GUIDING PROGRAM SYNTHESIS BY LEARNING TO GENERATE EXAMPLES

Anonymous authors
Paper under double-blind review

ABSTRACT

A key challenge of existing program synthesizers is ensuring that the synthesized program generalizes well. This can be difficult to achieve as the specification provided by the end user is often limited, containing as few as one or two input-output examples. In this paper we address this challenge via an iterative approach that finds ambiguities in the provided specification and learns to resolve these by generating additional input-output examples. The main insight is to reduce the problem of selecting which program generalizes well to the simpler task of deciding which output is correct. As a result, to train our probabilistic models, we can take advantage of the large amounts of data in the form of program outputs, which are often much easier to obtain than the corresponding ground-truth programs.

1 INTRODUCTION

Over the years, program synthesis has been successfully used to solve a range of tasks including string, number or date transformations (Gulwani, 2011; Singh & Gulwani, 2012; 2016), layout and graphic program generation (Bielik et al., 2018; Hempel & Chugh, 2016), data extraction (Barowy et al., 2014; Le & Gulwani, 2014), superoptimization (Phothilimthana et al., 2016; Schkufza et al., 2016) or code repair (Singh et al., 2013; Nguyen et al., 2013; D'Antoni et al., 2016). To capture user intent in an easy and intuitive way, many program synthesizers let its users provide a set of input-output examples $\mathcal{I}$ which the synthesized program should satisfy.

Generalization challenge A natural expectation of the end user in this setting is that the synthesized program works well even when $\mathcal{I}$ is severely limited (e.g., to one or two examples). This makes the synthesis problem difficult as the limited amount of supervision provided by the user combined with the rich hypothesis space over which programs are searched over, often leads to millions of programs consistent with $\mathcal{I}$, with only a small number generalizing well to unseen examples.

Existing methods Several approaches have provided ways to address the above challenge, including using an external model that *learns to rank* candidate programs returned by the synthesizer, modifying the search procedure by *learning to guide* the synthesizer such that it returns more likely programs directly, or *neural program induction* methods that replace the synthesizer with a neural network to generate outputs directly using a latent program representation. However, regardless of what other features these approaches use, such as conditioning on program traces (Shin et al., 2018; Ellis & Gulwani, 2017; Chen et al., 2019) or pre-training on the input data (Singh, 2016), they are limited by the fact that their models are conditioned on the initial, limited user specification.

This work We present a new approach for program synthesis from examples which addresses the above challenge. The key idea is to resolve ambiguity by iteratively strengthening the initial specification $\mathcal{I}$ with new examples. To achieve this, we start by using an existing synthesizer to find a candidate program p_1 that satisfies all examples in $\mathcal{I}$. Instead of returning p_1, we use it to find a distinguishing input x^* that leads to ambiguities, i.e., other programs p_i that satisfy $\mathcal{I}$ but produce different outputs $p_1(x^*) \neq p_i(x^*)$. To resolve this ambiguity, we first generate a set of candidate outputs for x^*, then use a neural model (which we train beforehand) that acts as an oracle and selects the most likely output, and finally, add x^* and its predicted output to the input specification $\mathcal{I}$. The whole process is then repeated. These steps are similar to those used in Oracle Guided Inductive Synthesis (Jha et al., 2010) with two main differences: *(i)* we automate the entire process by learning the oracle from data instead of using a human oracle, and *(ii)* as we do not use a human oracle to produce a correct output, we need to ensure that the set of candidate outputs contains the correct one.

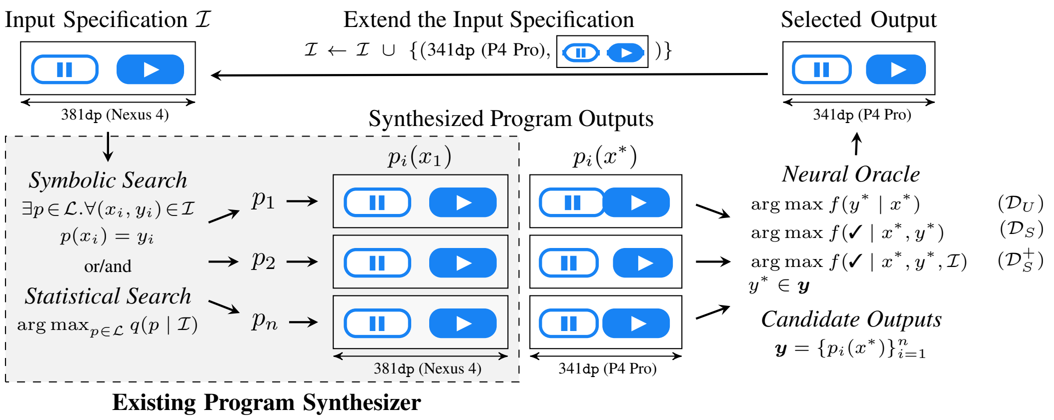

Input Specification $\mathcal{I}$
381dp (Nexus 4)
Extend the Input Specification
$\mathcal{I} \leftarrow \mathcal{I} \cup \{(341\text{dp (P4 Pro)}, \ldots)\}$
Selected Output
341dp (P4 Pro)
Synthesized Program Outputs
Symbolic Search
$\exists p \in \mathcal{L}.\forall(x_i, y_i) \in \mathcal{I}$
$p(x_i) = y_i$
or/and
Statistical Search
$\arg\max_{p \in \mathcal{L}} q(p \mid \mathcal{I})$
p_1
p_2
p_n
$p_i(x_1)$
$p_i(x^*)$
381dp (Nexus 4)
341dp (P4 Pro)
Neural Oracle
$\arg\max f(y^* \mid x^*)$ $(\mathcal{D}_U)$
$\arg\max f(✓ \mid x^*, y^*)$ $(\mathcal{D}_S)$
$\arg\max f(✓ \mid x^*, y^*, \mathcal{I})$ $(\mathcal{D}_S^+)$
$y^* \in \boldsymbol{y}$
Candidate Outputs
$\boldsymbol{y} = \{p_i(x^*)\}_{i=1}^n$
Existing Program Synthesizer

Figure 1: An overview of our approach which introduces a refinement loop around a black-box synthesizer that incrementally extends the input specification $\mathcal{I} = \{(x_1, y_1)\}$, which in this case contains a single example, with additional input-output examples until all ambiguities are resolved.

