Learning Programmatic Policies from ReLU Neural Networks

by

Spyros Orfanos

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

Oblique decision trees use linear combinations of features in the decision nodes. Due to the non-smooth structure of decision trees, training oblique decision trees is considerably difficult as the parameters are tuned using expensive non-differentiable optimization techniques or found by extensive search of a discretized space. Recent work showed that one can induce oblique decision trees from the derivatives of ReLU neural networks. For learning with gradient descent, the derivative-based model requires one to anneal from a smooth approximation of ReLU activation functions to ReLUs during training and to use a dynamic programming algorithm to efficiently compute the gradients.

In this thesis we show that regular ReLU neural networks trained with backpropagation can be written as oblique decision trees. We also show that hidden units from ReLU networks can be used to implicitly train oblique decision trees using computationally efficient algorithms for axis-aligned trees. Our result provides not only simple and efficient ways to induce oblique decision trees, but effective methods for synthesizing programmatic policies. This is because oblique decision trees can be seen as programs written in a domain-specific language commonly used in the programmatically interpretable reinforcement learning literature. All one needs to do is use a ReLU neural network to encode the policy, and learn using any policy gradient algorithm. Our methods can then map the policy learned with gradient descent to a program. Empirical results show that our approaches for synthesizing programmatic policies is competitive with the current state of the art if the synthesized programs are small; our methods outperforms the state of the art in almost all control problems evaluated if it is allowed to synthesize longer programs.

Preface

Parts of this thesis are under review as a conference paper with shared first co-authorship with Levi H.S. Lelis in the 32nd International Joint Conference on Artificial Intelligence. This thesis extends the results presented in the paper with new experiments for PID controller policies, discrete action domains, and supervised learning results using densely-connected networks.

Acknowledgements

I would like express my deepest gratitude to my advisor, Levi Lelis, for his endless support and invaluable guidance throughout my degree. Levi's positivity kept me motivated during ups and downs of research and his supervision, insights, and advice profoundly impacted my research acumen and creativity. Thank you so much, Levi.

I would also like to thank my peers from the Amii Synthesize reading group for introducing me to program synthesis and for their insightful discussions. I would like to acknowledge the financial support from the Natural Science and Engineering Research Council of Canada (NSERC) for the Alexander Graham Bell Canada Graduate Scholarship - Master's (NSERC CGS-M) and the Faculty of Graduate Studies and Research (FGSR) for the Walker H. Johns Graduate Fellowship and the Science Graduate Scholarship.

Lastly, I would like to thank my family for their endless support. I am grateful to my sister, Katerina, for her encouragement, my grandparents for their inspiring wisdom, and my parents, Chrisi and Andreas, for their tremendous support, motivation, and care. Each of you always believed in me and gave me the courage to pursue my ambitions.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Much of the recent success in artificial intelligence (AI) can be attributed to artificial neural networks. Artificial neural networks are widely used in nonlinear function approximation due to their ability to generalize complex hypotheses and their efficient training algorithms which leverage modern computing abilities. Approximate solution methods in reinforcement learning (RL) [41] often use a neural network to represent the policy, which defines the agent's behaviour. Such methods have defeated world champions in the games of chess, Go, and Dota 2 [31], [40] and have been applied to research areas such as robotics. While their success is remarkable, they produce opaque 'black-box' policies that are hard to interpret or explain, resulting in safety concerns when deployed in real-world settings.

In contrast, interpretable models can improve our understanding of how problems are solved since the agent's behaviour is known, thus increasing our trust and confidence in the model. Emphasis on interpretable AI models has lead to the development of the programatically interpretable reinforcement learning (PIRL) [46] framework in which policies are encoded in human-readable computer programs. Such policies are more amenable to verification [4] and tend to be easier to understand [46].

Decision trees are one of the most interpretable models in machine learning and widely used to encode PIRL policies. Decision trees are non-differentiable due to their if-then-else architecture, so one needs to *search* over a large and non-smooth spaces of programs. This contrasts gradient descent methods, which *'learn'* by adjusting the model parameters by a small amount in the direction opposite to the gradient. This is consequential because RL algorithms employ the latter approach by using the agent's experience to learn online, while interacting with the environment. Without gradient descent methods for incremental parameter updates, finding an optimal policy directly from the reward signal is generally infeasible as it cannot learn from previous experience.

As such, much of the work in PIRL relies on the search signal a neural oracle provides to guide the search [2], [4], [45], [46]. That is, RL is reformulated into a much simpler supervised learning problem known as imitation learning. VIPER [4] uses axis-aligned decision trees to encode programmatic polices, which are trained by imitating a pre-trained neural network policy. Axis-aligned decision trees [7] are particularly interpretable because the user needs to reason about a single feature at each node of the tree. Using a single feature in the decision nodes also allows for computationally efficient training algorithms; however, it severely limits the capacity to induce effective hypotheses. Program synthesis approaches to PIRL [2], [45], [46] also rely on imitation learning; however, they can induce more expressive decision tree programs that can encode more complex hypothesis. The domain-specific language (DSL) is the high-level language that defines the set of allowed programs. Program synthesis methods search over a possibly large DSL and require computationally expensive non-differentiable optimization of the tree's parameters. As a result, the scalability of program synthesis methods in the PIRL setting is limited to small programs.

Qiu and Zhu [32] argued that this imitation learning approach could lead to weak programmatic policies due to a possible representation gap. That is, it is possible that the neural policy used as an oracle cannot be represented in the programmatic space. In addition, these approaches are suboptimal since the search is guided by imitation learning instead of the environment's reward signals. Qiu and Zhu recently introduced a method, π-PRL, that uses a smooth approximation of a DSL to allow for the use of online gradient-based learning. π-PRL uses policy gradient algorithms to learn a programmatic policy, which is encoded as a differentiable approximation of a decision tree. The main drawback of π-PRL is that the time complexity for evaluating a state during training is exponential on the depth of the tree. Thus, the method can only synthesize small programs (they induced trees of depth 6 in their experiments).

Recent work in supervised learning introduced locally constant networks (LCN), a type of neural network model that can be mapped to an oblique decision tree [22]. Oblique decision trees account for all features of the problem in each node. Oblique decision trees are of particular interest because they can be seen as a program written in a commonly used DSL for synthesizing programmatic policies [13], [32], [39], [46], [51]. This language allows if-then-else structures with affine transformations of the input values in the decision nodes, which is exactly what oblique trees can represent. LCN learns a function of the derivatives of a ReLU neural network. As a result, the gradient signal for training with gradient descent is insufficient. The model is trained with an annealing procedure where it transitions from a smooth approximation of ReLU to ReLUs. LCN's training procedure also employs a dynamic programming procedure to compute all gradients efficiently as it cannot leverage gradient computation algorithms such as backpropagation [37].

In this thesis we present two methods that use ReLU networks trained with backpropagation to induce oblique decision trees.The first method, Oblique Trees from ReLU Neural Networks (OTR), leverages the activation patterns of a ReLU neural network [3], [15], [22], [28], [35], [50] to represent the network as an equivalent oblique decision tree. A ReLU neuron is active if it produces a value greater than zero and is inactive if it produces a value of zero. An activation pattern defines which neurons are active and which neurons are inactive for a given input. Like LCN, OTR rewrites the function of each neuron, one for each possible activation pattern, as a linear combination of the inputs. This is because the ReLU function produces either a linear combination of its inputs or the value of zero. Each path of the induced tree represents a different activation pattern and each decision node on a path represents a different neuron of the network. LCN maps a constant to each leaf node, so each path of the tree is locally constant. We show that this is not necessary, and that ReLU networks trained with back-propagation can also be mapped to oblique decision trees. If the network has n neurons, the tree induced by OTR will have 2^n paths, one for each possible activation pattern, and its depth will be n. In the leaf nodes, OTR returns a linear model instead of a constant.

The second method, Neurally Augmented Decision Trees (AugTree), is an imitation learning algorithm that uses the hidden units of the neural network oracle to augment the input space. AugTree leverages the computationally efficient CART [8] algorithm for training axis-aligned trees to implicitly induce oblique decision trees. We hypothesize that having access to network's hidden units can improve the decision tree's capacity to imitate the oracle and reduce the representation gap. In contrast to OTR which maps ReLU networks to oblique trees, AugTree trains oblique trees using pieces of ReLU networks. As such, AugTree approximates ReLU neural networks.

While both methods apply to supervised learning (e.g., regression and classification problems), we study them in the context of PIRL - namely to encode programmatic policies as oblique decision trees. Empirical results on several control problems show that the programs OTR and AugTree synthesizes are competitive with the current state-of-the-art in programmatic RL. In contrast to other methods in PIRL, ours can scale to train longer programs, which could result in stronger programmatic policies that are possibly less interpretable than those encoded in shorter programs.

1.1 List of Contributions

- The main contribution of this thesis is to show that a mapping between small ReLU networks and oblique decision trees can be used to supersede more complex PIRL approaches if the DSL only supports programs with if-then-else structures and affine transformations of the inputs.

- Accompanying code to map ReLU networks to oblique decision trees is publicly available at `https://github.com/spyrosUofA/OTR`.

- Neurally Augmented Decision Trees, an efficient imitation learning algorithm that can implicitly train oblique decision trees.

Chapter 2

Background Material

In this chapter we review algorithms for training oblique decision trees. Oblique decision trees are of particular interest because they have been used in other works (e.g., [13], [32], [39], [46], [51]) to encode programmatic reinforcement learning policies. Before outlining current approaches to programatically interpretable reinforcement learning, we summarize the necessary background material, namely program synthesis, reinforcement learning, and imitation learning.

2.1 Oblique Decision Trees

An oblique decision tree T is a binary tree whose nodes r define a function $P \cdot X + v \leq 0$ of the input X, where P and v are parameters of r. Each leaf of T contains a prediction for X. A tree T produces a prediction for X by defining a path from the root to a leaf of T, which we call an *inference path*. An inference path is determined as follows. If $P \cdot X + v \leq 0$ is true for the root, then one follows to the root's left child; the right child is followed otherwise. This rule is applied until we reach a leaf node. For classification tasks, the leaves return a label. For regression tasks, we consider linear model trees [19], where each leaf returns the value of $P \cdot X + v$ as the prediction for X.

2.1.1 Early Methods

Early methods for inducing oblique decision trees such as CART with linear combinations (CART-LC) [7] use a local search algorithm to find the hyperplanes of each node. Cart-LC starts with an arbitrary hyperplane and it perturbs it while it is able to make improvements with respect to a metric of impurity. Heath [16] used simulated annealing to search for the hyperplanes. OC1 uses an improved method for perturbing the coefficients of the hyperplanes. Linear Machine Decision Trees (LMDT) [44] trains a linear classifier for each node of the tree. The partition of the training data that the classifier induces defines the hyperplane of each node. Another classifier is then recursively trained on each of the resulting parts, until a purity threshold is achieved. Murthy et al. [29] compared OC1 and LMDT and found that OC1 tends to perform better than LMDT on non-linear problems.

HHCART applies a transformation to the training data that tries to increase the chances of an axis-aligned partition resulting in high purity values [48]. The axis-aligned partition encountered in the transformed space is equivalent to an oblique partition in the original space. Wickramarachchi et al. [48] showed that HHCART is competitive with OC1. We use HHCART as a baseline in our supervised learning experiments. Another algorithm we use as baseline in our experiments is TAO [12]. TAO iterates from bottom to top while optimizing for the parameters of linear classifiers of each node. TAO was shown to produce good empirical results, but it only works for classification problems.

All the algorithms reviewed so far can suffer with poor local minima. Others have developed exact methods for finding optimal partitions in trees [5], [6]. The drawback of these methods is that they do not scale to trees of even moderate size. Lin et al. [24] showed how to scale the training of optimal trees, but only for axis-aligned trees, not oblique trees.

OTR is able to leverage all the machinery developed for training neural networks, such as stochastic gradient descent for escaping local minima and backpropagation for efficiently computing gradients. As a result, and as we show in our experiments, OTR is able to induce much better trees than these previous methods. Note that while the main focus of this thesis is programmatic reinforcement learning, previous methods for training oblique trees are unsuitable for online learning. As such, we separately evaluate and compare these approaches to OTR on supervised learning problems (regression and classification).

