AI-Driven Course of Action Generation Using Neuro-symbolic Methods

Michael Harradon, Kevin Golan, Oliver Daniels-Koch, Avi Pfeffer, and Robert Hyland
Charles River Analytics, Inc.
Cambridge, MA
[mharradon, kgolan, odanielskoch, apfeffer, rhyland]@cra.com

ABSTRACT

Artificial intelligence (AI)–based systems show great promise for supporting complex decision-making and planning. AI systems can consider a massive option space that far exceeds current human processes. Notably, AI systems, particularly deep reinforcement learning (DRL), achieved expert levels of play for strategy games, generating innovative strategies by exploring numerous courses of action (COAs) to make successful strategic choices in complex scenarios. The opportunity exists for AI systems to assist human planning staffs with constructing more high-quality plans, analyzing plan strengths and weaknesses more deeply, and exploring a larger number of plan alternatives in a fixed planning time. AI-based modeling and simulation could accelerate COA planning activities that require reasoning across multiple, interconnected domains. Such planning support must consider interacting effects across physical domains (e.g., air, land, sea, undersea) and interacting support functions (e.g., logistics, communications)—a massive action space to consider—which exceeds the action space processed by state-of-the-art game-playing systems. This paper reports on a new DRL approach, called neural program policies (NPPs), which incorporates structure and domain-specific information into policies described by deep neural networks to vastly reduce the action space into a learned, compact, and meaningful set. We first describe the policy domain-specific language (DSL) that abstracts the actions and observations employed by a deep reinforcement learner. Then, we apply NPPs to two problem categories—control problem benchmarks provided by OpenAI gym and multi-domain reasoning using a StarCraft II simulation, extended with sea, undersea, and novel support functions. We conclude with results for (1) generating multi-domain COA traces and (2) NPP-based agent performance (e.g., running a surrogate model ~10,000x faster than real time, vastly reducing the action space, and enabling the interpretation of AI-generated COA traces). The resulting system supports a human-AI team paradigm to increase the number and quality of multi-domain plans considered.

ABOUT THE AUTHORS

Michael Harradon is a Senior Scientist at Charles River Analytics. He holds a ME in Computer Science, as well as BS degrees in Computer Science, Electrical Engineering, and Physics from MIT. His research interests span probabilistic modeling/programming, deep learning and reinforcement learning, and game theory.

Kevin Golan is an AI Scientist at Charles River Analytics. He holds an MS in Electrical Engineering from ETH Zürich and a BE in Electrical Engineering from the University of Manchester, UK. His technical expertise is in signal processing, machine learning, and probabilistic modeling.

Oliver Daniels-Koch is an AI Scientist at Charles River Analytics. He holds a BS in Computer Science from Brandeis University. His technical expertise is in reinforcement learning, natural language processing, and AI safety.

Avi Pfeffer, PhD, is Chief Scientist at Charles River Analytics. His research spans a variety of computational intelligence techniques including probabilistic reasoning, machine learning, and computational game theory.

Robert Hyland is a Principal Scientist and Director of Program Transition at Charles River Analytics. He holds an MS in Computer Science with a focus on AI. His areas of interest include human-AI teaming, behavioral modeling, and mixed-initiative decision support.

AI-Driven Course of Action Generation Using Neuro-symbolic Methods

Michael Harradon, Kevin Golan, Oliver Daniels-Koch, Avi Pfeffer, and Robert Hyland
Charles River Analytics, Inc.
Cambridge, MA
[mharradon, kgolan, odanielskoch, apfeffer, rhyland]@cra.com

INTRODUCTION

Artificial intelligence (AI) has achieved remarkable success in recent years across a wide range of applications, including vision, language generation, decision-making, autonomy, and robotics. In this paper, we focus on AI in decision-support, specifically using deep reinforcement learning (DRL), where effective modeling and simulation techniques are critical enablers. In DRL, an agent is tasked with finding the optimal sequence actions that will lead to maximized rewards given some environment. DRL drew much attention, following the defeat of professional Go champions by DeepMind's AlphaGo AI model. Since then, DRL systems have outperformed humans at increasingly complex strategic games, often with creative, unexpected moves. Such successes naturally led researchers and experts to pursue the potential of DRL in broader decision-making contexts, and this work has led to new methods enabling DRL to integrate with and expand learning and simulation techniques (Verma et al., 2019; Schrittwieser et al., 2020; Pateria et al., 2021). With these recent advances, DRL also appears particularly well suited as an assistive system for decision-making because it analyzes the strengths and weaknesses of a wide set of decision pathways at each model training iteration. The opportunity exists for a DRL-based AI system to support planning staffs, such as those who plan complex operations and logistics, dealing with task allocation problems in long-duration, real-world, multi-domain scenarios. In this paper, we explore the power of DRL-based AI agents to enable a human-AI team paradigm for decision-making, specifically for generating innovative plan strategies by exploring numerous courses of action (COAs) to increase the number and quality of multi-domain plans considered for complex scenarios. We define a COA as a time-phased or situation-dependent plan that features effective strategies to achieve human-specified and multi-domain objectives with dynamic uncertainty. AI agents can be thought to generate COAs if they output multiple action traces (i.e., "execution" paths recorded during simulation runs with respect to agent-chosen actions) that achieve the human-specified objectives through sequenced strategies and tactics. We call such AI-agent outputs *COA traces*. We experiment with how these COA traces can assist human planning staffs for constructing more high-quality plans, analyzing plan strengths and weaknesses more deeply, and exploring a larger number of plan alternatives in a fixed planning time. Such DRL-based modeling and simulation could accelerate COA planning activities, and the generation of creative or surprising plan strategies can be useful for training staff to develop more robust plans.

