.hutter1.net/ai/uaibook2.htm. 400+ pages, http://www.hutter1.net/ai/uaibook2.htm.

Chen, Y., Gilroy, S., Maletti, A., May, J., and Knight, K. Recurrent neural networks as weighted language recognizers. *arXiv preprint arXiv:1711.05408*, 2017.

Chomsky, N. Three models for the description of language. *IRE Transactions on information theory*, 2(3):113–124, 1956.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

Deletang, 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. In *The Eleventh International Conference on Learning Representations*, 2022.

Delétang, G., Ruoss, A., Duquenne, P.-A., Catt, E., Genewein, T., Mattern, C., Grau-Moya, J., Wenliang, L. K., Aitchison, M., Orseau, L., Hutter, M., and Veness, J. Language modeling is compression, 2023.

Dingle, K., Camargo, C. Q., and Louis, A. A. Input–output maps are strongly biased towards simple outputs. *Nature communications*, 9(1):761, 2018.

Elman, J. L. Finding structure in time. *Cogn. Sci.*, 1990.

Filan, D., Leike, J., and Hutter, M. Loss bounds and time complexity for speed priors. In *Artificial Intelligence and Statistics*, pp. 1394–1402. PMLR, 2016.

Finn, C., Abbeel, P., and Levine, S. Model-agnostic meta-learning for fast adaptation of deep networks. In *International conference on machine learning*, pp. 1126–1135. PMLR, 2017.

Genewein, T., Delétang, G., Ruoss, A., Wenliang, L. K., Catt, E., Dutordoir, V., Grau-Moya, J., Orseau, L., Hutter, M., and Veness, J. Memory-based meta-learning on non-stationary distributions. *International Conference on Machine Learning*, 2023.

Hahn, M. and Goyal, N. A theory of emergent in-context learning as implicit structure induction. *arXiv preprint arXiv:2303.07971*, 2023.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural Comput.*, 1997.

Hoffmann, J., Borgeaud, S., Mensch, A., Buchatskaya, E., Cai, T., Rutherford, E., Casas, D. d. L., Hendricks, L. A., Welbl, J., Clark, A., et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.

Hospedales, T., Antoniou, A., Micaelli, P., and Storkey, A. Meta-learning in neural networks: A survey. *IEEE transactions on pattern analysis and machine intelligence*, 44(9):5149–5169, 2021.

Hutter, M. *Universal artificial intelligence: Sequential decisions based on algorithmic probability*. Springer Science & Business Media, 2004.

Hutter, M. Human knowledge compression prize, 2006/2020. open ended, http://prize.hutter1.net/.

Hutter, M. On universal prediction and Bayesian confirmation. *Theoretical Computer Science*, 384(1):33–48, 2007. ISSN 0304-3975. doi: 10.1016/j.tcs.2007.05.016. URL http://arxiv.org/abs/0709.1516.

Hutter, M. Universal learning theory. In Sammut, C. and Webb, G. (eds.), *Encyclopedia of Machine Learning and Data Mining*, pp. 1295–1304. Springer, 2nd edition, 2017. ISBN 978-1-4899-7686-4. doi: 10.1007/978-1-4899-7687-1_867. URL http://arxiv.org/abs/1102.2467.

Hutter, M., Legg, S., and Vitányi, P. M. B. Algorithmic probability. *Scholarpedia*, 2(8):2572, 2007. ISSN 1941-6016. doi: 10.4249/scholarpedia.2572.

Ibarz, B., Kurin, V., Papamakarios, G., Nikiforou, K., Bennani, M., Csordás, R., Dudzik, A. J., Bošnjak, M., Vitvitskyi, A., Rubanova, Y., et al. A generalist neural algorithmic learner. In *Learning on graphs conference*, pp. 2–1. PMLR, 2022.

Joulin, A. and Mikolov, T. Inferring algorithmic patterns with stack-augmented recurrent nets. In *Advances in Neural Information Processing Systems 28*, 2015.

Kataoka, H., Okayasu, K., Matsumoto, A., Yamagata, E., Yamada, R., Inoue, N., Nakamura, A., and Satoh, Y. Pre-training without natural images. In *Proceedings of the Asian Conference on Computer Vision*, 2020.

Kenton, J. D. M.-W. C. and Toutanova, L. K. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of NAACL-HLT*, pp. 4171–4186, 2019.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Lattimore, T., Hutter, M., and Gavane, V. Universal prediction of selected bits. In *Algorithmic Learning Theory: 22nd International Conference, ALT 2011, Espoo, Finland, October 5-7, 2011. Proceedings 22*, pp. 262–276. Springer, 2011.

Lemley, J., Bazrafkan, S., and Corcoran, P. Smart augmentation learning an optimal data augmentation strategy. *Ieee Access*, 5:5858–5869, 2017.

Li, K., Hopkins, A. K., Bau, D., Viégas, F., Pfister, H., and Wattenberg, M. Emergent world representations: Exploring a sequence model trained on a synthetic task. In *The Eleventh International Conference on Learning Representations*, 2023a. URL https://openreview.net/forum?id=DeG07_TcZvT.

Li, M. and Vitanyi, P. M. Inductive reasoning and kolmogorov complexity. *Journal of Computer and System Sciences*, 44(2):343–384, 1992.

Li, M., Vitányi, P., et al. *An introduction to Kolmogorov complexity and its applications*. Springer, 4th edition, 2019.

Li, Y., Ildiz, M. E., Papailiopoulos, D., and Oymak, S. Transformers as algorithms: Generalization and implicit model selection in in-context learning. *arXiv preprint arXiv:2301.07067*, 2023b.

Liu, B., Ash, J. T., Goel, S., Krishnamurthy, A., and Zhang, C. Transformers learn shortcuts to automata. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=De4FYqjFueZ.

Mali, A., Ororbia, A., Kifer, D., and Giles, L. On the computational complexity and formal hierarchy of second order recurrent neural networks. *arXiv preprint arXiv:2309.14691*, 2023.

Mikulik, V., Delétang, G., McGrath, T., Genewein, T., Martic, M., Legg, S., and Ortega, P. Meta-trained agents implement bayes-optimal agents. *Advances in neural information processing systems*, 33:18691–18703, 2020.

Mingard, C., Rees, H., Valle-Pérez, G., and Louis, A. A. Do deep neural networks have an inbuilt occam's razor? *arXiv preprint arXiv:2304.06670*, 2023.

Müller, U. Brainf*ck. https://esolangs.org/wiki/Brainfuck, 1993. [Online; accessed 21-Sept-2023].

Ortega, P. A., Wang, J. X., Rowland, M., Genewein, T., Kurth-Nelson, Z., Pascanu, R., Heess, N., Veness, J., Pritzel, A., Sprechmann, P., et al. Meta-learning of sequential strategies. *arXiv preprint arXiv:1905.03030*, 2019.

Perez, L. and Wang, J. The effectiveness of data augmentation in image classification using deep learning. *arXiv preprint arXiv:1712.04621*, 2017.

Rathmanner, S. and Hutter, M. A philosophical treatise of universal induction. *Entropy*, 13(6):1076–1136, 2011.

Ruoss, A., Delétang, G., Genewein, T., Grau-Moya, J., Csordás, R., Bennani, M., Legg, S., and Veness, J. Randomized positional encodings boost length generalization of transformers. In Rogers, A., Boyd-Graber, J., and Okazaki, N. (eds.), *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 1889–1903, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-short.161. URL https://aclanthology.org/2023.acl-short.161.

Santoro, A., Bartunov, S., Botvinick, M., Wierstra, D., and Lillicrap, T. Meta-learning with memory-augmented neural networks. In *International conference on machine learning*, pp. 1842–1850. PMLR, 2016.

Schmidhuber, J. The speed prior: A new simplicity measure yielding near-optimal computable predictions. In *Proc. 15th Conf. on Computational Learning Theory (COLT'02)*, volume 2375 of *LNAI*, pp. 216–228, Sydney, Australia, 2002. Springer.

Sipser, M. *Introduction to the Theory of Computation*. Course Technology Cengage Learning, Boston, MA, 3rd ed edition, 2012. ISBN 978-1-133-18779-0.

Solomonoff, R. J. A formal theory of inductive inference. part i. *Information and control*, 7(1):1–22, 1964a.

Solomonoff, R. J. A formal theory of inductive inference. part ii. *Information and control*, 7(2):224–254, 1964b.

Sterkenburg, T. F. A generalized characterization of algorithmic probability. *Theory of Computing Systems*, 61: 1337–1352, 2017.

Stogin, J., Mali, A., and Giles, C. L. A provably stable neural network turing machine. *arXiv preprint arXiv:2006.03651*, 2020.

Sunehag, P. and Hutter, M. Principles of solomonoff induction and aixi. In *Algorithmic Probability and Friends. Bayesian Prediction and Artificial Intelligence: Papers from the Ray Solomonoff 85th Memorial Conference, Melbourne, VIC, Australia, November 30–December 2, 2011*, pp. 386–398. Springer, 2013.

Sunehag, P. and Hutter, M. Intelligence as inference or forcing Occam on the world. In *Proc. 7th Conf. on Artificial General Intelligence (AGI'14)*, volume 8598 of *LNAI*, pp. 186–195, Quebec City, Canada, 2014. Springer. ISBN 978-3-319-09273-7. doi: 10.1007/978-3-319-09274-4_18.

Suzgun, M., Gehrmann, S., Belinkov, Y., and Shieber, S. M. Memory-augmented recurrent neural networks can learn generalized dyck languages. *CoRR*, 2019.

Valle-Perez, G., Camargo, C. Q., and Louis, A. A. Deep learning generalizes because the parameter-function map is biased towards simple functions. *arXiv preprint arXiv:1805.08522*, 2018.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. In *Advances in Neural Information Processing Systems 30*, 2017.

Veličković, P., Badia, A. P., Budden, D., Pascanu, R., Banino, A., Dashevskiy, M., Hadsell, R., and Blundell, C. The CLRS algorithmic reasoning benchmark. In Chaudhuri, K., Jegelka, S., Song, L., Szepesvari, C., Niu, G., and Sabato, S. (eds.), *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pp. 22084–22102. PMLR, 17–23 Jul 2022. URL https://proceedings.mlr.press/v162/velickovic22a.html.

