Artificial Intelligence 299 (2021) 103521

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

Making sense of raw input ☆

Richard Evans [a,b,*], Matko Bošnjak [b,c], Lars Buesing [b], Kevin Ellis [d], David Pfau [b], Pushmeet Kohli [b], Marek Sergot [a]

[a] Imperial College London, United Kingdom
[b] DeepMind, London, United Kingdom
[c] University College London, United Kingdom
[d] Massachusetts Institute of Technology, United States of America

ARTICLE INFO

Article history:
Received 17 December 2019
Received in revised form 14 December 2020
Accepted 28 April 2021
Available online 3 May 2021

Keywords:
Interpretable AI
Unsupervised theory learning
Neuro-symbolic integration

ABSTRACT

How should a machine intelligence perform unsupervised structure discovery over streams of sensory input? One approach to this problem is to cast it as an *apperception task* [1]. Here, the task is to construct an explicit interpretable theory that both explains the sensory sequence and also satisfies a set of unity conditions, designed to ensure that the constituents of the theory are connected in a relational structure.

However, the original formulation of the apperception task had one fundamental limitation: it assumed the raw sensory input had already been parsed using a set of discrete categories, so that all the system had to do was receive this already-digested symbolic input, and make sense of it. But what if we don't have access to pre-parsed input? What if our sensory sequence is raw unprocessed information?

The central contribution of this paper is a neuro-symbolic framework for distilling interpretable theories out of streams of raw, unprocessed sensory experience. First, we extend the definition of the apperception task to include ambiguous (but still symbolic) input: sequences of sets of disjunctions. Next, we use a neural network to map raw sensory input to disjunctive input. Our binary neural network is encoded as a logic program, so the weights of the network and the rules of the theory can be solved jointly as a single SAT problem. This way, we are able to jointly learn how to perceive (mapping raw sensory information to concepts) and apperceive (combining concepts into declarative rules).

1. Introduction

There are, broadly speaking, two approaches to interpreting the results of machine learning systems [2–4]. In one approach, *post-hoc interpretation*, we take an existing machine learning system, that has already been trained, and try to understand its inner state. In the other approach, *designing explicit already-interpretable machine learning systems*, we constrain the design of the machine learning system to guarantee, in advance, that its results will be interpretable.

In this paper, we take the second approach to unsupervised learning. Our system takes as input a temporal sequence of raw unprocessed sensory information, and produces an interpretable theory capturing the regularities in that sequence. It combines an unsupervised program synthesis system for constructing explicit first-order theories, with a binary neural net-

☆ This paper is part of the Special Issue on Explainable AI.

* Corresponding author at: Imperial College London, United Kingdom.
E-mail address: richardevans@google.com (R. Evans).

https://doi.org/10.1016/j.artint.2021.103521

work that transforms raw unprocessed sensory information into symbolic information that can be accessed by the program synthesis system. Thus, the system jointly synthesizes an explanatory symbolic theory, connected to a learned, sub-symbolic perceptual front-end.

1.1. Unsupervised learning and apperception

Consider a machine, equipped with various sensors, receiving a stream of sensory information. Somehow, it must make sense of this sensory stream. But what, exactly, does "making sense" involve, and how, exactly, should it be performed?

Unsupervised learning occupies a curious position within the space of AI tasks in that, while it is acknowledged to be of central importance to the progress of the field,[1] it is also frustratingly ill-defined. What, exactly, does it mean to "make sense" of unlabelled data? There is no consensus on what the problem is, let alone the solution.

Self-supervised learning has emerged as a well-defined sub-field within unsupervised learning [6,7]. Here, the task is to use the unlabelled sensory sequence as a source of supervised learning problems: we try to predict future states given previous states. Now, the vague under-specified unsupervised learning problem has been replaced by the well-defined task of predicting certain data points conditioned on others.

But there is more, we submit, to making sense than just predicting (or retrodicting) held-out states. Predicting future held-out states is certainly *part* of what is involved in making sense of the sensory given—but it is not, on its own, sufficient.

Recently, we proposed an alternative approach to unsupervised learning [1]. The problem of "making sense" of sequences is formalised as an *apperception task*. Here, the task is to construct an explicit theory that both explains the sequence and also satisfies a set of unity conditions designed to ensure that the constituents of the theory—the objects, properties, and propositions—are combined together in a relational structure. We developed an implementation, the Apperception Engine, and showed, in a range of experiments, how this system is able to outperform recurrent networks and other baselines on a range of tasks, including Hofstadter's *Seek Whence* dataset [8].

But in our initial implementation, there was one fundamental limitation: we assumed the sensory input was provided in symbolic form. We assumed *some other system had already parsed the raw sensory input into a set of discrete categories*, so that all the Apperception Engine had to do was receive this already-digested symbolic input, and make sense of it. But what if we don't have access to pre-parsed input? What if our sensory sequence is raw unprocessed information—a sequence of noisy pixel arrays from a video camera, for example?

1.2. Overview

Our central contribution is an approach for unsupervised learning of interpretable symbolic theories from raw unprocessed sensory data. We achieve this through a major extension of the Apperception Engine so that it is able to work from this raw input. This involves two phases. First, we extend the Apperception Engine to receive ambiguous (but still symbolic) input: sequences of disjunctions. Second, we use a neural network to map raw sensory input to disjunctive input. Our binary neural network is encoded as a logic program, so the weights of the network and the rules of the theory can be found *jointly* by solving a single SAT problem. This way, we are able to simultaneously learn how to perceive (mapping raw sensory information to concepts) and apperceive (combining concepts into rules).

We tested our system in three domains. In the first domain, the Apperception Engine learned to solve sequence induction tasks, where the sequence was represented by noisy MNIST images [9]. In the second, it learned the dynamics of *Sokoban* from a sequence of noisy pixel arrays. In the third, it learned to make sense of sequences of noisy ambiguous data without knowledge of the underlying spatial structure of the generative model.

This system is, to the best of our knowledge, the first system that is able to learn explicit provably correct dynamics of non-trivial games from raw pixel input. We discover that generic inductive biases embedded in our system suffice to induce these game dynamics from very sparse data, i.e. less than two dozen game traces. We see this as a step toward machines that can flexibly adapt and even synthesize their own world models [10,11], starting from raw sub-symbolic input, while organizing and representing those models in a format that humans can comprehend, debug, and verify.

In Section 3, we describe the Apperception Engine as it operates on discrete symbolic input. In Section 4, we extend the system to handle raw unprocessed input. Section 5 shows our experimental results.

2. Background

2.1. Logic programming

In this paper, we use basic concepts and standard notation from first-order logic [12] and from logic programming [13–15]. A function-free atom is an expression of the form $p(t_1, \ldots, t_n)$, where p is a predicate of arity $n \geq 0$ and each t_i is

[1] Yann LeCun famously said that "If intelligence was a cake, unsupervised learning would be the cake, supervised learning would be the icing on the cake, and reinforcement learning would be the cherry on the cake. We know how to make the icing and the cherry, but we don't know how to make the cake." [5].

either a variable or a constant. We shall use $a, b, c, \ldots$ for constants, $X, Y, Z, \ldots$ for variables, and $p, q, r, \ldots$ for predicate symbols.

A **ground substitution** is a mapping from variables to constants. For example, applying ground substitution $\sigma = \{X \mapsto a, Y \mapsto b\}$ to atom $p(X, Y, X)$, written $p(X, Y, X)[\sigma]$, produces $p(a, b, a)$.

A set of first-order formulas $\alpha_1, \ldots, \alpha_n$ **entails** β, written $\alpha_1, \ldots, \alpha_n \models \beta$, if every interpretation that satisfies $\alpha_1, \ldots, \alpha_n$ also satisfies β. For details, see any standard logic textbook [12].

A **Datalog clause** is a definite clause of the form $\alpha_1 \wedge \ldots \wedge \alpha_n \rightarrow \alpha_0$ where each α_i is an atom and $n \geq 0$. A **Datalog program** is a set of Datalog clauses.

2.2. Related work

Unsupervised modelling of temporal sequences to understand and predict the underlying data-generating process is at the heart of statistics, engineering and the sciences, with countless applications. Most sequence models are probabilistic approaches and model the distribution of observed time series, allowing us to model the uncertainty and ambiguity arising from noisy measurements or partial observability. Traditionally, due to computational limitations, sequence models were informed by strong prior knowledge about particular domains and usually contained few tunable parameters, e.g. [16,17]. The advent of deep learning methods and the availability of large scale data sets has revolutionized sequence modelling [18,19] with applications in e.g. language modelling [20], model-based reinforcement learning [21] and health care [22], by incorporating general learned function approximation to replace strong prior assumptions.

Probabilistic sequence models can broadly be divided into two classes (with some overlap): *auto-regressive* models and *latent variable* models. Auto-regressive models directly capture the temporal dependencies of observed variables by modelling the distribution of each time slice conditioned on the history of observations. This results in conceptually simple and fast model training procedures using maximum-likelihood estimation, without having to commit to restrictive conditional independence assumptions [23–27]. At the same time, auto-regressive models for raw sensory data are usually large and therefore costly to evaluate at test time, hindering their application to e.g. planning problems with long-horizons where thousand of model evaluations are often necessary. Furthermore, they are usually not humanly interpretable and specifying prior domain knowledge is difficult.

Latent variable models can in principle address these shortcomings [28]. They aim to capture the statistical dependencies of time series by positing latent variables that underlie the observed data [29,30]. This latent structure is usually assumed to be simpler than the observed high-dimensional raw data, and can therefore in principle reduce the computational demands of long-rage predictions, therefore facilitating applications to e.g. reinforcement learning and planning [31,32,11]. Furthermore, low-dimensional latent models are often used for finding simple, structured explanations in exploratory data analysis [33,34]. Allowing for uncertainty over latent entities in order to capture multiple hypotheses of observed phenomena comes at the price of necessitating sophisticated approaches for fitting models to data. Most approaches either apply explicit probabilistic inference to determine the distributions over latent variables from observations, or they rely on implicit or likelihood-free methods [35,36].

Modern latent variable sequence models mostly capture latent structure with contiguous variables, as this allows approximate model-fitting methods with gradient descent on parameters as a sub-routine [37,38]. However, predictions under these models are often subject to degrading fidelity with increasing prediction horizon. Although often inevitable to some degree, this effect is exacerbated by the inability of latent variables to capture discrete, or categorical structure, resulting in a "conceptual drift" in domains where the ground truth can be described by discrete concepts. Discrete (or mixed discrete-continuous [39]) latent variable models can in principle alleviate this issues [40,41], however, fitting these efficiently to data remains challenging in spite of recent progress [42,43], making them often worse models than their continuous counterparts. Nevertheless, due to their attractive properties of compactness and interpretability, recent efforts have attempted to apply them to model-based reinforcement learning [44,45].

Neural-symbolic approaches. Neural networks and inductive logic programming have complementary advantages and disadvantages [46]. Neural networks are robust to noisy and ambiguous input, and can scale to enormous datasets, but they are data-hungry and can fail to generalise effectively to data outside the training distribution. Inductive logic programming, by contrast, is data-efficient and can achieve strong generalisation, but is often unable to handle noisy data, and struggles to handle ambiguous data (e.g., the raw pixel data from a camera).

Neural-symbolic computing (**NeSy**) is a collection of approaches that aim to combine the best of both worlds [47–49]. In these approaches, logical inference is re-implemented in a neural network: discrete facts with values from $\{0, 1\}$ are replaced by probabilities in $[0, 1]$, and crisp logical operators (such as Boolean conjunction) are replaced by probabilistic or fuzzy functions (such as a t-norm). This approach has been applied to classical propositional logic [50,51], non-classical propositional logics and modal logics [52,53], and first-order logic [54–56].

Our approach is similar to NeSy in that both approaches share the common goal of trying to combine the best of neural and symbolic models, by unifying the noise-robustness of neural nets with the strong generalisation of symbolic methods. But they differ in how exactly to achieve that unification: while NeSy reimplements logical inference inside a neural network, our approach reimplements the neural network inside a logical system. Instead of taking the neural network as basic,

and reimplementing logical inference inside the neural network, we take the logical system as basic, and reimplement the neural network inside the logical system (see Section 4.4).

Relational reinforcement learning. Another approach that learns first-order rules to achieve strong generalisation is relational reinforcement learning [57,58]. The key difference between their approach and ours is that their learned rules are used to represent a *policy*, while our learned rules are used to represent the next state transition function.

3. Discrete apperception

In describing the Apperception Engine, we shall use three increasingly complex forms of sequential input. We start by assuming that the sensory sequence has already been discretised into ground atoms of first-order logic representing sensor readings. So, for example, the (discrete) fact that light sensor a is *red* might be represented by the atom $red(a)$. Next, we expand to consider disjunctive input sequences. For example, to represent that sensor a is either *red* or *orange* we would represent our uncertainty by $red(a) \vee orange(a)$. Finally, we let go entirely of our simplifying assumption of already-discretised sensory input and consider sequences of *raw unprocessed input*. Consider, for example, a sequence of pixel arrays from a video camera.

3.1. Making sense of unambiguous symbolic input

Definition 1. An **unambiguous symbolic sensory sequence** is a sequence of sets of ground atoms. Given a sequence $S = (S_1, S_2, \ldots)$, a **state** S_t is a set of ground atoms, representing a *partial* description of the world at a discrete time step t.

Example 1. Consider, the following sequence $S_{1:10}$. Here there are two sensors a and b, and each sensor can be either *on* or *off*.

$$
\begin{array}{lll}
S_1 = \{\} & S_2 = \{off(a), on(b)\} & S_3 = \{on(a), off(b)\} \\
S_4 = \{on(a), on(b)\} & S_5 = \{on(b)\} & S_6 = \{on(a), off(b)\} \\
S_7 = \{on(a), on(b)\} & S_8 = \{off(a), on(b)\} & S_9 = \{on(a)\} \\
S_{10} = \{\} &&
\end{array}
$$

Note that there is no expectation that a sensory sequence contains readings for all sensors at all time steps. Some of the readings may be missing. In state S_5, we are missing a reading for a, while in state S_9, we are missing a reading for b. In states S_1 and S_{10}, we are missing sensor readings for both a and b. ◁

The Apperception Engine makes sense of a sensory sequence by constructing a unified theory that explains that sequence. The key notions, here, are "theory", "explains", and "unified". We consider each in turn.

Definition 2. A **theory** is a four-tuple (ϕ, I, R, C) where:

- ϕ is a type signature specifying the types of constants, variables, and arguments of predicates
- I is a set of initial conditions (ground atoms, well-typed according to ϕ)
- R is a set of rules in $Datalog^{\ominus}$, an extension of Datalog that incorporates rules describing how properties change over time
- C is a set of constraints

There are two types of rule in $Datalog^{\ominus}$. A **static rule** is a definite clause of the form $\alpha_1 \wedge \ldots \wedge \alpha_n \rightarrow \alpha_0$, where $n \geq 0$ and each α_i is an *unground* atom consisting of a predicate and a list of variables. Informally, a static rule is interpreted as: if conditions $\alpha_1, \ldots \alpha_n$ hold at the current time step, then α_0 also holds at that time step. A **causal rule** is a clause of the form $\alpha_1 \wedge \ldots \wedge \alpha_n \ominus \alpha_0$, where $n \geq 0$ and each α_i is an unground atom. A causal rule expresses how facts change over time. Rule $\alpha_1 \wedge \ldots \wedge \alpha_n \ominus \alpha_0$ states that if conditions $\alpha_1, \ldots \alpha_n$ hold at the current time step, then α_0 holds at the *next* time step. All variables in rules are implicitly universally quantified. So, for example, $on(X) \ominus off(X)$ states that for all objects X, if X is currently *on*, then X will become *off* at the next-time step.

There are three types of constraint. A **unary constraint** is an expression of the form $\forall X, p_1(X) \oplus \ldots \oplus p_n(X)$, where $n > 1$, meaning that for all X, exactly one of $p_1(X), \ldots, p_n(X)$ holds.[2] A **binary constraint** is an expression of the form $\forall X, \forall Y, r_1(X, Y) \oplus \ldots \oplus r_n(X, Y)$ where $n > 1$, meaning that for all objects X and Y, exactly one of the binary relations hold. A **uniqueness constraint** is an expression of the form $\forall X{:}t_1, \exists! Y{:}t_2, r(X, Y)$, which means that for all objects X of type t_1 there exists a *unique* object Y such that $r(X, Y)$.

[2] We write $p_1(X) \oplus \ldots \oplus p_n(X)$ to mean exclusive disjunction between n atoms, not the application of $n - 1$ *xor* operations.

Example 2. Consider the type signature $\phi = (T, O, P, V)$, consisting of types $T = \{s\}$, objects $O = \{a{:}s, b{:}s\}$, predicates $P = \{on(s), off(s), r(s,s), p_1(s), p_2(s), p_3(s)\}$, and variables $V = \{X{:}s, Y{:}s\}$. Consider the theory $\theta = (\phi, I, R, C)$, where:

$$I = \left\{ \begin{array}{l} p_1(b) \\ p_2(a) \\ r(a,b) \\ r(b,a) \end{array} \right\} \quad R = \left\{ \begin{array}{l} p_1(X) \Rightarrow p_2(X) \\ p_2(X) \Rightarrow p_3(X) \\ p_3(X) \Rightarrow p_1(X) \\ p_1(X) \rightarrow on(X) \\ p_2(X) \rightarrow on(X) \\ p_3(X) \rightarrow off(X) \end{array} \right\} \quad C = \left\{ \begin{array}{l} \forall X{:}s,\ on(X) \oplus off(X) \\ \forall X{:}s,\ p_1(X) \oplus p_2(X) \oplus p_3(X) \\ \forall X{:}s,\ \exists! Y{:}s\, r(X,Y) \end{array} \right\} \quad \triangleleft$$

Definition 3. Every theory $\theta = (\phi, I, R, C)$ generates an infinite sequence $\tau(\theta)$ of sets of ground atoms, called the **trace** of that theory. Here, $\tau(\theta) = (A_1, A_2, \ldots)$, where each A_t is the smallest set of atoms satisfying the following conditions:

- $I \subseteq A_1$
- If there is a static rule $\beta_1 \wedge \ldots \wedge \beta_m \rightarrow \alpha$ in R and a ground substitution σ such that A_t satisfies $\beta_i[\sigma]$ for each antecedent β_i, then $\alpha[\sigma] \in A_t$
- If there is a causal rule $\beta_1 \wedge \ldots \wedge \beta_m \Rightarrow \alpha$ in R and a ground substitution σ such that A_{t-1} satisfies $\beta_i[\sigma]$ for each antecedent β_i, then $\alpha[\sigma] \in A_t$
- *Frame axiom*: if α is in A_{t-1} and there is no atom in A_t that is incompossible with α w.r.t. constraints C, then $\alpha \in A_t$. Two ground atoms are **incompossible** if there is some constraint c in C and some substitution σ such that the ground constraint $c[\sigma]$ precludes both atoms being true. The intuition here is that once something is true, it keeps on being true until something else becomes true that is incompatible with it.

Note that the state transition function is deterministic: A_t is uniquely determined by A_{t-1}.

Example 3. The infinite trace $\tau(\theta) = (A_1, A_2, \ldots)$ for the theory θ of Example 2 begins with:

$$A_1 = \{on(a), on(b), p_2(a), p_1(b), r(a,b), r(b,a)\}$$
$$A_2 = \{off(a), on(b), p_3(a), p_2(b), r(a,b), r(b,a)\}$$
$$A_3 = \{on(a), off(b), p_1(a), p_3(b), r(a,b), r(b,a)\}$$
$$A_4 = \{on(a), on(b), p_2(a), p_1(b), r(a,b), r(b,a)\}$$
$$\ldots$$

Note that the trace repeats at step 4. It is possible to show in general that the trace always repeats after some finite set of time steps, since the set of ground atoms is finite and the state transition of Definition 3 is Markov [1]. $\triangleleft$

Next we define what it means for a theory to "explain" a sensory sequence.

Definition 4. Given finite sequence $S = (S_1, \ldots, S_T)$ and (not necessarily finite) S', $S \sqsubseteq S'$ if $S' = (S'_1, S'_2, \ldots)$ and $S_i \subseteq S'_i$ for all $1 \leq i \leq T$. If $S \sqsubseteq S'$, we say that S is **covered by** S', or that S' covers S. A theory θ **explains** a sensory sequence S if the trace of θ covers S, i.e. $S \sqsubseteq \tau(\theta)$.

In providing a theory θ that explains a sensory sequence S, we make S intelligible by placing it within a bigger picture: while S is a scanty and incomplete description of a fragment of the time-series, $\tau(\theta)$ is a complete and determinate description of the whole time-series.

Example 4. The theory θ of Example 2 explains the sensory sequence S of Example 1, since the trace $\tau(\theta)$ (as shown in Example 3) covers S. Note that $\tau(\theta)$ "fills in the blanks" in the original sequence S, both predicting final time step 10, retrodicting initial time step 1, and imputing missing values for time steps 5 and 9. $\triangleleft$

Next, we proceed from explaining a sensory sequence to "making sense" of that sequence. In order for θ to make sense of S, it is *necessary* that $\tau(\theta)$ covers S. But this condition is not, on its own, *sufficient*. The extra condition that is needed for θ to count as "making sense" of S is for θ to be *unified*. In this paper, we formalize what it means for a theory to be "unified" using elements from Kant's discussion of the "synthetic unity of apperception" [95].[3]

[3] In this paper, we do not focus on Kant exegesis, but do provide some key references. All references to the *Critique of Pure Reason* use the standard [A/B] reference system, where A refers to the First Edition (1781), and B refers to the Second Edition (1787). "The principle of the synthetic unity of apperception is the supreme principle of all use of the understanding" [B136]; it is "the highest point to which one must affix all use of the understanding, even the whole of logic and, after it, transcendental philosophy" [B134].

