Synthesizing Action Sequences for Modifying Model Decisions

Goutham Ramakrishnan*, **Yun Chan Lee***, **Aws Albarghouthi**
University of Wisconsin–Madison
{gouthamr,aws}@cs.wisc.edu, yunchan.c.lee@gmail.com

Abstract

When a model makes a consequential decision, e.g., denying someone a loan, it needs to additionally generate actionable, realistic feedback on what the person can do to favorably change the decision. We cast this problem through the lens of program synthesis, in which our goal is to synthesize an optimal (realistically cheapest or simplest) sequence of actions that if a person executes successfully can change their classification. We present a novel and general approach that combines search-based program synthesis and test-time adversarial attacks to construct action sequences over a domain-specific set of actions. We demonstrate the effectiveness of our approach on a number of deep neural networks.

1 Introduction

Today, predictive models are responsible for an ever-expanding spectrum of decisions, some of which are consequential to the lives and well-being of individuals—e.g., mortgage underwriting, job screening, healthcare decisions, criminal risk assessment, and many more. As such, questions about fairness and transparency have taken center stage in the debate over the increasing use of machine learning to automate decisions in sensitive domains, and a vibrant research community has emerged to explore and address the many facets of these questions.

In this paper, we are interested in the problem of providing actionable feedback to the subjects of algorithmic decision-making. For instance, imagine that you are denied a mortgage to buy your first home, thanks to a model that consumed a set of features and deemed you too risky. We envision that such an algorithmic process should additionally give you realistic, actionable feedback that will increase your chances of receiving a loan. For instance, you might receive the following feedback: *increase your down payment by $1000* and *limit credit card debt to a maximum of $5000 for the next two months*. This is probably reasonable advice, in contrast with, say, a much harder to fulfill feedback like *change your marital status from single to married*.

We view this problem through the lens of *program synthesis* (Gulwani, Polozov, and Singh 2017): we want to synthesize an *optimal sequence of instructions* that a human can *execute* so that they favorably change the decision of some model. Optimality here is with respect to a measure of how hard it is for a person to perform the provided actions—we want to provide the simplest, cheapest feedback. There are many challenges in solving this problem: (*i*) the combinatorial blow-up in the space of action sequences, (*ii*) the fact that actions are parameterized by real values and have variable cost (e.g., *increase savings by $X*), (*iii*) and the fact that action ordering is important, e.g., *you can only do A after you have done B* or *you can only do A if you are more than 35 years old*.

To attack this problem, we make the key observation that the problem resembles that of generating *adversarial examples* (Szegedy et al. 2013; Goodfellow, Shlens, and Szegedy 2015; Carlini and Wagner 2017; Papernot et al. 2017), where we usually want to slightly perturb the pixels of an input image to modify the classification, e.g., make a neural network think a dog is a panda. However, in our case, we are *not* looking for an imperceptible perturbation to the input features, but one that results from the application of real-world actions. With this view in mind, we present a new technique that adapts and combines (*i*) *search-based program synthesis* (Alur et al. 2018) to traverse the space of action sequences and (*ii*) *optimization-based adversarial example generation* (Carlini and Wagner 2017) techniques to discover action parameters. This combination allows our approach to handle a rich class of (differentiable) models, e.g., deep neural networks, and complex domain-specific actions and cost models.

Setting and Consequences. At this point, it is important to recognize the possibility that the solution we propose for the problem setting may be vulnerable to unethical practices. Although our technique, in principle, may be used by users to maliciously *game* the system, we believe in its importance and cannot envision a world in which subjects are unable to understand and *act* on black-box decisions. One setting we envision is where users cannot adversarially attack the decision-maker as they do not have access to the model. The intention would be to use our technique as a means for the service provider to give meaningful and actionable feedback to its users, making the decision process more transparent.

Most Relevant Work. To our knowledge, the idea of providing actionable feedback for algorithmic decisions was

*Equal contributions by authors

first advocated by Wachter et al. (2017) in a law article. Ustun et al. (2019) implemented this idea by searching for a minimal change to input features to modify the classification of simple linear models (logistic regression). Their approach discretizes the feature space and encodes the search as an integer programming (IP) problem. Zhang et al. (2018) consider a similar problem over neural networks composed of ReLUs, exploiting the linear structure of ReLUs to solve a series of LP problems to construct a convex region of positively classified points that are close to the input. Our work is different in a number of dimensions: (*i*) Our algorithm is quite general: by reducing the search to an optimization problem, à la test-time adversarial attacks, it can handle the general class of differentiable models (as well as differentiable action and cost definitions), instead of just linear models. (*ii*) We allow defining complex, nuanced actions that mimic real-world interventions, as opposed to arbitrary modifications to input features. (*iii*) Similarly, we allow encoding cost models to assign different costs to actions, with the goal of synthesizing the simplest feedback a person can act on. See Section 5 for an extended discussion of related work.

Contributions. Our contributions are:

- We define the problem of *synthesizing optimal action sequences* that can favorably change the output of a machine-learned model. We view the problem through the lens of program synthesis, where our goal is to synthesize a program over a domain-specific language of real-world actions specified by a domain expert.
- We present an algorithm that combines *search-based program synthesis* to traverse the space of action sequences and *optimization-based test-time adversarial attacks* to discover optimal parameters for action sequences. We demonstrate how to leverage the results of optimization to guide the combinatorial search.
- We implement our approach and apply it to a number of neural networks learned from popular datasets. Our results demonstrate the effectiveness of our approach, the benefits of our algorithmic decisions, and the robustness of the synthesized action sequences to noise.