2.1.2 Locally Constant Networks

Consider a ReLU neural network $f_\theta : X \in \mathbb{R}^d \to Y \in \mathbb{R}^l$ with m hidden layers and n_i neurons in the i^{th} layer. The j^{th} neuron in layer i before and after applying the ReLU activation is denoted Z^i_j and A^i_j, respectively, for $j \in \{1, 2, ..., n_i\}$. The set of neurons in the i^{th} layer before and after applying the ReLU activation are denoted as $Z^i \in \mathbb{R}^{n_i}$ and $A^i \in \mathbb{R}^{n_i}$, respectively. We denote the concatenation of neurons $(A^0, A^1, ..., A^i)$ as $\tilde{A}^i \in \mathbb{R}^{\tilde{n}_i}$, where $\tilde{n}_i = \sum_{j=0}^{i} n_i$ and $A^0 := X$. A neuron's excitement is defined as $O^i_j = \frac{\partial A^i_j}{\partial Z^i_i} = \mathbf{I}[Z^i_j \geq 0] \in \{0, 1\}$, where $\mathbf{I}[\cdot]$ is the indicator function. The activation pattern, which represents the excitement of all neurons, is denoted $\tilde{O}^m = (O^0, O^1, ..., O^m) \in \{0, 1\}^{\tilde{n}_m}$ where $O^i = (O^i_1, O^i_2, ..., O^i_{n_i})$. There are $2^{\tilde{n}_m}$ possible activation patterns.

Recent work used the derivatives of ReLU neural networks to train locally constant networks (LCN) that are equivalent to oblique decision trees [22]. Like OTR, LCN leverages the piece-wise linearity of ReLU networks [21] to map the $\tilde{n}_m$ neurons of f_θ to the decision nodes of a depth $\tilde{n}_m$ oblique tree. More detail on how the neurons are written as linear combinations of the features and then arranged in the tree is provided in the presentation of OTR (Chapter 3). For now, we accept that each inference path of the decision tree corresponds to a particular activation pattern of the network. In each leaf node, and hence for each activation pattern, LCN returns a the locally constant output given by a table g, where $g : \tilde{O}^m \in \{0, 1\}^m \to Y \in \mathbb{R}^l$. This describes the canonical architecture for LCNs; however, it is not practical for learning since g is not differentiable. The standard architecture is used in practice, where the tabular g is replaced by another neural network $g_\phi : J_X \tilde{A}^m \in \mathbb{R}^{(m \times d)} \to Y \in \mathbb{R}^l$. J_X denotes the Jacobian, so g_ϕ that maps the gradients of f_θ to the output of a leaf node. Since f_θ is piece-wise linear, each activation pattern is locally linear (i.e., the network reduces to a linear transformation for any activation region). Therefore, the gradients with respect to the input $J_X \tilde{A}^m$ are locally constant. The authors prove that there exists a bijection between $J_X \tilde{A}^m$ and $\tilde{O}^m$, from which it follows that $g_\phi(J_X \tilde{A}^m)$ and $g(\tilde{O}^i)$ are equivalently representative.

While the standard architecture is differentiable, inactive neurons do not provide any useful gradient signals so LCNs cannot be trained as easily as regular neural networks. To overcome this, LCN anneals between softplus activation functions, an infinitely differentiable approximation of the ReLU function, and ReLUs in order to increase the gradient signal. In the final training epoch, the exact ReLU function is used, so $J_x \tilde{A}^M$ produces exactly $2^{\tilde{n}_m}$ unique and locally constant gradients, which are input to g_ϕ. LCN computes the gradient of all neurons with respect to the input values, which requires a special dynamic programming approach to be done efficiently, in a single pass.

To summarize LCN, the underlying model f_θ has $2^{\tilde{n}_m}$ locally constant gradients which are the inputs to g_ϕ. The neurons from f_θ map to the decision nodes of the induced tree. g_ϕ outputs the final prediction for each leaf node. The standard LCN architecture, which is used in practice, assumes only one neuron in each layer and has a more complex training procedure than what is required to train regular ReLU networks. Furthermore, LCN's annealing procedure makes it unsuitable for incremental learning as it is not clear how each update repeats the annealing procedure.

With OTR, we show that all one needs to do to train an oblique tree is to train a regular ReLU neural network (fully connected or densely connected) with backpropagation. For classification tasks, the OTR tree is deeper by $l-1$ levels compared to LCN. For regression tasks, the OTR tree outputs a linear model instead of a constant.

2.2 Program Synthesis

Program synthesis methods use input-output pairs to construct a program, represented as an *abstract syntax tree* (AST), that correctly maps the input values to the output values. As an example, consider the following set of input-output pairs: $\{([x_1 = 1, x_2 = 2], 2), ([x_1 = 10, x_2 = -2], 10), ([x_1 = -1, x_2 = 0], 0)\}$, where x_1 and x_2 are the input values. A solution to this problem can represented by the following program: **if** $x_2 < x_1$ **then** x_1 **else** x_2, whose AST

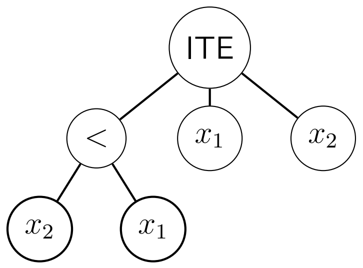


Figure 2.1: AST for program **if** $x_2 < x_1$ **then** x_1 **else** x_2

is shown in Figure 2.1. The root of the tree, `ITE` represents an if-then-else rule, where its leftmost child represents the Boolean expression, the middle child represents what is executed if the Boolean expression returns true, and the rightmost child what is executed if the Boolean expression returns false.

The domain-specific language (DSL) is the high-level symbolic language that defines the set of allowed programs and is represented as a context-free grammar. The DSL is composed of non-terminal symbols, terminal symbols, and a set of rules describing how a symbols can by replaced by another. An engineer specifies the DSL, so it can be arbitrarily general. For example, one may allow for binary operations (e.g., 'AND', 'OR', 'XOR'), arithmetic operations (e.g., addition, exponentiation), higher-order functions (e.g., fold), loops, and pre-determined functions. An example of a DSL equivalent to oblique decision trees is given in Figure 4.5, where E is the initial symbol, that can be replaced by either C or **if** B **then** E **else** E. The DSL accepts the program **if** $x_2 < x_1$ **then** x_1 **else** x_2 with the following rules: $E \to$ **if** B **then** E **else** E; $B \to C < C$, yielding **if** $C < C$ **then** E **else** E; $C \to x_2$ and $C \to x_1$ for the first and second C, respectively; $E \to C$ for the two Es, which are then replaced by x_1 and x_2 with $C \to x_1$ and $C \to x_2$.

Although designing a large, general DSL sounds enticing, defining a concise, succinct one is essential since program synthesis methods must search in the space of programs the DSL defines. Bottom-up search (BUS) [1] is a synthesis algorithm which enumerates all possible programs up to a specified size, removes the programs that are 'observationally equivalent' (i.e., if two programs produce exactly the same output for the set of inputs of interest, then BUS considers only one of the programs in search), and then evaluates each program. Such

$$\begin{aligned}E &::= C \mid \textbf{if } B \textbf{ then } E \textbf{ else } E\\ B &::= C < C\\ C &::= x_1 | x_2\end{aligned}$$

Figure 2.2: DSL for oblique decision trees (discrete output).

enumerative search algorithms are simply not feasible on large DSLs due to the size of programmatic space.

An alternative to enumerative search algorithms is local search algorithms. Simulated annealing (SA) [20] is a stochastic, local search algorithm where a temperature parameter specifies the greediness of the search. SA starts by randomly initializing an AST. Each iteration of SA mutates the current program by replacing a randomly chosen non-terminal node with a random sub-tree (e.g., the second child of `ITE` in Figure 2.1 could be replaced by another `ITE` tree). The new program is accepted based on a probability that depends on the temperature and the difference in utility compared to the current program. The temperature starts off high, so a wide range of the programs are accepted, which is useful for escaping local minima. The temperature is decreased over time, so SA evolves into a greedy search that only accepts strong programs. Local search algorithm do not always find the optimal solution, and having a large DSL can hinder the optimization. In any case, finding optimal coefficients or constants in an AST requires searching over a discrete set of allowed constants, or non-differentiable optimization techniques such as Bayesian optimization [27]. As a result, the scalability of program synthesis methods in the PIRL setting is limited to small programs.

2.3 Reinforcement Learning

Reinforcement learning (RL) [41] focuses on training an agent to act in a way that maximizes a reward signal. The environment that the agent interacts with is often formalized by a Markov decision process (MDP), which is defined by the 5-tuple $\{\mathcal{S}, \mathcal{A}, R, P, \gamma\}$ consisting of the set of states, the set of actions, the reward function, the transition probability matrix, and the discount factor, respectively. At each time step t, the agent observes a state $S_t = s$ and then samples an action $A_t = a$ from its policy $\pi(a|s)$. In the discrete control setting, $\pi(a|s)$ represents the probability distribution over the possible actions. In continuous control, actions are real numbers, and the policy typically specifies the parameters of a Gaussian distribution from which actions are sampled. In either setting, after executing an action, a reward signal $R_{t+1} = r$ is observed. The transition to the next state $S_{t+1} = s'$ is specified by the transition probability $p = P(s'|s,a)$. Episodic MDPs eventually terminate and so this interaction cycle gives rise to the sequence $\{S_0, A_0, R_1, S_1, A_1, ..., R_T, S_T\}$, which is called an *episode*. G_0 is the return of an episode, where $G_t = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1}$. The control problem (i.e., finding a policy that maximizes return) is considerably difficult since immediate actions have long term consequences on the objective.

Program synthesis methods approach this problem by searching the policy space, denoted Π, directly, to synthesize a policy. Synthesized policies are evaluated by the return they produce over some number of episodes. In algorithms like SA, the return may guide the search for subsequent policies. Such methods, however, do not consider what happened during the the episode, which means they cannot assign credit to specific actions or make updates to the agent as it interacts with the environment. In contrast, RL algorithms make incremental updates to the policy online, while interacting with the environment. Model-free RL algorithms leverage value functions to facilitate trial-and-error learning. The state-value function, $v_\pi(s)$, is defined as the expected return given the agent is currently in state s and acts according to a policy π. That is,

$$v_\pi(s) = E_\pi[G_t|S_t = s] = E_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s] = q_\pi(s, a = \pi(s))$$

The state-value function is useful for prediction and for ranking policies. An optimal policy, denoted π^*, is one that maximizes $v_\pi(s)$ for all states. Formally, π^* is optimal if $\forall_{\pi\in\Pi, s\in\mathcal{S}}\ v_{\pi*}(s) \geq v_\pi(s)$. The action-value function $q_\pi(s,a)$ is useful in the control problem. The action-value function is defined as the expected return given the agent is currently in state s, takes action a, and acts according to policy π thereafter:

$$q_\pi(s,a) = E_\pi[G_t|S_t = s, A_t = a] = E_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s, A_t = a]$$

Classic RL algorithms estimate the optimal state-value (denoted v_*) or the optimal action-value (denoted q_*) by maintaining a table with the estimated value for each state or state-action pair, respectively. It follows that $\pi^* = \text{argmax}_a \; q_*(s,a)$. By interacting with environment, the values for the table are updating using the temporal difference (TD) error. For example, Sarsa, which uses TD(0) error, has the following update rule for estimating $q_* \approx Q$:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

2.3.1 Policy Gradient Methods

Often the state space of an MDP is too large to enumerate, so learning an optimal policy using a tabular representation is not possible. Instead, approximate solutions methods in RL use function approximation (e.g., linear models, neural networks) to approximate the optimal policy. This is typically done with parametric approximation of the action-value (i.e., $q_* \approx \hat{q_\psi}$) or of the optimal policy directly (i.e., $\pi^* \approx \pi_\theta$). Both approaches use gradient descent-based learning to make incremental model updates, so differentiable models are required.

