Master Project Write-Up: Code Generation Using Interpretable Machine Learning Policies

David Bosomworth

Abstract

With machine learning being used for a wider range of problems, there are concerns about the lack of transparency and trustworthiness of policies. It is difficult to determine what factors influence the output of these policies and how much, without doing multiple tests. In order to address this, by fitting binary decision trees over the inputs and outputs of reinforcement learning policies and then interpreting those trees to generate python code, policies can be created that perform almost as well as the original policy that are also readable and debuggable.

Introduction

Machine learning algorithms have become more prevalent over time. These algorithms can be very effective for certain problems. However, an issue with them is that the policies generated by these algorithms are black boxes. It cannot be determined what action a policy will predict given a specific input. It also cannot be determined what values of an observation were important to determine the action. If one were to somehow look inside one of these policies, complicated mathematical functions would be found, neither readable nor reasonable to calculate the output of it. Some may not consider this a problem, if a policy always does well, who cares if the factors that determine a policy's output are opaque! But the policies don't always do well, there can be no explanation why, and the only way to address it is to train the policy more and hope it's better. This opaqueness also makes it more difficult to hold someone accountable if something erroneous or malicious happens, and can rightfully make people feel dubious of machine learning's applications.

In order to address this, there has been an increased interest in so-called *explainable* artificial intelligence (XAI). An explainable artificial intelligence system is one that can provide an explanation for its decisions in terms legible to the end-users and stakeholders (Adadi and Berrada 2018). By having decisions be explainable, users can know what factors do and don't determine the output of a policy, make their own judgement if the outputs are logical or fair to them, and determine if the policy is actually effective for whatever application it's to be used for.

A subset of the field of XAI is interpretable machine learning. A machine learning policy is interperatable if it is inherently explainable. Decision trees, which we primarily use in this work, are an example of an interpretable machine learning method.

Sequential decision making problems can be especially difficult to explain. This is partly because they are often solved using complex machine learning methods, such as deep neural networks. They are also difficult to explain because the motivation for a decision may be because of actions taken far in the past or actions that the agent has yet to take. To address this problem, I introduce an approach to make policies interpretable by fitting binary decision trees over data made up of an environment's observation states and a policy's corresponding predicted actions across a certain number of episodes— in essence, recreating the original policy. The decision trees are then parsed and translated into Python code.

To give a brief rundown for the rest of this write-up, the methods section is where I explain my process for interpreting a policy into generated code. In the experiments section, I detail how the environments that agents are being tested in work and the experiments done to test the effectiveness of binary decision trees and the generated code in comparison to the original reinforcement policy. I then state and discuss the results in the results and discussion section. Afterwards, the future work section is about the different directions someone could take this work in the future. The last section is the conclusion.

Methods

This process for interpreting policies and generating code from them is made up of several steps. The first step is getting a policy for a given environment. The next step is fitting a binary decision tree (BDT) off of data from running the original policy. After that, the BDT is split into a list of every possible branch; sorted into sets of branches that are grouped by the action most commonly represented in a branch's leaf node; and then recombined into trees that represent the conditions necessary for each of an agent's actions. Finally, the trees are used to generate Python code for deciding if an agent in a given environment should do a specific action.

First, we need to find or create policies for the environments: CartPole, MountainCar, and LunarLander. It doesn't actually matter how we get a policy, we just want one that can perform well in the specific environments so that they can act as a ground truth. For Cartpole and LunarLander, the Proximal Policy Optimization (PPO) algorithm from the stable-baseline3 python package was used to create policies for those environments. The only hyper-parameters that were set for those policies were that training would go for 250,000 timesteps in their environments, everything else was left as default. For the MountainCar environment, I used a pre-trained policy on GitHub that was created using a Deep Q-Network (Shikora 2018). The hyper-parameters for that policy were that it was trained for 2000 episodes and had an epsilon starting at 1, an epsilon decay rate of 0.9, and that epsilon stopped decaying at 0.001. The policies were then saved for future use.