Constructing agents with planning capabilities is a long-standing challenge for AI, and it is especially challenging for agents to generate complex multi-domain COAs. Traditional DRL techniques can be difficult to apply in environments or games with complex structures where widely different decisions need to be made. When developing a DRL model, a key challenge is to define the set of actions that may be relevant for the agent's problem and the reward functions that will steer the model toward learning an optimal policy. This challenge is further complicated if the environment includes uncertainty, such as environments or games described by partially observable Markov decision processes (POMDPs). Agents are often tasked to choose among actions that are completely orthogonal to each other in POMDPs. For instance, if an agent is learning to play StarCraft II, a real-time strategy computer game for commanding science fiction armies that has served as a productive AI agent testbed for many years, the agent must optimally construct a COA trace that simultaneously accomplishes multiple objectives: gathering and managing resources, constructing multi-domain forces, executing maneuvers, scouting, and executing attack and defense plans. These, in turn, may involve several subroutines that need to be well executed, under dynamic uncertainty. When there is multimodality in the tasks that need to be solved by the agent, it makes little sense to learn an all-encompassing policy that considers all possible actions for every observation. Some observations are more informative for action selection than others, and that varies by time and context. We refer to these problems where action spaces and observation structures vary over time as *open-structured POMDPs.* One straightforward approach to address this class of problems would be to construct different neural networks tailored for different tasks (e.g. one for maneuvers and one for effective attack

plans). However, doing so is challenging because it requires both subject matter expertise and deep learning expertise to carefully define the relevant policies and rewards and translate these policies into architectures that can jointly be learned within the environment.

In this paper, we introduce neural program policies (NPPs), a neuro-symbolic hybrid-AI architecture designed to address the challenges described above. Specifically, NPPs provide a framework that abstracts away the orchestration and design of deep learning architectures that are tailored to learn and act in a dynamic environment with multidimensional uncertainty and, instead, allows AI developers to focus on targeted policy design and representation via an expressive domain-specific language (DSL). The DSL is used to wrap the reinforcement learning problem into a more explainable representation that is easier to solve, understand, and analyze—enabling more end-user-oriented transparency than black-box end-to-end deep learning systems. As part of the NPP framework, AI developers and/or subject matter experts (SMEs) can encode prior domain knowledge using symbolic AI representations in a policy program (PP) DSL to express knowledge or insights about the problem domain (e.g., by restructuring the action space of the agent into a smaller or more easily learned set). This NPP neuro-symbolic framework can enable significantly more data-efficient learning. A summary of our contributions follows:

1. We design and implement a DSL that allows AI developers to describe structured policy and generalized COA templates that readily incorporate SME knowledge to express fundamental domain concepts and decompose the decision problem into learnable constructs.
2. We develop an adaptive deep neural network architecture that uses the DSL-encoded knowledge to generate useful and creative COA traces by fitting the set of functional approximations required to describe policies, predictions, and other needed outputs for the DSL-defined open-structure POMDP.
3. We run a series of experiments on two problem categories: (1) demonstrating effectiveness for benchmark control tasks, and (2) generating high-quality COA traces using a heavily modified version of StarCraft II with new physical domains and support functions (e.g., resource management, communications, and sensors).

In the following sections we discuss NPP technical details. We first provide a motivating example to lay the groundwork for what a complex, multi-domain DRL problem looks like and what typical DRL shortcomings emerge when interacting with more complex environments. The motivating example also illustrates ways an NPP application can materially help a human planning team. We then discuss the technical details underlying NPPs, providing some intuition about the DSL's policy representation, and we introduce how programs written in this DSL encapsulate open-structured POMDPs. We follow by describing the NPP architecture, which features an autoregressive core with a multiheaded structure for its observation and policy models. We then present a selection of experimental results showing the effectiveness of NPPs on DRL benchmarks such as the control problems provided by the OpenAI gym, as well as COA trace quality and performance results using an extended, multi-domain StarCraft II. Lastly, we discuss NPP's contributions as related to state-of-the-art DRL approaches, and we provide concluding remarks.

Motivating Example

In this section, we present a motivating example designed to advance the state of the art (SoA) in AI-based COA trace generation. We use computer games as the foundation for our motivating example and experimental concept because they provide useful AI benchmarks. For example, AI algorithms have demonstrated expert-level performance in board games and retro video game environments such as Atari (Ye et al., 2021). And perhaps more relevant for this work, DeepMind's AlphaStar reached grandmaster-level performance in StarCraft II (Vinyals et al., 2019). Though impressive, these SoA approaches possess shortcomings in their COA trace generation capability and their performance. For example, they require millions (or billions in the case of AlphaStar) of environmental steps to train on new problems, which is too slow to assist many human planning processes, and they do not express their plan or strategies in human-understandable representations. To highlight the differences in our approach, we use and extend StarCraft II. Here, AI agents are required to learn and present COA traces to assist human planning staffs in constructing more high-quality plans, analyzing plan strengths and weaknesses more deeply, and exploring a larger number of plan alternatives in the same amount of planning time as allotted to human staffs. The measurements of these actions are described in the results section. Key elements of the motivating example and experimental concept include the following:

- **A Challenge Scenario Requiring Novel Solutions**. We implemented a complex challenge scenario, created by human SMEs—multi-domain scenario design experts, which involved thousands of multi-domain units, consisting of various types of air, land, sea, and undersea assets. The scenario purposely disadvantaged one side (Red), so human or AI planners would require novel strategies and tactics to defeat the stronger adversary (Blue). For example, Red would need to conduct a multistaged application of novel strategies and clever resource management to defeat Blue. In our experimental concept, the AI agent plays Red to exploit weaknesses and defeat a Blue COA, so planners can improve the Blue COA under consideration.
- **Multi-domain Game Environment**. To better reflect real-world uncertainty and interconnections among multiple domains, we employ a heavily modified version of StarCraft II extended with specialized, higher-fidelity models to add sea and undersea physical domains, as well as model sensors, communication, and resource management. The AI-agent must learn environmental rules and winning strategies from multiple adjudicating models that add multidimensional uncertainty well beyond StarCraft II's built-in "fog of war."
- **Generation of COAs Faster than Human Planners**. To better reflect real-world, time-phased COAs that are robust to adversary and environmental uncertainty, the scenario spans longer periods of time (days of real-time play) compared to traditional StarCraft II games (15–20 minutes for real-time multiplayer games). The AI agent needs to generate high-quality plans that are robust to adversary options and generate many alternatives. The goal is to generate more high-quality COA traces than a human staff could generate in the same amount of time. To achieve this, the AI agent must feature a high-performing sample efficiency (i.e., the number of times the agent observes the simulation environment states) to achieve useful training times.
- **Human-Understandable Winning Insights**. The AI agent needs to both quickly demonstrate COA traces that defeat Blue and convey the strategies used (and why they worked) to assist a human planning process. This requires a series of plan visualization techniques that allow human planning staff to quickly understand plan alternatives and select one or more winning COA traces, perhaps more importantly, to deeply explore strengths and weaknesses of the adversary's plan options and gain insights for reducing the adversary's advantage. Based on the AI-generated output, the human staff selects a COA trace that exhibits winning insights and possesses a multiphased plan that reliably leads to victory.

