Explaining Deep Adaptive Programs via Reward Decomposition

Martin Erwig[1], **Alan Fern**[1], **Magesh Murali**[1], **Anurag Koul**[1]
[1] School of EECS
Oregon State University

Abstract

Adaptation Based Programming (ABP) allows programmers to employ “choice points" at program locations where they are uncertain about how to best code the program logic. Reinforcement learning (RL) is then used to automatically learn to make choice-point decisions to optimize the reward achieved by the program. In this paper, we consider a new approach to explaining the learned decisions of adaptive programs. The key idea is to include simple program annotations that define multiple semantically meaningful reward types, which compose to define the overall reward signal used for learning. Using these reward types we define the notion of reward difference explanations (RDXs), which aim to explain why at a choice point an alternative *A* was selected over another alternative *B*. An RDX gives the difference in the predicted future reward of each type when selecting *A* versus *B* and then continuing to run the adaptive program. Significant differences can provide insight into why *A* was or was not preferred to *B*. We describe a SARSA-style learning algorithm for learning to optimize the choices at each choice point, while also learning side information for producing RDXs. We demonstrate this explanation approach through a case study in a synthetic domain, which shows the general promise of the approach and highlights future research questions.

1 Introduction

Programmers are often faced with uncertain choices about program logic that can significantly impact the quality of the resulting program. Adaptation Based Programming (ABP) [Bauer *et al.*, 2011] is a programming paradigm in which programmers can explicitly declare their uncertainty about such choices via the insertion of adaptive variables into the program. We refer to programs with such adaptive variables as adaptive programs. In addition, adaptive programs include reward statements that indicate quality of program executions. The key idea of ABP is to use reinforcement learning (RL) to optimize the decisions of adaptive variables such that the reward acquired during program executions is optimized. In particular, in this work we focus on deep adaptive programs (DAP), where choice point decisions are automatically learned via deep neural networks.

To gain confidence in learned decisions of DAPs, it is important to produce explanations for those decisions. In this paper, we focus on explaining the reasons that one alternative *A* of an adaptive variable was preferred to another alternative *B*. The key questions are: (1) What form the explanations should take? (2) How to adjust the learning algorithms so that such explanations can be produced?

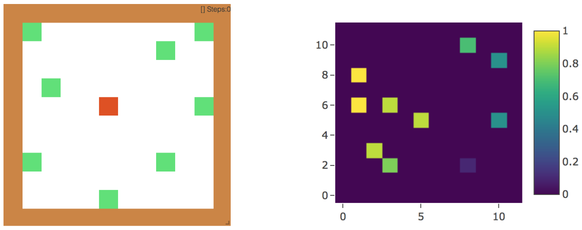
To aid in answering the first question, we allow the programmer to provide a simple form of program annotation. The annotations specify reward types, which label each reward asserted by the program by one of a fixed set of types. For example, in a real-time strategy game, one type of reward would be related to damage inflicted on the enemy and another type would be based on damage the enemy inflicts on us. The goal of learning is still to maximize the total reward, accumulated across all types. However, the types can provide significant insight into decisions. Indeed without type information, an explanation for why *A* was selected over *B* might be the trivial explanation that it is predicted that selecting *A* will lead to more future reward than *B*. Rather, with type information, we open the possibility to comparing *A* and *B* in terms of the predicted future reward of each type, which provides a finer-grained view of the trade-offs going into the decisions and significantly more insight.

To instantiate this idea we define the notion of reward difference explanations (RDXs), which explain why *A* is preferred to *B*. An RDX is simply a vector that records the difference in predicted future reward for each reward type when taking *A* versus *B* and then following the program thereafter. We further define the notion of a minimal sufficient explanation for an RDX as the minimal number of reward components needed to prove that *A* is preferred to *B*.