2 Optimal Action Sequences

In this section, we formally define the problem of synthesizing optimal action sequences.

Decision-Making Model. We shall use $f : X \to \{0, 1\}$ to denote a classifier over inputs in X. For simplicity of exposition, and without loss of generality, we restrict f to be a binary classifier—our approach extends naturally to k-ary classifiers.

A DSL of Actions. For a given classification domain, we assume that we have a *domain-specific* set of actions $A = \{a_1, \ldots, a_n\}$, perhaps curated by a domain expert. Each action $a \in A$ is a function $a : X \times R \to X$, where R is the set of parameters that a can take. For example, imagine $x \in X$ are features of a person applying for a loan. An action $a(x, 1000)$ may be one that increases x's savings by \$1000, resulting in $x' \in X$.

For each action $a_i \in A$, we associate a cost function $c_i : X \times R \to \mathbb{R}_{\geqslant 0}$, denoting the cost of applying a_i on a given input and parameters. Making c_i a function of inputs and parameters of an action allows us to define fine-grained cost functions, e.g., some actions may be easier for some people, but not for others. For instance, in the US, acquiring a credit card is much easier for someone with a credit history in contrast to someone who recently arrived on a work visa. Similarly, varying the parameter of an action should vary the cost, e.g., *increase savings by \$1000* should be much cheaper than *increase savings by \$1,000,000*.

Additionally, for each action a_i, we associate a Boolean *precondition* $pre_i : X \times R \to \mathbb{B}$, indicating whether action $a_i(x, r)$ is *feasible* for a given input x and parameter r. There are a number of potential use cases for preconditions. For instance, the action of renting a car may be only allowed if you are over 21 years old; this can be encoded as the precondition $age \geqslant 21$. Preconditions can also encode valid parameters, e.g., you cannot increase your credit score past 850, so an action which recommends increasing your credit score by r will have the precondition $creditScore + r \leqslant 850$.

Optimal Action Sequence. Fix an input $x \in X$ and assume that $f(x) = 0$. Informally, our goal is to find the least-cost, feasible sequence of actions that can transform x into an x' such that $f(x') = 1$.

Formally, we will define an action sequence using a pair of sequences $\langle \sigma, \rho \rangle$, denoting actions in A and their corresponding parameters in R, respectively. Specifically, σ is a sequence of integers in $[1, |A|]$ (action indices), and σ_i denotes the ith element in this sequence. ρ is a sequence of parameters such that each $\rho_i \in R$. We assume $|\rho| = |\sigma| = k$, and we will use k throughout to denote $|\sigma|$.

Given pair $\langle \sigma, \rho \rangle$, in what follows, we will use $x_i = a_{\sigma_i}(x_{i-1}, \rho_i)$, where $i \in [1, k]$ and $x_0 = x$. That is, variable x_i refers to the result of applying the first i actions defined by $\langle \sigma, \rho \rangle$ to the input x. We are therefore looking for a feasible, least-cost sequence of actions $a_{\sigma_1}, \ldots, a_{\sigma_k}$ and associated parameters $\rho_1, \ldots, \rho_k$, which, if applied starting at x, results in x_k that is classified as 1 by f. This is captured by the following optimization problem:[1]

$$\arg\min_{\langle \sigma, \rho \rangle} \sum_{i=1}^{k} c_{\sigma_i}(x_{i-1}, \rho_i) \tag{1}$$

$$\text{subject to } f(x_k) = 1 \text{ and } \bigwedge_{i=1}^{k} pre_{\sigma_i}(x_{i-1}, \rho_i)$$

3 An Algorithm for Sequence Synthesis

We now present our technique for synthesizing action sequences, based on the optimization objective outlined above. Our algorithm assumes a differentiable model, e.g., a deep neural network, of the form $f : \mathbb{R}^m \to \{0, 1\}$, as well as

[1] Equivalently, we can cast this as an optimal *planning* problem, where x is the initial state, our goal state x_k is one where $f(x_k) = 1$, and actions transition us from one state to another.

Algorithm 1 Full synthesis algorithm

function SYNTHESIZE(model f, instance x, actions A)
  $S \leftarrow \{\langle \sigma^{\emptyset}, \rho^{\emptyset} \rangle\}$, where $\langle \sigma^{\emptyset}, \rho^{\emptyset} \rangle$ are the empty action and parameters sequences, respectively
  repeat
    Let $\langle \sigma, \rho \rangle \in S$ and $a_i \in A$ be with smallest $score(\sigma, \rho, a_i)$ ▷ $\langle \sigma a_i, - \rangle \notin S$
    Compute parameters ρ' for sequence σa_i, $\rho' = \arg\min_{\rho} \; c \cdot h(x_k) + \sum_{i=1}^{k} C_{\sigma_i}(x_{i-1}, \rho_i)$
    $S \leftarrow S \cup \{\langle \sigma a_i, \rho' \rangle\}$
  until threshold exceeded ▷ *threshold can be, e.g., search depth*
  return Solution of Problem 1 restricted to sequences in S

differentiable actions, cost functions and preconditions. To solve Problem 1, defined in Section 2, we break it into two pieces: (*i*) a discrete search through the space of action sequences σ and (*ii*) a continuous-optimization-based search through the space of action parameters ρ, which we assume to be real-valued.