The following subsections describe the key technical innovations for implementing the above motivating example and experimental concept to achieve the COA generation goals also listed above.

Neural Program Policies

To achieve the stated COA generation goals, AI agents must address corresponding technical challenges, which are particularly important to advancing neuro-symbolic AI approaches, including (1) learning a vastly reduced action space to lower training time, (2) incorporating domain-specific knowledge into machine learning (ML) methods to scale to large scenarios, (3) explaining what is learned and support human planning processes, and (4) reasoning over open-structure problems (POMDPs) to manage uncertainty and interconnections among multiple domains.

As discussed above, NPPs offer tools that allow AI developers to define action sets and subpolicies that are targeted and effective for domain-specific tasks. This is primarily accomplished by authoring a PP. PPs are specified in a DSL that allows the developer to write programs that (1) *interface with the simulation* in which the agent operates in; (2) *transform the open-structured POMDP* into a more explainable, easier solved DRL representation; and (3) *incorporate domain expertise* into the way the agent should act within the modified representation. The DSL is used to write programs that *describe a policy skeleton* that the agent should follow for various AI developer–defined domains. NPPs integrate this PP DSL into a DRL architecture.

Our PP DSL has four instructions that allow developers to construct a PP and transform the DRL problem: OBSERVE, ACT, REWARD and DONE. OBSERVE is a function called with arguments representing information that the agent should use to update its internal recurrent state—for the purpose of affecting future actions or other predictions. In our current implementation OBSERVE must be passed an array of numbers, though the shapes of those arrays can vary freely during execution. ACT is a function that takes some specification of an action space and then returns a selection from that space chosen by the deep neural network (DNN) policy. In our current implementation, we focus on finite, discrete action spaces. REWARD and DONE have the standard interpretations as in ordinary DRL—specifying a reward to provide to the agent or signaling the completion of an episode, respectively. Note, an episode is the sequence

of interactions between an agent and its simulation, starting from an initial state and ends at a terminal state. Our current implementation embeds the DSL in the Python programming language, though it can be embedded in any general-purpose programming language, as only forward execution semantics are required to train or run NPP agents.

With a PP designed to effectively support these instructions, developers can focus agent learning on salient factors in the domain via targeted actions, observations, and rewards based on expertise or insight. For instance, one can strictly enforce that agents employ certain behaviors that would otherwise need to be learned over many episodes. For example, in StarCraft II one could write subroutines so that the agent pursues strategy A when encountering enemy entities on land and pursues strategy B when confronting enemies in the air. Furthermore, unlike other approaches, PPs can employ the full expressivity of the host programming language, including stateful computation, library calls, and even interface with other ML or AI systems.

Figure 1 shows a basic application of the DSL constructs in StarCraft II. These include ACT (shown in red), OBSERVE (shown in blue), and REWARD (shown in green) used to write subroutines in a PP. This implementation demonstrates simple agent subroutines to select troops and move them to a destination, learning optimal movement policies conditioned on observations. The OBSERVE instruction is used to inform the agent about the state of the environment. `NPP.discrete` is a specialization of ACT enabling the selection of actions from a discrete action space.

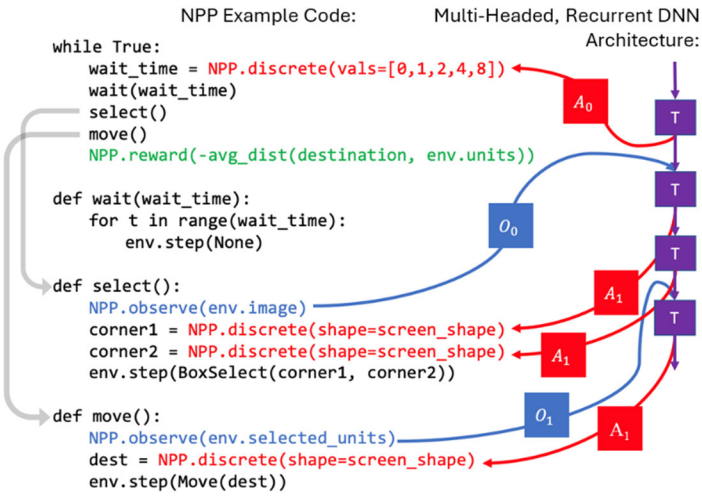

Figure 1: A simple NPP example using the PySC2 action and observation space directly; adaptive multiheaded recurrent architecture is shown in purple. Here, subscripts on models (e.g., A, O) indicate the type of model and the coorsponding type of the action space or observation.

One consequence of employing PPs is that, in general, PP-defined POMDPs are open structured. Since this involves varying action spaces and observation dimensions (e.g., in the motivating example, resource gathering observations will likely have different dimensions than enemy movement observations), we must construct a DNN architecture that can handle heterogenous representations. Furthermore, the expressiveness of the PP DSL requires that the architecture handle new structures on the fly, as future action and observation representations (and their orderings) cannot be known precisely in advance without executing the PP, which may include coupled execution with a simulator or external environment. To address this, NPPs reactively construct various "head" models of different shapes to match each action type and observation type encountered in PP execution. These head models then share a common fixed-sized core recurrent architecture that allows for learning long-range dependencies between observations and policy outputs. Figure 1 shows a schematic depiction of the unrolled, dynamically instantiated recurrent architecture alongside the previously described PP. All boxes represent a neural network. The red boxes represent "action models," the blue boxes are "observation models," and the purple boxes represent the unrolled recurrent neural network. In practice, our implementation uses a variant of a recurrent model with dual spatial and vector hidden state representations.