To address the second question, we present a SARSA-style algorithm for learning policies to control the decisions of adaptive variables and at the same time learning information needed to produce RDXs for any pair of decisions. This idea has been fully implemented within a Python-based ABP library for deep adaptive programs. We demonstrate this algorithm and explanation approach via a case study on a synthetic problem designed to include decisions that are not obvious to a human and hence require explanation. The results show that the approach is able to provide useful explanations and that the explanations tend to be quite compact.

2 Adaptation-Based Programming

To illustrate explanations in DAPs let us consider the following example problem shown in Figure 1. The agent in this example moves across a grid with the goal of collecting as many fruits as possible within 100 steps. The locations of the fruits are fixed. The environment also contains lightening events that occur randomly with every step within the grid. If the agent is struck by lightening, they die, and the game ends.



(a)

(b)

Figure 1: (a) Grid world for the fruit collection problem; red indicates the position of the agent, and green indicates the position of the fruits. (b) A heat map that shows the probabilities of locations of lightening events.

The fruit collection problem can be formulated as an ABP program as shown in Listing 1, in which the problem is represented as an OpenAI gym environment `env`. In addition to state initialization, this object provides the method `step` for making state transitions. The object `move` represents an adaptive variable that offers four choices for moving in different directions, and it contains methods for making RL-informed choices and collecting rewards.

```
state = env.reset()
move = Adaptive(choices = [UP, DOWN, LEFT, RIGHT])

while not done:
  direction = move.choose(state)
  state, reward, done = env.step(direction)
  move.adapt(reward)
```

Listing 1: Adaptive program for the fruit collection problem

The `step` function performs an action provided as input and returns the resulting state, the reward received, and a boolean indicating whether the current episode has ended. In this example this boolean is `True` when the agent has collected all the fruits, the maximum number steps of 100 has been reached, or the agent is struck by lightening. The `state` object is a two dimensional boolean matrix representing the position of the agent and the location of the fruits in the environment.

In each of the 100 steps available to the agent, the method `choose` first selects a direction to move, then the method `step` moves the agent in that direction. After that, the reward resulting from this step is accumulated with the method `adapt`. The adaptive variable uses RL to choose the correct value for the adaptive based on the current state [Bauer *et al.*, 2011]. We use a modified version of the SARSA RL algorithm in our approach. Using standard RL terminology, the adaptive variables are treated as decision points and the values of adaptive variables are treated as actions that can be chosen at those points. Each time a decision point is encountered the “state" of the decision point is used as a basis for the action/value selection. The library allows for the user to define arbitrary features of decision points that can be used as the state representation.

We note that while this example program contains only a single adaptive variable, for illustration purposes, our library supports programs that include an arbitrary number of adaptive variables. Using multiple adaptive variables allows for modeling a wide range of agent decision-making architectures such as those used in hierarchical RL [Dietterich, 2000; Parr and Russell, 1998]. The ABP framework also generalizes prior work on “partial programming" [Andre and Russell, 2002], where the semantics were defined relative to an external MDP rather than just the adaptive program.

After the initial training period, the adaptive variable is associated with a Q-function. The Q-function $Q(s, a)$ maps the state s of a decision point and choice a to a numeric value. In particular, $Q(s, a)$ is an estimate of the expected future reward that will be achieved if choice a is selected and then the adaptive program is followed thereafter. Given a decision point, an associated Q-function, the adaptive program will select the choice that maximizes the learned Q-function. In our current library, the Q-function for each adaptive variable is represented via a neural network that takes the state as input and outputs the Q-value for each possible choice. In our fruit collection example, the neural network input is the raw map, where a one-hot encoding is used to represent the contents of each cell.

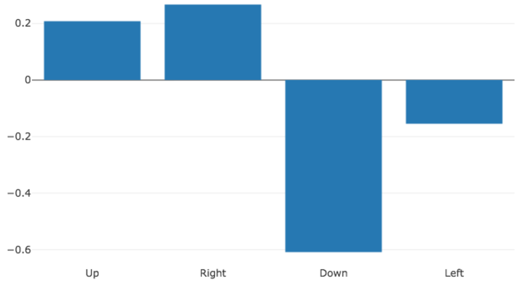


Figure 2: Q-values for the actions in the state shown in Figure 1(a).