Example of augmenting an existing synthesizer As a concrete example, Figure 1 illustrates our approach instantiated with a state-of-the-art synthesizer, called `InferUI` (Bielik et al., 2018), that creates an Android layout program which when rendered, places all components at their specified position. Here, each input-output example (x, y) consists of a set of views and their absolute view positions y on device x. In our example, the input specification $\mathcal{I}$ contains a single example with absolute view positions for a Nexus 4 device and the `InferUI` synthesizer easily finds multiple programs that satisfy it (dashed box). To apply our method and resolve the ambiguity, we find a distinguishing input x^*, in this case a narrower P4 Pro device, on which some of the candidate programs produce different outputs. Then, instead of asking the user to manually produce the correct output, we generate additional candidate outputs (to ensure that the correct one is included) and our learned neural oracle automatically selects one of these outputs (the one it believes is correct) and adds it to the input specification $\mathcal{I}$. In this case, the oracle selects output $p_2(x^*)$ as both buttons are correctly resized and the distance between them was reduced to match the smaller device width. In contrast, $p_1(x^*)$ contains overlapping buttons while in $p_n(x^*)$, only the left button was resized.

Automatically obtaining real-world datasets An important advantage of our approach is that we reduce the problem of selecting which program generalizes well to the simpler task of deciding which output is correct. As a result, obtaining a suitable training dataset can be easier as we do not require the hard to obtain ground-truth programs, for which currently no large real-world datasets exists (Shin et al., 2019). In fact, it is possible to train the oracle using unsupervised learning only, with a dataset consisting of correct input-output examples $\mathcal{D}_U$. For example, in layout synthesis an autoencoder can be trained over a large number of views and their positions extracted from running real-world applications, while for string processing tasks we can train a language model over the rows or columns of the Excel spreadsheet. However, instead of training such an unsupervised model, in our work we use $\mathcal{D}_U$ to automatically construct a supervised dataset $\mathcal{D}_S$ by labelling the samples in $\mathcal{D}_U$ as positive and generating a set of negative samples by adding suitable noise to the samples in $\mathcal{D}_U$. Finally, we also obtain the dataset $\mathcal{D}_{S+}$ that additionally includes the input specification $\mathcal{I}$. In the domain of Android layouts, although it is more difficult, such a dataset can also be collected automatically by running the same application on devices with different screen sizes.

Our contributions We present a new approach to address the ambiguity in existing program synthesizers by iteratively extending the user provided specification with new input-output examples. The key component of our method is a learned neural oracle used to generate new examples trained with datasets that do not require human annotations or ground-truth programs. We evaluate our approach by applying it to an existing Android layout synthesizer `InferUI`. To improve generalization, `InferUI` already contains a probabilistic model that scores programs $q(p \mid \mathcal{I})$ as well as a set of handcrafted robustness properties, achieving 35% generalization accuracy on a dataset of Google Play Store applications. We show that our method leads to a significant improvement and achieves 71% accuracy even when both optimizations of `InferUI` are disabled. This result suggests that our method is a promising step in increasing generalization capabilities of existing synthesizers.

2 RELATED WORK

In this section we discuss the work most closely related to ours.

Guiding program synthesis To improve scalability and generalization of program synthesizers several techniques have been proposed that guide the synthesizer towards good programs. The most widely used approach is to implement a statistical search procedure which explores candidate programs based on some type of learned probabilistic model – log-linear models (Menon et al., 2013; Long & Rinard, 2016), hierarchical Bayesian prior (Liang et al., 2010), probabilistic higher order grammar (Lee et al., 2018) or neural network (Balog et al., 2017). Kalyan et al. (2018) also takes advantage of probabilistic models but instead of implementing a custom search procedure, they use the learned model to guide an existing symbolic search engine. In addition to approaches that search for a good program directly (conditioned on the input specification), a number of works guide the search by first selecting a high-level sketch of the program and then filling in the holes using symbolic (Ellis et al., 2018; Murali et al., 2017; Nye et al., 2019), enumerative or neural search (Bosnjak et al., 2017; Gaunt et al., 2016). A similar idea is also used by (Shin et al., 2018), but instead of generating a program sketch the authors first infer execution traces (or condition on partial traces obtained as the program is being generated (Chen et al., 2019)), which are then used to guide the synthesis of the actual program.

In comparison to prior work, a key aspect of our approach is to guide the synthesis by generating additional input-output examples that resolve the ambiguities in the input specification. Guiding the synthesizer in this way has several advantages – *(i)* it is interpretable and the user can inspect the generated examples, *(ii)* it can be used to extend any existing synthesizer by introducing a refinement loop around it, *(iii)* the learned model is independent of the actual synthesizer (and its domain specific language) and instead is focused only on learning the relation between likely and unlikely input-output examples, and *(iv)* often it is easier to obtain a dataset containing program outputs instead of a dataset consisting of the actual programs. Further, our approach is complementary to prior works as it treats the synthesizer as a black-box that can generate candidate programs.

Learning to rank To choose among all programs that satisfy the input specification, existing program synthesizers select the syntactically shortest program (Liang et al., 2010; Polozov & Gulwani, 2015; Raychev et al., 2016), the semantically closest program to a reference program (D'Antoni et al., 2016) or a program based on a learned scoring function (Liang et al., 2010; Mandelin et al., 2005; Singh & Gulwani, 2015; Ellis & Gulwani, 2017; Singh, 2016). Although the scoring function usually extracts features only from the synthesized program, some approaches also take advantage of additional information – Ellis & Gulwani (2017) trains a log-linear model using a set of handcrafted features defined over program traces and program outputs while Singh (2016) leverages unlabelled data by learning common substring expressions shared across the input data.

Similar to prior work, our work explores various representations over which the model is learned. Because we applied our work to a domain where outputs can be represented as images (rather than strings or numbers), to achieve good performance we explore different types of models (i.e., convolutional neural networks). Further, we do not assume that the synthesizer can efficiently enumerate all programs that satisfy the input specification as in Ellis & Gulwani (2017); Singh (2016). For such synthesizers, applying a ranking of the returned candidates will often fail since the correct program is simply not included in the set of synthesized programs. Therefore, the neural oracle is defined over program outputs instead of actual programs. This reduces the search space for the synthesizer as well as the complexity of the machine learning models.

