PLAD: Learning to Infer Shape Programs with Pseudo-Labels and Approximate Distributions

R. Kenny Jones  
Brown University

Homer Walke  
UC Berkeley

Daniel Ritchie  
Brown University

Abstract

Inferring programs which generate 2D and 3D shapes is important for reverse engineering, editing, and more. Training models to perform this task is complicated because paired (shape, program) data is not readily available for many domains, making exact supervised learning infeasible. However, it is possible to get paired data by compromising the accuracy of either the assigned program labels or the shape distribution. Wake-sleep methods use samples from a generative model of shape programs to approximate the distribution of real shapes. In self-training, shapes are passed through a recognition model, which predicts programs that are treated as 'pseudo-labels' for those shapes. Related to these approaches, we introduce a novel self-training variant unique to program inference, where program pseudo-labels are paired with their executed output shapes, avoiding label mismatch at the cost of an approximate shape distribution. We propose to group these regimes under a single conceptual framework, where training is performed with maximum likelihood updates sourced from either Pseudo-Labels or an Approximate Distribution (PLAD). We evaluate these techniques on multiple 2D and 3D shape program inference domains. Compared with policy gradient reinforcement learning, we show that PLAD techniques infer more accurate shape programs and converge significantly faster. Finally, we propose to combine updates from different PLAD methods within the training of a single model, and find that this approach outperforms any individual technique.

1. Introduction

Having access to a procedure which generates a visual datum reveals its underlying structure, facilitating high-level manipulation and editing by a person or autonomous agent. Thus, inferring such programs from visual data is an important problem. In $\mathbb{R}^2$, inferring shape programs has applications in the design of diagrams, icons, and other 2D graphics. In $\mathbb{R}^3$, it has applications in reverse engineering of CAD models, procedural modeling for 3D games, and 3D structure understanding for autonomous agents.

We formally define shape program inference as obtaining a latent program $z$ which generates a given observed shape $x$. We model $p(z|x)$ with deep neural networks that train over a distribution of real shapes in order to amortize the cost of shape program inference on unseen shapes (e.g. a test set). This is a challenging problem: it is a structured prediction problem whose output is high-dimensional and can feature both discrete and continuous components (i.e. program control flow vs. program parameters). Nevertheless, learning $p(z|x)$ becomes tractable provided that one has access to paired $(X, Z)$ data (i.e. a dataset of shapes and the programs which generate them) [31]. Unfortunately, while shape data is increasingly available in large quantities [2], these shapes do not typically come with their generating program.

To circumvent this data problem, researchers have typically synthesized paired data by generating synthetic programs and pairing them with the shapes they output [24, 28]. However, as there is typically significant distributional mismatch between these synthetic shapes and “real” shapes from the distribution of interest, $S^*$, various techniques must be employed to fine-tune $p(z|x)$ models towards $S^*$.

A number of these fine-tuning strategies attempt to directly propagate gradients from geometric similarity measures back to $p(z|x)$. When the program executor is not a black-box, it may be possible to do this by implementing a differentiable relaxation of its behavior [16], but this requires knowledge of its functional form. One can also try learning a differentiable proxy of the executor’s behavior [28], but this approximation introduces errors. Moreover, as shape programs typically involve many discrete structural decisions, training such a model end-to-end is usually infeasible in many domains. Thus, many prior works often resort to general-purpose policy gradient reinforcement learning [8, 24], which treats the program executor as a (non-differentiable) black-box. The downside of this strategy is that RL is notoriously unstable and slow to converge.

In this paper, we study a collection of methods that create
we explore combining training updates from a mixture of maximum likelihood estimation (MLE) updates while treating the program executor as a black-box. As discussed, ground-truth (shape, program) pairs are often unavailable, so these techniques must make compromises in how they formulate paired data. In wake-sleep, a generative model \( p(z|x) \) is trained to convergence on alternating cycles with respect to \( p(z|x) \). When training \( p(z|x), \) paired data can be created by sampling from \( p(z) \). Each program label \( z \) is valid with respect to its associated \( x \) shape, but there is often a distributional mismatch between the generated set of shapes, \( X \), and shapes from the target distribution, \( S^* \). In self-training, one uses \( p(z|x) \) to infer latent \( z \)'s for unlabeled input \( x \)'s; these \( z \)'s then become “pseudo-labels” which are treated as ground truth for another round of supervised training. In this paradigm, there is no distributional shift between \( X \) and \( S^* \), but each \( z \) is only an approximately correctly label with respect to its paired \( x \).