Veness, J., Sunehag, P., and Hutter, M. On ensemble techniques for aixi approximation. In *International Conference on Artificial General Intelligence*, pp. 341–351. Springer, 2012.

Wang, X., Zhu, W., and Wang, W. Y. Large language models are implicitly topic models: Explaining and finding good demonstrations for in-context learning. *arXiv preprint arXiv:2301.11916*, 2023.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q. V., Zhou, D., et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35: 24824–24837, 2022.

Willems, F., Shtarkov, Y., and Tjalkens, T. Reflections on "the context tree weighting method: Basic properties". *Newsletter of the IEEE Information Theory Society*, 47 (1), 1997.

Willems, F. M. The context-tree weighting method: Extensions. *IEEE Transactions on Information Theory*, 44(2): 792–798, 1998.

Willems, F. M., Shtarkov, Y. M., and Tjalkens, T. J. The context-tree weighting method: Basic properties. *IEEE transactions on information theory*, 41(3):653–664, 1995.

Wood, I., Sunehag, P., and Hutter, M. (Non-)equivalence of universal priors. In *Proc. Solomonoff 85th Memorial Conference*, volume 7070 of *LNAI*, pp. 417–425, Melbourne, Australia, 2011. Springer. ISBN 978-3-642-44957-4. doi: 10.1007/978-3-642-44958-1_33. URL http://arxiv.org/abs/1111.3854.

Wood, I., Sunehag, P., and Hutter, M. (non-) equivalence of universal priors. In *Algorithmic Probability and Friends. Bayesian Prediction and Artificial Intelligence: Papers from the Ray Solomonoff 85th Memorial Conference, Melbourne, VIC, Australia, November 30–December 2, 2011*, pp. 417–425. Springer, 2013.

Xie, S. M., Raghunathan, A., Liang, P., and Ma, T. An explanation of in-context learning as implicit bayesian inference. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=RdJVFCHjUMI.

Zhou, J., Cui, G., Hu, S., Zhang, Z., Yang, C., Liu, Z., Wang, L., Li, C., and Sun, M. Graph neural networks: A review of methods and applications. *AI open*, 1:57–81, 2020.

Zhou, Y., Alon, U., Chen, X., Wang, X., Agarwal, R., and Zhou, D. Transformers can achieve length generalization but not robustly. *arXiv preprint arXiv:2402.09371*, 2024.

A. Solomonoff Samples

Sampling from semimeasures. We can sample strings from a semimeasure μ as follows: Start with the empty string $x = \epsilon$.
With probability $\mu(a|x) := \mu(xa)/\mu(x)$ extend $x \leftarrow xa$ for $a \in \mathcal{X}$. Repeat.
With probability $1 - \sum_{a\in\mathcal{X}} \mu(a|x)$ return x.

Let $D := (x^1, ..., x^J)$ be J (in)finite sequences sampled from μ. If we only have these samples, we can estimate μ as follows:

$$\hat{\mu}_D(x) \;:=\; \frac{1}{|D|}\sum_{y\in D}[\![\ell(y) \geq \ell(x) \;\wedge\; y_{1:\ell(x)} = x]\!] \quad \stackrel{w.p.1}{\longrightarrow} \quad \mu(x) \quad \text{for} \quad |D| \to \infty \tag{2}$$

Proof: Let $D_x := (y \in D : \ell(y) \geq \ell(x) \;\wedge\; y_{1:\ell(y)} = x)$ be the elements in D that start with x. Since x^j are sampled i.i.d. from μ, the law of large numbers implies $|D_x|/|D| \to \mu(x)$ for $J \to \infty$. □

Limit normalization. A simple way of normalization is

$$\widetilde{M}_{s,L,n}(x_{1:t}) \;:=\; \frac{\sum_{x_{t+1:n}} M_{s,L,n}(x_{1:n})}{\sum_{x_{1:n}} M_{s,L,n}(x_{1:n})} \quad \text{for} \quad t \leq n \quad \text{and} \quad 0 \quad \text{else}$$

This is a proper measure for sequences up to length n. Sampling from it is equivalent to sampling from $M_{s,L,n}$ but discarding all sequences shorter than n. Let $\widetilde{D} := (x^j \in D^J : \ell(x^j) \geq n)$. Then

$$\hat{\widetilde{M}}_{\widetilde{D}}(x) \;=\; \frac{1}{|\widetilde{D}|}\sum_{y\in\widetilde{D}}[\![y_{1:\ell(x)} = x]\!] \quad \longrightarrow \quad M(x) \quad \text{for} \quad s, L, n, J \to \infty$$

Proof: First, $|\widetilde{D}|/|D|$ is the relative fraction of sequences that have length n, and $\sum_{x_{1:n}} M_{s,L,n}(x_{1:n})$ is the probability that a sequence has length n, hence the former converges to the latter for $J \to \infty$. Second,

$$\begin{aligned}
\hat{\widetilde{M}}_{\widetilde{D}}(x_{1:n}) \;&=\; \frac{1}{|\widetilde{D}|}\sum_{y\in\widetilde{D}}[\![y_{1:\ell(x)} = x_{1:n}]\!] \;=\; \frac{|D|}{|\widetilde{D}|}\frac{1}{|D|}\sum_{y\in D}[\![\ell(y) \geq n \;\wedge\; y_{1:\ell(x)} = x_{1:n}]\!] \\
&=\; \frac{|D|}{|\widetilde{D}|}\hat{M}_{D^J}(x_{1:n}) \quad \stackrel{J\to\infty}{\longrightarrow} \quad \frac{M_{s,L,n}(x_{1:n})}{\sum_{x_{1:n}} M_{s,L,n}(x_{1:n})} \;=\; \widetilde{M}_{s,L,n}(x_{1:n})
\end{aligned}$$

Third, take the sum $\sum_{x_{t+1:n}}$ on both sides, and finally the limit $s, L, n \to \infty$ and set $x = x_{1:t}$. □

A disadvantage of this normalization scheme is that the probability of a sequence x depends on n even if $\ell(x) < n$, while $M_{s,L,n}(x)$ and $M^{norm}_{...}(x)$ below are essentially independent of n.

Proposition 4. *Let $D := (x^1, ..., x^J)$ be J (in)finite sequences sampled from a semimeasure μ (e.g. M). We can estimate μ as follows. Given*

$$\hat{\mu}_D(x) := \frac{1}{|D|}\sum_{y\in D}[\![\ell(y) \geq \ell(x) \;\wedge\; y_{1:\ell(x)} = x]\!],$$

then $\hat{\mu}_D(x) \stackrel{w.p.1}{\longrightarrow} \mu(x)$ for $|D| \to \infty$.

Proof: Let $D_x := (y \in D : \ell(y) \geq \ell(x) \;\wedge\; y_{1:\ell(y)} = x)$ be the elements in D that start with x. Since x^j are sampled i.i.d. from μ, the law of large numbers implies $|D_x|/|D| \to \mu(x)$ for $J \to \infty$. □

Proposition 6. *Let now $D^J := (x^1, ..., x^J)$ be samples from the measure $M_{s,L,n}$. Given*

$$\hat{M}_{D^J}(x) = \frac{1}{J}\sum_{y\in D^J}[\![\ell(y) \geq \ell(x) \;\wedge\; y_{1:\ell(x)} = x]\!]$$

then, $\hat{M}_{D^J}(x) \quad \longrightarrow \quad M_{s,L,n}(x)$ for $J \to \infty$.

Proof: It follows directly from Proposition 4.

Proposition 8. *Using the definitions from Proposition 6 we have that*

$$\hat{M}^{norm}_{s,L,n}(x_t|x_{<t}) \;=\; \frac{\sum_{y\in D^J}[\![\ell(y)\geq t \;\wedge\; y_{1:t}=x_{1:t}]\!]}{\sum_{y\in D^J}[\![\ell(y)\geq t \;\wedge\; y_{<t}=x_{<t}]\!]} \stackrel{J\to\infty}{\longrightarrow} \; M^{norm}_{s,L,n}(x_t|x_{<t})$$

Then, we can take the product over $t=1,...,n$ *to obtain* $\hat{M}^{norm}_{s,L,n}(x)\to M^{norm}_{s,L,n}(x)\to M^{norm}(x)$.

Proof: For $x=x_{<t}$ and $a=x_t$, we have

$$\begin{aligned}\sum_{a\in\mathcal{X}}\hat{M}_{D^J}(xa) \;&=\; \frac{1}{J}\sum_a\sum_{y\in D^J}[\![\ell(y)\geq\ell(xa)\;\wedge\;y_{1:\ell(xa)}=xa]\!]\\ &=\; \frac{1}{J}\sum_{y\in D^J}[\![\ell(y)\geq t\;\wedge\;\exists a: y_{1:t}=xa]\!]\end{aligned}$$

$$\text{hence}\quad \hat{M}^{norm}_{s,L,n}(a|x) \;=\; \frac{\hat{M}_{D^J}(xa)}{\sum_a\hat{M}_{D^J}(xa)} \stackrel{J\to\infty}{\longrightarrow} \frac{M_{s,L,n}(ax)}{\sum_a M_{s,L,n}(ax)} \;=\; M^{norm}_{s,L,n}(a|x) \tag{3}$$

□

B. Training with Transformers

Using Transformers for estimating M. Most Transformer implementations require sequences of fixed length (say) n. We can mimic shorter sequences by introducing a special absorbing symbol $\perp\notin\mathcal{X}$, and pad all sequences x^j shorter than n with $\perp$s. We train the Transformer on these (padded) sequences with the log-loss. Under the (unrealistic) assumptions that the Transformer has the capacity to represent $\hat{M}_{...}$, and the learning algorithm can find the representation, this (tautologically) implies that the Transformer distribution converges to $\hat{M}_{...}$. Similarly if the Transformer is trained on x^j sampled from $M_{s(j),L(j),n}(x)$ for $j=1,2,3,...$, it converges to $M_{\infty,\infty,n}$. For a Transformer with context length n increasing over time, even $\hat{M}_{...}\to M$ could be possible. To guarantee normalized probabilities when learning $\widetilde{M}_{...}$ and $M^{norm}_{...}$, we do *not* introduce a $\perp$-padding symbol. For $\widetilde{M}_{...}$ we train on $\widetilde{D}$ which doesn't require padding. For training towards $M^{norm}_{...}$, we pad the x^j in D^J to length n with arbitrary symbols from $\mathcal{X}$ and train on that, but we (have to) cut the log-loss short $-\sum_{t=1}^{\ell(x)}\log(\text{LLM}(x_t|x_{<t}))$, i.e. $\ell(x)$ rather than n, so as to make the loss hence gradient hence minimum independent of the arbitrary padding.

