Supplement Contents

A Implementation Details

A.1 The Simulation Tasks

In this section, we discuss in further detail the ScoopBall and PourWater simulation tasks we introduced in Section 4.1.

A.1.1 Shared Properties of Simulation Tasks

For both tasks, point cloud observations have a maximum of $N = 2000$ points, and there may be fewer points at certain time steps. We do not zero-pad the point clouds to keep them a fixed data dimension. To obtain segmented point clouds, we use the depth images from SoftGym and project each pixel into world coordinates to create a point cloud. Since we use the observable point cloud at each time step for the tool, there may be occlusions, i.e., if certain parts of the tool are not visible to the depth camera, they will not be included in the point cloud. For example, the second flow visualization in Figure 4 shows that the ball occludes part of the point cloud.

Both tasks use a fixed episode length of 100 time steps (there is no early termination) along with an action repeat of 8, meaning that each action $\mathbf{a}$ from the policy or demonstrator is executed 8 times "internally" which corresponds to 1 of the 100 time steps. For further investigation, we test each task using two different action spaces. Each task and action space comes with a scripted demonstrator. Videos of the demonstrator for all tasks are available on the project website: `https://tinyurl.com/toolflownet`.

The simulation is built upon FleX [59], which is a particle-based simulator. Hence, we sometimes may use "water particles" or "ball particles" to describe the physics of those items. The tools for both tasks are not particle-based. In FleX simulation, when the tools move, they affect the position of the particles (but not vice versa).

A.1.2 ScoopBall, Details of the Task and Demonstrator

Overview. The agent controls a ladle, and each episode begins with the ladle above a box that contains water and a single ball floating on it. The objective is to scoop the ball above a certain height. Each episode has a different initial position of the ball. There are 2 versions of the task, with 4 DoF and 6 DoF actions. For the former, the action space $\mathbf{a} = (\Delta x, \Delta y, \Delta z, \Delta\theta)$ is 4D, which consists of the change in the coordinates of the ladle tip, and the change in rotation about the vertical axis $\Delta\theta$ coinciding with the ladle's tip. See Figure 4 for visuals of the translation and rotation actions. We ignore the unused 2 dimensions from the 6D output transformation of ToolFlowNet to get a 4D action. For the 6 DoF action version, the action is 6D and consists of 3D delta translation and 3D delta rotation (which is the 3D delta axis-angle in the local tool frame). We also add a hole in the ladle for the 6 DoF action version to let water leak from it, which significantly stabilizes the water-ball simulator dynamics.

Point Cloud. The point cloud uses two classes, corresponding to the tool (i.e., ladle) and the target ball. The water particles are not part of the point cloud, as knowledge of the water particles is not critical to succeed at the task. The task uses the observable point cloud for both the tool and the ball, so the tool can occlude the ball (and vice versa). The water particles do not occlude the tool.

Success Criteria. A binary success is triggered when the agent keeps the ball above a height threshold (above the water height) for at least 10 time steps, which ensures that the ball must be at a reasonably stable state for a success. The agent can only do this by using its ladle to scoop the ball.

Physics Considerations. We create a ladle model in Blender[1] and import the resulting model into SoftGym as a signed-distance function. Since the FleX physics backend does not provide collision checking for arbitrary meshes, we implement our own approximate version. Whenever the agent applies an action, we compute the sphere formed from "completing" the ladle's bowl, and then

[1] `https://www.blender.org/`

compute whether it has intersected with the walls of the box. If an intersection exists, then we clip each action dimension such that the resulting state is still legal (i.e., respects collision boundaries).

The physics of water-ball interaction in FleX have a great impact on this task. We set the ball so that it has a density that should make it always float on water. However, when the agent scoops the ball, the water particles may easily "push" the ball away, causing the ball to fall back into the water. (This behavior does not happen in our physical experiments.) Furthermore, when the ball drops from midair into the water, sometimes the ball remains sunk afterwards, making it impossible for the agent to succeed with the given action space and ladle physics. In future work, we will investigate simulators that have improved liquid-solid physics interactions. For this task, we originally began tests with a 4 DoF action space which used a ladle with a solid bowl, which meant during scooping that its water paticles could push away the ball. The 6 DoF action version, however, uses a ladle with a hole in it to let water drain, which significantly improved success rates.

Demonstrator. With 4 DoF actions, we implement an algorithmic demonstrator which first lowers the ladle into the water, attempts to move the ladle so that its bowl is underneath the ball, then lifts the ball. The demonstrator continually rotates the ladle so that the ladle's handle is facing the direction of the ball. When the demonstrator lifts, it rotates again so that the ladle is back at the starting rotation. The process of lowering and lifting the ladle results in y-coordinate[2] changes of 0.004 in SoftGym simulation units, while translations within the water results in actions similarly bounded by 0.004 units in both coordinate directions. When translations get scaled by 250X (see Appendix B.3), the targets have per-component magnitude upper bounded by $0.004 \times 250 = 1.0$. Each rotation consists of a change in 0.5 degrees about the axis coinciding to the ladle's "stick." Largely owing to the aforementioned water-ball simulation artifacts, the demonstrator success rate is 63.2%. We filter the resulting data to only imitate successful demonstrations.

With the 6 DoF version with the different ladle, we re-script the demonstrator to execute a more visually natural scoop which uses all its rotations, and then moves towards the ball, then rotates back to a neutral position and lifts upwards. The demonstrator success rate here is 100.0%.

A.1.3 PourWater, Details of the Task and Demonstrator

Overview. We use the PourWater task from SoftGym [58]. The task comes with a 3 DoF action space, so to explore more complex action spaces, we modify the code to support a 6 DoF action space. The agent controls a box which contains water and must pour the water into a fixed target box. There are two versions of the task, with 3 DoF actions and 6 DoF actions. The 3 DoF action space $\mathbf{a} = (\Delta x, \Delta y, \Delta \theta)$ consists of the change in the x and y coordinates of the controlled box's center and the change in rotation $\Delta\theta$ around the bottom center of the box along a single coordinate axis. See Figure 1 (left) for the translation and Figure 1 (right) for the rotation. For the 6 DoF version, the action is 6D and consists of the translation change in all x, y, z coordinates, and rotational change around the bottom center of the box along all 3 coordinate axes. The rotation is expressed as Euler angles represented as an extrinsic rotation about the Z-Y-X axes in that order. The rotational change is a delta in the Euler angles for all axes. In each episode, we vary the sizes of both boxes, the amount of water in the controlled box, and the starting distance between the boxes. For the 6 DoF action version, we also vary the initial orientation of the controlled box. The proposed ToolFlowNet generates full 6D transformations, and we ignore the unused 3 action dimensions at test time for the 3 DoF action version.

To make the task more tractable, we slightly reduce the maximum possible box to be about 75% of the maximum size compared to the public version. For the 6 DoF action space, we add more randomness to the initial starting pose of the controlled box. Other than that, for the 3 DoF action space, we keep the PourWater settings as consistent with the open-source code as possible to potentially facilitate comparisons with other work using this task.