We observe that shape program inference has a unique property that makes it especially well-suited for self-training: the distribution \( p(x|z) \) is known a priori—this is a delta distribution defined by the program executor. When using a model \( p(z|x) \) to infer a program \( z \) from some shape \( x \) of interest, one can use this executor to produce a shape \( x \) that is consistent with the program \( z \): in the terminology of self-training, \( z \) is guaranteed to be the “correct label” for \( x \). However, similar to wake-sleep, formulating \( X \) as shape executions produced by model inferred programs can cause a distributional shift between \( X \) and \( S^* \). Since this variant of self-training involves executing the inferred latent program \( z \), we call this procedure latent execution self-training (LEST).

As all of the aforementioned fine-tuning regimes use either Pseudo-Labels or Approximate Distributions to formulate (shape, program) pairs, we group them under a single conceptual framework: PLAD. We evaluate PLAD methods experimentally, using them to fine-tune shape program inference models in multiple shape domains: 2D and 3D constructive solid geometry (CSG), and assembly-based modeling with ShapeAssembly, a domain-specific language for structures of manufactured 3D objects [14]. We find that PLAD training regimes offer substantial advantages over the de-facto approach of policy gradient reinforcement learning, achieving better shape reconstruction performance while requiring significantly less computation time. Further, we explore combining training updates from a mixture of PLAD methods, and find that this approach leads to better performance compared with any individual method. Code for our method and experiments can be found at found at https://github.com/rkjones4/PLAD.

In summary, our contributions are:

1. Proposing the PLAD conceptual framework to group a family of related self-supervised learning techniques for shape program inference.
2. Introducing latent execution self-training, a PLAD method, to take advantage of the unique properties of the shape program inference problem.
3. Experiments across multiple 2D and 3D shape program inference domains, demonstrating that (i) fine-tuning under PLAD regimes outperforms policy gradient reinforcement learning and (ii) combining PLAD methods is better than any individual technique.

2. Related Work

Program synthesis is a broad field that has employed many techniques throughout its history. A program synthesizer takes as input a domain-specific language (DSL) and a specification; it outputs a program in the DSL that meets the specification. Machine learning has been used to improve performance on program synthesis tasks by e.g. performing a neurally-guided search over all possible programs or letting a recurrent network predict program text directly [1, 5, 7, 22, 27, 32].

In this work, we are interested in the sub-problem of shape program inference, which is a type of visual program induction problem [4]. In our case, the specification is a visual representation of an object which the inferred program’s output must geometrically match. Here, we discuss prior work that has attacked this problem, organized by methodology used to learn \( p(z|x) \); see Table 1 for an overview. As discussed, a common practice in this prior work is to start with a model that has been pretrained on synthetically generated (shape, program) pairs with supervised learning, and then perform fine-tuning towards a distribution of interest [8, 9, 19, 24, 28].

Policy Gradient Reinforcement Learning The most general method for fine-tuning a pretrained \( p(z|x) \) is reinforcement learning: treating \( p(z|x) \) as a policy network and using policy gradient methods [30]. The geometric similarity of the inferred program’s output to its input is the reward function; the program executor \( p(x|z) \) can be treated as a (non-differentiable) black-box. CSG-Net uses RL for fine-tuning [24, 25], as does other recent work on inferring CSG programs from input geometry [8]. While CSG-Net has been improved to allow it to converge without supervised pretraining [34], not starting from the supervised model results in worse performance. The main problem with policy gradient RL is its instability due to high variance gradients, leading to slow convergence. Like RL, PLAD methods treat the program executor as a black-box, but as we show experimentally, they converge faster and achieve better reconstruction performance.

Differentiable Executor If the functional form of the program executor \( p(x|z) \) is known and differentiable, then the gradient of the reward with respect to the parameters of \( p(z|x) \) can be computed, making policy gradient unnecessary. Shape programs are typically not fully differentiable, as

9872
they often involve discrete choices (e.g. which type of primitives to create). UCSGNet uses a differentiable relaxation to circumvent this issue [16]. Other work trains a differentiable network to approximate the behavior of the program executor [28], which introduces errors. PLAD regimes do not require the program executor to be differentiable, yet they perform better than other approaches (e.g. policy gradient RL) that share this desirable property.