Figure 2 illustrates a situation in which the agent has to move right in order to maximize the reward. However having a single Q-value does not give us a sufficient explanation. It compresses the knowledge about the problem into a single value. It is not clear why the agent made the decision: were they struck by lightning when moving in the other directions? Which fruit is are they trying to collect?

3 Explaining via Reward Decomposition

Our main strategy for obtaining better explanations is to notice that in many RL environments the reward can be broken down into multiple reward types, each corresponding to semantically distinct ways of acquiring reward. This allows for decomposing the overall reward function into a number of different reward component functions, which sum up to the total reward. Often natural reward decompositions are obvious to a programmer or environment designer, since defining the reward function requires thinking about the distinct circumstances where reward is generated. The traditional RL paradigm, however, does allow a developer to explicitly expose that knowledge.

In the fruit collection example we can decompose the reward into 9 different reward types, one for each location of the fruit plus one for when the traveller gets hit by lightning. In order to support this, we have extended the ABP library to use reward annotations. The program for the fruit collection problem can be extended by annotations that use locations to distinguish the different fruits and a tag for the lightening reward.

Given a reward decomposition, we can adjust the learning algorithm so that it learns a distinct Q-function for each reward component, each predicting the future reward of the corresponding component only, see Listing 2. It is straightforward to see that the sum of the individual component Q-functions is equal to the Q-function for the overall cumulative reward. In our fruit collection problem, there will be a Q-function for each fruit location and for lightning. As an example, the component Q-function for the lightning reward type gives the expected future reward (negative) due to lightning strikes after taking an a particular action in a particular state and then following the program thereafter.

```
rewards = []
lightning_pos = generate_lightning()

if agent_location in fruit_locations:
   rewards.append((agent_location, 2))

if agent_location in lightning_pos:
  rewards.append(("Lightning Strike", -1))
```

Listing 2: Decomposed reward generation in each step

The part of the adaptive program that represents the core of the example needs only a minor modification. The only change is the replacement of the single call to `adapt`, which uses one reward value, by a loop that iterates over a list of rewards and corresponding reward types, see Listing 3. In this scenario, this list always contains either zero or one element.

```
state = env.reset()
move = Adaptive(choices = [UP, DOWN, LEFT, RIGHT])

while not done:
  direction = move.choose(state)
  state, rewards, done = env.step(direction)
  for typedReward in rewards:
     move.adapt(typedReward)
```

Listing 3: Adaptive Program using reward decomposition

Given a set of component Q-functions for each reward type, decisions are made by selecting the action that maximizes the cumulative Q-function, which is equal to the sum of the components. Thus, learning should both ensure that the cumulative Q-function converges to the optimal Q-function for the overall reward, while also ensuring that the component Q-functions converge to their correct values. A similar learning problem has been studied in prior work [Russell and Zimdars, 2003] on multi-agent learning. In that work, the total reward was decomposed across multiple learning agents that were controlled by a centralized policy. If we equate the agents with reward types, the two formulations become identical. Thus, we use the SARSA-style algorithm develop in that work in the context of ABP. The algorithm uses on-policy SARSA learning [Sutton and Barto, 1998] to update each of the component Q-functions and drives exploration using an ϵ-greedy policy based on the cumulative Q-function.

We now consider an example decision of the agent for the state shown in Figure 1(a). This decision can be explained much better using the decomposed Q-values and reward types. If we take a look at the decomposed Q-values and reward types in Figure 3, we can observe that moving right is better than moving down because moving right has more positive rewards for almost all fruits and at the same time less of a chance of being struck by lightening than moving down. However, the decomposition of the rewards itself does only part of the job. For example, it is not so clear why moving right is better than moving up.

