Learning Within the World:

A Definition of Learning, of Physical World Learning, and a Machine for Creating Knowledge.

tiplur-bilrex

1. Introduction:

Karl Popper is one of numerous philosophers to put forth the argument that inductive probability is impossible: that observed ‘evidence’ cannot probabilistically support a defined ‘theory’ [PM83]. Since then, the field of Machine Learning has had continuing success at generating solutions to fit various predefined problems through application of probabilistic Artificial Neural Networks (ANNs). However, the field has had subjectively less success at generating the kind of discrete knowledge that is familiar in the discipline of science.

To address these issues, I propose a minimally constrained, non-probabilistic, definition of learning, that uses sequential evaluations of the relative sizes of observed intersection sets between ‘knowledge’ definitions and an unknown target definition. I constrain the abstract learning definition to define learning as a physical process within our world. The added constraint is ‘physical existence’, which produces a corresponding optimization problem. I follow these definitions with a scheme for a solver that leverages a customized policy-based RL technique and a clock to generate improving knowledge within our world. I expect the proposed system will be able to run continuously without human direction and generate structured knowledge that resembles the knowledge generated by human science.

2. An Assumption Within Probabilistic Learning

Probability theory underlies a vast majority of machine learning algorithms [BN06]. These algorithms assume that the target function is stochastic by either implicitly or explicitly assuming the existence of a random variable in the definition of the target function. The inclusion of a random variable is essential for defining error as a scalar measure of the difference between the learned distribution and the true (target function) distribution. This difference measure is used to direct updates of a specific parameterized model [SS15].

An optimal abstract definition of learning should exclude any excess assumptions about the definition of the target function. Although an assumption that randomness exists is useful, its acceptance constrains the set of learnable knowledge that can be generated by these kinds of algorithms, and I suspect it is limiting progress toward improving our theories of knowledge. I believe statistical learning

theory is best defined as a subtype of a more foundational learning algorithm. This ‘foundational learning algorithm’ is absent from popular machine learning designs.

3. A Minimum Assumption Learning Algorithm

This definition of a Learning Algorithm (LA) assumes that (1) the best definition (knowledge) of a target function is the definition that has the largest set of intersecting points with the target function, and that (2) knowledge definitions must be evaluated using finite sets of ‘observed’ intersecting points. Because the target function may define an infinite set of input-output pairs, it is impossible to prove that any specific definition of it is correct; definitions can only be evaluated relative to one another and the observable set of input-output states generated by the target function.

The knowledge function, *f*, and the target function, *w*, produce an intersection set, $\{x \mid f(x)=w(x)\}$ where *x* represents input points. Using a finite set of states generated by the target function, the LA evaluates the sizes of intersection sets of candidate knowledge functions and selects the one with the largest intersection set. Learning is defined as the sequential selection of ‘better’ *f* definitions, having increasing intersection set sizes as new observed states computed by *w* are received.

Below is a formal definition of the learning computation over a finite set of observed states:

3.1. Formal Definition

Let D_{in} be a universal set of possible inputs and D_{out} be a universal set of possible outputs.

Let *E* be a set of observed input-output tuples computed by the target function, *w*.

Let *X* be the set of all unique inputs that appear as the first element in any tuple within *E:*

$$X=\{x|\exists\, y \text{ such that } (x,y)\in E\} \quad .$$

Therefore, I define *E* in terms of *w* and *X*:

$$E=\{(x,w(x))|x\in X\} \quad .$$

Let *f* be a candidate knowledge function, where:

$$f:\mathscr{P}(X)\rightarrow D_{out} \quad ,$$

Let *P* be the set of input-output tuples (predictions) generated by *f*:

$$P=\{(x,f(x))|x\in X\} \quad .$$

P requires that for any input-output tuple (x_n, $f_k(x_n)$) element of a prediction set P_k, the input x_n and the function f_k must not contain any values derived from observation values with indexes greater than n.

Let T be the number of identical input-output tuples that are present in both E and P; that is, the cardinality of the intersection of E and P:

$$T = |E \cap P| \quad .$$

Substituting the definitions of E and P:

$$T = |\{(x, w(x)) | x \in X\} \cap \{(x, f(x)) | x \in X\}| \quad .$$

Equivalently, T counts the number of inputs x in X for which the functions w and f produce the same output. This can be expressed using the indicator function $\boldsymbol{I}(\cdot)$, evaluating to 1 when the outputs of w and f are equal and 0 when they are not equal:

$$T = \sum_{x \in X} \boldsymbol{I}(w(x) = f(x)) \quad .$$

A function, f_k, may be defined as knowledge if its observed intersection set T_k exceeds 1. Learning is defined by a sequence of knowledge functions where the value of T_k for each sequential f_k increases.

Assuming there exists a set of candidate knowledge functions, F, where:

$$F = \{f_0, \ldots, f_k\} \quad ,$$

and each element, f_k, of the set has a corresponding intersection value, T_k; then I define a knowledge evaluator function, h, that returns the element of F with the largest corresponding T value:

$$h(F, \{T\}) = \phi^{-1}(max(\{T\})) \quad .$$

In addition to returning the function from F corresponding to max(T), when h evaluates a set of knowledge functions it returns a boolean state 1 (meaning h is satisfied by the new f definition) when the T value of a new f exceeds the T value of the previous satisfying f definition (for a given observation set E). If no new f definition has a T value exceeding the previous satisfying f definition, then the boolean state of h is 0 (h is not satisfied).

3.2. Implications of the Definition

For the definition of the abstract LA, one may assume candidate f definitions are randomly generated. Other than the information constraint imposed by the knowledge evaluator function, there is no constraint on the content of f. For example, f may be a single function like an ANN or a complex mapping of the observed state sequences to compositions of discrete function and input definitions. An example of the latter may be:


```
{"obs_id": "id_12",
  "func_name": "simulate_falling_object",
  "args": ["ref:obj_4_initial_velocity", "expr:(ref:id_8 + 4.0)"]},
```

where *f* is a function that interprets the above input and returns the corresponding output. The output will be evaluated against an appropriate state from *E*, such as the output with index 12, where $x_{12}=w(x_0, \ldots, x_{11})$.