**Generative Model Learning** Shape program inference has also been explored in the context of learning a generative model \( p(x, z) \) of programs and the shapes they produce. The most popular approach for training such models is variational Bayes, in particular the variational autoencoder [17]. This method simultaneously trains a generative model \( p(x, z) \) and a recognition model \( p(z|x) \) by optimizing a lower bound on the marginal likelihood \( p(x) \). When the \( z \)’s are shape programs, the program executor is \( p(x|z) \), so learning the generative model reduces to learning a prior over programs \( p(z) \). Training such models with gradient descent requires that the executor \( p(x|z) \) be differentiable. When this is not possible, the wake-sleep algorithm is a viable alternative [13]. This approach alternates training steps of the generative and recognition models, training one on samples produced by the other. Recent work has used wake-sleep for visual program induction [10, 12]. If one trains the generative model and the inference model to convergence before switching to training the other, this is equivalent to expectation maximization (viewed as alternating maximization [21]).

**Self-Training** Traditionally, self-training has been employed in weakly-supervised learning paradigms to increase the predictive accuracy of simple classification models [20, 23, 33]. Recently, renewed interest in self-training-inspired data augmentation approaches have demonstrated empirical performance improvements for neural models in domains such as large-scale image classification, machine translation, and speech recognition [11, 15, 35]. But while self-training has been shown to yield practical gains for some domains, for others it can actually lead to worse performance, as training on too many incorrect pseudo-labels can cause learning to degrade [3, 26]. For self-training within the PLAD framework, the assigned pseudo-label for each example changes during fine-tuning whenever the inference model discovers a program that better explains the input shape; similar techniques have been proposed for learning programs that perform semantic parsing under the view of iterative maximum likelihood [18]. To our knowledge, self-training has not been applied for fine-tuning visual program inference models, likely because it is somewhat unintuitive to view a program as a “label” for a visual datum.

### 3. Method

In this section, we describe the PLAD framework: a conceptual grouping of fine-tuning methods for shape program inference models. Our formulation assumes three inputs: a training dataset of shapes from the distribution of interest, \( S^* \), a program inference model, \( p(z|x) \), and a program executor that converts programs into shapes, \( E \). Throughout this paper, we assume that the \( p(z|x) \) passed as input has undergone supervised pretraining on a distribution of synthetically generated shapes. Methods within the PLAD framework return a fine-tuned \( p(z|x) \) specialized to the distribution of interest from which \( S^* \) was sampled.

We depict the PLAD procedure both algorithmically and pictorially in Figure 1. To fine-tune \( p(z|x) \) towards \( S^* \), PLAD methods iterate through the following steps: (1) use \( p(z|x) \) to find visually similar programs to \( S^* \), (2) construct a dataset of shape and program pairs \( (X, Z) \) using the inferred programs, and (3) fine-tune \( p(z|x) \) with maximum likelihood estimation updates on batches from \( (X, Z) \). Through successive iterations, these steps bootstrap one another, forming a virtuous cycle: improvements to \( p(z|x) \) create \( (X, Z) \) pairs that more closely match the statistics of \( S^* \), and training on better \( (X, Z) \) pairs specializes \( p(z|x) \) to \( S^* \).

Methods that fall within the PLAD framework differ in how the paired \((X, Z)\) data is created within each round. We detail this process for wake-sleep (Section 3.1), self-training (Section 3.2), and latent execution self-training (Section 3.3). In Section 3.4 we explain our program inference procedure.
3.1. Wake-Sleep (X, Z) Construction

Wake-sleep uses a generative model, \( p(z|x) \) to construct (X, Z). In our implementation, we choose to model \( p(z|x) \) as a variational auto-encoder (VAE) [17], where the encoder consumes visual data and the decoder outputs a program. To create data for \( p(z|x) \), we take the current best programs discovered for \( S^* \), \( P^{BEST} \), and execute each program to form a set of shapes \( X^G \). \( p(z) \) is then trained on pairs from \( (X^G, P^{BEST}) \) in the typical VAE framework. Note that the design space for \( p(z) \) is quite flexible, for instance, \( p(z) \) can trained without access to \( X^G \) if implemented with a program encoder.