Neural program induction Devlin et al. (2017) and Parisotto et al. (2017) explore the design of end-to-end neural approaches that generate the program output for a new input without the need for an explicit search. In this case the goal of the neural network is not to find the correct program explicitly, but rather to generate the most likely output for a given input based on the input specification. These approaches can be integrated in our work as one technique for generating a set of candidate outputs for a given distinguishing input instead of obtaining them using a symbolic synthesizer. However, the model requirements in our work are much weaker – whereas the goal of program induction is to achieve 100% precision for all possible inputs (as otherwise the network is producing mistakes), in our work it is enough if the correct output is among the top n most likely candidates.

3 LEARNING TO GENERATE NEW INPUT-OUTPUT EXAMPLES

Let $\mathcal{I} = \{(x_i, y_i)\}_{i=1}^{N}$ denote the input specification consisting of input-output examples provided by the user. Further, we assume we are given an existing synthesizer which can find a program p that satisfies all examples in $\mathcal{I}$, i.e., $\exists p \in \mathcal{L}, \forall (x_i, y_i) \in \mathcal{I}.\ p(x_i) = y_i$, where $p(x_i)$ denotes the output obtained by running program p on input x_i. To reduce clutter, we use the notation $p \models \mathcal{I}$ to denote that program p satisfies all examples in $\mathcal{I}$. We extend the synthesizer such that a program p not only satisfies all examples in $\mathcal{I}$ but also generalizes well to unseen examples as follows:

1. Generate a candidate program $p_1 \models \mathcal{I}$ that satisfies the input specification $\mathcal{I}$.
2. Find a *distinguishing input* x^* and candidate outputs $\boldsymbol{y} = [p_1(x^*); p_2(x^*); \cdots ; p_n(x^*)]$, such that all programs $p_1, \ldots, p_n$ satisfy the input specification $\mathcal{I}$ but produce different outputs when evaluated on x^*. If no distinguishing input x^* exists, return program p_1.
3. Query an oracle to determine the correct output $y^* \in \boldsymbol{y}$ for the input x^*.
4. Extend the input specification with the *distinguishing input* and its corresponding output $\mathcal{I} \leftarrow \mathcal{I} \cup \{(x^*, y^*)\}$ and continue with the first step.

Finding a distinguishing input To find the distinguishing input x^* we take advantage of existing symbolic synthesizers by asking the synthesizer to solve $\exists x^* \in \mathcal{X}, p_2 \in \mathcal{L}.\ p_2 \models \mathcal{I} \wedge p_2(x^*) \neq p_1(x^*)$, where $\mathcal{X}$ is a set of valid inputs and $\mathcal{L}$ is a hypothesis space of valid programs. The result is both a distinguishing input x^* as well as a program p_2 that produces a different output than p_1. Programs p_1 and p_2 form the initial sequence of candidate outputs $\boldsymbol{y} = [p_1(x^*); p_2(x^*)]$ which is extended until the oracle is confident enough that $\boldsymbol{y}$ contains the correct output (described later in this section).

To make our approach applicable to any existing synthesizer, including those that can not solve the above satisfiability query directly (e.g., statistical synthesizers), we note that the following sampling approach can also be applied: first, use the synthesizer to generate the top n most likely programs, then randomly sample a valid input x^* not in the input specification, and finally check if that input leads to ambiguities by computing the output of all candidate programs.

Finding candidate outputs To extend $\boldsymbol{y}$ with additional candidate outputs once the distinguishing input x^* is found, three techniques can be applied: *(i)* querying the synthesizer for another program with a different output: $\exists p \in \mathcal{L}.\ p \models \mathcal{I} \wedge \forall_{y_i \in \boldsymbol{y}} p(x^*) \neq y_i$, *(ii)* sampling a program induction model $P(y \mid x^*, \mathcal{I})$ and using the synthesizer to check whether each sampled output is valid, or *(iii)* simply sampling more candidate programs, running them on x^* and keeping the unique outputs. It is possible to use the second approach as we are only interested in the set of different outputs, rather than the actual programs. The advantage of *(i)* is that it is simple to implement for symbolic synthesizers and is guaranteed to find different outputs if they exist. In contrast, *(ii)* has the potential to be faster as it avoids calling the synthesizer and works for both statistical and symbolic synthesizers. Finally, *(iii)* is least effective, but it does not require pretraining and can be applied to any synthesizer.

Neural oracle The key component of our approach is a neural oracle which selects the correct program output from a set of candidate outputs $\boldsymbol{y}$. Formally, the neural oracle is defined as $\arg\max_{y^* \in \boldsymbol{y}} f_\theta(\checkmark \mid x^*, y^*, \mathcal{I})$, where f is a function with learnable parameters θ (in our case a neural network) that returns the probability of the output y^* being correct given the input x^* and the input specification $\mathcal{I}$. We train the parameters θ using a supervised dataset $\mathcal{D}_{S+} = \{(\checkmark, x_i, y_i, \mathcal{I}_i)\}_{i=1}^{N} \cup \{(\text{✗}, x_j, y_j, \mathcal{I}_j)\}_{j=1}^{M}$ which, for a given distinguishing input x^* and input specification $\mathcal{I}$, contains both the correct ($\checkmark$) as well as the incorrect (✗) outputs. Because it might difficult to obtain such a dataset in practice, we also define a simpler model $f_\theta(\checkmark \mid x^*, y^*)$ that is trained using a supervised dataset $\mathcal{D}_S = \{(\checkmark, x_i, y_i)\}_{i=1}^{N} \cup \{(\text{✗}, x_j, y_j)\}_{j=1}^{M}$ which does not include the input specification. In the extreme case, where the dataset contains only the correct input-output examples $\mathcal{D}_U = \{(x_i, y_i)\}_{i=1}^{N}$, we define the oracle as $f_\theta(y^* \mid x^*)$. Even though the dataset does not contain any labels, we can still train f in an unsupervised manner. This can be achieved for example by splitting the output into smaller parts $y_i = y_i^1 \cdots y_i^t$ (such as splitting a word into characters) and training f as an unsupervised language model. Alternatively, we could also train an autoencoder that first compresses the output into a lower dimensional representation, with the loss corresponding to how well it can be reconstructed. To achieve good performance, the architecture used to represent f is tailored to the synthesis domain at hand, as discussed in the next section.