In Section 3.1, we begin by describing the optimization technique, by assuming we have a fixed sequence of actions and setting up an optimization problem—an adaptation of Carlini-Wagner's adversarial attack (Carlini and Wagner 2017)—to learn the parameters to those actions. Then, in Section 3.2, we present the full algorithm, a search-based synthesis algorithm for discovering action sequences, which uses the optimization technique from Section 3.1 as a subroutine.

Remark on optimality: We note that the constrained optimization Problem 1 is hard in general—e.g., even very limited numerical planning problems that can be posed as Problem 1 are undecidable (Helmert 2002). Our use of adversarial attacks in the following section necessarily relaxes some of the constraints and is therefore not guaranteed to result in optimal action sequences.

3.1 Adversarial Parameter Learning

We now assume that we have a fixed sequence of actions σ, as defined in Section 2. Our goal is to find the parameters ρ such that $\langle \sigma, \rho \rangle$ satisfies the constraints of Problem 1. Specifically, our solution to this problem is an adaptation of Carlini and Wagner's seminal adversarial attack technique against neural networks (Carlini and Wagner 2017), but in a setting where the "attack" is comprised of a sequence of actions with preconditions and varying costs.

Henceforth, we shall assume that the model is a neural network $f : \mathbb{R}^m \rightarrow \mathbb{R}^2$ (where the output denotes a distribution over the two classification labels). Additionally, $f(x) = \text{softmax}(g(x))$, i.e., function g is the output of the pre-softmax layers of the network.

Boolean Precondition Relaxation. Our goal is to construct a tractable optimization problem whose solution results in the parameters ρ to the given action sequence σ. We begin by defining the following constrained optimization problem which relaxes the Boolean precondition constraints:

$$\arg\min_{\rho} \sum_{i=1}^{k} c_{\sigma_i}(x_{i-1}, \rho_i) + pre'_{\sigma_i}(x_{i-1}, \rho_i) \qquad (2)$$
$$\text{subject to } f(x_k)_1 > f(x_k)_0$$

where $f(x)_j$ is the probability of class j, and the function pre'_i is a continuous relaxation of the Boolean precondition pre_i. Specifically, we encode preconditions by imposing a high cost on violating them. For instance, if $pre_i(x, r) = x > c$, where c is a constant, then we define $pre'_i(x, r) = \tau \exp(-\tau'(x - c))$, where τ and τ' are hyperparameters.[2] The hyperparameters τ and τ' determine the steepness of the continuous boundaries; the values we choose are inversely proportional to the size of the domain of x. Conjunctions of Boolean predicates are encoded as a summation of their relaxations. We can now define $C_{\sigma_i}(x_{i-1}, \rho_i)$ to be the overall cost incurred by the action-parameter pair $\langle \sigma_i, \rho_i \rangle$, i.e.

$$C_{\sigma_i}(x_{i-1}, \rho_i) = c_{\sigma_i}(x_{i-1}, \rho_i) + pre'_{\sigma_i}(x_{i-1}, \rho_i)$$

Carlini–Wagner Relaxation. Now that we have relaxed preconditions, what is left is the classification constraint $f(x_k)_1 > f(x_k)_0$ in Problem 2. Following Carlini and Wagner, we transform the constraint $f(x_k)_1 > f(x_k)_0$ into the objective function that is the distance between logit (pre-softmax) output: $h(x_k) = max(0, g(x_k)_0 - g(x_k)_1)$.[3]

This results in the following optimization problem:

$$\arg\min_{\rho} \; c \cdot h(x_k) + \sum_{i=1}^{k} C_{\sigma_i}(x_{i-1}, \rho_i) \qquad (3)$$

In practice, we perform an adaptive search for the best value of the hyperparameter c as we solve the optimization problem: at a variable length interval t of minimization steps, we determine how close the search is to the decision boundary and adjust c and t accordingly.

3.2 Sequence Synthesis and Optimization

Now that we have defined the optimization problem for discovering parameters ρ of a given action sequence σ, we proceed to describe the full algorithm, where we search the space of action sequences.

[2] We assume that expressions in preconditions are differentiable.

[3] Note that there many alternative relaxations of $f(x_k)_1 > f(x_k)_0$; Carlini and Wagner explore a number of alternatives, e.g., using f instead of g, and show that this outperforms them.

Dataset/model	Network architecture	#Features	#Actions
German Credit Data	2 dense layers of 40 ReLUs each	20	7
Adult Dataset	2 dense layers of 50 ReLUs each	14	6
Fannie Mae Loan Perf.	5 dense layers of 200 ReLUs each	21	5
Drawing Recognition	3 1D conv. layers, 1 dense layer of 1024 ReLUs	512 (pixels)	1

Table 1: Overview of datasets/models for evaluation; we will refer to a model by the italicized prefix of its name

Algorithm Description. Algorithm 1 is a simple search guided by a parametric *score* function that directs the search—lower score is better. The algorithm maintains a set of sequences S, which initially is the pair $\langle \sigma^{\emptyset}, \rho^{\emptyset} \rangle$, containing two empty sequences. In every iteration, the algorithm picks an action sequence in S, extends it with a new action from A, and solves optimization Problem 3 to compute a new set of parameters. The search process continues until some preset threshold is exceeded, e.g., we have covered all sequences of some length or we have discovered a sequence that is below some cost upper bound. Finally, we can return the best pair in S, i.e., the one with the minimal cost that changes the classification and satisfies all preconditions, as per Problem 1.