In addition to learning policy and value outputs at each time step, our NPP learns predictive distributions for each observation that arrives, in this case learning to predict StarCraft II and adversary dynamics. Via the recurrent architecture, these predictions are made autoregressively by conditioning on all past observations. These include observations of the agent's own actions, as the potential for repeated action "types" makes this a natural choice. As a result, the autoregressive predictive model describes the full dynamics of the environment. We similarly learn a predictive model for step-by-step rewards. Both models take the form of diagonal Gaussians with means and variances parameterized by the outputs of DNN "heads" taking the recurrent hidden state as input. These predictive models form

the basis for a "surrogate" model of the simulation, providing a source of data for the NPP to train against in lieu of actual simulation samples. This can be run much faster than many simulations via fast forward passing of DNNs. Furthermore, as the predictions are only made (and only needed) when actions or observations occur, the surrogate model "steps" are defined as PP abstractions that occur in execution rather than the native simulation stepping time. In later results we will refer to measures of this sort as "episode duration." These factors drive the exceptional speedup of the surrogate model—in practice reaching relative speedups of 1,000x or more over the reference simulation. For instance, in our control examples, the pulse-width modulation PP can choose subroutines in three or four actions that fully define control outputs of the simulation for hundreds of environment steps (without further learned actions).

METHODS

For our experiments we demonstrate NPPs on two problem categories. First, we evaluate performance on several classic control examples, which provide well-established metrics to benchmark DRL performance. Second, we consider performance on a custom-designed StarCraft II scenario, characterized in the above motivating example.

For our control examples we compare two agents with different PPs. The first uses the primitive action space and observation space from the domain and thus implements a standard recurrent actor-critic agent for the task. We refer to this as the Baseline agent. The second PP utilizes NPP's unique features. It encodes domain knowledge as a subpolicy to enable alternating output between two values with some chosen frequency and duty cycle. This is a natural fit for many of the classic control problems, where optimal output often involves near-resonant stimulation of some form. This second agent is referred to as the "pulse-width modulation" (PWM) agent, and it exercises NPP-specific neuro-symbolic capabilities. We include a simplified version of the source PP code (Figure 2) for an PWM NPP agent, as the details are illustrative of many types of NPP applications.

```
def run_policy_program(self):
    while True:
        self.observe(self.last_obs)

        on_action = self.act(DiscreteAction(shape=self.action_space_shape))

        # Use prior specification to prevent off action from being the same
        # Since duty cycles of 1.0 are allowed, this would have redundant effect
        mask_on_action = self.all_but_on_action(on_action)
        off_action = self.act(DiscreteAction(shape=space_shape,
                                             name='action',
                                             log_prior=mask_on_action))

        # On times and off times for PWM are chosen from a logarithmically spaced set
        # This covers a wider range of values with fewer finite choices
        on_time = self.act(DiscreteAction(vals=self.log2_space(MAX_TIME_LOG2))
        off_time = self.act(DiscreteAction(vals=[0] + self.log2_space(MAX_TIME_LOG2)))

        # Similarly the options for number of periods are spaced logarithmically
        n_periods = self.act(DiscreteAction(vals=self.log2_space(MAX_N_PERIOD_LOG2)))

        self.pwm(on_action,
                 off_action,
                 on_time,
                 off_time,
                 n_periods)

def log2_space(self, maxval):
    return [2**i for i in range(maxval)]

def all_but_on_action(self, on_action):
    mask_on_action = np.zeros(self.action_space_shape)
    mask_on_action[on_action] = -np.inf
    return mask_on_action

def pwm(self, on_action, off_action, on_time, off_time, n_periods):
    for i in range(n_periods):
        for j in range(on_time + off_time):
            if j < on_time:
                self.envstep(on_action)
            else:
                self.envstep(off_action)
```

Figure 2: Simple "pulse-width modulation" (PWM) NPP

In our second problem category, we train NPPs on a large, long-duration, multi-domain scenario that considers thousands of units running in a heavily modified version of StarCraft II, extended with specialized, higher-fidelity models for sensors, communication, and resource management. We use a preestablished reference COA for one set of forces (Blue) and then allow the NPP to learn to control the other (Red). The PP we designed includes many dozens of subroutines for issuing commands to different sets of multi-domain units and scheduling their behavior over time.

RESULTS

Control Problems

We provide results on three well-established control problems of increasing complexity—Cart Pole, Acrobot, and Mountain Car. For easy-to-understand, animated depictions of each problem visit gymnasium.farama.org. In Cart Pole shown in Figure 3 (left), a pole is connected to a cart via a yellow joint which is unactuated. The objective of this control task is to actuate the left-and-right movements of the cart so that the pole does not fall over. In Acrobot (Figure 3, center), there is a chain composed of two teal links via a yellow joint. The yellow joint on the top end of the chain

is fixed but the link below freely swings, whereas the connecting yellow joint is actuated. The goal in this problem is to apply torque on the central actuated joint so that the free end swings above some target height. In the Mountain Car control problem (Figure 3, right), a car is placed at the center of a valley between two hills. The objective is to get the car to reach the top of the right hill. To do so, the agent must learn to roll back and forth to reach the hilltop. In both Acrobot-v1 and MountainCar-v0 no reward is provided until reaching a goal state, often resulting in episodes terminating with zero reward and requiring long training times.