Point Cloud. The point cloud uses three classes, corresponding to the tool (i.e., the box that starts with water), the target box for the water, and the water itself. Here, we use the observable point

[2]In SoftGym, the positive y-axis points upwards, while the x- and z-axes form a flat horizontal plane.

cloud for the two boxes, but for the water, we follow the existing SoftGym implementation and use the ground truth water particle positions which we can query at each time step. We include the water in the point cloud because knowledge of the water is essential for pouring.

Success Criteria. We set a binary success threshold based on if at least 75% of the water particles end in the target box.

Physics Considerations. One of the FleX simulation artifacts is that water particles can "seep through" the corners and edges of both boxes. The 75% particle threshold we use is high enough to convey reasonable task success, but not so high that it cannot tolerate some water particles escaping from the boxes. To handle collisions, for the 3 DoF action space, we use the existing collision checking code from SoftGym without further modification. If the tool intersects with the bottom floor, or intersects with the target box, the action is not applied, which can cause the tool to "freeze" if repeatedly applying collision-violating actions. We implement a similar collision checking code for the new 6 DoF action space.

Demonstrator. For both action versions, we script a demonstrator which moves the box towards the target and rotates to pour the water. With 3 DoF actions, we implement an algorithmic demonstrator which moves the box towards the target, lifts the box, then rotates to pour the water in the target. The act of moving towards the box consists of translations in the positive x direction of 0.003 units, lifting consists of translations in the positive y direction of 0.003 units, and then rotating consists of rotating 0.5 degrees in the positive direction to pour, and then rotating negative 0.5 degrees to reset back to the original orientation. When translations get scaled by 250X, this creates targets with per-component magnitudes of $0.003 \times 250 = 0.75$. The demonstrator success rate is 90.6%.

We script a similar demonstrator for the 6 DoF action version. The controlled box starts from a more complex configuration, so the demonstrator rotates and translates it to align it with the target. Then, it does a similar maneuver as the 3 DoF demonstrator to pour the box with water into the target box. Its success rate is 81.5%.

A.2 More Details on Simulation Experiments

We present more details of the simulation experiments reported in Section 4 (and in Appendix B).

A.2.1 Training Hyperparameters

Network	Epochs	LR	Batch	Params
PointNet++	500	1e-4	24	1.4M
CNN	500	1e-4	128	3.0M

Table S1: Some hyperparameters used in experiments. We report the number of training epochs, the Adam learning rate, the batch size, and the number of network parameters.

For a representative set of hyperparameters, see Table S1. We train models for the same number of epochs with a common Adam [62] learning rate of 1e-4. However, the CNN uses a larger batch size and has more than 2X as many parameters as compared to the PointNet++. Here, we use PointNet++ to refer to both the segmentation version (as used in ToolFlowNet) and the classification version (used in some baselines), which have almost the same number of parameters (1.4M).

A.2.2 Experiment Protocol and Evaluation Metrics

For each task, we generate a set of starting configurations and divide them into training and testing configurations. Each configuration has a slightly different arrangement of particles, so that the policies cannot succeed by memorizing the training data. We use an algorithmic demonstrator (described in Appendix A.1) to generate a fixed set of training demonstrations from the starting training configurations. Following prior work in imitation learning [63], we filter the demonstrations to keep only the successful ones for Behavioral Cloning. For a fair comparison, all comparisons among methods on a single task and action space train on the same set of demonstrations.

For simulation experiments, we standardize on 100 training demonstrations for all tasks except for ScoopBall 6D, for which we use 25 training demonstrations. See Appendix B.10 for experiments where we adjust the number of training demonstrations.

We evaluate Behavioral Cloning performance by training for 500 epochs on task-specific demonstration data. For each method, we perform 5 independent runs (each with a different random seed), and evaluate every 25 epochs on 25 fixed held-out starting configurations. In other words, we test "snapshots" of each training run every 25 epochs. As all episodes have a binary success outcome, averaging the 25 test episodes gives us one quantitative number for each epoch in a training run. We then average over the 5 Behavioral Cloning runs for each epoch, and treat that number as the method's performance at each epoch. Thus, each epoch's resulting metric reflects $25 \times 5 = 125$ total evaluation rollouts, and we then consider the maximum over all the epochs, since in Behavioral Cloning we often just care about the best snapshot at any time (due to no environment interaction). That provides a number corresponding to raw success rate. We finally normalize by dividing this value by the demonstrator performance, which gives us the final normalized success rate.

Due to noise and variance in the learning process [64], for a given method, we report not just the average normalized performance but also the corresponding standard error of the mean, which here is the sample standard deviation divided by $\sqrt{5}$. In addition, **when bolding numbers in tables** to indicate the "best" method, we bold both the best number *and* those that have overlapping standard errors. For example, when comparing just $x_1 \pm y_1$ versus $x_2 \pm y_2$, if $x_1 > x_2$, then we would bold the x_1 value in a table, *along with* x_2 if the condition $x_2 + y_2 \geq x_1 - y_1$ holds.

In Appendix B.4, we report an alternative evaluation metric where we instead take an average over all epochs instead of picking the best one.

A.2.3 Implementation of Baseline Methods

To keep experimental settings fair, we strive to apply as consistent settings as possible among the methods. For example, if we evaluate using an action space with fewer than 6 DoFs, we perform the same procedure to convert a 6D action prediction into a 3D action (for PourWater) or a 4D action (for ScoopBall) for all methods by zeroing out unused action dimensions. In addition, the baselines that use the classification PointNet++ architecture, **Direct Vector** with either the MSE (Equation 2) or Point Matching (Equation 3) losses, use an architecture with roughly the same amount of parameters (approximately 1.4M) as the segmentation PointNet++ architecture.

We test with two baselines we call **Dense Transformation** with (again) variants based on using the MSE or Point Matching losses. For these methods, we pick one fixed point on the tool and use that to represent the point of interest. The segmentation PointNet++ produces per-point outputs, so this fixed point tells us which one, out of all the output points, provides the transformation (i.e., action). For ScoopBall we use the tip of the ladle, and for PourWater we use the center of the bottom of the box. These both coincide with the center of the tool rotation, and which we treat as synthetic points in the tool point cloud. We extract them using ground truth simulator knowledge (instead of the observable point cloud, since they may be occluded or out of view) and insert them into the segmented point cloud $\mathbf{P}$. These form one point on the tool; the point cloud still contains the usual amount of tool points in the observable point cloud.

For methods that process images, we use a Convolutional Neural Network (CNN) to process the images. We use an architecture similar to the design of the CNN encoder in the SAC/CURL code repository [65], but where we slightly reduce the parameter count to be about 3M, which is still more than the 1.4M parameters for the PointNet++ models. In PyTorch print string format, assuming a three-channel image input, the network is expressed as:

```
Actor(
  (encoder): PixelEncoder(
    (convs): ModuleList(
      (0): Conv2d(3, 16, kernel_size=(3, 3), stride=(2, 2))
      (1): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1))
      (2): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1))
      (3): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1))
    )
    (fc): Linear(in_features=29584, out_features=100, bias=True)
    (ln): LayerNorm((100,), eps=1e-05, elementwise_affine=True)
  )
  (trunk): Sequential(
    (0): Linear(in_features=100, out_features=256, bias=True)
    (1): ReLU()
    (2): Linear(in_features=256, out_features=256, bias=True)
    (3): ReLU()
    (4): Linear(in_features=256, out_features=6, bias=True)
  )
)
```

The RGB and depth input images have resolution 100x100. When testing with RGB and depth (i.e., RGBD) images, we stack the images channel-wise. We also augment these baseline methods to include binary segmentation image masks as extra image channels. This is designed to reproduce the same segmentation information that is present in a segmented point cloud. With ScoopBall, there are two binary segmentation mask images, one for the tool and one for the ball. If testing using RGB image inputs with segmentation masks (denoted as RGB+S in tables), for example, then this results in 5-channel input images to the CNN, with the first three for RGB and the last two for the segmentation masks. PourWater, however, has three binary segmentation masks, corresponding to the controlled cup, the target cup, and the water. Thus, for PourWater, RGB+S image inputs are 6-channel images.