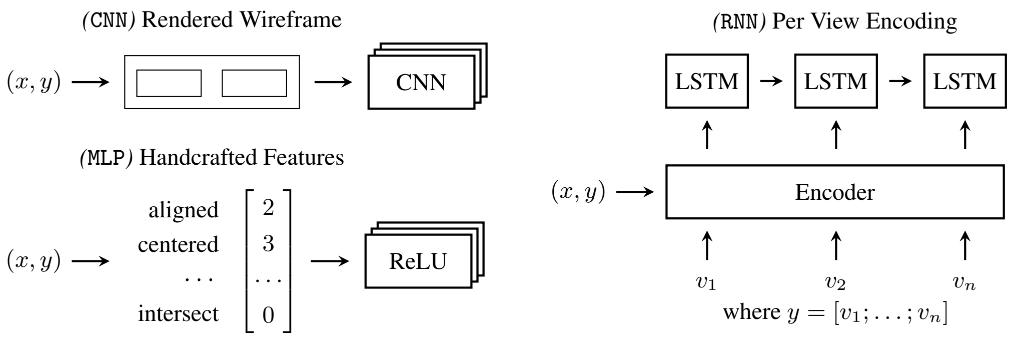


Figure 2: Illustration of three different models used to implement the neural oracle – `CNN`, `RNN` and `MLP`. All models include a fully-connected ReLU layer and a softmax layer (not shown in the image) which computes the probability that the output y is correct. Note, that the features in the `MLP` model are shown unnormalized, that is, `centered` $=$ `3` denotes that three views are centered.

Dynamically controlling the number of generated candidate outputs Since generating a large number of candidate outputs is time consuming, we allow our models to dynamically control the number of sampled candidates for each distinguishing input x^*. That is, instead of generating all candidate outputs $\boldsymbol{y}$ first and only then querying the neural oracle to select the most likely one, we query the oracle after each generated candidate output and let the model decide whether more candidates should be generated. Concretely, we define a threshold hyperparameter $t \in \mathbb{R}^{[0,1]}$ which is used to return the first candidate output y^* for which the probability of the output being correct is above this threshold. Then, only if there are no candidate outputs with probability higher that t, we return $\arg\max$ of all the candidates. Note for $t = 1$ this formulation is equivalent to returning the $\arg\max$ of all the candidates while for $t = 0$, it corresponds to always returning the first candidate, regardless of its probability of being correct. We show the effect of different threshold values as well as the number of generated candidate outputs in Section 5.

4 Instantiation of our approach to Android layout synthesis

In this section we describe how to apply our approach to the existing Android layout program synthesizer `InferUI` (Bielik et al., 2018). Here, the input $x \in \mathbb{R}^4$ defines the absolute positions of the given device screen while the output $y \in \mathbb{R}^{n\times4}$ consists of n views and their absolute positions.

Finding a distinguishing input and candidate outputs Because `InferUI` uses symbolic search, finding a distinguishing input and candidate outputs is encoded as a logical query solved by the synthesizer as described in Section 3. However, instead of synthesizing the layout program containing correct outputs of all the views at once (as done in `InferUI`), we run the synthesizer n times, each time predicting the correct output for only a single view (starting from the largest view) which is then added as an additional input-output example. This is necessary since there are exponentially many combinations of the view positions when considering all the views at once and the `InferUI` synthesizer is not powerful enough to include the correct one in the set of candidate outputs (e.g., for samples with more than 10 views in less than 4%). The advantage of allowing the position of only a single view to change, while fixing the position of all prior views, is that the correct candidate output is much easier to include. The disadvantage is that the neural oracle only has access to partial information (consisting of the prior view positions) and therefore performs a sequence of greedy predictions rather than optimizing all view positions jointly.

Neural oracle Because the input x has the same dimensions as each view, we encode it as an additional view in all our network architectures. In Figure 2 we show three different neural architectures that implement the oracle function f, each of which uses a different way to encode the input-output example into a hidden representation followed by a fully-connected ReLU layer and a softmax that computes the probability that the output is correct. In the following, we describe the architecture of all the models. The formal feature definitions are included in Appendix A.

In *(*`CNN`*)*, the output is converted to an image (each view is drawn as a rectangle with a 1px black border on a white background) and used as an input to a convolutional neural network (CNN) with 3 convolutional layers with 5×5 filters of size 64, 32 and 16 and max pooling with kernel size 2 and stride 2. To support outputs computed for different screen sizes, the image dimensions are slightly larger than the largest device size. We regularize the network during training by positioning the outputs with a random offset such that they are still fully inside the image, instead of placing them in the center. This is possible as the image used as input to the CNN is larger than the device size.

In *(*`MLP`*)*, the output is transformed to a normalized feature vector and then fed into a feedforward neural network with 3 hidden layers of size 512 with ReLU activations. To encode the properties of a candidate output y, we use high-level handcrafted features adapted from `InferUI` such as the number of view intersections or whether the views are aligned or centered. By instantiating the features for both horizontal and vertical orientation we obtain a vector of size 30, denoted as $\varphi_{\texttt{MLP}}(x^*, y^*)$, which is used as input to the neural network. For the model $f(\checkmark \mid x^*, y^*, \mathcal{I})$ the network input is a concatenation of output features (as before) $\varphi_{\texttt{MLP}}(x^*, y^*)$, features of each sample in the input specification $[\varphi_{\texttt{MLP}}(x, y)]_{(x,y)\in\mathcal{I}}$, and their difference $[\varphi_{\texttt{MLP}}(x^*, y^*) - \varphi_{\texttt{MLP}}(x, y)]_{(x,y)\in\mathcal{I}}$. This difference captures how the views have been resized or moved between the devices with different screen dimensions. This allows the model to distinguish between outputs that are all likely when considered in isolation but not when also compared to examples in $\mathcal{I}$.