The latter approach describes policy gradient methods, which offer several advantages over the former: the policy gradient theorem is a form of policy improvement, updates to the policy are smooth and incremental, and they extend naturally to continuous action spaces. Actor-critic algorithms are a type of policy gradient method which learn two models: an actor π_θ representing the policy, and a critic $\hat{v_\psi}$ (or $\hat{q_\psi}$) representing the value function. In this thesis, we use various actor-critic algorithms to learn policies represented by ReLU neural networks. In Chapter 3, we show that ReLU networks are equivalent to oblique decision trees, so small actor networks can be encoded as interpretable programs. We briefly describe one such algorithm - proximal policy optimization (PPO) [38]. PPO uses λ-returns, G_t^λ, to make TD(λ) updates to the policy and as the target for $\hat{v}_\psi(s)$. The surrogate loss objective (line 12 of Algorithm 1) clips the estimated advantage to prevent the policy from changing drastically (hence proximal policy), thereby improving the stability of the actor.

Algorithm 1 PPO Pseudo-code

```
1: Initialize network weights $\theta$, $\psi$
2: for each episode $k = 1, ...$ do
3:   Experience: $\{S_t^k, A_t^k, R_{t+1}^k\}_{t=0}^{T_{k-1}}$ by acting according to $\pi_\theta$
4:   Compute: $\{\hat{v}_\psi^k(s_t)\}_{t=0}^{T_{k-1}}$, $\{G_t^{\lambda,k}\}_{t=0}^{T_{k-1}}$
5:   if $k$ mod K = 0 then
6:     $\theta' \leftarrow \theta$, $\psi' \leftarrow \psi$
7:     Compute $\hat{A}_t = G_t^\lambda - \hat{v}_\psi(s_t)$ for each $t, k$ in the entire batch, then
       normalize.
8:     for each E epochs do
9:       Shuffle the batch, slice into N mini-batches
10:      for each mini-batch do
11:        $l(\theta', \psi')$ = -mean$\Big[\zeta_t(\theta, \theta', \psi)\Big]$ + mean$\Big[\left(G_t^\lambda - \hat{v}_{\psi'}(s_t)\right)^2\Big]$
12:        where $\zeta_t(\theta, \theta', \psi) = \min\Big[\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)}\hat{A}_t, clip\left(\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)}, 1-\epsilon, 1+\epsilon\right)\hat{A}_t\Big]$
13:        Update network weights $\theta', \psi'$ using Adam with loss $l(\theta', \psi')$
14:    $\theta \leftarrow \theta'$, $\psi \leftarrow \psi'$
```

2.4 Imitation Learning

Imitation learning is a form of supervised learning that involves minimizing the error on a dataset of state-action pairs labelled by an expert, known as the oracle. In this thesis and related works, the oracle is represented by a neural network. The distribution of states is not independent and identically distributed since they depend on the transition dynamics and the policy. This is significant because distribution of states between the 'learner' policy and oracle may differ. The 'learner' may find itself in a state unseen in the dataset, which could be problematic. Dataset Aggregation (DAGGER) [36] is an online imitation learning algorithm that remedies this by rolling out the 'learner' model for some number of episodes, and then re-labelling the actions with the optimal actions provided by the oracle. Each time the dataset is aggregated, the student model is updated and the cycle continues. The pseudocode for DAGGER is given in Algorithm 2.

Algorithm 2 Imitation Learning with Dataset Aggregation (DAgger)

```
1: Input: POMDP $E$, neural policy $\pi_{\mathcal{N}}, M, N$
2: Initialize dataset $\mathcal{D} \leftarrow \varnothing$
3: Initialize policy $\hat{\pi}_0 \leftarrow \pi_{\mathcal{N}}$
4: for $i = 1$ to $N$ do
5:     Sample $M$ trajectories $\mathcal{D}_i \leftarrow \{(s, \pi_{\mathcal{N}}(s)) \sim d^{\hat{\pi}_{i-1}}\}$
6:     Aggregate dataset $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}_i$
7:     Imitate oracle $\hat{\pi}_i \leftarrow \arg\min_{\hat{\pi}'} \sum_{s \in \mathcal{D}} ||\hat{\pi}'(s) - \pi_{\mathcal{N}}(s)||$
8: Output: $\hat{\pi}^* \leftarrow$ best_reward_policy$(\hat{\pi}_1, \hat{\pi}_2, ..., \hat{\pi}_N)$
```

2.5 Related Work

In programmatically interpretable reinforcement learning (PIRL) [46] one encodes policies as a human-readable programs. Programmatic policies have shown to be easier to understand [46] and formally verify [4]. They were also shown to better generalize to unseen problems [18]. Most approaches to PIRL encode policies as some kind of decision tree program (e.g., axis-aligned decision trees, oblique decision trees, non-linear decision trees). Decision trees are non-differentiable due to their discrete if-then-else structure, so updates require training the policy from scratch by searching over the program space. This is problematic because RL with function approximation makes online updates using stochastic gradient descent. Although searching for a decision tree program directly is possible on simple MDPs [13], most work in PIRL overcomes this in one of two ways. The first way is to guide the program search by way of imitation learning. This search is much simpler as it circumvents many challenges present in RL (e.g., exploration-exploitation trade-off, delayed feedback, learning from bootstrapped targets). The second approach to PIRL is to use a differentiable approximation of a program or program space to allow for gradient descent based learning. We review several oracle-guided and oracle-free PIRL approaches.

2.5.1 Oracle-guided PIRL

Verifiable reinforcement learning through policy extraction (Viper) uses DAgger-style imitation learning to induce axis-aligned decision trees to encode policies [4]. VIPER uses the samples from the dataset according to the 'importance' of each state. A state's importance is estimated using the oracle's action-value function, and calculated as $l(s) = \hat{q}_\phi(s, \pi_\theta(s)) - \min_{a' \in \mathcal{A}} \hat{q}_\phi(s, a')$. That is, states where the difference between taking the best and worst action is big are given more attention. Although VIPER programs can be trained easily using CART, axis-aligned trees are often not very expressive. This is problematic as the programmatic policy may require thousands of nodes to perform well or the program space lacks the capacity to represent the oracle.

Neurally directed program search (NDPS) [46] uses program synthesis with Bayesian optimization to encode policies as programs. Like VIPER, NDPS uses DAGGER to guide the search. The search only enumerates programs from a small, dynamic neighborhood of program templates and parameters that are known to perform well. The authors state that the algorithm is sensitive to the initialization of this neighborhood. In the paper, NDPS is used to synthesize programs with oblique decision nodes, which are of course more expressive than VIPER's axis-aligned decision nodes. Every iteration involves costly non-smooth optimization for each AST in the neighborhood of programs, which severely limits NDPS to synthesizing very small programs.

A major concern with these methods is that one might not be able to represent the teacher's policy in the programmatic space. As a result, the teacher might guide the search toward weak programmatic policies [32]. PROPEL attempts to mitigate this issue by training neural policies that are not 'too different' from the synthesized programs [45]. Unlike the previous methods where the oracle is a fixed, pre-trained neural network policy, PROPEL trains the oracle alongside the programmatic policy. PROPEL uses policy gradients methods to train the neural policy; however, mirror descent is used to update the policy in the constrained policy space. The constrained space is a mix of the oracle's actions and the programmatic policy's actions, so PROPEL only applies to continuous action spaces. The constrained oracle guides the search (program synthesis or decision tree) for the programmatic policy using DAGGER. SKETCH-SA is less likely to suffer from this representation gap problem because it uses imitation learning to synthesize a sketch of a policy;

the policy is synthesized from the sketch by evaluating it directly in the environment [26]. Another caveat with oracle-guided PIRL methods is that they are suboptimal: instead of maximizing the RL objective, the objective is to minimize the error on a dataset.

2.5.2 Oracle-free PIRL

Differentiable or 'soft' decision trees are smooth approximations of decision trees, meaning they can be trained using gradient descent. Like oblique decision trees, the decision nodes of soft decision trees consider linear combinations of all input features. To make the tree differentiable, at each node the probability of taking the right child node is $\sigma(P \cdot X + v)$, where σ is the logistic sigmoid function and $P \cdot X + v$ is the decision node. The output of the tree is defined as the weighted average of the leaf nodes, weighted by the inference path probability. Silva *et al.* [39] use soft decision trees to learn a programmatic policies using policy gradient methods. To improve interpretability, the authors 'discretize' the tree and transform it to an axis-aligned tree. Each decision rule is simplified to $X_i < \frac{-v}{P_i}$, where i is the index of P with the largest value. Unsurprisingly, this simplification worsens the performance on most domains.

Oracle-free programatically interpretable reinforcement learning (π-PRL) [32] constructs a differentiable DSL by approximating if-then-else nodes with the logistic sigmoid function. In doing so, π-PRL leverages policy gradient methods to train the programmatic policy directly, without the need of an oracle. Once again, their DSL requires oblique decision nodes. Learning π-PRL programs requires iterative bilevel optimization, where updates to the AST alternate between policy parameter optimization and policy architecture optimization. Both types of updates maximize the surrogate objective used in PPO (or other policy gradient algorithms). During policy parameter updates, the upper-level program parameters (i.e., weights from sigmoid) are frozen and the low-lever program parameters (i.e., coefficients in decision nodes and leaf nodes) are optimized. For architecture optimization, the low-level program parameters are frozen and the upper-level program parameters are optimized. The model returns an action for a given state during training by evaluating all nodes in the tree and averaging them according to the sigmoid functions. Upon convergence, the tree is discretized by a top-down greedification of the upper-level parameters. Finally, the low-level parameters are further optimized. π-PRL does not suffer from the representation gap because policy gradient algorithms are able to optimize the programmatic policy directly with gradient ascent; however, its drawback is its exponential complexity for evaluating states during training (it needs to evaluate all leaf nodes of the decision tree for every decision during training). As a result, π-PRL is only able to synthesize small decision trees. However, compared to the other PIRL methods we surveyed, π-PRL achieves compelling results in many complex MDPs.

Chapter 3

Oblique Trees From ReLU Networks

In this chapter we present algorithms to induce oblique decision trees from neural networks that use ReLU activation functions in their hidden layers. We first introduce Oblique Trees from ReLU Neural Networks (OTR), a one-to-one mapping from ReLU networks to oblique decision trees. In our presentation we assume a single neuron in the output layer that uses either a Logistic function (classification tasks) or a linear activation function (regression tasks). Later we show how OTR generalizes to multi-class tasks where the network uses a Softmax function in its output layer. Finally, we approximate ReLU networks using oblique decision trees. First, by pruning OTR trees, and then using a novel and computational efficient method we call Neurally Augmented Decision Trees (AugTree).