Limit-normalized $\widetilde{M}$. This is the easiest case: $\widetilde{D}$ removes strings shorter than n from D^J (sampled from M), so $\widetilde{D}$ has distribution $\widetilde{M}$, hence for $D=\widetilde{D}$, the log-loss is minimized by $p_\theta=\hat{\widetilde{M}}$, i.e. training on $\widetilde{D}$ makes p_θ converge to $\hat{\widetilde{M}}$ (under the stated assumptions).

Unnormalized M. For this case we need to augment the (token) alphabet $\mathcal{X}$ with some (absorbing) padding symbol $\perp$: Let $D_\perp$ be all $x\in D^J$ but padded with some $\perp$ to length n. We can extend $M:\mathcal{X}^*\to[0;1]$ to $M_\perp:\mathcal{X}^*\cup\{\perp\}\to[0;1]$ by

$$\begin{aligned}M_\perp(x) &:= M(x) && \text{for all} && x\in\mathcal{X}^*\\ M_\perp(x\perp^t) &:= M(x)-\textstyle\sum_{a\in\mathcal{X}}M(xa) && \text{for all} && x\in\mathcal{X}^* \text{ and } t\geq 1\\ M_\perp(x) &:= 0 && \text{for all} && x\notin\mathcal{X}^*\{\perp\}^*\end{aligned}$$

It is easy to see that $D_\perp$ has distribution $M_\perp$, hence for $D=D_\perp$, the log-loss is minimized by $p_\theta=\hat{M}_\perp$. Since $\hat{M}_\perp(x)$ restricted to $x\in\mathcal{X}^*$ is just $\hat{M}(x)$, training on $D_\perp$ makes $p_\theta(x)$ converge to $\hat{M}(x)$ for $x\in\mathcal{X}^*$. Though it is possible to train neural models that would converge in the limit to the standard (computable) Solomonoff prior, we focus on the normalized version due to Remark 7.

Training variation: Note that for M, the Transformer is trained to predict $x\perp$ if $\ell(x)<n$. If $\ell(x)<n$ is due to the time limit s in U^s, it is preferable to *not* train the Transformer to predict $\perp$ after x, since for $s\to\infty$, which we are ultimately interested in, x may be extended with proper symbols from $\mathcal{X}$. One way to achieve this is to cut the log-loss (only) in this case at $t=\ell(x)$ similar to M^{norm} below to not reward the Transformer for predicting $\perp$.

B.1. Normalized Solomonoff Loss

Here is the derivation of the loss.

$$\begin{aligned}\text{Loss}(\theta) &:= -\frac{1}{J}\sum_{x\in D^J}\log p_\theta(x) = -\frac{1}{J}\sum_{x\in D^J}\sum_{t=1}^{\ell(x)}\log p_\theta(x_t|x_{<t}) \\ &= -\frac{1}{J}\sum_{t=1}^{n}\sum_{x\in D^J\wedge\ell(x)\geq t}\log p_\theta(x_t|x_{<t}) = -\sum_{t=1}^{n}\sum_{x_{1:t}}\hat{M}_{D^J}(x_{1:t})\log p_\theta(x_t|x_{<t}) \\ &= -\sum_{t=1}^{n}\sum_{x_{<t}}\Big(\sum_{x_t}\hat{M}_{D^J}(x_{1:t})\Big)\Big(\sum_{x_t}\hat{M}^{norm}(x_t|x_{<t})\log p_\theta(x_t|x_{<t})\Big)\end{aligned}$$

where the last equality follows from (3).

C. Generalized Solomonoff Semimeasure

Streaming functions. A streaming function φ takes a growing input sequence and produces a growing output sequence. In general, input and output may grow unboundedly or stay finite. Formally, $\varphi:\mathcal{X}^\#\to\mathcal{X}^\#$, where $\mathcal{X}^\#:=\mathcal{X}^\infty\cup\mathcal{X}^*$. In principle input and output alphabet could be different, but for simplicity we assume that all sequences are binary, i.e. $\mathcal{X}=\{0,1\}$. For φ to qualify as a streaming function, we need to ensure that extending the input only extends and does not modify the output. Formally, we say that

$$\varphi \text{ is monotone} \quad \text{iff} \quad [\forall q\sqsubseteq p:\varphi(q)\sqsubseteq\varphi(p)]$$

where $q\sqsubseteq p$ means that q is a prefix of p i.e. $\exists r\in\mathcal{X}^\#: qr=p$, and $\sqsubset$ denotes strict prefix $r\neq\epsilon$. p is φ-minimal for x if $\exists r:\phi(p)=xr$ and $\forall r\forall q\sqsubset p:\phi(q)\neq xr$. We will denote this by $\varphi(p)=x*$. p is the shortest program outputting a string starting with x.

Monotone Turing Machines (MTM). A Monotone Turing machine T is a Turing machine with left-to-right read-only input tape, left-to-right write-only output tape, and some bidirectional work tape. The function φ_T it computes is defined as follows: At any point in time after writing the output symbol but before moving the output head and after moving the input head but before reading the new cell content, if p is the content left of the current input tape head, and x is the content of the output tape up to the current output tape head, then $\varphi_T(p):=x$. It is easy to see that φ_T is monotone. We abbreviate $T(p)=\varphi_T(p)$. There exist (so called optimal) universal MTM U that can emulate any other MTM via $U(i'q)=T_i(q)$, where $T_1,T_2,...$ is an effective enumeration of all MTMs and i' a prefix encoding of i (Hutter, 2004; Li et al., 2019).

C.1. Proof of Theorem 9

Theorem 9 (Universality of generalized Solomonoff semimeasures)**.** *$M_U^Q(x)$ is strongly universal, provided Q is a computable measure and $Q(q)>0\;\forall q\in\mathcal{X}^*$ and $Q(q_{1:n})\to 0$ for $n\to\infty$. More precisely, for all universal monotone TM U and all Q with the above properties, there exists a universal MTM V (as constructed in the proof) s.th. $M_U^Q(x)=M_V(x)\;\forall x$. Proof in Appendix C.*

We can effectively sample from any computable Q if we have access to infinitely many fair coin flips. The conditions on Q ensure that the entropy of Q is infinite, and stays infinite even when conditioned on any $q\in\mathcal{X}^*$. This also allows the reverse: Converting a sample from Q into infinitely many uniform random bits. Forward and backward conversion can be achieved sample-efficiently via (bijective) arithmetic (de)coding. This forms the basis of the proof below. The condition of Q being a proper measure rather than just being a semimeasure is also necessary: For instance, for $Q(q)=4^{-\ell(q)}$, on a Bernoulli($\frac{1}{2}$) sequence $x_{1:\infty}$, $M_U(x_t|x_{<t})\to\frac{1}{2}$ as it should, one can show that $M_U^Q(x_t|x_{<t})<\frac{1}{3}$ for infinitely many t (w.p.1).

Proof. (sketch) Let $0.q_{1:\infty}\in[0;1]$ be the real number with binary expansion $q_{1:\infty}$. With this identification, Q can be regarded as a probability measure over $[0;1]$. Let $F:[0;1]\to[0;1]$ be its cumulative distribution function, which can explicitly be represented as $F(0.q_{1:\infty})=\sum_{t:q_t=1}Q(\Gamma_{q_{<t}0})$, since $[0;0.q_{1:\infty})=\biguplus_{t:q_t=1}0.\Gamma_{q_{<t}0}$, where $0.\Gamma_q=[0.q0^\infty;0.q1^\infty)$ and $\biguplus$ denotes disjoint union. Now assumption $Q(q)>0\;\forall q\in\mathcal{X}^*$ implies that F is strictly increasing,

and assumption $Q(q_{1:n}) \to 0$ implies that F is continuous. Since $F(0) = 0$ and $F(1) = 1$, this implies that F is a bijection. Let $0.p_{1:\infty} = F(0.q_{1:\infty})$ and $0.q_{1:\infty} = F^{-1}(0.p_{1:\infty})$. [4]. Further for some finite prefix $q \sqsubset q_{1:\infty}$, we partition the interval

$$[0.p^0_{1:\infty};\, 0.p^1_{1:\infty}) \;:=\; [F(0.q0^\infty); F(0.q1^\infty)) \;=:\; \dot{\bigcup}_{p\in\Phi(q)} 0.\Gamma_p$$

into a minimal set of binary intervals $0.\Gamma_p$, where $\Phi(q)$ is a minimal prefix free set in the sense that for any p, at most one of p, $p0$, $p1$ is in $\Phi(q)$. An explicit representation is

$$\Phi(q) \;:=\; \{p^0_{<t}1 : t > t_0 \wedge p^0_t = 0\} \,\dot{\cup}\, \{p^1_{<t}0 : t > t_0 \wedge p^1_t = 1\}$$

where t_0 is the first t for which $p^0_t \neq p^1_t$. Now we plug

$$Q(q) \;=\; F(0.q1^\infty) - F(0.q0^\infty) \;=\; \sum_{p\in\Phi(q)} |0.\Gamma_p| \;=\; \sum_{p\in\Phi(q)} 2^{-\ell(p)} \quad \text{into}$$

$$M^Q_U(x) \;\equiv\; \sum_{q:U(q)=x*} Q(q) \;=\; \sum_{q:U(q)=x*}\; \sum_{p\in\Phi(q)} 2^{-\ell(p)} \;=\; \sum_{p:V(p)=x*} 2^{-\ell(p)} \;=\; M_V(x)$$

where $V(p) := U(q)$ for the maximal q such that $p \in \Phi(q)$. The maximal q is unique, since $\Phi(q) \cap \Phi(q') = \{\}$ if $q \not\sqsubseteq q'$ and $q' \not\sqsubseteq q$, and finite since F is continuous.