For consistency, the CNN architecture we use is the same across all methods that process image inputs.

A.2.4 Implementation of ToolFlowNet

For the policy architecture π_θ, we build on the PointNet++ implementation from PyTorch Geometric [66]. We keep the architecture and hyperparameters similar to those in PyTorch Geometric. Both the segmentation and classification PointNet++ use two Set Abstraction levels [15] with ratio parameters 0.5 and 0.25 and ball radius parameters 0.2 and 0.4 for the two layers, respectively. These two Set Abstraction levels are then followed by a third "Global" Set Abstraction layer which performs a global max-pooling operation. The segmentation version applies Feature Propagation layers to upsample. The PointNet++ networks we use in experiments have approximately 1.4M parameters. The PyTorch print string of the model with 3D flow output is:

```
Actor(
  PointNet2_Flow(
    (sa1_module): SAModule(
      (conv): PointNetConv(local_nn=MLP(5, 64, 64, 128), global_nn=None)
    )
    (sa2_module): SAModule(
      (conv): PointNetConv(local_nn=MLP(131, 128, 128, 256), global_nn=None)
    )
    (sa3_module): GlobalSAModule(
      (nn): MLP(259, 256, 512, 1024)
    )
    (fp3_module): FPModule(
      (nn): MLP(1280, 256, 256)
    )
    (fp2_module): FPModule(
      (nn): MLP(384, 256, 128)
    )
    (fp1_module): FPModule(
      (nn): MLP(130, 128, 128, 128)
    )
    (mlp): MLP(128, 128, 128, 3)
  )
)
```

To implement the differentiable, parameter-less Singular Value Decomposition (SVD) layer, we use PyTorch3D [67].

A.2.5 Implementation of ToolFlowNet Ablations

Below, we expand upon the description of ablations in Section 4.3.

- **ToolFlowNet, No Skip Connections**: removes the skip connections in the segmentation PointNet++, which we implement by not invoking a `torch.cat([x, x_skip])` command in the feature propagation layers [15], where `x_skip` represents features from an earlier layer.
- **ToolFlowNet, MSE after SVD**: tests applying an MSE loss on the induced transformation from SVD instead of point matching. The output of the SVD is a transformation, which is equivalently expressed as a 6D translation and rotation action vector $\mathbf{a}$ which we can directly use with MSE against the ground truth actions. We still use the consistency loss (Eq. 4) to supervise the internal per-point flow vectors, as that may help regularize the predicted flow.
- **ToolFlowNet, Point Matching (PM) Before SVD**: tests using the PM loss (Eq. 3) before the SVD layer, so the loss does not back-propagate through the SVD layer. Here, the SVD is not differentiable, but is used at test time because for a given input point cloud $\mathbf{P}$, the output of the neural network is an $(N' \times 3)$-sized flow array, which must then be converted to an action $\mathbf{a}$. We do not test this ablation with consistency since this would add a second objective to the internal flow predictions which could confuse the network.
- **ToolFlowNet, No Consistency**: tests removing the consistency loss (and just using Eq. 3), or equivalently, setting $\lambda = 0.0$ in L_{combo}. This tests to see whether it is feasible to just use the point matching loss without having a loss that directly supervises the predicted flow vectors. We investigate this ablation further in Table S3.

B Additional Simulation Experiments and Analysis

We analyze existing experiments and present new ones to further investigate ToolFlowNet in simulation (see Appendix C for physical experiments). Unless stated otherwise, we use the experimental settings from Appendix A.2.2.

In Appendix B.1, we present several theories on when using tool flow may be helpful. We investigate these theories in new experiments where we test on translation-only data for ScoopBall (Appendix B.1.1) and where we test on a new task to investigate a "locality hypothesis" (Appendix B.1.2). Next, we explore hyperparameters and target scaling in Appendix B.2 and Appendix B.3 for the main ToolFlowNet method we present. Building upon these results, we extend our main set of results in Appendix B.4. We then extend experiments from the paper in Appendix B.8 and Appendix B.9 on noise injection and the number of tool points, respectively. Appendix B.10 contains new experiments where we test performance based on the training data size.

B.1 Why is Tool Flow Helpful?

Using tool flow as a representation in ToolFlowNet is helpful for our simulation tasks as compared to "Direct Vector" representations which directly regress to a single vector. Why is that the case? Building upon our discussion in Section 4.3, we consider two possible theories:

- Tool flow is helpful because it represents rotations in a format that is easier to learn.
- Tool flow is helpful because of locality bias.

The first theory is relevant to how there are multiple ways to represent rotations. Prior work has shown that naively regressing onto some rotation representations such as quaternions, axis-angles, and Euler angles is challenging [60, 61, 68], and hence flow may be a representation that induces easier learning. See Appendix B.1.1 for experiments to probe this theory in comparison with axis-angle rotations, and see Appendix B.12 for experiments with other rotation representations.

The second theory may be relevant to the object-centric nature of the tasks. In ScoopBall the policy must reason about the relationship between the ladle and the ball, and in PourWater it must reason about the relationship between the controlled box and the target box. See Appendix B.1.2 for experiments to investigate this theory.

B.1.1 Learning from Translation-Only Data

Method	Demo Type	ScoopBall
PCL Direct Vector (MSE)	3DoF	**0.817±0.04**
ToolFlowNet (No SVD)	3DoF	**0.808±0.04**
ToolFlowNet	3DoF	**0.769±0.18**
PCL Direct Vector (MSE)†	4DoF	0.544±0.03
ToolFlowNet †	4DoF	**1.152±0.07**

†Results are directly from Table 1.

Table S2: Results on ScoopBall for 3DoF translation-only demonstrators and, for comparison purposes, the 4DoF demonstration data which is used in other experiments. For each demonstrator type, we bold the best performance numbers from the methods, along with any with overlapping standard errors.

To better understand when tool flow as an action representation is beneficial, we run a smaller-scale experiment on ScoopBall 4D where we now use a translation-only demonstrator.[3] We script this demonstrator to lower the ladle, then to translate it in water to get its bowl under the target ball, then to lift the ladle (ideally with the ball); its success rate is 0.832. This environment uses the same ladle and starting structure as ScoopBall 4D from the paper, and thus does not use the alternative tool used in ScoopBall 6D.

[3]We use ScoopBall since a policy can succeed without using rotations; this is not the case with PourWater.