3.1 Notation

We denote matrices with upper-case letters and scalar values with lower-case letters. We consider fully-connected neural networks with m layers $(1, \cdots, m)$, where the first layer is given by the input values X and the m-th layer the output of the network. For example, $m = 3$ for the network shown in Figure 3.1. Each layer j has n_j neurons $(1, \cdots, n_j)$ where $n_1 = |X|$. The parameters between layers i and $i+1$ of the network are denoted by $W^i \in \mathbb{R}^{n_{i+1} \times n_i}$ and $B^i \in \mathbb{R}^{n_i \times 1}$. The k-th row vector of W^i and B^i, denoted W^i_k and B^i_k, represent the weights

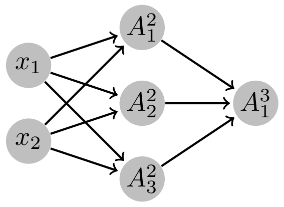
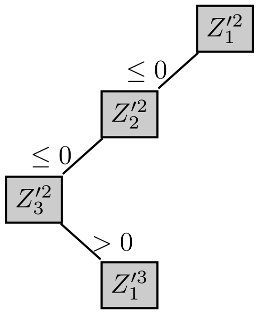
Neural Network	**Weights and Biases**	**Inference Path**
	$W^1 = \begin{bmatrix} -2.7 & -0.8 \\ 0.2 & 2.0 \\ 1.0 & -0.1 \end{bmatrix}, B^1 = \begin{bmatrix} -0.4 \\ 0.6 \\ 1.2 \end{bmatrix}$ $W^2 = \begin{bmatrix} -2.0 & -2.4 & 1.2 \end{bmatrix}, B^2 = \begin{bmatrix} 1.4 \end{bmatrix}$ $P^1 = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1.0 & -0.1 \end{bmatrix}, V^1 = \begin{bmatrix} 0 \\ 0 \\ 1.2 \end{bmatrix}$ $P^2 = \begin{bmatrix} 1.2 & -0.12 \end{bmatrix}, V^2 = \begin{bmatrix} 2.84 \end{bmatrix}$	 $Z_1'^2$ ≤ 0 $Z_2'^2$ ≤ 0 $Z_3'^2$ > 0 $Z_1'^3$
Z′-functions	$Z_1'^2 = -2.7x_1 - 0.8x_2 - 0.4, Z_2'^2 = 0.2x_1 + 2.0x_2 + 0.6$ $Z_3'^2 = 1.0x_1 - 0.1x_2 + 1.2, Z_1'^3 = 1.2x_1 - 0.12x_2 + 2.84$	

Figure 3.1: Neural network (left); set of weights and biases (W and B) of the network and of the inference path for input $X = [0.5 \ -0.5]^T$ of the tree OTR induces (P and V) (middle); inference path for X (right). Each $Z_k'^i$-value (bottom) is equal to its corresponding Z_k^i-value for the input X.

and the bias term of the k-th neuron of the $(i+1)$-th layer. Figure 3.1 shows an example where $n_1 = 2$ and $n_2 = 3$. Thus, $W^1 \in \mathbb{R}^{3\times 2}$ and the first row vector of W^1 ($[-2.7 \ -0.8]$) and the first entry of B^1 (-0.4) provide the weights and bias of the first neuron in the model's hidden layer. Let $A^i \in \mathbb{R}^{n_i \times 1}$ be the values the neurons of the i-th layer, where $A^1 = X$ and A^m is the output of the model. A forward pass in the model computes the values of $A^i = g(Z^i)$, where $g(\cdot)$ is an activation function and $Z^i = W^{i-1} \cdot A^{i-1} + B^{i-1}$. We compute the values of A^i in order of $i = 2, \cdots, m$.

We consider ReLU activation functions: $\text{ReLU}(x) = \max(0, x)$ [30] for all neurons in hidden layers; we consider Logistic and linear activation functions for neurons in the output layer for classification and regression problems, respectively. OTR also works with Leaky ReLUs [25] in the hidden layers and Softmax in the output layer; we discuss the use of such functions in Section 3.2.

3.2 Mapping ReLU Networks to Oblique Trees

OTR produces a one-to-one mapping from ReLU neural networks to oblique decision trees by leveraging the metric of activation patterns of a neural network [35], which considers the "active" and "inactive" neurons for a fixed input. For ReLU functions, the k-th neuron of the i-th layer is active if $Z_k^i > 0$

and is inactive otherwise.

A network with n neurons results in a tree with depth n, where each node on a path from root to a leaf represents a different neuron and each path represents an activation pattern. For such a network, there are 2^n activation patterns and hence 2^n inference paths. Since the ReLU function is piece-wise linear, the output of the network is a series of linear transformations, which can be reduced to a single linear transformation. Once an activation pattern is defined, the Z^i_k-values of the network are simply a linear transformation of the input X [22]. OTR chooses the parameters P and v of each node such that $Z^i_k = P \cdot X + v$ for the node representing the k-th neuron of the i-th layer. For neurons in the first layer, the nodes in the decision tree are simply the neurons from the network, ...

Algorithm 3 Oblique Trees from ReLU Networks (OTR)

```
Require: Neural Network's Weights $W$ and biases $B$, problem type $t$ (classification or regression)
Ensure: Oblique tree $T$.
1: Initialize $P$ as a set of matrices $P^1, P^2, \cdots, P^{m-1}$, where $P^i \in \mathbb{R}^{n_{i+1} \times n_1}$.
2: Initialize $V$ as a set of matrices $V^1, V^2, \cdots, V^{m-1}$, where $V^i \in \mathbb{R}^{n_{i+1} \times 1}$.
3: $P^1 \leftarrow W^1$, $V^1 \leftarrow B^1$
4: $r \leftarrow$ Empty-Node
5: Induce-Oblique-Decision-Tree($r$, $t$, 1, 1, $W$, $B$, $P$, $V$)
6: return $r$
```

Algorithms 3 and 4 show the pseudocode of OTR, which receives the weights and biases of a network R and the problem type t (either binary classification or regression); OTR returns an oblique decision tree that is equivalent to R. OTR recursively processes all neurons of layer i before processing neurons of layer $i + 1$. It starts with the root of the tree, which represents the first neuron of layer $j = 2$ ($l = 1$ and $k = 1$ in the pseudocode) and finishes with the output neuron.

As illustrated in our example, OTR rewrites the functions Z of the network in terms of the input values X, the resulting functions are denoted Z'. Similarly to how the Z-values are computed in terms of W and B, the Z'-values are computed in terms of matrices P and V. We define $Z'^i_k = P^{i-1}_k \cdot X + V^{i-1}_k$, where

Algorithm 4 INDUCE OBLIQUE DECISION TREE

```
Require: Node $r$, problem type $t$, layer $l$ and neuron $k$, matrices $W$, $B$, $P$, $V$.
 1: if $l > 1$ then
 2:     $P_k^l \leftarrow W_k^l \cdot P^{l-1}$
 3:     $V_k^l \leftarrow W_k^l \cdot V^{l-1} + B_k^l$
 4: if $l = m - 1$ then
 5:     if $t$ is classification then
 6:         $r \leftarrow (P_k^l, V_k^l, \text{left} = 0, \text{right} = 1)$ # represents output neuron for classification
 7:     if $t$ is regression then
 8:         $r \leftarrow (P_k^l, V_k^l)$ # represents output neuron for regression
 9:     return
10: $e \leftarrow$ EMPTY-NODE, $d \leftarrow$ EMPTY-NODE
11: $r \leftarrow (P_k^l, V_k^l, \text{left} = e, \text{right} = d)$ # represents $k$-th neuron of layer $l+1$
12: if $k = n_l$ then
13:     $k \leftarrow 1$
14:     $l \leftarrow l + 1$
15: else
16:     $k \leftarrow k + 1$
17: INDUCE-OBLIQUE-DECISION-TREE($d$, $t$, $l$, $k$, $W$, $B$, $P$, $V$)
18: $P_k^l \leftarrow \begin{bmatrix} 0 & \cdots & 0 \end{bmatrix}$, $V_k^l \leftarrow \begin{bmatrix} 0 \end{bmatrix}$
19: INDUCE-OBLIQUE-DECISION-TREE($e$, $t$, $l$, $k$, $W$, $B$, $P$, $V$)
```

P_k^{i-1} and V_k^{i-1} are the k-th row vector of P^{i-1} and V^{i-1}. Once the values of P_k^i and V_k^i are first computed (line 3 of Algorithm 3 or lines 2 and 3 of Algorithm 4), $Z_k'^{i+1} = Z_k^{i+1}$. However, the value $Z_k'^{i+1}$ of a node needs to be equal to A_k^{i+1}, so that the weights P^{i+1} and V^{i+1} of the next layer can be computed (in our example P_k^1 and V_k^1 are such that $A_k^{i+1} = Z_k'^{i+1} = P_k^i \cdot X + V_k^i$). In line 18 of Algorithm 4 the values of P_k^i and V_k^i are set to zero, so that the $Z_k'^{i+1}$ computed from the V and P matrices passed as parameters to the recursive calls in lines 19 and 17 are equal to A_k^{i+1}. The first recursive call treats the case where $A_k^{i+1} = Z_k^{i+1}$ and it recursively creates the right child of node r. The second call treats the case where $A_k^{i+1} = 0$ and it recursively creates the left child of r.

The matrices P^1 and V^1 are equal to W^1 and B^1 (line 3) because the functions Z^2 are defined in terms of X. Matrices P^l for $l > 1$ are computed in line 2 with the operation $W_k^l \cdot P^{l-1}$. This operation performs a weighted sum of the values p_j of the neurons q from the previous layer; the sum is weighted by the value in W representing the connection between the neuron k being processed with neuron q from the previous layer. In our example, this operation was used to compute $P^2 = W_1^2 \cdot P^1 = [1.2 \ \ -0.12]$. Similarly, V^l is computed with the operation $W_k^l \cdot V^{l-1} + B_k^l$, which is a weighted sum of the bias terms of the neurons from the previous layer, added to the bias term of the current neuron. In our example, we computed $V^2 = W_1^2 \cdot V^1 + B_1^2 = 2.84$. Once the k-th row of P^l and V^l are computed, we create the node representing the k-th neuron of layer $l+1$ with the parameters P_k^l and V_k^l (line 11).

For classification tasks, the left child of the node representing the output neuron returns the label 0, while its right child returns the label 1 (line 6). For regression tasks, the node representing the output neuron is a leaf and it returns the value of $P_1^{m-1} \cdot X + V_1^{m-1}$ as the prediction for X. As one can note, for classification tasks, the tree has one extra level if one considers the nodes with labels.

Theorem 1 *Let W and B be the weights and biases of a fully-connected neural network R whose hidden-layer units use ReLU activation functions and the single unit of its output layer uses either a Logistic or a linear activation function. The oblique decision tree T* OTR *induces with W and B is equivalent to R, i.e., T and R produce the same output for any input X.*

Proof. For a fixed input X, we prove that $Z_k'^i = A_k^i$, where $Z_k'^i = P_k^{i-1} \cdot X + V_k^{i-1}$ and $A_k^i = \text{ReLU}(Z_k^i)$ for all values of $Z_k'^i$ encountered along the inference path of X in T. If $Z_k'^i = A_k^i$ for any i and k, the Z'-value of the leaf node on the inference path matches the output of R for a fixed X. Thus, both T and R produce the same output for any fixed input X.

Our proof is by induction on the layer i. The base case considers $i = 2$, the model's first layer.

$$\begin{aligned} Z_k^2 &= W_k^1 \cdot X + B_k^1 \text{ (definition of } Z^2) \\ &= P_k^1 \cdot X + V_k^1 \text{ (line 3 of Algorithm 3)} \end{aligned}$$

During inference, if $P_k^1 \cdot X + V_k^1 \leq 0$, we follow the left child of the node with parameters P_k^1 and V_k^1. In this case, P_k^1 is set to a vector of zeros and V_k^1 is set to zero (line 18 of Algorithm 4), thus $P_k^1 \cdot X + V_k^1 = 0$. If $P_k^1 \cdot X + V_k^1 > 0$, then we follow the right child and $P_k^1 \cdot X + V_k^1 = W_k^1 \cdot X + B_k^1$. Therefore, $Z_k'^2 = A_k^2$, for any k. The inductive hypothesis assumes that $A_k^{i-1} = P_k^{i-2} \cdot X + V_k^{i-2}$.

For the inductive step we have the following.

$$\begin{aligned}
Z_k'^i &= P_k^{i-1} \cdot X + V_k^{i-1} && (3.1)\\
&= (W_k^{i-1} \cdot P_k^{i-2}) \cdot X + W_k^{i-1} \cdot V_k^{i-2} + B_k^{i-1} && (3.2)\\
&= W_k^{i-1}(P_k^{i-2} \cdot X + V_k^{i-2}) + B_k^{i-1} && (3.3)\\
&= W_k^{i-1} A_k^{i-1} + B_k^{i-1} && (3.4)\\
&= Z_k^i && (3.5)
\end{aligned}$$

Step 3.1 uses the definition of $Z_k'^i$, while step 3.2 is due to the computation in lines 2 and 3 of Algorithm 4. Step 3.4 uses the inductive hypothesis and step 3.5 the definition of Z_k^i.