Definition 5. A trace $\tau(\theta)$ is (i) a sequence of (ii) sets of ground atoms composed of (iii) predicates and (iv) objects.[4] For the theory θ to be **unified** is for unity to be achieved at each of the following four levels:

- Objects are united in space.[5] A theory θ satisfies **spatial unity** if for each state A_i in $\tau(\theta) = (A_1, A_2, \ldots)$, for each pair (x, y) of distinct objects, x and y are connected via a chain of binary atoms $\{r_1(x, z_1), \ldots r_n(z_{n-1}, z_n), r_{n+1}(z_n, y)\}$ in A_i. The intuition here is that every pair of objects are always connected, directly or indirectly.
- Predicates are united via constraints.[6] A theory $\theta = (\phi, I, R, C)$ satisfies **conceptual unity** if for each unary predicate p in ϕ, there is some xor constraint in C of the form $\forall X{:}t,\ p(X) \oplus q(X) \oplus \ldots$ containing p; and, for each binary predicate r in ϕ, there is either some xor constraint in C of the form $\forall X{:}t_1, \forall Y{:}t_2, r(X, Y) \oplus s(X, Y) \oplus \ldots$ or some $\exists!$ constraint in C of the form $\forall X{:}t, \exists! Y{:}t_2, r(X, Y)$. The intuition here is that every predicate is connected to some other predicate via some constraint.
- Ground atoms are united into sets (states) by jointly respecting constraints and static rules.[7] A theory $\theta = (\phi, I, R, C)$ satisfies **static unity** if every state A_i in $\tau(\theta) = (A_1, A_2, \ldots)$ satisfies all the constraints in C and is closed under the static rules in R.
- States (sets of ground atoms) are united into a sequence by causal rules.[8] A sequence $(A_1, A_2, \ldots)$ of states satisfies **temporal unity** with respect to a set $R_{\Rightarrow}$ of causal rules if, for each $\alpha_1 \wedge \ldots \wedge \alpha_n \Rightarrow \alpha_0$ in $R_{\Rightarrow}$, for each ground substitution σ, for each time step t, if $\{\alpha_1[\sigma], \ldots, \alpha_n[\sigma]\} \subseteq A_t$ then $\alpha_0[\sigma] \in A_{t+1}$. Note that, from the definition of the trace in Definition 3, the trace $\tau(\theta)$ *automatically* satisfies temporal unity.

Example 5. The theory θ of Example 2 satisfies the four unity conditions since:

- For each state A_i in $\tau(\theta)$, a is connected to b via the singleton chain $\{r(a, b)\}$, and b is connected to a via $\{r(b, a)\}$.
- The predicates of θ are $on, off, p_1, p_2, p_3, r$. Here, on and off are involved in the constraint $\forall X{:}s,\ on(X) \oplus off(X)$, while p_1, p_2, p_3 are involved in the constraint $\forall X{:}s,\ p_1(X) \oplus p_2(X) \oplus p_3(X)$, and r is involved in the constraint $\forall X{:}s,\ \exists! Y{:}s\, r(X, Y)$.
- Let $\tau(\theta) = (A_1, A_2, A_3, A_4, \ldots)$. It is straightforward to check that A_1, A_2, and A_3 satisfy each constraint in C. Observe that A_4 repeats A_1, and the dynamics are Markov, so all subsequent states are copies of A_1, A_2, and A_3. Hence, every state in the infinite sequence $\tau(\theta)$ satisfies each constraint.
- Temporal unity is automatically satisfied by the definition of the trace $\tau(\theta)$ in Definition 3. ◁

Now we are ready to define the central notion of "making sense" of a sequence.

Definition 6. A theory θ **makes sense** of a sensory sequence S if θ explains S, i.e. $S \sqsubseteq \tau(\theta)$, and θ satisfies the four conditions of unity of Definition 5.

Example 6. The theory θ of Example 2 explains the sensory sequence S of Example 1, since $S \sqsubseteq \tau(\theta)$ (Example 4) and θ satisfies the four conditions of unity (Example 5). ◁

Definition 7. The input to an apperception task is a triple (S, ϕ, C) consisting of a sensory sequence S, a suitable type signature ϕ, and a set C of (well-typed) constraints such that (i) each predicate in S appears in some constraint in C and (ii) S can be extended to satisfy C: there exists a sequence S' covering S such that each state in S' satisfies each constraint in C.

Given such an input triple (S, ϕ, C), the **simple apperception task** is to find the lowest cost theory $\theta = (\phi', I, R, C')$ such that ϕ' extends ϕ, $C' \supseteq C$, and θ makes sense of S. (Note that C' extends C, adding additional constraints between invented predicates). Here, the **cost** of θ is just the total number of ground atoms in I plus the total number of unground atoms in the rules of R.

Example 7. Recall that theory θ of Example 2 makes sense of the sequence from Example 1. Now consider an alternative theory based on the type signature $\phi_2 = (T_2, O_2, P_2, V_2)$, where $T_2 = \{s\}$, $O_2 = \{a{:}s, b{:}s, c{:}s\}$, $P_2 = \{on(s), off(s), right(s, s)\}$, and $V_2 = \{X{:}s, Y{:}s\}$. Let the alternative theory $\theta_2 = (\phi_2, I_2, R_2, C_2)$ where

$$I_2 = \left\{ \begin{array}{lll} on(a) & on(b) & off(c) \\ right(a, b) & right(b, c) & right(c, a) \end{array} \right\}$$

[4] Strictly speaking, since traces are composed of purely *syntactic* entities, we really mean *constants* not objects, throughout.

[5] See [B203]. See also the *Third Analogy* [A211-215/B256-262].

[6] See [A103-11]. See also: "What the form of disjunctive judgement may do is contribute to the acts of forming categorical and hypothetical judgements the perspective of their possible *systematic unity*", [59, p.105].

[7] See the *schema of community* [A144/B183-4].

[8] See the *schema of causality* [A144/B183].

$$R_2 = \left\{ \begin{array}{l} right(X,Y) \wedge off(X) \supset off(Y) \\ right(X,Y) \wedge on(X) \supset on(Y) \end{array} \right\}$$

$$C_2 = \left\{ \begin{array}{l} \forall X{:}sensor,\ on(X) \oplus off(X) \\ \forall X{:}sensor,\ \exists!Y{:}sensor,\ right(X,Y) \end{array} \right\}$$

Now θ_2 also makes sense of the sensory sequence S from Example 1. But while θ used three invented predicates (p_1, p_2, and p_3), θ_2 makes use of an invented *object*, c, postulated to explain the sensory sequence more concisely. Note that while θ has cost 16, θ_2 has cost 12. ◁

3.2. The Apperception Engine

Next, we describe how our system, the Apperception Engine, solves apperception tasks. Given as input an apperception task (S, ϕ, C), the engine searches for a type signature ϕ' and a theory $\theta = (\phi', I, R, C')$ such that ϕ' extends ϕ, constraints $C' \supseteq C$ and θ is a unified interpretation of S. We use the type signature to constrain the set of available theories, as type-checking has been shown to drastically reduce the search space in program synthesis tasks [60].

A **template** is a structure for circumscribing a large but finite set of theories. It is a type signature together with constants that bound the complexity of the rules in the theory.

The Apperception Engine enumerates templates of increasing complexity. For each template, it finds the theory with the lowest cost that satisfies the conditions of unity. If it finds such a theory, it stores it. When it has run out of processing time, it returns the lowest cost theory it has found from all the templates it has considered.

The most complex part of the process is finding, for a given template, a lowest cost theory that both explains the given sensory sequence and also satisfies the four unity conditions. In order to jointly abduce a set I (of initial conditions) and induce sets R and C (of rules and constraints), we implement a $Datalog^{\supset}$ interpreter in Answer Set Programming (ASP). This interpreter takes a set I of atoms (reified as a set of ground ASP terms) and sets R and C of rules and constraints (also reified as terms), and computes the trace of the theory $\tau(\theta) = (S_1, S_2, \ldots)$ up to a finite time limit.

We implement the conditions of unity as ASP constraints, and we implement the cost minimization as an ASP program that counts the number of atoms in each rule and in each initialisation atom in I, and uses an ASP weak constraint [61] to minimize this total. Then we generate ASP programs representing the sequence S, the initial conditions, the rules and constraints. We combine the ASP programs together and ask the ASP solver (`clingo` [62]) to find a lowest cost solution. (There may be multiple solutions that have equally lowest cost; the ASP solver chooses one of them). We extract a readable interpretation θ from the ground atoms of the answer set. We do not go into more detail, for reasons of space.[9] The source code is available at `https://github.com/RichardEvans/apperception`.

4. Raw apperception

In this section, we extend the Apperception Engine so that it can handle raw unprocessed sensory input. This is the central contribution of the paper. First, we shall define what it means to make sense of a disjunctive sensory sequence. Second, we shall show how to use a neural network to transform raw unprocessed sensory input into a disjunctive sensory sequence.

4.1. Making sense of disjunctive symbolic input

In this section, we extend the Apperception Engine to handle disjunctive sensory input.

Definition 8. A **disjunctive input sequence** is a sequence of sets of disjunctions of ground atoms.

A disjunctive input sequence generalises the input sequence of Section 3.1 to handle uncertainty. Now if we are unsure if a sensor a satisfies predicate p or predicate q, we can express our uncertainty as $p(a) \vee q(a)$.

Example 8. Consider, for example, the following sequence $D_{1:10}$. This is a disjunctive variant of the unambiguous sequence from Example 1. Here there are two sensors a and b, and each sensor can be *on* or *off*.

$$\begin{array}{lll} D_1 = \{\} & D_2 = \{off(a), on(b)\} & D_3 = \{off(a) \vee on(a), off(b)\} \\ D_4 = \{on(a), on(b)\} & D_5 = \{off(a) \vee on(a), on(b)\} & D_6 = \{on(a), off(b)\} \\ D_7 = \{on(a), on(b)\} & D_8 = \{off(a), on(b)\} & D_9 = \{off(a) \vee on(a)\} \\ D_{10} = \{\} & & \end{array}$$

$D_{1:10}$ is weaker than $S_{1:10}$ from Example 1, as each D_i is entailed by S_i, but not vice-versa. Observe that D_3 is unsure whether a is *on* or *off*, while in S_3 a is definitely *on*. ◁

[9] For more information, see [1].

Recall that the $\sqsubseteq$ relation describes when one (finite) sequence is covered by another. We extend the $\sqsubseteq$ relation to handle disjunctive input sequences in the first argument.

Definition 9. Let $D = (D_1, \ldots, D_T)$ be a (finite) disjunctive input sequence and S be an input sequence. $D \sqsubseteq S$ if S contains a finite subsequence $(S_1, \ldots, S_T)$ such that $S_i \models D_i$ for all $i = 1...T$.

Example 9. The theory θ of Example 2 explains the disjunctive sequence D of Example 8, since the trace $\tau(\theta)$ (as shown in Example 3) covers D. For example, $D_3 = \{off(a) \vee on(a), off(b)\}$, the trace of θ at time-step 3 is $A_3 = \{on(a), off(b), p_1(a), p_3(b), r(a,b), r(b,a)\}$, and $A_3 \models D$. $\triangleleft$

The disjunctive apperception task generalises the simple apperception task of Definition 7 to disjunctive input sequences.

Definition 10. The input to a disjunctive apperception task is a triple (D, ϕ, C) consisting of a disjunctive input sequence D, a suitable type signature ϕ, and a set C of (well-typed) constraints such that (i) for each disjunction featuring predicates $p_1, \ldots, p_n$ there exists a constraint in C featuring each of $p_1, \ldots, p_n$. (ii) D can be extended to satisfy C.

Given such an input triple (D, ϕ, C), the **disjunctive apperception task** is to find a lowest cost theory $\theta = (\phi', I, R, C')$ such that ϕ' extends ϕ, $C' \supseteq C$, $D \sqsubseteq \tau(\theta)$, and θ satisfies the four unity conditions of Definition 5. (Note that the unity conditions carry over entirely unchanged when we move from a simple to a disjunctive apperception task).

4.2. Making sense of raw input

The reason for introducing the disjunctive apperception task is as a stepping stone to the real task of interest: making sense of sequences of raw uninterpreted sensory input.

Definition 11. Let $\mathbf{R}$ be the set of all possible raw inputs. A **raw input sequence** of length T is a sequence $(\mathbf{r}_1, \ldots, \mathbf{r}_T)$ in $\mathbf{R}^T$. Here $\mathbf{R}$ is the set of all possible raw inputs for a single time step, e.g. the set of all 20×20 binary pixel arrays.

A raw apperception framework uses a neural network $\pi_{\mathbf{w}}$, parameterised by weights $\mathbf{w}$, to map subregions of each $\mathbf{r}_i$ into subsets of classes $\{1, \ldots, n\}$. Then the results of the neural network are transformed into a disjunction of ground atoms, transforming the raw input sequence into a disjunctive input sequence.

Definition 12. A **raw apperception framework** is a tuple $(n, \pi_{\mathbf{w}}, \Delta, \phi, C)$, where:

- n is the number of classes that the perceptual classifier $\pi_{\mathbf{w}}$ uses
- $\pi_{\mathbf{w}}$ is a neural network, a multilabel classifier mapping subregions of $\mathbf{r}_i$ to subsets of $\{1, \ldots, n\}$; π is parameterised by weight vector $\mathbf{w}$
- Δ is a "disjunctifier" that converts the results of the neural network $\pi_{\mathbf{w}}$ into a set of disjunctions; it takes as input the result of repeatedly applying[10] $\pi_{\mathbf{w}}$ to the N subregions $\{\mathbf{p}_i^1, \ldots, \mathbf{p}_i^N\}$ of $\mathbf{r}_i$, and produces as output a set of N disjunctions of ground atoms
- ϕ is a type signature
- C is a set of constraints

The input to a raw apperception task is a raw sequence together with a raw apperception framework. Given sequence $\mathbf{r} = (\mathbf{r}_1, \ldots, \mathbf{r}_T)$ and framework $(n, \pi_{\mathbf{w}}, \Delta, \phi, C)$, the **raw apperception task**[11] is to find the lowest cost weights w and theory θ such that θ is a solution to the disjunctive apperception task $((D_1, \ldots, D_T), \phi, C)$ where $D_i = \Delta(\pi_{\mathbf{w}}(\mathbf{p}_i^1), \ldots, \pi_{\mathbf{w}}(\mathbf{p}_i^N))$.

The best $(\theta, \mathbf{w})$ pair is:

$$\arg\max_{\theta, \mathbf{w}} \log p(\theta) + \sum_{i=1}^{T} \sum_{j=1}^{N} \log \frac{1}{|\{\mathbf{p} \in \mathbf{P} \mid k_i^j \in \pi_{\mathbf{w}}(\mathbf{p})\}|}$$

where $\mathbf{P} = \{\mathbf{p}_i^j \mid i = 1..T, j = 1..N\}$ is the union of the subregions appearing at each position j and at each time-step i, and k_i^j is the atom in the i'th state of $\tau(\theta)$ that represents the class of the object in region j.

The intuition here is that $p(\theta) \propto 2^{-cost(\theta)}$ represents the prior probability of the theory θ, while the second term penalises the neural network $\pi_{\mathbf{w}}$ for mapping many elements to the same class. In other words, it prefers highly selective

[10] This repeated application of the same neural net to each subregion is inspired by the convolutional neural network [63].

[11] For concrete examples of this rather abstract definition, see Sections 5.2.2 and 5.3.3.

classes, minimising the number of elements that are assigned by $\pi_{\mathbf{w}}$ to the same class. This particular optimisation can be justified using Bayes theorem, as we now show.

4.3. Finding the most probable interpretation

We are given a raw sequence $\mathbf{r} = (\mathbf{r}_1, \ldots, \mathbf{r}_T)$ together with a raw framework $(n, \pi_{\mathbf{w}}, \Delta, \phi, C)$, where neural network π is parameterised by weight vector $\mathbf{w}$. We want to find the most probable theory θ and weights $\mathbf{w}$ given our raw input sequence $\mathbf{r}$:

$$\arg\max_{\theta,\,\mathbf{w}} \; p\,(\theta, \mathbf{w} \mid \mathbf{r}) \tag{1}$$

By Bayes rule, this is equivalent to

$$\arg\max_{\theta,\,\mathbf{w}} \; \frac{p\,(\mathbf{r} \mid \theta, \mathbf{w}) \cdot p(\theta, \mathbf{w})}{p\,(\mathbf{r})} \tag{2}$$

Since the denominator does not depend on θ or $\mathbf{w}$, this is equivalent to:

$$\arg\max_{\theta,\,\mathbf{w}} \; p\,(\mathbf{r} \mid \theta, \mathbf{w}) \cdot p(\theta, \mathbf{w}) \tag{3}$$

Assuming the priors of θ and $\mathbf{w}$ are independent, $p(\theta, \mathbf{w})$ can be decomposed to get:

$$\arg\max_{\theta,\,\mathbf{w}} \; p\,(\mathbf{r} \mid \theta, \mathbf{w}) \cdot p(\theta) \cdot p(\mathbf{w}) \tag{4}$$

Let us assume the prior $p(\mathbf{w})$ on the weight vector is uniform, so can be dropped:

$$\arg\max_{\theta,\,\mathbf{w}} \; p\,(\mathbf{r} \mid \theta, \mathbf{w}) \cdot p(\theta) \tag{5}$$

Let the trace $\tau(\theta) = (A_1, A_2, \ldots, A_T)$. Since $\tau(\theta)$ is deterministically generated from the theory θ, $p(\mathbf{r} \mid \theta, \mathbf{w}) = p(\mathbf{r}, \tau(\theta) \mid \theta, \mathbf{w})$. As the dynamics are Markov, $p(\tau(\theta) \mid \theta, \mathbf{w}) = p(A_1 \mid \theta, \mathbf{w}) \cdot \prod_{i=2}^{T} p(A_i \mid A_{i-1}, \theta, \mathbf{w})$. Together with each $\mathbf{r}_i$ being conditionally independent of $\mathbf{r}_{i-1}$ given θ, we get:

$$\begin{aligned}
p(\mathbf{r} \mid \theta, \mathbf{w}) &= p(\mathbf{r}, \tau(\theta) \mid \theta, \mathbf{w}) \\
&= p(A_1 \mid \theta, \mathbf{w}) \cdot p(\mathbf{r}_1 \mid A_1, \theta, \mathbf{w}) \cdot \prod_{i=2}^{T} p(A_i \mid A_{i-1}, \theta, \mathbf{w}) \cdot p(\mathbf{r}_i \mid A_i, \theta, \mathbf{w}) \\
&= p(A_1 \mid \theta, \mathbf{w}) \cdot \prod_{i=2}^{T} p(A_i \mid A_{i-1}, \theta, \mathbf{w}) \cdot \prod_{i=1}^{T} p(\mathbf{r}_i \mid A_i, \theta, \mathbf{w}) \\
&= p(\tau(\theta) \mid \theta, \mathbf{w}) \cdot \prod_{i=1}^{T} p(\mathbf{r}_i \mid A_i, \theta, \mathbf{w})
\end{aligned}$$

This enables us to rewrite Formula (5) as:

$$\arg\max_{\theta,\,\mathbf{w}} \; p(\theta) \cdot p(\tau(\theta) \mid \theta, \mathbf{w}) \cdot \prod_{i=1}^{T} p(\mathbf{r}_i \mid A_i, \theta, \mathbf{w}) \tag{6}$$

Since the latent symbolic trace $\tau(\theta)$ is deterministically generated from the theory θ, $p(\tau(\theta) \mid \theta, \mathbf{w}) = 1$, and we can remove this term to get:

$$\arg\max_{\theta,\,\mathbf{w}} \; p(\theta) \cdot \prod_{i=1}^{T} p(\mathbf{r}_i \mid A_i, \theta, \mathbf{w}) \tag{7}$$

Since $\mathbf{r}_i$ is conditionally independent of θ given A_i and $\mathbf{w}$, we can rewrite to:

$$\arg\max_{\theta,\,\mathbf{w}} \; p(\theta) \cdot \prod_{i=1}^{T} p(\mathbf{r}_i \mid A_i, \mathbf{w}) \tag{8}$$

Assuming raw data $\mathbf{r}_i$ can be decomposed into disjoint non-overlapping subregions $\mathbf{p}_i^1, \ldots, \mathbf{p}_i^N$, we can rewrite to:

$$\arg\max_{\theta,\,\mathbf{w}} \; p(\theta) \cdot \prod_{i=1}^{T} \prod_{j=1}^{N} p(\mathbf{p}_i^j \mid A_i, \mathbf{w}) \tag{9}$$

Assume that subregion $\mathbf{p}_i^j$ is stochastically sampled conditioned on the latent k_i^j in A_i. Here, k_i^j is an atom featuring a class label from $\{1, \ldots, n\}$ representing the type of object in region j at time i. Assume the raw subregions are sampled uniformly. Then the probability of the particular subregion $\mathbf{p}_i^j$ is 1 divided by the number of subregions that are mapped to class k_i^j:

$$p(\mathbf{p}_i^j \mid A_i, \mathbf{w}) = p(\mathbf{p}_i^j \mid k_i^j, \mathbf{w}) = \frac{1}{|\{\mathbf{p} \in \mathbf{P} \mid k_i^j \in \pi_w(\mathbf{p})\}|} \tag{10}$$

where $\mathbf{P} = \{\mathbf{p}_i^j \mid i = 1..T, j = 1..N\}$ is the union of the subregions appearing at each position j and at each time-step i.

