Using Program Induction to Interpret Transition System Dynamics

Svetlin Penkov [1] Subramanian Ramamoorthy [1]

Abstract

Explaining and reasoning about processes which underlie observed black-box phenomena enables the discovery of causal mechanisms, derivation of suitable abstract representations and the formulation of more robust predictions. We propose to learn high level functional programs in order to represent abstract models which capture the invariant structure in the observed data. We introduce the π-machine (program-induction machine) – an architecture able to induce interpretable LISP-like programs from observed data traces. We propose an optimisation procedure for program learning based on backpropagation, gradient descent and A* search. We apply the proposed method to two problems: system identification of dynamical systems and explaining the behaviour of a DQN agent. Our results show that the π-machine can efficiently induce interpretable programs from individual data traces.

1. Introduction

Learning models of transition systems has been a core concern within machine learning, with applications ranging from system identification of dynamical systems (Schmidt & Lipson, 2009) and inference of human choice behaviour (Glimcher, 2011; Brendel & Todorovic, 2011) to reverse engineering the behaviour of a device or computer program from observations and traces (Vaandrager, 2017). With the increasing use of these learnt models in the inner loops of decision making systems, e.g., in robotics and human-machine interfaces, it has become necessary to ensure not only that these models are accurate predictors of behaviour, but also that their causal mechanisms are exposed to the system designer in a more interpretable manner. There is also the need to explain the model in terms of counterfactual reasoning (Bottou et al., 2013), e.g., what would we expect the system to do if a certain variable were changed or removed, or model checking (Baier & Katoen, 2008) of longer term properties including safety and large deviations in performance. We address these needs through a program induction based framework.

We propose to learn high level functional programs in order to represent abstract models which capture the invariant structure in the observed data. Recent works have demonstrated the usefulness of program representations in capturing human-like concepts (Lake et al., 2015). Used in this way, program-based representations boost generalisation and enable one-shot learning. Also, and arguably more importantly, they are significantly more amenable to model checking and human interpretability.

In this paper, we introduce the π-machine (program-induction machine), an architecture which is able to induce LISP-like programs from observed transition system data traces in order to explain various phenomena. Inspired by differentiable neural computers (Graves et al., 2014; 2016), the π-machine, as shown in Figure 1, is composed of a memory unit and a controller capable of learning programs from data by exploiting the scalability of stochastic gradient descent. However, the final program obtained after training is not an opaque object encoded in the weights of a controller neural network, but a LISP-like program which provides a rigorous and interpretable description of the observed phenomenon. A key feature of our approach is that we allow the user to provide a set of predicates of interest in order to specify the properties they are interested in understanding as well as the context in which the data is to be explained. By exploiting the equivalence between computational graphs and functional programs we describe a hybrid optimisation procedure based on backpropagation, gradient descent, and A* search which is used to induce programs from data traces.

We evaluate the performance of the π-machine on two different problems. Firstly, we apply it to data from physics experiments and show that it is able to induce programs which represent fundamental laws of physics. The learning procedure has access to relevant variables, but it does not have any other prior knowledge regarding physical laws which it has discovered in the same sense as in (Schmidt & Lipson, 2009) although far more computationally tractably.

[1] The University of Edinburgh, Edinburgh, United Kingdom. Correspondence to: Svetlin Penkov <sv.penkov@ed.ac.uk>.

Proceedings of the 34^{th} International Conference on Machine Learning, Sydney, Australia, PMLR 70, 2017. Copyright 2017 by the author(s).

Secondly, we study the use of the proposed procedure in explaining control policies learnt by a deep Q-network (DQN). Starting from behaviour traces of a reinforcement learning agent that has learnt to play the game of Pong, we demonstrate how the π-machine learns a functional program to describe that policy.

2. Related work

Explainability and interpretability. The immense success of deep neural network based learning systems and their rapid adoption in numerous real world application domains has renewed interest in the interpretability and explainability of learnt models (Gunning, 2016). There is recognition that Bayesian rule lists (Letham et al., 2015; Yang et al., 2016), decision trees and probabilistic graphical models are interpretable to the extent that they impose strong structural constraints on models of the observed data and allow for various types of queries, including introspective and counterfactual ones. In contrast, deep learning models usually are trained 'per query' and have numerous parameters that could be hard to interpret. Zeiler & Fergus (2014) introduced deconvolutional networks in order to visualise the layers of convolutional networks and provide a more intuitive understanding of why they perform well. Zahavy et al. (2016) describe Semi-Aggregated Markov Decision Process (SAMDP) in order to analyse and understand the behaviour of a DQN based agent. Methods for textual rationalisation of the predictions made by deep models have also been proposed (Harrison et al., 2017; Hendricks et al., 2016; Lei et al., 2016). While all of these works provide useful direction, more generic methods are required which need not be hand-crafted to explain specific aspects of individual models. In this sense, we follow the model-agnostic explanation approach of Ribeiro et al. (2016), who provide "textual or visual artefacts" explaining the prediction of any classifier by treating it as a black-box. Similarly to the way in which (Ribeiro et al., 2016) utilise local classifiers composed together to explain a more complex model, we present an approach to incrementally constructing functional programs that explain a complex transition system from more localised predicates of interest.