In *(*`RNN`*)*, we use the fact that each output consists of a set of views $y^* = [v_1; \dots; v_n]$ by using an encoder to first compute a hidden representation of each view. These are then combined with a LSTM to compute the representation of the whole output. To encode a view v_i, we extract pairwise feature vectors with all other views (including the input) $\varphi_{\texttt{RNN}}(v_i, x^*, y^*) = [\phi(v_i, v_j)]_{v_j\in\{x^*\}\cup y^*\setminus v_i}$ and combine them using a LSTM. Here, $\phi : \mathbb{R}^4 \times \mathbb{R}^4 \to \mathbb{R}^n$ is a function that extracts n real valued features from a pair of views. For each pair of views, we apply 11 simple transformations of the view coordinates, capturing their distance (4 vertical, 4 horizontal), the size difference (in width and height) and the ratio of the aspect ratios. For the model $f(\checkmark \mid x^*, y^*)$ we additionally use 17 high-level features computed for each view that are adapted from `InferUI`. When using $f(\checkmark \mid x^*, y^*, \mathcal{I})$, these additional high-level features are not required and instead we only use the 11 simple transformations combined in the same way as for the `MLP` model. That is, by concatenating $\varphi_{\texttt{RNN}}(v_i, x^*, y^*)$, $[\varphi_{\texttt{RNN}}(v_i, x, y)]_{(x,y)\in\mathcal{I}}$ and their difference $[\varphi_{\texttt{RNN}}(v_i, x^*, y^*) - \varphi_{\texttt{RNN}}(v_i, x, y)]_{(x,y)\in\mathcal{I}}$.

Datasets To train our models we obtained three datasets $\mathcal{D}_U$, $\mathcal{D}_S$ and $\mathcal{D}_S^+$, each containing an increasing amount of information at the expense of being harder to collect.

The unsupervised $\mathcal{D}_U = \{(x_i, y_i)\}_{i=1}^N$ is the simplest dataset and contains only positive input-output samples obtained by sampling $\approx 22,000$ unique screenshots (including the associated metadata of all the absolute view positions) of Google Play Store applications taken from the `Rico` dataset (Deka et al., 2017). Since the screenshots always consist of multiple layout programs combined together, we approximate the individual programs by sorting the views in decreasing order of their size and taking a prefix of random length (of up to 30 views). For all of the datasets, we deduplicate the views that have the same coordinates and filter out views with a negative width or height.

The supervised $\mathcal{D}_S = \{(\checkmark, x_i, y_i)\}_{(x_i,y_i)\in\mathcal{D}_U} \cup \big\{ \bigcup_{j=1}^{\leq 15}\{(\text{✗}, x_i, y_i + \epsilon_{ij})\}\big\}_{(x_i,y_i)\in\mathcal{D}_U}$ contains both correct and incorrect input-output examples. In our work this dataset is produced synthetically from $\mathcal{D}_U$ by extending it with incorrect samples. Concretely, the positive samples correspond to those in the dataset $\mathcal{D}_U$ and for each positive sample we generate up to 15 negative samples by applying a transformation ϵ_{ij} to the correct output. The transformations considered in our work are sampled from the common mistakes the synthesizer can make – resizing a view, shifting a view horizontally, shifting a view vertically or any combination of the above.

The supervised dataset is $\mathcal{D}_{S+} = \{(\checkmark, x_i, y_i, \mathcal{I}_i)\}_{i=1}^N \cup \{(\text{✗}, x_j, y_j, \mathcal{I}_j)\}_{j=1}^M$, where each $\mathcal{I}_i$ contains the same application rendered on multiple devices. We downloaded the same applications as used in the `Rico` dataset from the Google Play Store and executed them on three Android emulators with different device sizes. The number of valid samples is ≈ 600 since not all applications could be downloaded, executed or produced the same set of views (or screen content) when executed on three different devices. The negative examples are generated by running the synthesizer with the input specification $\mathcal{I}$ containing a single sample and selecting up to 16 outputs that are inconsistent with the ground-truth output for the other devices.

Table 1: Generalization accuracy of the existing `InferUI` synthesizer.

`InferUI` (Bielik et al., 2018)	$p \models \mathcal{I}$ *baseline*	$\arg\max_{p \models \mathcal{I}} q(p \mid \mathcal{I})$ *+ probabilistic model*	$\arg\max_{p \models \mathcal{I} \wedge \phi(p)} q(p \mid \mathcal{I})$ *+ probabilistic & robustness model*
Accuracy	15.5%	24.7%	35.2%

Table 2: Generalization accuracy of different models used as neural oracle in our approach.

		Accuracy			
Training Dataset	Model	MLP	CNN	RNN	RNN + CNN
$\mathcal{D}_S$	$f(\checkmark \mid x^*, y^*)$	14.3%	14.2%	23.8%	**32.1%**
$\mathcal{D}_{S+}$	$f(\checkmark \mid x^*, y^*, \mathcal{I})$	20.7%	33.2%	63.4%	**71.0%**

5 Evaluation

We evaluate our approach by applying it to an existing Android layout synthesizer called `InferUI` (Bielik et al., 2018) as described in Section 4. `InferUI` is a symbolic synthesizer which encodes the synthesis problem as a set of logical constraints that are solved using the state-of-the-art SMT solver Z3 (De Moura & Bjørner, 2008). To improve generalization, `InferUI` already implements two techniques – a probabilistic model that selects the most likely program among those that satisfy the input specification, and a set of handcrafted robustness constraints $\phi(p)$ that prevent synthesizing layouts which violate good design practices. We show that even if we disable these two optimizations and instead guide the synthesizer purely by extending the input specification with additional input-output examples, we can still achieve an accuracy increase from 35% to 71%.

In all our experiments, we evaluate our models and `InferUI` on a test subset of the $\mathcal{D}_{S+}$ dataset which contains 85 Google Play Store applications, each of which contains the ground truth of the absolute view positions on three different screen dimensions. We use one screen dimension as the input specification $\mathcal{I}$, the second as the distinguishing input and the third one only to compute the generalization accuracy. The generalization accuracy of a synthesized program $p \models \mathcal{I}$ is defined as the percentage of views which the program p renders at the correct position.

InferUI Baseline To establish a baseline, we run `InferUI` in three modes as shown in Table 1. The *baseline* mode returns the first program that satisfies the input specification, denoted as $p \models \mathcal{I}$, and achieves only 15.5% generalization accuracy. In the second mode the synthesizer returns the most likely program according to a *probabilistic model* $q(p \mid \mathcal{I})$ which leads to an improved accuracy of 24.7%. The third mode additionally defines a set of robustness properties $\phi(p)$ that the synthesized program needs to satisfy, which together with the probabilistic model achieve 35.2% accuracy. The generalization accuracy of all `InferUI` models is relatively low as we are using a challenging dataset where each sample contains on average 12 views. Note however, that this is expected since increasing the number of views leads to an exponentially larger hypothesis space.