Substituting Equation (10) in Formula (9) gives:

$$\arg\max_{\theta,\,\mathbf{w}} \; p(\theta) \cdot \prod_{i=1}^{T} \prod_{j=1}^{N} \frac{1}{|\{\mathbf{p} \in \mathbf{P} \mid k_i^j \in \pi_{\mathbf{w}}(\mathbf{p})\}|} \tag{11}$$

Since $log(.)$ is monotonic, we can rewrite to:

$$\arg\max_{\theta,\,\mathbf{w}} \; \log p(\theta) + \sum_{i=1}^{T} \sum_{j=1}^{N} \log \frac{1}{|\{\mathbf{p} \in \mathbf{P} \mid k_i^j \in \pi_{\mathbf{w}}(\mathbf{p})\}|} \tag{12}$$

The left summand of Equation (12) represents the prior probability of theory θ. We assume $p(\theta) \propto 2^{-len(\theta)}$. The right summand is maximum when the fewest raw images $\mathbf{p}$ are assigned to each class k_i^j. Optimizing this value means preferring lots of highly selective classes over high entropy classes. Other approaches which also seek to minimize entropy include [64–66].

4.4. Applying the Apperception Engine to raw input

In the experiments that follow, we use a binary neural network [67–71] for our parameterised perceptual classifier. Binary neural networks (BNNs) are increasingly popular because they are more efficient (both in memory and processing) than standard artificial neural networks. But our interest in BNNs is not so much in their resource efficiency as in their *discreteness*.

In the BNNs that we use [70], the node activations and weights are all binary values in $\{0, 1\}$. If a node has n binary inputs $x_1, \ldots, x_n$, with associated binary weights $w_1, \ldots, w_n$, the node is activated if the total sum of inputs *xnor*-ed with their weights is greater than or equal to half the number of inputs. In other words, the node is activated if

$$\sum_{i=1}^{n} \mathbb{1}[x_i = w_i] \geq \left\lceil \frac{n}{2} \right\rceil$$

Because the activations and weights are binary, the state of the network can be represented by a set of atoms, and the dynamics of the network can be defined as a logic program.[12] This means we can combine the low-level perception task (of mapping raw data to concepts) and the high-level apperception task (of combining concepts into rules) *into a single logic program* in ASP, and solve both simultaneously using SAT.

4.5. Example

Suppose we start with the simple sequence *abababababab* ... We shall re-present this alphabetic sequence as a sequence of bit vectors, letting a be represented by either 00 or 01, and b be represented by either 10 or 11. Using this encoding, one possible representation of *ababab* is 00, 10, 01, 11, 01, 10.

When this sequence of bit vectors is given to the Apperception Engine, the engine needs to jointly learn a perceptual classifier (that groups the bit vectors into classes, thereby transforming raw data into ground atoms), and a theory (that explains how the ground atoms change over time). For this simple problem, a suitable raw apperception framework is:

[12] See the Supplementary Material for the details of how the binary neural network is implemented in ASP.

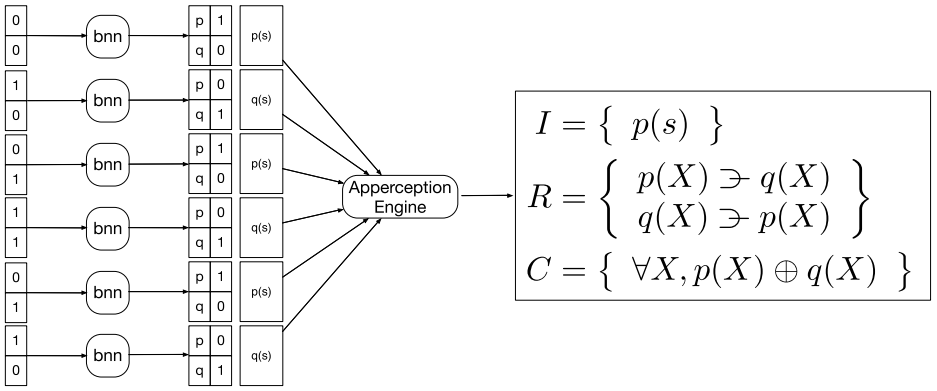


Fig. 1. Interpreting the simple example of Section 4.5. The binary neural network maps each bit vector to a distribution over the classes p and q. The disjunctifier turns the network's output into a ground atom, $p(s)$ or $q(s)$. The Apperception Engine takes the sequence of sets of ground atoms, and produces a theory that explains how the atoms change over time.

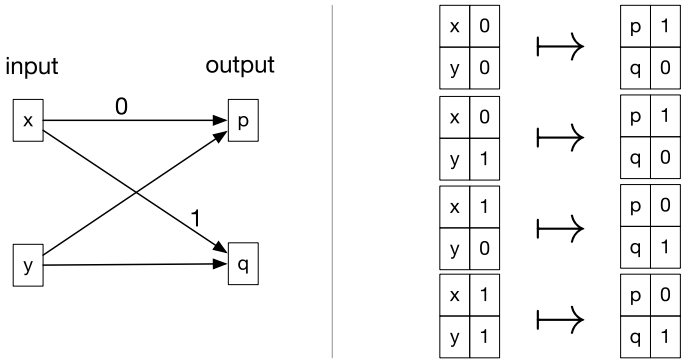


Fig. 2. A simple binary neural network with no hidden layers that solves the simple worked example of Section 4.5. The weights of the network are shown on the left, and the mapping produced by the network is shown on the right. Note that two of the weights are unlabelled, as these weights can be either 0 or 1, without affecting the result.

- $n = 2$, since there are two classes. We shall call these classes p and q. Note that these classes p and q are distinct from the original labels a and b. Here, p and q are the predicates that the Apperception Engine uses to make sense of the raw data, as opposed to the original class labels a and b, which are not available to the engine.
- $\pi_{\mathbf{w}}$ is a binary neural network mapping bit vectors (of length 2) to vectors (also of length 2, but this time representing the distribution over the classes p and q). In this simple example, the neural network does not need any hidden layers. (For examples that require networks with hidden layers, see Sections 5.1, 5.2, and 5.3).
- Δ is a "disjunctifier" that converts the results of the neural network $\pi_{\mathbf{w}}$ into a set of disjunctions. In this case, if the network outputs $p = 1, q = 0$ then the disjunctifier produces $p(s)$; otherwise, it produces $q(s)$. (Note there are no ambiguities in the input, and hence no disjunctions needed for this simple example. For examples that use disjunctions, see Sections 5.1, 5.2, and 5.3.)
- ϕ is a type signature with one object s, and two unary predicates p and q.
- $C = \{\forall X, p(X) \oplus q(X)\}$

The network is parameterized by weights $\mathbf{w}$, and the theory is represented by θ. The best $\theta, \mathbf{w}$ pair found by the Apperception Engine is shown in Figs. 1 and 2. The theory θ states that $p(s)$ is true initially, and then s alternates between p and q according to the rules $p(X) \supset\!\!\!- q(X)$ and $q(X) \supset\!\!\!- p(X)$. This theory produces the sequence $\{p(s)\}, \{q(s)\}, \{p(s)\}, \{q(s)\}, \{p(s)\}, \{q(s)\}, \ldots$ Thus, the engine has recaptured the original sequence *ababab*, but with a replaced by p, and b replaced by q.

The log probability of this interpretation, according to Equation (12), depends on two factors: the size of the theory and the number of bit vectors that are assigned to the same class. In this case, the size of the theory is 5 (as there is one ground atom in the initial conditions, and four atoms in the rules). Since the network assigns two bit vectors to class p and two to class q

$$\sum_{i=1}^{T}\sum_{j=1}^{N}\log_2\frac{1}{|\{\mathbf{p}\in\mathbf{P}\mid k_i^j\in\pi_{\mathbf{w}}(\mathbf{p})\}|}=6*\log_2(\frac{1}{2})=-6$$

Thus, the total log probability for this theory is $-5+-6=-11$.

Consider, by way of contrast, an alternative degenerate interpretation that maps all four bit vectors to p and none to q. The theory for this degenerate interpretation is just:

$$I=\{p(s)\}$$
$$R=\{\}$$
$$C=\{\forall X, p(X)\oplus q(X)\}$$

This interpretation is the great leveller, blurring all distinctions. Since it sees all bit vectors as essentially the same, it does not need to postulate rules to explain how things change, as it doesn't perceive any change.

The degenerate interpretation has a shorter theory of size 1, but the second summand in Equation (12) fares worse, since four vectors are assigned the same class:

$$\sum_{i=1}^{T}\sum_{j=1}^{N}\log_2\frac{1}{|\{\mathbf{p}\in\mathbf{P}\mid k_i^j\in\pi_{\mathbf{w}}(\mathbf{p})\}|}=6*\log_2(\frac{1}{4})=-12$$

The total log probability for the degenerate theory is $-1+-12=-13$.

Thus, the first theory gets a better score (-11) than the degenerate theory (-13). Further, as the length of the sequence increases, the difference between the two theories increases linearly. Thus, Equation (12) prefers the interpretation that makes more fine-grained discriminations, even if the theory needed to explain how the fine-grained classifications change is more complex.

This is, of course, a simplified example for expository purposes. For a more complex example along the same lines, where some of the bit vectors are ambiguous, and where the system is not provided with the length of the bit vectors, see Section 5.3.

5. Experiments

5.1. Seek Whence with noisy images

The *Seek Whence* dataset is a set of challenging sequence induction problems designed by Douglas Hofstadter [8]. In each problem, you are given a sequence of symbols, and have to predict the next symbol in the sequence. For example:

- **2, 2, 2, 3, 3, 2, 2, 2, 3, 3, 2, 2, 2, 3, 3, ...**
- **1, 6, 2, 6, 6, 3, 6, 6, 6, 4, 6, 6, ...**
- **1, 0, 1, 1, 1, 1, 1, 2, 1, 1, 3, 1, 1, 4, 1, ...**

Hofstadter called the third sequence the "theme song" of the *Seek Whence* project because of its difficulty. There is a "perceptual mirage" in the sequence because of the sub-sequence of five 1's in a row that makes it hard to see the intended pattern: $(1, x, 1)^*$ for increasing x.

5.1.1. The data

In Hofstadter's original dataset, the sequences are lists of discrete symbols. In our modified dataset, we replaced each discrete symbol with a corresponding MNIST image.

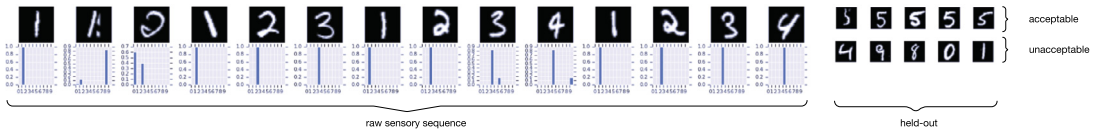
To make it more interesting (and harder), we deliberately chose particularly ambiguous images. Consider Fig. 3a. Here, the leftmost image could be a 0 or a 2, while the next could be a 5 or possibly a 6. Of course, we humans are unphased by these ambiguities because the low Kolmogorov complexity [72] of the high-level symbolic sequence helps us to resolve the ambiguities in the low-level perceptual input. We would like our machines to do the same.

For each sequence, the held-out data used for evaluation is a set of acceptable images, and a set of unacceptable images, for the final held-out time step. See Fig. 3. We provide a slice of the sequence as input, and use a held-out time step for evaluation. If the correct symbol at the held-out time step is s, then we sample a set of unambiguous images representing s for our set of acceptable next images, and we sample a set of unambiguous images representing symbols other than s for our set of unacceptable images.

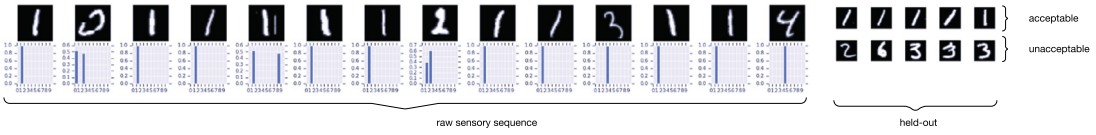
5.1.2. The model

In this experiment, we combined the Apperception Engine with a three-layer perceptron with dropout that had been *pre-trained* to classify images into ten classes representing the digits $0-9$. For each image, the network produced a probability distribution over the ten classes.

(a) The sequence $0, 5, 1, 5, 2, 5, 3, 5, 4, 5, 5, \ldots$ with held-out value 5



(b) The sequence $1, 1, 2, 1, 2, 3, 1, 2, 3, 4, 1, 2, 3, 4, \ldots$ with held-out value 5



(c) The sequence $1, 0, 1, 1, 1, 1, 1, 2, 1, 1, 3, 1, 1, 4, \ldots$ with held-out value 1

Fig. 3. Three Seek-Whence tasks using MNIST images. The left section of each diagram shows the given sensory sequence, while the right section shows the held-out time step. At the final held-out time step, there is a set of acceptable images, and a set of unacceptable images. Note that the acceptable and unacceptable images are used only for evaluation, not for training.

We chose a threshold (0.1), and stipulated that if the probability of a particular digit exceeded the threshold, then the image possibly represents that digit. According to this threshold, some of the images (the first, second, and sixth) of Fig. 4a are ambiguous, while others (the third, fourth, and fifth) are not.

Our pre-trained neural network MNIST classifier has effectively turned the raw apperception task into a disjunctive apperception task. Once the input has been transformed into a sequence of disjunctions, we apply the Apperception Engine to resolve the disjunctions and find a unified theory that explains the sequence.

In terms of the formalism of Section 4.2, the raw input $r = (r_1, \ldots, r_T)$ is a sequence of MNIST images from $[0, 1]^{28 \times 28}$. The framework $(n, \pi_{\mathbf{w}}, \Delta, \phi, C)$ consists of:

- $n = 10$, representing the digits '0'–'9'
- $\pi_{\mathbf{w}}$, a pre-trained MNIST classifier with frozen weights $\mathbf{w}$
- Δ takes the output of the pre-trained MNIST classifier, and produces a single disjunction of all the classes for which the network outputs a probability that is above the threshold
- A type signature $\phi = (T, O, P, V)$ consisting of two types: sensors and integers, and two predicates: $value(sensor, int)$ representing the numeric value of a sensor, and $succ(int, int)$ representing the successor relation
- C contains one constraint that insists that every sensor always has exactly one numeric value

The input type signature ϕ and initial constraints C are:

$$\phi = \begin{cases} T = \{sensor, int\} \\ O = \{s{:}sensor, 0{:}int, 1{:}int, 2{:}int, \ldots\} \\ P = \{value(sensor, int), succ(int, int)\} \\ V = \{X{:}sensor, N{:}int\} \end{cases}$$

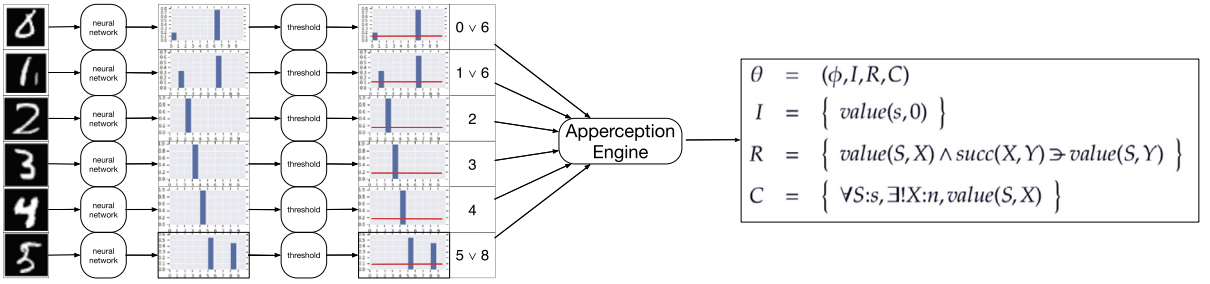
$$C = \left\{ \forall X{:}sensor,\ \exists! N{:}int\ value(X, N) \right\}$$

We ran the Apperception Engine on a standard Unix desktop, allowing 4 hours for each sequence. Fig. 4 shows some results.

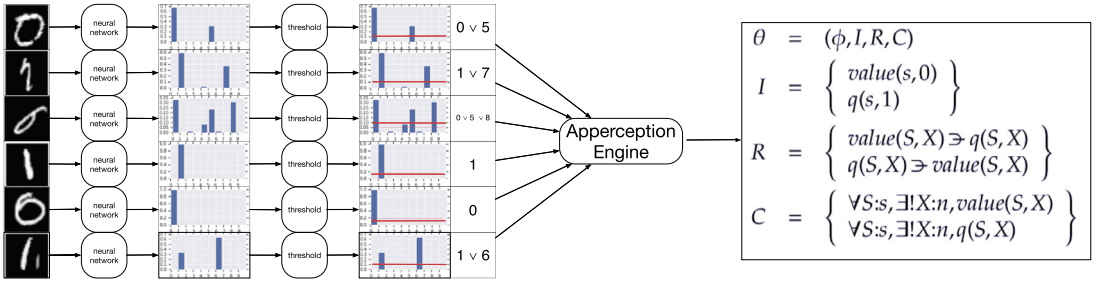
5.1.3. Understanding the interpretations

Fig. 5a shows the unified theory found for the "theme song"' sequence, while Fig. 5b shows the interpretation in detail.

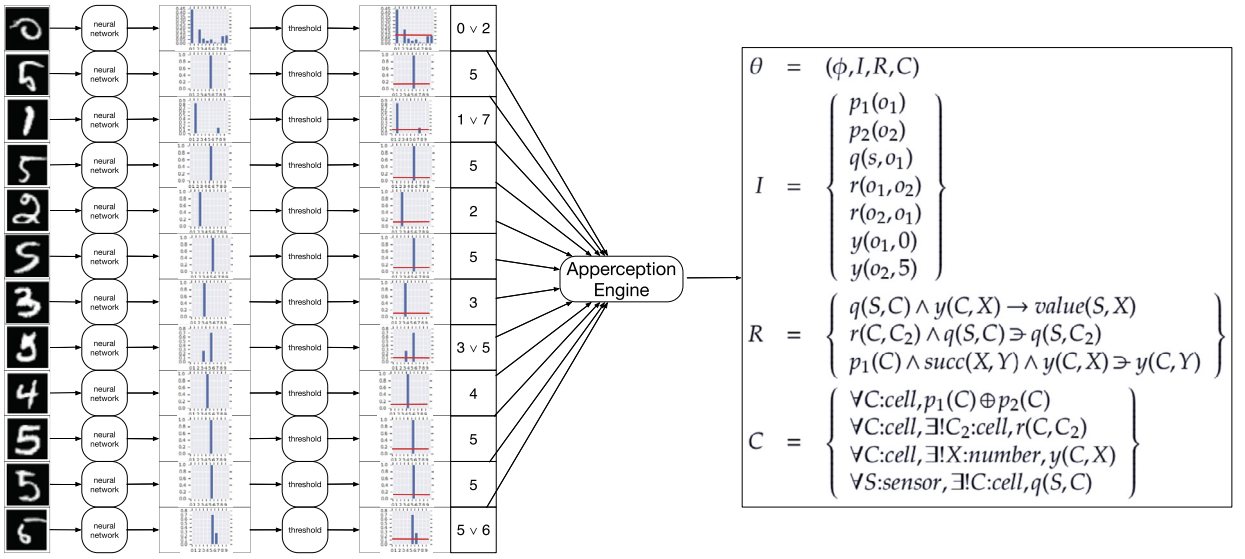
Let us use try to understand, in detail, why the Apperception Engine believes the bottom MNIST image in Fig. 5a (at time step 15) should be interpreted as a '1', rather than a '6'. According to the neural network, the image could either be classified as a '1' or a '6'. In fact, the network thinks it is rather more likely to be a '6'. Nevertheless, the overall assessment of the Apperception Engine is that the image represents a '1'. Why is this?