The π-machine treats the process which has generated the observed data as a black-box and attempts to induce LISP-like program which can be interpreted and used to explain the data. We show that the proposed method can be applied both to introspection of machine learning models and to the broader context of autonomous agents.

Program learning and synthesis. Program learning and synthesis has a long history, with the long-standing challenge being the high complexity deriving from the immense search space. Following classic and pioneering work such as by Shapiro (1983) who used inductive inference in a logic programming setting, others have developed methods based on a variety of approaches ranging from SAT solvers (Solar-Lezama et al., 2006) to genetic algorithms (Schmidt & Lipson, 2009), which tend to scale poorly hence often become restricted to a narrow class of programs. Recently, deep neural networks have been augmented with a memory unit resulting in models similar to the original von Neumann architecture. These models can induce programs through stochastic gradient descent by optimising performance on input/output examples (Graves et al., 2014; 2016; Grefenstette et al., 2015) or synthetic execution traces (Reed & De Freitas, 2015; Cai et al., 2017; Ling et al., 2017). Programs induced with such neural architectures are encoded in the parameters of the controller network and are, in general, not easily interpretable (particularly from the point of view of being able to ask counterfactual questions or performing model checking). Another approach is to directly generate the source code of the output program which yields consistent high level programs. Usually, these types of approaches require large amounts of labelled data - either program input/output examples (Devlin et al., 2017; Balog et al., 2016) or input paired with the desired output program code (Yin & Neubig, 2017).

Determining how many input/output examples or execution traces are required in order to generalise well is still an open research problem. However, in this paper, we focus attention more on the explanatory power afforded by programs rather than on the broader problems of generalisation in the space of programs. While these characteristics are of course related, we take a view similar to that of (Ribeiro et al., 2016), arguing that it is possible to build from locally valid program fragments which provide useful insight into the black-box processes generating the data. By combining gradient descent and A* search the π-machine is able to learn informative and interpretable high-level LISP-like programs, even just from a single observation trace.

3. Problem definition

Consider the labelled transition system $\Omega(\mathcal{S}, \mathcal{A}, \delta)$ where $\mathcal{S}$ is a non-empty set of states, $\mathcal{A}$ is a non-empty set of actions, each parametrised by $\theta \in \mathbb{R}^D$, and $\delta : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ is the state transition function. We define an observation trace $\mathcal{T}$ as a sequence of observed state-action pairs $(s_t, a_t(\theta_t)) \in \mathcal{S} \times \mathcal{A}$ generated by the recursive relationship $s_{t+1} = \delta\left(s_t, a_t(\theta_t)\right)$ for $1 \leq t \leq T$. We are interested in inducing a LISP-like functional program ρ which when executed by an abstract machine is mapped to an execution trace $\mathcal{T}_\rho$ such that $\mathcal{T}_\rho$ and $\mathcal{T}$ are equivalent according to an input specification.

We represent the abstract machine as another labelled transition system $\Pi(\mathcal{M}, \mathcal{I}, \varepsilon)$ where $\mathcal{M}$ is the set of possible