E may be defined as a static set or as an incremental expanding set, increasing in size as new input-output tuples are generated by *w*. If *E* is expanding, we may define new input-output pairs added to *E'* as (*E*, w(*E*)). If the size of *E* continuously increases, new candidate knowledge functions are created and evaluated by *h*, and the 'best' knowledge function (returned by *h*) always remains within *F*, then learning will continuously progress and the size of the observed intersection set between *f* and *w* will approach infinity.

4. A Learning Algorithm Within the World:

I will call a learning algorithm that is optimized to learn within the physical world a World Learning Algorithm (WLA). In addition to the definitions in the LA, a WLA has a physical 'existence' constraint that creates a new optimization problem. The following is a definition of the 'existence' constraint, its optimization problem, and a new definition of the conjecture function. While the LA is an algorithm that selects knowledge, the WLA is an algorithm that pursues knowledge through *w*, both for the purpose of defining *f* that has the maximum intersection set with *w*.

4.1. The Constraint of Existence

The abstract LA does not assume that the LA exists *within w*. The WLA assumes that the LA is defined within *w*. Therefore, every operation computed by the LA depends on operations computed by *w*. Because of its physical dependence on *w*, the WLA definition has a larger set of constraints than the LA.

The problem of the LA is to maximize the size of observed intersection between *f* and *w*. When an LA operates, as a physical computer within *w*, maximizing the intersection of *f* and *w* depends on maximizing:

1. the quantity of operations performed by the LA (maximizing the size of the observable intersection set within *w*), and
2. the rate of operations performed by the LA (minimizing the distance between existence and non-existence of the LA within *w*).

Assuming learning continues indefinitely, this can be expressed as optimizing for:

$$max\left(\frac{+\Delta T}{+\Delta time}\right) \quad .$$

Efficiency of defining *w* using a WLA can be improved by optimizing for potential, rather than observed ΔT. That is, optimizing for generating the definitions that can correctly predict observed states

from *w* (satisfying *h*) rather than optimizing generating observations of those predictions. This may be achieved by optimizing for:

$$max(\frac{1}{+\Delta time}) \quad ,$$

where time periods are measured between successful updates to *f* (as determined by *h* evaluation). That is, the objective of the WLA becomes minimizing time, subject to the condition of evaluating a successful update to *f*. This second function will avoid the physical computation cost of observing states that are not useful for updating *f*. This optimization function can be applied to *c*, allowing learning to continue via the physically embodied LA. In the context of the WLA, *c* may be viewed as defining a direction of the trajectory of the WLA through *w*.

5. Constructing a World Learning Algorithm

5.1. Initializing the Conjecture Function

The ability of *c* to generate an update to *f* that satisfies *h* (where *f*' has increased observed intersection with *w*) with greater frequency than a random update, implies the existence of intersection between *c* and *w*. Therefore, increasing intersection between *c* and *w* by defining *c* as a function of *f*, may increase the ability of *c* to successfully update *f*. Because *f* is not a function of *c*, *c* does not need to be evaluated by *h*, and *c* may update independent of *f*.

c may receive as input the various readable states available to the WLA (definitions of *f*, sensor data, observation data, etc.). *c* must must output computable definitions to update *f*. Modern LLMs are a set of functions that can plausibly read the various states available to the WLA and generate definitions of *f* that will successfully satisfy *h*.

Therefore, I will define *c* as a trained LLM. This LLM will be given access to *f* (e.g. a Python file) in the form of a 'tool' [Pa+22, Sc+23]. It will also be given API access read sensor, observation, and other data available to the WLA. It will be responsible for generating updates to *f* and generating new predictions using *f*. Updating *f* and satisfying *h* will be treated as a sequential decision making task, where *c*, the agent, interacts with an external environment, outputs actions and receives data until a knowledge update is accepted by the LA.

5.2. Utility-Adjusted Reward

A policy-based RL algorithm will update *c*. I selected the REINFORCE algorithm because it requires relatively few assumptions about the importance of knowledge wrt the action trajectory. See the following reference for a definition of the REINFORCE algorithm [Mu25]. The reward for this algorithm is computed using the WLA optimization problem of max(1/+Δtime), where reward is

computed episodically as 1/+Δtime when *h* is satisfied. When applied to RL, this reward may need to be standardized using a running estimate of the mean and standard deviation of the reward observed during training, but I do not elaborate on how this should be done.

Maximizing the *potential* future observed intersection between *f* and *w* across an indefinite learning sequence requires preferentially generating updates to *f* that will be useful for generating future updates to *f*, in proportion to their usefulness. The learning sequence is indefinite because learned definitions of features in our world are always underdetermined, meaning that learning the physical world will be a continuing task; the agent-environment interaction is potentially infinite. Therefore, optimal updates to *c* will increase its preference for creating the kind of knowledge that is specifically useful for creating better knowledge. The REINFORCE algorithm is modified to reward *c* according to the utility of definitions and and to avoid reward hacking.

5.2.1. Trajectory States and Boundaries

Trajectories (sequences of agent-environment interactions) of the WLA are generated by computation tracing, where each step in the trajectory corresponds to an operation performed by the WLA (Figure 1). This definition of 'trajectory' differs from the conventional RL definition, in which every state-action pair of a trajectory is computed by a single function (an ANN).