Once \( p(z) \) has converged, we use it to sample \( |S^*| \) programs by decoding normally distributed random vectors. This set of programs becomes Z, and X is formed by executing each program in Z. In this set of (X, Z) programs, Z is always the correct label for X, so the gradient estimates during \( p(z|x) \) training will be low-variance and unbiased. However, X is not guaranteed to be close to \( S^* \), it is only an approximate distribution. Note though, that as \( P^{BEST} \) better approximates \( S^* \), the distributional mismatch should become smaller, as long as the generative model has enough capacity to properly model \( p(z) \).

3.2. Self-Training (X, Z) Construction

Self-training constructs (X, Z) by assigning labels from the current best program set, \( P^{BEST} \), to shape instances from \( S^* \). Formally, \( X \leftarrow S^* \) and \( Z \leftarrow P^{BEST} \). This framing maintains the nice property that \( X = S^* \), so there will never be distributional mismatch between these two sets. The downside is that unless programs from \( P^{BEST} \) exactly recreate their paired shapes from \( S^* \) when executed, we know that the pseudo-labels from \( P^{BEST} \) are ‘incorrect’. From this perspective, we can consider gradient estimates that come from such (X, Z) pairs to be biased. However, as we will show experimentally, when X forms a good approximation to \( S^* \), sourcing gradient estimates from these pseudo-labels leads to strong reconstruction performance.

3.3. LEST (X, Z) Construction

A unique property of shape program inference is that the distribution \( p(x|z) \) is readily available in the form of the program executor \( E \). We leverage this property to propose LEST, a variant of self-training that does not create mismatch between pseudo-labels and their associated visual data. Sim-
ilar to the self-training paradigm, LEST first constructs $Z$ as the current best program set, $P_{\text{BEST}}$. Then, differing from self-training, LEST constructs $X$ as the executed version of each program in $Z$. By construction, the labels in $Z$ will now be correct for their paired shapes in $X$. The downside is that, like wake-sleep, LEST may introduce a distributional mismatch between $X$ and $S^\ast$. But once again, as $P_{\text{BEST}}$ better approximates $S^\ast$, the mismatch between the two distributions will decrease.

### 3.4. Inferring Programs with $p(z|x)$

During each round of fine-tuning, PLAD methods rely on $p(z|x)$ to infer programs that approximate $S^\ast$. We propose to train PLAD methods such that the best matching inferred programs for $S^\ast$ are maintained across rounds. Specifically, we construct a data structure $P_{\text{BEST}}$ that maintains a program for each training shape in $S^\ast$. In this way, as more iterations are run, $P_{\text{BEST}}$ always forms a closer approximation to $S^\ast$. There is an alternative framing, where $P_{\text{BEST}}$ is reset each epoch, but we show experimental results in the supplemental material that this can lead to worse generalization.

To update $P_{\text{BEST}}$ each round, we employ an inner-loop search procedure. For each shape in $S^\ast$, $p(z|x)$ suggests high-likelihood programs, and the $P_{\text{BEST}}$ entry is updated to keep the program whose execution obtains the highest similarity to the input shape; the specific similarity metric varies by domain. While there are many ways to structure this inner-loop search, we choose beam-search, as we find it offers a good trade-off between speed and performance. Experimentally, we demonstrate that PLAD methods are capable of performing well even as the time spent on inner-loop search is varied (Section 4.4).

### 3.5. Training $p(z|x)$ with multiple PLAD methods

As detailed in the preceding sections, the main difference between PLAD approaches is in how they construct the $(X, Z)$ dataset used for fine-tuning $p(z|x)$. However, there is no strict requirement that these different $(X, Z)$ distributions be kept separate. We explore how $p(z|x)$ behaves under fine-tuning from multiple PLAD methods, such as combining LEST and self-training. We implement these mixtures on a per-batch basis. Before $p(z|x)$ training, we construct distinct $(X, Z)$ distributions for each method in the combination. Then, during training, each batch is randomly sampled from one of the $(X, Z)$ distributions. We experimentally validate the effectiveness of this approach in the next section.

### 4. Results

We evaluate a series of methods on their ability to fine-tune shape program inference models across multiple domains. We describe the different domains in Section 4.1 and details of our experimental design in Section 4.2. In Section 4.3, we compare the reconstruction accuracy of each method, and study how they are affected by varying the time spent on inner-loop search (Section 4.4) and the size of the training set (Section 4.5). Finally, we explore the convergence speed of each method in Section 4.6.