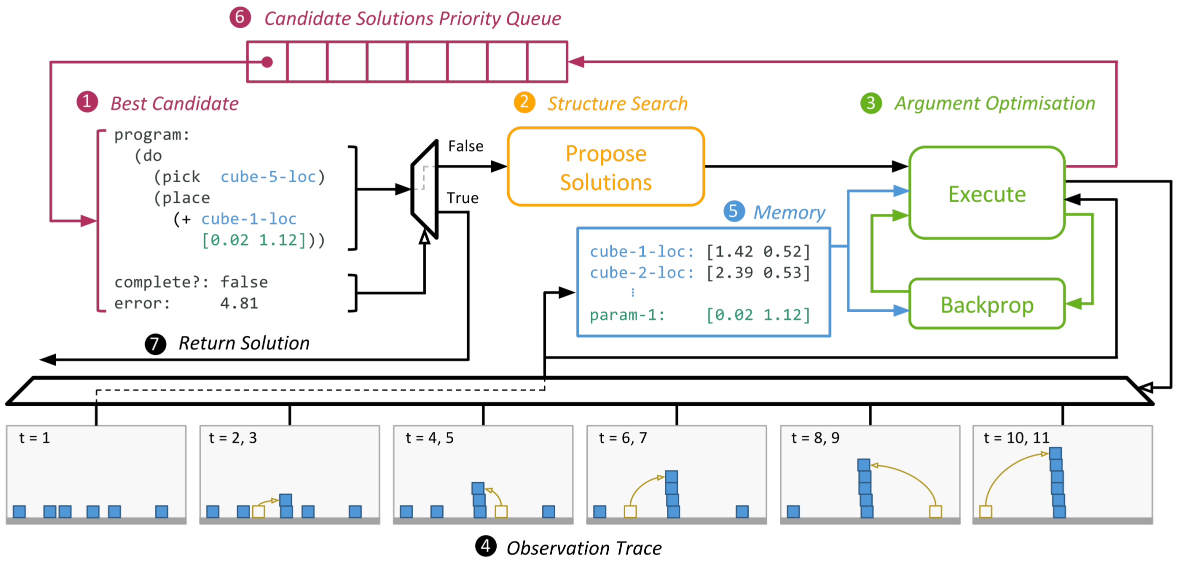


Figure 1. Overall architecture of the π-machine. The current best candidate solution (1) is used to propose new, structurally more complex candidates (2). Each one of the new candidate programs ρ is optimised (3) through gradient descent by comparing its execution trace to the observation trace (4). The observation trace in this case is a demonstration of a tower building task. During execution, the program has access to memory (5) which stores both state variables and induced parameters. All new candidate programs are scored based on their performance and complexity and are inserted in the candidate solutions priority queue (6). Once the execution trace of a candidate matches the observation trace the final solution is returned (7).

memory state configurations, $\mathcal{I}$ is the set of supported instructions and $\varepsilon : \mathcal{M} \times \mathcal{I} \rightarrow \mathcal{M}$ specifies the effect of each instruction. We consider two types of instructions – primitive actions which emulate the execution of $a \in \mathcal{A}$ or arithmetic functions $f \in \mathcal{F}$ such that $\mathcal{I} = \mathcal{A} \cup \mathcal{F}$. Furthermore, a set of observed state variables $\mathcal{M}_v \subseteq \mathcal{S}$, which vary over time, are stored in memory together with a set of induced free parameters $\mathcal{M}_p$. The variables in $\mathcal{M}_v$ form a context which the program will be built on. A custom detector $\mathcal{D}_v$, operating on the raw data stream, could be provided for each variable, thus enabling the user to make queries with respect to different contexts and property specifications.

The execution of a program containing primitive actions results in a sequence of actions. Therefore, we represent a program ρ as a function which maps a set of input variables $x_v \subset \mathcal{M}_v$ and a set of free parameters $x_p \subset \mathcal{M}_p$ to a finite sequence of actions $\hat{a}_1(\hat{\theta_1}), \dots \hat{a}_{T'}(\hat{\theta_{T'}})$. We are interested in inducing a program which minimises the total error between the executed and the observed actions:

$$L(\rho) = \sum_{t=1}^{\min(T,T')} \sigma_{act}(\hat{a}_t, \hat{\theta}_t, a_t, \theta_t) + \sigma_{len}(T, T') \quad (1)$$

The error function σ_{act} determines the difference between two actions, while σ_{len} compares the lengths of the generated and observed action traces. By providing the error functions σ_{act} and σ_{len} one can target different aspects of the observation trace to be explained as they specify when two action traces are equivalent.

4. Method

The proposed program induction procedure is based on two major steps. Firstly, we explain how a given functional program can be optimised such that the loss $L(\rho)$ is minimised. Secondly, we explain how the space of possible program structures can be searched efficiently by utilising gradient information. An architectural overview of the π-machine is provided in Figure 1.

4.1. Program optimisation

Neural networks are naturally expressed as computational graphs which are the most fundamental abstraction in computational deep learning frameworks (Tokui et al., 2015; Bergstra et al., 2010; Abadi et al., 2016). Optimisation within a computational graph is usually performed by pushing the input through the entire graph in order to calculate the output (forward pass) and then backpropagating the error signal to update each parameter (backward pass). A key observation for the development of the π-machine is that computational graphs and functional programs are equivalent as both describe arbitrary compositions of pure functions applied to input data. Therefore, similarly to a computational graph, a functional program can also be optimised by executing the program (forward pass), measuring the error signal and then performing backpropagation to update the program (backward pass).