Defining the Scoring Function. The definition of the scoring function $score$ dictates the speed with which the algorithm arrives at an best action sequence. In our evaluation, we consider a number of definitions, the first of which, the *vanilla* definition, is simple, but often inefficient:

$$score_v(\sigma, \rho, a_i) = k + 1$$

This definition turns the search into a breadth-first search, as shorter sequences are evaluated first.

A more informed score function we consider is to simply return the value of the objective function in Problem 3 for a given sequence $\langle \sigma, \rho \rangle$. We call this function $score_o$. Notice that $score_o$ does not consider the action to apply, so the action with which to expand the sequence is chosen arbitrarily.

Next, we consider a more sophisticated scoring function: we want to pick the action that modifies the most important features. To do so, we use the gradient of model features, with respect to the target loss, as a proxy for the most important features. The idea is that we want to pick the action that modifies the features with the largest gradient. For every action $a_i \in A$, we use the *fp*(a_i) to denote its *footprint*: the set of indices of the input features it can modify, e.g., *fp*$(a_i) = \{1, 2\}$ means that it modifies features 1 and 2 and leaves all others unchanged. Given $\langle \sigma, \rho \rangle \in S$, let x' be the result of applying $\langle \sigma, \rho \rangle$ to the input instance x. We now define the score function as:

$$score_g(\sigma, \rho, a_i) = - \operatorname*{mean}_{j \in fp(a_i)} \left| \frac{d\ell(f(x'))}{dx_j} \right|$$

In other words, the score of applying a_i after the sequence σ depends on the average gradient of the target loss ℓ—binary cross-entropy loss with respect to the target label, i.e., 1—with respect to the features in a_i's footprint.

4 Implementation and Evaluation

Implementation. Our algorithm is implemented[4] in Python 3, using TensorFlow (Abadi et al. 2016). Actions, along with their costs and preconditions, are implemented as instances of an `Action` Python class. The Adam Optimizer (Kingma and Ba 2014) is used to solve optimization Problem 3. For fast experimentation, we implemented a brute-force version of Algorithm 1 where all sequences up to some length n are optimized in parallel using AWS Lambda—i.e., each sequence is optimized as a separate Lambda.

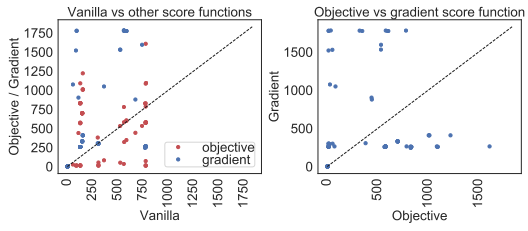
Research Questions. We have designed our experiments to answer the following research questions: ***Q1***: Can our technique synthesize action sequences for non-trivial models and actions? ***Q2***: How do different score functions impact algorithm performance? ***Q3***: How robust are the synthesized action sequences to noise? Further, ***Q4***: we explore other applications of our technique, beyond consequential decisions.

Domains for Evaluation. For exploring questions ***Q1-3***, we consider three popular datasets: The German Credit Data (Dua and Graff 2017) and the Fannie Mae Single Family Loan Performance (Mae 2014) datasets have to do with evaluating loan applications—high or low risk. The Adult Dataset (Dua and Graff 2017) predicts income as high or low—the envisioned use case is it can be used to set salaries. Table 1 summarizes our datasets and models: For each of the three datasets, we (*i*) trained a deep neural network for classification (in the case of the Fannie Mae dataset, we used the neural network from (Zhang, Solar-Lezama, and Singh 2018)), (*ii*) constructed a number of realistic actions along with their associated costs and preconditions, and (*iii*) randomly chose 100 negatively classified instances (i.e., 300 instances overall) from the test sets to apply our algorithm to.

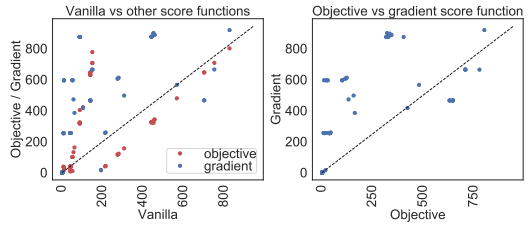
The actions constructed for each domain cover both numerical and categorical features; a number of actions for each domain modify multiple features—e.g., change the debt-to-income ratio, or get a degree (which takes time and therefore increases age).

To explore ***Q4***, we also consider the Drawing Recognition task (Zhang, Solar-Lezama, and Singh 2018) based on Google's *Quick, Draw!* dataset (Google 2017). The goal is to extend a drawing, represented as a set of straight lines, so as it is classified as, e.g., a cat. Hence, we only build one action for this model: add line from point (a, b) to (a', b').