#### 4.1. Shape Program Domains

We run experiments across three shape program domains: 2D Constructive Solid Geometry (CSG), 3D CSG, and ShapeAssembly. Details can be found in the supplemental.

In CSG, shapes are created by declaring parametric primitives (e.g. circles, boxes) and combining them with Boolean operations (union, intersection, difference). CSG inference is non-trivial: as CSG uses non-additive operations (intersection, difference), inferring a CSG program does not simply reduce to primitive detection. For 2D CSG, we follow the grammar defined by CSGNet [24], using 400 shape tokens that correspond to randomly placed circles, triangles and rectangles on a 64 x 64 grid. For 3D CSG, we use a grammar that has individual tokens for defining primitives (ellipsoids and cuboids), setting primitive attributes (position and scales), and the three Boolean operators. Attributes are discretized into 32 bins.

ShapeAssembly is designed for specifying the part structure of manufactured 3D objects. It creates objects by declaring cuboid part geometries and assembling those parts together via attachment and symmetry operators. Our grammar contains tokens for each command type and parameter value; to handle continuous values, we discretize them into 32 bins.

#### 4.2. Experimental Design

**Fine-Tuning Methods** We compare the ability of the following training schemes to fine-tune a model on a specific domain of interest:

- **SP**: $p(z|x)$ with supervised pretraining.
- **RL**: Fine-tuning with REINFORCE.
- **WS**: Fine-tuning with wake-sleep.
- **ST**: Fine-tuning with self-training.
- **LEST**: Fine-tuning with latent execution self-training.
- **LEST+ST**: combining LEST and ST.
- **LEST+ST+WS**: combining LEST, ST and WS.

**Shape Datasets** Fine-tuning methods learn to specialize $p(z|x)$ against a distribution of real shapes $S^\ast$. For each domain, we construct a dataset of shapes $S^\ast$, and split it into train, validation, and test sets. We perform early-stopping with respect to the validation set. For 2DCSG, we use the CAD dataset from CSGNet [24], which consists of front and side views of chairs, desks, and lamps from the Trimble 3D warehouse. We split the dataset into $10K$ shapes for training, $3K$ shapes for validation, and $3K$ shapes for testing. For 3D CSG and ShapeAssembly, we use CAD shapes from the chair, table, couches, and benches categories of ShapeNet; voxelizations are provided by [6]. We split the dataset into
10K shapes for training, 1K shapes for validation, and 1K shapes for testing.

**Model Architectures** For all experiments, we model $p(z|x)$ in an encoder-decoder framework, although the particular architectures vary by domain. In all cases, the encoder is a CNN that converts visual data into a latent variable, and the decoder is an auto-regressive model that decodes the latent variables into a sequence of tokens. For 2D CSG, we use the same $p(z|x)$ architecture as CSGNet. A CNN consumes $64 \times 64$ binary mask shape images to produce a latent code that initializes a GRU-based recurrent decoder. For 3D CSG and ShapeAssembly, we use a 3D CNN that consumes $32 \times 32 \times 32$ occupancy voxels. This CNN outputs a latent code that is attended to by a Transformer decoder network [29]; this network also attends over token sequences in a typical auto-regressive fashion.

**Supervised Pretraining** Before fine-tuning, $p(z|x)$ undergoes supervised pretraining on synthetically generated programs until it has converged on that set. For 2D CSG, we follow CSGNet’s approach. For 3D CSG, we construct valid programs by (i) sampling a set of primitives within the allotted grid (ii) identifying potential overlaps (iii) constructing a binary tree of boolean operations using these overlaps. For ShapeAssembly, we propose programs by sampling random grammar expansions according to the language’s typing system. We then employ a validation step where a program is rejected if any of its part are not the sole occupying part of at least 8 voxels (to discourage excessive part overlaps). For 3D CSG and ShapeAssembly we sample 2 million synthetic programs and train until convergence on a validation set of 1000 programs. Full details provided in the supplemental.

### 4.3. Reconstruction Accuracy

We evaluate the performance of each fine-tuning method according to reconstruction accuracy: how closely the output of a shape program matches the input shape from which it was inferred, on a held out set of test shapes. The specific metric varies by domain. For 2D CSG, we follow CSGNet and use Chamfer Distance (CD), where lower distances indicate more similar shapes. For 3D CSG and ShapeAssembly, we use volumetric intersection over union (IoU).