Forward pass. When a program is executed it is interpreted to a sequence of instructions $i_1, \ldots, i_n \in \mathcal{I}$ which are executed by recursively calling $\varepsilon(\ldots\varepsilon(\varepsilon(\mathcal{M}_1, i_1), i_2)\ldots, i_n)$. $\mathcal{M}_1$ is the initial memory state initialised with the observed variables from s_1 and any induced parameters. The π-machine keeps a time counter t which is initialised to 1 and is automatically incremented whenever a primitive action instruction is executed. If the instruction i_k is a primitive action, $i_k \in \mathcal{A}$, then the π-machine automatically sets $\hat{a}_t = i_k$ and invokes the error function $\sigma_{act}(\hat{a}_t, \hat{\theta}_t, a_t, \theta_t)$, where $\hat{\theta}_t$ has been calculated by previous instructions. If the error is above a certain threshold e_{max} the program execution is terminated and the backward pass is initiated. Otherwise, the time counter is incremented and the values of the variables in $\mathcal{M}_v$ are automatically updated to the new observed state. Essentially, the π-machine simulates the execution of each action reflecting any changes it has caused in the observed state. Alternatively, if the currently executed instruction i_k is a function, $i_k \in \mathcal{F}$, then the resulting value is calculated and i_k, together with its arguments, is added to a detailed call trace χ maintained by the π-machine. Importantly, each function argument is either a parameter or a variable read from memory at time t or the result of another function. All this information is kept in χ which eventually contains the computational tree of the program.

Backward pass. The gradients of the loss function $L(\rho)$ with respect to the program inputs x_v and x_p are required to perform a gradient descent step. Crucially, programs executed by the π-machine are automatically differentiated. The π-machine performs reverse-mode automatic differentiation, similarly to Autograd (Maclaurin et al., 2015), by traversing the call trace χ, and post-multiplying Jacobian matrices. We assume that the Jacobian matrix with respect to every input argument of any function $f \in \mathcal{F}$ or any specified error function σ_{act} is known a priori. Let $f \in \mathcal{F}$ be a function whose output needs to be differentiated with respect to the input arguments. There are three types of derivatives, which need to be considered in order to traverse backwards the entire tree of computations:

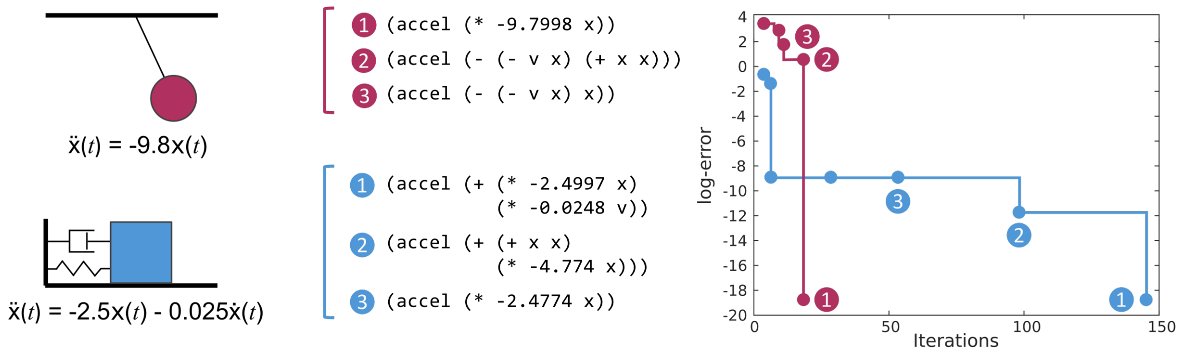
1. Let $g \in \mathcal{F}$, then $\frac{\partial f}{\partial g}$ is the Jacobian matrix of f with respect to the output of g and can be directly calculated.
2. Let $p \in x_p$, then the gradient $\frac{\partial f}{\partial p}$ is calculated by multiplying the corresponding Jacobian matrix of f with the value of p.
3. Let $v \in x_v$, then the gradient $\left.\frac{\partial f}{\partial v}\right|_{t=t_r}$ is calculated by multiplying the corresponding Jacobian of f with the value of the variable at the time it was read from memory t_r.