Further, to establish an upper bound on how effective a candidate ranking approach can be, we query the synthesizer for up to 100 different candidate programs (each producing a unique output) and check how often the correct program is included. While for small samples (with up to 4 views) the correct program is almost always included, for samples with 6 views it is among the synthesized candidates in only 30% of the cases and for samples with more than 10 views in less than 4%. Sampling more outputs will help only slightly as increasing the number of views would require generating exponentially more candidates.

Our Work We apply our approach to the `InferUI` synthesizer by iteratively generating additional input-output examples that strengthen the input specification. The specification initially contains absolute positions of all the views for one device and we extend the specification by adding one view at a time (rendered on a different device) as described in Section 4.

Table 3: Effect of the threshold t and the maximum number of generated candidate outputs $|\boldsymbol{y}|$ on the accuracy and the average number of generated candidate outputs per view (shown in brackets).

Model		$t = 0.9$			$t = 1$
		$\lvert\boldsymbol{y}\rvert \leq 4$	$\lvert\boldsymbol{y}\rvert \leq 9$	$\lvert\boldsymbol{y}\rvert \leq 16$	$\lvert\boldsymbol{y}\rvert = 16$
$f(\checkmark \mid x^*, y^*, \mathcal{I})$	RNN + CNN	40.8% (2.8)	68.3% (4.7)	**71.3% (6.0)**	71.0% (14.7)

Generalization Accuracy The results of our approach instantiated with various neural oracle models are shown in Table 2. The best model trained on the dataset $\mathcal{D}_S$ has almost the same accuracy as InferUI with all its optimizations enabled. This means that it is possible to replace existing optimizations and handcrafted robustness constraints by training on an easy to obtain dataset consisting of correct outputs and their perturbations. More importantly, when training on the harder to obtain dataset $\mathcal{D}_{S+}$, the generalization accuracy more than doubles to 71% since the model can also condition on the input specification $\mathcal{I}$. However, the results also show that the design of the model using the neural oracle is important for achieving good results. In particular, both MLP models achieve poor accuracy since the high level handcrafted features adapted from the InferUI synthesizer are not expressive enough to distinguish between correct and incorrect outputs. The CNN models achieve better accuracy but are limited for the opposite reason, they try to learn all the relevant features from the raw pixels which is challenging as many features require pixel level accuracy across large distances (e.g., whether two views in the opposite parts of the screen are aligned or centered). The RNN model performs the best, especially when also having access to the input specification. Even though it also processes low level information, such as distance or size difference between the views, it uses a more structured representation that first computes individual view representations that are combined to capture the whole output.

Number of Candidate Outputs We show the effect of different threshold values t used by the neural oracle to dynamically control whether to search for more candidate outputs as well as the maximum number of candidate outputs in Table 3. We can see that using the threshold both slightly improves the accuracy ($+0.3\%$) but more importantly, significantly reduces the average number of generated candidate outputs from 14.7 to 6.0 using the same number of maximum generated outputs $|\boldsymbol{y}| = 16$.

Incorporating User Feedback Even though our approach significantly improves over the InferUI synthesizer, it does not achieve perfect generalization accuracy. This is because for many synthesizers the perfect generalization is usually not achievable – the correct program and its outputs depends on a user preference, which is only expressed as severely underspecified set of input-output examples. For example, for a given input specification there are often multiple good layout programs that do not violate any design guidelines and which one is chosen depends on a particular user. To achieve 100% in practice, we perform an experiment where the user can inspect the input-output examples generated by our approach and correct them if needed. Then, we simply count how many corrections were required. The applications in our dataset have on average 12 views and for our best model, no user corrections are required in 30% of the cases and in 27%, 15%, 12%, 5% of the cases the user needs to provide 1, 2, 3 or 4 corrections, respectively. In contrast, InferUI with all optimizations enabled requires on average twice as many user corrections and achieves perfect generalization (i.e., zero user corrections) in only 3.5% of the cases.

6 Conclusion

In this work we present a new approach to improve the generalization accuracy of existing program synthesizers. The main components of our method are: *(i)* an existing program synthesizer, *(ii)* a refinement loop around that synthesizer, which uses a neural oracle to iteratively extend the input specification with new input-output examples, and *(iii)* a neural oracle trained using an easy to obtain dataset consisting of program outputs. To show the practical usefulness of our approach we apply it to an existing Android layout synthesizer called InferUI (Bielik et al., 2018) and improve its generalization accuracy by $2\times$, from 35% to 71%, when evaluated on a challenging dataset of real-world Google Play Store applications.

REFERENCES

Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

Daniel W. Barowy, Sumit Gulwani, Ted Hart, and Ben Zorn. Flashrelate: Extracting relational data from semi-structured spreadsheets using examples. In *PLDI '15 Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Microsoft Research Technical Report, April 2014. Distinguished Artifact Award.

Pavol Bielik, Marc Fischer, and Martin Vechev. Robust relational layout synthesis from examples for android. *Proc. ACM Program. Lang.*, 2(OOPSLA):156:1–156:29, October 2018. ISSN 2475-1421. doi: 10.1145/3276526.

Matko Bosnjak, Tim Rocktäschel, Jason Naradowsky, and Sebastian Riedel. Programming with a differentiable forth interpreter. In Doina Precup and Yee Whye Teh (eds.), *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pp. 547–556. PMLR, 2017.

Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In *International Conference on Learning Representations*, 2019.

Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. Qlose: Program repair with quantiative objectives. In *27th International Conference on Computer Aided Verification (CAV 2016)*, July 2016.

Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pp. 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3-540-78799-2, 978-3-540-78799-0.

Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. Rico: A mobile app dataset for building data-driven design applications. In *Proceedings of the 30th Annual Symposium on User Interface Software and Technology*, UIST '17, 2017.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy I/O. In Doina Precup and Yee Whye Teh (eds.), *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pp. 990–998. PMLR, 2017.

Kevin Ellis and Sumit Gulwani. Learning to learn programs from examples: Going beyond program structure. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pp. 1638–1645, 2017. doi: 10.24963/ijcai.2017/227.

Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. Learning to infer graphics programs from hand-drawn images. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (eds.), *Advances in Neural Information Processing Systems 31*, pp. 6059–6068. Curran Associates, Inc., 2018.

Alexander L. Gaunt, Marc Brockschmidt, Rishabh Singh, Nate Kushman, Pushmeet Kohli, Jonathan Taylor, and Daniel Tarlow. Terpret: A probabilistic programming language for program induction. *CoRR*, abs/1608.04428, 2016.

Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In Thomas Ball and Mooly Sagiv (eds.), *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pp. 317–330. ACM, 2011. doi: 10.1145/1926385.1926423.

Brian Hempel and Ravi Chugh. Semi-automated svg programming via direct manipulation. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, UIST '16, pp. 379–390, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4189-9. doi: 10.1145/2984511.2984575.

Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pp. 215–224, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-719-6. doi: 10.1145/1806799.1806833.

Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. Neural-guided deductive search for real-time program synthesis from examples. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, 2018.

Vu Le and Sumit Gulwani. Flashextract: a framework for data extraction by examples. In Michael F. P. O'Boyle and Keshav Pingali (eds.), *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pp. 542–553. ACM, 2014. doi: 10.1145/2594291.2594333.

Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. Accelerating search-based program synthesis using learned probabilistic models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pp. 436–449, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5698-5. doi: 10.1145/3192366.3192410.

Percy Liang, Michael I. Jordan, and Dan Klein. Learning programs: A hierarchical bayesian approach. In Johannes Fürnkranz and Thorsten Joachims (eds.), *Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel*, pp. 639–646. Omnipress, 2010.

Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pp. 298–312, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837617.

David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: Helping to navigate the api jungle. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pp. 48–61, New York, NY, USA, 2005. ACM. ISBN 1-59593-056-6. doi: 10.1145/1065010.1065018.

Aditya Krishna Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Tauman Kalai. A machine learning framework for programming by example. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, pp. I–187–I–195. JMLR.org, 2013.

Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. Bayesian sketch learning for program synthesis. *CoRR*, abs/1703.05698, 2017.

Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pp. 772–781, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-3076-3.

Maxwell I. Nye, Luke B. Hewitt, Joshua B. Tenenbaum, and Armando Solar-Lezama. Learning to infer program sketches. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, pp. 4861–4870, 2019.

Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. Scaling up superoptimization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pp. 297–310, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4091-5. doi: 10.1145/2872362.2872387.

Oleksandr Polozov and Sumit Gulwani. Flashmeta: A framework for inductive program synthesis. In *OOPSLA 2015 Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 107–126, October 2015.

Veselin Raychev, Pavol Bielik, Martin T. Vechev, and Andreas Krause. Learning programs from noisy data. In Rastislav Bodík and Rupak Majumdar (eds.), *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pp. 761–774. ACM, 2016. doi: 10.1145/2837614.2837671.

Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic program optimization. *Commun. ACM*, 59(2):114–122, 2016. doi: 10.1145/2863701.

Richard Shin, Illia Polosukhin, and Dawn Song. Improving neural program synthesis with inferred execution traces. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (eds.), *Advances in Neural Information Processing Systems 31*, pp. 8917–8926. Curran Associates, Inc., 2018.

Richard Shin, Neel Kant, Kavi Gupta, Chris Bender, Brandon Trabucco, Rishabh Singh, and Dawn Song. Synthetic datasets for neural program synthesis. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

Rishabh Singh. Blinkfill: Semi-supervised programming by example for syntactic string transformations. *PVLDB*, 9(10):816–827, 2016. doi: 10.14778/2977797.2977807.

Rishabh Singh and Sumit Gulwani. Synthesizing number transformations from input-output examples. In P. Madhusudan and Sanjit A. Seshia (eds.), *Computer Aided Verification*, pp. 634–651, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-31424-7.

Rishabh Singh and Sumit Gulwani. Predicting a correct program in programming by example. In Daniel Kroening and Corina S. Păsăreanu (eds.), *Computer Aided Verification*, pp. 398–414, Cham, 2015. Springer International Publishing. ISBN 978-3-319-21690-4.

Rishabh Singh and Sumit Gulwani. Transforming spreadsheet data types using examples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pp. 343–356, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837668.

Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pp. 15–26, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2014-6. doi: 10.1145/2491956.2462195.

APPENDIX

We provide two appendices. Appendix A contains an in depth description of the feature transformations for the MLP and RNN models. Appendix B provides visualizations of positive and negative candidate outputs and the CNN regularization.

A FEATURE DEFINITIONS

In this section we describe in detail the feature transformations used in Section 4. As mentioned in Section 4 each view consists of the 4 coordinates: x_l (x_{left}), y_t (y_{top}), x_r (x_{right}), y_b (y_{bottom}). We use $v.w$ for the width, $v.h$ for the height and $v.r$ for the aspect ratio ($v.w/v.h$) of view v.

A.1 MLP

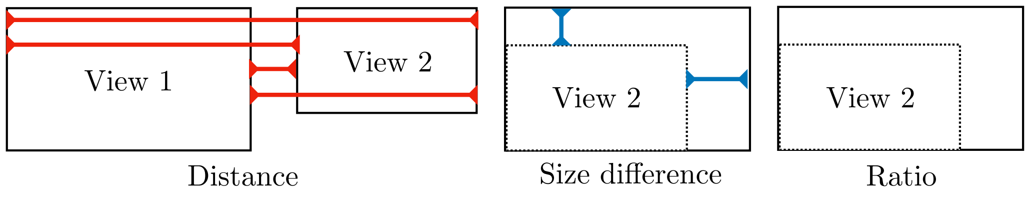
In Table 4 we define the 7 feature types that lead to a vector of size 30 used in the MLP model. All the features are normalized (divided) by the factor in the normalization column.

Table 4: Feature definitions for the MLP.

Size	Name	Description	Normalization
1	Number of off-screen views	-	$\|V\|$
3	Number of views with a specific aspect ratio	Instantiated for the ratios 1, [3/4, 4/3] and [9/16, 9/16]	$\|V\|$
1	Number of view intersections	Pairwise comparison if two views intersect (without counting fully contained views.)	$\|V\|^2$
1	Number of views which have the same dimension	The same dimension is defined as: $v_1.w = v_2.w \wedge v_1.h = v_2.h$	$\|V\|^2$
1 vertical + 1 horizontal	Number of view alignments	Pairwise comparison if two views align (e.g. for vertical alignment: $v_1.x_l = v_2.x_l \vee v_1.x_l = v_2.x_r \vee v_1.x_r = v_2.x_l \vee v_1.x_r = v_2.x_r$)	$\|V\|^2$
9 vertical + 9 horizontal	Number of views with a specific margin to another view	Instantiated for the margins 0, 16, 28, 32, 40, 48, 60, 64 and 96.	$\|V\|^2$
1 vertical + 1 horizontal	Number of centered views	Pairwise comparison if one view is centered within another view.	$\|V\|^2$
1 vertical + 1 horizontal	Number of centered views between two different views	Comparison if one view is centered between 2 other views.	$\|V\|^3$

A.2 RNN

We formally define the 11 transformations used as the pairwise view feature vector $\phi\colon \mathbb{R}^4 \times \mathbb{R}^4 \to \mathbb{R}^n$ in the RNN model. These features capture properties like the view's distance to the other view or the size difference. The feature vectors extracted for each pair of views are combined to a fixed length representation by passing them through the LSTM, in the decreasing order of view size. In Figure 3 all the properties are listed and visualized.