For each domain, we run each fine-tuning method to convergence, starting with the same $p(z|x)$ model that has undergone supervised pretraining. The reward for RL models follows the similarity metric in each domain: CD for 2D CSG; IoU for 3D CSG and ShapeAssembly. For PLAD fine-tuning methods, the similarity metric in each domain determines which program is kept during updates to $P_{BEST}$. At inference time, when evaluating the reconstruction performance of each $p(z|x)$, we employ a beam search procedure, decoding multiple programs in parallel, and choosing the program that achieves the highest similarity to the target shape. We use a beam size of 10, unless otherwise stated.

We present quantitative results in Table 2. Looking at the middle four rows, the two self-training variants (ST and LEST) outperform RL as a fine-tuning method in all the domains we studied. The wake-sleep variant (WS) also outperforms RL for both 3D CSG and ShapeAssembly. These are more challenging domains with larger token spaces, posing difficulties for policy gradient fine-tuning. As demonstrated by the last two rows, further improvement can be had by combining multiple methods: for each domain, the best performance is achieved by LEST+ST+WS. In fact, for 2DCSG, the test-set reconstruction accuracy achieved by LEST+ST+WS (0.811) outperforms previous state-of-the-art results, CSGNet (1.14) [24] and CSGNetStack (1.02) [25], for paradigms where the executor is treated as a black-box.

Mixing updates from multiple PLAD methods is beneficial because, in this joint paradigm, each method can cover the other’s weaknesses. For instance, employing ST ensures that some samples of $X$ are sourced from $S^*$, and employing LEST ensures that some samples of $X$ have paired $Z$ programs which are exact labels. We present qualitative results in Figure 3. The reconstructions from the PLAD combination methods better reflect the input shapes, reinforcing the quantitative trends.

<table>
<thead>
<tr>
<th>Method</th>
<th>2D CSG CD</th>
<th>3D CSG IoU</th>
<th>ShapeAssembly IoU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supervised Pretraining (SP)</td>
<td>1.580</td>
<td>41.0</td>
<td>37.6</td>
</tr>
<tr>
<td>REINFORCE (RL)</td>
<td>1.097</td>
<td>53.4</td>
<td>50.8</td>
</tr>
<tr>
<td>Wake-Sleep (WS)</td>
<td>1.118</td>
<td>67.4</td>
<td>57.2</td>
</tr>
<tr>
<td>Self-Training (ST)</td>
<td>0.841</td>
<td>67.3</td>
<td>61.3</td>
</tr>
<tr>
<td>LEST</td>
<td>0.976</td>
<td>69.8</td>
<td>56.5</td>
</tr>
<tr>
<td>LEST+ST</td>
<td>0.829</td>
<td>70.8</td>
<td>66.0</td>
</tr>
<tr>
<td>LEST+ST+WS</td>
<td><strong>0.811</strong></td>
<td><strong>74.3</strong></td>
<td><strong>66.4</strong></td>
</tr>
</tbody>
</table>

Table 2. Test-set reconstruction performance across multiple shape program inference domains. The top row contains results for the pretrained $p(z|x)$ model fine-tuned by the other methods. For 2D CSG the metric is Chamfer distance (CD, lower is better). For 3D CSG and ShapeAssembly the metric is intersection over union (IoU, higher is better). Individual PLAD methods outperform RL, and combining PLAD methods achieves the best performance across all domains (LEST+ST+WS).
Figure 2. Experiments exploring properties of PLAD methods on 2D CSG. On the X-axis we plot the beam size used during the $p_{BEST}$ update (Left), the number of training shapes (Middle), and the training time (Right). The Y-axis of each plot measures reconstruction accuracy on test-set shapes.