Gradient descent step. Once the gradient $\nabla_p L(\rho)$ of the loss function with respect to each input parameter $p \in x_p$ is calculated we utilise AdaGrad (Duchi et al., 2011) to update the values of all parameters after each program execution. The gradient $\nabla_v L(\rho)$ with respect to each input variable $v \in x_v$ is also available. However, a variable cannot be simply updated in the direction of the gradient as it represents a symbol, not just a value. Variables can only take values from memory which is automatically updated according to the observation trace during execution. Nevertheless, the gradient provides important information about the direction of change which we utilise to find variables that minimise the loss. Whenever the memory state is automatically updated, a KD-tree is built for each type of variable stored in memory. We assume that the variables in memory are real vectors with different length. So, we represent the KD-tree which stores all D-dimensional variables in memory at time t as $\mathcal{K}_t^D$. If a d-dimensional variable v is to be optimised it is replaced with a temporary parameter p_{temp} initialised with v_t which is the value of v read from memory at the respective time step t. The temporary parameter p_{temp} is also updated with AdaGrad (Duchi et al., 2011). After each descent step, the nearest neighbour of the updated value p'_{temp} is determined by querying the KD-tree with $\mathcal{K}_t^d(p'_{temp})$. If the result of the query is a different d-dimensional variable u then the temporary parameter is immediately set to $p_{temp} = u_t$. As this often shifts the solution to a new region of the error space the gradient history for all parameters $p \in x_p$ is reset. Eventually, when a solution is to be returned, the temporary parameters are substituted with their closest variables according to the respective $\mathcal{K}_t^D$. The forward and backward passes are repeated until the error is below the maximum error threshold e_{max} or a maximum number of iterations is reached. After that the optimised program ρ^* is scored according to its error and complexity, and pushed to a priority queue holding potential solutions.

4.2. Structure search

We represent the space of possible program structures as a graph $G = (T^{AST}, E)$ where each node $T_i \in T^{AST}$ is a valid program abstract syntax tree (AST). There is an edge from T_i to T_j if and only if T_j can be obtained by replacing exactly one of the leaves in T_i with a subtree T_s of depth 1. The program induction procedure always starts with an empty program. So, we frame structure search as a path finding problem, solved through the use of A* search.

Score function. The total cost function we use is $f_{total}(\rho) = C(\rho) + L(\rho)$, where $L(\rho)$ is the loss function defined in equation (1) and $C(\rho)$ is a function which measures the complexity of the program ρ. $C(\rho)$ can be viewed as the cost to reach ρ and $L(\rho)$ as the distance to the desired



```
1 (accel (* -9.7998 x))
2 (accel (- (- v x) (+ x x)))
3 (accel (- (- v x) x))

1 (accel (+ (* -2.4997 x)
            (* -0.0248 v))
2 (accel (+ (+ x x)
            (* -4.774 x)))
3 (accel (* -2.4774 x))
```

Figure 2. The π-machine explaining the behaviour of a pendulum (top) and a linear oscillator(bottom). The best 3 solutions for each system are shown in the middle.

goal. The complexity function $C(\rho)$ is the weighted sum of (i) maximum depth of the program AST; (ii) the number of free parameters; (iii) the number of variables used by the program; the weights of which we set to $w_C = [10, 5, 1]$. These choices ensure that short programs, maximally exploiting structure of the observation trace, are preferred.

Neighbours expansion. When the current best candidate solution is popped from the priority queue, we check if it matches the observation trace according to the input specification. If so, the candidate can be returned as the final solution, otherwise it is used as a seed to propose new candidate solutions. Typically in A* search, all neighbouring nodes are expanded and pushed to the priority queue, which is not feasible in our case, though. Therefore, we utilise the available gradients in order to perform a guided proposal selection. Each leaf in the abstract syntax tree T_ρ of a seed candidate solution ρ corresponds to a parameter or a variable. According to the definition of G we need to select exactly 1 leaf to be replaced with a subtree T_s of depth 1. We select leaf $l \in T_\rho$ according to:

$$l = \underset{x \in x_p \cup x_v}{\arg\max} \, \|\nabla_x L(\rho)\|_2 \tag{2}$$