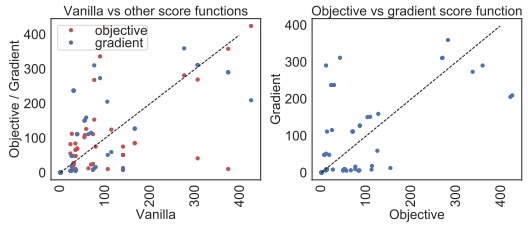
[4] https://github.com/goutham7r/synth-action-seq



(a) Results for the German model



(b) Results for the Adult model



(c) Results for the Fannie Mae model

Figure 1: Number of optimization problems solved—i.e., loop iterations of Algorithm 1—before arriving at the solution sequence for different score functions. Each dot represents an instance.

4.1 Results

We are now ready to discuss the results. Henceforth, when we refer to *solution sequence*, we mean the best solution we find after Algorithm 1 has explored all sequences of length less than an upper bound.

Instances Solved. For our primary models, we make our algorithm consider all sequences of length $\leqslant 4$. The rationale behind this choice is that we usually want a small set of instructions to provide to an individual. Our algorithm was able to find solutions to 100/100 instances in the German model, 90/100 instances in the Adult model, and 62/100 instances in the Fannie Mae model. Note that inability to find a solution could be due to insufficient actions or incompleteness of the search—sequence length limit, relaxation of optimization problem, or local minima. In particular, the relatively inferior success rate on the Fannie Mae model may be a direct result of the fact that the neural network is much deeper.

To give an example of a synthesized action sequence by our algorithm, consider the following sequence of 3 actions for Fannie Mae: *Increase credit score by 17 points*, *Reduce loan term by 43 months*, and *Increase interest rate by 0.621*.

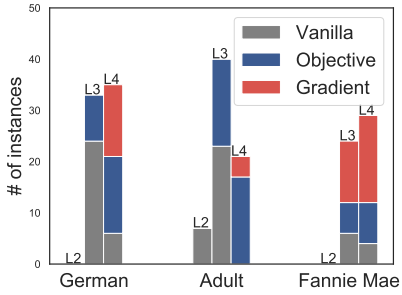


Figure 2: Performance of score functions as length of solution sequence increases—Lx denotes sequence of length x

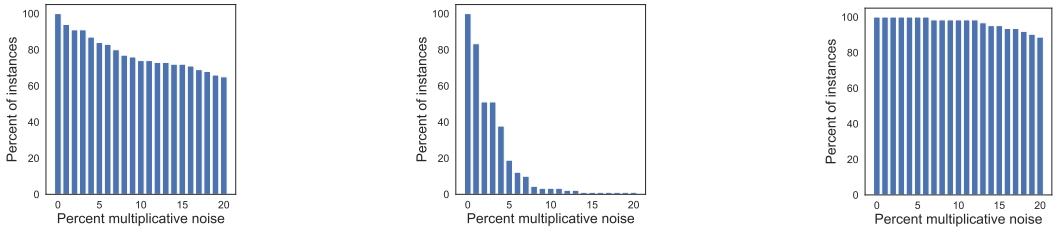
Effects of Score Function. Next, we explore the effects of different score functions. Recall, in Section 3.2, we defined the *vanilla* score function $score_v$, where sequences are explored by length (a breadth-first search); the *objective* score function $score_o$, where the sequence with the smallest solution to Problem 3 is explored; and the *gradient* score function $score_g$, where gradient of cross-entropy loss is used to choose the sequence and action to explore.

Figure 1 shows the results of the German, Adult, and Fannie Mae models. Each point is one of the instances and the axes represent the iteration at which a score function arrived at the solution sequence.[5] The left plot compares $score_v$ vs $score_g$ (blue) and $score_o$ (red); the right plot compares $score_o$ vs $score_g$. We make two important observations: First, the vanilla score function excels on many instances (points above diagonal). After investigating those instances, we observe that they have short solution sequences—of length 1 or 2. This is perhaps expected, as $score_o$ and $score_g$ may quickly lead the search towards longer sequences, missing short solutions until much later. We further illustrate this observation in Figure 2, where we plot the number of times each score function outperformed others (in terms of number of iterations of Algorithm 1) when the solution sequence is of length 2,3, and 4. We see that when solution sequences are longer, $score_v$ stops being effective. Second, we observe that both the gradient and objective score functions, $score_g$ and $score_o$, have their merits, e.g., for Fannie Mae, $score_g$ dominates while for Adult $score_o$ dominates.

Robustness of Synthesized Sequences. We investigate how robust our solutions are to perturbations in their parameters. The idea is that a person given feedback from our algorithm may not be able to fulfil it precisely. We simulate this scenario by adding noise to the parameters of the sequences. Specifically, for each synthesized solution sequence and each parameter r in the sequence, we uniformly sample values from the interval $[(1-\theta)r, (1+\theta)r]$, where θ denotes the maximum percentage change.