(a) Interpreting the sequence **0, 1, 2, 3, 4, 5, ...**



(b) Interpreting the sequence **0, 1, 0, 1, 0, 1, ...**

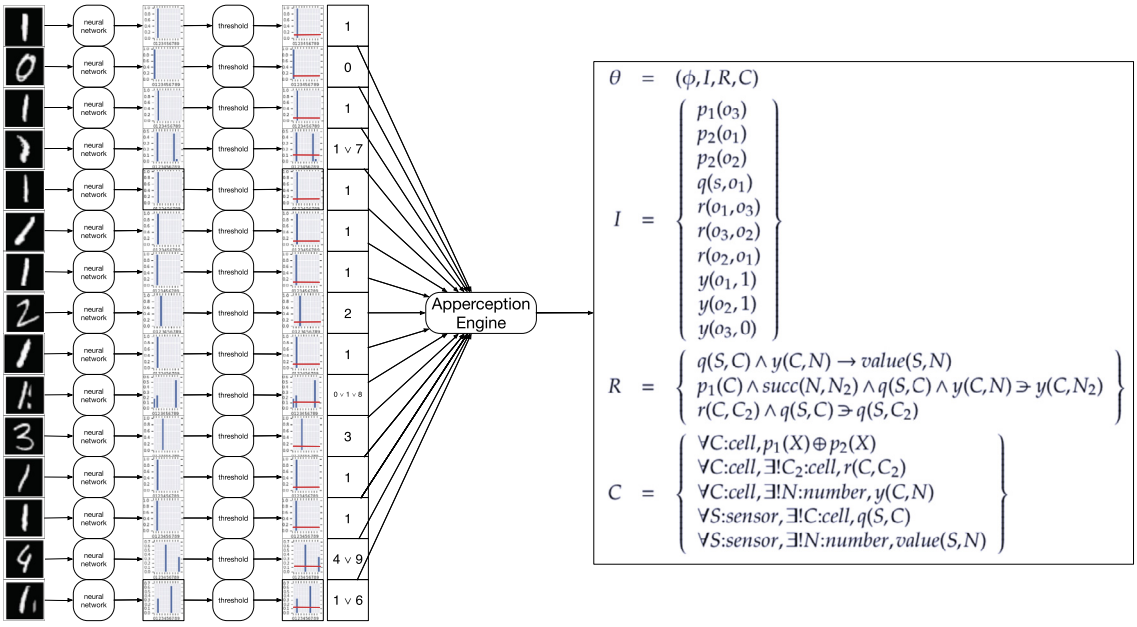


(c) Interpreting the sequence **0, 5, 1, 5, 2, 5, 3, 5, 4, 5, 5, 5, ...**

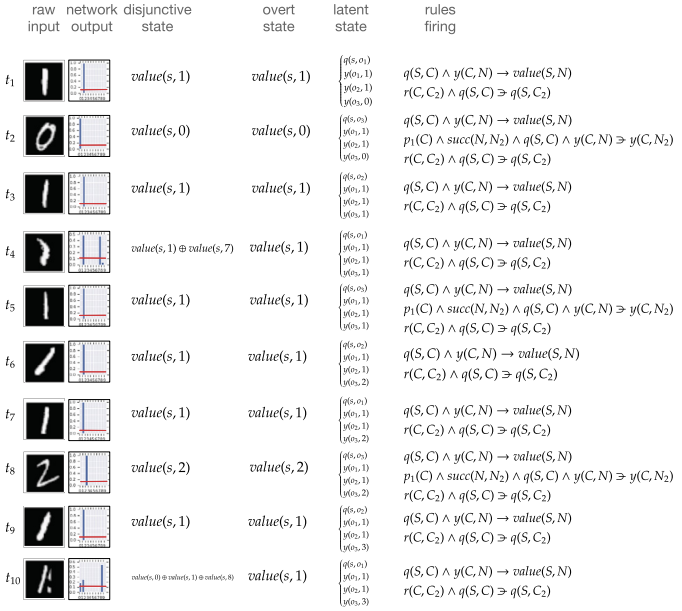
Fig. 4. Interpreting *Seek Whence* sequences from raw images. Each MNIST image is passed to a pre-trained neural network classifier that emits a distribution over the digits 0–9. A threshold of 0.1 is applied to the probability distribution, generating a disjunction over the values of the sensor. For example, the disjunction $0 \vee 6$ is short-hand for $value(s, 0) \vee value(s, 6)$. The sequence of disjunctions is passed to the Apperception Engine, which produces a unified theory that resolves the disjunctions and explains the sequence. The predicates *value* and *succ* are provided to the system, while all other predicates are invented.

At a high level, the explanation for this interpretation is that the whole sequence exhibits a particular regularity described by a single general pattern with low Kolmogorov complexity, and that, given this overall structure, the best way to read the final symbol is as a '1' rather than as a '6'.

More specifically, Fig. 5a describes the following simple process: the sensor is a read-write head that moves between three cells, in a cycle. These cells are o_1, o_2, and o_3, which are placed in the order: o_1, o_3, o_2. Initially, cells o_1 and o_2 have value 1, while cell o_3 has value 0. The head reads the value of the current cell, and writes it onto an output tape. When the



(a) Generating a theory to make sense of the sequence.



(b) The left column shows the raw MNIST images, while the second column shows the output layer of the pre-trained neural network when given as input the MNIST image on the left. The third column shows the disjunctive sensor state, and the fourth column shows the overt state after the disjunctions have been resolved. The fifth column shows the latent state imputed by the Apperception Engine. The sixth column shows the rules that fire at each time step.

Fig. 5. Interpreting the sequence **1, 0, 1, 1, 1, 1, 1, 2, 1, 1, 3, 1, 1, 4, 1, ...**

head moves over the middle cell (o_3), it increments the value of that cell. When it moves over either of the other two cells, its value remains unchanged.

Note that the only predicates that are given to the Apperception Engine are *value* (provided by the neural network) and the *succ* relation (provided as prior knowledge). Every other predicate is *invented*, its meaning entirely determined by its inferential role in the rules and constraints of the theory in which it is embedded.

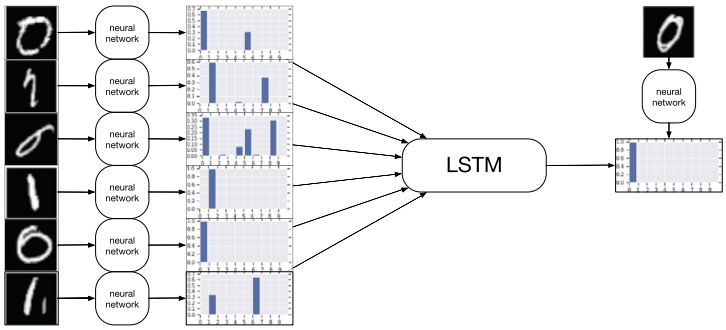


Fig. 6. Neural baseline for the *Seek Whence* task, utilising a pre-trained digit-recognition network (denoted with 'neural network') and a vanilla LSTM predicting the label of the following digit.

Now, at this particular moment (time step 15 of Fig. 5a), the read-write head is on the cell o_2. This cell has value 1. So the sensor must be reading a '1' rather than a '6'. There is no comparably simple theory that makes sense of the data which resolves the final image to a '6'. So, given the plausibility (simplicity) of the whole theory that explains all the data, we are compelled to interpret the image as a '1'.

5.1.4. The baselines

Given the raw input to the Apperception Engine, neural models are the most appropriate baselines for comparison. However, the modes of operation of these two systems differ greatly. The Apperception Engine outputs a compact theory which aims to fully explain the sequence, making these rules useful for prediction, imputation, retrodiction as well as explanation. With a neural model, it is hard to induce a verifiably correct and explainable theory. However, we can compare it to the Apperception Engine in terms of their predictive capabilities.

In order to make a fair "like-for-like" comparison between the neural baseline and the Apperception Engine, we impose the following requirements on the baseline:

- It must learn using *self-supervision*, predicting future time-steps from earlier time-steps.
- It must be able to work with *variable-length data*, since the different trajectories are of different lengths.
- It must be able to handle *noisy or ambiguous data*, since the raw data in all three of our experiments is noisy and ambiguous.

These requirements imply we should use a model from the Recurrent Neural Network family of models, of which we opt for the Long Short-Term Memory (LSTM) [73], given its stable and favourable performance.

However, given that the Apperception Engine is able to induce a meaningful theory from a single sequence, we also want to compare how well the baseline LSTM fares with:

- *learning from small amount of data*, since the Apperception Engine is able to learn from a handful of data.
- *interpretability*, which, though we cannot fully achieve with a neural model, can be incorporated into a model which can induce an explainable representation of the next state. This provides explainability at the level of the state only, as the state transition function itself remains opaque and inscrutable.

These two added dimensions enable us to compare Apperception Engine and the baseline(s) on fairer grounds by:

- learning from an increasing amount of data
- employing two baseline variations, differing in their ability to produce an interpretable representation of the next state: i) the *vanilla LSTM*, which does not produce an interpretable state at all and is a purely auto-regressive model, predicting the next raw sensory image from the previous ones, and ii) a *latent state LSTM* using a relaxed discrete distribution as a bottleneck, in which an LSTM encodes an almost discrete representation of the next state, before being decoded into a raw sensory output. Note these two baselines have the same number of hyperparameters.

We will abide by these desiderata, making slight adjustment on the *Seek Whence* task to ensure fair comparison and same testing conditions for both systems. Concretely, here we only compare the Apperception Engine to the *vanilla LSTM*, and do not compare it to the *latent state LSTM*. This is because since the Apperception Engine uses a pre-trained MNIST model to classify digits, providing the pre-trained MNIST model to the *vanilla LSTM* too obviates the need for an interpretable latent state, as the baseline is effectively predicting an already interpretable digit label from previous digit labels.

We opted for this approach on the *Seek Whence* task only, since we can utilise the pre-trained MNIST model to produce the target for the next sequence element. This is possible because we do not retrain the MNIST model, as training would otherwise lead to unstable learning, and degenerate solutions, which we observed in our earlier experiments.

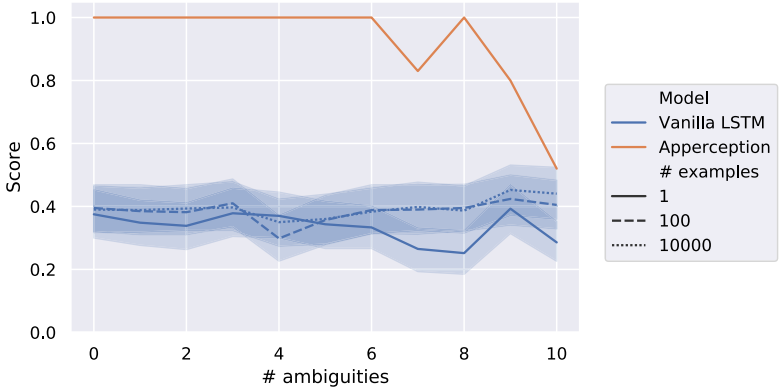


Fig. 7. The evaluation for the noisy *Seek Whence* sequences from MNIST, for the Apperception Engine and the *vanilla LSTM*. The x axis records the number of ambiguous images in the sequence while the y axis records the mean percentage accuracy over the ten sequences. The three variants of the *vanilla LSTM* are trained on an increasing number of training examples (1, 100, 10000). The shaded area is the 95% confidence interval across all the sequences and the 10 runs with different random seeds. (For interpretation of the colours in the figure(s), the reader is referred to the web version of this article.)

The *vanilla LSTM* model, depicted in Fig. 6, is using the same pre-trained MNIST model and an LSTM with 10 hidden units. It is optimised with the Adam optimiser [74] with a learning rate of 0.01. Every experiment is repeated 10 times on different random seeds, and the presented results are averaged over these 10 seeds.

5.1.5. Results

Our *Seek Whence* experiments contained 10 sequences:

0, 0, 0, 0, 0, 0, ...	**0, 0, 0, 1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4, ...**
0, 1, 2, 3, 4, 5, ...	**0, 5, 1, 5, 2, 5, 3, 5, 4, 5, 5, 5, ...**
5, 4, 3, 2, 1, 0, ...	**1, 1, 2, 1, 2, 3, 1, 2, 3, 4, 1, 2, 3, 4, 5, ...**
0, 1, 0, 1, 0, 1, ...	**1, 0, 1, 1, 1, 2, 1, 3, 1, 4, 1, 5, ...**
0, 0, 1, 1, 2, 2, 3, 3, 4, 4, ...	**1, 0, 1, 1, 1, 1, 1, 2, 1, 1, 3, 1, 1, 4, 1, ...**

For each symbolic sequence, we generated multiple MNIST image sequences. To generate an MNIST image sequence, we chose α, the number of ambiguities, and then sampled an image sequence with exactly α ambiguous images. We let α range from 0 to 10. An image counts as ambiguous relative to our threshold of 0.1 if two or more classes are assigned a probability of higher than 0.1 by our pre-trained neural network.

Fig. 7 shows how accuracy deteriorates as we increase the number of ambiguous images. The interpretations are very robust to a small number of ambiguous images. Eventually, once we have 10 ambiguous images (for sequences of average length 12), the results begin to degenerate, as we would expect. But the key point here is that the Apperception Engine's accuracy is robust to a number of ambiguities.

Comparing with the baseline, we can see that the performance of *vanilla LSTM* also drops with the increasing number of ambiguous images, when trained on a single example, though not as significantly as the Apperception Engine. This problem is fixed with an increasing number of training examples, as expected for a neural model, which perform well with noisy inputs. Qualitative analysis of models per sequence shows that the *vanilla LSTM* can easily learn to predict elements of easy sequences such as the all-zero and the alternating zero-one sequences. However, they struggle with other sequences, correctly predicting only static elements of a sequence (e.g. 1 in $(1, x, 1)^*$), but failing to learn an approximation to the *succ* relation that is necessary to learn a sequence of increasing numbers. Though seemingly unfair—requiring a neural model to learn the *succ* relation—we emphasise that any background knowledge needs to be explicitly hard-coded into the architecture of the model necessitating non-trivial modifications per task, as opposed to the Apperception Engine where the addition of background knowledge is straightforward. In addition, model performance is highly dependent on the parameter initialisations, shown by the confidence intervals in Fig. 7.

5.2. Sokoban

In Section 5.1, we used a hybrid architecture where the output of a *pre-trained* neural network was fed to the Apperception Engine. We assumed that we already knew that the images fell into exactly ten classes (representing the digits $0-9$), and that we had access to a network that already knew how to classify images.

But what if these assumptions fail? What if we are doing pure unsupervised learning and don't know how many classes the inputs fall into? What if we want to jointly train the neural network and solve the apperception problem *at the same time*?

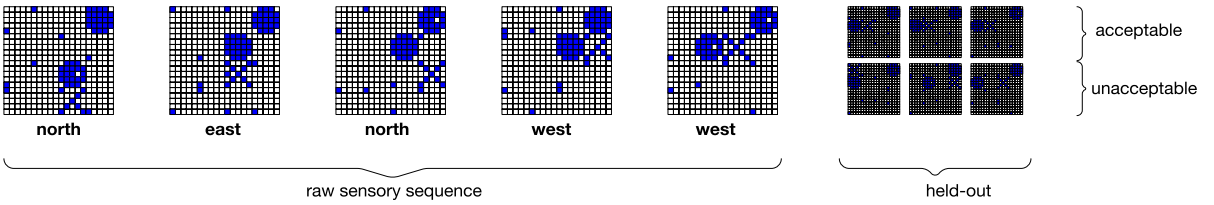


Fig. 8. The *Sokoban* task. The input is a sequence of (image, action) pairs. For the held-out time step, there is a set of acceptable images, and a set of unacceptable images. Note that the acceptable and unacceptable images are used only for evaluation, not for training.

In this next experiment, we combined the Apperception Engine with a neural network, simultaneously learning the weights of the neural network and also finding an interpretable theory that explains the sensory given.

We used *Sokoban*[13] as our domain. Here, the system is presented with a sequence of noisy pixel images together with associated actions. The system must jointly (i) parse the noisy pixel images into a set of persistent objects, and (ii) construct a set of rules that explain how the properties of those objects change over time as a result of the actions being performed. We wanted the learned dynamics to be 100% correct. Although next-step prediction models based on neural networks are able, with sufficient data, to achieve accuracy of 99% [31], this is insufficient for our purposes. If a learned dynamics model is going to be used for long-term planning, 99% is insufficiently accurate, as the roll-outs will become increasingly untrustworthy as we progress through time, since 0.99^t quickly approaches 0 as t increases.

5.2.1. The data

In this task, the raw input is a sequence of pairs containing a binarised 20×20 image together with a player action from $\mathcal{A} = \{north, east, south, west\}$. In other words, $\mathcal{R} = \mathbb{B}^{20\times 20} \times \mathcal{A}$, and $(r_1, \ldots, r_T)$ is a sequence of (image, action) pairs from $\mathcal{R}$.

Each array is generated from a 4×4 grid of 5×5 sprites. Each sprite is rendered using a certain amount of noise (random pixel flipping), and so each 20×20 pixel image contains the accumulated noise from the various noisy sprite renderings.

Each trajectory contains a sequence of (image, action) pairs, plus held-out data for evaluation. Because of the noisy sprite rendering process, there are many possible acceptable pixel arrays for the final held-out time step. These acceptable pixel arrays were generated by taking the true underlying symbolic description of the *Sokoban* state at the held-out time step, and producing many alternative renderings. A set of unacceptable pixel arrays was generated by rendering from various symbolic states distinct from the true symbolic state. Note that the acceptable and unacceptable images are used only for evaluation, not for training. Fig. 8 shows an example.

In our evaluation, a model is considered accurate if it accepts every acceptable pixel array at the held-out time step, and rejects every unacceptable pixel array. This is a stringent test. We do not give partial scores for getting some of the predictions correct.

5.2.2. The model

In outline, we convert the raw input sequence into a disjunctive input sequence by imposing a grid on the pixel array and repeatedly applying a binary neural network to each sprite in the grid. In detail:

1. We choose a sprite size k, and assume the pixel array can be divided into squares of size $k \times k$. We assume all objects fall exactly within grid cell boundaries. In this experiment,[14] we set $k = 5$.
2. We choose a number m of persistent objects $o_1, \ldots, o_m$. We choose a number n of distinct types of objects $v_1, \ldots, v_n$, and add an additional type v_0 (where v_0 is a distinguished identifier that will be used to indicate that there is *nothing* at a grid square). We choose a total map $\kappa : \{o_1, \ldots, o_m\} \to \{v_1, \ldots, v_n\}$ from objects to types. For example, we might choose three objects ($m = 3$) and two types ($n = 2$), where o_1 is of type v_1, while both o_2 and o_3 are of type v_2.
3. We apply a binary neural network (**BNN**) to each $k \times k$ sprite in the grid. The BNN implements a mapping $\mathbb{B}^{k\times k} \to \{v_0, v_1, \ldots, v_n\}$. If sprite s is at (x, y), then $BNN(s) = v_i$ can be interpreted as: it looks as if there is *some object* of type v_i at grid cell (x, y), for $i > 0$. If $BNN(s) = v_0$, it means that there is *nothing* at (x, y). See Fig. 9. For each time step, for each grid cell, we convert the output of the BNN into a disjunction of ground atoms: if sprite s is at (x, y), and $BNN(s) = v_i$, then we create a disjunction featuring each object o of type v_i stating that *any* of them could be at (x, y). See Fig. 10.
4. We use the Apperception Engine to solve the disjunctive apperception task generated by steps 1–3.

[13] *Sokoban* is a puzzle game where the player controls a man who moves around a two-dimensional grid world, pushing blocks onto designated target squares.

[14] Giving the system the grid cell boundaries is a substantial piece of information that helps the Apperception Engine overcome the combinatorial complexity of the problem. But for a series of experiments in which we do *not* provide any spatial information, see Section 5.3.

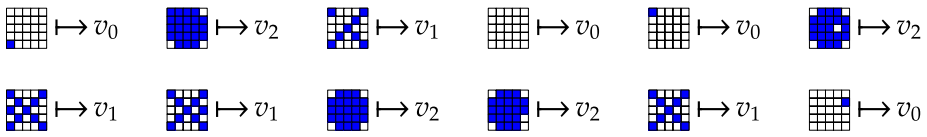


Fig. 9. A binary neural network maps sprite pixel arrays to types $\{v_0, v_1, v_2\}$.

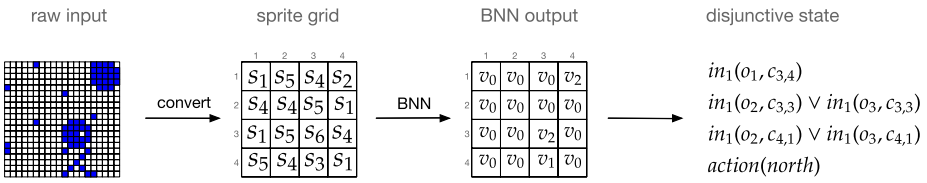


Fig. 10. A binary neural network converts the raw pixel input into a set of disjunctions. There is one object o_1 of type v_1 and two objects o_2, o_3 of type v_2. If sprite s is at (x, y), and $BNN(s) = v_i$, then we create a disjunction featuring each object o of type v_i stating that any of them could be at (x, y).

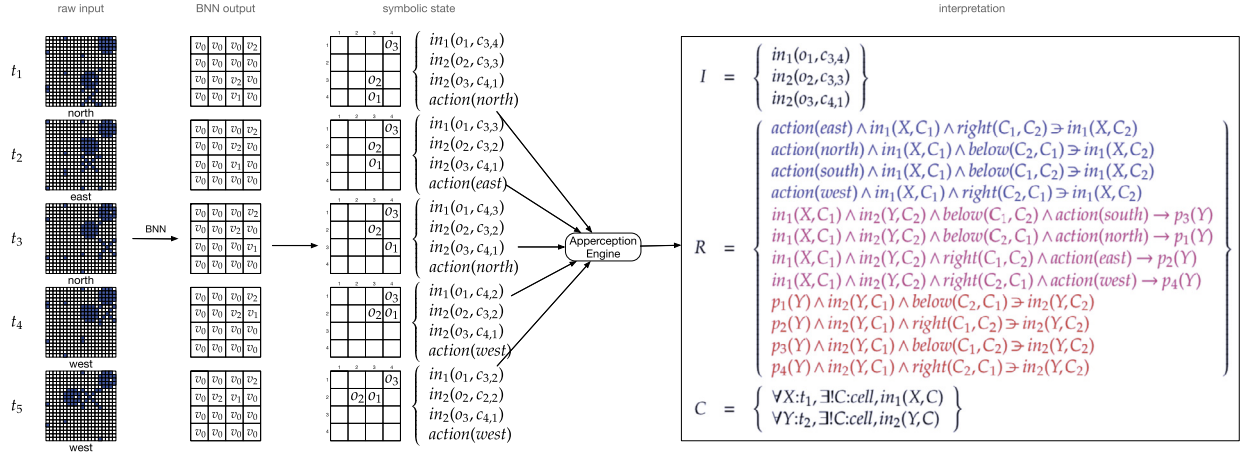


Fig. 11. Interpreting *Sokoban* from raw pixels. Raw input is converted into a sprite grid, which is converted into a grid of types v_0, v_1, v_2. The grid of types is converted into a disjunctive apperception task. The Apperception Engine finds a unified theory explaining the disjunctive input sequence, a theory which explains how objects' positions change over time. The top four rules of R (in blue) describe how the man X moves when actions are performed. The middle four rules (in magenta) define four invented predicates $p_1, \ldots, p_4$ that are used to describe when a block is being pushed in one of the four cardinal directions. The bottom four rules (in red) describe what happens when a block is being pushed in one of the four directions.