We consider the two possible cases for Z_k^i:

1. $Z_k^i \leq 0$: OTR sets P_k^{i-1} and V_k^{i-1} to zeros (line 18 of Algorithm 4) so $Z_k'^i = Z_k^i = A_k^i = 0$.

2. $Z_k^i > 0$: we have from the derivation above that $Z_k'^i = Z_k^i = A_k^i$.

Thus, $Z_k'^i = A_k^i$.

The parameters P_1^{m-1} and V_1^{m-1} of leaf nodes are never set to zero because line 18 of Algorithm 4 is not reached for them. Therefore, $Z_1'^m = Z_1^m = P_1^{m-1} \cdot X + V_1^{m-1}$. In regression tasks, T returns the value $Z_1'^m = Z_1^m$ of the leaf node as its prediction (line 8). In classification tasks, the leaf node with label 0 is reached if $P_1^{m-1} \cdot X + V_1^{m-1} \leq 0$ for the node representing the output neuron because $g(z) \leq 0.5$ if and only if $z \leq 0$ for the Logistic function g; the leaf node with label 1 is reached otherwise. Therefore, T and R produce the same output for a fixed input X. □

3.2.1 Example

Let us consider the network R in Figure 3.1. OTR induces an oblique decision tree T that is equivalent to R (the value of the leaf node on the inference path for fixed values of x_1 and x_2 is equal to the output of R for the same inputs). Figure 3.1 shows the inference path (right-hand side) for $x_1 = 0.5$ and $x_2 = -0.5$ of the tree OTR induces from R. Each node on an inference path of T defines a function Z'^i_k of the input values that matches the value of Z^i_k of R for a fixed input (we omit the parameters of the functions to ease the notation). The Z'-functions of nodes representing neurons in layer $i = 2$ of R are equal to R's Z-functions because Z is already defined in terms of the input. Here, $Z'^2_1 = -2.7x_1 - 0.8x_2 - 0.4 = Z^2_1$, $Z'^2_2 = 0.2x_1 + 2.0x_2 - 0.6 = Z^2_2$, and $Z'^2_3 = 1.0x_1 - 0.1x_2 + 1.2 = Z^2_3$. The value of Z^3_1 is computed in R according to the output values of the neurons in layer $i = 2$. OTR defines $Z'^3_1 = Z^3_1$ in terms of x_1 and x_2 as follows.

We define matrices P^1 and V^1 where the k-th row of P^1 defines the weights of the k-th neuron of layer 2 in terms of x_1 and x_2. Similarly, the k-th entry of V^1 defines the bias term of the k-th neuron when the value the neuron produces is written in terms of x_1 and x_2. In this example, for fixed $x_1 = 0.5$ and $x_2 = -0.5$, P^1 and V^1 are W^1 and B^1 with the first two rows filled with zeros. This is because the first two neurons of layer $i = 2$ are inactive for x_1 and x_2. The weights of Z'^3_1 are given by P^2_1, where $P^2_1 = W^2_1 \cdot P^1$ and the bias term by V^2_1, where $V^2_1 = W^2_1 \cdot V^1 + B^2_1$ (Figure 3.1 shows the matrices P^2 and V^2). As can be verified, $Z'^3_1 = Z^3_1$ for $x_1 = 0.5$ and $x_2 = -0.5$. For the decision tree: $Z'^3_1 = 1.2 \cdot 0.5 - 0.12 \cdot (-0.5) + 2.84 = 3.5$. For the neural network:

$$Z^2 = \begin{bmatrix} -2.7 & -0.8 \\ 0.2 & 2.0 \\ 1.0 & -0.1 \end{bmatrix} \cdot \begin{bmatrix} 0.5 \\ -0.5 \end{bmatrix} + \begin{bmatrix} -0.4 \\ 0.6 \\ 1.2 \end{bmatrix} = \begin{bmatrix} -1.35 \\ -0.30 \\ 1.75 \end{bmatrix}, A^2 = \begin{bmatrix} 0 \\ 0 \\ 1.75 \end{bmatrix}$$

$$Z^3 = \begin{bmatrix} -2.0 & -2.4 & 1.2 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 0 \\ 1.75 \end{bmatrix} + \begin{bmatrix} 1.4 \end{bmatrix} = 3.5$$

If the output neuron uses a linear function, then the output of network $A^3_1 = Z^3_1 = Z'^3_1$; if it uses a Logistic function g, then T predicts class 1 because $Z'^3_1 = 3.5$ and $g(x) > 0.5$ if and only if $x > 0$ for the Logistic function; it would predict class 0 if $Z'^3_1 \leq 0$.

3.2.2 Extensions to OTR

Densely-connected Networks. OTR is also able to induce oblique decision trees that are equivalent to densely connected networks [17]. In densely connected networks, every neuron in layer i receives as input the output of layer $i-1$ and additionally the network input, X. The values of P_k^l and V_k^l must be appended to the matrices P^i and V^i for $i > l$ because the output of the k-th neuron of layer l is used as input in all the following layers.

Corollary 1 *Let W and B be the weights and biases of a densely connected neural network R whose hidden-layer units use ReLU activation functions and the single unit of its output layer uses either a Logistic or a linear activation function. The oblique decision tree T* OTR *induces with W and B is equivalent to R, i.e., T and R produce the same output for any input X.*

Leaky ReLU. OTR can be modified to handle Leaky ReLU functions: $\text{LReLU}(x) = \max(x, a \cdot x)$, where $0 < a < 1$. Similarly to ReLUs, the right child of a node handles the $A_k^i = Z_k^i$ case and the left child handles the $A_k^i = a \cdot Z_k^i$ case. Instead of setting the values of P_k^l and V_k^l to zero in line 18 of Algorithm 4, OTR assigns the values of $a(W_k^l \cdot P^{l-1})$ to P_k^l and $a(W_k^l \cdot V^{l-1} + B_k^l)$ to V_k^l.

Other Activation Functions for Output Layer OTR is compatible with ReLU networks with any activation function for the output layer. For regression problems, any output activation function g (e.g., identity function, Poisson link function, hyperbolic tangent) is suitable since it can be applied to the output of the induced tree. In the following two paragraphs, we discuss extensions to the binary classification problem.

Multi-Class Tasks. For multi-class tasks OTR can handle networks with multiple neurons in the output layer and Softmax functions. This is achieved by implementing, as part of the tree, a maximum function for the Z'-values of the output neurons. For example, a node s checks if $Z_1'^{m-1} - Z_2'^{m-1} \leq 0$ (is output 2 larger than output 1?) and s's left child checks $Z_2'^{m-1} - Z_3'^{m-1} \leq 0$ (is output 3 larger than output 2?) while its right child checks $Z_1'^{m-1} - Z_3'^{m-1} \leq 0$, and so on. For a classification problem with c classes, OTR maps a ReLU network with n hidden neurons to an oblique tree of depth $n + c - 1$.

Multi-Label Tasks. For multi-label classification tasks, we train one model for each label so our results are directly comparable with those of previous work [22].

Sparse Oblique Trees. Axis-aligned decision trees tend to be easier to interpret than oblique trees because each axis-aligned node considers a single feature (e.g., $x_i \leq b$). By contrast, each node in an oblique tree considers all features. In sparse oblique trees some of the weights p_i related to x_i are set to zero, thus increasing interpretability [12]. OTR allows for the induction of sparse trees if one uses L1-regularization while training the underlying neural network [42]. L1 regularization is effective in inducing sparse oblique trees with OTR if the network has a single hidden layer. Since L1 regularization is able to drive some of the weights of the model toward zero, nodes representing neurons in the model's first hidden layer will have some of its p_i-values also set to zero (because $P^1 = W^1$). Nodes representing neurons in layers $i > 2$ are less likely to be sparse as they depend on a combination of w-values being set to zero or adding up to zero in the operation $W_k^i \cdot P^{i-1}$.

3.2.3 Synthesizing Programmatic Policies with OTR

In the context of PIRL, OTR offers an effective way to learn programmatic policies. One must simply train a small ReLU network policy using any policy gradient algorithm, and map it to a small program encoded as an oblique decision tree. Previous methods that encode policies as oblique decision trees are much more complicated - they require search over a non-smooth program space or use a continuous approximation of the program space and then discretize the resulting program. Furthermore, due to its annealing schedule, LCN is not compatible for online learning problems like RL.

We note that other works (e.g., [3], [15], [22], [28], [50]) have established that ReLU neural networks produce piece-wise linear functions. However, to our knowledge, the LCN paper was the first to make the connection between ReLU networks and oblique decision trees. Compared to LCN, OTR offers a much simpler, fully differentiable training procedure at the expense of slightly more complex decision trees (i.e., in classification OTR trees require additional depth to account for the output layer, and in regression OTR's leaf nodes are linear models). Nonetheless, we are the first to make the connection to PIRL to obtain a simple and effective method for synthesizing programmatic policies.

As long as the DSL used to synthesize programmatic policies only accounts for if-then-else structures and affine transformations of the input values, OTR supersedes most previous PIRL methods because it strikes a good balance between simplicity and effectiveness, as we show in the empirical section of this thesis. Previous PIRL methods such as NDPS, Propel, and SketchSA are still needed in cases of more complex DSLs, such as those including loops. π-PRL is still the method of choice if the problem can be solved with small programs and one is interested in searching for both the program structure as well as the parameters of the program; the structure of the program OTR induces is defined by the architecture of the ReLU neural network encoding the policy.

3.3 Approximating ReLU Networks with Oblique Trees

3.3.1 Pruning OTR Trees

Recent work showed that ReLU networks realize surprisingly few activation patterns compared to the maximum possible [15]. Lee and Jaakkola [22] also observed this phenomenon in their LCN experiments. Activation patterns that are not encountered (or rarely encountered) on a sufficiently large data set can possibly be removed with little to no effect on the model's performance. For OTR trees this translates to pruning entire inference paths, which can significantly reduce the complexity of the learned model and improve its interpretability. We propose $\widehat{\text{OTR}}$, an approximation to OTR in which we prune branches that are not reached while executing the model in the environment for some number of episodes or on the training dataset. We also propose $\widehat{\text{OTR}}(k)$, where only the k most frequently encountered inference paths are kept and all others are removed. As we observe in our experiments, for some problems, training a larger OTR model and then approximating it using $\widehat{\text{OTR}}$ is easier than training an small OTR model directly.

3.3.2 Neurally Augmented Decision Trees

We present Neurally Augmented Decision Trees (AugTree), an imitation learning algorithm that augments the input space with the hidden units of the oracle, which is represented by a ReLU neural network. AugTree leverages the computationally efficient training procedures for axis-aligned tees (i.e., CART [8], linear model trees [19]) to implicitly construct oblique decision trees. In the PIRL setting, we use DAgger to guide the imitation learning search for the programmatic policy. Since the oracle's hidden units have been optimized for success in the given problem, we hypothesize that they contain structured pieces of useful information. This information is readily available by querying the oracle and may improve the capacity to approximate (i.e., imitate) the neural network, thus reducing the representation gap between the programmatic policy and the oracle.

Consider a dataset $\mathcal{D} = \{(X^{(t)}, y^{(t)})\}_{t=1}^{k}$ where each $X^{(t)} = [x_1, x_2, \cdots, x_d]^T$ and an oracle represented by a ReLU neural network with m layers and n_i neurons in each layer $i \in \{1, 2, \cdots, m\}$. Let Z_j^i denote the value of neuron j in the i^{th} layer before applying the ReLU activation function. AugTree uses the hidden units from the oracle's first hidden layer to augment the features from $X = [x_1, x_2, \cdots, x_d]^T$ to $\tilde{X} = [x_1, x_2, \cdots, x_d, Z_1^1, Z_2^1, \cdots, Z_{n_1}^1]^T$. Only the hidden units from the first hidden layer are considered since they are written in terms of the input features by default. Whenever the oracle is queried, Z^1 is calculated intermediately in the forward pass of the network, so there is no computational cost to access this information in an imitation learning setting. Augmenting the input space this way, we learn a mapping from $\tilde{X} to Y$. Like OTR, AugTree applies to both discrete and continuous output problem by learning a mapping. It follows that AugTree induces oblique decision trees since nodes are of the form $Z^i_j \leq c$ or $x_i \leq c$, with the latter being a specific case of an oblique decision node. The time complexity for training the decision tree using increases from $\mathcal{O}(dk\ log_2 k)$ to $\mathcal{O}((d + n_1)k\ log_2 k)$.