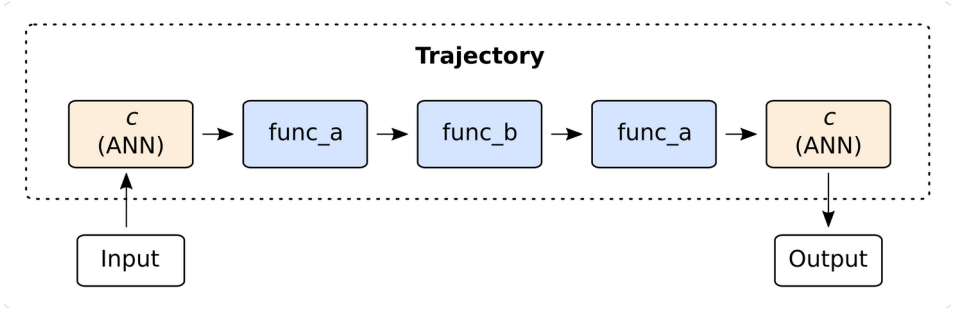

Figure 1: An example WLA trajectory segment where *c* interacts with definitions from *f*. States (blue and beige boxes) in the trajectory consist of operations computed by the WLA, including *c* (an ANN; beige) and functions previously generated by *c* (blue).

Boundaries of rewarded trajectories are defined by successful evaluations of the LA, where *h*=1 for some updated *f* generated by *c*. Every successful update to *f* (a discrete step of knowledge creation) has a corresponding finite trajectory. A trajectory begins at the computation index where *h* was last satisfied, immediately preceding a generated update to *f*. When *h* is satisfied (*h*=1) by the updated *f*, the trajectory of that update terminates and the positive environment reward is computed.

Rewarded trajectories of specific updates to *f* may overlap (Figure 2). This can happen if for example, a conjectured definition is abandoned and replaced with a new definition prior to successfully observing an update as satisfying *h*; or if a new conjecture is generated, observed and satisfies *h* prior to the original conjecture satisfying *h*. This does not change how the boundaries of trajectories are defined,

beginning from the index where *h* was satisfied prior to the conjecture generation and ending when the specific updated *f* satisfies *h*.

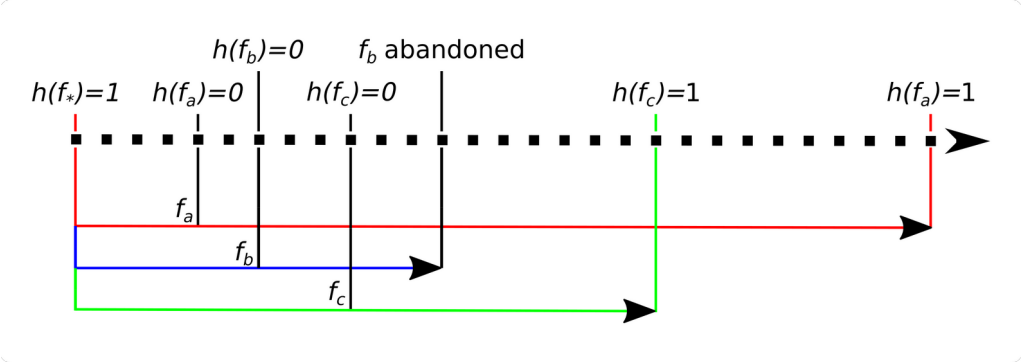

Figure 2: Example of episodic reward trajectories of a WLA's agent-environment interaction. Each new trajectory begins at the index of the last reward (the index where *h* evaluates f_* to 1). Indexes where new definitions (f_a, f_b, and f_c) are generated are indicated by black vertical lines. The trajectories of definitions f_a, f_b, and f_c are respectively indicated by red, blue, and green horizontal lines. The trajectory of a definition terminates when the definition is used to compute an observed state that results in the definition achieving maximum observed intersection with *w* (i.e. *h* evaluates the definition to 1). In this example, the definition of f_b is never accepted and the definition is abandoned; it does not define a complete trajectory and does not generate a reward.

5.2.2. Rewardable Ancestor Set

A definition, from the perspective of the WLA, is any object stored in readable memory; it may include definitions in *f* or definitions stored elsewhere, such as within the context window of *c*. Rewardable definitions include only the definition *c* and definitions that have been generated by *c*.

The set of rewardable definitions for an operation step is determined using computational dependency data of that step. A similar computation tracing was used in my prototype "Explanatory World Model" learning system [Ti24], which relies on Python's low-level sys.settrace() function to record definitions accessed during prediction generation. In the prototype design, as in the WLA, only accessed definitions that have previously been defined by the ANN are recorded, while other definitions such as built-in Python definitions are excluded.

The execution-trace dependency graph of any terminating computation is a Directed Acyclic Graph (DAG), which inherently exhibits hierarchical structure. From this perspective, every operation performed by the WLA will have a specific ancestor set. Further, every output (definition) of every operation performed by the WLA will have a specific ancestor set. The ancestor set of an output is equivalent to the ancestor set of the operation that generated it. The ancestors of an operation in a DAG include the operation definition itself plus all the nodes (definitions) from which there is a directed path to that node (where edges point from prerequisite to dependent). The ancestors of an operation

correspond to the set of all nodes (both values and operations) that must be defined before a target operation can be evaluated. Rewardable definitions are those that have non-empty rewardable ancestor sets (i.e. that contain *c* or a definition generated by *c*).

The distribution of the positive episodic reward for a trajectory is determined by weights (which sum to 1) assigned to definitions in the rewardable ancestor sets. Each rewardable operation in a rewarded trajectory receives a weight equal to the inverse of the total number of rewardable operations in the trajectory.

Let $S=(o_1,\ldots,o_N)$ be the sequence of N rewardable operations in a trajectory.

Let $W(o)$ be the weight assigned to a single rewardable operation in the trajectory, where:

$$W(o)=\frac{1}{|S|} \quad .$$

When a weight is assigned to an operation in the trajectory it flows to the set of rewardable definitions in that operations ancestor set. Each rewardable definition in the ancestor set of a rewardable operation receives a fraction of the weight assigned to that operation. Expressed more formally:

For each $o_i \in S$, let A_i be its non-empty rewardable ancestor set ($|A_i|\geq 1$).

