Alchemist: Learning Guarded Affine Functions

Shambwaditya Saha, Pranav Garg, and P. Madhusudan

University of Illinois, Urbana-Champaign

Abstract. We present a technique and an accompanying tool that learns guarded affine functions. In our setting, a teacher starts with a guarded affine function and the learner learns this function using equivalence queries only. In each round, the teacher examines the current hypothesis of the learner and gives a counter-example in terms of an input-output pair where the hypothesis differs from the target function. The learner uses these input-output pairs to learn the guarded affine expression. This problem is relevant in synthesis domains where we are trying to synthesize guarded affine functions that have particular properties, provided we can build a teacher who can answer using such counter-examples. We implement our approach and show that our learner is effective in learning guarded affine expressions, and more effective than general-purpose synthesis techniques.

1 Introduction

We consider the problem of learning guarded affine functions, where the function is expressed using linear inequality guards that delineate regions, and where in each region, the function is expressed using an affine function. More precisely, guarded affine functions are those expressible as guarded linear expressions, which are given by the following grammar:

$$\begin{aligned} gle &:= ite(lp, gle, gle) \mid le \\ lp &:= le < le \ \mid \ le \leq le \ \mid \ le = le \\ le &:= c \mid cx \mid le + le \end{aligned}$$

where x ranges over a fixed finite set of variables V with domain D (which can be reals, rationals, integers or natural numbers), and where c ranges over rationals.

We are interested in the problem of learning guarded affine functions using only a sample of its behavior on a finite set of points. More precisely, consider the learning setting where we have a teacher that has a target guarded affine function $f : \mathbb{R}^d \longrightarrow \mathbb{R}$. We start with the behavior on a finite set of samples $S \subseteq \mathbb{R}^d$, and the teacher gives the value of the function on these points. The learner then must respond with a guarded linear expression hypothesis H that is consistent with these points. The teacher then examines H and checks if the learner has learned the target function (i.e., checks whether $H \equiv f$). If yes, we are done; otherwise, the teacher *adds* a new sample $s \in \mathbb{R}^d$ where they differ (i.e., $H(s) \neq f(s)$), and we iterate the learning process with the learner coming up with a new hypothesis. The goal of the learner is to learn the target guarded affine function f.

The above model can be seen as a learning model with *equivalence queries* only, or an *online learning model* [3]. This learning model is motivated by synthesis problems, where the idea is to synthesize functions that satisfy certain properties. For instance, in an effort to help a programmer identify a guarded affine function within a program, we can consider the program with this *hole*, capture the correctness requirement (perhaps expressed using input-output examples to the program) and then build a teacher who can check correctness of hypothesized expressions [10]. Combined with the learner that we build, we will then obtain synthesis algorithms for this problem.

We do emphasize, however, that the problem and solution we consider here works only if there is an effective teacher who knows the target concept. In some synthesis contexts, where the specification admits *many* acceptable guarded affine functions as solutions, we would have to use heuristics to use our approach (for example, the teacher may decide how the function behaves on some inputs, from the class of possible outputs, to instruct the learner). However, as a learning problem, our problem formulation is simple and clean.

The black-box learning approach to synthesis is in fact very common in synthesis solvers. For instance, the CEGIS (counter-example guided inductive synthesis) approach [11] is similar to learning in the sense that it too synthesizes from samples, and solvers in the SyGuS format for synthesis, including the enumerative solver, the stochastic solver, the symbolic solver, and Sketch [10], are based on synthesizing using concrete valuations of variables in the specification [1].

There has also been previous work on the construction of piece-wise affine models of hybrid dynamical systems from input-output examples [2,4,5,12] (also see [7] for an extensive survey of the existing techniques). The problem setting, in these works, attacks a different problem, where the goal is to learn guarded linear functions that *approximate* the sample. Consequently, the work in [2] uses techniques such as regression, while we use accurate algorithms. Also, in our setting, we have an active learning setup where the learner keeps learning using counterexamples till the process converges to the target function.

Contribution: In this paper, we build a learning algorithm for guarded affine functions, that learns from a sample of points and how the target function evaluates on that sample: $\{(\boldsymbol{x}_i, f(\boldsymbol{x}_i))\}_{i=1,\ldots,n}$. Our goal is to learn a *simple* guarded linear expression that is consistent with this sample (the learning bias is towards simpler expressions). A guarded linear expression can be thought of as a nested if-then-else expression with linear guards, and linear expressions at its leaves. Our algorithm is composed of two distinct phases:

- Phase 1: [Geometry] First, we synthesize the set of linear expressions that will occur at the leaves of our guarded linear expression. This problem is to essentially find a small set of planes that include all the given points in the $(d + 1)$-dimensional space (viewing the space describing the inputs and the output of the function being synthesized), and we use a greedy algorithm that uses computational geometry techniques for achieving this. At the end of this phase, we have *labeled* every point in the sample with a plane that correctly gives the output for that point.