AugTree provides a very efficient way to induce oblique decision trees without actually needing to optimize the weights in the decision nodes. Since OTR proves a one-to-one correspondence between ReLU networks and oblique trees, AugTree can be seen as an approximation of ReLU networks. Similar to LCN, AugTree uses the hidden units from the underlying ReLU network in the decision nodes, and the leaf nodes are trained separately (LCN uses g_ϕ). OTR maps the entire requires a depth of $n_1 + \cdots + n_m + c - 1$. AugTree's maximal depth is set by the user so, the approximated tree can potentially map to shallower trees. For continuous output tasks, if AugTree uses Cart to train the model, the leaf nodes in the resulting oblique tree are constants, like in LCN. If one uses linear model trees, the result would be an oblique decision tree with linear models in the leaf nodes, like in OTR.

AugTree also offers advantages over $\widehat{\text{OTR}}$ and $\widehat{\text{OTR}}(k)$. The latter approaches can only prune branches of the OTR tree - there is no freedom to adjust the decision node parameters, leaf node parameters, or restructure the tree by changing the order of nodes. AugTree on the other hand, has more flexibility: it fits new bias terms in decision nodes, fits its own models in the leaf nodes, and freely decides the how to arrange the neurons in a decision tree. This is an important distinction since AugTree makes it possible to aggregate similar leaf nodes together in a more principled and automatic than what pruning allows. For example, a pruned OTR tree may reroute certain inputs X to leaf nodes that are very different to the one from the original inference path.

Variants to AugTree

There are several variants to AugTree that one might consider. One may use OTR to write the neurons from other layers in terms of input; however, this would require additional computational cost and the feature space would be augmented by $n_1 + 2^{n_2+\cdots+n_m}$ new features. Furthermore, one might prefer to learn a mapping from $\{Z_1^1, Z_2^1, \cdots, Z_{N_1}^1\}$ to the output. If $n_1 < d$ this is actually faster than training on X yet likely to result in a more expressive model. In addition, one could potentially augment the feature space with Z^1 from networks that use activation functions which are approximately linear (e.g., the hyperbolic tangent, sigmoid, LeakyReLU).

Chapter 4

Experimental Results

We performed experiments on reinforcement learning and supervised learning problems. In the RL setting, we considered both continuous and discrete action spaces for episodic MDPs. In the supervised learning setting, we consider both regression and classification problems. All reinforcement learning and supervised learning experiments for OTR were completed in approximately 504 hours and 2 hours, respectively, on single CPUs. Our RL models were implemented in the Stable Baselines3 repository [34], which is available under an MIT license. Our supervised learning models were implemented in Lee and Jaakkola's repository[1] [22].

4.1 RL for Continuous Action Spaces

We begin by training programmatic policies on eight continuous action control problems from OpenAI Gym [11] and MuJoCo [43]: Reacher, Walker2D, Hopper, HalfCheetah (HC), Ant, Swimmer, BipedalWalker (BW), and Pendulum. With the exception of Pendulum, the action spaces are multi-dimensional, so each action dimension has a linear model in the leaf nodes. We use OTR to train depth-six oblique decision tree policies so our results are comparable to those of π-PRL, which also uses depth-six trees. We also train oblique trees of depth thirty-two, and use $\widehat{\text{OTR}}$ to reduce their size to demonstrate that our methods can scale to produce much larger trees which can easily be simplified.

Since we are interested in finding a programmatic policy representation and

[1] https://github.com/guanghelee/iclr20-lcn

$$
\begin{aligned}
E &::= C \mid \textbf{if } B \textbf{ then } E \textbf{ else } E \\
B &::= P \cdot X + v \leq 0 \\
C &::= P \cdot X + v
\end{aligned}
$$

Figure 4.1: DSL for affine policies and regression ODTs.

not a programmatic value function, we use actor-critic methods to train small neural network policies and arbitrarily sized value networks, and apply OTR on the policy network only. We run PPO [38] for 3 million environment steps on BipedalWalker and Swimmer, SAC [14] for 3 million steps on Pendulum and Reacher, and SAC for 4 million steps on Walker2D, Hopper, HC, and Ant. We 'squeeze' the actions the model produces by using a hyperbolic tangent function in the leaf nodes for SAC policies.

We use hyperparameter values that are similar to the default values of known open-source implementations. For BW, we use a learning rate of 0.0003, minibatch size of 64, 2048 steps between updates, GAE parameter of 0.95, 10 epochs, clip factor of 0.2, and discount factor of 0.99, which are the default hyperparameters in Stable-Baselines3. For swimmer, we used a learning rate of 0.0002, minibatch size of 256, 1024 steps between updates, GAE parameter of 0.98, 10 epochs, clip factor of 0.2, and discount factor of 0.9999 - which are similar to the PPO parameters in Baselines3 Zoo [33]. For Pendulum, Reacher, HC, Hopper, Walker2d and Ant we use SAC with hyperparameters which are the default values for TD3 in Stable-Baselines3 and very similar to those used in π-PRL$\dot{\text{W}}$e used a learning rate of 0.001, minibatch size of 100, buffer size of 1000000, 10000 learning starts, and added Gaussian noise $\mathcal{N}(0, 0.1)$ to actions. We use ReLU activation functions and critic networks with two hidden layers of size 256 each on all domains except Walker2d, for which we used the LeakyReLU.

We use NDPS [46], π-PRL [32] and VIPER [4], and a modified version of VIPER that uses linear model trees [19] (LM-VIPER) as baselines. NDPS and π-PRL use the language from Figure 4.5, VIPER and LM-VIPER produce axis-aligned trees. We use NDPS and not PROPEL [45] because previous work noted that the former performs better than the latter on the OpenAI Gym and Mujoco domains [32]. Recall that Viper LM-Viper and NDPS are imitation learning methods whereas π-PRL and OTR use policy gradient methods to learn a policy directly. For NDPS, instead of searching for an oblique decision tree policies via program synthesis, which is computationally expensive, we use π-PRL (OTR works too) to imitate the oracle using DAgger.

We perform ten independent runs of each system, except NDPS and π-PRLwhich are run three times, and evaluate the best policy from each run for 100 consecutive episodes. The best policy was found during training by evaluating the policy every $10,000$ steps for 10 episodes. Interpretable models are typically static so that they can be reasoned before being put into production. As such, one should deploy the best suited model, so evaluating the best policy is a reasonable metric. Furthermore, the imitation learning baselines we consider also return the best policy by design.

The mean and standard deviation across the runs are reported in Table 4.1. The best average result for each domain is highlighted in bold. OTR is capable of solving a variety of problems using small actor networks, and outperforms oracle-guided approaches (Viper, LM-Viper, and NDPS) by a large margin in Walker2D, Hopper, HC, Swimmer, and Pedulum, and is overall competitive with π-PRL. OTR has better average reward than π-PRL in five of the eight domains tested, but π-PRL is a close competitor in all eight domains. π-PRL outperforms OTR in HC and Ant. In contrast with π-PRL, OTR is able to scale and train longer programmatic policies. $\widehat{\text{OTR}}$ policies trained with 32 neurons perform well across all domains with the exception of Swimmer. The pruned tree depths range from 8 to 12, which is a major reduction in size from the original depth of 32. These results show that OTR can synthesize longer and more powerful policies, at the cost of possibly having less interpretable policies.

Example of an Interpretable Policy

Figure 4.2 shows an OTR policy trained with a single hidden neuron on the mountain car problem. The goal of mountain car is to reach the top of the

Table 4.1: Reward and standard deviation of depth-6 policies over 100 episodes, averaged over ten (three for NDPS and π-PRL) independent runs of each algorithm. The last column shows the performance for depth-32 $\widehat{\text{OTR}}$ policies. In brackets is the estimated pruned depth, $\log_2 a$ (a is the number of activation patterns realized). The best average for each domain is highlighted in bold.

Environment	Viper	LM-Viper	NDPS	π-PRL	OTR	$\widehat{\text{OTR}}$	
Reacher	-6.3 ± 0.3	-4.3 ± 0.2	-5.8 ± 0.1	-5.1 ± 0.1	-4.9 ± 0.4	$\mathbf{-4.0 \pm 0.1}$	(9.0)
Walker2d	710.9 ± 186.7	2847.2 ± 1168.8	3671.7 ± 1196.2	5178.0 ± 16.3	4829.4 ± 338.9	$\mathbf{5339.4 \pm 454.8}$	(9.5)
Hopper	618.6 ± 196.8	2139.2 ± 896.5	1646.3 ± 588.2	3535.3 ± 23.3	$\mathbf{3641.5 \pm 126.9}$	3300.0 ± 703.1	(9.6)
HC	1764.2 ± 992.2	3429.4 ± 1477.0	3569.3 ± 50.2	$\mathbf{10772.7 \pm 60.2}$	7317.3 ± 683.7	10307.9 ± 1320.6	(11.2)
Ant	2080.3 ± 637.9	4379.8 ± 1052.4	4874.7 ± 188.9	$\mathbf{5679.7 \pm 844.3}$	3504.7 ± 904.7	5392.5 ± 875.6	(10.6)
Swimmer	335.1 ± 62.9	333.7 ± 65.0	334.7 ± 0.9	340.3 ± 31.4	$\mathbf{365.9 \pm 1.1}$	340.7 ± 62.4	(7.9)
BW	203.9 ± 52.5	278.4 ± 21.8	273.0 ± 12.3	274.3 ± 18.3	$\mathbf{294.0 \pm 12.6}$	284.8 ± 21.0	(12.6)
Pendulum	-196.3 ± 21.8	-267.0 ± 47.7	-146.7 ± 4.5	-144.7 ± 3.3	-138.3 ± 1.0	$\mathbf{-135.0 \pm 1.2}$	(8.4)

mountain on located to the right. The state is defined by the car's position, x, and its velocity, v_x. The one-dimensional continuous action is a real number in [-1, 1] and represents the directional force applied to the car. The car starts near the bottom of a valley, and cannot reach the goal by simply moving to the right: it must first gain enough potential energy by moving to the left before moving to the right. We interpret the program as follows. We notice that the velocity term dominates the other terms in the boolean expression, so we simplify the node to $v_x \leq 0$. Similarly, the velocity term dominates the equation in the "False", and so $\max(5117v_x, 1.0) \approx 1.0$. As such, we can simplify the policy and interpret it as follows: if the car is moving to the right, keep moving to the right. Otherwise, move to the left.

```
if $-2.2 - 3.8x + 114.3v_x \leq 0$ then
  return $-6.1$
else
  return $-102.3 - 169.8x + 5116.5v_x$
```

$\Longrightarrow$

```
if $v_x \leq 0$ then
  return $-1.0$
else
  return $1.0$
```

Figure 4.2: Original (left) and simplified (right) OTR programmatic policies for Mountain Car Continuous. These policies achieve an average reward of 92.8 and 92.2, respectively (90 is considered solved).

4.1.1 PID Controller Policies

Proportional-integral-derivative (PID) controllers have long been used to stabilize control systems due to their robustness and stability guarantees. More recently, discretized PID controllers have been used in programatically interpretable RL [32], [45], [46] where the proportional (P), integral (I), and derivative (D) are approximated as follows:

$$P = (\epsilon - s), \qquad I = \textbf{fold}(+, \epsilon - h), \qquad D = \textbf{peek}(h, -1) - s$$

where $\mathbf{s} \in \mathbb{R}^d$ is the current state of the environment, $\epsilon \in \mathbb{R}^d$ is the known fixed target for which the system is stable, h is a history of the previous k states (we use $k = 5$), **fold** is a higher-order function which sums the input sequence along its dimension, and **peek**(h, -1) returns the last state in h.