For each *Sokoban* trajectory, we gave the Apperception Engine 48 hours running on a standard Unix desktop to find the lowest cost interpretation according to the score of Definition 12.[15]

5.2.3. Understanding the interpretations

Fig. 11 shows the best theory[16] found by the Apperception Engine from one trajectory of 17 time steps. When neural network next-step predictors are applied to these sequences, the learned dynamics typically fail to generalise correctly to different-sized worlds or worlds with a different number of objects [31]. But the theory leaned by the Apperception Engine applies to all *Sokoban* worlds, no matter how large, no matter how many objects. Not only is this learned theory correct, but it is provably correct.[17]

Fig. 12 shows the evolving state over time. The grid on the left is the raw perceptual input, a grid of 20×20 pixels. The second element is the output of the binary neural network: a 4×4 grid of predicates v_0, v_1, v_2. If v_i is at (x, y), this means "it looks as if there is *some* object of type i at (x, y)" (but we don't yet know which particular object). So, for example, the grid in the top row states that there is some object of type 1 at $(3, 4)$, and some object of type 2 at $(4, 1)$. Here, v_0 is a distinguished predicate meaning there is *nothing* at this grid square.

[15] See the Supplementary Materials for more details on this experiment.

[16] The rules in R describe the state transitions conditioned on actions being performed. They do not describe the conditions under which the particular actions are available. For example, in *Sokoban*, you cannot push a block left if there is another block to the left of the block that you are trying to push. Action availability is not represented explicitly in the theory $\theta = (\phi, I, R, C)$.

[17] The fixed rules of Sokoban determine a deterministic state transition function $tr : S \times A \rightarrow S$ from board states and actions to board states. We can show that, for any board state S and action A, if $\tau(\theta)$ contains $S \cup A$ at time t, then $\tau(\theta)$ contains $tr(S, A)$ at time $t + 1$.

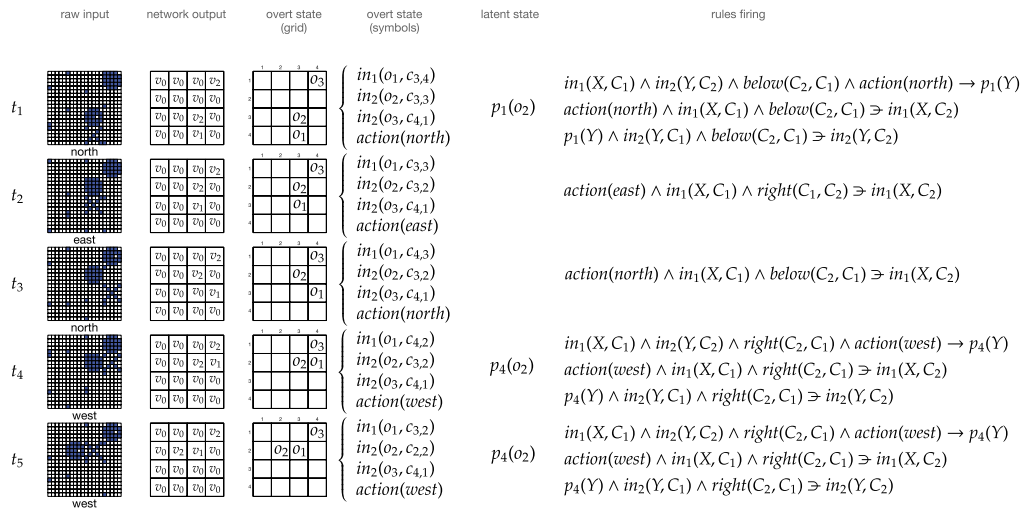


Fig. 12. The state evolving over time. Each row shows one time step. We show the raw pixel input, the output of the binary neural network, the set of ground atoms that are currently true, and the rules that fire.

The third element is a 4×4 grid of persistent objects: if o_i is at (x, y) this means: the particular persistent object o_i is at (x, y). The fourth element is a set of ground atoms. This is a re-representation of the persistent object grid (the fourth element) together with an atom representing the player's action. The fifth element shows the latent state. In *Sokoban*, the latent state stores information about which objects are being pushed in which directions. Here, in the top row, $p_1(o_2)$ means that persistent object o_2 is being pushed up. The sixth element shows which rules fire in which situations. In the top row, three rules fire. The first rule describes how the man moves when the *north* action is performed. The second rule concludes that a block is pushed northwards if a man is below the block and the man is moving north. The third rule describes how the block moves when it is pushed northwards.

Looking at how the engine interprets the sensory sequence, it is reasonable—in fact, we claim, inevitable—to attribute *beliefs* to the system. In the top row of Fig. 11, for example, the engine believes that the object at $(3, 3)$ is the same type of thing as the object at $(4, 1)$, while the object at $(3, 4)$ is not the same type of thing as the object at $(4, 1)$. As well as beliefs about particular situations, the system also has general beliefs that apply to all situations. For example, whenever the *north* action is performed, and the man is below a block, then the block is pushed upwards. One of the reasons for using a purely declarative language such as *Datalog*$^{\ni}$ is that individual atoms and clauses can be interpreted as beliefs. If, on the other hand, the program that generated the trace had been a procedural program, it would have been much less clear what beliefs, if any, the procedure represented (Fig. 13).

5.2.4. The baselines

We use two baselines for the Sokoban task, per the desiderata of Section 5.1.4. Both the baselines apply an array of parameter-sharing multilayer perceptrons (MLPs) to each block of the game state, and concatenate the result with the one-hot representation of the actions as the model input. The *vanilla LSTM*, is fed the model input, and trained to directly predict the desired raw output, following the output MLP, without any constraints on the internal representation. The *latent state LSTM*, on the other hand, is fed the model input to produce the parameters of the Gumbel-Softmax continuous approximation to the categorical distribution, one per each block of the state, which are then mapped to the next state with an output MLP. These distributions, when the model is learned well, can encode a close-to-symbolic representation of the current state without direct supervision. The output MLP, present in both baselines, is a two-layer perceptron which targets the next raw state of the sequence. Note that these two baselines have the same number of trainable parameters and differ only in the ability to produce an interpretable latent state (Fig. 14).

Given that these presented baselines are generative models over a large state space, in order to compare them to the Apperception Engine, we add a density estimation classifier on its output. The classifier fits a Gaussian per class, trained on log-probabilities of independently sampled acceptable and unacceptable test states calculated over the Bernoulli distribution outputted by the model. Note that this is done for the purpose of our evaluation only.

We trained the baselines with the Adam optimizer, varying the learning rate in $[0.05, 0.01, 0.005, 0.001]$, batch size in $[512, 1024]$ and executing each experiment 10 times. We selected the best set of hyperparameters by choosing the parameters with the best development set performance, and averaged the performance across 10 repetitions with different random seeds. During training of the *late state LSTM*, we annealed the temperature of the Gumbel-Softmax with an exponential decay, from 2.0 to 0.5 with a per-epoch decay of 0.0009.

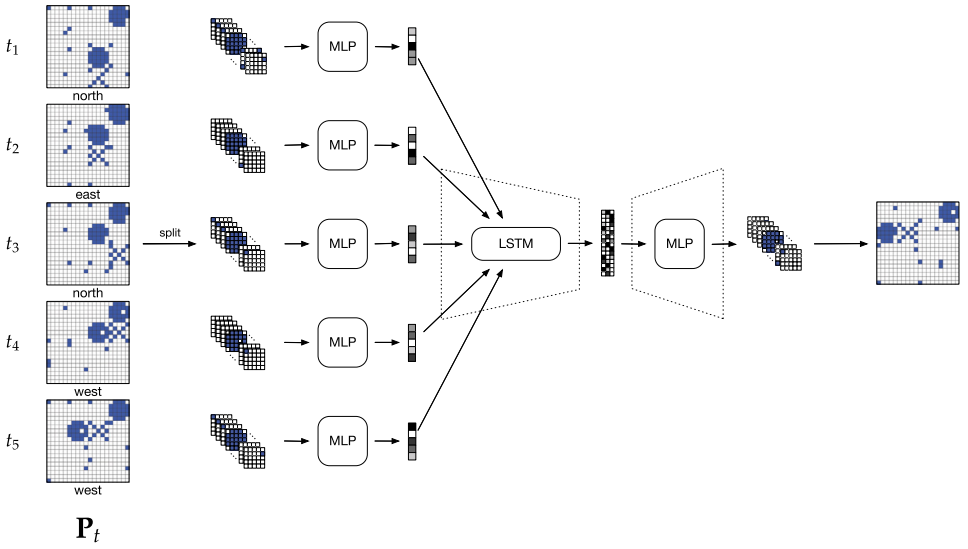


Fig. 13. The *latent state LSTM* baseline for the Sokoban task. The perceptual input data is per-sprite fed into an array of parameter-sharing MLPs and then concatenated with the action data. The result is fed into an LSTM which predicts the parameters of an array of Gumbel-Softmaxes, one per sprite. These distributions approximate a symbolic state of the board, which is then decoded back into perceptual data of the following step. This differs from the *vanilla LSTM* baseline for the same task which does not map to the latent state of Gumbel-Softmax distributions only, meaning the induced state of the board is dense and non-interpretable.

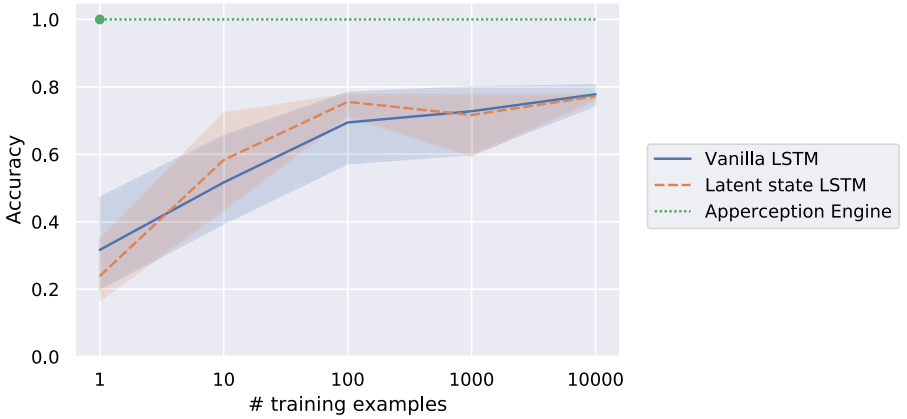


Fig. 14. The results on the Sokoban task. Apperception is trained on only a single example and the dashed line represents the apperception results on only that example. The neural baseline is trained on an increasing number of training examples. The shaded area is the 95% confidence interval on 10 runs with different random seeds.

5.2.5. Results

We took ten trajectories of length 22. For each trajectory, we evaluated on eight subsequences of lengths 3 to 17 in increments of 2. For each subsequence of length n, we used the remaining $22 - n$ time-steps for evaluation. The results are shown in Fig. 15. While most of the trajectories do not contain enough information for the engine to extract a correct theory, three of them are able to achieve 100% accuracy on the held-out portion of the trajectory. Of course, getting complete accuracy on the held-out portion of a single trajectory is necessary, but not sufficient, to confirm that the induced theory is actually correct on all possible Sokoban configurations. We checked each of the three accurate induced theories, and verified by inspection that one of the three theories was correct on all possible Sokoban maps, no matter how large, and no matter how many objects.[18]

Next, we compare the Apperception Engine to the LSTM baselines. We train both baseline models on a single trajectory containing enough information to extract the correct theory. In addition, we train the neural baseline on an increasingly large training set.

[18] Note that state of the art ILP systems are unable to learn the correct dynamics of Sokoban given from hundreds of trajectories [75].

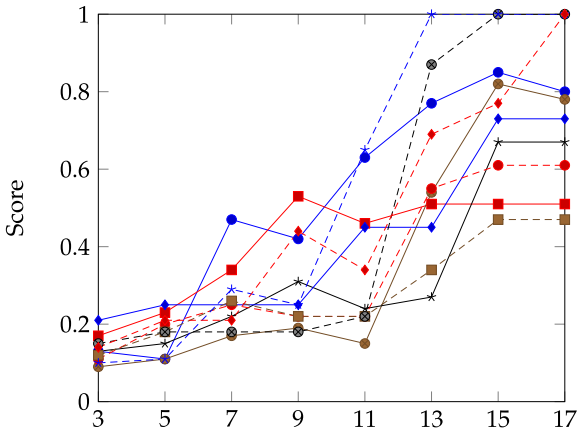


Fig. 15. The results for *Sokoban* on ten trajectories. The x axis records the number of time-steps provided as input. The y axis records the mean percentage accuracy over the held-out time-steps.

Both the *vanilla LSTM* and *latent state LSTM* are not able to absolutely correctly distinguish between acceptable and unacceptable next steps, neither from the single example, nor a large number of examples. However, as expected, the accuracy of both baselines increases with increasing size of the training set, though it shows the tendency to plateau without reaching the maximum. By inspecting the latent distributions of the *latent state LSTM*, we see that the model learns to approximate the symbolic state of the board well—the resulting distribution roughly corresponds to the state, though the visual inspection of the decoded state shows that the model largely focuses on the large objects (such as the block *O*) the best, while possibly ignoring smaller objects (such as the man *X*). An important thing to emphasise here is that the performance of the model is highly dependent on the initial random seed which makes cherry-picking a necessity. From these findings, we conclude that the neural networks can somewhat learn to predict the next state, and even induce a near-to-symbolic representation of the state, though the model requires a larger number of training instances and the performance of the model is not fully reliable.

On the other hand, the Apperception Engine is able to learn a fully explainable theory of the said sequence from that single example.

5.3. Fuzzy sequences

In the *Sokoban* experiments described in Section 5.2 above, the system jointly solved low-level perception and high-level apperception. It performed low-level perception by finding the weights of the binary neural network and it performed high-level apperception by finding a unified theory that solves the apperception task. Because both tasks were encoded as a single SAT problem, and solved jointly, information could flow in both directions, both bottom-up and top-down.

But there were two pieces of domain-specific knowledge that we injected: the dimensions of the sprite grid and the number of distinct types of objects. In this final set of experiments, we investigate what happens when we jointly solve low-level perception and high-level apperception *without* providing a spatial structure or any hint as to the number of classes.

5.3.1. The data

In these experiments, the inputs are binary sequences that were generated by a stochastic process from an underlying symbolic sequence with low Kolmogorov complexity. See Fig. 16. We start with a simple symbolic sequence, e.g., *aabbaabbaabb* ... We generate a map from symbols to sets of binary vectors. This map contains some ambiguities, some binary vectors that are associated with multiple symbols: in Fig. 16, for example, 011 is ambiguous between *a* and *b*. We convert the sequence of symbols into a sequence of binary vectors by sampling (uniformly randomly) for each symbol in the sequence one of the corresponding vectors. Then we concatenate the binary vectors into one large sequence, thus throwing away the information about where the sequence is segmented. Fig. 17 shows six example sequences.

We want the Apperception Engine to recover the underlying symbolic structure from this fuzzy, ambiguous sequence, without giving it privileged access to the segmentation information—we want the system to recover the segmentation information as part of the perceptual process.

5.3.2. The held-out data

To evaluate the accuracy of the models, we consider what the model predicts about a held-out portion of the sequence. Because the sequence is ambiguous, there are many different acceptable continuations of it (see Fig. 18).

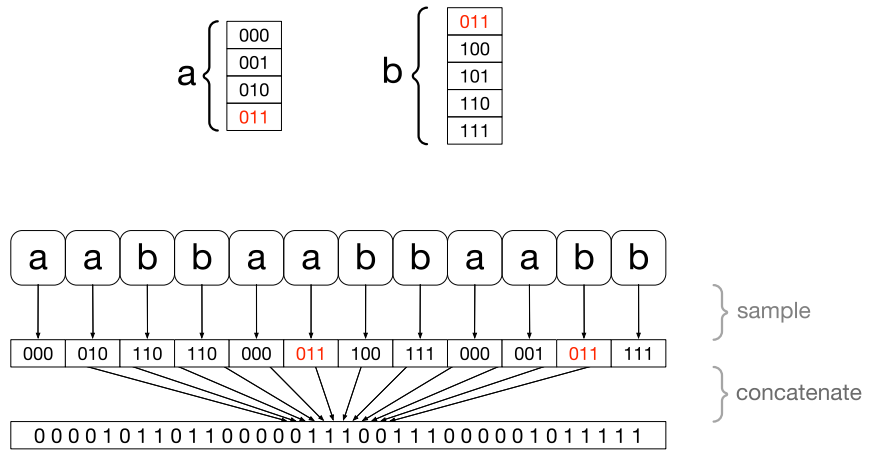


Fig. 16. Generating fuzzy sequences. We start with a symbolic sequence, and convert into a binary vector. We use a map from discrete symbols to sets of binary vectors. Some of the binary vectors are ambiguous between different symbols; these are shown in red. For each symbol in the original symbolic sequence, we sample one of the corresponding vectors using the map. Finally, we concatenate the binary vectors to produce one large sequence where the segmentations have been thrown away.

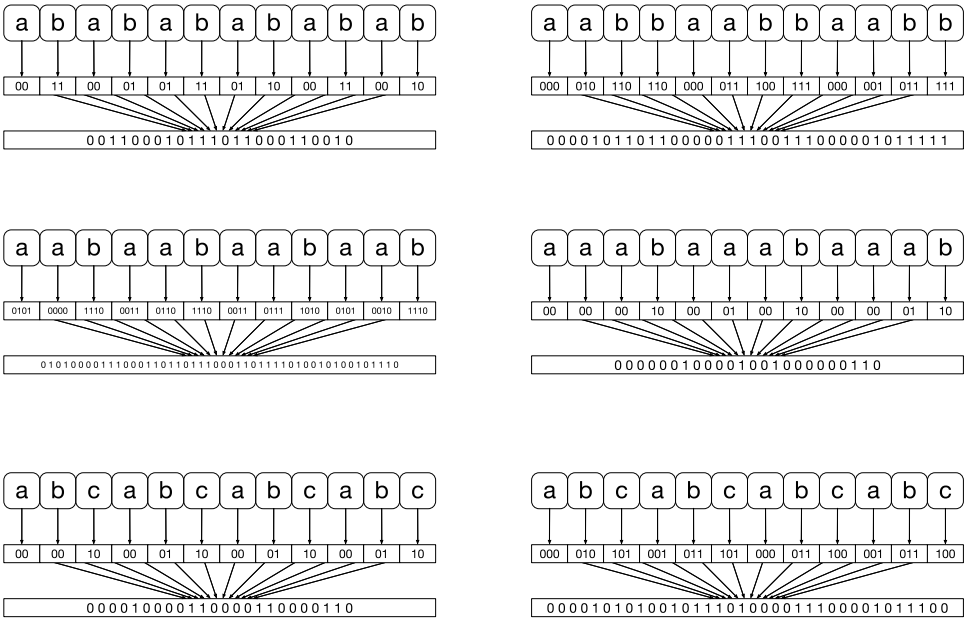


Fig. 17. Six example sequences. In each, we show the original symbolic sequence, the randomly sampled vectors, and the final concatenated result.

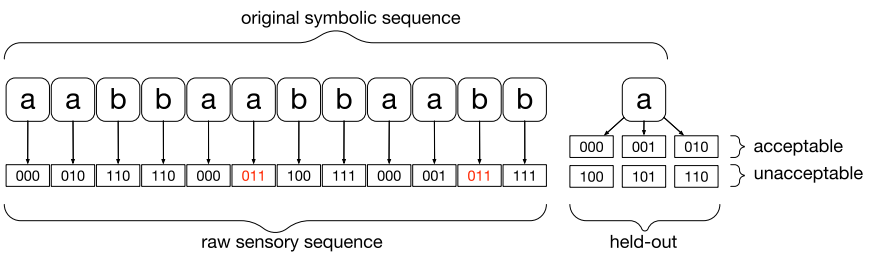


Fig. 18. A fuzzy sequence with held-out data. Ambiguous vectors are shown in red.