– Phase 2: [Classifier] Given the linear expressions from the first phase, we then synthesize the guards. This is done using a classification algorithm, which decides how to assign all points in $\mathbb{R}^d$ to planes such that all the samples get mapped to the correct planes. We use a decision tree classifier [6, 8] for this, which is a fast and scalable machine-learning algorithm that can discover such a classification based on Boolean guards.

Neither phase is meant to return the best solution. The geometry phase tries to find k planes that cover all points, and uses a greedy algorithm that need not necessarily work; in this case, we may increase k, and hence our algorithm might find more planes than necessary. The second phase for finding guards also does not necessarily find the minimal tree. Needless to say, the optimal versions of these problems are intractable. However, the algorithms we employ are extremely efficient; there are no NP oracles (SAT/SMT solvers) used. The learning of decision trees is based on information theory to choose the best guards at each point, and work well in practice in producing small trees [6].

We implement our learning algorithm and also build a teacher who has particular target functions, and instructs the learner using counter-examples obtained with the help of an SMT solver. We show that for many functions with reasonable guards and linear expressions, our technique performs extremely well. Furthermore, we can express the problem of learning guarded affine functions in the SyGuS framework [1], and hence use the black-box synthesis algorithms that are implemented for SyGuS. We show that our tool performs much better than these general-purpose synthesis techniques for this class of problems.

2 A Learning Algorithm based on Geometry and Decision Trees

The learner learns from a set of sample points $S = \{(\boldsymbol{x}_i, v_i), i = 1, \ldots n\}$. A guarded linear expression e satisfies such a set of samples S if the function f defined by the expression e maps each $\boldsymbol{x}_i$ to v_i.

As we mentioned earlier, the learner works in two phases. The first phase finds the leaf expressions using geometry and the second phase finds a guarded expression that maps points to these planes. We now describe these two phases.

2.1 Finding leaf planes using geometric techniques

The first phase, based on geometry, finds a small set of (unguarded) linear expressions P such that for every sample point, there is at least one linear expression in P that will produce the right output for that point. This phase hence discovers the set of leaf expressions in the guarded linear expression. Let $|S| = n$ where n is large, and let us assume that we want to find k planes that cover all the points, where k is small. Let the function being synthesized be of arity d. Each sample point in S can be viewed as an input-output pair, $p = (\boldsymbol{x}, y)$ such that $f(\boldsymbol{x}) = y$. We view them as points in a $(d + 1)$-dimensional space, and try to find, using a greedy strategy, a small number of planes such that every point falls on at least one plane. We start with a small budget for k and increase k when it doesn't suffice.

Assuming that there are k planes that cover all the points, there must be at least $\lceil n/k \rceil$ points that are covered by a single plane. Hence, our strategy is to find a plane in a greedy manner that covers at least these many points. Once we find such a plane, we can remove all the points that are covered by that plane, and recurse, decrementing k.

Note that in a $(d+1)$-dimensional space, one can always a construct a plane that passes through any $(d+1)$ points. Hence, our strategy is to choose sets of $(d+2)$ points and check if they are coplanar (and then check if they cover enough points in the sample). Since we are synthesizing a guarded linear expression, it is likely that the leaf planes are defined over a local region, and hence we would like to choose the $d+2$ points such that they are *close* to each other. Our algorithm `Construct-Plane`, depicted below, searches for a plane by (a) choosing a random point p and taking the closest $2d$ points next to p, by computing the distance of all points to p, sorting them, and picking the closest $2d$ points and (b) choosing *every* combination of $(d+2)$ points from this set and checking it for coplanarity.

```
CONSTRUCT-PLANE($S$)
1  Select a random point $p = (\boldsymbol{x}, y) \in S$
2  $C$ = set of $2d$ points closest to $p$
3  $Y$ = collection of all subsets of $(d+1)$ points in $C$
4  repeat for all $Z$ in $Y$
5      if the set of points in $(Z \cup p)$ are coplanar
6          $pln = find\_plane\ (Z \cup p)$
7          $Sel$ = set of points in $S$ that lie on plane $pln$
8          if $|Sel| > \lceil |S|/k \rceil$
9              label the points in $Sel$ as $pln$
10             return $Sel, pln$
```

$$\begin{vmatrix} x_1^1 & x_1^2 & \dots & x_1^{d+1} & 1 \\ x_2^1 & x_2^2 & \dots & x_2^{d+1} & 1 \\ \vdots & \vdots & & \vdots & \vdots \\ x_{d+2}^1 & x_{d+2}^2 & \dots & x_{d+2}^{d+1} & 1 \end{vmatrix} = 0$$

Fig. 1. (a) Algorithm for constructing planes that cover the input points (b) Coplanarity check for a set of points.