Once we have a policy for an environment, a binary decision tree was created off of that policy. This was done by having an agent use the policy for its environment for a certain number of episodes. Data of the agents was recorded over 10, 40, 100, and 200 episodes. This data was lists all of the observation states that the agent encountered and then the actions taken in response. A BDT was then created and fitted according to these lists using the binary decision tree classifier from the Scikit Learn python package, with the states as the input and the actions as the output. The lists of observation states, actions, and trees were save for future use or reference.

Before going further in this process, some class structures need to be explained. The BDTs are disassembled and reassembled as a set of binary trees that represent nested if statements. The root and internal nodes of these new trees will have a Condition class as their value, while the leaf nodes will have an Action class as their value. The Condition class represents the expression of an if statement. It has the name and variable type for the feature of the observation state that it will be checking; the threshold value that the feature will compared to; and a boolean that indicates if the condition being true would take an agent down the left or right path of the BDT, which in conjunction with the feature variable type, determines if the binary operator of the condition statement will be in the positive or negative. The left sub-tree of a node with a condition class represents the code block that will be executed if the condition is true, the right sub-tree represents the code block that will be executed if the condition is false. The Action class represents a return statement, returning a value that would tell an agent to do a specific action if all the previous conditions leading to it are met.

After a BDT is made, it is split into a list of every possible branch. A branch meaning a binary tree where the root and internal nodes only have one child node, and ends with a single leaf node. The root and internal nodes represent the sequence of conditions compared against an environment's observation state, and a single leaf node representing the action that was most commonly chosen if all the previous conditions are met. This splitting of the BDT is done by a pre-order traversal of the BDT. Starting at the root of the BDT, two binary trees are created from a copy of the branch given as an argument. Using the index number given as an argument, we create a Condition class using the values from the node at the index in the BDT. That created node is then added to the end of the first binary tree copy, representing the left branch. The splitBDT function is then recursively called using the index of the left child of the current node and the left branch as parameters. The recursion repeats until the base case of reaching a leaf node in the BDT is achieved, an Action class is then created and set as the value of a node, that node is added to the end of the branch given in the argument, and then that branch is add to the list of branches. The function then returns up one level and does the same process for the right side of the current node in the BDT. The function is finished once every node has been visited and thus every branch has been conceived. Refer to algorithm 1 for psuedocode on this.

Algorithm 1 splitBDT

Input: index starting at 0, branchSoFar and branchList starting as empty lists
Output: branchList with all possible branches
if BDT at node $index$ is a leaf **then**
  create node with Action as the value
  add node to end of treeBranch
  add treeBranch to branchList
else
  define leftBranch, rightBranch as copy of branchSoFar
  create node with Condition that would go left in BDT
  add node to leftBranch
  $splitBDT(BDT.leftChild(index), leftBranch)$

  create node with condition that would go right in BDT
  add node to rightBranch
  $splitBDT(BDT.rightChild(index), rightBranch)$
end if
return branchList

The next step is combining branches together into binary trees that correspond to each possible action an agent could take in an environment. Using the list of all possible branches, branches are grouped together by what action their leaf node recommends. Each group of branches is then merged together into a binary tree representing all of the possible conditions that would need to be satisfied in order for an agent to take a certain action. This merging is done through a recursive function that receives two trees as arguments. If either tree is empty, it returns the other as the combined tree. Otherwise, the function travels down the first tree, adding any branches of the second tree that the first doesn't have, finishing when there's no more branches to add. Refer to algorithm 2 for psuedocode on this.

Finally, we generate the Python code. A Python file is opened or created, and a predict function is written. It takes an observation state as an argument and then splits that up into named variables that indicate what they represent in the environment. Inside the predict function, a series of if state-