The weight of a rewardable ancestor of a rewardable operation is equal to the inverse of the size of the rewardable ancestor set of that step,

$$W(a_i)=\frac{1}{|A_i|} \quad .$$

Note that I later refer to the above weighting as ‘Flat’ and I define an ‘Operation Dominant’ variation of this algorithm in the ‘Example’ section below. If an ancestor definition is not *c* (it is a definition generated by *c*), then that definition passes its weight to its rewardable ancestor set. Weight continues passing from definitions to ancestors until all weight is assigned to *c*, where each receiving c has a specific state-action pair and computable or recorded log-probability. Said differently, there are two kinds of definitions in the ancestor set of any WLA definition: probabilistic (*c*; ‘absorbing nodes’) and non-probabilistic (definitions generated by *c*; ‘flow nodes’). Weight assigned to a non-probabilistic definition flows to the ancestor set of that definition. Weight assigned to a probabilistic definition is absorbed by that definition.

The approximate total weight $W(a)$ assigned to each absorbing member, *c*, of the union of all rewardable ancestor sets of a rewardable trajectory is given by:

$$W(a)=\sum_{i=1}^{N}\frac{\boldsymbol{I}(a\in A_i)}{N\cdot|A_i|} \quad ,$$

where $\boldsymbol{I}(a \in A_i)$ is the indicator function, equal to 1 if $a \in A_i$ (if the definition is an element of the rewardable ancestor set of that step) and 0 otherwise.

The precise weight is greater because it includes all transmitted weight from the non-probabilistic definitions ('flow nodes') of which the specific *c* (absorbing node) is an ancestor. The total of the precise weights assigned of all absorbing ancestor definitions is equal 1.

5.2.3. Gradients and Policy Updating

The episodic environment reward, R_S, though perhaps in a standardized form, is precisely that which is defined in the optimization problem of the WLA:

$$R_S = \frac{1}{\Delta time} \quad .$$

There is no intrinsic reward generated by the agent. Step cost and discount factor (i.e. γ=1) are intentionally absent, consistent with the absence of assumptions about the relative importance of any step in a trajectory. The positive episodic reward is divided among the rewardable probabilistic (absorbing) definitions in the trajectory. Longer trajectories avoid receiving inappropriately large reward.

The gradient (derived from the specific log probability and action) of each absorbing *c* is multiplied by the total weight assigned to that *c*. All weighted gradients are summed. The summed gradients are multiplied by the trajectory reward. This reward weighted gradient is used to update the policy of *c*.

The weighted gradient g_i of the log-probability of an action for rewardable ancestor (absorbing definition of *c*) a_i is expressed as:

$$g_i = W(a_i) \nabla_\theta \log \pi_\theta(u_i|s_i)$$

Where π_θ is the current policy (*c*) definition, with parameters θ, s_i is the input 'state' and u_i is the output 'action' of a specific absorbing definition. $W(a_i)$ is the weight assigned to a specific rewardable ancestor.

If this gradient, computed using the current policy is available for an absorbing definition, it can be weighted and used to compute the policy update. If the definition of *c* has been updated since the gradient was recorded an 'imitation gradient' is computed using the recorded state and action data of that absorbing definition and the current policy definition. It is identical to the prior gradient definition.

The policy update is thus:

$$\theta \leftarrow \theta + \alpha R_S \sum_{i=0}^{A} g_i \quad .$$

where α is the learning rate, R_s is the reward for the trajectory, and *A* is the set of all ancestor definitions in the trajectory.

5.2.4. Example

To illustrate, Table 1 shows a simple trajectory segment with rewardable ancestor sets and absorbing definition weights.

Table 1: Example of a simplified trajectory segment, showing the sequence of 10 rewardable operations computed by the WLA and the rewardable ancestor set of these operations.

Step	Function	Ancestor Set	Output	Rewardable Ancestor Set	$\|A_i\|$
1	c_0	c_0	API call to sensor: “<<s1:‘read’:-100:>>”. [Latest 100 states from sensor 1 are appended to context; denoted as ‘s1_A’ in ancestor sets.]	c_0	1
2	c_1	s1_A c_1	API call to *f*: “<<f:{read:input_A}>>”. [input_A definition is appended to context; denoted as ‘input_A’]	s1_A c_1	2
3	c_2	s1_A c_1 input_A class_A c_2	API call to *f*: “<<f{‘assign’: {‘input_B’:8}}, ctx{‘app’:’input_B=8’}>>”. [“input_B=8” string is appended to context.]	s1_A c_1 input_A class_A c_2	5
4	c_3	s1_A c_1 input_A class_A c_2 input_B c_3	API call to *f:* “<<f: {call:f.func_A(input_A, input_B)}>>”	s1_A c_1 input_A class_A c_2 input_B c_3	7
5	op_A1	s1_A c_1 input_A class_A c_2 input_B c_3 func_A op_A1	result_A1	s1_A c_1 input_A class_A c_2 input_B c_3 func_A op_A1	9