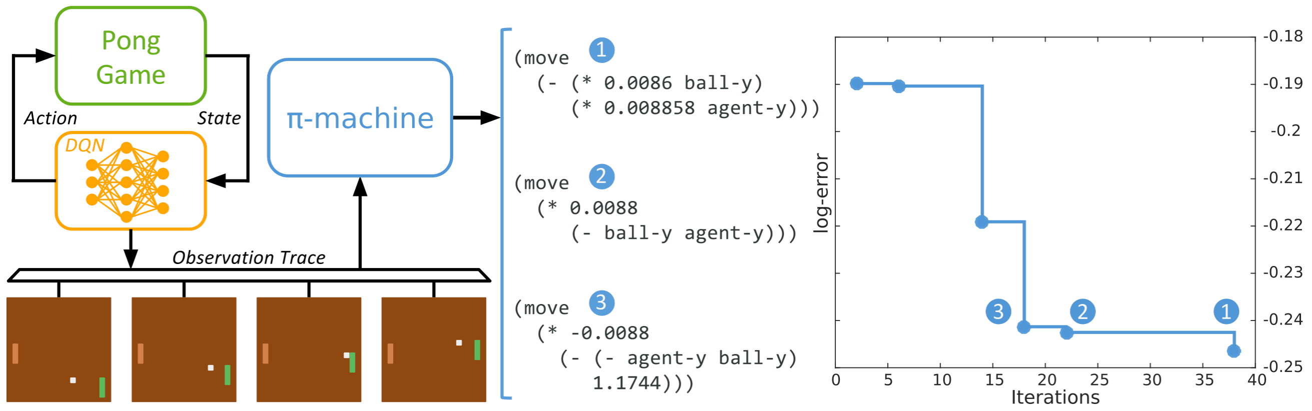
After that, all possible replacement subtrees are constructed. An AST subtree T_s of depth 1 represents a function call. We prune the number of possible functions in $\mathcal{F}$ by ensuring type consistency. Each leaf of T_s can be a parameter or a variable. So, all possible combinations are considered. New variable leaves are initialised to a random variable with suitable type from memory, while new parameter leaves are sampled from the multivariate normal distribution $\mathcal{N}(0, 0.1)$. As a result, if n_f functions are type compatible with l and each function takes n_a arguments at most, then there are $2^{n_a} \cdot n_f$ replacement subtrees, resulting in that many new candidates. All newly proposed candidates are optimised in parallel, scored by f_{total} and pushed to the priority queue.

5. Experimental results

The π-machine is implemented in Clojure, which is a LISP dialect supporting powerful data structures and homoiconic syntax. All experiments are run on an Intel Core i7-4790 processor with 32GB RAM and use the following list of functions, $\mathcal{F}$: vector addition, subtraction and scaling.

Physical systems. Firstly, we apply the π-machine to model learning for physical systems. The transition dynamics of a second order dynamical system is written as $\ddot{\mathbf{x}}(t) = k_1\mathbf{x}(t) + k_2\dot{\mathbf{x}}(t)$, where $\mathbf{x}(t)$ is the state of the system at time t and k_1, k_2 are system coefficients. We have recreated an experiment described in Schmidt & Lipson (2009), where the authors show the learning of physical laws associated with classical mechanical systems including the simple pendulum and linear oscillator. A diagram of these two systems is shown in Figure 2 (left). We set $\mathcal{A} = \{accel(\theta)\}$ where $\theta \in \mathbb{R}$ for both experiments. The observation trace for each system is generated by simulating the dynamics for 1s at 100Hz. We specify the action error function as $\sigma_{act} = \|\hat{\theta} - \theta\|_2$ and set $\sigma_{len} = 0$. In both experiments $\mathbf{x} \in \mathbb{R}$ and $\mathbf{v} = \dot{\mathbf{x}} \in \mathbb{R}$ represent linear position and velocity.

The three best solutions found by the π-machine for each system are shown in Figure 2 (middle). The best solution for each system correctly represents the underlying laws of motion. The program describing the behaviour of the pendulum was induced in 18 iterations, while the linear oscillator program needed 146 iterations. The total number of possible programs with AST depth of 2, given the described experimental setup, is approximately 1.7×10^4. The average duration of an entire iteration (propose new programs, optimise and evaluate) was $0.6s$. Schmidt & Lipson (2009) achieve similar execution times, but distributed over 8 quad core computers (32 cores in total). The experimental results demonstrate that the π-machine can efficiently induce programs representing fundamental laws of physics.



```
(move 1
  (- (* 0.0086 ball-y)
     (* 0.008858 agent-y)))

(move 2
  (* 0.0088
     (- ball-y agent-y)))

(move 3
  (* -0.0088
    (- (- agent-y ball-y)
       1.1744)))
```

Figure 3. The π-machine explaining the behaviour of a DQN agent playing ATARI Pong. The best 3 solutions are shown in the middle.

Deep Q-network. This experiment is based on our view that the core deep neural network based policy learner and the explanation layer play complementary roles. There are numerous advantages to performing end-to-end policy learning, such as DQN-learning from raw video, however, there is also a need to explain the behaviour of the learnt policy with respect to user-defined properties of interest. We consider explaining the behaviour of a DQN agent playing the ATARI Pong game and are interested in the question: how does the network control the position of the paddle in order to hit the ball when it is in the right side of the screen. A diagram of the experimental setup is shown in figure 3 (left). The behaviour of the DQN is observed during a *single* game. Since the environment is deterministic, the state transition function, which generates the observation trace for this experiment, is the policy $\pi(s)$ that the DQN has learnt. We would like to explain the behaviour of the DQN in terms of the position of the opponent, the ball and the DQN agent (so, not just in terms of RAM memory values, for instance). Therefore, the observation trace contains those positions which are extracted from each frame by a predefined detector. We set $\mathcal{A} = \{move(\theta)\}$ where $\theta \in \mathbb{R}$ and represent the discrete actions of the network `left`, `right`, `nop` as $move(1)$, $move(-1)$, $move(0)$ respectively. We specify the action error function as $\sigma_{act} = \|\hat{\theta} - \theta\|_2$ and set $\sigma_{len} = 0$.