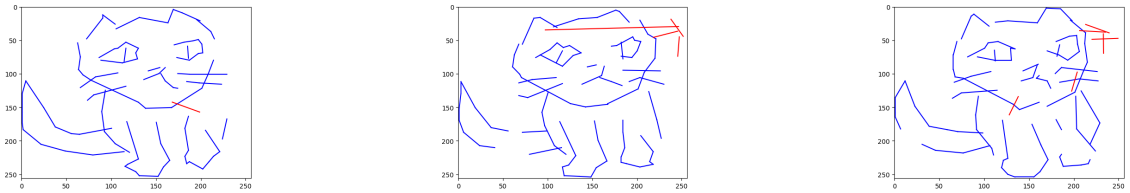
Figure 3 summarizes the results for the three primary models. We plot the number of instances that *succeed* with

[5]Time/iteration is $\sim$15s across instances; we thus focus on number of iterations as performance measure.



(a) German (b) Adult (c) Fannie Mae

Figure 3: Percent of instances (y-axis) that tolerate noise θ (x-axis) noise with probability $\geqslant 0.8$



(a) sequence length = 1 (b) sequence length = 4 (c) sequence length = 6

Figure 4: Three *different* Quick, Draw! examples. Blue strokes comprise the input instance; red strokes are the results of applying actions.

a probability $\geqslant 0.8$ (i.e., are still solutions more than 80% of the time) as the amount of noise θ increases. Obviously, when θ is 0, all instances succeed with probability 1; as θ increases, the success rate of a number of instances falls below 0.8. We notice that the German and Fannie Mae solutions are quite robust, while only half of the Adult solutions can tolerate $\theta = 0.03$ noise.

While our problem formulation does not enforce robustness of solutions, the results show that most instances are quite robust to noise. We attribute this phenomenon to the Carlini–Wagner relaxation. Recall that we minimize $\max(0, g(x_k)_0 - g(x_k)_1)$. So solutions need only get to a point where $g(x_k)_0 \leqslant g(x_k)_1$—intuitively, barely beyond the decision boundary. However, we notice that, for most instances, solutions end up far from the boundary. Specifically, for the two models with more robust solutions, German and Fannie Mae, the average relative difference between g_1 and g_0—i.e., $(g(x_k)_1 - g(x_k)_0)/g(x_k)_1$—is 3.41 (sd=6.79) and 1.92 (sd=5.40), respectively. For Adult, the average relative difference is much smaller, 0.09 (sd=0.06), indicating that most solutions where quite close to the decision boundary, explaining their sensitivity to noise.

It would be interesting to further improve robustness by incorporating it as a first-class constraint in our problem, e.g., by reformulating Problem 3 as a *robust optimization* (Ben-Tal, El Ghaoui, and Nemirovski 2009) problem, so as we only discover solutions that are tolerant to noise. We plan to investigate this in future work.

Further Demonstration and Discussion. To further explore applications of our algorithm, we consider the Drawing Recognition model (Zhang, Solar-Lezama, and Singh 2018). Each drawing in this model is composed of up to 128 straight line strokes. We considered 16 sketches of cats that are not classified as cats by this model. We constructed an action that adds a single line stroke, where the parameters to the action are the source and target (x, y) coordinates. To ensure that the results of the actions are visible to the human eye, we add the precondition that stroke length is between 0.1 and 0.6 in length, where the image is 1×1. The cost of an action is the length of the stroke.

We ran our algorithm up to and including length 6 ($score$ has no effect since there is a single action). Our algorithm managed to synthesize action sequences for 11/16 instances. Three representative solutions are shown in Figure 4. The first solution, of length 1, appears to be an additional whisker to the cat. The second and third solutions, lengths 4 and 6, appear to be more arbitrary and thus may be more adversarial in nature.

Note that our task is qualitatively different from (Zhang, Solar-Lezama, and Singh 2018). They want to find the closest image across the decision boundary that has an ϵ-ball around it. So they start with an adversarial example and incrementally expand it into a *region* of examples. Our problem is motivated by application of real-world actions, and therefore we search for a sequence of actions to modify an image.

The results of our technique on the Drawing Recognition model may seem to suggest that the solutions are adversarial in nature. However, it is hard to formally characterize the difference between an adversarial attack and a reasonable action sequence. In image-based attacks, it's easy to tell if the modification is meaningful, but generally, e.g., in loans, this can probably be addressed by a domain-expert on a case-by-case basis. Observationally, we see that all our results look reasonable, i.e., there are no actions of the form *modify X by* ϵ, where ϵ is very small for practical purposes. Moreover, our experiments show that most synthesized se-

quences are robust to random perturbations in their parameters, suggesting that they are not adversarial corner cases. One concrete way we can protect against generating adversarial feedback is to restrict our technique to models that are trained to be *robust* against adversarial attacks (Madry et al. 2018)—however, the definition of robustness will have to be tailored to the specific domain.

The seemingly unrealistic strokes produced as solutions in the Drawing Recognition model may stem from the fact that the cost function simply penalizes the length of the stroke and in no way drives the drawing of a 'realistic' stroke (In fact, it is not obvious how one can specify a cost function that encourages strokes which look realistic). The demonstration of our technique on this dataset serves to exhibit the versatility of our problem setting and proposed solution.

5 Related Work

We focus on works not discussed in Section 1.