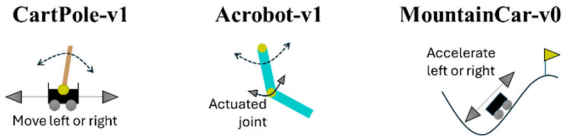

Figure 3: Illustrations of DRL control tests

For each control task, we show training curves for both the PWM NPP agent and the Baseline agent, measuring the *mean episode reward* over the course of training. This is a standard DRL metric, and it is one of the most direct measures of how well an agent is performing in terms of the task it has been trained to do. For this metric, high-performing agents reach a maximum reward and stabilize as quickly as possible. Here, we run experiments on many fewer episodes than typical DRL experiments on these problems (~10x–100x shorter than typical DRL training times for these problems). Across all three control problems (Figure 4) the PWM NPP agents rapidly achieve high reward levels in a stable fashion, evidenced as "plateauing" of the reward curve on the right-hand side of the graphs. In contrast, the Baseline agent does not achieve high performance in the limited number of training episodes, as it takes 10x-100x longer to reach the optima. Note that the x-axis is measured in training steps (sequential parameter updates by gradient descent), so the PWM agent learns in less wall-clock time and with fewer episodes.

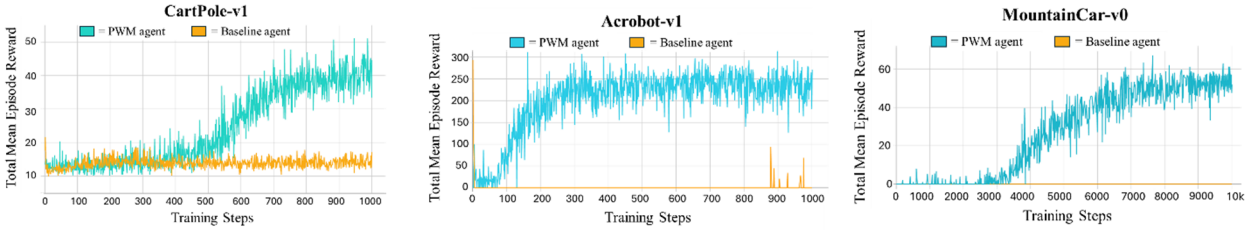

Figure 4: Plots of the "mean episode reward" evaluation metric for agent performance show the PWM NPP agent outperforming the standard Baseline agent across increasing complex control task problems

Results on MountainCar-v0 are of particular note (Figure 5). Mountain Car in its original configuration only provides a nonzero reward after a car successfully oscillates back and forth over many steps. Random exploration has an extremely low likelihood of achieving a reward, and as a result it is very difficult to train standard DRL approaches against it without more sophisticated exploration strategies or reward shaping. In contrast, the PWM NPP has some sequences of choices to make the cart "escape" the valley much more rapidly. Quickly learning good choices is shown in Figure 5 with an estimate for the log of the tree width (left) and the total number of actions and observations in an episode (right). Not only does PWM NPP decrease episode duration (left, in terms of number of actions taken in the PP), but it also produces shorter episodes than Baseline (right), as the agent selects longer-duration output sequences that hit resonant frequencies for multiple periods. We say more on these metrics in the StarCraft II experiment section.

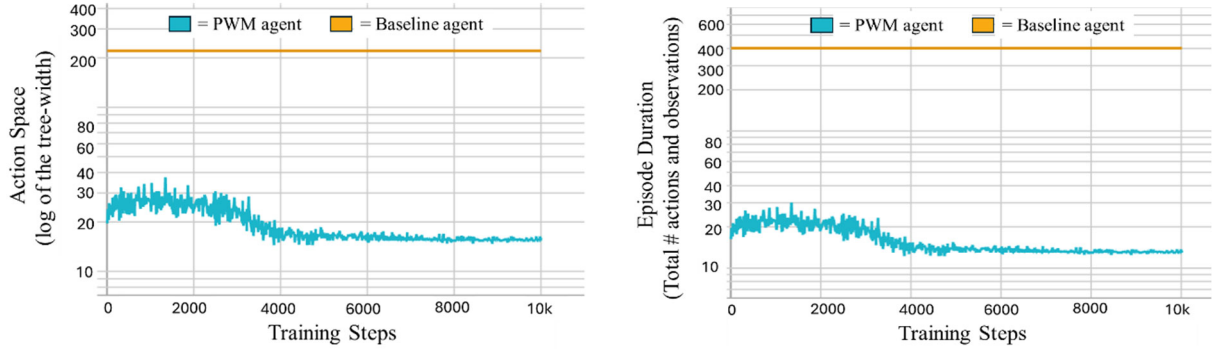

Figure 5: For training standard DRL appraoches, such as Baseline (orange), MountainCar is very difficult without more sophisticated exploration strategies, very slow to learn action space (orange left) because episodes terminate by hitting max time (orange right) versus receiving a task completion reward; in contrast, the NPP agent learns a compact action space (blue left) and finishes in shorter episode duration (blue right)

Extended Multi-Domain StarCraft II

In StarCraft II we constructed a scenario incorporating a fixed initial set of assets positioned across a map. We encoded a policy for an AI agent (Red) whose performance was deemed effective by experts and players. A team of SMEs assessed AI-agent-generated COA traces based on COA quality, effectiveness, and explainability. Compelling COA quality is measured through statistically significant COA improvements, such as objective completion speed, resource usage, and/or reduced adversary resiliency, all while achieving game-winning objectives (i.e., Red COA traces that defeat the superior Blue forces) compared to a baseline high-performing, human-authored Red COA. Effectiveness is measured by differences in total unit losses on each side. Explainability is measured by the ability of SMEs to find high-quality COA traces (i.e., COA traces that "won") and obtain insights, such as understanding the sequencing and application of compelling strategies and understanding why they worked. Insights were ranked high only if SMEs could apply improvements to the original human-authored (Blue) COA to avoid the superior force being defeated.