It remains to show that V is universal. Let $p^\imath$ be such that $0.\Gamma_{p^\imath} \subseteq [F(0.\imath'0^\infty); F(0.\imath'1^\infty))$. The choice doesn't matter as long as it is a computable function of $\imath$, but shorter is "better". This choice ensures that $F^{-1}(0.p^\imath*) = 0.\imath'...$ whatever the continuation $*$ is. Now let $F(q_{1:\infty})_{\text{tail}} := F(q_{1:\infty})_{\ell(p^\imath)+1:\infty} = p_{\ell(p^\imath)+1:\infty}$ if $q_{1:\infty}$ starts with $\imath'$, and arbitrary, e.g. $F(q_{1:\infty})$, otherwise. Let T be a MTM with $T(q_{1:\infty}) := U_0(F(q_{1:\infty})_{\text{tail}})$ for some universal MTM U_0. By Kleene's 2nd recursion theorem (Sipser, 2012, Chp.6), there exists an i such that $T_i(q) = T(i'q)\ \forall q$. Let $\dot{k} := \ell(i') + 1$ and $\dot{\ell} := \ell(p^i) + 1$ and $q_{<\dot{k}} := i'$, hence $p_{<\dot{\ell}} = p^i$. Now $V(p_{1:\infty}) = U(q_{1:\infty})$ implies

$$V(p^i p_{\dot{\ell}:\infty}) \;=\; U(i'q_{\dot{k}:\infty}) \;=\; T_i(q_{\dot{k}:\infty}) \;=\; T(i'q_{\dot{k}:\infty}) \;=\; U_0(F(i'q_{\dot{k}:\infty})_{\text{tail}}) \;=\; U_0(p_{\dot{\ell}:\infty})$$

hence V is universal, which concludes the proof. □

Practical universal streaming functions. Turing machines are impractical and writing a program for a universal streaming function is another layer of indirection which is best to avoid. Programming languages are already universal machines. We can define a conversion of real programs from/to binary strings and prepend it to the input stream. When sampling input streams $q_{1:\infty}$ we convert the beginning into a program of the desired programming language, and feed it the tail as input stream.

D. Experiment methodology details

D.1. Architecture details

RNN. A vanilla multi-layer RNN (Elman, 1990) with hidden sizes and multi-layer perceptron (MLP) before and after the RNN layers as described in Table 1.

Stack-RNN. A multi-layer RNN controller with hidden sizes and MLP exactly the same as the RNN and LSTMs on Table 1 with access to a differentiable stack (Joulin & Mikolov, 2015). The controller can perform any linear combination of PUSH, POP, and NO-OP on the stack of size according to Table 1, with action weights given by a softmax over a linear readout of the RNN output. Each cell of the stack contains a real vector of dimension 6 and the stack size is 64 for all (S, M and L) sizes.

[4]Note that $p_{1:m}$ is uniformly distributed and is (for some m) essentially the arithmetic encoding of $q_{1:n}$ with one caveat: The mapping from sequences to reals conflates $0.q10^\infty = 0.q01^\infty$. Since the set of all conflated sequences has probability 0, (under Q as well as Bernoulli($\frac{1}{2}$)), any error introduced due to this conflation has no effect on the distribution $M^Q_U(x)$.

Table 1: Architectures

RNN and LSTMs	S	M	L
RNN Hidden size	16	32	128
Number of RNN layers	1	2	3
MLP before RNN layers	(16,)	(32, 32)	(128, 128, 128)
MLP after RNN layers	(16,)	(32, 32)	(128, 128, 128)
Transformer SINCOS			
Embedding dimension	16	64	256
Number of heads	2	4	4
Number of layers	2	4	6

Tape-RNN. A multi-layer RNN controller with hidden sizes according to the Table 1 with access to a differentiable tape, inspired by the Baby-NTM architecture (Suzgun et al., 2019). The controller can perform any linear combination of WRITE-RIGHT, WRITE-LEFT, WRITE-STAY, JUMP-LEFT, and JUMP-RIGHT on the tape, with action weights given by a softmax. The actions correspond to: writing at the current position and moving to the right (WRITE-RIGHT), writing at the current position and moving to the left (WRITE-LEFT), writing at the current position (WRITE-STAY), jumping ℓ steps to the right without writing (JUMP-RIGHT), where ℓ is the length of the input, and jumping ℓ steps to the left without writing (JUMP-LEFT). As in the Stack-RNN, each cell of the tape contains a real vector of dimension 6 and the tape size is 64 for all (S, M and L) sizes.

LSTM. A multi-layer LSTM (Hochreiter & Schmidhuber, 1997) of hidden sizes according to Table 1.

Transformer decoder. A vanilla Transformer decoder (Vaswani et al., 2017). See Table 1 for the embedding dimension, number of heads and number of layers for each model size (S, M and L). Each layer is composed of an attention layer, two dense layers, and a layer normalization. We add a residual connections as in the original architecture (Vaswani et al., 2017). We consider the standard sin/cos (Vaswani et al., 2017) positional encoding.

Model sizes Model sizes (see 2) were determined by pilot experiments on VOMS data using the LSTM and Transformer architecture, where we increased the architecture size until models could fit the VOMS data well i.e. minimal log-loss on a 500k budget iterations. This formed the L-variants of the models and the M, and S variants used reduced numbers of layers and hidden sizes. All other recurrent networks use the same hidden sizes and layers as the LSTM for better comparison. We used this procedure to assess performance within an architecture class since the number of parameters is not a good metric across architecture classes.

D.2. CTW

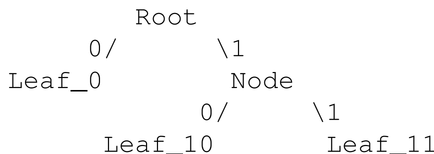
Below is an ultra-compact introduction to (sampling from) CTW (Willems et al., 1995; 1997). For more explanations, details, discussion, and derivations, see (Catt et al., 2024, Chp.4).

A variable-order Markov process. is a probability distribution over (binary) sequences $x_1, x_2, x_3, ...$ with the following property: Let $S \subset \{0,1\}^*$ be a complete suffix-free set of strings (a reversed prefix-free code) which can equivalently be viewed as a perfect binary tree. Then $p(x_t = 0|x_{<t}; S, \Theta_S) := \theta_s$ if (the unique) context of x_t is $s = x_{t-\ell(s):t-1} \in S$, and $\Theta_S := (\theta_s \in [0;1] : s \in S)$. We arbitrarily define $x_t = 0$ for $t \leq 0$.

Architecture	Hidden Size	Num Layers	Num Parameters
RNN-S	16	3	898
RNN-M	32	6	7,554
RNN-L	128	9	182,274
LSTM-S	16	3	2,466
LSTM-M	32	6	19,970
LSTM-L	128	9	477,954
Stack-RNN-S	16	3	1,147
Stack-RNN-M	32	6	8,532
Stack-RNN-L	128	9	188,061
Tape-RNN-S	16	3	6,717
Tape-RNN-M	32	6	21,592
Tape-RNN-L	128	9	224,931
Transformer-S	16	2	6,498
Transformer-M	64	4	199,170
Transformer-L	256	6	4,733,442

Table 2: Architectures, width, depth and number of parameters.

Intuition about Variable-order Markov sources VOMS considers data generated from tree structures. For example, given the binary tree



and given the history of data "011" (where 0 is the first observed datum and 1 is the last one) the next sample uses Leaf$_{11}$ (because the last two data points in history were 11) to draw the next datum using a sample from a Beta distribution with parameter Leaf$_{11}$. Say we sample a 0, thus history is then transformed into "0110" and Leaf$_{10}$ will be used to sample the next datum (because now the last two datapoints that conform to a leaf are "10"), and so forth. This way of generating data is very general and can produce many interesting patterns ranging from simple regular patterns like 01010101 or more complex ones that can have stochastic samples in it. Larger trees can encode very complex patterns indeed.

Sampling from CTW. Context Tree Weighting (CTW) is a Bayesian mixture over all variable-order Markov sources of maximal order $D \in \mathbb{N}_0$, i.e. over all trees S of maximal depth D and all $\theta_s \in [0;1]$ for all $s \in S$. The CTW distribution is obtained as follows: We start with an empty (unfrozen) $S = \{\epsilon\}$. Recursively, for each unfrozen $s \in S$ with $\ell(s) < D$, with probability $1/2$ we freeze s; with probability $1/2$ we split $S \leftarrow S \setminus \{s\} \cup \{0s, 1s\}$ until all $s \in S$ are frozen or $\ell(s) = D$. Then we sample θ_s from Beta($1/2, 1/2$) for all $s \in S$. Finally for $t = 1, 2, 3, ...$ we sample x_t from $p(x_t|x_{<t}; S, \Theta_S)$.

Computing CTW. The CTW probability $P_{\text{CTW}}(x_{1:n})$ can be calculated as follows: Let $a_s := |\{t \in \{1, ..., n\} : x_t = 0 \wedge x_{t-\ell(s):t-1} = s\}|$ count the number of $x_t = 0$ immediately preceded by context $s \in \{0,1\}^*$, and similarly $b_s := |\{t : x_t = 1 \wedge x_{t-\ell(s):t-1} = s\}|$. Let $x^s_{1:n} \in \{0,1\}^{a_s+b_s}$ be the subsequence of x_t's that have context s. For given θ_s for $s \in S$, $x^s_{1:n}$ is i.i.d. (Bernoulli($1-\theta_s$)). Hence for $\theta_s \sim$ Beta($1/2, 1/2$), $P(x^s_{1:n}|s \in S) = P_{\text{KT}}(a_s, b_s) := \int_0^1 \theta_s^{a_s}(1-\theta_s)^{b_s}\text{Beta}(1/2, 1/2)(\theta_s)d\theta_s$. If $s \notin S$, we split $x^s_{1:n}$ into $x^{0s}_{1:n}$ and $x^{1s}_{1:n}$. By construction of S, a tentative $s \in S$ gets replaced by $0s$ and $1s$ with 50% probability, recursively, hence $P_{\text{CTW}}(x^s_{1:n}) = \frac{1}{2}P_{\text{KT}}(a_s, b_s) + \frac{1}{2}P_{\text{CTW}}(x^{0s}_{1:n})P_{\text{CTW}}(x^{1s}_{1:n})$ terminating with $P_{\text{CTW}}(x^s_{1:n}) = P_{\text{KT}}(a_s, b_s)$ when $\ell(s) = D$. This completes the definition of $P_{\text{CTW}}(x_{1:n}) \equiv P_{\text{CTW}}(x^\epsilon_{1:n})$. Efficient $O(nD)$ algorithms for computing $P_{\text{CTW}}(x_{1:n})$ (and updating $n \to n+1$ in time $O(D)$) and non-recursive definitions can be found in Catt et al. (2024, Chp.4).