This DSL for PID policies (Figure 4.3) uses linear combinations of features in the decision nodes. As ReLU neural networks are capable of tuning PID parameters, denoted θ_P, θ_I and θ_D, we can apply OTR to induce a PID controller policy. However, terminal nodes of OTR programs are linear functions of the input (e.g., $\theta_P = \theta_P(s) = M \cdot X + v$), which is more expressive than the given DSL in other works where the PID parameters are constants. One could perhaps use LCN since the leaf nodes are constant instead of linear. We instead use aggressive L1 regularization on the network's output weights so that the PID parameters are as close as possible to the common DSL.

$$\begin{aligned} E &::= C \mid \textbf{if } B \textbf{ then } E \textbf{ else } E \\ B &::= M \cdot X + v \leq 0 \\ C &::= \theta_P \cdot P + \theta_I \cdot I + \theta_D \cdot D \end{aligned}$$

Figure 4.3: DSL for PID Controller Policies

We use OTR to train a PID policy on the pendulum problem from OpenAI Gym. The state is given by the vector $\mathbf{s} = [x = cos(\omega), y = sin(\omega), \dot{\omega}]$, where ω is the angle relative to the upright position and $\dot{\omega}$ is the angular velocity. The objective is to balance the pendulum upright by applying a leftward or rightward torque, so the stable point is $\epsilon = [1, 0, 0]$. We use DDPG [23] to train the PID controller, and then we apply OTR on the actor network to induce the PID controller policy. The actor network consists of a single hidden neuron and nine output neurons, three for each θ_P, θ_I and θ_D. We used L1 regularization with $\alpha = 2.5$ for the weights in the output layer.

Figure 4.4 shows the resulting policy, which achieves an average reward of -162.6 over 1000 consecutive episodes. Notice that the L1 regularization is effective in eliminating the state dependence on the θ_P and θ_D parameters. Verma *et al.* report scores of -187.7 for PROPEL and -435.7 for NDPS.

```
**if** $7.78 - 15.7x - 21.7y - 8.48\dot{\omega} \leq 0$ **then**
    **return** $[3.15,\ 3.24,\ 0.65] \cdot P + [0.12, 0.52, 0.04] \cdot I + [10.94, 11.89, 0.13] \cdot D$
**else**
    **return** $[3.15,\ 3.24,\ 0.65] \cdot P + \theta_I \cdot I + [10.94, 11.89, 0.13] \cdot D$
```

where $\theta_I = [-0.25 + 0.75x + 1.04y + 0.41\dot{\omega}, 0.52, 0.10 - 0.13x - 0.17y - 0.07\dot{\omega}]$

Figure 4.4: OTR PID controller policy for Pendulum. This policy achieves an average reward of -162.6 over 1000 consecutive episodes.

4.2 RL with Discrete Action Spaces

$$\begin{aligned} E &::= C \mid \textbf{if } B \textbf{ then } E \textbf{ else } E \\ B &::= P \cdot X + v \leq 0 \\ C &::= a \in \mathcal{A} \end{aligned}$$

Figure 4.5: DSL for discrete output oblique decision trees.

We evaluate our algorithms for inducing oblique decision tree policies on lunar lander, a discrete action environment from OpenAI Gym. The goal of the lunar lander task is to safely land the space ship on a landing pad by taking one of four actions: do nothing (0), activate the left engine (1), activate the main engine (2), or activate the right engine (3). The state is given by an eight-dimensional vector consisting of six continuous variables representing the horizontal coordinate x, vertical coordinate y, horizontal velocity v_x, vertical velocity v_y, angle θ, and angular velocity v_θ, and two binary variables c_L and c_R which equal 1 if the left and right legs have contact, respectively. The problem is considered solved after achieving an average return above 200 over 100 consecutive episodes.

We trained ReLU actor networks with two hidden units using PPO for 2 million time steps and mapped the policy to an oblique tree using OTR. Averaged over five independent trials, the resulting depth-five [2] oblique decision tree programs achieved a return of 269.0 ± 16.6 over 100 episodes. Differentiable decision tree approaches to PIRL such as SDT and CDT [39] were unable to solve this problem using policy gradient methods. Before and after discretizing the depth-five trees, SDT achieved an average return of 97.8 ± 10.5 and -88.0 ± 20.4, respectively. Similarly, CDT achieved a score of approximately 100 for trees with depths ranging from four and six.

For this reason we only use VIPER (axis-aligned trees) as a baseline and compare it against AUGTREE, which is also an oracle-guided approach. We use DAGGER with fifty rollouts and twenty-five episodes per rollout for both methods. Furthermore, we assess the robustness of AUGTREE by augmenting the policy space with layer-one hidden units from oracle networks of various architectures, denoted $\mathcal{N}$, ranging from small and shallow to large and deep. Since the hidden units from the critic network are not used, we fix its architecture to two hidden layers of size 128. Figure 4.6 shows the performance of both methods (averaged over fifteen independent trials) for various tree depths when guided by different oracles.

AUGTREE was able to solve the problem with trees of depth two, which is the minimum depth to account for all four actions. With depth-three trees, AUGTREE was on par with the oracle's performance. These results were consistent among all four oracle networks architectures. Interestingly, augmenting the input space with the layer-one hidden units from deep, multi-layer oracles networks was enough for this problem. VIPER required trees of depth eight to solve the problem, and depth-ten trees to perform similarly to the oracle. It is clear that axis-aligned trees require much deeper trees than their oblique counterparts, which can actually hinder interpretability according to certain metrics [13].

[2]2 hidden units + 4 units in output layer - 1 = 5

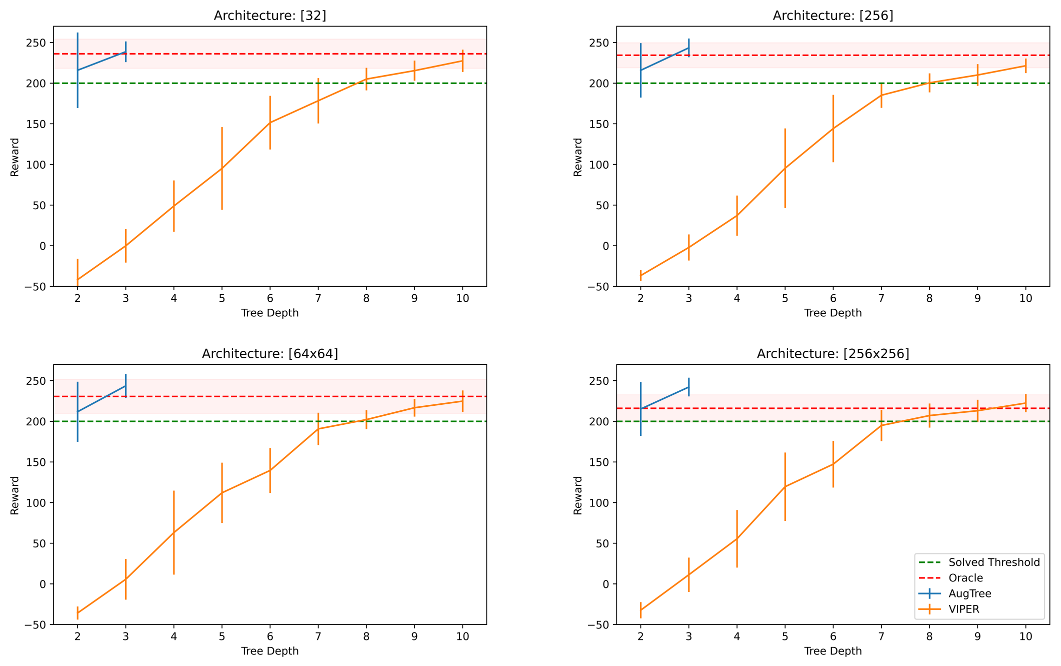


Figure 4.6: Performance of AugTree (blue), Viper (orange) and NDPS(red) for various tree depths on Lunar Lander. Average return and standard deviation across fifteen runs.

In Figure 4.7 we present a policy learned using AugTree. The policy was simplified a posteriori by retraining the left and right decision nodes using linear discriminant analysis with L1 regularization. The original AugTree policy achieves an average reward of 249.1 while the simplified policy achieves an average reward of 246.8. The decision node of left child can be interpreted as follows: if the ship's downward velocity is more than half of its vertical position, fire the main engine. Otherwise, fire the the left engine. The right decision node turns off the engine if either leg has contact so that the ship can come to rest. If neither leg has contact, the right engine is activated. The actions to fire the left and right engines are on different branches of this small tree, so the root node perhaps considers the craft's horizontal position.

```
if $-0.09x + 0.12y - 0.12v_x + 0.27v_y + 0.22\omega + 0.18\dot{\omega} + 0.22c_L + 0.20c_R \leq 0$ then
  if $0.45y + v_y \leq -0.02$ then
    return 2
  else
    return 1
else
  if $c_L + c_R > 0$ then
    return 0
  else
    return 3
```

Figure 4.7: A simplified AugTree policy for lunar lander. This policy averages a return of 246.8.

4.3 Supervised Learning

In order to compare OTR with LCN, we evaluate OTR on the five supervised learning problems from the MoleculeNet benchmark [49] LCN was evaluated on: Bace, HIV, SIDER, Tox21, PDBbind. Each instance in all five problems are described by 2,048 binary features. The first four tasks are classification tasks and the last one a regression task. Bace and HIV are binary classification tasks, SIDER has 27 binary outputs and Tox21 has 12 binary outputs. Our empirical methodology for supervised learning tasks is identical to that used in the LCN original paper [22], including the training, validation, and test splits, so that our results are directly comparable to theirs. We present the performance of the classifiers in terms of area under the curve (AUC) for classification tasks and root mean squared error (RMSE) for the regression task. We train one model for each label in the multi-label tasks (like LCN does) and present the average results for all labels.

We use as baselines the axis-aligned trees Cart induces [7] as well as the oblique trees HHCART [48], TAO [12], LCN [22] induce. The TAO authors did not consider regression trees in their study, so we do not use TAO on the PDBbind domain. The LCN authors also introduced variants of the LCN model, which we do not include in our experiments because they were not proven to be equivalent to oblique trees. While all algorithms tested might perform better if used in an ensemble [9], [10], we focus on evaluating single-tree

Table 4.2: Performance of different methods for training oblique trees on supervised tasks. The best average is highlighted in bold.

Algorithm	Bace (AUC)	HIV (AUC)	SIDER (AUC)	Tox21 (AUC)	PDBbind (RMSE)
Cart	0.652 ± 0.024	0.544 ± 0.009	0.570 ± 0.010	0.651 ± 0.005	1.573 ± 0.000
Hhcart	0.545 ± 0.016	0.636 ± 0.000	0.570 ± 0.009	0.638 ± 0.007	1.530 ± 0.000
Tao	0.734 ± 0.000	0.627 ± 0.000	0.577 ± 0.004	0.676 ± 0.003	-
Lcn	**0.839 ± 0.013**	0.728 ± 0.013	0.624 ± 0.044	0.781 ± 0.017	1.508 ± 0.017
OTR	0.815 ± 0.017	**0.741 ± 0.019**	**0.656 ± 0.088**	**0.796 ± 0.071**	**1.467 ± 0.030**

models because ensembles are considered black boxes [47].

We trained fully-connected neural networks with two hidden layers with 8 neurons in each layer. The output layer contained either one neuron with a linear function for the regression task or one neuron with a Logistic function for the classification tasks. For multi-class tasks we trained one model for each label, so we are consistent with previous work [22]. All our models had approximately 16K parameters for all domains. We followed the procedure used in the original LCN experiments where the architectures of the densely connected models were tuned for each domain [17]. Namely, the number of parameters of the LCN models range from 43K (HIV) to 5.8M (PDBbind).