Algorithm 2 mergeBranches

Input: ruleTree1, ruleTree2
Output: merged binary tree

if ruleTree1 is empty **then**
  return ruleTree2
end if
if ruleTree2 is empty **then**
  return ruleTree1
end if
$s \leftarrow []$
$temp \leftarrow [Tree1 = ruleTree1,$
  $Tree2 = ruleTree2]$
$s.append(temp)$
$n \leftarrow None$
while length of $S \neq 0$ **do**
  $n \leftarrow s[-1]$
  $s.pop()$
  if n.Tree1 == None or n.Tree2 == None **then**
    continue
  end if
  if n.Tree1.leftChild == None **then**
    n.Tree1.leftChild ← n.Tree2.leftChild
  else
    $t \leftarrow [Tree1 =$n.Tree1.leftChild,
      $Tree2 =$n.Tree2.leftChild]
    s.append(t)
  end if
  if n.Tree1.rightChild == None **then**
    n.Tree1.rightChild ← n.Tree2.rightChild
  else
    t ← $[Tree1 =$n.Tree1.rightChild,
      $Tree2 =$n.Tree2.rightChild]
    s.append(t)
  end if
end while
return ruleTree1

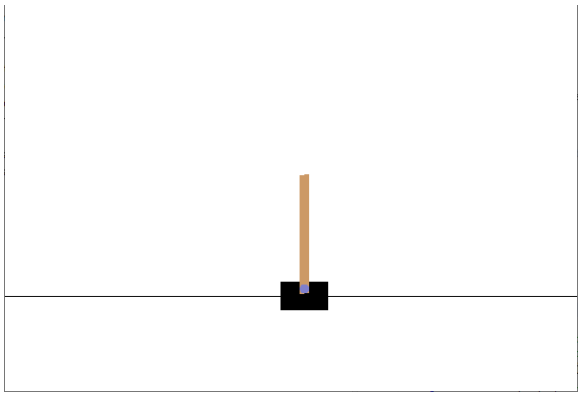

Figure 1: CartPole

ments are printed, with each condition statement calling a boolean function that will determine if an action should be done or not. A return statement at the end of the predict function returns a number that corresponds to the action should be taken. The boolean functions are then printed. For each function that determines if an action should be done, the program uses the corresponding tree and visits each node in a tree in pre-order traversal and calls a function that prints parts of an if statment with Python syntax to the Python file. Once the if statement is finished, a return false statement is added to the end of each boolean function in case the observation state meets none of the necessary conditions for that action. That's how a policy is turned into generated code.

Experiments

The Environments

Before explaining the experiments, let me give an overview of what the environments used for the experiments are and how they work. For my project, I picked 3 environments of varying complexity, CartPole, MountainCar, and LunarLander. These environments are part of the OpenAI Gym package, a python interface for representing reinforcement learning problems. They are all single agent, static, discrete, deterministic, and sequential environments. CartPole and MountainCar are fully observable, while LunarLander is partially observable.

CartPole CartPole is a 2-dimensional environment where a cart that's on a track running along a horizontal axis is balancing a pole on top of it. The only 2 actions that an agent can take in this environment are to either move left or right. The values that an agent can observe in response to an action are the position of the cart, the velocity of cart, the angle of the pole being balanced on the cart, and the angular velocity of the pole. The goal in this environment is to keep the pole on top of the cart balanced upright for as long as possible or until the episode ends at 500 time steps, the cart must also stay within a certain range on the x-axis. The reward for an episode represents how long the pole on the cart stayed balanced and on screen. An episode that ends with a episodic reward above 475 is considered successful. A score of 500 indicates that the pole stayed balance for the entire episode.