The best 3 programs found by the π-machine are shown in Figure 3 (middle), where it took 38 iterations for the best one (average iteration duration 3.2s). By inspecting the second solution it becomes clear that the neural network behaviour can be explained as a proportional controller minimising the vertical distance between the agent and the ball. However, the best solution reveals even more structure in the behaviour of the DQN. The coefficient in front of the agent position is slightly larger than the one in front of the ball position which results in a small amount of damping in the motion of the paddle. Thus, it is evident that the DQN not only learns the value of each game state, but also the underlying dynamics of controlling the paddle. Furthermore, we have tested the performance of an agent following a greedy policy defined by the induced program. In our experiments over 100 games this agent achieved a score of $11.1(\pm 0.17)$. This is not quite the score of $18.9(\pm 1.3)$ obtained by an optimised DQN, but it is better than human performance 9.3 (Mnih et al., 2015). This difference of course emanates from the predefined detector not capturing all aspects of what the perceptual layers in DQN have learnt, so improved detector choices should yield interpretable programs that also attain performance closer to the higher score of the black-box policy.

6. Discussion

The π-machine can be viewed as a framework for automatic network architecture design (Zoph & Le, 2017; Negrinho & Gordon, 2017), as different models can be expressed as concise LISP-like programs. Deep learning methods for limiting the search space of possible programs, which poses the greatest challenge, have been proposed (Balog et al., 2016), but how they can be applied to more generic frameworks such as the π-machine is an open question. The specification of variable detectors not only addresses this issue, but enables the user to make targeted and well grounded queries about the observed data trace. Such detectors can also be learnt from raw data in an unsupervised fashion (Garnelo et al., 2016; Kim & Canny, 2017).

7. Conclusion

In conclusion, we propose a novel architecture, the π-machine, for inducing LISP-like functional programs from observed data traces by utilising backpropagation, stochastic gradient descent and A* search. The experimental results demonstrate that the π-machine can efficiently induce interpretable programs from short data traces.

References

Abadi, Martín, Agarwal, Ashish, Barham, Paul, Brevdo, Eugene, Chen, Zhifeng, Citro, Craig, Corrado, Greg S, Davis, Andy, Dean, Jeffrey, Devin, Matthieu, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

Baier, Christel and Katoen, Joost-Pieter. *Principles of Model Checking*. The MIT Press, 2008.

Balog, Matej, Gaunt, Alexander L, Brockschmidt, Marc, Nowozin, Sebastian, and Tarlow, Daniel. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

Bergstra, James, Breuleux, Olivier, Bastien, Frédéric, Lamblin, Pascal, Pascanu, Razvan, Desjardins, Guillaume, Turian, Joseph, Warde-Farley, David, and Bengio, Yoshua. Theano: A cpu and gpu math compiler in python. In *Proc. 9th Python in Science Conf*, pp. 1–7, 2010.

Bottou, Léon, Peters, Jonas, Candela, Joaquin Quinonero, Charles, Denis Xavier, Chickering, Max, Portugaly, Elon, Ray, Dipankar, Simard, Patrice Y, and Snelson, Ed. Counterfactual reasoning and learning systems: the example of computational advertising. *Journal of Machine Learning Research*, 14(1):3207–3260, 2013.

Brendel, William and Todorovic, Sinisa. Learning spatiotemporal graphs of human activities. In *Computer vision (ICCV), 2011 IEEE international conference on*, pp. 778–785. IEEE, 2011.

Cai, Jonathon, Shin, Richard, and Song, Dawn. Making neural programming architectures generalize via recursion. In *International Conference on Learning Representations (ICLR)*, April 2017.

Devlin, Jacob, Uesato, Jonathan, Bhupatiraju, Surya, Singh, Rishabh, Mohamed, Abdel-rahman, and Kohli, Pushmeet. Robustfill: Neural program learning under noisy i/o. *arXiv preprint arXiv:1703.07469*, 2017.

Duchi, John, Hazan, Elad, and Singer, Yoram. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

Garnelo, Marta, Arulkumaran, Kai, and Shanahan, Murray. Towards deep symbolic reinforcement learning. *arXiv preprint arXiv:1609.05518*, 2016.

Glimcher, Paul. *Foundations of Neuroeconomic Analysis*. Oxford University Press, 2011.