Distributions of Trees. A tree has depth $\leq d$ if either it is the empty tree or if both its subtrees have depth $< d$. Therefore the probability of sampling a tree of depth $\leq d$ is $F(d) = \frac{1}{2} + \frac{1}{2}F(d-1)^2$, with $F(0) = \frac{1}{2}$. Therefore the probability

of sampling a tree of depth d is $P(d) = F(d) - F(d-1)$ for $d < D$ and $P(D) = 1 - F(D-1)$. The theoretical curve ($P(0) = \frac{1}{2}$, $P(1) = \frac{1}{8}$, $P(2) = \frac{9}{128}$,...) is plotted in Fig. 7a together with the empirical distribution. More meaningful is probably the expected number of leaf nodes at each level d. Since each node at level d is replaced with prob. $\frac{1}{2}$ by two nodes at level $d+1$, the expected number of leaf nodes $E(d)$ is the same at all levels $d < D$. Since $E(0) = \frac{1}{2}$, we have $E(d) = \frac{1}{2}$ for all $d < D$ and $E(D) = 1$, hence the total expected number of leaf nodes is $E_+ = \frac{1}{2}D + 1$. While this doesn't sound much, it ensures that for $N = 10'000$ samples, we uniformly test $5'000$ contexts for each length $d < D$. We can get some control over the distribution of trees by splitting nodes at level d with probability $\alpha_d \in [0;1]$ instead of $\frac{1}{2}$. In this case, $E(d) = 2\alpha_0 \cdot ... \cdot 2\alpha_{d-1}(1 - \alpha_d)$ for $d < D$. So for $\alpha_d > \frac{1}{2}$ we can create trees of size exponential in D, and (within limits) any desired depth distribution.

D.3. Chomsky

Table 3: Table taken from (Deletang et al., 2022). Tasks with their level in the Chomsky hierarchy and example input/output pairs. The † denotes permutation-invariant tasks; the ⋆ denotes counting tasks; the ∘ denotes tasks that require a nondeterministic controller; and the × denotes tasks that require superlinear running time in terms of the input length.

Level	Name	Example Input	Example Output
Regular (R)	`Even Pairs`	*aabba*	True
	`Modular Arithmetic (Simple)`	$1+2-4$	4
	`Parity Check`†	*aaabba*	True
	`Cycle Navigation`†	011210	2
Deterministic context-free (DCF)	`Stack Manipulation`	*abbaa* POP PUSH *a* POP	*abba*
	`Reverse String`	*aabba*	*abbaa*
	`Modular Arithmetic`	$-(1-2)\cdot(4-3\cdot(-2))$	0
	`Solve Equation`∘	$-(x-2)\cdot(4-3\cdot(-2))$	1
Context-sensitive (CS)	`Duplicate String`	*abaab*	*abaababaab*
	`Missing Duplicate`	10011021	0
	`Odds First`	*aaabaa*	*aaaaba*
	`Binary Addition`	$10010 + 101$	10111
	`Binary Multiplication`×	$10010 * 101$	1001000
	`Compute Sqrt`	100010	110
	`Bucket Sort`†★	421302214	011222344

E. UTMs: Brainf*ck and BrainPhoque

Our BrainPhoque (BP) UTM produces program evaluation traces that are equivalent to those of brainf*ck (BF) programs (Müller, 1993) (see also $\mathcal{P}''$ (Böhm, 1964)), but the programs are written slightly differently: they are even less human-readable but have better properties when sampling programs.

We start by giving a quick overview of the BF machine, then explain why we need a slightly different machine, and its construction. Finally we explain how to shorten sampled programs and calculate an upper bound on the log-loss.

See Figure 6 for some sample programs and outputs.

E.1. Brainf*ck

BF is one of the smallest and simplest Turing-complete programming languages. It features a read-only input tape, a working tape, and a write-only output tape. These tapes are assumed infinite but for practical purposes they are usually fixed at a finite and constant length and initialized with 0.[5] Each tape cell can contain a non-negative integer, which can grow as large as the 'alphabet size'. Above that number, it loops back to 0. In the paper, we choose an alphabet size of 17.

[5] The tape could also grow on request, but this tends to slow down program evaluation.

Each tape has a pointer. For simplicity, the pointer of the working tape is called WTP, and the value at the WTP is called *datum*, which is an integer.

BF uses 8 instructions `<>+-[],.` which are:

- `<` and `>` decrement and increment the WTP, modulo the length of the tape.
- `+` and `−` increment and decrement the datum, modulo the alphabet size.
- `[` is a conditional jump: if the datum is 0, the instruction pointer jumps to the corresponding (matching) `]`.
- `]` is an unconditional jump to the corresponding `[`.[6]
- `,` copies the number under the reading tape pointer into the datum cell, and increments the reading pointer.
- `.` copies the datum to the output tape at the output pointer and increments the output pointer.

In this paper we do not use an input tape, so we do not use the `,` instruction.

When evaluating a program, the instruction pointer is initially on the first instruction, the output tape is empty, and the working tape is filled with zeros. Then the instruction under the instruction pointer is evaluated according to the above rules, and the instruction pointer is moved to the right. Evaluation terminates when the number of evaluated instructions reaches a given limit, or when the number of output symbols reaches a given limit.

For a sequence of instructions `A[B]C`, where `A`, `B` and `C` are sequences of (well-balanced) instructions, we call `B` the *body* of the block and `C` the *continuation* of the block.

E.2. BrainPhoque: Simultaneous generation and evaluation

We want to sample arbitrary BF programs and evaluate them for T steps each. To maximize computation efficiency of the sampling and running process, programs containing unbalanced parentheses are made valid, in particular by skipping any additional `]`.

Since we want to approximate *normalized* Solomonoff induction 3, we can make a few simplifications. In particular, programs do not need to halt explicitly, which removes the need for a halting symbol and behaviour.[7] Hence we consider that *all* programs are infinite, but that at most T instructions are evaluated. The difficulty with BF programs is that the evaluated instructions can be at arbitrary locations on the program tape, since large blocks `[...]` may be entirely skipped, complicating the sample-and-evaluate process.

This can be fixed by generating BF programs as trees, where branching on opening brackets `[`: The left branch corresponds to the body of the block (and terminates with a `]`), while the right branch corresponds to the continuation of the block. When encountering an opening bracket for the first time during evaluation, which branch is evaluated next depends on the datum. Hence, to avoid generating both branches, we need to generate the program *as it is being evaluated*: when sampling and evaluating a `[`, if the datum is 0 we follow the right branch and start sampling the continuation without having to sample the body (for now); conversely, if the datum is not zero, we follow the left branch and start sampling and evaluating the continuation. If the same opening bracket is later evaluated again with a different datum value, the other branch may be generated and evaluated.

Our implementation of program generation and evaluation in BrainPhoque uses one growing array for the program, one jump table, and one stack for yet-unmatched open brackets.

If the instruction pointer is at the end of the program, a new instruction among `+−<>[].` is sampled; if it is `[` and the datum is 0, it is changed to {. The new instruction is appended to the program, and is then evaluated. If the new instruction is `[`, the next instruction to be sample (and appended to the program) is the beginning of the body of the block, but if instead the new instruction is {, the next instruction to be sampled (and appended to the program) is the continuation of the body. At

[6] For efficiency reasons the instruction `]` is usually defined to jump to the matching `[` if the datum is non-zero. We stick to a unconditional jump for simplicity reasons.

[7] The halting behaviour can be recovered by ending programs with a particular infinite loop such as `[]+[]` (which loops whether the datum is zero or not), and terminate the evaluation (instead of looping forever) upon evaluating this sequence.

this point the jump table does not yet need to be updated — since the next instruction to evaluate is also the next instruction in location. The jump table is updated to keep track of where the continuations and bodies are located in the program. If the instruction pointer eventually comes back for a second time of an opening bracket `[` (resp. {) and the datum is now 0 (resp. not 0), the continuation (resp. body) of the block must now be sampled and appended to the program; and now the jump table must be updated accordingly.

The stack of unmatched brackets is updated only when the body of a block is being generated.

Some properties of BrainPhoque:

- If a program is run for $t + k$ steps, it behaves the same on the first t steps for all values of k.[8] In particular, unmatched opening brackets behave the same whether they will be matched or not.
- Program generation (sampling) only requires a single growing-only array. A tree structure is not required. This is the reason for having the additional { instruction, which makes it clear — once evaluated the second time — whether the body or the continuation has already been generated.
- If the instruction pointer is at cell n, then all instructions to the left of n have been evaluated at least once. If this is the first evaluation of cell n, then no instruction to the right of n have been evaluated yet.

It must be noted that, in order to fully adhere to Solomonoff's definition, the programming language needs to have access to a sequence x of random bits which probability must be $2^{\ell(x)}$. To enable this, the comma operator `,` can be used to read random bits from a input tape. The number of random bits read must be accounted for in the code length. For simplicity, in the experiments we set the probability of the comma operator to 0, which in essence only covers all deterministic sequences.

E.3. Solomonoff log-loss upper bound and shortening programs

We tried to provide a meaningful upper bound for the loss of Solomonoff induction for Figure 4, but this is far from easy. See Section 4 for context. As mentioned there, to calculate a more meaningful upper bound, we shorten programs by recursively removing unnecessary open brackets and closing brackets that are unmatched, as well as all self-cancelling pairs of instructions (+−, −+, <>,><). Moreover, we remove all instructions of the program that have been evaluated for the first time after the last evaluation of a print `.` instruction (since they do not participate in producing the output). This procedure often reduces programs by a third. Programs that do not output anything are thus reduced to the empty program (probability 1).