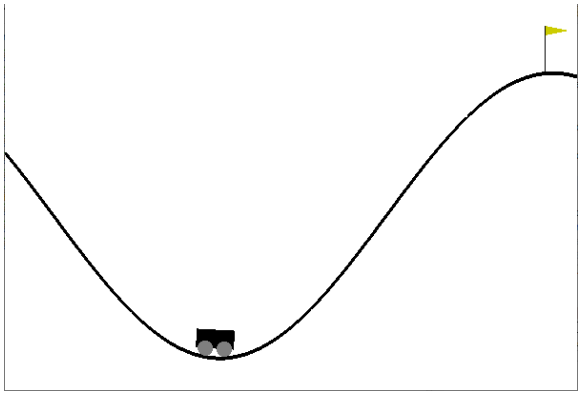

Figure 2: MountainCar

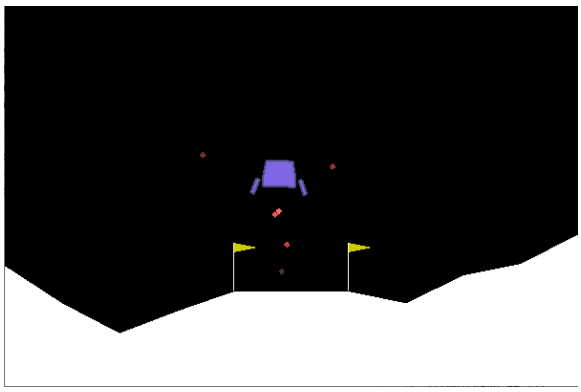

Figure 3: LunarLander

MountainCar It is a 2-dimensional environment where a car is at the bottom of a valley in between two mountains, one on the left and a bigger mountain on the right with a flag on top. An agent can only take 3 actions in this environment: accelerate left, do nothing, or accelerate right. The values an agent can observe in response to an action are the position of the car along the X-axis and velocity of the car. The goal of MountainCar is for the car to move past the flag at the top of the taller mountain before the episode ends at 200 time steps. The car can not build up enough momentum to move up the mountain on the right by just accelerating right, the car has to move back and forth until it has enough momentum to climb the mountain. The reward for an episode represents how quickly an agent can move the car past past the flag. A reward of -200 would indicate that the car never made it to the flag before the episode ended, any score less negative than -200 means the car did reach the flag, the higher (less negative) the score, the better. A score higher than -110 means the environment is considered successful.

LunarLander Finally, we have LunarLander, a 2-dimensional environment where an agent must pilot a landing craft and safely land on a flat surface between 2 flags surround by a randomly generated terrain. At the start of an episode, the lander is placed at the top center of the screen, with a random initial force applied to its center of mass. The only actions an agent can take in this environment are to do nothing, use the left engine, the main engine, or the right engine. An agent gets eight values it can observe in response to an action: the x and y coordinates of the lander, the x and y velocity of the lander, the angle and angular velocity of the lander, and two boolean values corresponding with the left and right leg of the lander that indicate if a leg is in contact with the ground. The values we cannot observe are the various points of elevation of the terrain. The goal of an agent is to pilot the lander such that it lands on the landing pad, and does so while landing upright, and with little force before the episode ends. there are several factors that affect the calculation of the reward. starting at zero, the reward is penalized by -0.3 every timestamp The agent uses the main, and by -0.03 every timestamp The agent uses either the left or right engines. Moving from the starting position and coming to a rest on the landing pad rewards between 100–140 points. Crashing the lander gives a penalty of -100 points, while landing rewards 100 points. Each leg of the lender that is in contact with the ground rewards another 10 points. An episode that ends with a total reward above 200 is considered successful.

Experimental Methodology

In order to test the capability of agents that use the generated code in comparison to agents that use the original reinforcement learning policies, I did a series of tests with each environment.

I generated a sequence of 100 random numbers between 10 and 1000 to act as random seeds for the environments. The random seed fixes the state of an environment to a specific configuration. For CartPole and MountainCar, the seed determines the position and orientation of the agent. For LunarLander, the seed determines that and the shape of the terrain surrounding the launch pad. The outputs of the policies, binary decision trees, and generated code are deterministic based on the seed of the environment. This means that an agent with a specific policy will achieve the same episodic reward for all episodes with the same seed.

So with each environment, I had a set of agents do 100 episodes with the same set of 100 random seeds. First as baselines for comparison, I had an agent picking actions at random, then an agent using a reinforcement learning policy to chose actions. Then agents using a binary decision tree fitted on 10 episodes of policy agent data, and the Python code interpreted from that BDT in order to compare performance. Finally, agents were tested with BDTs fitted on data over 40, 100, and 200 episodes in order to test how more data affects performance. After testing the agents for 100 episodes, I average the episodic rewards and compare the results to other agents.

Additionally, in order to give a sense of what these BDTs look like, I made a chart noting the max depth and number of nodes each tree has.