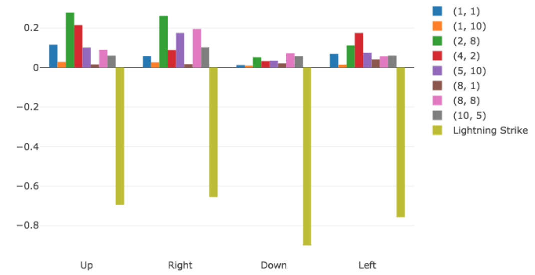


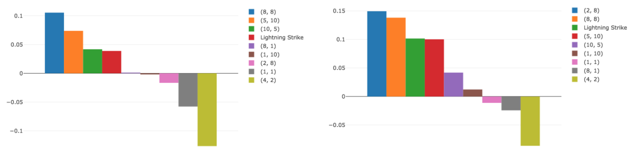
Figure 3: The Q-values from Figure 2 decomposed by different reward types for fruit locations and lightening.

To better compare the decomposed Q-values for different actions, we can compute their difference and then order the result by magnitude. In the context of decomposed Q-values, $Q_\pi(s, A)$ is a vector that returns returns the component Q-values for each reward type for the current policy π (generally the ϵ-greedy policy) in state s and action A. We write $Q^R_\pi(s, A)$ for the Q-value component of reward type R. We call the reward-type-indexed difference between the decomposed reward for two actions $A$and B their *Reward Difference Explanation* (RDX).

$$\Delta_\pi(s, A, B) = Q_\pi(s, A) - Q_\pi(s, B)$$

A positive value in an RDX mapping for a reward type R indicates that the adaptive variable predicts a higher reward of type R for A than for B. Correspondingly, a negative value indicates that action B promises higher reward of type R.

To compare moving `RIGHT` with moving `UP` for the state shown in Figure 1(a) we compute the RDX $\Delta_\pi(s, \texttt{RIGHT}, \texttt{UP})$ and visualize the resulting mapping in Figure 4(a), ordering the RDX components by magnitude. To convince ourselves that `RIGHT` is indeed the better action, we have to ensure that the sum of the positive reward differences, representing the advantages of `RIGHT` over `UP` is greater than the sum of the negative ones, which represent the disadvantages.



(a) $\Delta_\pi(s, \texttt{RIGHT}, \texttt{UP})$ (b) $\Delta_\pi(s, \texttt{RIGHT}, \texttt{LEFT})$

Figure 4: Reward Difference Explanations for two action pairs.

The four reward types with negative values sum up to -0.20, while the sum of the first three positive reward yields 0.22, which is enough to be sure that RIGHT is expected to be the preferred action. In other words, moving RIGHT promises enough reward for collecting the three fruits at the shown locations that makes up for any other expected disadvantages. This example also shows that we don't have to consider all positive reward types to make this determination. This phenomenon is even more salient when we compare the actions RIGHT and LEFT. In this case, the reward for one fruit with a value of 0.14 is sufficient to cover all expected losses, which are about -0.12.

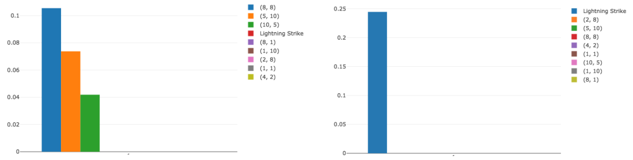
This observation leads to the definition of a *minimal sufficient explanation* (MSX), which is the minimal subset of $\Delta_\pi(s, A, B)$ whose sum of rewards exceeds sum of all negative rewards from $\Delta_\pi(s, A, B)$. We denote the sum of all individual rewards of a reward decomposition mapping Q with $\Sigma(Q) = \sum_{(r,x)\in Q} x$. Now with:

$$R^-(Q) = \Sigma(\{(r, x) \in Q \wedge x < 0\})$$
$$Sub^+(Q) = \{S \subseteq Q \mid (r, x) \in S \Rightarrow x > 0\}$$

we can define:

$$\mu_\pi(s, A, B) = S \in Sub^+(Q) : \Sigma(S) > R^-(Q) \wedge |S| \text{ is minimal}$$
$$\text{where } Q = \Delta_\pi(s, A, B)$$

The minimal sufficient explanations for the reward differences from Figure 4 are shown in Figure 5. They allow the user of an explanation to focus on a subset of reward types.