Graves, Alex, Wayne, Greg, and Danihelka, Ivo. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Graves, Alex, Wayne, Greg, Reynolds, Malcolm, Harley, Tim, Danihelka, Ivo, Grabska-Barwińska, Agnieszka, Colmenarejo, Sergio Gómez, Grefenstette, Edward, Ramalho, Tiago, Agapiou, John, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476, 2016.

Grefenstette, Edward, Hermann, Karl Moritz, Suleyman, Mustafa, and Blunsom, Phil. Learning to transduce with unbounded memory. In *Advances in Neural Information Processing Systems*, pp. 1828–1836, 2015.

Gunning, David. DARPA Explainable Artificial Intelligence (XAI) Program. `http://www.darpa.mil/program/explainable-artificial-intelligence`, 2016. [Online; accessed 14-June-2017].

Harrison, Brent, Ehsan, Upol, and Riedl, Mark O. Rationalization: A neural machine translation approach to generating natural language explanations. *arXiv preprint arXiv:1702.07826*, 2017.

Hendricks, Lisa Anne, Akata, Zeynep, Rohrbach, Marcus, Donahue, Jeff, Schiele, Bernt, and Darrell, Trevor. Generating visual explanations. In *European Conference on Computer Vision*, pp. 3–19. Springer, 2016.

Kim, Jinkyu and Canny, John. Interpretable learning for self-driving cars by visualizing causal attention. *arXiv preprint arXiv:1703.10631*, 2017.

Lake, Brenden M., Salakhutdinov, Ruslan, and Tenenbaum, Joshua B. Human-level concept learning through probabilistic program induction. *Science*, 350(6266): 1332–1338, 2015.

Lei, Tao, Barzilay, Regina, and Jaakkola, Tommi. Rationalizing neural predictions. *arXiv preprint arXiv:1606.04155*, 2016.

Letham, Benjamin, Rudin, Cynthia, McCormick, Tyler H, Madigan, David, et al. Interpretable classifiers using rules and bayesian analysis: Building a better stroke prediction model. *The Annals of Applied Statistics*, 9(3): 1350–1371, 2015.

Ling, Wang, Yogatama, Dani, Dyer, Chris, and Blunsom, Phil. Program Induction by Rationale Generation:Learning to Solve and Explain Algebraic Word Problems. *arXiv preprint arXiv:1705.04146*, 2017.

Maclaurin, Dougal, Duvenaud, David, and Adams, Ryan P. Autograd: Effortless gradients in numpy. In *ICML 2015 AutoML Workshop*, 2015.

Mnih, Volodymyr, Kavukcuoglu, Koray, Silver, David, Rusu, Andrei A, Veness, Joel, Bellemare, Marc G, Graves, Alex, Riedmiller, Martin, Fidjeland, Andreas K, Ostrovski, Georg, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

Negrinho, Renato and Gordon, Geoff. Deeparchitect: Automatically designing and training deep architectures. *arXiv preprint arXiv:1704.08792*, 2017.

Reed, Scott and De Freitas, Nando. Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279*, 2015.

Ribeiro, Marco, Singh, Sameer, and Guestrin, Carlos. Why should i trust you?: Explaining the predictions of any classifier. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2016.

Schmidt, Michael and Lipson, Hod. Distilling free-form natural laws from experimental data. *Science*, 324 (5923):81–85, 2009.

Shapiro, Ehud Y. *Algorithmic Program Debugging*. MIT Press, Cambridge, MA, USA, 1983.

Solar-Lezama, Armando, Tancau, Liviu, Bodik, Rastislav, Seshia, Sanjit, and Saraswat, Vijay. Combinatorial sketching for finite programs. *ACM SIGOPS Operating Systems Review*, 40(5):404–415, 2006.

Tokui, Seiya, Oono, Kenta, Hido, Shohei, and Clayton, Justin. Chainer: a next-generation open source framework for deep learning. In *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*, 2015.

Vaandrager, Frits. Model learning. *Commun. ACM*, 60(2): 86–95, January 2017.

Yang, Hongyu, Rudin, Cynthia, and Seltzer, Margo. Scalable bayesian rule lists. *arXiv preprint arXiv:1602.08610*, 2016.

Yin, Pengcheng and Neubig, Graham. A syntactic neural model for general-purpose code generation. *arXiv preprint arXiv:1704.01696*, 2017.

Zahavy, Tom, Ben-Zrihem, Nir, and Mannor, Shie. Graying the black box: Understanding dqns. *arXiv preprint arXiv:1602.02658*, 2016.

Zeiler, Matthew D and Fergus, Rob. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pp. 818–833. Springer, 2014.

Zoph, Barret and Le, Quoc. Neural architecture search with reinforcement learning. In *International Conference on Learning Representations (ICLR)*, April 2017.