Name	Description	Formula
Distance	Horizontal distance from one view to another view	$d_{ll}(v_1, v_2) = v_1.x_l - v_2.x_l$ $d_{lr}(v_1, v_2) = v_1.x_l - v_2.x_r$ $d_{rl}(v_1, v_2) = v_1.x_r - v_2.x_l$ $d_{rr}(v_1, v_2) = v_1.x_r - v_2.x_r$
	Vertical distance from one view to another view	$d_{tt}(v_1, v_2) = v_1.y_t - v_2.y_t$ $d_{tb}(v_1, v_2) = v_1.y_t - v_2.y_b$ $d_{bt}(v_1, v_2) = v_1.y_b - v_2.y_t$ $d_{bb}(v_1, v_2) = v_1.y_b - v_2.y_b$
Size difference	Size difference of two views	$s_w(v_1, v_2) = v_1.w - v_2.w$ $s_h(v_1, v_2) = v_1.h - v_2.h$
Ratio	Relation of the aspect ratio	$r(v_1, v_2) = v_1.r / v_2.r$

Figure 3: Definition of the pairwise view feature vector $\phi \colon \mathbb{R}^4 \times \mathbb{R}^4 \rightarrow \mathbb{R}^n$ used in the RNN model (bottom) and their visualization (top).

To guide the model for $f(\checkmark \mid x^*, y^*)$, we defined 17 more abstract features, adapted from `InferUI`, which are defined using the simple transformations shown in Figure 3. Concretely, we define the following 17 features:

- For alignments (8 features: 4 horizontal, 4 vertical): Compare if view v_1 is aligned with v_2, such that one of the 4 distance functions is 0, e.g. $d_{ll}(v_1, v_2) = 0$
- For centering (2 features: 1 horizontal, 1 vertical): Compare if view v_1 is centered in v_2, such that $d_{ll}(v_1, v_2) = -d_{rr}(v_1, v_2)$.
- For overlaps (4 features: 2 horizontal, 2 vertical): Check if the v_1 and v_2 can possibly intersect, i.e. have overlapping x-coordinates in the horizontal case: $v_1.x_l \geq v_2.x_l$ and $v_1.x_l \leq v_2.x_r$.
- For the same size (2 features): Compare if the height or width difference of v_1 and v_2 is equal to 0, such that $s_w(v_1, v_2) = 0$ and $s_h(v_1, v_2) = 0$.
- For the same ratio (1 feature): Compare if the ratio v_1 and v_2's aspect ratios is 1, such that $r(v_1, v_2) = 1$.

B VISUALIZATION

Here, we provide visualizations of two concepts introduced in Section 4 – the rendered output candidates and how the CNN input is shifted for regularization.

B.1 POSITIVE AND NEGATIVE OUTPUT CANDIDATES

Figure 4 visualizes four different rendered output candidates. The correct candidate is on the left and the three incorrect ones are on the right. The candidates differ in their 6th view which is moved around and overlaps in all of the three incorrect candidates. The three negative examples are generated by the synthesizer as described in Section 4 (for dataset $\mathcal{D}_{S+}$) or applying perturbations sampled from the synthesizer mistakes (for dataset $\mathcal{D}_S$).

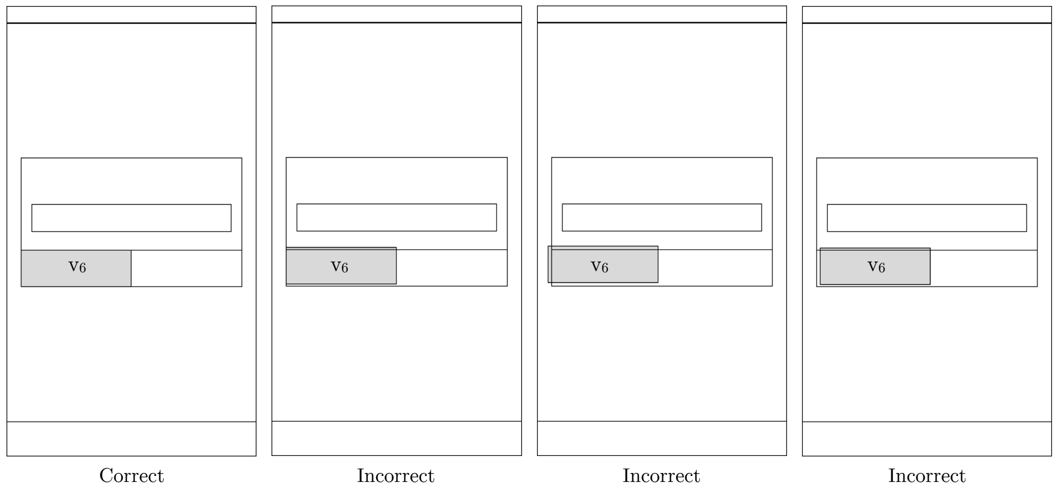


Figure 4: One positive example (left) and 3 negative examples (right).

B.2 CNN REGULARIZATION

Figure 5 visualizes how the robustness of the CNN is increased by placing the input randomly within the input image as described in Section 4.

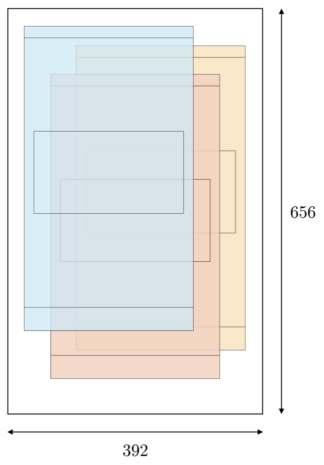


Figure 5: Illustration of three possible shifts (denoted by different colors) of the CNN input used during training.