If q is a sampled program, then $\tilde{q}$ is the corresponding shortened program. We calculate an upper bound on the loss of the Solomonoff predictor, with U = BrainPhoque, on a set of sampled programs $\hat{Q} = (q^1, \dots, q^J)$ and corresponding outputs $(U(q^1)_{1:256}, \dots, U(q^J)_{1:256})$,

$$\text{Loss}(M_U, \hat{Q}) = \sum_{q \in \hat{Q}} - \log \sum_{p: U(p)_{1:256} = U(q)_{1:256}} 7^{-\ell(p)} \leq \sum_{q \in \hat{Q}} - \log 7^{-\ell(\tilde{q})} = \log(7) \sum_{q \in \hat{Q}} \ell(\tilde{q}) \tag{4}$$

since the program alphabet is not binary but has 7 instructions. Unfortunately, even after reduction this bound is still quite loose, but improving this bound meaningfully would likely require a much larger amount of computation.

E.4. Filtering out boring sequences

As an additional experiment, we built a Markov model to learn an improved distribution over the BrainPhoque instructions than the uniform one. See Table 4. To do this, we use a filter on programs and sequences to learn from. A sequence is 'boring' if its shortened program is longer than a threshold (program is too complex; we used a threshold of 100), or if its output is not of maximum length (program is too slow).

A sequence is also boring if it is too repetitive. See Algorithm 1 for a function that counts the number of repetitions in a sequence for a given delay between symbols. Then we search for the delay that maximizes the repeating count. If this maximum repeating count is larger than 70% of the sequence length, the sequence is boring. For example, the sequence "abababab..." is boring, while the sequence enumerating the binary numbers is not boring.

[8] While this is an obviously desirable property, it is also easy to overlook.

Markov chain order 0

Ctx.	<	>	+	-	[	]	.	Freq.
	.14	.14	.14	.15	.08	.08	.27	1.000

Markov chain order 1

Ctx.	<	>	+	-	[	]	.	Freq.
_	.19	.19	.19	.20	.00	.00	.23	.018
+	.18	.17	.17	.00	.14	.07	.27	.141
-	.17	.17	.00	.17	.13	.08	.28	.144
.	.14	.15	.15	.15	.07	.09	.25	.272
<	.17	.00	.19	.18	.06	.10	.30	.139
>	.00	.18	.17	.19	.05	.11	.30	.140
[	.15	.14	.15	.15	.12	.01	.28	.082
]	.15	.17	.16	.17	.01	.09	.25	.064

Markov chain order 2

Ctx.	<	>	+	-	[	]	.	Freq.
__	.19	.19	.19	.20	.00	.00	.23	.018
_+	.22	.24	.19	.00	.11	.00	.24	.004
_-	.15	.27	.00	.21	.13	.00	.24	.004
_.	.17	.22	.21	.17	.00	.00	.23	.004
_<	.23	.00	.28	.23	.00	.00	.26	.004
_>	.00	.18	.17	.26	.00	.00	.39	.003
++	.19	.17	.17	.00	.15	.06	.26	.023
+.	.15	.13	.14	.13	.11	.08	.26	.039
+<	.18	.00	.19	.19	.05	.09	.30	.025
+>	.00	.19	.19	.18	.06	.09	.29	.024
+[	.16	.14	.15	.14	.12	.01	.28	.020
+]	.12	.17	.17	.16	.02	.11	.25	.006
--	.17	.17	.00	.17	.15	.07	.27	.024
-.	.14	.15	.13	.14	.11	.09	.24	.040
-<	.17	.00	.20	.19	.07	.08	.29	.025
->	.00	.20	.18	.20	.05	.08	.29	.024
-[	.16	.14	.14	.15	.13	.01	.27	.019
-]	.18	.18	.18	.19	.01	.06	.20	.007
.+	.17	.16	.16	.00	.12	.09	.30	.041
.-	.17	.17	.00	.17	.11	.09	.29	.040
..	.14	.15	.16	.15	.06	.11	.23	.066
.<	.18	.00	.19	.17	.05	.10	.31	.039
.>	.00	.16	.18	.18	.05	.12	.31	.041
.[	.14	.15	.14	.17	.11	.01	.28	.019
.]	.16	.17	.16	.16	.01	.08	.26	.019
<+	.19	.18	.16	.00	.16	.04	.27	.026
<-	.21	.16	.00	.18	.13	.06	.26	.025
<.	.14	.16	.14	.15	.03	.11	.27	.042
<<	.18	.00	.19	.19	.05	.09	.30	.024
<[	.14	.16	.17	.16	.11	.01	.25	.008
<]	.14	.17	.16	.18	.01	.11	.23	.012
>+	.18	.16	.19	.00	.14	.06	.27	.025
>-	.17	.18	.00	.17	.15	.05	.28	.027
>.	.14	.14	.16	.16	.05	.10	.25	.042
>>	.00	.18	.18	.21	.05	.09	.29	.025
>[	.15	.15	.15	.17	.11	.01	.26	.007
>]	.16	.15	.15	.18	.01	.09	.26	.013
[+	.17	.14	.14	.00	.12	.13	.30	.012
[-	.17	.11	.00	.16	.09	.15	.32	.013
[.	.15	.17	.16	.14	.10	.01	.27	.023
[<	.15	.00	.12	.13	.07	.21	.32	.012
[>	.00	.15	.10	.14	.07	.21	.33	.012
[[	.14	.13	.17	.14	.09	.00	.33	.010
[]	.11	.11	.33	.11	.00	.06	.28	.001
]+	.16	.15	.16	.00	.14	.15	.24	.010
]-	.13	.17	.00	.15	.15	.17	.23	.011
].	.15	.19	.16	.19	.01	.08	.22	.016
]<	.17	.00	.17	.16	.07	.14	.29	.009
]>	.00	.19	.19	.19	.05	.12	.26	.011
][	.13	.13	.10	.27	.10	.00	.27	.001
]]	.13	.20	.17	.17	.00	.09	.24	.005

Table 4: **Pre-trained BP program sampling probabilities** Instead of sampling programs uniformly, we can sample them w.r.t. any probability distribution Q that satisfies Theorem 9. We initially sampled programs uniformly and filtered out 'boring' sequences. Then we trained Q via cross-entropy to mimic the distribution of 'interesting' sequences. We used a 2nd-order Markov process as a model for Q. While uniform sampling resulted in only 0.02% interesting sequences, sampling from Q increased it to 2.5%, a 137-fold improvement. The table on the left shows the 0th, 1st, and 2nd order Markov processes $Q(p_t)$, $Q(p_t|p_{t-1})$, and $Q(p_t|p_{t-2}p_{t-1})$ from which BP programs are sampled, for $p_. \in \{\texttt{<>+-[]\{.}\}$, but where results for `[` and $\{$ have been merged. Each row corresponds to a context (none or p_{t-1} or $p_{t-2}p_{t-1}$). We also included $Q(p_1|p_0{:=}_)$ and $Q(p_1|p_{-1}p_0{:=}__)$. The entries in each column correspond to the sampling probability of p_t in the corresponding row-context. Training on interesting sequences has led to a non-uniform distribution Q. Universality is preserved for any k-order Markov process, provided all transition probabilities are non-zero. The probability $Q(\texttt{.})$ of outputting a symbol has nearly doubled from 0.14 to 0.27 on average, while the probability of loop brackets (`[`, `]`) reduced to 0.07 each on average. The marginal probabilities $Q(<) \approx Q(>) \approx Q(+) \approx Q(-) \approx 1/7$ have not changed much, but many of the conditional ones have. Certain combination of instructions are now blocked: For instance $+-$ and $-+$ and $<>$ and $><$ have probability close to 0, since they cancel each other and hence are redundant. Some triples such as `][`$-$ and $<+$ and $>-$ and others are enhanced.

Caveat: We did not have time to retrain our NN models on these newly generated sequences (experiments are still running). But since the statistics is improved, we expect the results in Figures 4 and 5 to improve or at least not deteriorate.

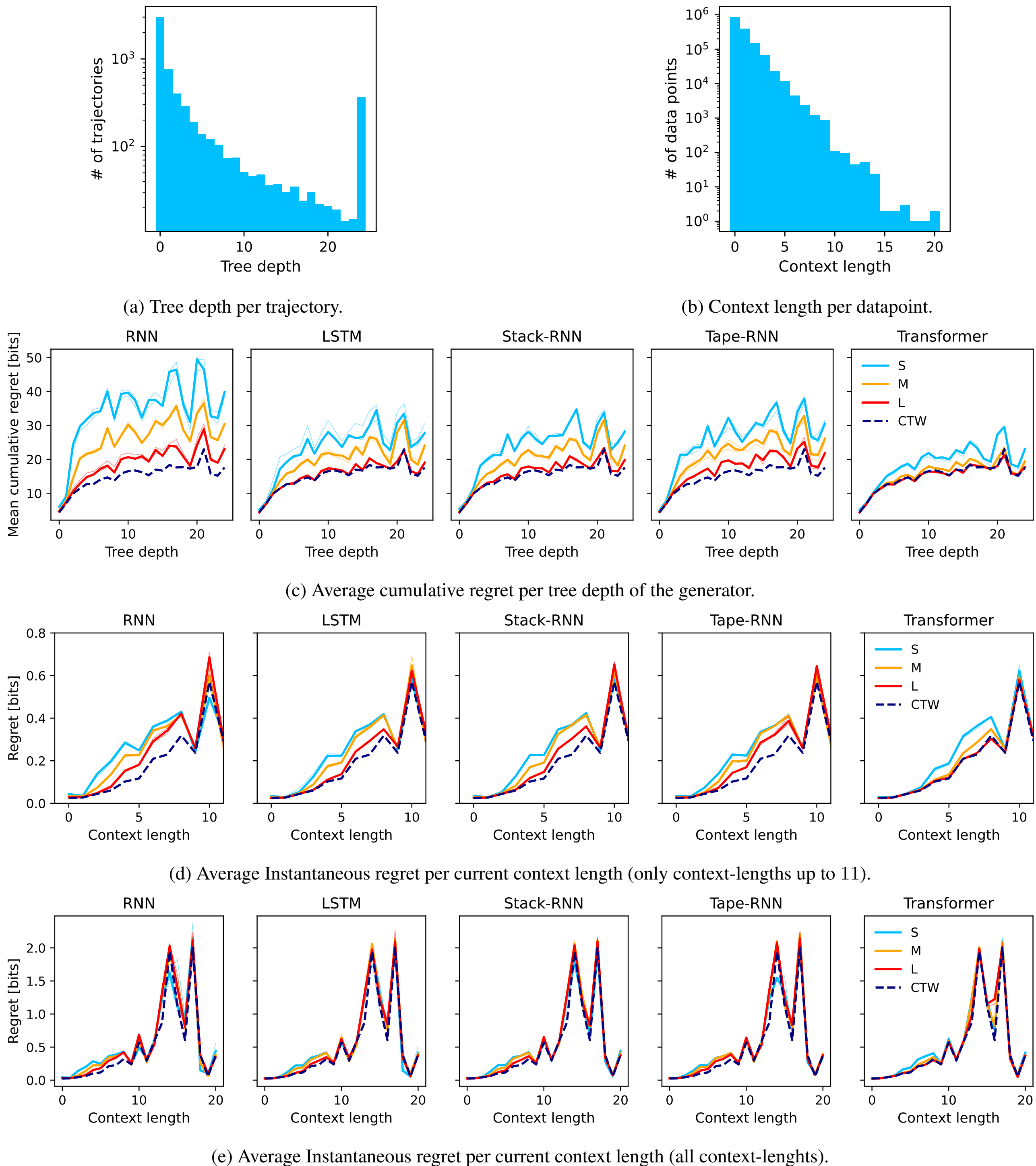
Figure 6: Some BrainPhoque programs and their corresponding outputs (truncated at 256 symbols). The smallest bars (in red) correspond to the value 0, and the largest bars (in gray) correspond to value 16. The programs have been reduced after evaluation by removing a set of unnecessary instructions. Most of the generated outputs are regular, and only about 1 in 5000 sampled programs exhibits non-regular patterns. But see Table 4 for a way to improve these numbers and generate more interesting and complex sequences.