Results

Looking at the chart of results we can make some conclusions. The random agents for all three environments roundly perform poorly; getting an average reward of 20.69 in Cart-

# of training episodes	CartPole		MountainCar		LunarLander	
	max depth	node count	max depth	node count	max depth	node count
10 episodes	12	351	7	37	30	1125
40 episodes	17	1117	10	65	34	3585
100 episodes	19	2309	10	73	36	8003
200 episodes	21	4157	11	77	31	14641

Agents	Environments		
	CartPole	MountainCar	LunarLander
random	20.69	-200	-171.0455583
policy	500	-103.07	248.0189265
tree 10 eps	496.63	-104.47	-147.8376632
code based on 10 episode tree	496.63	-104.47	-147.837636
tree 40 eps	500	-103.12	91.175442
tree 100 eps	500	-103.06	215.4990992
tree 200 eps	500	-103.04	220.0045474

pole, -200 in MountainCar, and -171.0455583 in LunarLander when the threshold for a solved environment is an average episodic reward over 100 consecutive episode that's greater than 475, -110, and 200 respectively (OpenAi 2022). Conversely, the agents using the reinforcement learning policies perform well as expected, since they were picked because they could achieve a solved reward in those environments, getting an average reward of 500, -103.07, and 248.0189265.

The agents that run on Python code and the agents using the respective BDT that code was generated from have the exact same average episode rewards. This is expected since the proccess for interpreting the BDT is using the exact values the BDT uses for making decisions, the enviornments are deteministic, so the same decisions get the same outcome and reward.

Finally, we look at the trends of the agents using BDTs fitted with more episode data. For CartPole, with the exception of the 10 episode tree agent getting an average reward of 496.63, all the other tree agents achieved the maximum possible reward of 500.

For MountainCar, there is a slight upward trend in average reward as the trees are fitted on more data, getting -104.47, -103.12, -103.06, and -103.04 respectively. However, outside of the 10 episode tree agent being 1 timestep slower, the rest are all functionally achieving the same reward of about -103.

With LunarLander, a definite trend can be observed with the tree agents in relation to how much each was fitted. The more episodes that the BDTs were fitted for, the better the average reward, going from -147.8376632 for the 10 episode tree, 91.175442 for the 40 episode tree, 215.4990992 for the 100 episode tree, and 220.0045474 for the 200 episode tree agent.

Looking at the max tree depth and node count for the BDTs is interesting.

The BDTs for Cartpole are about what I expected. They are not pruned any, so they are very big, and the number of nodes scales with number of episodes.

The statistics for the BDTs of MountainCar are unexpected for me. In terms of node count, the trees for MountainCar are much smaller compared to the other environments. The tree node count doesn't increase at a rate scaled with the number of episodes, going from 37, 65, 73, and 77 nodes respectively. The max tree depth also doesn't grow much, starting 7 levels deep, staying at 10 levels for the 40 and 100 episode trees, and 11 levels for the 200 episode tree. This might be due to how in the MountainCar environment, outside of the beginning variance of an episode, the actions an agent takes will be largely the same. It'll always accelerate right, left, and right again in a episode.

For the statistics of the LunarLander BDTs, the trees are even bigger, they have about 3.5 times more nodes than the CartPole trees, coming in at 1125, 3585, 8003, and 14641 nodes. Interestingly, the tree depth decreases for the 200 episode tree unlike the other environments, the max depth going 30, 34, 36, and then 31 levels deep.

Discussion

CartPole Discussion

For CartPole, the random agent is the only agent that got a score less than the solved threshold of 475 in this environment. It got an average reward of 20.69, which means that on average over 100 episodes, the pole on the cart stayed balanced for only 20.69 time steps. To observe an average episode for the random agent, this looks like the pole on the cart begins to fall over as soon as the episode starts.

All of the other agents, such as the ones using a policy, a BDT, or generated code to pick actions, got an average reward over 475. The trees that were fitted with more than 10 episodes of data got a perfect average reward of 500, meaning the pole stayed balanced for the entirety of an episode for all 100 episodes. As far as what this looks like, the more fitted a tree, the less it will drift in a direction. For the 10 episode tree, the pole never actually looses balance, it just slowely drifts to one side of the screen, out of bounds. The policy and trees with more episodes, may drift in a direction initially, but then stay in place balancing the pole.