The AI agent generated several thousand COA traces. However, only a very small fraction of COA traces "won," and many traces were variations of the same strategy, forming approximately five clusters. Three clusters contained COA traces that met or exceeded all assessment criteria, and in some cases, featured surprising uses of resources. One exemplary COA trace included a pincher move that probed the "front" of the adversary's command element with high risk, high-cost moves to reduce resilience of command defenses, while assembling an attack force to exploit weaknesses in the rear, destroying the command. A second exemplary COA trace featured a successful frontal attack with a novel aggregation of attack assets. A third exemplary COA trace employed a long game of probing lightly-defended adversary assets which restricted re-supply and formed a dragnet that destroyed a key adversary asset, preventing the adversary from building up to a victory condition. These COA traces provided key insights to the SME team, as they were able to explore plan strengths and weaknesses more deeply and more quickly than a human planning staff. Planning staffs often require weeks of effort to conduct a fraction of the detailed simulation analysis generated by the NPP AI agent. Ultimately the SMEs were able to apply improvements to their original (Blue) COA to remove the weaknesses discovered and exploited by the AI agent. In fact, the SMEs derived insights from all five clusters, even the two that did not meet all assessment criteria.

The successful COA traces were made possible by addressing the technical challenges of training time speedups, scaling to large scenarios, vastly reducing the action space, adding explanatory capabilities to DRL-based architectures, and reasoning over open-structure problems (POMDPs).

Despite the massive size of the scenario and the astronomical action space of the extended multi-domain StarCraft II, the AI-agent generated high-quality COA traces in less than one day of wall-clock time execution on a standard workstation-grade machine (i.e., a single NVIDIA A100 GPU, 32 CPU cores, 64 GB RAM). Tens of episodes of our large-scale StarCraft II scenario also complete in one day. We also observed that the accuracy of the surrogate model defined by the autoregressive predictions increased on a similar one-day timescale (as shown in Figure 6). The cross-entropy between predicted and actual outcomes of scalar feature variables approaches ~2.2, corresponding to approximately 1% relative error based on our estimates of feature variance. Evaluations are made online against simulation observables as they arrive, so measurements are against outcomes prior to training. Outcomes cluster in time (rapid changes in the orange line) due to common simulation durations (~12 hours). Multiple instances of the scenario are run at 24x parallelism. As shown in Figure 6 (bottom right), the episode reward also increases on a similar timescale (~one day).

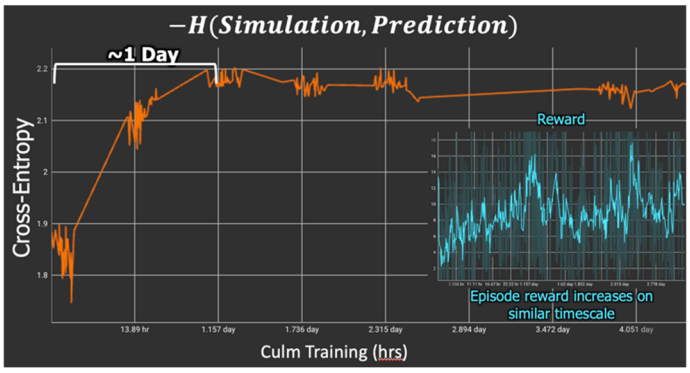

Figure 6: In one training day, our surrogate model accurately predicts StarCraft II outcomes with ~1% relative error, shown via cross-entropy between true simulation observables and predicted distribution of observables

The dramatic speed of the surrogate model enabled the evaluation of 1,000x more COA traces compared to evaluating with our expanded StarCraft II scenario directly, which takes hours for each episode. Figure 7 shows the measured runtime of surrogate model episodes over the course of execution—decreasing from around 200 seconds (s) to nearly

50 s per episode, using only four compute cores. This is ~100x faster than we were able to run our StarCraft II scenario (~10,000 s) and ~10,000x faster than our estimate for real-time equivalent engagement. This decrease occurs because the NPP trains to favor shorter execution paths, thus ignoring actions that do on contribute to better performance. Certain actions can cause execution paths to massively expand, should they require far more follow-on actions or observations than others. Each action or observation must be stepped in turn, so one would expect execution time to scale approximately linearly with the number of "NPP states" encountered. Figure 8 and Figure 9 show this directly.

NPPs massively reduced the action space. In conditions with positive expected future rewards, one expects reward discounts to encourage the NPP to select execution paths with fewer actions if they result in similar rewards to paths with more actions; effectively, the NPP attempts to get the positive reward "sooner." The decrease in the number of "NPP states" over time supports this hypothesis. Similarly, one would hypothesize the opposite effect if the expected reward were negative; the NPP attempts to "defer" receiving a negative penalty. The open-structure nature of the PP-induced POMDP causes these results to vary over time with the execution tendencies of the policy program. With discount and positive total reward there is a natural inclination for the NPP to pick execution paths with fewer actions/observations to receive the reward more quickly (Figure 8). The colored shapes in Figure 8 indicate possible PP subroutines and execution paths incorporating learned behavior (e.g., candidate actions or observations available to the agent), and the dotted-shape outlines denote PP subroutines that the agent learned not to use (e.g., effectively ignore) over the course of training.

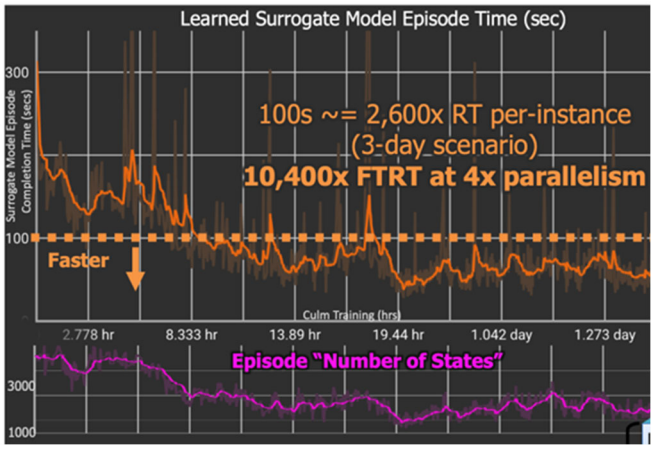

Figure 7: A depiction of surrogate model runtime over the course of training, running 10,000x faster than real time (FTRT). Variation in PP execution paths can alter the number of observations and actions over time, so surrogate model episode runtime is non-stationary.