F. Additional Results Details

Below we show additional results of the experiments on the VOMS (Figure 7), the Chomsky tasks (Figure 8) and UTM source (Figures 10 and 11). Finally, on Figure 12 we show further details of the length generalization analysis.

Algorithm 1 Returns the number of repetitions in sequence for a given delay between symbols.

```
def repeating_count(output, delay):
  count = 0 # number of equal elements
  for i in range(delay + 1, len(output)):
    if output[i] == output[i-delay]: count += 1
  return count
```



(a) Tree depth per trajectory.

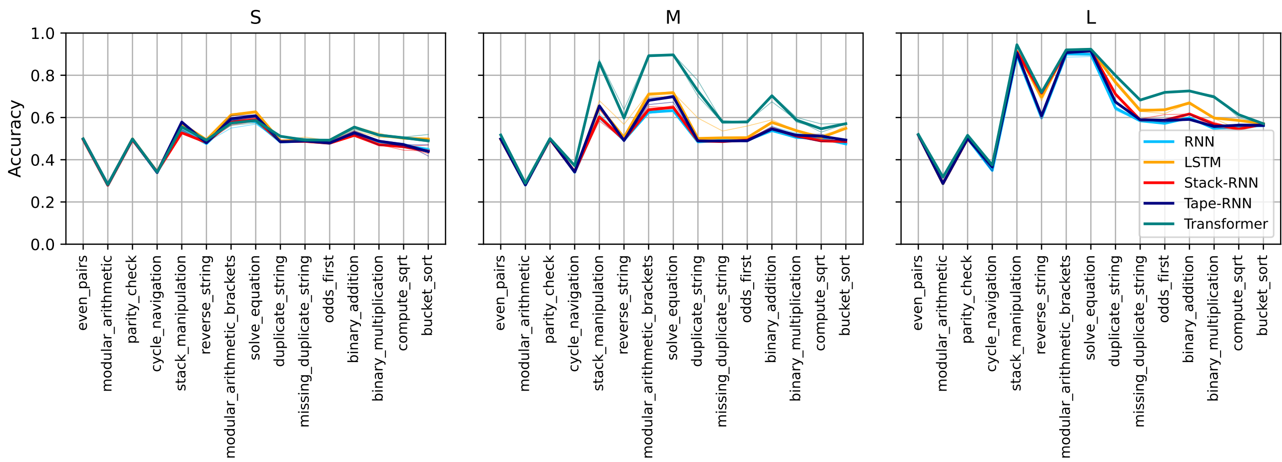
(b) Context length per datapoint.

(c) Average cumulative regret per tree depth of the generator.

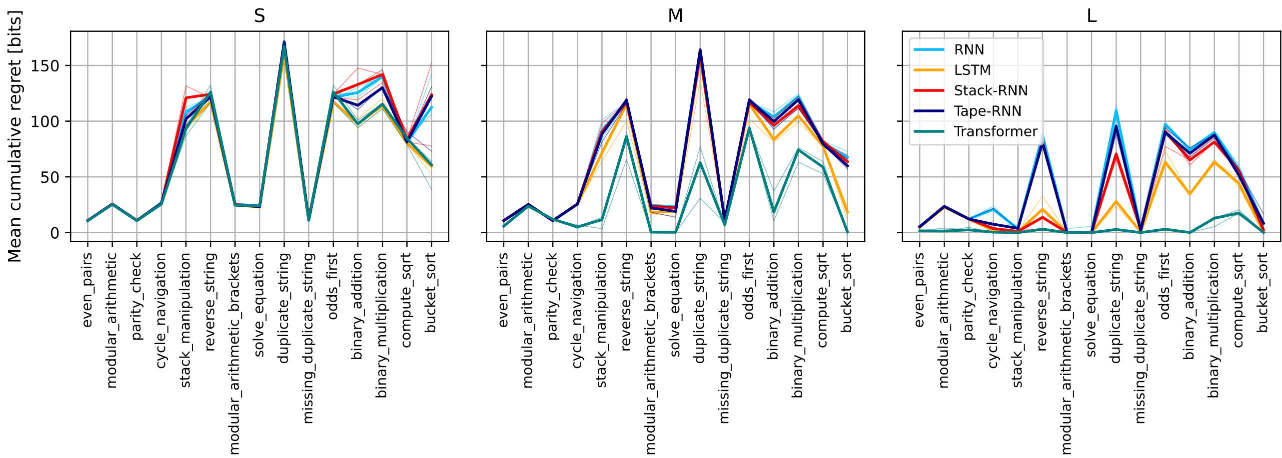
(d) Average Instantaneous regret per current context length (only context-lengths up to 11).

(e) Average Instantaneous regret per current context length (all context-lenghts).

Figure 7: Detailed results for the same 6k sequences as in Figure 2. Top two panels show histograms over tree depth (for all trajectories) and current context length (over all datapoints of all trajectories) use for evaluation in Figure 2. As expected, most generated trees have low depth and most datapoints have short contexts. The three lower panels show average cumulative regret per tree depth, and average instantaneous regret per context length respectively. Thin lines correspond to individual models (with different random initialization), bold lines show the median per model size. Across architectures smaller models only predict well for very short tree depth or very short context lengths (the maximum context length is upper bounded by the tree depth, but many contexts are much shorter than the maximum tree depth). Context lenghts ≥ 11 are rare, which makes quantitative results in this regime less reliable.



(a) Mean accuracy per Chomsky task, grouped by model size.



(b) Mean cumulative regret per Chomsky task, grouped by model size.

Figure 8: Detailed performance of networks trained and evaluated on the Chomsky tasks (6k sequences, 400 sequences per task; main results shown in Figure 3). Thin lines correspond to a single random initialization of a model, bolt lines show the median respectively.

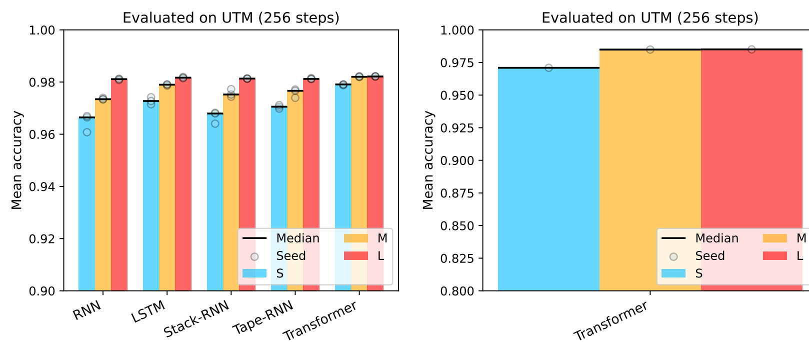
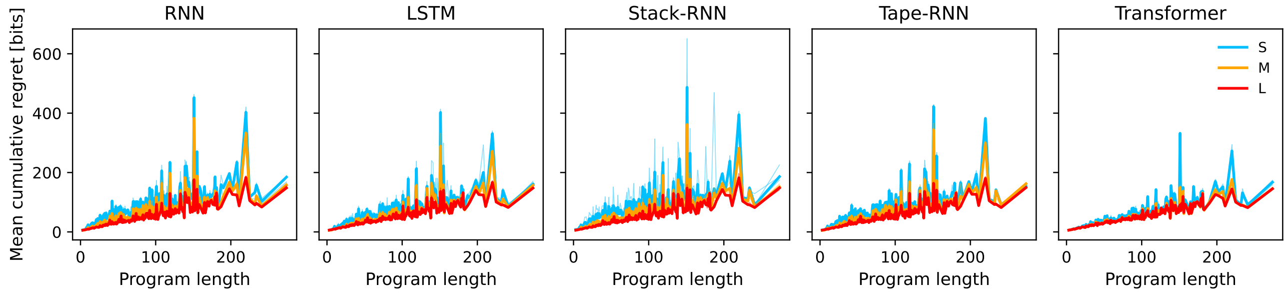
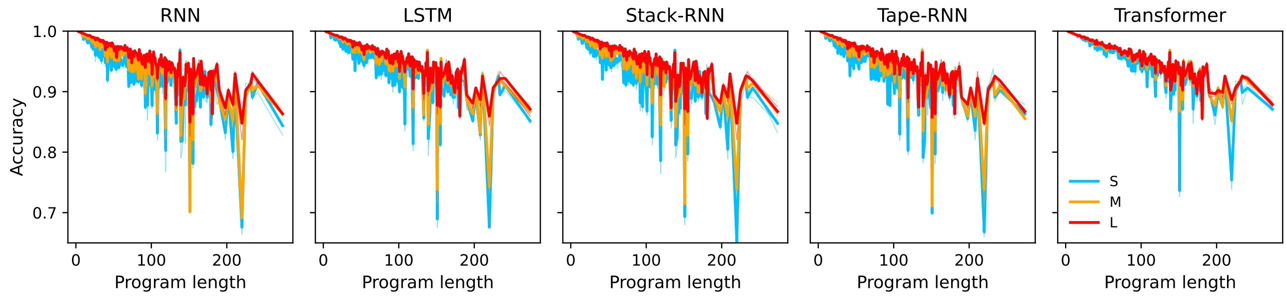


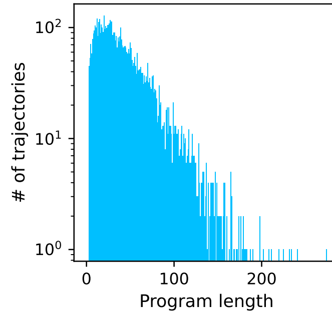
Figure 9: Additional UTM Figures. Left: Zoomed-in version of Figure 4. Right: UTM Experiments with alphabet size 500.