Figure 3. Qualitative comparisons of shape programs inferred for test-set shapes made by different fine-tuning methods for 2D CSG (Top), 3D CSG (Middle), and ShapeAssembly (Bottom). We provide additional qualitative results in the supplemental.
4.4. Inner-loop Search Time

PLAD methods make use of $P_{\text{BEST}}$ to generate $(X, Z)$ datasets that train $p(z|x)$. To study how time spent on inner-loop search affects each technique, we ran an experiment using different beam sizes to update $P_{\text{BEST}}$. We present results in Figure 2, left. On the X-axis we plot beam size; on the Y-axis we plot test-set reconstruction Chamfer distance. Unsurprisingly, spending more time on the inner-loop search leads to better performance; finding better programs for training shapes improves test time generalization. That said, across all beam sizes, we find that it is always best to train under a combination of PLAD methods; the LEST+ST and the LEST+ST+WS variants always outperform any individual fine-tuning scheme. Note, RL is not included in this experiment because REINFORCE, as defined, has no inner-loop search mechanism. In this way, PLAD provides an additional control lever, where time spent on inner-loop search modulates a trade-off between convergence speed and test-set reconstruction performance.

4.5. Number of Training Shapes from $S^*$

All the fine-tuning methods make use of a training distribution of shapes that are sampled from $S^*$. For some domains, the size of available samples from $S^*$ may be limited. We run an experiment on 2D CSG to see how different fine-tuning methods are affected by training data size. We present the results of this experiment in Figure 2, middle. We plot the number of training shapes on the X-axis and test set reconstruction accuracy on the Y-axis. All fine-tuning methods improve as the training size of $S^*$ increases, but once again, combining multiple PLAD methods leads to the best performance in all regimes. This study also demonstrates the sample efficiency of PLAD combinations: LEST+ST and LEST+ST+WS trained on 1,000 shapes achieve better test set generalization than RL trained on 10,000 shapes.

4.6. Convergence Speed

Beyond reconstruction accuracy, we are also interested in the convergence properties of a fine-tuning method. Policy gradient RL is notoriously unstable and slow to converge, which is undesirable. For 2D CSG, we record the convergence speed of each method and present these results in Figure 2, right. We plot reconstruction accuracy (Y-axis) as a function of training wall-clock time (X-axis); all timing information was collected on a machine with a GeForce RTX 2080 Ti GPU and an Intel i9-9900K CPU. All PLAD techniques converge faster than policy gradient RL. For instance, RL took 36 hours to reach its converged test-set CD of 1.097, while LEST matched this performance at 1.1 hours (32x faster) and LEST+ST matched this performance at 0.85 hours (42x faster).

5. Conclusion

We presented the PLAD framework to group a family of techniques for fine-tuning shape program inference models with Pseudo-Labels and Approximate Distributions. Within this framework, we proposed LEST: a self-training variant that creates a shape distribution $X$ approximating the real distribution $S^*$ by executing inferred latent programs. Experiments on 2D CSG, 3D CSG, and ShapeAssembly demonstrate that PLAD methods achieve better reconstruction accuracy and converge faster than policy gradient RL, the current standard approach for black-box fine-tuning. Finally, we found that combining updates from multiple PLAD methods outperforms any individual technique.

While fine-tuning $p(z|x)$, PLAD methods construct $(X, Z)$ sets approximating the statistics of $S^*$, specializing $p(z|x)$ towards $S^*$. As a consequence, $p(z|x)$ may not generalize as well to shapes outside of $S^*$; we explore this phenomenon in the supplemental material. Training a general-purpose inference model for all shapes expressible under the grammar is an interesting line of future work.

While our work focuses on reconstruction quality, producing programs with ‘good’ structure matters just as much, if the program is to be used for editing tasks. Currently, the synthetic pretraining data is the only place where knowledge about what constitutes “good program structure” can be injected. Such knowledge must be expressed in procedural form, which may be harder to elicit from domain experts than declarative knowledge (i.e. “a good program has these properties” vs. “this is how you write a good program”). Finding efficient ways to elicit and inject such knowledge is an important future direction.

Finally, we believe that ideas from the PLAD framework are applicable to a broader class of program inference problems than those we evaluated in this paper. In principle, these approaches can be used to train an approximate inference model $p(z|x)$ for any domain in which (1) an executor $p(x|z)$ is available and (2) the executed output $x$ takes the form of some concrete artifact which can be encoded via neural network (image, audio, text, etc.). For example, one could imagine using PLAD techniques to infer graphics shader programs which produce certain textures or audio synthesizers which sound like certain real-world instruments.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful suggestions. This work was funded in parts by NSF award #1941808 and a Brown University Presidential Fellowship. Daniel Ritchie is an advisor to Geopipe and owns equity in the company. Geopipe is a start-up that is developing 3D technology to build immersive virtual copies of the real world with applications in various fields, including games and architecture.
References