For this variant, in addition to the standard ToolFlowNet method, we consider a "ToolFlowNet (No SVD)" variant, which is a segmentation PointNet++ that (instead of an SVD layer) ends with an "averaging layer" which averages all the predicted tool flow vectors. We test this variant because if the demonstration data is translation-only, then the SVD layer in ToolFlowNet is supervised to produce the identity rotation, which could be challenging as it would require that all tool flow vectors have the same direction. We supervise ToolFlowNet (No SVD) with the MSE loss and we do not use a consistency loss. We compare with the PCL Direct Vector (MSE) baseline, which is the same as ToolFlowNet (No SVD) except it uses a classification PointNet++ instead of a segmentation PointNet++. From Table S2, we find that with 3DoF demonstration data, the naive vector policy actually slightly outperforms both ToolFlowNet and ToolFlowNet (No SVD), indicating that ToolFlowNet brings the most benefits when considering both translations and rotations. The results, however, are fairly close with overlapping standard errors (which we consider as per our evaluation practices in Appendix A.2.2), as $0.817 - 0.04 = 0.777$ for Direct Vector on the lower end of the interval, which is less than the value at the positive end of ToolFlowNet's interval: $0.769 + 0.18 = 0.949$.

B.1.2 Locality Bias Hypothesis

One hypothesis we have on why ToolFlowNet is better than the naive Direct Vector baselines is that the dense representation in ToolFlowNet brings better locality bias, i.e., ToolFlowNet might reason better about the relationship between the tool and target object (e.g., the ball in ScoopBall, and the target cup in PourWater) when the tool is near the target object. To test this hypothesis, we design a simple task where the goal is to move the tool, represented as a sphere, to the target, represented as another sphere. The action is the 3D translation of the tool sphere, and the reward is the negative distance between the tool and the target sphere. Since the action is translation only, we average the predicted flow from ToolFlowNet to get the final translation action without using the SVD layer. For the observation, we randomly sample 100 points on the surface of the tool sphere, and another 100 points on the surface of the target sphere. The expert demonstration is simply moving the tool towards the target sphere. As in the main paper, ToolFlowNet is trained using the point matching loss and the consistency loss. We vary the initial distance between the tool and the target sphere, and if the locality hypothesis is true, then we would expect a larger gap between ToolFlowNet and the naive vector baseline when the tool is near the target, and a smaller gap between ToolFlowNet and the naive vector baseline when the tool is far away from the target.

As Figure S1 shows, the normalized performance gap between ToolFlowNet and the baseline (computed as the normalized performance of ToolFlowNet minus the normalized performance of the vector baseline) indeed decreases as the initial distance between the tool and the target increases (results averaged across 5 seeds), which provides support for the locality bias hypothesis.

B.2 Consistency Loss Hyperparameter

Method	λ	ScoopBall 4D	ScoopBall 6D	PourWater 3D	PourWater 6D
ToolFlowNet	0.0	0.861±0.04	0.744±0.12	0.468±0.09	**0.609±0.06**
ToolFlowNet	0.1	**1.152±0.05**	**0.952±0.02**	**0.768±0.04**	**0.667±0.03**
ToolFlowNet	0.5	0.987±0.02	**0.912±0.04**	**0.795±0.05**	**0.658±0.09**
ToolFlowNet	1.0	0.873±0.05	**0.944±0.03**	**0.777±0.04**	**0.628±0.05**

Table S3: Performance of ToolFlowNet on all combinations of the environments and action spaces. We vary the consistency weight λ for the consistency loss in Eq. 4. We bold the best results in each column along with those that have overlapping standard errors.

We test different values of the λ weight for the consistency loss: $\lambda \in \{0.0, 0.1, 0.5, 1.0\}$. Using $\lambda = 0$ is the same as the ablation named "ToolFlowNet, No Consistency" in Table 1. In Table S3 we report the BC test-time performance (using the standard metric of 5 independent BC runs and taking the best epoch) for both tasks. We find that for both action spaces of ScoopBall, the best value seems to be $\lambda = 0.1$ (and leads to slightly outperforming the demonstrator). For PourWater 3D and 6D, the best values are $\lambda = 0.5$ and $\lambda = 0.1$, respectively though there are multiple values

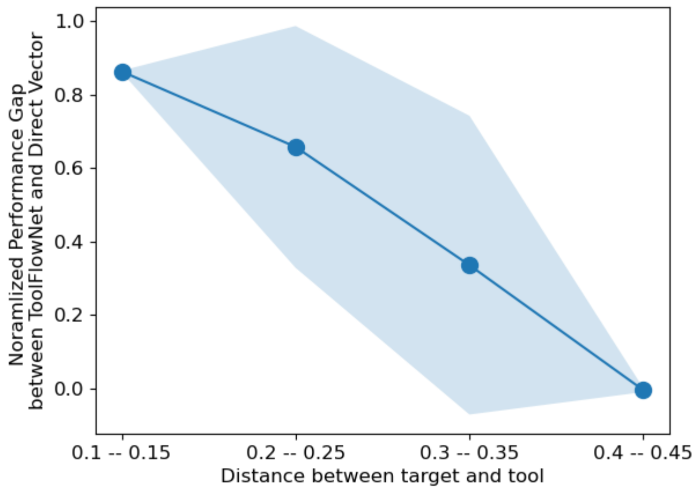


Figure S1: The normalized performance gap between ToolFlowNet and the Direct Vector baseline, with varying initial distance between the tool and the target sphere. For x-axis ticks, $0.1 - 0.15$ means the initial distance is uniformly sampled from the range $[0.1, 0.15]$.

of λ for which performance is similar based on the range of standard errors. Consequently, in other experiments (such as the ones we report in Table 1) we use $\lambda = 0.1$ for ScoopBall 4D and 6D, $\lambda = 0.5$ for PourWater 3D, and $\lambda = 0.1$ for PourWater 6D for ToolFlowNet.

B.3 Scaling Targets During Training

Success	Scale?	ScoopBall 4D	PourWater 3D
ToolFlowNet [†]	✓	1.152±0.07	0.795±0.05
Direct Vector (MSE)[†]	✓	0.544±0.03	0.530±0.08
ToolFlowNet	✗	0.722±0.14	0.706±0.02
Direct Vector (MSE)	✗	0.000±0.00	0.000±0.00

[†]These numbers are directly from Table 1.

Table S4: Success rates of ToolFlowNet and the Direct Vector baseline based on whether targets are scaled, by 250X, or kept at defaults (scale 1) with the latter resulting in per-component values within ± 0.004.

One strategy to improve training of ToolFlowNet is to scale the flow vector targets. In simulation, each time step is a single continuous action which results in extremely small changes in the translation and rotation of the tool pose. (In Section A.1 we state quantitative numbers.) For our experiments, we scale the flow targets by a factor of 250X to empirically get flow vector magnitudes to be bounded by approximately -1 and 1 in each of the three coordinate dimensions. We do not scale the input point cloud $\mathbf{P}$ for the forward pass through the PointNet++ network, but we *do* scale it to ensure correctness in computing the point matching loss in Eq. 3, since the computation must be done with all values expressing the same units. Intuitively, the scaling acts as a way of converting the units to make the raw values more suitable for training (e.g., going from meters to millimeters).