(a) Average cumulative regret per program length.



(b) Average accuracy per program length.



(c) Histogram over program lengths.

Figure 10: Results per program length for UTM in-distribution evaluation (same data as in Figure 4; 6k sequences, length 256).

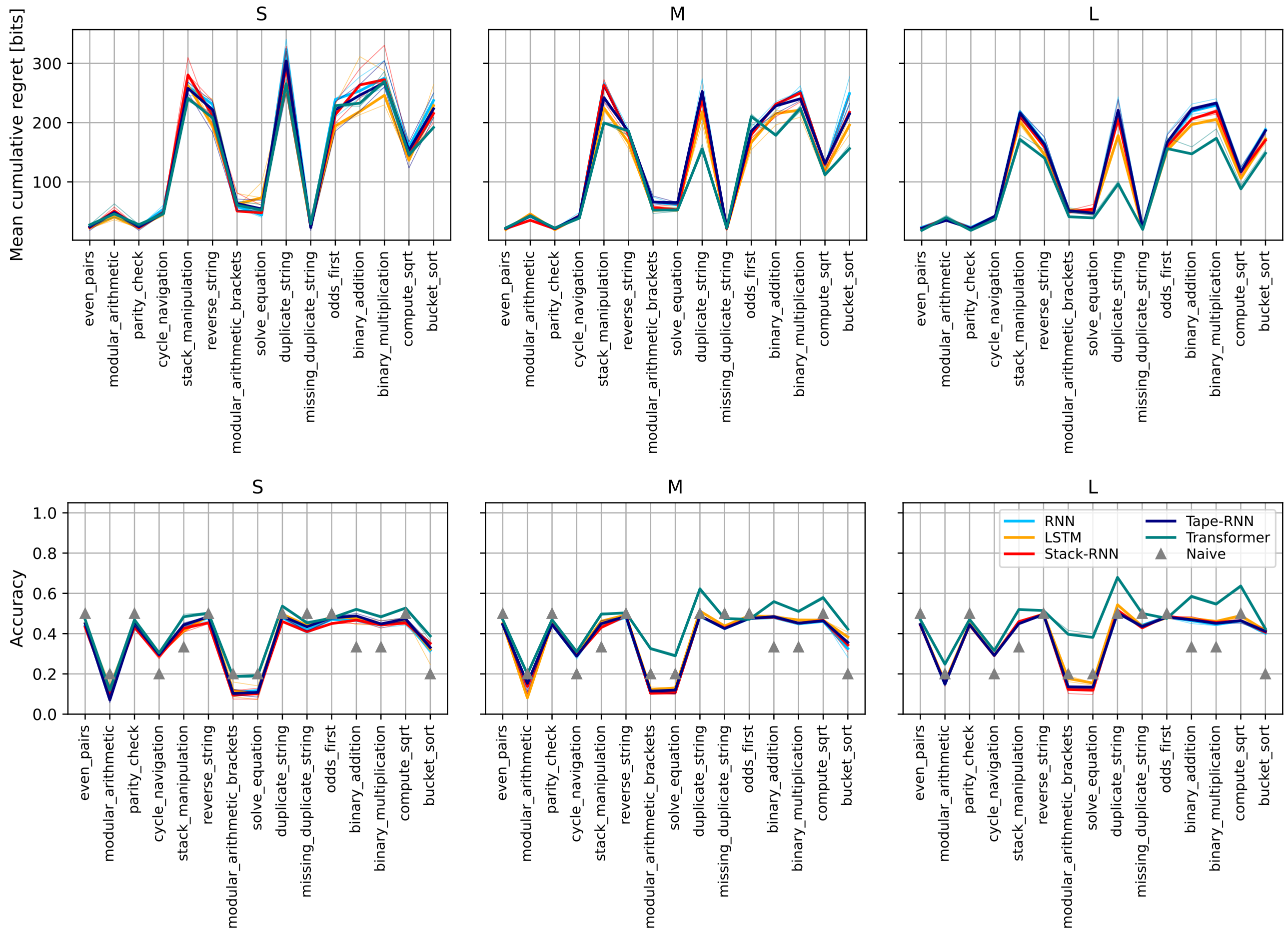
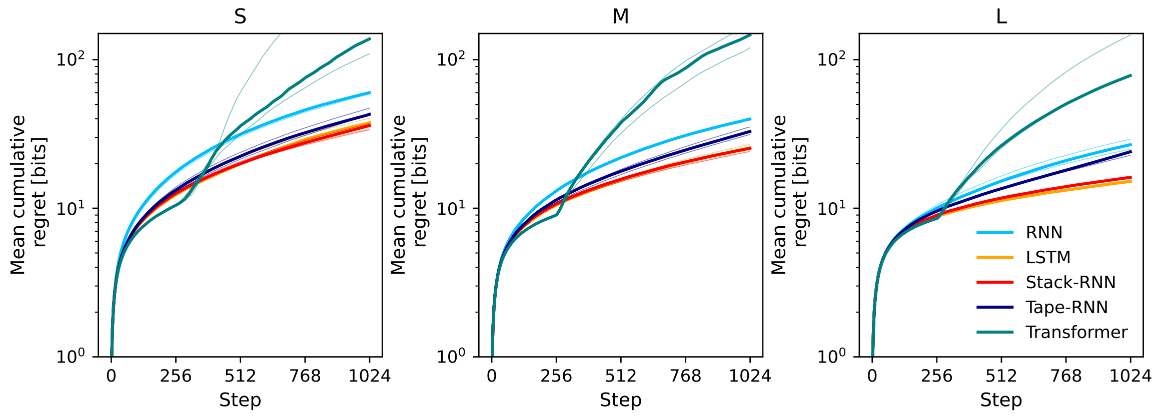
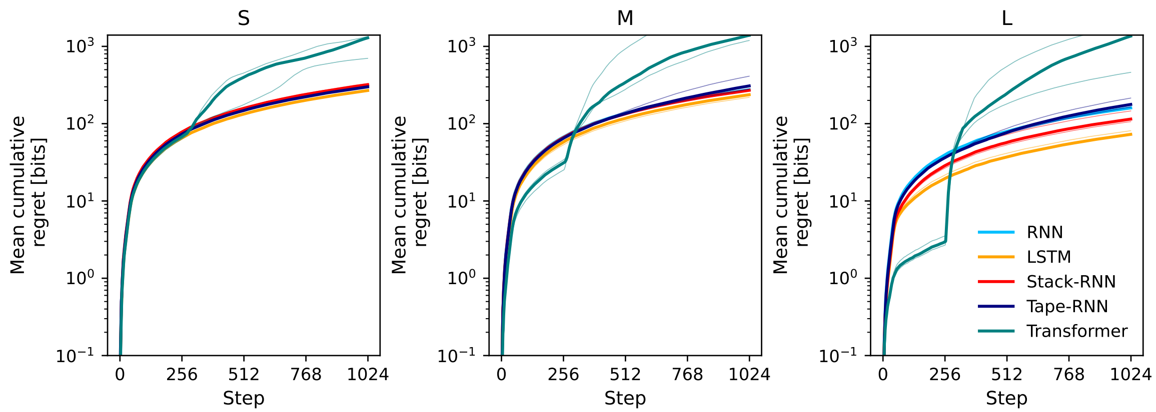


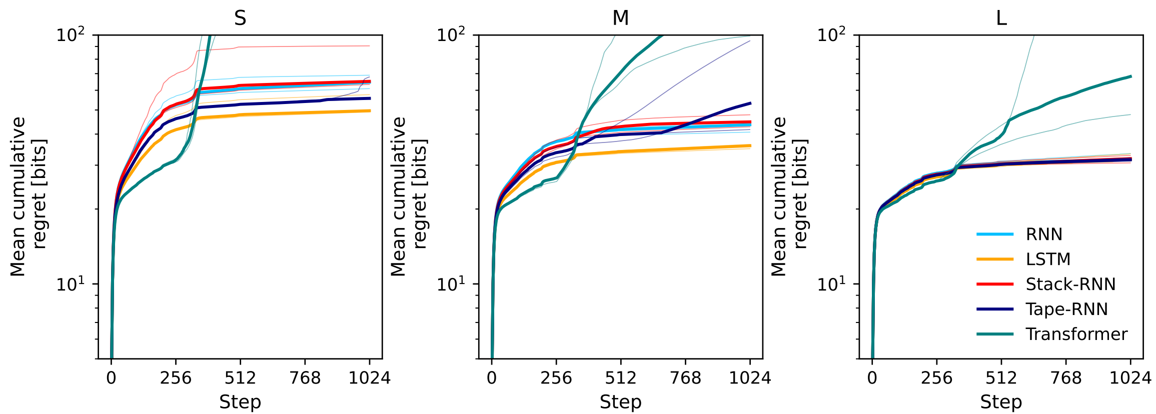
Figure 11: UTM transfer to Chomsky tasks.



(a) Variable-order Markov source (CTW) data.



(b) Chomsky tasks.



(c) UTM data.

Figure 12: Full details of sequence-length generalization results. Models were trained on sequences of length 256 on their respective tasks, and are evaluated on 6k sequences of length 1024 from the same data generator type. Thin lines show individual models, bold lines are the median across random initializations of the same model. As expected, all models perform fairly well up to their trained sequence length, and then performance deteriorates more or less sharply. Most notably, prediction performance of the transformer models, regardless of their size, degrades very rapidly after step 256 and is often an order of magnitude worse than the other models. Across all experiments, LSTMs perform best in terms of generalizing to longer sequences.