MountainCar Discussion

The random agent does very bad, it isn't able to build momentum in any direction so it stays put at the bottom of the valley, getting a average reward of -200 because it never got past the flag at the top of the mountain in any episode. The agent using the DQN policy for its actions performs well, getting a average reward of -103.07, meaning the car was able to move pass the flag at the top of the mountain in 103.07 times steps on average. The agents that use trees or generated code, perform about as well as the policy, with the

generated code performing exactly the same as the tree it is based off of.

LunarLander Discussion

Like the other environments, the agent that picks only random actions performs very poorly. The lander can't maintain a thrust in any particular direction, so it just drops to the surface and crashes.

The agent using the PPO policy performs very well in most seeds. The lander quickly stabilizes from it's initial falling trajectory and then slowly descends downward toward the landing pad, regularly getting an episodic score above 200. There are two general situations where the policy agent receives a score below 200. The first is when the lander lands on terrain that slopes to an elevation below the landing pad and steep enough that the lander will slide away from the landing pad. The agent will fire whichever side engine will stop or slow its descent until the episode times out, penalizing it for using its engine and also not being rewarded the 100 points for successfully coming to a rest. The other typical scenario for a unsuccessful landing with the policy agent is if there is a severe peak in the terrain, and the lander descends onto it. Like sitting on a thumbtack, the body of lander is pierced from below and the agent is penalized 100 points as if it had crashed.

Just like the previous environments, the agent that uses code generated off of a binary decision tree performs exactly the same as the agent using the BDT used as the template. Unlike the previous environments though, LunarLander is much more complicated and a more pronounced trend is observed in the agents using BDTs to pick actions. The agents using BDTs fitted with fewer episodes have a great variability in results; sometimes safely landing and getting scores above 200, other times getting scores much worse than the random agent due to flying out of bounds rather than just crashing. Agents using BDTs fitted with more episodes, however, are more likely to adjust and descend like the policy agent.

Future Work

The work is never done. There are several ways that someone else could take the work from this project and expand on it in the future.

The first would be examining ways of pruning the binary decision trees or making the generated code more succinct. The generated trees can get very big and resulting Python code can be very long, especially for more complicated environments like LunarLander. One of the challenges is that for LunarLander, there is a significant decrease in performance well before the tree is a reasonable size with just regular cost-complexity pruning. More than just making the BDT smaller needs to be done, the generated code should combine conditions where possible and eliminate redundant ones. The benefit of this work would be that the generated code would be easier to understand in its entirety.

Another way to expand on this work would be to try this process with environments that have continuous action spaces. The BDTs and code only decide which action should be taken, there isn't any decision about how much or how long an action should be done, or if multiple actions need to be done at the same time. Trying to represent a policy like that with one BDT would probably lead it to rapidly ballooning in size. Figuring this out would allow for this process to be used on a wider range of problems.

One more direction to take this work would be experimenting with scalability in terms of state space size. For example, breaking episodes in to stages, or modeling time for an environment, and seeing if using generated code for specific section of time makes for a better agent or more understandable code. Like the problem of continuous action spaces, making the state space bigger may also balloon the decision tree fitted from this bigger data and maybe better suited being represented with multiple BDTs. Doing this could lead to breaking the generated Python code down into smaller and easier to understand functions.

Conclusion

While there has been much research in developing explainable AI methods, little has been done to create explainable policies for reinforcement learning. In this work, I seek to explore how we can construct inherently interpretable policies. By using a decision tree to model a trained reinforcement learning policy, I have shown that I can construct interpretable policies that retain their integrity with respect to the original policy. In addition, I provide a method for converting these decision trees into Python code for easier use. My hope with this process is that while the agent using Python code may not get quite as high a reward on average as the original reinforcement learning policy, people will be inclined to use the policies that are using generated code because those policies can be read, they can be debugged, and they can be changed in a way the original cannot.

References

Adadi, A., and Berrada, M. 2018. Peeking inside the black-box: a survey on explainable artificial intelligence (xai). *IEEE access* 6:52138–52160.

OpenAi. 2022. gym. https://github.com/openai/gym/wiki.

Shikora, M. 2018. Mountaincar-v0. https://github.com/mshik3/MountainCar-v0/tree/master.