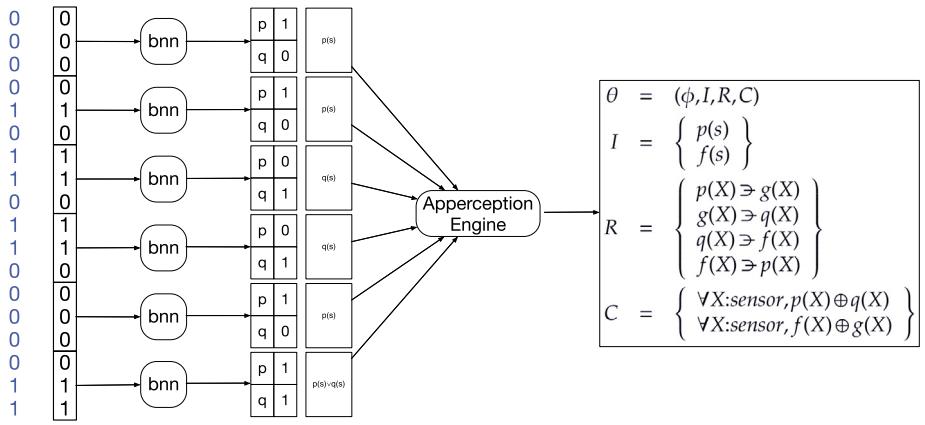


Fig. 19. Solving the fuzzy sequence with $k_g = 3$ and $n_g = 2$ (the correct guesses). The interpretation discerns the underlying pattern *ppqqppqqppqq*... which is isomorphic to the original symbolic sequence *aabbaabbaabb*....

We evaluate a model as accurate on the sequence if it accepts *every* correct continuation and rejects *every* incorrect one, stringently giving no partial credit.

5.3.3. The models

To find the best interpretation of a fuzzy sequence, we consider a set of models, and find the one with the highest probability (see Definition 12). Each model is an Apperception Engine combined with a binary neural network.

Recall that the given binary sequence is formed by starting with a symbolic sequence $(S_1, \ldots, S_T)$ from an alphabet of size n, then sampling, for each S_i, a binary vector of length k, and then concatenating the vectors together to produce a single binary vector of size $T \times k$.

We withhold certain crucial information from our model: we do not provide the size k of the constituent binary vectors, and we do not provide the number n of symbols in the alphabet. Instead, we perform a grid search over pairs (k_g, n_g), where k_g is the guessed value of k and n_g is the guessed value of n, and choose the pair with the best score.

For a particular (k_g, n_g) pair, we divide the given binary sequence into $T \cdot k/k_g$ vectors $\mathbf{v}_1, \ldots, \mathbf{v}_{T \cdot k/k_g}$ each of size k_g, and create a binary neural network with output layer of size n_g. We apply the binary neural network to each vector. We use a type signature with n_g unary predicates $p_1, \ldots, p_{n_g}$.

We apply the network to each vector $\mathbf{v}_1, \ldots, \mathbf{v}_{T \cdot k/k_g}$ and generate, for each $\mathbf{v}_t$, a disjunction $p_{x_1}(s) \vee \ldots \vee p_{x_m}(s)$ holding at time t, where $\{p_{x_1}, \ldots, p_{x_m}\}$ are the subset of predicates $\{p_1, \ldots, p_{n_g}\}$ such that the network's x_i'th output is 1. For example, if $n_g = 5$ and the neural network's output layer is $(1, 0, 1, 1, 0)$ on input $\mathbf{v}_t$, then the disjunction $p_1(s) \vee p_3(s) \vee p_4(s)$ is added at time t. The s is a distinguished constant representing the (single) sensor.

In terms of the formalism of Section 4.2, the raw input sequence is a sequence $(r_1, \ldots, r_T)$ of binary vectors from $\mathbb{B}^k$. The framework $(n, \pi_{\mathbf{w}}, \Delta, \phi, C)$ consists of:

- A number n of classes
- A binary neural network $\pi_{\mathbf{w}}$ mapping $\mathbb{B}^k \to \mathbb{B}^n$
- A "disjunctifier" Δ that translates the output of $\pi_{\mathbf{w}}$ into a single disjunction of ground atoms
- A type signature $\phi = (T, O, P, V)$ consisting of one type, *sensor*, one object s of type sensor, and predicates $p_1, .., .p_n$ for each of the output classes.
- C contains the constraint that every sensor satisfies exactly one of the p_i predicates

In a little more detail, the binary neural network BNN, parameterised by weights w, takes a binary vector of length k and maps it to a binary vector of length n. Now the perceptual classifier π uses the binary neural network's output to generate a disjunction: if the i'th output is 1, then the sensor s could satisfy p_i:

$$\pi_w(r) = \left\{ \bigvee \{ p_i(s) \mid BNN_w(r)[i] = 1 \} \right\}$$

Figs. 19 and 20 provide two examples. In Fig. 19, the guesses are correct as $k_g = k$ and $n_g = n$. Here, each vector $\mathbf{v}_i$ is of length 3, just as in the true generative process, and there are two predicates p and q corresponding to the two symbols of the original symbolic sequence. In Fig. 20, the guesses are incorrect as $k_g = 2 \neq k$ and $n_g = 3 \neq n$.

Repeated application of the binary neural network to the vectors $\mathbf{v}_1, \ldots, \mathbf{v}_{T \cdot k/k_g}$ produces $T \cdot k/k_g$ disjunctions of the form $p_{x_1}(s) \vee \ldots \vee p_{x_m}(s)$. See the fifth column in Figs. 19 and 20. The disjunctive sensory sequence is passed to the Apperception Engine which attempts to resolve the disjunctions and find a unified interpretation. (Note that strictly speaking there is no temporal sequence here: the weights of the binary neural network, the resolutions of the disjunctions, and the unified

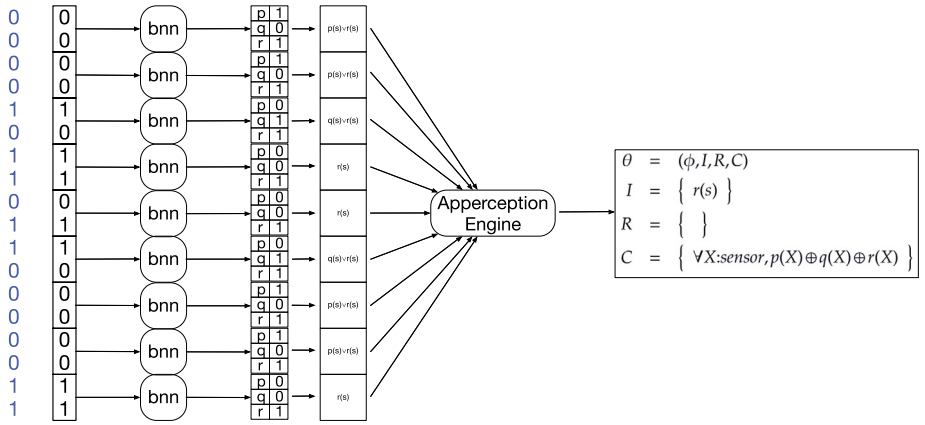


Fig. 20. Solving the fuzzy sequence with $k_g = 2$ and $n_g = 3$ (the wrong guesses). The interpretation maps all vectors to c and produces a degenerate interpretation in which $r(s)$ remains true and nothing changes.

theory are found jointly and simultaneously. But for some simplified expository purposes it does make sense to think of the binary neural network as operating before the Apperception Engine).

Once we have chosen k_g and n_g, we create an initial type signature with n_g unary predicates $p_1, \ldots, p_{n_g}$, and then iterate through increasingly complex templates, with an increasingly large number of invented predicates and additional rules. This iterative procedure produces a set of theories that need to be compared. From each pair of guesses of k_g and n_g, we find a vector w of network weights plus a theory θ that satisfies the unity conditions. We score (w, θ) using the function of Definition 12.

5.3.4. Understanding the interpretations

Fig. 20 shows one interpretation of the fuzzy sequence of Fig. 16. In this interpretation, the guessed vector length $k_g = 2$ is wrong as the actual vector length is 3. The guessed number of predicates $n_g = 3$ is also wrong as the fuzzy sequence was generated from the symbol sequence *aabbaabbaabb* ... that uses 2 symbols.

In this interpretation, vectors are mapped to concepts as follows:

$$(0,0) \mapsto p(s) \vee r(s) \quad (0,1) \mapsto r(s) \quad (1,0) \mapsto q(s) \vee r(s) \quad (1,1) \mapsto r(s)$$

Note that *every* vector is mapped to r.

The interpretation found is very simple. The atom $r(s)$ is initially true, and then remains true forever. Nothing changes. Because the guessed vector size is wrong, the system is unable to discern any distinctions in the input, and maps everything to the single concept r. This interpretation makes no discriminations, and blurs all distinctions. It is inaccurate on the held-out data.

Fig. 19 shows another interpretation of the same noisy sequence. In this case, the guessed vector size $k_g = 3$ is correct, as is the guessed number of predicates $n_g = 2$. In this interpretation, vectors are mapped to concepts as follows:

$$(0,0,0) \mapsto p(s) \quad (0,0,1) \mapsto p(s) \quad (0,1,0) \mapsto p(s) \quad (0,1,1) \mapsto p(s) \vee q(s)$$
$$(1,0,0) \mapsto q(s) \quad (1,0,1) \mapsto q(s) \quad (1,1,0) \mapsto q(s) \quad (1,1,1) \mapsto q(s)$$

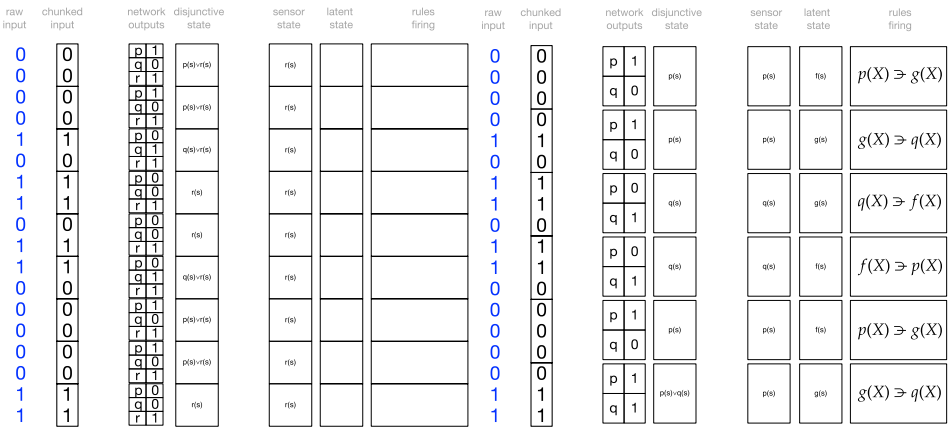
Here, the mapping has one ambiguity on vector $(0,1,1)$. Note that this ambiguity is unavoidable given the original ambiguous mapping in Fig. 16.

Note that the system has discerned the underlying symbolic sequence *ppqqppqqppqq* ..., isomorphic to the original symbolic sequence *aabbaabbaabb* ... of Fig. 16 that was used to generate the fuzzy sequence. The rules R use f and g as invented predicates to count how many times we are in the two states of p and q.

It is pleasing that the system is able to recover the underlying symbolic sequence as well as the low-level mapping from vectors to concepts, from fuzzy ambiguous sequences. This interpretation is accurate on all the held-out data (see Section 5.3.2).

The two interpretations are compared in Fig. 21. The left figure (a) shows the interpretation of Fig. 20 which blurs all distinctions. The right figure (b) shows the interpretation of Fig. 19, which correctly discerns the underlying symbolic structure.

The probability of the accurate interpretation (see Definition 12) is significantly higher than the probability of the inaccurate interpretation. In general, throughout our experiments, the most probable interpretations coincide with the accurate interpretations. This means we are able to retrieve the correct values of k_g and n_g by taking the interpretation with the highest probability. See Section 5.3.6.



(a) Incorrect guess with $k_g = 2$ and $n_g = 3$ (b) Correct guess with $k_g = 3$ and $n_g = 2$

Fig. 21. Two interpretations of a sequence generated from *aabbaabbaabb* ... with $k = 3$. In both (a) and (b), we show the raw concatenated input, the input divided into chunks of size k_g, the output of the binary neural network, and the disjunction generated by the multiclass classifier. The sensor state column shows how each disjunction is resolved, while the latent state shows the ground atoms that were invented to explain the surface sequence. The final column shows all the rules whose preconditions are satisfied at that moment.

5.3.5. The baselines

Both baselines for the *Fuzzy Sequences* follow the desiderata of Section 5.1.4 too. Concretely, after applying a two-layer MLP to each input element, the input is passed to either i) a *vanilla LSTM*, followed by a two-layer MLP, a model which directly predicts the next raw element of the sequence, or ii) the *latent state LSTM* which predicts the interpretable Gumbel-Softmax distribution, followed by a two-layer MLP which decodes the samples from the distribution into the following raw element of the sequence.

We trained the baselines with the Adam optimizer, and trained the models on two dataset sizes, one containing a single sequence, the other containing 1000 sequences. For the single sequence trained dataset, we set the learning rate to 0.01, and for the 1000-sequence dataset, we did a hyperparameter search over the batch size in $[256, 512]$ and the learning rate in $[0.01, 0.005]$. After noticing that the *latent state LSTM* struggles with producing a crisp distribution, we introduced the KL-weighing β parameter [76] and set it to $\beta = 0.1$ to produce better representations. We ran both baselines on each instance of the task 10 times on different random seeds.

Given that the output of these baselines is constrained into a small space, we do not apply the density estimation classifier as in the *Sokoban* baseline case, but simply discretise the output and check whether it is in the test set of acceptable or unacceptable sequences. If the output was in the set of acceptable sequences, we scored the model with 1.0, 0.0 if it was in the set of unacceptable ones and 0.5 if it was in neither. We present the model accuracy averaged over the 10 different random seeds.

5.3.6. Results

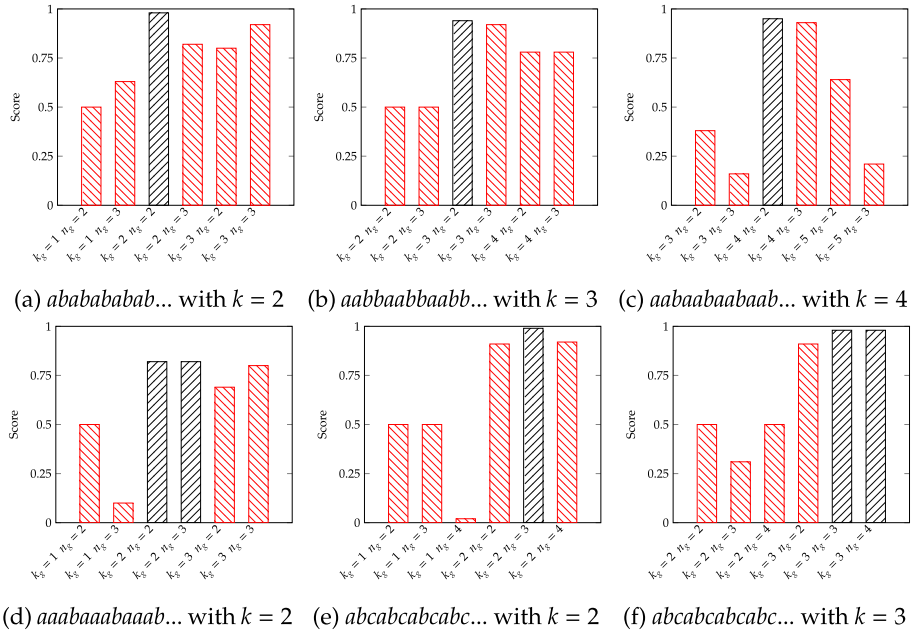
Fig. 22 shows, for six fuzzy sequences, an evaluation of different theories with different guesses for k_g and n_g. The accurate theories (those that correctly predict *all* held-out data) are shown in black, while inaccurate theories are shown in red. Notice that the score (based on the log probability of the (w, θ) pair from Definition 12) is a reliable indicator of the accuracy of the interpretation. This means that we can run a grid search over guesses for k_g and n_g, choose the interpretation with the highest score, and confidently expect that this interpretation will be accurate on the held-out data. The central point here is that we do not need to provide the system with information about the way the fuzzy sequence is grouped into chunks. Rather, *the system itself can induce the correct way to group the data as part of the apperception process*.

Since the induced baseline representations often were not sharply discrete, we did not compare it to the Apperception Engine on the same scoring but we evaluated it only on its capacity to correctly predict elements of the sequence.

Training on a single sequence, in Fig. 23, we observe that both the baselines correctly learn to predict only the *abababab...* sequence and the *aaabaaabaaab...* sequence. The *latent state LSTM* also correctly learns to predict the *abcabcabcabc...* (k = 2) sequence, with *vanilla LSTM* missing the correct choice of parameters on said sequence.

On 1000 sequences, in Fig. 24, we observe on average higher accuracies achieved by both baselines, in particular for the *aabbaabbaabb...* sequence, even though these accuracies cannot indicate the correct choice of parameters of that sequence. This indicates that with more data the models are able to learn the sequences better, in all cases but most interestingly the *abababab...* sequence, where we now see a significant drop in accuracy. In addition, the *latent state LSTM* can now predict the parameters of the *aabaabaabaab...* sequence correctly.

Our baselines, thus, though they can certainly learn the correct sequence in some seeds, on average cannot provide a reliable accuracy for choosing correct parameter guesses across all sequences. We conjectured this might be due to



(a) *abababababab*... with $k = 2$ (b) *aabbaabbaabb*... with $k = 3$ (c) *aabaabaabaab*... with $k = 4$

(d) *aaabaaabaaab*... with $k = 2$ (e) *abcabcabcabc*... with $k = 2$ (f) *abcabcabcabc*... with $k = 3$

Fig. 22. Evaluating interpretations for six fuzzy sequences. Interpretations that are accurate (on *all* held-out data) are shown in black, while inaccurate interpretations are shown in red. In all our experiments, the highest-scoring interpretations are always accurate.

underfitting—i.e. the model needing more capacity, so we experimented with significantly larger models (up to 10 times the number of parameters). However, we observed that in those cases, the models memorised the sequences and increased in accuracy in general, but the accuracy again could not be reliably used to accurately pick the correct sequence parameters. We also took a look at the curious results on the *abababababab*... sequence, wondering why a model trained on a single sequence performed better than the one trained on 1000 sequences. Interestingly, both models correctly predicted the first digit of both symbols (*a* encoded as either 00 or 01 and *b* encoded as 01 or 10 or 11), but they hedged their bets on the second digit often encoding it as a value close to 0.5. In the case of the model trained on a single sequence, the model overfit to the sequence, finding a good optimum more frequently on average than the model trained on 1000 sequences which averaged the second digit more often and made predictions of the ambiguous 01 instead of 00 for *a*. On a related note, looking into the Gumbel-Softmax parameters of the *latent state LSTM* shows that, when the model learns the sequence well, it does induce a meaningful crisp distribution, but when it does not it learns a distribution which is not useful for interpretation. We also notice that for the sequences these models cannot learn, they exhibit severe overfitting on the single example, an expected phenomenon when trained on a low number of examples. In addition, yet again we noticed that the model is seed dependent, stable only when learning the sequence in the two aforementioned cases.

6. Discussion

In conclusion, we outline the key strengths and limitations of our system.

6.1. Appealing features of the Apperception Engine

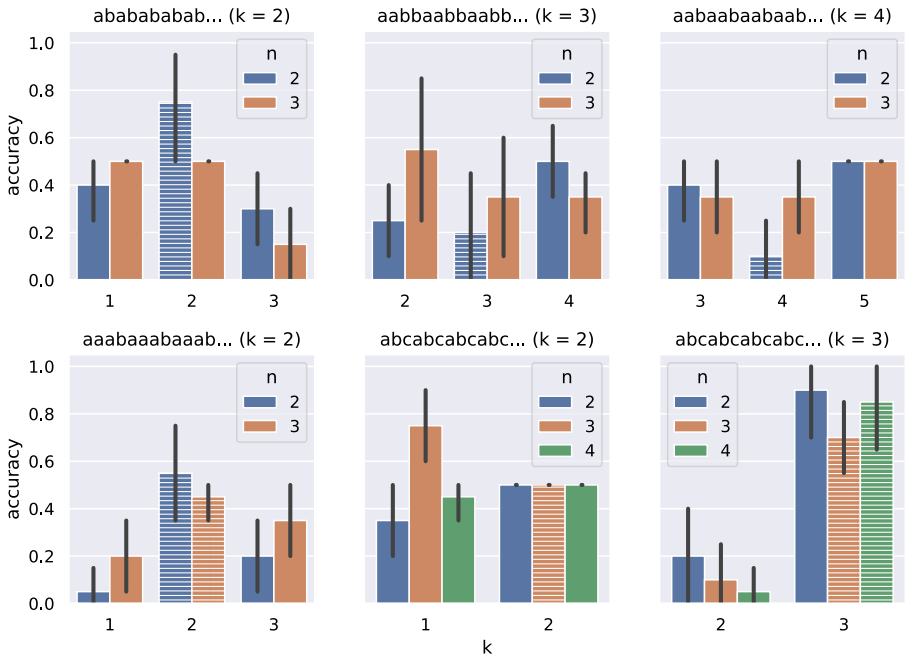
As a system for unsupervised induction of general laws from raw unprocessed data, the Apperception Engine has the following appealing features: it is (i) interpretable, (ii) accurate, and (iii) data-efficient. We shall consider each in turn.