To verify our design choice, we run an experiment where we test ToolFlowNet (with the point matching and consistency loss), with and without scaling. We run 5X Behavioral Cloning runs, and report the average (and standard error of the mean) of the best epoch. The results in Table S4 suggest clear benefits to scaling the flow vectors in both tasks.

For consistency, we perform a similar scaling for the baseline, non-flow methods by multiplying their translational magnitudes by 250X. We also perform a similar scaling of the rotations to get

Method	Loss	Dense PN++?	N. Success ScoopBall 4D	N. Success ScoopBall 6D	N. Success PourWater 3D	N. Success PourWater 6D	Average N. Success
PCL Direct Vector	MSE	✗	0.408±0.01	0.640±0.03	0.337±0.04	0.264±0.01	0.412
PCL Direct Vector	PM	✗	0.128±0.07	0.002±0.00	0.045±0.02	0.042±0.01	0.055
PCL Dense Transformation	MSE	✓	0.427±0.02	0.669±0.06	0.372±0.02	0.212±0.03	0.420
PCL Dense Transformation	PM	✓	0.235±0.03	0.158±0.04	0.316±0.01	0.020±0.01	0.182
D Direct Vector	MSE	✗	0.119±0.03	0.744±0.02	0.013±0.00	0.020±0.00	0.224
D+S Direct Vector	MSE	✗	0.311±0.04	0.804±0.03	0.656±0.03	0.231±0.01	0.500
RGB Direct Vector	MSE	✗	0.213±0.03	0.646±0.01	0.607±0.03	0.216±0.01	0.420
RGB+S Direct Vector	MSE	✗	0.326±0.03	**0.872±0.02**	**0.734±0.01**	0.179±0.01	0.528
RGBD Direct Vector	MSE	✗	0.263±0.03	0.817±0.02	0.662±0.02	0.221±0.01	0.491
RGBD+S Direct Vector	MSE	✗	0.423±0.04	**0.883±0.02**	**0.713±0.03**	0.227±0.01	0.561
ToolFlowNet, No Skip Conn.	PM+C	✓	**0.768±0.03**	0.130±0.02	0.000±0.00	0.000±0.00	0.225
ToolFlowNet, MSE after SVD	MSE+C	✓	0.011±0.01	0.643±0.04	0.324±0.01	**0.604±0.04**	0.395
ToolFlowNet, PM before SVD	PM	✓	0.550±0.04	0.708±0.03	0.410±0.02	0.430±0.02	0.525
ToolFlowNet, No Consistency	PM	✓	0.585±0.04	0.461±0.04	0.289±0.11	0.375±0.03	0.427
ToolFlowNet (Ours)	PM+C	✓	**0.813±0.02**	0.799±0.02	0.692±0.03	0.536±0.01	**0.710**

Table S5: Results from the same set of experiments reported in Table 1, except we use a different evaluation metric, based on averaging the normalized test-time success rate across all evaluation (every 25) epochs, instead of picking the best one epoch. Hence, the raw numbers are lower. See Appendix B.4 for further details. We bold the best numbers in the columns, plus any with overlapping standard errors.

their values to be roughly the same order of magnitude as the translation magnitudes. Consider the "Direct Vector" method, which uses a classification PointNet++ to directly regresses to the action vector. We supervise this with the MSE loss. The results with and without scaling, also in Table S4, show that without scaling, the training collapses and the performance is zero. Nonetheless, despite strengthening the baseline with scaling of the targets, it remains worse versus ToolFlowNet.

When scaling targets, we "undo" the scaling at inference time when performing test rollouts.

B.4 Main Experimental Results

We report the main set of experimental results in Table 1 with the evaluation metrics described in Appendix A.2.2. As there are five independent BC runs, we report standard errors of the mean for each normalized success rate metric.

The results indicate that ToolFlowNet outperforms baselines and ablations, on average, across all the task and action variants. It has the highest average normalized success rate of 0.892, while the next highest baseline is RGBD+S Direct Vector with an average of 0.753 across the four evaluated tasks and actions.

We also observe an intriguing result with the no skip connection ablation of ToolFlowNet, in that it has strong performance on ScoopBall but never succeeds on PourWater. From inspecting the policy rollouts, we find that without skip connections, the policy cannot perform any rotations. This occurs because if there are no skip connections, then in the upsampling procedure of segmentation PointNet++ (i.e., the interpolation layers), the same latent vector is copied to every point, so the final predicted flow is the same for every point. When SVD converts the flow to a transformation, this results in a translation-only transformation with no rotation. Upon further analysis, this is due to global pooling layer in the middle of the architecture [15].

For further analysis, Table S5 reports the same experimental runs and settings as Table 1, except with an alternative evaluation metric. To clarify, other than for the current analysis in this subsection, we do *not* use this alternative evaluation metric anywhere else in the paper. Here, instead of taking the best snapshot among all saved snapshots (each is associated with an epoch, and saved once every 25 epochs), we average the normalized performance across all 20 epochs from 25, 50, and so on, up to 500, and take another average over random seeds, and report that. The advantage of this metric is that it may be more robust to noisy evaluation rollouts as it would average across the full training history. Moreover, it can be useful if one cares more about convergence speed. From analyzing Tables 1 and S5, we find that the results are consistent among both evaluation metrics, with both suggesting that ToolFlowNet outperforms other methods. From Table S5, ToolFlowNet

Method	Loss	Dense PN++?	R. Success ScoopBall 4D Demo: 0.632	R. Success ScoopBall 6D Demo: 1.000	R. Success PourWater 3D Demo: 0.906	R. Success PourWater 6D Demo: 0.815	Average R. Success
PCL Direct Vector	MSE	✗	0.344±0.02	0.848±0.05	0.480±0.07	0.328±0.03	0.500
PCL Direct Vector	PM	✗	0.144±0.08	0.048±0.04	0.120±0.07	0.072±0.03	0.096
PCL Dense Transformation	MSE	✓	0.328±0.04	0.824±0.06	0.488±0.04	0.280±0.02	0.480
PCL Dense Transformation	PM	✓	0.232±0.04	0.360±0.10	0.528±0.03	0.040±0.02	0.290
D Direct Vector	MSE	✗	0.120±0.05	**0.952±0.02**	0.032±0.01	0.056±0.02	0.290
D+S Direct Vector	MSE	✗	0.464±0.07	**0.928±0.03**	**0.704±0.03**	0.248±0.02	0.586
RGB Direct Vector	MSE	✗	0.224±0.03	0.776±0.05	0.632±0.02	0.264±0.04	0.474
RGB+S Direct Vector	MSE	✗	0.424±0.04	**0.944±0.02**	**0.728±0.03**	0.288±0.03	0.596
RGBD Direct Vector	MSE	✗	0.264±0.06	0.920±0.02	0.664±0.06	0.288±0.02	0.534
RGBD+S Direct Vector	MSE	✗	0.464±0.07	**0.968±0.02**	**0.752±0.03**	0.392±0.02	0.644
ToolFlowNet, No Skip Conn.	PM+C	✓	0.624±0.05	0.304±0.06	0.000±0.00	0.000±0.00	0.232
ToolFlowNet, MSE after SVD	MSE+C	✓	0.056±0.03	0.792±0.09	0.448±0.02	**0.744±0.04**	0.510
ToolFlowNet, PM before SVD	PM	✓	0.496±0.05	0.880±0.05	0.560±0.04	0.552±0.04	0.622
ToolFlowNet, No Consistency	PM	✓	0.544±0.04	0.744±0.12	0.424±0.09	0.496±0.05	0.552
ToolFlowNet (Ours)	PM+C	✓	**0.728±0.05**	**0.952±0.02**	**0.720±0.05**	0.544±0.03	**0.736**

Table S6: These results are the *raw*, un-normalized success rates (R. Success) for the same set of experiments reported in Table 1, which normalizes success rates by dividing them by the raw demonstrator performance. The number after "Demo:" in the table shows the raw demonstrator success rate. Since ScoopBall 6D has a demonstrator performance of 1.000, the raw values are the same as the normalized values reported in Table 1, but the other columns will show different values.