(a) $\mu_\pi(s, \texttt{RIGHT}, \texttt{UP})$ (b) $\mu_\pi(s, \texttt{RIGHT}, \texttt{LEFT})$

Figure 5: Minimal Sufficient Explanations for two action pairs.

The concept of minimal sufficient explanations (called MSE) was also introduced in closely related work on explanations for optimal Markov Decision Process (MDP) policies [Khan *et al.*, 2009]. There are several key distinctions between that work and ours. First, that work assumed a perfect model of the environment and approached the problem from an automated planning perspective. Rather, we are in an RL setting, which introduces challenges due to the lack of a model. Second, that work focused on explaining the optimal decision of an optimal policy. In particular, the notion of MSE used there required that the MSE proved that the optimal action was better than all other actions. Such explanations will often be significantly larger than our pairwise explanations, since an optimal action may be better than other actions for orthogonal reasons. In such cases, their MSE must include all of those reasons. Finally, the prior work did not introduce the notion of reward decomposition. Rather they relied on a generic, but extremely fine grained decomposition that effectively defined a reward component for each state. Such an approach is quite limited to domains with small state spaces.

One question that remains is: How effective are minimal sufficient explanations? In the fruit collection scenario we have 9 different reward types, and so it might be possible for any $\mu_\pi(s, A, B)$ to vary in size from 0 to 8. (Here a size of 0 means that the RDX has no negative component and that action A is better in every aspect.) To answer this question we computed the frequencies of the sizes $|\mu_\pi(s, A, B)|$ for all action pairs for an optimal policy π. As Figure 6 shows, in the vast majority of cases a single reward type is sufficient to explain an action, which shows that the technique is quite effective in focusing the attention of users on particular parts of reward components.

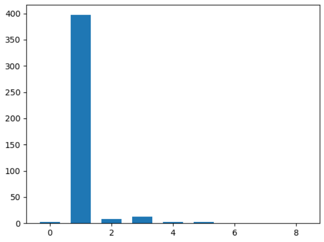


Figure 6: Frequencies of $|\mu_\pi(s, A, B)|$

4 Conclusions

We have demonstrated an approach to explain machine-learned decisions in the context of adaptation-based programming by decomposing rewards into labeled parts. Decomposition and reward typing together provide the opportunity to justify decisions by comparing advantages and disadvantages of decisions with respect to different reward types. This idea is formalized though reward difference explanations. In addition, through the concept of minimal sufficient explanations we can often generate explanations that need to mention only few of the potentially large set of different reward types. As our analysis has shown, in the example scenario the vast majority of explanations require mention of only one reward type.

Acknowledgements

This work is partially supported by the National Science Foundation under the grant CCF-1717300 and by DARPA under the grant N66001-17-2-4030.

References

[Andre and Russell, 2002] David Andre and Stuart J Russell. State abstraction for programmable reinforcement learning agents. In *AAAI/IAAI*, pages 119–125, 2002.

[Bauer *et al.*, 2011] Tim Bauer, Martin Erwig, Alan Fern, and Jervis Pinto. Adaptation-based programming in java. In *Proceedings of the 20th ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pages 81–90. ACM, 2011.

[Dietterich, 2000] Thomas G Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000.

[Khan *et al.*, 2009] Omar Zia Khan, Pascal Poupart, and James P Black. Minimal sufficient explanations for factored markov decision processes. In *ICAPS*, 2009.

[Parr and Russell, 1998] Ronald Parr and Stuart J Russell. Reinforcement learning with hierarchies of machines. In *Advances in neural information processing systems*, pages 1043–1049, 1998.

[Russell and Zimdars, 2003] Stuart J Russell and Andrew Zimdars. Q-decomposition for reinforcement learning agents. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, pages 656–663, 2003.

[Sutton and Barto, 1998] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.