6	op_B	s1_A c_1 input_A class_A c_2 input_B c_3 func_A op_A1 result_A1 op_B	result_B	s1_A c_1 input_A class_A c_2 input_B c_3 func_A op_A1 result_A1	10
7	op_A2	s1_A c_1 input_A class_A c_2 input_B c_3 func_A op_A1 result_A1 op_B result_B op_A2	result_func_A [result_func_A definition “int(4)” is appended to context.]	s1_A c_1 input_A class_A c_2 input_B c_3 func_A op_A1 result_A1 result_B op_A2	12
8	c_4	s1_A c_1 input_A class_A c_2 input_B c_3 func_A op_A1 result_A1 op_B result_B op_A2 result_func_A c_4	API call to *f*: “<<f{‘assign’: {‘var_1’:4}}>>”.	s1_A c_1 input_A class_A c_2 input_B c_3 func_A op_A1 result_A1 result_B op_A2 result_func_A c_4	14
9	c_5	s1_A	API call to context manager: “<<ctx:	s1_A	15

		c_1 input_A class_A c_2 input_B c_3 func_A op_A1 result_A1 op_B result_B op_A2 result_func_A c_4 c_5	{del:}>>". [Context is cleared.]	c_1 input_A class_A c_2 input_B c_3 func_A op_A1 result_A1 result_B op_A2 result_func_A c_4 c_5	
10	c_6	c_6	API call to sensor: "<<s_1:'read':-100:>>". [Latest 100 states from sensor 1 are appended to context.]	c_6	1

The trajectory begins with *c* calling the API to read data from 'Sensor 1'. This operation, defined by *c*, produces 'Step 1' of the trajectory, with a specific state-action pair and gradient of the log probability of the action output by *c*, denoted c_0. The rewardable ancestor set of the operation at Step 1 contains one element: c_0.

The data received from Sensor 1 is added to the context window of *c* (included in the state of Step 2). Next, *c* calls the API to *f*, requesting to read the definition of the variable 'input_A',which is defined in *f*. The sensor data within the context of the operation at Step 2 is treated as a definition, stored in memory of the WLA, which I will identify as 's1_A' (Sensor 1 data set A). Because the operation, c_1, at Step 2 depends on the definition 's1_A' (which is in that operations input), it is included in the set of rewardable ancestors of the operation.

The sequence of operations continues, defining inputs to, and computing with, the function 'func_A' defined in *f*. Both 'input_A' and 'class_A' (of which 'input_A' is a member) are defined in *f*. 'func_A' stores definitions of operations 'op_A1' and 'op_A2' and a reference to an externally defined operation 'op_B'. The definitions of 'input_A', 'class_A', 'func_A', 'op_A1', and 'op_A2' have previously been generated by *c* and satisfied *h*; each of these definitions has a specific ancestor set and corresponding set of weighted absorbing definitions (originating from the prior operations that generated these definitions). The definition of 'input_B' is generated by *c* during the current trajectory; it is dynamically defined in *f* (e.g. a runtime variable in an initialized f.py). 'op_B' is a built-in definition of a Python library. Neither 'input_B' or 'op_B' have satisfied *h*. Because 'input_B' was generated by *c*, it has a rewardable ancestor set. The definition of 'op_B' did not depend on *c*; thus it is excluded from

rewardable ancestor sets. The entire sequence of rewardable operations and their ancestor sets appear in Table 1. In many ancestor sets, such as that of Step 3, definition '*c*' appears multiple times. Although the parameters of the current *c* definition are static, the log-probability value of each *c* element differs; these are identified by subscript indices in the example.

Each definition appearing in the rewardable ancestor sets of operations has its own rewardable ancestor set (Figure 3). If the definition is not '*c*' the weight assigned to it from the operation will flow to its ancestor set, until all weight is distributed to '*c*'. All weight assigned to 's1_A' is absorbed by 'c_0'. All weight assigned to 'c_1' is absorbed by 'c_1'. Weight assigned to 'input_B' flows in equal parts to: 'A_{s1_A}', 'c_1', 'A_{input_A}', 'A_{class_A}', and 'c_2'. Here, '*A*' with a subscript means the rewardable ancestor set of the subscript definition. When weight flows to an ancestor set, it is distributed equally to each rewardable definition in the set.

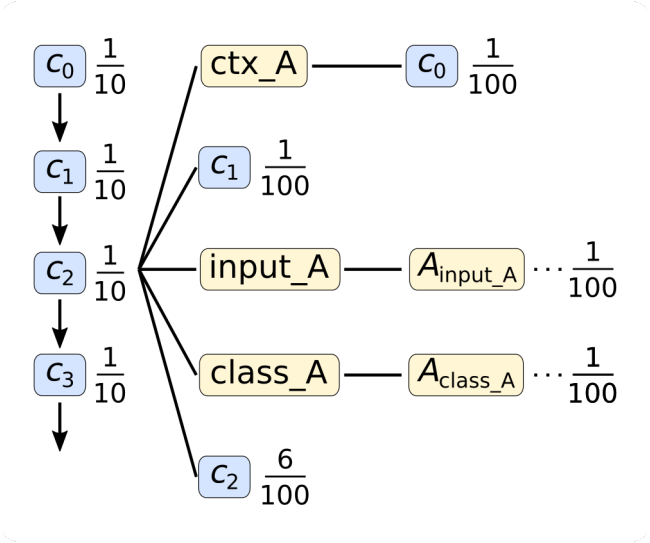

Figure 3: Example of weight assignment in a 10 step trajectory. The total trajectory weight, 1, is divided by the number of steps. Each step is assigned 1/10 weight and distributes the weight to its rewardable ancestors, which include the operation definition if it is rewardable. This example shows an Operation Dominant variation of weight distribution, where rather than assigning weight equally to all rewardable ancestors, an operation assigns half of its weight to itself, and the other half to its ancestor set.

Table 2: WLA definitions from the example simplified trajectory segment, their corresponding rewardable ancestor sets, and the total reward assigned to each definition. Two different approaches to weighting 'Flat' and 'OpDom', with different weight flow through operations are shown. Only the definitions 'c_0' to 'c_6' can absorb weight. The definitions 'input_A', 'class_A', 'func_A', 'op_A1', and 'op_A2' pass their weight to their rewardable ancestor sets (not shown).