Method	Loss	Dense PN++?	R. Success ScoopBall 4D Demo: 0.632	R. Success ScoopBall 6D Demo: 1.000	R. Success PourWater 3D Demo: 0.906	R. Success PourWater 6D Demo: 0.815	Average R. Success
PCL Direct Vector	MSE	✗	0.258±0.01	0.640±0.03	0.305±0.04	0.215±0.01	0.354
PCL Direct Vector	PM	✗	0.081±0.04	0.002±0.00	0.041±0.02	0.034±0.00	0.040
PCL Dense Transformation	MSE	✓	0.270±0.01	0.669±0.06	0.337±0.02	0.172±0.02	0.362
PCL Dense Transformation	PM	✓	0.148±0.02	0.158±0.04	0.286±0.01	0.016±0.01	0.152
D Direct Vector	MSE	✗	0.075±0.02	0.744±0.02	0.012±0.00	0.016±0.00	0.212
D+S Direct Vector	MSE	✗	0.196±0.03	0.804±0.03	0.594±0.03	0.188±0.01	0.446
RGB Direct Vector	MSE	✗	0.134±0.02	0.646±0.01	0.550±0.02	0.176±0.01	0.377
RGB+S Direct Vector	MSE	✗	0.206±0.02	**0.872±0.02**	**0.665±0.01**	0.146±0.01	0.472
RGBD Direct Vector	MSE	✗	0.166±0.02	0.817±0.02	0.600±0.02	0.180±0.01	0.441
RGBD+S Direct Vector	MSE	✗	0.267±0.03	**0.883±0.02**	**0.646±0.03**	0.185±0.01	0.495
ToolFlowNet, No Skip Conn.	PM+C	✓	**0.485±0.02**	0.130±0.02	0.000±0.00	0.000±0.00	0.154
ToolFlowNet, MSE after SVD	MSE+C	✓	0.007±0.00	0.643±0.04	0.293±0.01	**0.492±0.03**	0.359
ToolFlowNet, PM before SVD	PM	✓	0.348±0.01	0.708±0.03	0.372±0.02	0.351±0.02	0.444
ToolFlowNet, No Consistency	PM	✓	0.370±0.02	0.461±0.04	0.262±0.10	0.306±0.03	0.349
ToolFlowNet (Ours)	PM+C	✓	**0.514±0.01**	0.799±0.02	0.627±0.01	0.437±0.01	**0.594**

Table S7: The raw, un-normalized results from Table S5 which reports normalized success rates computed by taking the average performance over all evaluation epochs (instead of taking the maximum as in Tables 1 and S6). As with Table S6, we repeat the demonstrator raw performance after "Demo:" in the relevant columns.

gets the highest average normalized success of 0.710. The next best method is RGBD+S Direct Vector again, with 0.561 average success.

For a more complete set of results, we also present additional tables that show the *raw* success rate instead of the normalized success rate. The results with the raw success rate are shown in Table S6 which corresponds to normalized results in Table 1, and Table S7, which corresponds to normalized results in Table S5.

B.5 Deep Reinforcement Learning Baseline

To get a rough sense of how RL compares against IL, we try SAC-CURL [65] from the open-source SoftAgent repository[4] on the PourWater and ScoopBall environments using RGB image inputs. For both environments, and for both action variants for each environment, we train SAC-CURL for 1 million training steps (i.e., environment interaction steps) and perform 10 test-time evaluation steps every 5000 steps. We run 3 random seeds for each experiment setting. We use dense rewards for

[4]https://github.com/Xingyu-Lin/softagent

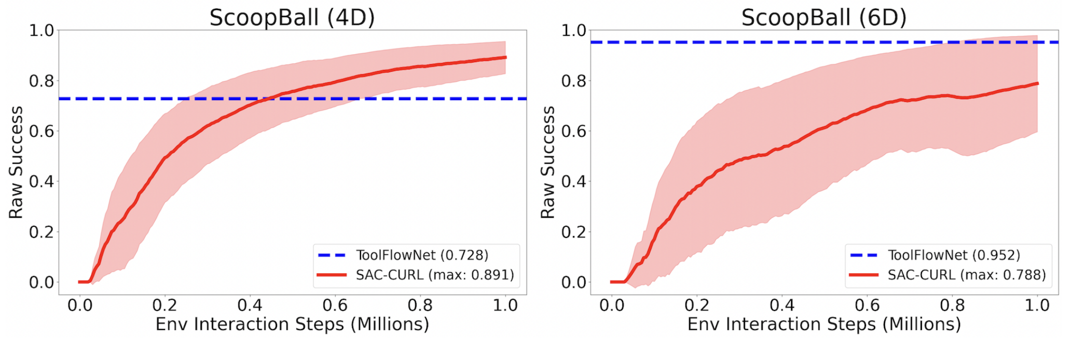


Figure S2: Performance of SAC-CURL on ScoopBall with the two action space variants we test in this paper (4D and 6D). We show the raw (not normalized) test-time success rate, and the curve is smoothed and averaged over 3 random seeds. For comparison, we overlay the performance of ToolFlowNet. The legend contains the performance of ToolFlowNet and the maximum performance along the SAC-CURL performance curve.

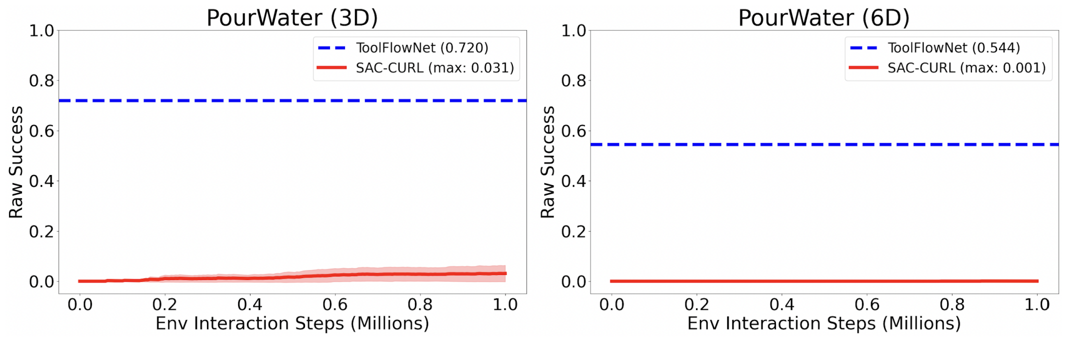


Figure S3: Performance of SAC-CURL on PourWater with the two action space variants we test in this paper (3D and 6D). The plot is formatted in a similar manner as in Figure S2.

both environments. ScoopBall's dense reward is the relative height of the ball, and PourWater's dense reward is the fraction of water particles inside the target.