Interpretable Machine Learning. Recently, there has been a huge interest in explainability in machine learning, particularly for deep neural networks. Most of the works have to do with *highlighting* the important features that led to a prediction, e.g., pixels of an image or words of a sentence. For instance, the seminal work on LIME (Ribeiro, Singh, and Guestrin 2016) trains a simple local classifier and uses it to rank features by importance—many other works employ different techniques to hone in on important features, e.g., (Datta, Sen, and Zick 2016; Sundararajan, Taly, and Yan 2017; Lundberg and Lee 2017). This is usually not enough: knowing, for instance, that your credit score affected the loan decision does not tell you how much you need to increase it by to be eligible for a loan, or whether there are other actions you can take. This is the distinguishing aspect of our work—providing actionable feedback.

Program Synthesis. We view our algorithm through the lens of program synthesis. Our algorithm is a form of enumerative program synthesis, a simple paradigm that has shown to be performant in many domains—see (Alur et al. 2018) for an overview. Our work is also related to differentiable programming languages (Reed and De Freitas 2015; Bošnjak et al. 2017; Gaunt et al. 2017; 2016; Valkov et al. 2018). The idea is to use numerical optimization to fill in the holes (parameters) in differentiable programs. Our work is similar in that we define a differentiable language of actions and costs and use numerical optimization to learn appropriate parameters for those actions. At an abstract level, our algorithm is similar in nature to that of (Valkov et al. 2018). They enumerate functional programs over neural networks and use optimization to learn parameters; here, we enumerate action sequences and use optimization to learn their parameters. There is also a growing body of work on using deep learning to perform and guide program synthesis, e.g., (Parisotto et al. 2017; Chen, Liu, and Song 2019; Bunel et al. 2018)

Symbolic synthesis techniques typically use SAT/SMT solvers to search the space of programs (Solar-Lezama et al. 2006; Gulwani et al. 2011). Unfortunately, the range of programs they can encode is limited by decidable and practically efficient theories. While our problem can be encoded as an optimal SMT problem in linear arithmetic (Li et al. 2014)—equivalently, an MILP problem—we will have to restrict all actions, costs, and models to be linear. In practice, this is quite restrictive. While our approach is incomplete, unlike in decidable first-order theories, it offers the flexibility and generality of being able to handle arbitrary differentiable models and actions.

Planning and Reinforcement Learning. Our problem can also be viewed as a planning problem in a continuous (or hybrid) domain. Most such planners are restricted to linear domains that are favorable to an SMT or MILP encoding or restricted forms of non-linearity, e.g., (Cashmore et al. 2016; Piotrowski et al. 2016; Bryce et al. 2015). Some such planners also combine search and a form of optimization, typically LP, e.g., (Fernández-González, Williams, and Karpas 2018; Coles et al. 2012). Recently, (Wu, Say, and Sanner 2017) presented a scalable approach by reducing the planning problem to optimization and solving it with TensorFlow. In their setting, they deal with simple domains, e.g., 1 action; they do not have a goal state (in our case changing classification of the model), just a reward maximization objective; and they do not incorporate preconditions. Further, they are interested in long plans over some time horizon, while we focus on generating small, actionable plans.

Our problem is also related to reinforcement learning (RL) with continuous action spaces, e.g., (Lillicrap et al. 2015). The power of RL is its ability to construct a general policy that can lead to a goal state. Thus, given a model, it would be interesting to use RL to learn a single policy that we can then apply to any input, in contrast with our approach that learns a specific sequence of actions for every input.

6 Conclusion

We described a solution to the problem of presenting simple actionable feedback to subjects of a decision-making model so as to favorably change their classification. We presented a general solution where a domain expert specifies a differentiable set of actions that can be performed along with their costs. Then, we combine a search-based technique with an optimization technique to construct an optimal sequence of actions that leads to classification change. Our results demonstrate the promise of our technique and its applicability. There are many potential avenues for future work, e.g., exploring effects of different relaxations on optimality and robustness of results and adapting the algorithm to a complete black-box setting where we can only query the model. Another interesting avenue is to explore how our approach can be used to *game* the system in a *strategic classification* setting (Hardt et al. 2016).

Acknowledgments. We thank Xiaojin Zhu, Yingyu Liang, Loris D'Antoni, Hakan Memisoglu, and Owen Levin for their insights. This work is supported by the National Science Foundation CCF under awards 1566015 and 1652140.

References

Abadi, M.; Barham, P.; Chen, J.; Chen, Z.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Irving, G.; Isard, M.; et al. 2016. Tensorflow: A system for large-scale machine learning. In *OSDI*.

Alur, R.; Singh, R.; Fisman, D.; and Solar-Lezama, A. 2018. Search-based program synthesis. *CACM* (12).

Ben-Tal, A.; El Ghaoui, L.; and Nemirovski, A. 2009. *Robust optimization*, volume 28. Princeton University Press.

Bošnjak, M.; Rocktäschel, T.; Naradowsky, J.; and Riedel, S. 2017. Programming with a differentiable forth interpreter. In *ICML*.

Bryce, D.; Gao, S.; Musliner, D.; and Goldman, R. 2015. SMT-based nonlinear PDDL+ planning. In *Workshops at the Twenty-Ninth AAAI Conference on Artificial Intelligence*.

Bunel, R.; Hausknecht, M.; Devlin, J.; Singh, R.; and Kohli, P. 2018. Leveraging grammar and reinforcement learning for neural program synthesis. In *ICLR*.