Definition	Ancestor Set	Assigned Weight (Flat)	Assigned Weight (OpDom)
c_0	$\{c_0\}$	0.248	0.175
c_1	$\{c_1\}$	0.148	0.125
c_2	$\{c_2\}$	0.098	0.119
c_3	$\{c_3\}$	0.065	0.086
c_4	$\{c_4\}$	0.014	0.057
c_5	$\{c_5\}$	0.007	0.053
c_6	$\{c_6\}$	0.100	0.100
s1_A	$\{c_0\}$		
input_A	$A_{\text{input\_A}}$	0.098	0.050
class_A	$A_{\text{class\_A}}$	0.098	0.050
input_B	$\{A_{\text{s1\_A}}, c_1, A_{\text{input\_A}}, A_{\text{class\_A}}, c_2\}$		
func_A	$A_{\text{func\_A}}$	0.051	0.029
op_A1	$A_{\text{op\_A1}}$	0.051	0.090
op_A2	$A_{\text{op\_A2}}$	0.023	0.064
result_A1	$\{A_{\text{s1\_A}}, c_1, A_{\text{input\_A}}, A_{\text{class\_A}}, c_2, A_{\text{input\_B}}, c_3, A_{\text{func\_A}}, A_{\text{op\_A1}}\}$		
result_B	$\{A_{\text{s1\_A}}, c_1, A_{\text{input\_A}}, A_{\text{class\_A}}, c_2, A_{\text{input\_B}}, c_3, A_{\text{func\_A}}, A_{\text{op\_A1}}, A_{\text{result\_A1}}\}$		
result_func_A	$\{A_{\text{s1\_A}}, c_1, A_{\text{input\_A}}, A_{\text{class\_A}}, c_2, A_{\text{input\_B}}, c_3, A_{\text{func\_A}}, A_{\text{op\_A1}}, A_{\text{result\_A1}}, A_{\text{result\_B}}, A_{\text{op\_A2}}\}$		
var_1	$\{A_{\text{s1\_A}}, c_1, A_{\text{input\_A}}, A_{\text{class\_A}}, c_2, A_{\text{input\_B}}, c_3, A_{\text{func\_A}}, A_{\text{op\_A1}}, A_{\text{result\_A1}}, A_{\text{result\_B}}, A_{\text{op\_A2}}, A_{\text{result\_func\_A}}, c_4\}$		

The rewardable ancestor sets of some definitions have not been defined in the current trajectory. Therefore, the ancestor sets of these definitions must be read from memory so weights may continue to flow to absorbing '*c*' definitions; this is not shown in the example.

The total weight assigned to each absorbing definition is the sum of all rewards assigned to that definition from the current rewarded trajectory. The final assigned weights for the definitions are shown in Table 2. Again, note that weight assigned to definitions other than '*c*', like 'input_A' must continue to flow to that definition's ancestor set; however, this is excluded for brevity. The absorbing definitions from within the current trajectory include 'c_1' to 'c_6', which together receive 68% of the assigned weight. The remaining 32% is distributed to other '*c*' definitions (not shown) that absorb weights from the definitions of 'input_A', 'class_A', 'func_A', 'op_A1', and 'op_A2'.

I am uncertain if the ideal weight distribution from operations to their ancestors should be flat (as described above) or if there should be a hierarchical distribution where operation definitions dominate the hierarchy. For example, rather than a rewardable operation distributing weight equally to each member of its ancestor set, it may distribute half to the rewardable ancestors of its definition and half to the rewardable ancestors of the specific operation (including the operation definition plus the definitions of inputs). In cases where the operation is not '*c*', this may produce only a small difference in the flow of weight, but if the operation is '*c*' (an absorbing definition) then that '*c*' definition will receive a substantially larger reward. The weight distribution of an operation-dominant flow is shown in the rightmost column of Table 2. With 'OpDom' weighting, the absorbing definitions within the current trajectory ('c_1' to 'c_6'), together receive 72% of the assigned weight.

[Amendment: The ancestor sets should include two variants of 'input_B', one defined in *f* ('f_input_B') and one defined in the context ('ctx_input_B'). These definitions would be included in the ancestor sets of the operation Steps 4-9 ('ctx_input_B') and Steps 5-9 ('f_input_B').]

5.3. Learning Algorithm

Prediction generation and observation by the WLA follows the definitions of the LA. Observable input-output pairs (elements of *E*) are generated by *w*. They may be derived from a 'sensor', whose state it updated by *w*. States generated by the sensor may be processed by some other function (e.g. *c*), where that function is simply a component of *w* from the perspective of the LA. A processed sensor output sequence may define the observable that the predictions from *f* are evaluated against.

When predictions are generated, no element of prediction set *P* can contain an output generated from a from an input or function that was in any way derived from observed states with indexes including or exceeding the output index. That is, predictions cannot be generated using the output states they are being compared against. This means that the set of observable elements in *E* (and the possible size of *T*) will shrink each time *c* updates *f* using information from prior observations. I suspect this will not impair learning (e.g. incentivizing *c* to sequester knowledge from *f*) by this system the context of the

indirect observation and reward weighting mechanisms, but I have not proved this. I believe it will be important that *h* always evaluates new candidate *f* definitions against the last definition that satisfied *h*.

E need not contain more than one element for h to be satisfied, but if *E* is large (e.g. it includes many states observed from the prior sequence of successful knowledge updates), then it may be possible minimize the number of these states that must be re-computed by a candidate *f'* by searching the ancestor sets of all prior observed predictions generated by *f* and checking if any of those definitions have been modified in the updated *f'*; only the predictions with modified ancestor definitions must be recomputed using *f'*, others can inherit the observation value generated using the preceding *f* definition. This may substantially decrease the cost of computing max(*T*) if the set *E* increases in size during learning.

5.3.1. Defining Physical Sensors

From the perspective of a WLA, any readable memory location whose state is updated by *w* may qualify as a sensor. A WLA may access multiple sensor channels. I expect that it will initially learn most rapidly if the sensor data resembles the kind of data (e.g. human-readable data) that the ANN is already familiar with, and that it can 'intuitively' express as code. Conceivably, a sensor can be a communication channel with a human or with an LLM (such an arrangement may be the most efficient early during learning).