Figures S2 and S3 show the SAC-CURL performance curves for ScoopBall and PourWater, respectively. We plot the performance curve for SAC-CURL and smooth it using an exponentially weighted averaging. We also take the performance of the ToolFlowNet policy (using *raw* success, from Table S6) and plot its performance in the figures with horizontal dashed lines.

On ScoopBall 4D, SAC-CURL obtains a maximum success rate of 0.891 after 1 million training steps. While this is an impressive raw performance, it required over 400,000 steps of environment interaction before surpassing the performance of the ToolFlowNet policy. For ScoopBall 4D, ToolFlowNet learned from 100 demonstration episodes of 100 time steps each, resulting in a total of just 10,000 (offline) state-action pairs. The SAC-CURL policy learns to avoid scooping the water, since accumulating water in the ladle causes unstable physics in that water tends to push the ball out of the ladle's control. This may explains the lower success rate of ToolFlowNet compared to SAC-CURL, because ToolFlowNet was imitating a demonstrator which scooped water.

For ScoopBall 6D, ToolFlowNet achieves a higher success rate of 0.952 because it imitates a much more reliable demonstrator and uses a ladle which allows water to pass through it, which addresses some physics instability. The 0.952 value is higher than the final average achieved by SAC-CURL (0.788) even after 1 million environment steps.

The results for PourWater for both action variants show an even more pronounced benefit for imitation learning using ToolFlowNet over SAC-CURL. Even after 1 million environment interaction steps, SAC-CURL gets *close to zero* binary success rate for both variants of PourWater, whereas

ToolFlowNet is significantly more reliable with raw success rates of 0.720 and 0.544 for 3 DoF and 6 DoF action spaces, respectively.

As shown on the project website[5], the policies learned from SAC-CURL tend to qualitatively look jerkier and more unstable compared to policies from imitation learning. Overall, these results may provide evidence for the benefits of imitation learning in these environments over reinforcement learning. An interesting future direction to explore for these tasks would be to combine imitation learning with reinforcement learning [45, 69, 70].

B.6 State-Based Policy Baseline

Method	ScoopBall 4D	ScoopBall 6D	PourWater 3D	PourWater 6D	Average
State (G.T.) Direct Vector	**1.152±0.04**	0.336±0.06	**0.768±0.02**	**0.785±0.07**	0.760
State (Learned) Direct Vector	0.835±0.12	0.824±0.06	0.433±0.07	0.226±0.03	0.579
ToolFlowNet [†]	**1.152±0.07**	**0.952±0.02**	**0.795±0.05**	0.667±0.03	**0.892**

[†]These results are directly from Table 1.

Table S8: Normalized success rates on the four task and action space combinations explored in the paper. We compare ToolFlowNet with state-based policies; see Section B.6 for more details.

As another set of baselines, we consider state-based policies which assume access to ground-truth tool and object poses. For ScoopBall, the state input is a concatenated vector of the 7D ladle pose (position and quaternion) and the 3D center of the ball, resulting in a 10D state vector. For PourWater, the state input is a concatenated vector of the state of the controlled box and the target box. Each box has 11 values in its state: its 3D center position, its 4D quaternion, its 3D dimensions (width, length, and height), and a 1D scalar representing the fraction of water particles in it. With two boxes, the state vector is thus 22D.

We train two variants of state-based methods, called **State (G.T.) Direct Vector** and **State (Learned) Direct Vector**, both trained with MSE on the action vectors. For State (G.T.) Direct Vector, we directly use access to the ground truth poses and pass that state information as input to an MLP policy network. The MLP policy network consists of a fully connected network with two layers of 256 nodes each with ReLU activations, producing a single 6D action vector output.

For State (Learned) Direct Vector, we first train a neural network policy which processes segmented point clouds as input and predicts the state information. (The segmented point clouds are the same type of inputs that we provide to ToolFlowNet.) The neural network policy is a PointNet++ built on the standard "classification" architecture for PointNet++. Then, we fix this network and, in a second training stage, use the output from this network as input to an MLP, which is trained to predict the actions. This MLP has the same architecture as in the State (G.T.) Direct Vector baseline. To clarify, even the "State (Learned) Direct Vector" baseline requires access to the ground-truth pose of objects in the environment during training in order to train the state estimators, whereas ToolFlowNet does not require access to such ground-truth state information.

In Table S8, we report the normalized success rates of the state-based policy baselines. The results suggest that State (G.T.) Direct Vector performs well. It attains similar performance as ToolFlowNet (in that standard error ranges overlap) on ScoopBall 4D and PourWater 3D, outperforms it on PourWater 6D, and performs much worse on ScoopBall 6D, though on average, ToolFlowNet performs slightly better (0.892 versus 0.760). For State (Learned) Direct Vector, performance is worse compared to ToolFlowNet on all experiments, and it only outperforms State (G.T.) Direct Vector on 6D pouring.

While State (G.T.) Direct Vector policies are able to achieve similar performance as ToolFlowNet, they assume access to ground-truth tool and object poses (and for PourWater, the fraction of water particles in the boxes). While knowledge of object poses has been be used in prior work for learning 6D pose transformations [71, 72, 73, 74], ToolFlowNet does *not* require access to such information.

[5]`https://tinyurl.com/toolflownet`

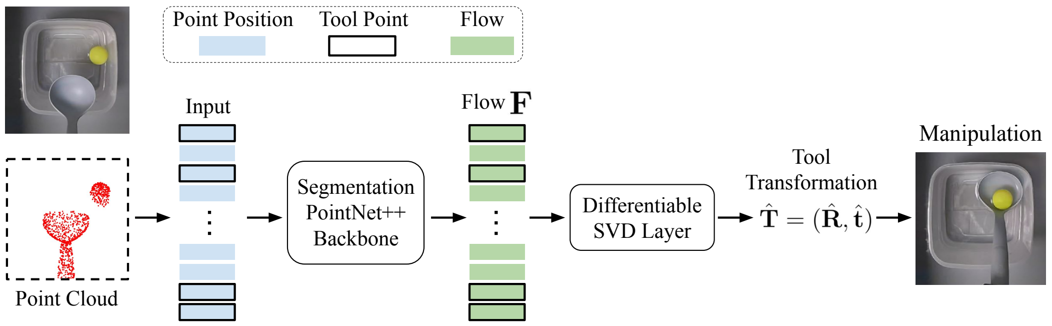


Figure S4: ToolFlowNet but with *non-segmented* point clouds as input. The network only takes in the 3D position coordinates of the input point cloud and uses all points during the SVD layer. We color the input point cloud only for visualization, and outline the tool points in bold to emphasize that both tool and non-tool points are provided to the SVD. See Section B.7 for further details, and Figure 2 for a reference comparison with the standard ToolFlowNet architecture.