Figure 9 shows a measurement of an estimate of an "empirical action tree width," in effect measuring the typical number and size of action spaces encountered in execution. The metric $\log(W)$ is defined as follows (where $|A_i|$ denotes the number of choices in the ith encountered action space):

$$\log(W) = \sum_i \log(|A_i|)$$

This is equivalent to the log of the full tree width for homogenous POMDPs (i.e., those with fixed durations and constant action space sizes at a given step). In our setting, it instead represents an approximation assuming said homogeneity. We estimate a primitive action space for our StarCraft II scenario would measure at greater than 1e5 (10,000 steps x ln(1,000 units x 100 choices per unit)), thus implying our system achieves 50x reduction in the log of the full action tree width (an astronomical reduction in action space size). Equivalently, one could view this measurement as stating that the duration of the raw scenario is 50x longer than the PP-transformed scenario (at equivalent action space sizes).

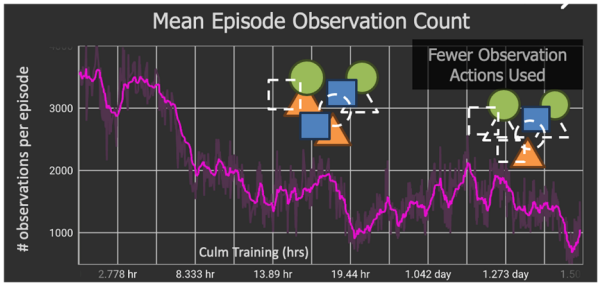

Figure 8: Action space reduction: A visualization of the reduction in (cumulative) number of observations over the course of training time

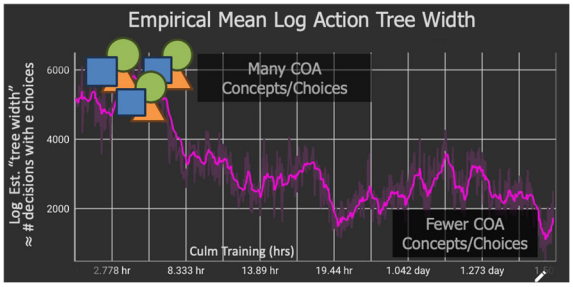

Figure 9: Action space reduction: A visualization of the reduction in (cumulative) action space sizes (e.g., fewer decisions each with variable number of choices) over the course of training time

For COA trace explainability, a custom-built web-based human-machine interface (HMI) supports exploration of the PP structure and execution "trace" generated during an episode (Figure 10). These traces include all observations (including the simulation state, such as troop movements, battles, and resource levels) and record all Red and Blue action information during execution, as well as the various auxiliary information indicated by the developer via a Python "@trace" decorator in the PP file. This decorator also records the arguments and return values of any decorated methods, enabling further analysis of NPP behavior. This information is presented along the bottom of the screen via a hierarchical timeline view. The top left of Figure 10 shows part of the StarCraft II PP employed, while the top right shows a geographic map view with a play head to allow the user to animate the execution of a COA trace with the agent's actions and simulation environment outcomes (e.g. unit losses, where, when, and why).

Figure 10: Visualization of NPP execution in our web-based HMI to inspect and animate a COA trace on a map showing how battles unfold (upper right); (bottom) the timeline depicts the execution of select traced subroutines in the NPP (as specified via the "`@trace`" decorator) through a completed episode; (top left) clicking each event in the timeline exposes values of the arguments and returned results of the traced method

RELAVANCE WITH RESPECT TO RELATED DRL WORK

Our NPP neuro-symbolic DRL method and results significantly contribute to the state of the art in DRL research. Below, we highlight related DRL approaches and describe how NPPs extend the work or overcome key challenges.

Hierarchical Reinforcement Learning: Feudal reinforcement learning, introduced by Dayan and Hinton (1992), was one of the first hierarchical reinforcement learning (HRL) frameworks that sought to achieve learning at multiple time- and space scales. This work introduced a learning hierarchy using a variant of Q-learning, where learning managers assign tasks to submanagers that then seek to maximize associated rewards. This spawned a wide body of research in the field of HRL, which is nicely captured in Pateria et al. (2021). The work of Nachum et al. (2018) addresses the challenge of data efficiency in HRL problems, which typically stem from the drift in policies between high- and low-level learners. Using off-policy training and setting observation states as goals for high-level learners, which instruct low-level learners, they achieve great results with efficient learning in a variety of navigation tasks. NPPs were motivated in part to enable the construction of agents that employ such hierarchical reasoning effectively. This can be accomplished both via direct construction of appropriate task hierarchies in PPs, as well as various forms of learning by constructing PPs where hierarchies are built and traversed during program execution.

Model-Based Deep Reinforcement Learning: MuZero (Schrittwieser et al., 2020) is a sample efficient reinforcement learning approach that achieved great success in board games. DNN models are trained to learn the environment and its dynamics, the optimal policy, and policy valuation. MuZero distinguishes itself by using Monte Carlo tree search simulations of the environment to sample actions with a weighted probability given by the number of node visits in the tree. DreamerV3 (Hafner et al., 2024) uses recurrent neural networks to learn (1) the environment in which the model operates (i.e., the "world" model), (2) the actions the agent should take given the state of the

environment (i.e., the "actor" model), and (3) the value that should be attributed to each action in a wide variety of games. The three neural network architectures are autoregressive, and their world model is a recurrent state-space model (RSSM). Similarly to NPPs, their RSSM is used to predict potential future observations, which in turn allows the agent to adjust its action policy to consider richer sets of outcomes. The NPP implementation employs a variant of these approaches where predictions are made of observation intrinsics rather than raw environment features. Our training against these learned surrogate models operates via Monte Carlo evaluation of outcomes sampled from predictive distributions.