5.3.2. Indirect Observation Using the Conjecture Function

In this WLA design, I define *c* as a processing function for observable sensor states. Therefore, elements of E are defined by *c*. While it may be possible to construct a WLA, where *f* directly receives unprocessed states generated by *w*, I expect that more rapid learning of a broad range of useful features from *w* occur if an ANN first processes raw sensor data. In this case, *c* will receive state information from a sensor and return a structured input for the candidate *f* definitions. Next, *c* will process the corresponding of those *f* definitions back into raw sensor data. This feature of the design rests on an assumption that, in this context, decreased loss equates to increased intersection between *f* and *w*. I think this assumption is justified, but I have not proved this. With this arrangement the following sequence of events occur:

1. *c* generates an updated definition of *f* (called *f'*);
2. *c* reads raw sensor data until it decides to define a region of yet unread sensor data for observing a prediction;
3. *c* generates a single input to pass to the two candidate versions of *f* (e.g. "calculator.prod(9918,88.31)";
4. *c* separately runs *f* and *f'* and passes each the input;
5. *c* receives the distinct outputs from *f* (e.g. "Error") and *f'* (e.g. "875858.58") then processes the unread sensor data;

6. the sequence loss is separately calculated for *c*'s forward pass over the sensor data, given the outputs of *f* and *f'* (where the presence of these outputs is the only difference between passes);
7. if the sequence loss from the pass with the output of *f'* is lower (beyond a threshold) than that of the pass with the output of *f*, the prediction computed by *f'* is evaluated as intersecting ($T_{\text{prediction}}$=1) with *w*. Reward can only be obtained when the loss using *f'* is lower than the loss using *f* (no reward is assigned in the opposite case).

Recall that *c* uses *f* as a tool and it can already call *f*. Thus, during the comparative evaluation loss is being compared between two instances of *c* that can both call *f* (the old knowledge definition), one instance is given the prediction generated by *f* and the other is given the prediction generated by *f'*.

I believe it will be useful to define a minimum loss difference to evaluate a prediction as consistent. Assuming there will be two groups of loss differences, random (where there is no difference between the consistency of the predictions with *w*) and real (where there is a difference in the consistency of the predictions with *w*), then a loss threshold may be defined using a clustering algorithm like K-means clustering (with K=2). The loss threshold will adjust as the WLA observes more predictions.

The ANN defining *c* is only rewarded for decreasing its own loss wrt the observable world. It decreases its loss by formally defining features of the world.

5.4. Defining Internal and External Actions

In this design, I have defined *c* as the 'agent' of the WLA. Along state-action trajectories, *c* can interact with APIs for updating and computing with *f*, reading sensors, generating and evaluating predictions, and potentially more. It is conceivable that *c* will also have an API to emit actions into the world, in addition to its internal action set. For example, *c* may be able to send text strings to another LLM or to a human.

If given a means to emit actions externally, it is conceivable that *c* will develop a preference for physical actions that accelerate its learning (defined by the LA). Assuming that action emission can alter the features of *w* that are observable via the sensors, the WLA will emit the kinds of actions that result in either conjecturing updates to *f* that generate more easily observable predictions (e.g. seeking external sources of knowledge), or actions that result in more successful observation of predictions from conjectured updates to *f* (moving specific features of *w* into its view by constructing observable experiments).

6. Conclusion

I have defined a learning algorithm as a function that evaluates and selects 'knowledge' definitions according to the size of of their observed intersection set with a target function. This definition was

primarily influenced by Karl Poppers proof of the impossibility of inductive probability [PM83], and by his [Po62] and David Deutsch's attempts at defining knowledge.

Whether recognized as the definition of learning or not, the size of the observed intersection set has been used in at least one recent attempt at world-model learning [Ta+24]. The algorithm was also subjectively effective in my prototype 'Explanatory World Model' learning system [Ti24], which learns a model of our world, defined in a Python file, from an LLM. Although my prototype appears capable of learning an improving model of our world, it it computationally inefficient, it cannot learn *how* to learn, and it had no incentive to learn 'useful' knowledge.

The World Learning Algorithm is an attempt at solving these problems, first by defining the optimization problem of learning within our world, then by defining an scheme for building a machine that addresses the problem. The machine does not depend on an engineer to define a specific dataset, problem, or reward, supposedly corresponding to some concerning feature of our physical world. The machine uses a parameterized ANN as the 'direction' function for a learning algorithm. The learning direction is updated toward improving learning success as defined by a Learning Algorithm. The system will learn to generate knowledge that is particularly useful for generating more knowledge. The the learned knowledge is expected to have a maximum possible observable intersection set with the physical world.

I expect the behavior of this World Learning Algorithm algorithm to mirror the behaviors that we informally define as 'science'. If this formal definition of learning in our world is correct, it can be used to automate the creation of knowledge, perhaps automating science as well.

7. Speculated Behavior

Below are some behaviors I expect a WLA will develop. These seem particularly speculative and were intentionally removed from the main text.

7.1. Experimentum Crucis

I expect a learning system with this design will find trajectories that resemble experimentum crucis, where there is a preference to update heavily used functions, with the update capable of reproducing prior observed predictions of the prior function. A single consistent observation, unreproducible using the old definition, will support replacement of the old definition with the new definition. All future weight that would have been assigned to the old definition is 'shunted' to the new definition.

7.2. Context Management

I expect this design to learn to remove non-essential definitions from the ancestor sets of operations. Definitions that are not immediately needed can be deleted or stored in memory for later retrieval (e.g.

a dynamic state in an initialized *f* or elsewhere). It may learn behaviors like context window clearing (as in Step 9 of the Example trajectory). When rewardable definitions are moved out of an operations ancestor set, the remaining rewardable definitions will receive a larger wight than they would have otherwise, as long as doing so does not disproportionately extend episode time.