6.1.1. Interpretability

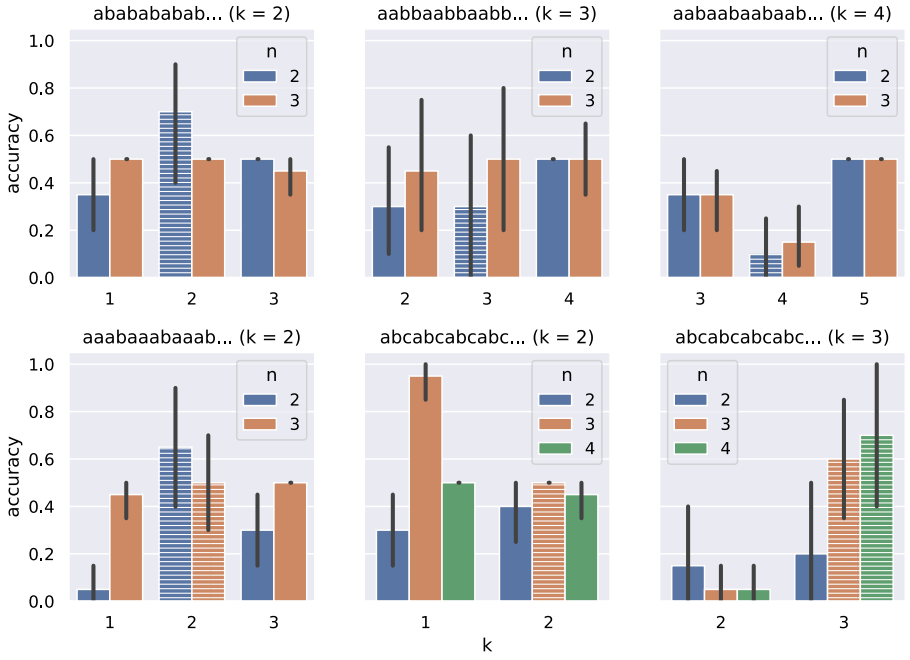
The Apperception Engine produces a theory, an explicit program in $Datalog^{\ni}$, to make sense of its given input. This theory is interpretable at three levels.

First, we can understand the *general ontology* that the system is using: we know what persistent objects the system has posited, the types of those persistent objects, and the sorts of predicates that can apply to those objects. In *Sokoban*, for example, we understand there are three objects: o_1 of type t_1, and o_2 and o_3 of type t_2 (see Section 5.2.3). The synthesised constraints act like type judgements to restrict the set of models. For example, the constraint $\forall Y{:}t_2, \exists! C{:}cell, in_2(Y, C)$ states that every block is placed in exactly one cell.

Second, we can understand how the system interprets *particular moments in time*. In *Sokoban*, in Fig. 11, at time step t_1, for example, we understand that the system thinks o_1 is below o_2, that o_3 is in the top right corner, and that o_2 is



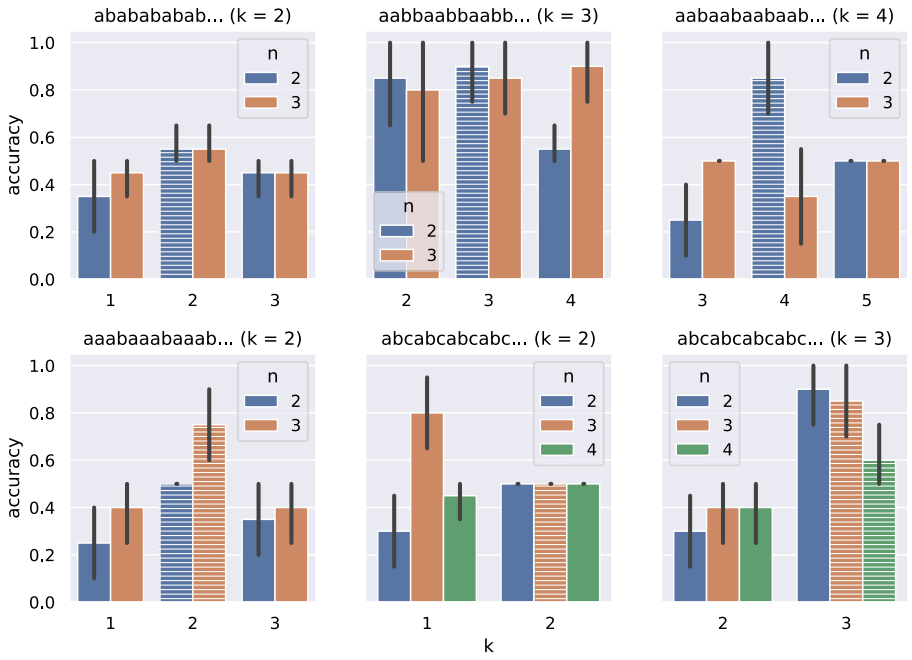
(a) *Vanilla LSTM*



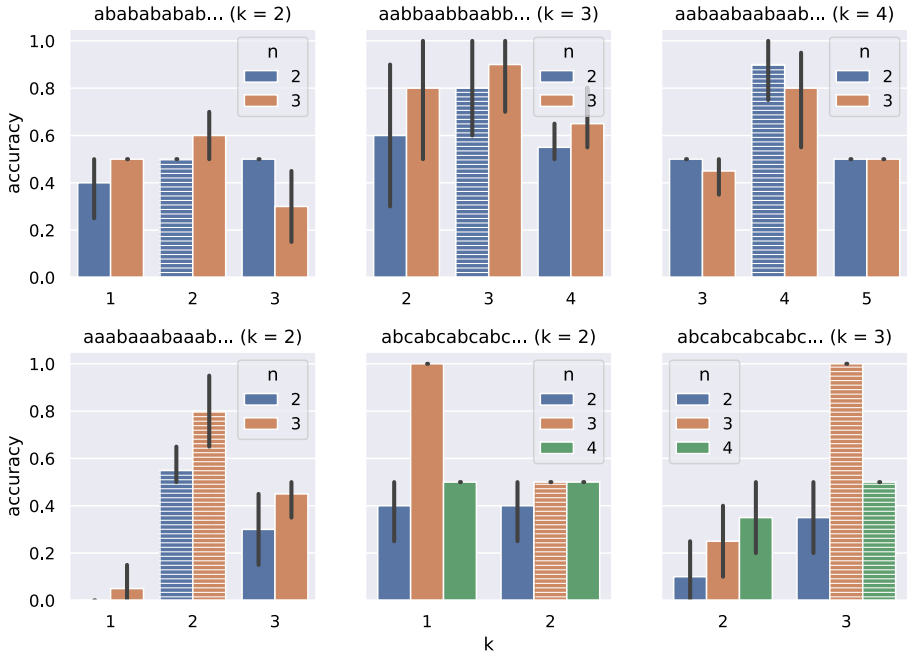
(b) *Latent state LSTM*

Fig. 23. The results of the neural baselines, a) the *vanilla LSTM*, and b) the *latent state LSTM* trained on a single sequence of the *Fuzzy Sequences* task. Striped bars denote the correct (n, k) possibilities, as in Fig. 22.

being pushed up by o_1. As well as being able to interpret the fluent properties and relations, we can also interpret how the system connects the raw perceptual input to the persistent objects at each moment. In Fig. 11, for example, we can see how subregions of the 20×20 pixel array correspond to particular persistent objects.



(a) *Vanilla LSTM*



(b) *Latent state LSTM*

Fig. 24. The results of the neural baselines, a) the *vanilla LSTM*, and b) the *latent state LSTM* trained on 1000 sequences of the *Fuzzy Sequences* task. Striped bars denote the correct (n, k) possibilities, as in Fig. 22.

Third, we can understand the *general dynamics* that the system believes hold universally (for all objects, and for all times). The engine is designed to satisfy the Kant-inspired constraint that whenever something changes, there must be a general universal law that explains that change: there is no change that is not intelligible. When we inspect the synthesised laws, we understand how the system believes properties change over time.

For example, the fifth rule learned for *Sokoban* is:

$$in_1(X, C_1) \wedge in_2(Y, C_2) \wedge below(C_1, C_2) \wedge action(south) \rightarrow p_3(Y)$$

Now p_3 is an invented predicate; its meaning is not apparent just from this single rule. But if we look at the other rule in which p_3 figures:

$$p_3(Y) \wedge in_2(Y, C_1) \wedge below(C_1, C_2) \supset in_2(Y, C_2)$$

we can see that p_3 is being used to represent that a block is being pushed south. Now we can understand the rule whose head is p_3 as: when the *south* action is performed, when the man X is above a block Y, then Y is pushed downwards.

It must be acknowledged, however, that the theories produced by the Apperception Engine are only readable by a small subset of humans—those comfortable reading logic programs. Whenever we say that a system is "interpretable", we mean interpretable for a particular audience in a particular context. So, although we have provided evidence that the system is interpretable for *some* people, there is much work to do to provide interpretations accessible to a wider audience.

As a logic program increases in size, and as the number of invented predicates increases, the program becomes less and less interpretable [77]. However, logic programs are – we claim – more interpretable than their procedural counterparts because they are so *concise*. A single Datalog clause is a powerful computational construct: each quantified variable in the clause represents a single for-loop in a procedural language. In an evaluation of program-verification tasks, a Datalog program was found to be up to two orders of magnitude shorter than its Java counterpart [78].

6.1.2. Accuracy

The Apperception Engine attempts to discern the nomological structure that underlies the raw sensory input. In our experiments, we found the induced theory to be very accurate as a predictive model, no matter how many time steps into the future we predict. For example, in *Seek Whence* (Section 5.1), the theory induced in Fig. 5a allows us to predict all future time steps of the series, and the accuracy of the predictions does not decay with time.

In *Sokoban* (Section 5.2), the learned dynamics are not just 100% correct on all test trajectories, but they are *provably* 100% correct. These laws apply to all *Sokoban* worlds, no matter how large, and no matter how many objects. Our system is, to the best of our knowledge, the first that is able to go from raw video of non-trivial games to an explicit first-order nomological model that is provably correct.

In the noisy sequences experiments (Section 5.3), the induced theory is an accurate predictive model. In Fig. 19, for example, the induced theory allows us to predict all future time steps of the series, and does not degenerate as we go further into the future.

6.1.3. Data efficiency

Neural nets are able to solve some sequence induction IQ tasks from raw input when trained on a sufficiently large number of training examples [79]. Neural nets are also able to learn the dynamics of *Sokoban* from raw input, when trained on a sufficiently large number of episodes [80].

But these models are notoriously data-hungry. In comparison with humans, who are often capable of learning concepts from a handful of data [81], artificial neural networks need thousands or millions of examples to reach human performance.

The Apperception Engine, by contrast, is much more data-efficient. While a neural network needs millions of trajectories to achieve reasonable accuracy on *Sokoban* [31], our system is able to learn a perfectly accurate model from a *single* trajectory. While a neural network needs hundreds of thousands of examples to achieve human-level performance on Raven's Progressive Matrices [79], our system is able to discern a pattern from a *single* sequence. The reason for our system's unusual data efficiency is the strong (but domain-independent) inductive bias that we inject via the $Datalog^{\supset}$ language (Definition 2) and the unity constraints (Definition 5).

A system that can learn an accurate dynamics model from a handful of examples is extremely useful for model-based reinforcement learning. Standard model-free algorithms require millions of episodes before they can reach human performance on a range of tasks [31]. Algorithms that learn an implicit model are able to solve the same tasks in thousands of episodes [82]. But a system that learns an accurate dynamics model from a handful of examples should be able to apply that model to *plan*, anticipating problems in imagination rather than experiencing them in reality [83], thus opening the door to extremely sample efficient model-based reinforcement learning. We anticipate a system that can learn the dynamics of an ATARI game from a handful of trajectories,[19] and then apply that model to plan, thus playing at reasonable human level *on its very first attempt*.

6.1.4. Summary

We can see, then, that a number of problems that have dogged neural networks since their very conception are solved or finessed when we move to a hybrid neuro-symbolic architecture that combines low-level perception with high-level apperception.

[19] Some complex games, such as *Montessori's Revenge*, will need more trajectories, and active exploration, in order to discern all the game dynamics.

The Apperception Engine inherits the traditional advantages of Inductive Logic Programming methods, in being data-efficient, generalising well, and supporting continual learning. But our system has two key features which distinguish it from standard ILP. First, it does not require human-labelled training data, but works with unsupervised sequences of sensory input. Second, it does not expect its input in pre-processed symbolic form; rather, it is able to work with raw unprocessed sensory input (e.g. noisy pixels).

6.2. What makes it work

What is it about the architecture that enables the Apperception Engine to satisfy the desiderata listed above? We identify three features which are critical to its success: (i) the declarative logic programming language that is used as the target language for program synthesis, (ii) the strong inductive bias injected into the system, and (iii) the hybrid architecture that combines binary neural networks with symbolic program synthesis.

6.2.1. The declarative logic programming language $Datalog^{\ni}$

When designing a program synthesis system, a critical decision is: what form should the target language take? Our target language, $Datalog^{\ni}$, has two features which we regard as critical to the system's success.

First, the language is very *concise*. A single datalog clause is a powerful computational construct: each quantified variable in the clause represents a single for-loop in a procedural language. In an evaluation of program-verification tasks, a datalog program was found to be two orders of magnitude shorter than its Java counterpart [78]. Concision is very important in program synthesis: the search space of programs considered is b^n where b is the mean branching factor and n is the program length. Thus, a concise language (in which n is shorter) is much more tractable for search [84] The conciseness of $Datalog^{\ni}$ is a key feature allowing us to synthesise theories for non-trivial domains (see the experiments in Sections 5.1.1, 5.2, and 5.3). If we had used a less concise target language, we would not have been able to solve these problems.

The second critical feature of $Datalog^{\ni}$ is that the language is *declarative*. The constituents of $Datalog^{\ni}$ programs are individual clauses. Each clause can be interpreted separately as a *judgement* that makes a distinctive claim about the world. Of course, the meaning of one clause depends on the set of clauses in which it is embedded, but (given its embedding context) a single clause still has a unique meaning as a particular claim about the world.

Contrast this *declarative decomposability* of datalog with the procedural case: in an imperative program, the constituents are procedures, not clauses, and a procedure cannot be interpreted as a judgement with a truth-condition—a procedure is just a recipe for getting something done. The declarative decomposability of $Datalog^{\ni}$ was critical to the interpretations of Sections 5.1.3, 5.2.3, and 5.3.4.

Michalski [85] was well aware of the importance of declarative decomposability:

> The results of computer induction should be symbolic descriptions of given entities, semantically and structurally similar to those a human expert might produce observing the same entities. *Components of these descriptions should be comprehensible as single 'chunks' of information*, directly interpretable in natural language, and should relate quantitative and qualitative concepts in an integrated fashion

6.2.2. Our inductive bias is essential to data-efficient learning

In each of our experiments, the Apperception Engine is shown to be significantly more data-efficient than the neural network baselines. This data efficiency is only possible because of the significant inductive bias that has been injected into the system. This inductive bias involves three main aspects.

First, there is inductive bias in the *form of clauses* that are allowed in the $Datalog^{\ni}$ language. The only rules that the system is allowed to produce are general rules that quantify over all objects and all times. The system is simply incapable of formulating a rule that applies only to a particular individual, or only to a particular time. In other words, the system is *doomed to generalise*. This inductive bias comes from Kant. He argued that all judgements are universal (apply to all objects).[20] In Kant's cognitive architecture, there is no such thing as a specific judgement. Our system respects this Kantian restriction.[21] Although our system can only construct universally quantified rules, it is capable of constructing complex theories that treat different cases differently. But the simplicity prior in Equation (12) means we prefer theories with a shorter description length, all other things being equal.

The second form of inductive bias is the introduction of *persistent objects*.[22] The system is forced to reinterpret the ephemeral play of transitory sense data as a re-presentation of a set of persistent objects, with properties that change over time. Again, this inductive bias is inspired by Kant.[23]

[20] This follows directly from two central claims, that judgements are rules [*Prolegomena* 4:305], and that rules, in turn, are "the representation of a universal condition" [*Critique of Pure Reason*, A113].

[21] LFIT has a similar inductive bias [86].

[22] The introduction of persistent objects is inevitable in domains like *Sokoban*. But it is notable that, even in domains like *Seek Whence* that do not feature persistent objects at the surface, it is the ability to posit latent persistent objects, with properties that change over time, that is needed to make sense of the sequences.

[23] See the *Critique of Pure Reason* [A182/B224] ff.

The third form of inductive bias is the *unity conditions on an acceptable theory* (Definition 5). These include spatial unity, conceptual unity, static unity, and temporal unity. These constraints, again, are inspired by Kant's discussion in the *Critique of Pure Reason*.[24]

The standard objection to inductive bias, of course, is that although it helps the system learn efficiently in certain domains, the same bias also prevents the system learning effectively in others. According to this objection, inductive bias must be *domain-specific* bias that can only help performance in some domains while hindering performance in others.[25]

We do not accept this argument. The inductive bias we inject is intended to be maximally general bias that applies to all domains that we can understand. The general assumptions we make—that the world is composed of persistent objects, that changes to objects must be covered by general explanatory rules, and so on—are not domain-specific insights but rather general insights about any situation that we are capable of making sense of.

6.2.3. Our hybrid neuro-symbolic architecture is essential for learning explicit theories from raw data

Our hybrid architecture allows both neural networks and symbolic program synthesis methods to play to their respective strengths. It has often been noted that artificial neural networks and inductive logic programming have complementary strengths and weaknesses [46]: neural networks are robust to noisy and ambiguous data, but are data inefficient and inscrutable. Inductive logic programming approaches to machine learning, by contrast, are data-efficient and provide interpretable models, but do not easily handle noisy data,[26] let alone ambiguous data. Our hybrid architecture attempts to combine the best of both worlds, using a neural network to map noisy ambiguous raw input to discrete concepts, and using program synthesis to generate interpretable models from handfuls of data.

The overall architecture, because it represents both the binary neural network and the unsupervised program synthesis system as a single ASP program, allows information to flow both ways: both bottom-up and top-down. Because the weights of the neural network and the rules of the Apperception Engine are found jointly by a single SAT problem, considerations from one are able to influence the other. In the bottom-up direction, considerations from the binary neural network used for low-level perception are able to influence the rules formed by the symbolic system used for high-level apperception. In the top-down direction, considerations from the rules formed by the symbolic system can influence the sorts of typifications produced by the neural network.

6.3. Limitations and further work

We highlight three types of limitation: (i) expressive limitations of the *Datalog*$^{\ni}$ language, (ii) performance limitations of the program synthesis system, and (iii) limitations of the system as a whole to provide useful human-readable explanations.

6.3.1. Expressive limitations of Datalog$^{\ni}$

The Apperception Engine generates theories expressed in *Datalog*$^{\ni}$. Thus, limitations of the expressive power of this target language restrict the types of theories that can be formed and the types of domains that can be understood.

A fundamental limitation of using *Datalog*$^{\ni}$ is that it requires that the underlying dynamics can be expressed as rules that operate on *discrete* concepts. While the system is capable of handling raw, noisy, continuous sensory input, it assumes that the *underlying dynamics* of the system can be represented by rules that operate on discrete concepts. There are many domains where the underlying dynamics are discrete while the surface output is noisy and continuous: Raven's progressive matrices, puzzle games, and ATARI video games, for example. But our system will struggle in domains where the underlying dynamics are best modelled using continuous values, such as models of fluid dynamics. Here, the best our system could do is find a discrete model that crudely approximates the true continuous dynamics. Extending *Datalog*$^{\ni}$ to represent continuous change would be a substantial and ambitious project.

Another major limitation of *Datalog*$^{\ni}$ is that it assumes that causal rules are strict, universal and exceptionless. There is no room in the current representation for defeasible causal rules (where normally, all other things being equal, a causes b) or non-deterministic causal rules (where a causes either b or c). In future work, we plan to implement non-deterministic causal rules either by synthesising normal logic programs under the stable model semantics or by synthesising disjunctive rules.

6.3.2. Performance limitations of the program synthesis system

In our approach, making sense of sensory input means finding a theory that explains that input. Finding a theory means searching through the space of logic programs. This is a huge and daunting task. For example, the Apperception Engine takes 5 GB of RAM and 48 hours on a standard 4-core Unix desktop to make sense of a single *Sokoban* trajectory consisting of 17 pixel arrays of size 20×20. This is, undeniably, a computationally expensive process.

We would like to scale our approach up so that we can learn the dynamics of ATARI games from raw pixels. But this will prove to be challenging, as games such as Pacman are substantially harder than our *Sokoban* test-case in every dimension:

[24] For spatial and temporal unity, see the *Schematism*. For static and temporal unity, see the *Principles*.

[25] For thoughtful discussions of the "no free lunch theorem" see [87] and [88].

[26] There have been some notable recent attempts to address this [89].

it requires us to increase the number of pixels, the number of colours, the number of time-steps, the number of trajectories, the number of objects, and the complexity of the dynamics.

The dominant reason for our system's scaling difficulties is that it uses a maximising SAT solver to search through the space of logic programs. Finding an optimal solution to an ASP program with weak constraints is in Σ_2^P; but this complexity is a function of the number of ground atoms, and the number of ground atoms of our ASP program is exponential in the length of the *Datalog*$^{\exists}$ program we are synthesising.

We are considering various different ways of improving the performance of our system so we can scale up to harder problems such as ATARI. One option is to focus on optimising the ASP representation, for example by pruning the grounding size by using the HEX extension of ASP [90]. Another option is to consider entirely different ways of synthesising logic programs, for example by using a neural network to guide the search through the space of logic programs [91,92].[27]

We are excited by the prospect of scaling up the Apperception Engine to the next level, in order to induce robust causal models for ATARI, 3D simulations, and other complex worlds. But it will, we believe, require substantial further research.

6.3.3. Limitations of the Apperception Engine as an explanatory system

In Sections 5.1.3, 5.2.3, and 5.3.4, we tried to show how the Apperception Engine interpreted a particular problem. By displaying the ground atoms representing each moment in time, and by displaying the *Datalog*$^{\exists}$ rules holding over all moments in time, we attempted to show what it was "thinking", to explain why it made the predictions that it did.