Neuro-symbolic Programming: Neuro-symbolic programming (Chaudhuri et al., 2021) represents the body of work that seeks to merge deep learning concepts with program synthesis. Program synthesis focuses on the creation of programs that satisfy high-level specifications from a given set of existing modules. Such programs are written using the syntax and structure specified by some DSL. In neuro-symbolic programming, neural components are embedded into the program synthesis framework. These components are differentiable and are optimized to best fit some dataset. The value of this framework lies in its transparency and explainability. Neuro-symbolic programming ideas have progressed in reinforcement learning (Verma et al., 2019; Cheng et al., 2019; Shi et al., 2019) and behavior analysis (Sun et al., 2021). Rather than focusing on automatic program synthesis, which can have scaling and explainability limitations, NPPs prioritized human authoring of PPs, much like one would author a DNN architecture. This enables improved AI-explainability as well as highly expressive ways for domain knowledge or context to be employed to support higher-performing systems.

CONCLUSION

Neural program policies (NPPs) provide an expressive, powerful architecture for constructing AI systems with many exceptional properties. In this paper we showed significant NPP benefits in terms of learning a vastly reduced action space to lower training time, improving task performance, and explaining DNN results to human planners via a web-based HMI. We showed how NPPs enable the incorporation of domain knowledge into a novel DRL architecture encapsulating open-structure POMDPs. Our experiments show how this approach dramatically reduces the size of problem action spaces and can be employed to construct ultrafast surrogate models (surpassing 10,000x real time for our complex, multi-domain StarCraft II scenario). We included results on basic control theory problems as well as results on a more complex large-scale COA trace generation for real-time strategy games. Finally, we showed how the PP-designed abstractions can be employed to support human reasoning and understanding of AI results through the NPP @trace mechanism that directly links NPP outputs to an interactive web-based HMI. The results demonstrate how an AI system can help human planning staffs construct more high-quality plans, analyze plan strengths and weaknesses more deeply, and explore a larger number of plan alternatives in a fixed planning time.

More broadly, NPPs present a novel and still underexplored direction for AI system construction and definition. In this work we present NPPs that can be used by AI developers— experts with skill sets employed in DNN architecture definition, interface design, or any of the other complex design decisions associated with the construction of an AI system. An immense range of opportunities is opened up by enabling domain/subject matter experts and even end users to engage in this design process. The PP DSL was explicitly embedded in a programming language to maximize expressivity, enabling the description of complex structure not possible with flowcharts, graphs, or other structure definitions. This does not imply, however, that all NPP design and engineering must be performed by a programmer. Various higher-level structures (e.g. the aforementioned flowcharts, timelines) could be used by nonprogrammers as design interfaces for the definition or manipulation of a PP structure. We hope to explore this in future work.

It is also worth considering NPPs in the context of distinctly different AI approaches, especially those focused on fine-tuning of foundation models, e.g., large language models (LLMs). NPPs were designed to enhance the engineering of safe, effective AI systems by design. In contrast, foundation model approaches prioritize discovering highly generalized features from datasets with tremendous breadth but at the potential cost of AI we understand less and have limited control over. In many ways these represent complementary approaches—an NPP could use an image foundation model as part of the DNN architecture, or it could explicitly make queries against a remote LLM in PP execution, for example. Rather than relying solely on the generality induced by massive datasets, however, NPPs offer an additional dimension of system design that can be engineered for the desired task. We believe this will continue to be an important advantage to AI system construction for the foreseeable future, especially for any task with bounded compute resources.

REFERENCES

Chaudhuri, S., Ellis, K., Polozov, O., Singh, R., Solar-Lezama, A., & Yue, Y. (2021). N programming. *Foundations and Trends® in Programming Languages*, *7*(3), 158–243. http://doi.org/10.1561/2500000049

Cheng, R., Verma, A., Orosz, G., Chaudhuri, S., Yue, Y., & Burdick, J. W. (2019). Control regularization for reduced variance reinforcement learning. arXiv. http://doi.org/10.48550/arXiv.1905.05380

Dayan, P., & Hinton, G. E. (1992). Feudal Reinforcement Learning. In *Advances in Neural Information Processing Systems* (Vol. 5). Morgan-Kaufmann. https://papers.nips.cc/paper/1992/hash/d14220ee66aeec73c49038385428ec4c-Abstract.html

Hafner, D., Pasukonis, J., Ba, J., & Lillicrap, T. (2024). Mastering diverse domains through world models. arXiv. http://doi.org/10.48550/arXiv.2301.04104

Nachum, O., Gu, S., Lee, H., & Levine, S. (2018). Data-efficient hierarchical reinforcement learning. arXiv. http://doi.org/10.48550/arXiv.1805.08296

Pateria, S., Subagdja, B., Tan, A., & Quek, C. (2021). Hierarchical reinforcement learning: A comprehensive survey. *ACM Comput. Surv.*, *54*(5), 109:1-109:35. http://doi.org/10.1145/3453160

Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., … Silver, D. (2020). Mastering Atari, Go, chess and shogi by planning with a learned model. *Nature*, *588*(7839), 604–609. http://doi.org/10.1038/s41586-020-03051-4

Shi, G., Shi, X., O'Connell, M., Yu, R., Azizzadenesheli, K., Anandkumar, A., … Chung, S.-J. (2019). Neural Lander: Stable drone landing control using learned dynamics. In *2019 International Conference on Robotics and Automation (ICRA)* (pp. 9784–9790). http://doi.org/10.1109/ICRA.2019.8794351

Sun, J. J., Kennedy, A., Zhan, E., Anderson, D. J., Yue, Y., & Perona, P. (2021). Task programming: Learning data efficient behavior representations. arXiv. http://doi.org/10.48550/arXiv.2011.13917

Verma, A., Murali, V., Singh, R., Kohli, P., & Chaudhuri, S. (2019). Programmatically interpretable reinforcement learning. arXiv. http://doi.org/10.48550/arXiv.1804.02477

Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., … Silver, D. (2019). Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, *575*(7782), 350–354. http://doi.org/10.1038/s41586-019-1724-z

Ye, W., Liu, S., Kurutach, T., Abbeel, P., & Gao, Y. (2021). Mastering Atari Games with Limited Data. arXiv. Retrieved from http://arxiv.org/abs/2111.00210