Table 4.2 presents the results. OTR's oblique trees are far superior to the axis-aligned trees Cart induces, which is expected as oblique trees tend to encode better models than axis-aligned trees at the cost of reduced interpretability. Both LCN and OTR, the two neural-based methods, substantially outperform the oblique trees HHCART and TAO induce, in all domains evaluated. Despite using a simpler training procedure (equivalent to that of training a fully-connected ReLU neural network) and models with many fewer trainable parameters, OTR is competitive with LCN in all domains tested.

Training Sparse Models. We also trained two neural networks with a single hidden layer with 16 neurons for the Bace problem. For one of the models we used L1 regularization with a coefficient of 0.00001; the other model did not use regularization. Both models were trained for 100 epochs and learning rate of 0.1. Both regularized and non-regularized models achieved an average AUC value of 0.814 over 10 seeds. The regularized model had approximately 40% of its weights smaller than 10^{-5} whereas the non-regularized models had fewer than 1% of weights smaller than 10^{-5}. If these small weights can be ignored, the regularized model may significantly improve the interpretability of the learned model.

Chapter 5

Conclusion

In this dissertation, we presented an algorithm for mapping neural networks trained with back-propagation that use ReLU activation functions to oblique decision trees. Furthermore, we showed how hidden units from ReLU networks can be used to augment input spaces and implicitly train oblique decision trees. Both of our methods require significantly less computational cost than other methods to induce oblique decision trees. Oblique decision trees can provide concise and interpretable solutions to complex problems. Our work is applicable to a variety of tasks including supervised learning (regression and classification), reinforcement learning (discrete and continuous control), and PID controller tuning. We focused on PIRL since current approaches for inducing interpretable programs are quite complex. The experimental results demonstrate that OTR and AugTree are competitive with many current approaches to programmatically interpretable reinforcement learning while being significantly more scalable and easier to implement.

It would be interesting to couple AugTree with PropeLas this may bridge the representation gap between neural networks and programmatic policies even further. Augmenting the DSL by adding the oracle's hidden units as function could be used to synthesize more powerful programs that include loops, for example. Furthermore, using AugTree in the imitation learning phase of Sketch-SA and then relying on program synthesis for the direct policy search phase would improve the run time and perhaps the quality of the program. We can also use our algorithms to augment DSLs with the hidden units. Finally, extending LCN to policy gradient methods could be useful for learning oblique decision tree programs with constant leaf nodes (e.g., the DSL for PID controllers in Chapter 4). However, this approach remains in question as LCN cannot leverage efficient gradient computation methods such as backpropagation and requires an annealing schedule that .

References

[1] A. Albarghouthi, S. Gulwani, and Z. Kincaid, "Recursive program synthesis," in *CAV*, 2013.

[2] G. Anderson, A. Verma, I. Dillig, and S. Chaudhuri, *Neurosymbolic reinforcement learning with formally verified exploration*, 2020. DOI: 10.48550/ARXIV.2009.12612. [Online]. Available: https://arxiv.org/abs/2009.12612.

[3] R. Balestriero and R. G. Baraniuk, "Mad max: Affine spline insights into deep learning," *Proceedings of the IEEE*, vol. 109, no. 5, pp. 704–727, 2021. DOI: 10.1109/JPROC.2020.3042100.

[4] O. Bastani, Y. Pu, and A. Solar-Lezama, "Verifiable reinforcement learning via policy extraction," in *Proceedings of the International Conference on Neural Information Processing Systems*, Curran Associates Inc., 2018, pp. 2499–2509.

[5] K. P. Bennett, "Global tree optimization: A non-greedy decision tree algorithm," in *Computing Science and Statistics*, 1994, pp. 156–160.

[6] D. Bertsimas and J. Dunn, "Optimal classification trees," *Machine Learning*, vol. 106, no. 7, pp. 1039–1082, 2017.

[7] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and Regression Trees*. Monterey, CA: Wadsworth and Brooks, 1984.

[8] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, "Classification and regression trees," 1983.

[9] L. Breiman, "Bagging predictors," *Machine Learning*, vol. 24, no. 2, pp. 123–140, 1996. DOI: 10.1007/BF00058655.

[10] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001, ISSN: 0885-6125.

[11] G. Brockman, V. Cheung, L. Pettersson, *et al.*, *Openai gym*, 2016. eprint: arXiv:1606.01540.

[12] M. A. Carreira-Perpiñán and P. Tavallali, "Alternating optimization of decision trees, with application to learning sparse oblique trees," in *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., vol. 31, Curran Associates, Inc., 2018.

[13] L. L. Custode and G. Iacca, "Evolutionary learning of interpretable decision trees," *ArXiv*, vol. abs/2012.07723, 2020.

[14] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," 2017.

[15] B. Hanin and D. Rolnick, "Deep relu networks have surprisingly few activation patterns," *ArXiv*, vol. abs/1906.00904, 2019.

[16] D. G. Heath, "A geometric framework for machine learning," UMI Order No. GAX93-13375, Ph.D. dissertation, USA, 1993.

[17] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 2261–2269. DOI: `10.1109/CVPR.2017.243`.

[18] J. P. Inala, O. Bastani, Z. Tavares, and A. Solar-Lezama, "Synthesizing programmatic policies that inductively generalize," in *International Conference on Learning Representations*, 2020. [Online]. Available: `https://openreview.net/forum?id=S1l8oANFDH`.

[19] A. Karalič, "Employing linear regression in regression tree leaves," in *Proceedings of the 10th European Conference on Artificial Intelligence*, ser. ECAI '92, Vienna, Austria: John Wiley amp; Sons, Inc., 1992, pp. 440–441, ISBN: 0471936081.

[20] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983, ISSN: 00368075, 10959203. [Online]. Available: `http://www.jstor.org/stable/1690046` (visited on 11/07/2022).

[21] G.-H. Lee, D. Alvarez-Melis, and T. S. Jaakkola, *Towards robust, locally linear deep networks*, 2019. DOI: `10.48550/ARXIV.1907.03207`. [Online]. Available: `https://arxiv.org/abs/1907.03207`.

[22] G.-H. Lee and T. S. Jaakkola, "Oblique decision trees from derivatives of relu networks," in *International Conference on Learning Representations*, 2020. [Online]. Available: `https://openreview.net/forum?id=Bke8UR4FPB`.

[23] T. P. Lillicrap, J. J. Hunt, A. Pritzel, *et al.*, *Continuous control with deep reinforcement learning*, 2015. DOI: `10.48550/ARXIV.1509.02971`. [Online]. Available: `https://arxiv.org/abs/1509.02971`.

[24] J. Lin, C. Zhong, D. Hu, C. Rudin, and M. Seltzer, "Generalized and scalable optimal sparse decision trees," in *International Conference on Machine Learning*, 2020.

[25] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," in *ICML Workshop on Deep Learning for Audio, Speech and Language Processing*, 2013.

[26] L. C. Medeiros, D. S. Aleixo, and L. H. S. Lelis, "What can we learn even from the weakest? Learning sketches for programmatic strategies," in *Proceedings of the AAAI Conference on Artificial Intelligence*, AAAI Press, 2022.

[27] J. Mockus, V. Tiesis, and A. Zilinskas, "The application of bayesian methods for seeking the extremum," in Sep. 2014, vol. 2, pp. 117–129, ISBN: 0-444-85171-2.

[28] G. Montúfar, R. Pascanu, K. Cho, and Y. Bengio, "On the number of linear regions of deep neural networks," in *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'14, Montreal, Canada: MIT Press, 2014, pp. 2924–2932.

[29] S. K. Murthy, S. Kasif, and S. Salzberg, "A system for induction of oblique decision trees," *Journal of Artificial Intelligence Research*, vol. 2, no. 1, pp. 1–32, 1994, ISSN: 1076-9757.

[30] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the International Conference on International Conference on Machine Learning*, 2010, pp. 807–814.

[31] OpenAI, : C. Berner, *et al.*, *Dota 2 with large scale deep reinforcement learning*, 2019. DOI: 10.48550/ARXIV.1912.06680. [Online]. Available: https://arxiv.org/abs/1912.06680.

[32] W. Qiu and H. Zhu, "Programmatic reinforcement learning without oracles," in *International Conference on Learning Representations*, 2022. [Online]. Available: https://openreview.net/forum?id=6Tk2noBdvxt.

[33] A. Raffin, *RL Baselines3 Zoo*, https://github.com/DLR-RM/rl-baselines3-zoo, 2020.

[34] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, "Stable-baselines3: Reliable reinforcement learning implementations," *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021. [Online]. Available: http://jmlr.org/papers/v22/20-1364.html.

[35] M. Raghu, B. Poole, J. Kleinberg, S. Ganguli, and J. S. Dickstein, "On the expressive power of deep neural networks," in *Proceedings of the International Conference on Machine Learning*, 2017, pp. 2847–2854.

[36] S. Ross, G. J. Gordon, and J. A. Bagnell, "A reduction of imitation learning and structured prediction to no-regret online learning," in *AISTATS*, 2011.

[37] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning Representations by Back-propagating Errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986. DOI: 10.1038/323533a0. [Online]. Available: http://www.nature.com/articles/323533a0.

[38] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, *Proximal policy optimization algorithms*, 2017. DOI: 10.48550/ARXIV.1707.06347. [Online]. Available: https://arxiv.org/abs/1707.06347.

[39] A. Silva, M. C. Gombolay, T. W. Killian, I. D. J. Jimenez, and S.-H. Son, "Optimization methods for interpretable differentiable decision trees applied to reinforcement learning," in *AISTATS*, 2020.

[40] D. Silver, T. Hubert, J. Schrittwieser, *et al.*, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *CoRR*, vol. abs/1712.01815, 2017. arXiv: 1712.01815. [Online]. Available: http://arxiv.org/abs/1712.01815.

[41] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT Press, 1998, ISBN: 0-262-19398-1. [Online]. Available: http://www.cs.ualberta.ca/%7Esutton/book/ebook/the-book.html.

[42] R. Tibshirani, "Regression shrinkage and selection via the lasso," *Journal of the Royal Statistical Society*, vol. 58, pp. 267–288, 1996.

[43] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, IEEE, 2012, pp. 5026–5033.

[44] P. E. Utgoff and C. E. Brodley, "Linear machine decision trees," Department of Computer Science, University of Massachusetts, Amherst, Massachusetts, 01003, USA, Tech. Rep., 1991.

[45] A. Verma, H. M. Le, Y. Yue, and S. Chaudhuri, "Imitation-projected programmatic reinforcement learning," in *Proceedings of the International Conference on Neural Information Processing Systems*, Curran Associates Inc., 2019.

[46] A. Verma, V. Murali, R. Singh, P. Kohli, and S. Chaudhuri, "Programmatically interpretable reinforcement learning," *CoRR*, vol. abs/1804.02477, 2018. arXiv: 1804.02477. [Online]. Available: http://arxiv.org/abs/1804.02477.

[47] C. Wang, B. Han, B. Patel, and C. Rudin, "In pursuit of interpretable, fair and accurate machine learning for criminal recidivism prediction," *Journal of Quantitative Criminology*, 2022. DOI: 10.1007/s10940-022-09545-w.

[48] D. Wickramarachchi, B. Robertson, M. Reale, C. Price, and J. Brown, "Hhcart: An oblique decision tree," *Computational Statistics Data Analysis*, vol. 96, pp. 12–23, 2016, ISSN: 0167-9473. DOI: https://doi.org/10.1016/j.csda.2015.11.006. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167947315002856.

[49] Z. Wu, B. Ramsundar, E. Feinberg, *et al.*, "Moleculenet: A benchmark for molecular machine learning," *Chemical Science*, vol. 9, Mar. 2017. DOI: 10.1039/C7SC02664A.

[50] L. Zhang, G. Naitzat, and L.-H. Lim, “Tropical geometry of deep neural networks,” in *International Conference on Machine Learning*, PMLR, 2018, pp. 5824–5832.

[51] H. Zhu, Z. Xiong, S. Magill, and S. Jagannathan, “An inductive synthesis framework for verifiable reinforcement learning,” in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM, Jun. 2019. DOI: 10.1145/3314221.3314638. [Online]. Available: https://doi.org/10.1145%2F3314221.3314638.