Carlini, N., and Wagner, D. 2017. Towards evaluating the robustness of neural networks. In *S&P*.

Cashmore, M.; Fox, M.; Long, D.; and Magazzeni, D. 2016. A compilation of the full PDDL+ language into SMT. In *Workshops at AAAI*.

Chen, X.; Liu, C.; and Song, D. 2019. Execution-guided neural program synthesis. In *ICLR*.

Coles, A. J.; Coles, A. I.; Fox, M.; and Long, D. 2012. Colin: Planning with continuous linear numeric change. *JAIR* 44:1–96.

Datta, A.; Sen, S.; and Zick, Y. 2016. Algorithmic transparency via quantitative input influence: Theory and experiments with learning systems. In *S&P*.

Dua, D., and Graff, C. 2017. UCI machine learning repository.

Fernández-González, E.; Williams, B.; and Karpas, E. 2018. Scottyactivity: mixed discrete-continuous planning with convex optimization. *JAIR* 62:579–664.

Gaunt, A. L.; Brockschmidt, M.; Singh, R.; Kushman, N.; Kohli, P.; Taylor, J.; and Tarlow, D. 2016. Terpret: A probabilistic programming language for program induction. *arXiv preprint arXiv:1608.04428*.

Gaunt, A. L.; Brockschmidt, M.; Kushman, N.; and Tarlow, D. 2017. Differentiable programs with neural libraries. In *ICML*.

Goodfellow, I. J.; Shlens, J.; and Szegedy, C. 2015. Explaining and harnessing adversarial examples. In *ICLR*.

Google. 2017. The quick, draw! dataset.

Gulwani, S.; Jha, S.; Tiwari, A.; and Venkatesan, R. 2011. Synthesis of loop-free programs. In *PLDI*.

Gulwani, S.; Polozov, O.; and Singh, R. 2017. Program synthesis. *Foundations and Trends® in Programming Languages* (1-2):1–119.

Hardt, M.; Megiddo, N.; Papadimitriou, C.; and Wootters, M. 2016. Strategic classification. In *ITCS*.

Helmert, M. 2002. Decidability and undecidability results for planning with numerical state variables. In *AIPS*.

Kingma, D. P., and Ba, J. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Li, Y.; Albarghouthi, A.; Kincaid, Z.; Gurfinkel, A.; and Chechik, M. 2014. Symbolic optimization with smt solvers. In *ACM SIGPLAN Notices*, number 1.

Lillicrap, T. P.; Hunt, J. J.; Pritzel, A.; Heess, N.; Erez, T.; Tassa, Y.; Silver, D.; and Wierstra, D. 2015. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.

Lundberg, S. M., and Lee, S.-I. 2017. A unified approach to interpreting model predictions. In *NeurIPS*.

Madry, A.; Makelov, A.; Schmidt, L.; Tsipras, D.; and Vladu, A. 2018. Towards deep learning models resistant to adversarial attacks. In *ICLR*.

Mae, F. 2014. Fannie Mae single-family loan performance data. http://www.fanniemae.com/portal/funding-the-market/data/loan-performance-data.html.

Papernot, N.; McDaniel, P.; Goodfellow, I.; Jha, S.; Celik, Z. B.; and Swami, A. 2017. Practical black-box attacks against machine learning. In *ACCS*.

Parisotto, E.; Mohamed, A.-r.; Singh, R.; Li, L.; Zhou, D.; and Kohli, P. 2017. Neuro-symbolic program synthesis.

Piotrowski, W. M.; Fox, M.; Long, D.; Magazzeni, D.; and Mercorio, F. 2016. Heuristic planning for hybrid systems. In *Thirtieth AAAI Conference on Artificial Intelligence*.

Reed, S., and De Freitas, N. 2015. Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279*.

Ribeiro, M. T.; Singh, S.; and Guestrin, C. 2016. Why should i trust you?: Explaining the predictions of any classifier. In *KDD*.

Solar-Lezama, A.; Tancau, L.; Bodik, R.; Seshia, S.; and Saraswat, V. 2006. Combinatorial sketching for finite programs. *ACM Sigplan Notices* (11).

Sundararajan, M.; Taly, A.; and Yan, Q. 2017. Axiomatic attribution for deep networks. In *ICML*.

Szegedy, C.; Zaremba, W.; Sutskever, I.; Bruna, J.; Erhan, D.; Goodfellow, I.; and Fergus, R. 2013. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*.

Ustun, B.; Spangher, A.; and Liu, Y. 2019. Actionable recourse in linear classification. In *FAT**.

Valkov, L.; Chaudhari, D.; Srivastava, A.; Sutton, C.; and Chaudhuri, S. 2018. Houdini: Lifelong learning as program synthesis. In *NeurIPS*.

Wachter, S.; Mittelstadt, B.; and Russell, C. 2017. Counterfactual explanations without opening the black box: Automated decisions and the GDPR. *Harvard Journal of Law & Technology* (2).

Wu, G.; Say, B.; and Sanner, S. 2017. Scalable planning with tensorflow for hybrid nonlinear domains. In *NeurIPS*.

Zhang, X.; Solar-Lezama, A.; and Singh, R. 2018. Interpreting neural network judgments via minimal, stable, and symbolic corrections. In *NeurIPS*.