7.3. Update Tracking

I expect the WLA will learn to 'remind itself' of recent definitions it has learned, by storing identifiers of these definitions in a specific memory region. I expect this because imitation gradients will push *c* to use old, possibly outdated definitions in *f* and if c is 'unaware' of updates to *f* it will have suboptimal success at receiving reward. Therefore, *c* can improve performance by storing information about updates to *f* in a static memory region until *c* develops an internal representation of the update.

7.4. Check-Pointing

I expect the WLA will use the discrete knowledge in *f* to define 'checkpoints' along it's trajectories. In order to observe a specific prediction it will require new sensor data that corresponds with specific input and output definitions of a computation from *f*. Successfully defining sensor data regions that can evaluate a specific prediction may require *c* to learn an 'intuitive' mapping of sensor data to definitions in *f*. For example, prediction from some specific update to *f* will only be observable in specific regions of *w*. These regions will have corresponding discrete definitions in *f*. If sensor data received by the WLA is inconsistent with the definitions of these regions, the WLA may preemptively abandon an attempt to observe a specific update to *f* in favor of a more easily observed update.

To illustrate, the definition 'gravitational_force' may have high empirical utility, receiving substantial weight across rewarded episodes. The WLA may thus develop a preference to find a trajectory that will improve the definition of 'gravitational_force'. *c* may define a more accurate and precise definition of 'gravitational_force'; however, it may be unable to finding an observable set of states from *w* where the two different predictions produce a meaningful difference in loss (e.g. it may require access to precise measuring equipment). If, during a lengthy trajectory the WLA has failed to find observable states suitable for the new definition of 'gravitational_force' it may abandon the definition in favor of another. If the required context for observing a specific definition has definitions within *f* (e.g. to observe the new 'gravitational_force' an object of class 'Spring' with a particular spring constant is required), then these definitions may create checkpoints used to anticipate and decide successful trajectories.

7.5. Learning from Errors

I expect the WLA will learn to use inconsistent observation data to improve it's definitions. This inconsistent data will indicate opportunities to define new observable constraints to the model in *f*.

For example if the object 'my_ball_A' is defined defined as a member of class 'TennisBall' but the class attribute "color='yellow' or diameter=6.7" cannot be observed, the WLA may recognize an

opportunity define a different class for ‘my_ball_A’ (e.g. as a member of class ‘BowlingBall’) with observable class attributes. This error guidance behavior is used in the Explanatory World Model learning prototype [Ti24], and I expect it will be learned by a WLA.

7.6. Active Centering within the World

A WLA may be gradually move its physical ‘self’ through *w* toward a ‘point of greatest freedom’ or ‘central node’ in *w* (as defined in *f*). Because a best approximation of the existence constraint is included in its optimization function, the WLA specifically create a model of ‘itself’ in the world. The WLA may seek to move its defined observable position in the world toward a point in *w* from which the WLA has the most time efficient path to updating its knowledge, which may correspond to the point in the virtual possibility space (defined in *f*) where ‘self’ can reach the largest volume of the defined manifold.

7.7. Offloading Computation to the Knowledge Function

I expect, as the knowledge in *f* improves, c will hand over a larger proportion of computation to *f*. In the extreme, *c* will approach an identity function that transfers data unaltered from its context to *f*.

8. References

[BN06] Bishop, C. M., & Nasrabadi, N. M. (2006). Pattern recognition and machine learning (Vol. 4, No. 4, p. 738). New York: Springer. *https://www.microsoft.com/en-us/research/wp-content/uploads/2006/01/Bishop-Pattern-Recognition-and-Machine-Learning-2006.pdf*

[De11] Deutsch, David. (2011). The beginning of infinity: Explanations that transform the world. Penguin UK.

[Mu25] Murphy, K. P. (2025). Reinforcement learning: A comprehensive overview. In Policy-based RL: REINFORCE (Chapter 3, Section 3.2, p. 45). arXiv. *https://arxiv.org/pdf/2412.05265v2*

[Pa+22] Parisi, A., Zhao, Y., & Fiedel, N. (2022). TALM: Tool augmented language models. arXiv Preprint arXiv:2205.12255. *https://arxiv.org/pdf/2205.12255*.

[PM83] Popper, K., & Miller, D. (1983). A proof of the impossibility of inductive probability. Nature, 302(5910), 687-688.

[Po62] Popper, K. (1962). Truth, rationality, and the growth of scientific knowledge (Part 3. Truth and content: Verisimilitude versus probability). In Conjectures and refutations: The growth of scientific knowledge (Chapter 10). *http://ninthstreetcenter.org/Popper.pdf*.

[Sc+23] Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Hambro, E., ... & Scialom, T. (2023). Toolformer: Language models can teach themselves to use tools. Advances in Neural Information Processing Systems, 36, 68539-68551. *https://arxiv.org/pdf/2302.04761*.

[SS15] Stulp, F., & Sigaud, O. (2015). Many regression algorithms, one unified model: A review. Neural Networks, 69, 60-79. *https://hal.science/hal-01162281/file/hal-01162281-v2.pdf*

[Ta+24] Tang, H., Key, D., & Ellis, K. (2024). Worldcoder, a model-based llm agent: Building world models by writing code and interacting with the environment. Advances in Neural Information Processing Systems, 37, 70148-70212. *https://openreview.net/pdf?id=QGJSXMhVaL*.

[Ti23] tiplur-bilrex. (2023). Applied Fallibilism. [Blog]. *https://join.tlon.io/0v3.er2qr.gugd7.a7l6s.om30i.ij783*.

[Ti24] tiplur-bilrex. (2024). Explanatory World Model Learner, Version 1. [Computer code]. GitHub. *https://github.com/tiplur-bilrex/ewm_learner/tree/main*. Demonstration video: *https://x.com/tiplur_bilrex/status/1872500868417003929*.