These explanations are, perhaps, acceptable to a particular audience—technical people who are comfortable interpreting logic programs. But different audiences have different requirements and competencies. There is much further work to do to make these interpretations accessible to a wider audience.

We would also like, in further work, to evaluate empirically how effective the interpretations provided are, by testing whether the machine's interpretations are able to help humans to understand the domain. In a recent paper, Muggleton *et al.* [93] showed that when human subjects were given the output of an ILP system, and were able to read and interpret those results, their performance in a task markedly increased. We would like to try a similar experiment with the Apperception Engine on the *Seek Whence* tasks. We plan to give a subset of subjects the theory generated by the engine, and see if they are able to use the theory to predict the continuations more accurately.

6.4. Concepts

We conclude by discussing how the Apperception Engine represents concepts.

What does it mean to understand a concept? When we claim that a particular agent understands a particular concept, what exactly are we attributing to it?

In Robert Brandom's monumental *Making It Explicit* [94], he provides an **inferentialist**[28] interpretation of concept understanding, in which an agent understands a concept when both the following conditions are satisfied:

1. it knows when to apply the concept; in other words, it knows the **circumstances of application**
2. it knows the inferential commitments of applying the concept; in other words, it knows the **consequences of application**

For example, an agent understands the concept "red" if:

1. it is able, when confronted with objects that are red, to apply the concept "red" to them
2. it understands the inferential consequences of saying that something is red: it knows that no (monochromatic) red object is also blue, that red objects are coloured, that crimson objects are red, and so on.

Both of these capacities are required. Neither on its own is sufficient for concept understanding.

Consider, for example, a parrot that has been trained to utter "red" when it sees something that looks red. The parrot knows when to apply the concept, thus satisfying the first of the two conditions for concept understanding. But it does not know the consequences of applying the concept: it does not know that "red" and "blue" are incompatible, that red things are coloured, and so on.

Or consider, for example, Frank Jackson's famous thought experiment:

> Mary is a brilliant scientist who is, for whatever reason, forced to investigate the world from a black and white room via a black and white television monitor. She specializes in the neurophysiology of vision and acquires, let us suppose, all the physical information there is to obtain about what goes on when we see ripe tomatoes, or the sky, and use terms like "red", "blue", and so on.

[27] We cannot just apply these techniques "out of the box", however, because our problem is unsupervised.

[28] Inferentialism comes to us from Wilfrid Sellars who in turn was attempting to rearticulate Kant's vision of concept understanding.

Now Mary knows the inferential consequences of "red". In fact, as a leading neurophysiologist, she understands the inferential consequences of colour concepts better than anyone. But, as she has spent all her life in a black and white room, she does not yet know when to apply the concept "red". If she opens the door and is confronted with a red colour patch, she will not immediately know what colour it is.

We can use this two-aspect inferentialist interpretation of concept understanding to diagnose the limitations of both connectionism and symbolic AI. The trouble with connectionism, according to the inferentialist, is that it focuses only on the circumstances of application, while ignoring the equally important consequences of application. A neural network can be trained to emit "dog" when presented with an image of a dog, but it does not know that all dogs are mammals, that no dog is also a cat, or that corgis are a type of dog. The trouble with symbolic AI, according to the inferentialist, is that it focuses only on the consequences of application—the inferential relations between concepts—while ignoring the equally crucial circumstances of application. This criticism applies to good old-fashioned AI (GOFAI) as well as more modern forms of symbolic AI such as inductive logic programming. In traditional GOFAI, a human hand-engineers the logical rules describing the inferential connections between concepts, while in inductive logic programming, the system constructs the rules itself. But in both cases, the symbolic system does not have a way of mapping raw perceptual input onto concepts. If we want to build a concept understanding system, then, we will need the system to understand *both* the circumstances of application and the consequences of application. The Apperception Engine, when connected to a neural network in the manner described above, is an attempt to realise both aspects of the inferentialist's interpretation of concept understanding.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A. Supplementary material

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.artint.2021.103521.

References

[1] R. Evans, J. Hernandez-Orallo, J. Welbl, P. Kohli, M. Sergot, Making sense of sensory input, preprint, arXiv:1910.02227, 2019.
[2] T. Miller, Explanation in artificial intelligence: insights from the social sciences, Artif. Intell. (2018).
[3] C. Rudin, Please stop explaining black box models for high stakes decisions, preprint, arXiv:1811.10154, 2018.
[4] W.J. Murdoch, C. Singh, K. Kumbier, R. Abbasi-Asl, B. Yu, Interpretable machine learning: definitions, methods, and applications, Proc. Natl. Acad. Sci. USA (2019).
[5] Y. LeCun, Personal communication on Facebook, https://www.facebook.com/yann.lecun/posts/10153426023477143.
[6] P. Sermanet, C. Lynch, Y. Chebotar, J. Hsu, E. Jang, S. Schaal, S. Levine, Time-contrastive networks: self-supervised learning from pixels, in: 2018 IEEE International Conference on Robotics and Automation, ICRA, IEEE, 2018, pp. 1134–1141.
[7] D. Pathak, P. Agrawal, A.A. Efros, T. Darrell, Curiosity-driven exploration by self-supervised prediction, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops, 2017, pp. 16–17.
[8] D.R. Hofstadter, Fluid Concepts and Creative Analogies, Basic Books, 1995.
[9] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, et al., Gradient-based learning applied to document recognition, Proc. IEEE 86 (11) (1998) 2278–2324.
[10] J. Schmidhuber, Making the world differentiable: on using self-supervised fully recurrent neural networks for dynamic reinforcement learning and planning in non-stationary environments, http://people.idsia.ch/~juergen/FKI-126-90_(revised)bw_ocr.pdf.
[11] D. Ha, J. Schmidhuber, World models, preprint, arXiv:1803.10122, 2018.
[12] R.M. Smullyan, First-Order Logic, Courier Corporation, 1995.
[13] R. Kowalski, Predicate logic as programming language, in: IFIP Congress, vol. 74, 1974, pp. 544–569.
[14] K.R. Apt, Logic programming, in: Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics (B), 1990, pp. 493–574.
[15] J.W. Lloyd, Foundations of Logic Programming, Springer Science & Business Media, 2012.
[16] R.E. Kalman, A new approach to linear filtering and prediction problems, J. Basic Eng. 82 (1) (1960) 35–45.
[17] F. Black, M. Scholes, The pricing of options and corporate liabilities, J. Polit. Econ. 81 (3) (1973) 637–654.
[18] A. Graves, Generating sequences with recurrent neural networks, preprint, arXiv:1308.0850, 2013.
[19] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A.N. Gomez, Ł. Kaiser, I. Polosukhin, Attention is all you need, in: Advances in Neural Information Processing Systems, 2017, pp. 5998–6008.
[20] J. Devlin, M.-W. Chang, K. Lee, K. Toutanova, Bert: pre-training of deep bidirectional transformers for language understanding, in: Proceedings of NAACL-HLT, 2018.
[21] L. Kaiser, M. Babaeizadeh, P. Milos, B. Osinski, R.H. Campbell, K. Czechowski, D. Erhan, C. Finn, P. Kozakowski, S. Levine, et al., Model-based reinforcement learning for Atari, preprint, arXiv:1903.00374, 2019.
[22] N. Tomašev, X. Glorot, J.W. Rae, M. Zielinski, H. Askham, A. Saraiva, A. Mottram, C. Meyer, S. Ravuri, I. Protsyuk, et al., A clinically applicable approach to continuous prediction of future acute kidney injury, Nature 572 (7767) (2019) 116–119.
[23] B. Uria, I. Murray, H. Larochelle, Rnade: the real-valued neural autoregressive density-estimator, in: Advances in Neural Information Processing Systems, 2013, pp. 2175–2183.
[24] J. Oh, X. Guo, H. Lee, R.L. Lewis, S. Singh, Action-conditional video prediction using deep networks in Atari games, in: Advances in Neural Information Processing Systems, 2015, pp. 2863–2871.
[25] A.v.d. Oord, N. Kalchbrenner, K. Kavukcuoglu, Pixel recurrent neural networks, in: Proceedings of International Conference on Machine Learning, 2016.
[26] A. v, d. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, K. Kavukcuoglu, Wavenet: a generative model for raw audio, preprint, arXiv:1609.03499, 2016.
[27] W. Xiong, W. Luo, L. Ma, W. Liu, J. Luo, Learning to generate time-lapse videos using multi-stage dynamic generative adversarial networks, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2018, pp. 2364–2373.

[28] J.C. Loehlin, Latent variable models: an introduction to factor, path, and structural analysis, Lawrence Erlbaum Associates, Inc, 1987.
[29] Z. Ghahramani, M.I. Jordan, Factorial hidden Markov models, in: Advances in Neural Information Processing Systems, 1996, pp. 472–478.
[30] A. Kosiorek, H. Kim, Y.W. Teh, I. Posner, Sequential attend, infer, repeat: generative modelling of moving objects, in: Advances in Neural Information Processing Systems, 2018, pp. 8606–8616.
[31] L. Buesing, T. Weber, S. Racaniere, S. Eslami, D. Rezende, D.P. Reichert, F. Viola, F. Besse, K. Gregor, D. Hassabis, et al., Learning and querying fast generative models for reinforcement learning, preprint, arXiv:1802.03006, 2018.
[32] D. Hafner, T. Lillicrap, I. Fischer, R. Villegas, D. Ha, H. Lee, J. Davidson, Learning latent dynamics for planning from pixels, in: Proceedings of the International Conference on Machine Learning, 2018.
[33] M.Y. Byron, J.P. Cunningham, G. Santhanam, S.I. Ryu, K.V. Shenoy, M. Sahani, Gaussian-process factor analysis for low-dimensional single-trial analysis of neural population activity, in: Advances in Neural Information Processing Systems, 2009, pp. 1881–1888.
[34] M.B. Ahrens, M.B. Orger, D.N. Robson, J.M. Li, P.J. Keller, Whole-brain functional imaging at cellular resolution using light-sheet microscopy, Nat. Methods 10 (5) (2013) 413.
[35] J. Li, W. Monroe, T. Shi, S. Jean, A. Ritter, D. Jurafsky, Adversarial learning for neural dialogue generation, in: Proceedings of the Conference on Empirical Methods in Natural Language Processing, 2017.
[36] A.v.d. Oord, Y. Li, O. Vinyals, Representation learning with contrastive predictive coding, preprint, arXiv:1807.03748, 2018.
[37] D.P. Kingma, M. Welling, Auto-encoding variational Bayes, in: Proceedings of the International Conference on Learning Representations, 2013.
[38] J. Chung, K. Kastner, L. Dinh, K. Goel, A.C. Courville, Y. Bengio, A recurrent latent variable model for sequential data, in: Advances in Neural Information Processing Systems, 2015, pp. 2980–2988.
[39] M.J. Johnson, D.K. Duvenaud, A. Wiltschko, R.P. Adams, S.R. Datta, Composing graphical models with neural networks for structured representations and fast inference, in: Advances in Neural Information Processing Systems, 2016, pp. 2946–2954.
[40] A. Mnih, D.J. Rezende, Variational inference for Monte Carlo objectives, in: Proceedings of the International Conference on Machine Learning, 2016.
[41] A. van den Oord, O. Vinyals, et al., Neural discrete representation learning, in: Advances in Neural Information Processing Systems, 2017, pp. 6306–6315.
[42] E. Jang, S. Gu, B. Poole, Categorical reparameterization with Gumbel-Softmax, in: Proceedings of the International Conference on Learning Representations, 2016.
[43] C.J. Maddison, A. Mnih, Y.W. Teh, The concrete distribution: a continuous relaxation of discrete random variables, in: Proceedings of the International Conference on Learning Representations, 2016.
[44] T. Kurutach, A. Tamar, G. Yang, S.J. Russell, P. Abbeel, Learning plannable representations with causal InfoGAN, in: Advances in Neural Information Processing Systems, 2018, pp. 8733–8744.
[45] A. Zhang, A. Lerer, S. Sukhbaatar, R. Fergus, A. Szlam, Composable planning with attributes, in: Proceedings of the International Conference on Machine Learning, 2018.
[46] R. Evans, E. Grefenstette, Learning explanatory rules from noisy data, J. Artif. Intell. Res. 61 (2018) 1–64.
[47] T.R. Besold, A. d. Garcez, S. Bader, H. Bowman, P. Domingos, P. Hitzler, K.-U. Kühnberger, L.C. Lamb, D. Lowd, P.M.V. Lima, et al., Neural-symbolic learning and reasoning: a survey and interpretation, preprint, arXiv:1711.03902, 2017.
[48] A. d. Garcez, M. Gori, L.C. Lamb, L. Serafini, M. Spranger, S.N. Tran, Neural-symbolic computing: an effective methodology for principled integration of machine learning and reasoning, preprint, arXiv:1905.06088, 2019.
[49] L. De Raedt, S. Dumančić, R. Manhaeve, G. Marra, From statistical relational to neuro-symbolic artificial intelligence, preprint, arXiv:2003.08316, 2020.
[50] S. Hölldobler, Y. Kalinke, F.W. Ki, et al., Towards a new massively parallel computational model for logic programming, in: ECAI'94 Workshop on Combining Symbolic and Connectionist Processing, Citeseer, 1991.
[51] A.S.A. Garcez, G. Zaverucha, The connectionist inductive learning and logic programming system, Appl. Intell. 11 (1) (1999) 59–77.
[52] A. d. Garcez, L.C. Lamb, D.M. Gabbay, A connectionist inductive learning system for modal logic programming, in: Proceedings of the 9th International Conference on Neural Information Processing, vol. 4, ICONIP'02, 2002, IEEE, 2002, pp. 1992–1997.
[53] A.S. d'Avila Garcez, L.C. Lamb, D.M. Gabbay, Neural-symbolic intuitionistic reasoning, in: Design and Application of Hybrid Intelligent Systems, 2003, pp. 399–408.
[54] S. Hölldobler, Y. Kalinke, H.-P. Störr, Approximating the semantics of logic programs by recurrent neural networks, Appl. Intell. 11 (1) (1999) 45–58.
[55] S. Bader, P. Hitzler, S. Hölldobler, Connectionist model generation: a first-order approach, Neurocomputing 71 (13–15) (2008) 2420–2432.
[56] L. Serafini, A.d. Garcez, Logic tensor networks: deep learning and logical reasoning from data and knowledge, preprint, arXiv:1606.04422, 2016.
[57] S. Džeroski, L. De Raedt, K. Driessens, Relational reinforcement learning, Mach. Learn. 43 (1–2) (2001) 7–52.
[58] V. Zambaldi, D. Raposo, A. Santoro, V. Bapst, Y. Li, I. Babuschkin, K. Tuyls, D. Reichert, T. Lillicrap, E. Lockhart, et al., Relational deep reinforcement learning, preprint, arXiv:1806.01830, 2018.
[59] B. Longuenesse, Kant and the Capacity to Judge, Princeton UP, Princeton, 1998.
[60] R. Morel, A. Cropper, L. Ong, Typed meta-interpretive learning of logic programs, in: Proceedings of JELIA 2019, 2019, pp. 973–981.
[61] F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, F. Ricca, T. Schaub, Asp-core-2: Input Language Format, ASP Standardization Working Group, 2012.
[62] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Clingo = asp+ control: preliminary report, preprint, arXiv:1405.3694, 2014.
[63] Y. LeCun, Y. Bengio, et al., Convolutional networks for images, speech, and time series, The Handbook of Brain Theory and Neural Networks 3361 (10) (1995) 1995.
[64] S.C. Zhu, Y.N. Wu, D. Mumford, Minimax entropy principle and Its application to texture modelling, Neural Comput. 9 (8) (1997) 1627–1660.
[65] K. Ellis, D. Ritchie, A. Solar-Lezama, J. Tenenbaum, Learning to infer graphics programs from hand-drawn images, in: Advances in Neural Information Processing Systems, 2018, pp. 6059–6068.
[66] L. Tian, K. Ellis, M. Kryven, J. Tenenbaum, Learning abstract structure for drawing by efficient motor program induction, Adv. Neural Inf. Process. Syst. 33 (2020).
[67] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, Y. Bengio, Binarized neural networks, in: Advances in Neural Information Processing Systems, 2016, pp. 4107–4115.
[68] M. Kim, P. Smaragdis, Bitwise neural networks, preprint, arXiv:1601.06071, 2016.
[69] M. Rastegari, V. Ordonez, J. Redmon, A. Farhadi, XNOR-Net: ImageNet classification using binary convolutional neural networks, in: European Conference on Computer Vision, Springer, 2016, pp. 525–542.
[70] C.-H. Cheng, G. Nührenberg, C.-H. Huang, H. Ruess, Verification of binarized neural networks via inter-neuron factoring, in: Working Conference on Verified Software: Theories, Tools, and Experiments, Springer, 2018, pp. 279–290.
[71] N. Narodytska, S. Kasiviswanathan, L. Ryzhyk, M. Sagiv, T. Walsh, Verifying properties of binarized deep neural networks, in: Thirty-Second AAAI Conference on Artificial Intelligence, 2018, pp. 6615–6624.
[72] M. Li, P. Vitányi, An Introduction to Kolmogorov Complexity and its Applications, vol. 3, Springer, 2008.
[73] S. Hochreiter, J. Schmidhuber, Long short-term memory, Neural Comput. 9 (8) (1997) 1735–1780.
[74] D.P. Kingma, J. Ba Adam, A method for stochastic optimization, in: Proceedings of the International Conference for Learning Representations, 2015.
[75] A. Cropper, R. Evans, M. Law, Inductive general game playing, preprint, arXiv:1906.09627, 2019.

[76] I. Higgins, L. Matthey, A. Pal, C. Burgess, X. Glorot, M. Botvinick, S. Mohamed, A. Lerchner, beta-VAE: learning basic visual concepts with a constrained variational framework, ICLR 2 (5) (2017) 6.
[77] U. Schmid, C. Zeller, T. Besold, A. Tamaddoni-Nezhad, S. Muggleton, How does predicate invention affect human comprehensibility?, in: International Conference on Inductive Logic Programming, Springer, 2016, pp. 52–67.
[78] J. Whaley, D. Avots, M. Carbin, M.S. Lam, Using datalog with binary decision diagrams for program analysis, in: Asian Symposium on Programming Languages and Systems, Springer, 2005, pp. 97–118.
[79] D.G. Barrett, F. Hill, A. Santoro, A.S. Morcos, T. Lillicrap, Measuring abstract reasoning in neural networks, preprint, arXiv:1807.04225, 2018.
[80] S. Racanière, T. Weber, D. Reichert, L. Buesing, A. Guez, D.J. Rezende, A.P. Badia, O. Vinyals, N. Heess, Y. Li, et al., Imagination-augmented agents for deep reinforcement learning, in: Advances in Neural Information Processing Systems, 2017, pp. 5690–5701.
[81] B.M. Lake, R. Salakhutdinov, J.B. Tenenbaum, Human-level concept learning through probabilistic program induction, Science 350 (6266) (2015) 1332–1338.
[82] L. Kaiser, M. Babaeizadeh, P. Milos, Model based reinforcement learning for Atari, preprint, arXiv:1903.00374v2, 2019.
[83] D. Ha, J. Schmidhuber, Recurrent world models facilitate policy evolution, in: Advances in Neural Information Processing Systems, 2018, pp. 2455–2467.
[84] A. Cropper, Playgol: learning programs through play, preprint, arXiv:1904.08993, 2019.
[85] R.S. Michalski, A theory and methodology of inductive learning, in: Machine Learning, Springer, 1983, pp. 83–134.
[86] K. Inoue, T. Ribeiro, C. Sakama, Learning from interpretation transition, Mach. Learn. 94 (1) (2014) 51–79.
[87] T. Lattimore, M. Hutter, No free lunch versus Occam's razor in supervised learning, in: Algorithmic Probability and Friends. Bayesian Prediction and Artificial Intelligence, Springer, 2013, pp. 223–235.
[88] T. Everitt, T. Lattimore, M. Hutter, Free lunch for optimisation under the universal distribution, in: 2014 IEEE Congress on Evolutionary Computation, CEC, IEEE, 2014, pp. 167–174.
[89] S. Muggleton, W.-Z. Dai, C. Sammut, A. Tamaddoni-Nezhad, J. Wen, Z.-H. Zhou, Meta-interpretive learning from noisy images, Mach. Learn. 107 (7) (2018) 1097–1118.
[90] T. Kaminski, T. Eiter, K. Inoue, Meta-interpretive learning using hex-programs, in: Proceedings of the 28th International Joint Conference on Artificial Intelligence, AAAI Press, 2019, pp. 6186–6190.
[91] M. Balog, A.L. Gaunt, M. Brockschmidt, S. Nowozin, D. Tarlow, Deepcoder: learning to write programs, preprint, arXiv:1611.01989, 2016.
[92] T. Pierrot, G. Ligner, S. Reed, O. Sigaud, N. Perrin, A. Laterre, D. Kas, K. Beguir, N. de Freitas, Learning compositional neural programs with recursive tree search and planning, preprint, arXiv:1905.12941, 2019.
[93] S.H. Muggleton, U. Schmid, C. Zeller, A. Tamaddoni-Nezhad, T. Besold, Ultra-strong machine learning: comprehensibility of programs learned with ILP, Mach. Learn. 107 (7) (2018) 1119–1140.
[94] R. Brandom, Making It Explicit, Harvard University Press, 1994.
[95] I. Kant, Critique of Pure Reason, Cambridge University Press, 1781.