Coplanarity can be verified by checking the value of determinant as above (Figure 1b), and the plane defined by these $(d+2)$ points can be constructed by solving for the co-efficients c_i in the set of equations $\Sigma_{i=1}^{n} c_i x_i = c_{n+1}$, where we substitute the x_i's with the points we have chosen. The above two require numerical solvers and can be achieved using software like MatLab or Octave.

If the above process discovers k planes that cover all points in the sample, then we are done. If not, we are either left with too few points ($< d+2$) or too many points and have run out of the budget of planes. In the former case, we ignore these points, compute a guarded linear expression for the points that we have already covered using the second phase, and then add these points back as special points on which the answers are fixed using the appropriate constants in the sample. In the latter case, we increase our budget k, and continue.

There are several parameters that can be tuned for performance, including (a) how many points the teacher returns in each round, (b) the number of points in the window from which we select points to find planes, (c) the threshold of coverage for selecting a plane, etc. These parameters can be tweaked for better performance on the application at hand.

2.2 Finding conditionals using decision tree learning

The first phase identifies a set of planes and labels each input in the sample with a plane from this set that correctly maps it to its output. In the second phase, we use state-of-the-art decision tree classification algorithm [6, 8] to synthesize the conditional guards that classify all inputs such that the inputs in the sample are mapped to the correct planes. Decision trees can classify points according to a finite set of *numerical attributes*. We choose numerical attributes that are linear combinations of the variables with bounded integer coefficients (since we expect the coefficients in the guards to be small, the learner considers attributes of the form $\Sigma a_i x_i$ where $\Sigma a_i < K$ for a small K, where K increases in an outer loop). The decision-tree learner then constructs a classifier that uses Boolean combinations of formulas of the form $a \leq t$, where a is a numerical attribute and t is a threshold (constant) which it synthesizes. Note that the linear coefficients for the guards are enumerated by our tool— the decision tree learner just picks appropriate numerical attribute and synthesizes the thresholds.

The decision tree learner that we use is a standard state-of-the-art decision tree algorithm, called C5.0 [8,9], and is extremely scalable and has an inductive bias to learn smaller trees. It constructs trees using an algorithm that does no back-tracking, but chooses the best attributes heuristically using *information gain*, calculated using Shannon's entropy measure. We disable some features in the C5.0 algorithm such as *pruning*, which is traditionally performed to reduce overfitting, since we want a classifier that works precisely on the given sample and cannot tolerate errors. During the construction of the tree, if there are several attributes with the highest information gain, we choose the attribute that has the smallest absolute value. This heuristic biases the learner towards synthesizing guards that have smaller threshold values.

3 Evaluation

We implemented the two phases of the learner as follows: The geometric phase is implemented using a numerical solver, Octave, and the classifier phase is implemented using decision tree classification algorithm C5.0. The output of both these two phases is then combined to construct a hypothesis that is conjectured as the target guarded linear expression.

In order to evaluate our tool, we also implemented a teacher which knows a target guarded affine function f and provides counter-examples to the learner using a constraint solver. Given a hypothesis H the teacher checks if there is some valuation for variables $\boldsymbol{x}$ such that $H(\boldsymbol{x}) \neq f(\boldsymbol{x})$. If such a valuation exists, the teacher returns $(\boldsymbol{x}, f(\boldsymbol{x}))$ as a counterexample to the learner.

All experiments were performed on a system with an Intel Core i7 2.2 GHz processor and 4GB RAM running 64-bit Ubuntu 14.04 OS, with a timeout of $600s$. In Table 1 we tabulate our experimental results comparing our learner[1] with the enumerative, the stochastic, and the symbolic SyGus solver [1] for learning various target guarded affine functions of increasing complexity, both in

[1] Our tool can be accessed at `http://web.engr.illinois.edu/~ssaha6/Alchemist/`

Target Guarded Affine Function	**Enumerative solver**	**Stochastic solver**	**Symbolic solver**	**Alchemist**
$x + y$	$0.0s$	$0.3s$	$5.4s$	$0.7s$
$3x + 3y + 3$	$2m12.4s$	$16.1s$	*timeout*	$0.7s$
$5x + 5y + 5$	*timeout*	$4m1.1s$	*timeout*	$0.7s$
$max2 : ite(x < y,\ y,\ x)$	$0.0s$	$0.2s$	$2.6s$	$0.6s$
$max3 : ite(x < y, ite(y < z,\ z,\ y), ite(x < z,\ z,\ x))$	*timeout*	$5.4s$	*timeout*	$1.1s$
$max4(x, y, z, u)$	*timeout*	$6m21.6s$	*timeout*	$20.5s$
$max5(x, y, z, u, v)$	*timeout*	*timeout*	*timeout*	$1m30.0s$
$ite(x + y \leq 1, x + y, x - y)$	$1.2s$	$2.5s$	*timeout*	$0.9s$
$ite(x + y + z \leq 1, x + y, x - y)$	$12.8s$	$6.5s$	*timeout*	$0.8s$
$ite(2x + y + z \leq 1, x + y, x - y)$	$6m42.7s$	$8.1s$	*timeout*	$1.4s$
$ite(2x + 2y + z \leq 1, x + y, x - y)$	*timeout*	$7.9s$	*timeout*	$1.4s$
$ite(x + y \geq 1, ite(x + z \geq 1, x + 1, y + 1), z + 1)$	*timeout*	$20.7s$	*timeout*	$1.8s$
$ite(x + y \geq 1, ite(x + z \geq 1, x + 1, y + 1),$ $ite(y + z \geq 1, y + 1, z + 1))$	*timeout*	$4m37.2s$	*timeout*	$2.9s$
$ite(x \geq 5, 5x + 3y + 17, 3x + 1)$	*timeout*	*timeout*	*timeout*	$0.9s$
$ite(x \leq y + 4, min(x, y, z), max(x, y, z))$	*timeout*	*timeout*	*timeout*	$1.8s$
if $x + y <= 1$ *then* $10x + 10y + 10$ *elseif* $x + y <= 2$ *then* $20x + 20y - 20$ *elseif* $x + y <= 3$ *then* $30x + 30y + 30$ *elseif* $x + y <= 4$ *then* $40x + 40y - 40$ *elseif* $x + y <= 5$ *then* $50x + 50y + 50$ *else* $60x + 60y - 60$	*timeout*	*timeout*	*timeout*	$27.9s$

Table 1. Experimental results. The timeout is set to $600s$.

terms of the Boolean structure and in terms of the coefficients. We also evaluate on relevant SyGuS benchmarks. For the stochastic solver, we report the time averaged over ten runs. From the results, it seems the SyGus solvers are very general and do not exploit the geometry of this domain well. Also, the machine learning algorithms seem better at sifting through candidate guards and picking a small subset that work.

Apart from the above mentioned solvers, we also tried the SyGus solver based on Sketch [10] but it failed to execute for most of the problems. We could not try [2] as being a passive algorithm which approximates the solution makes it very hard to compare the tools empirically.

4 Conclusions

The learning based synthesis of guarded affine functions we have proposed seems very promising and a good alternative to existing synthesis techniques. An earlier version of this solver was submitted as a solver to the SyGuS synthesis competition; however, note that in a more general setting, building a teacher is not easy as the teacher does not know precisely the function to be synthesized. We hence built a teacher who would identify at least certain inputs on which the function was determined and feed these to the learner. A more general approach that extends our work to solving general synthesis problems involving guarded affine functions is an interesting direction for future work.

5 Acknowledgments

We thank Sariel Har-Peled for discussions on geometric techniques for synthesizing leaf expressions. This work was partially supported by NSF Expeditions in Computing ExCAPE Award #1138994.

References

1. Alur, R., Bodík, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013. pp. 1–17 (2013)
2. Alur, R., Singhania, N.: Precise piecewise affine models from input-output data. In: Proceedings of the 14th International Conference on Embedded Software. pp. 3:1–3:10. EMSOFT '14, ACM, New York, NY, USA (2014)
3. Angluin, D.: Queries and concept learning. Mach. Learn. 2(4), 319–342 (Apr 1988)
4. Bemporad, A., Garulli, A., Paoletti, S., Vicino, A.: A bounded-error approach to piecewise affine system identification. IEEE Trans. Automat. Contr. 50(10), 1567–1580 (2005)
5. Ferrari-Trecate, G., Muselli, M., Liberati, D., Morari, M.: A clustering technique for the identification of piecewise affine systems. Automatica 39(2), 205–217 (Feb 2003)
6. Mitchell, T.M.: Machine learning. McGraw Hill series in computer science, McGraw-Hill (1997)
7. Paoletti, S., Juloski, A.L., Ferrari-Trecate, G., Vidal, R.: Identification of hybrid systems: A tutorial. Eur. J. Control 13(2-3), 242–260 (2007)
8. Quinlan, J.R.: Induction of decision trees. Machine Learning 1(1), 81–106 (1986)
9. Quinlan, J.R.: C4.5: Programs for Machine Learning. Morgan Kaufmann (1993)
10. Solar-Lezama, A.: Program sketching. STTT 15(5-6), 475–495 (2013)
11. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006. pp. 404–415 (2006)
12. Vidal, R., Soatto, S., Sastry, S.: An algebraic geometric approach to the identification of a class of linear hybrid systems. In: Proceedings of the IEEE Conference on Decision and Control. vol. 1, pp. 167–172 (December 2003)