B.7 ToolFlowNet with Non-Segmented Point Clouds

Method	ScoopBall 4D	ScoopBall 6D	PourWater 3D	PourWater 6D	Average
ToolFlowNet, Non-Segm	0.987±0.06	**0.928±0.01**	0.371±0.03	**0.579±0.08**	0.716
ToolFlowNet [†]	**1.152±0.07**	**0.952±0.02**	**0.795±0.05**	**0.667±0.03**	**0.892**

[†]These results are directly from Table 1.

Table S9: Normalized success rates of ToolFlowNet performance without segmented point cloud inputs ("Non-Segm" in the table) and comparing it with the standard input we use for ToolFlowNet. See Section B.7 for more details.

We investigate whether we can alleviate a key assumption we make for ToolFlowNet: that we require segmented point cloud observations as input. To modify ToolFlowNet so that it does not use segmentation information, we remove the per-point one-hot classification vector. Thus, the input point cloud consists only of the positions of each point, and has dimension $N \times 3$. Then, in the forward pass, the SVD layer uses the predicted flow from all points (both tool and non-tool). See Figure S4 for a visualization of this method. For supervision, we form the per-point, ground-truth flow labels by using a similar method as in the segmented point cloud version (see Section 3.2). As before, we apply the intended action from the demonstrator to transform points, except we do this to all points, not just the tool points.

Table S9 compares ToolFlowNet results with non-segmented point cloud inputs versus the standard point cloud inputs, under the same experimental conditions and metrics as in Table 1. We find that, on average, performance without per-point segmentation information is worse, as expected. Nonetheless, in ScoopBall 6D and PourWater 6D, the results with and without segmentation information have overlapping standard errors. Furthermore, the average value of 0.716 for ToolFlowNet without segmentation exceeds the average value for all of the baselines reported in Table 1 with the exception of RGBD+S Direct Vector, which has an average normalized success of 0.753. This suggests that even without segmentation information, ToolFlowNet can still be effective for imitation learning from point clouds.

In addition to these results, we explore another method for learning from segmentation-less point cloud inputs. The SVD layer can utilize learnable per-point weights which indicate how much value to weigh each point during the SVD forward pass. In the standard ToolFlowNet formulation, the weights are not learned and fixed to be 1 for all tool points (thus weighing each tool point equally) and 0 for non-tool points. We adjust this to use learnable weights during ToolFlowNet training, as we hypothesize that there might be enough supervision to learn weights with higher values for tool points and lower values for non-tool points. We implement this by having the forward pass through the segmentation PointNet++ architecture produce four output values, three for the standard flow

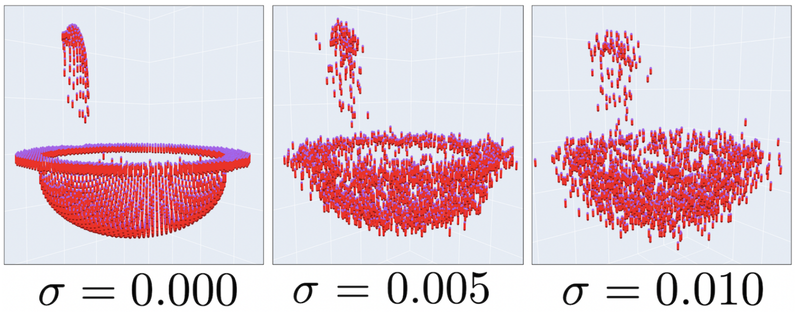


Figure S5: Examples of point clouds (blue points) and the corresponding flow (red vectors) visualizations for the tool (i.e., the ladle) for ScoopBall based on different noise injection levels. We test with $\sigma \in \{0.000, 0.005, 0.010\}$ as described in Appendix B.8. The samples above are from the training data, where the demonstrator happened to perform translation-only actions, so the flow vectors all point downwards and with the same magnitude.

predictions and one extra value for the per-point weights. We then pass these raw weights through a sigmoid layer, and then through a normalization layer before passing it to the SVD layer. However, the results for this method were worse than the approach presented earlier of assuming that the SVD layer uses all tool and non-tool points, each with equal weight.

B.8 Noisy Point Clouds

Method	StDev σ	ScoopBall 4D	PourWater 3D
ToolFlowNet	0.010	0.785±0.09	0.521±0.08
ToolFlowNet	0.005	**1.013±0.12**	0.503±0.06
ToolFlowNet [†]	0.000	**1.152±0.07**	**0.795±0.05**

[†]These noise-free results are directly from Table 1.

Table S10: Experiments with injecting Gaussian noise into point clouds. We bold the best results in each task's column and those runs with overlapping standard errors.

We next study Behavioral Cloning using ToolFlowNet when we alter the nature of the point clouds. To explore the potential for transfer to real settings with noisier sensor readings, we inject independent, identically distributed Gaussian noise to each point's 3D position in all training and testing data point clouds. See Table S10 for results with testing $\sigma \in \{0.000, 0.005, 0.010\}$ with two tasks; the no-noise case of $\sigma = 0.000$ is the default setting for other experiments in this paper. We only inject noise in the point cloud positions (for the tool and other items), and we do not perturb the demonstrator's tool flow vectors. The noise injection happens once to each point cloud in the training data and is fixed; this is different from adding noise each time a training data is sampled, which is a type of data augmentation. At test time, we apply a similar level of noise injection to each (new) point cloud observation.

It may be more meaningful to interpret σ values by comparing them with the size of the tool, since in simulation we can make the tool of arbitrary size. The value of 0.005 units in simulation is 2.7% of the radius of the ladle's bowl's for ScoopBall and 2.1% of the average box length in PourWater, where for the latter, we refer to the box that is the tool (the target box has similar dimensions). This is the average box length; in PourWater the box sizes are randomized, whereas in ScoopBall the ladle is of a fixed size. For visualizations of different noise injections, see Figure S5. This shows both point clouds (in blue) and the ground-truth flow vectors (in red) from the ScoopBall demonstrations.

The results suggest that ToolFlowNet may be robust to some levels of noise. In particular, for ScoopBall 4D, using $\sigma = 0.005$ means the best performance is 1.013 and nearly matches the 1.152 performance of the method in the noise-free case (both slightly outperform the demonstrator). As expected, in general with increasing noise, performance deteriorates, though interestingly, in PourWater 3D, using $\sigma = 0.010$ is slightly better than $\sigma = 0.005$.

B.9 Fewer Tool Points

Method and Task	#tool	Performance
ToolFlowNet, ScoopBall 4D	10	**1.063±0.09**
ToolFlowNet, ScoopBall 4D†	1284§	**1.152±0.07**
ToolFlowNet, PourWater 3D	10	**0.883±0.02**
ToolFlowNet, PourWater 3D†	633§	0.795±0.05

†Results are directly from Table 1.
§Represents the average number of tool points in a point cloud.

Table S11: Performance of ToolFlowNet based on using either a subset of 10 tool points, or the standard number of observable tool points.

In these experiments, we investigate the performance of ToolFlowNet while using different numbers of tool points in the point cloud. We use 10 points for PourWater 3D, based on 10 fixed keypoints located on the box. For ScoopBall 4D, we similarly use 10 keypoints located on the ladle. These form the 10 tool points in the segmented point cloud. In contrast, for experiments from Table 1, the ScoopBall and PourWater data have an average of 1284 and 633 tool points, respectively, per observation.

See Table S11 for results. Interestingly, using just 10 tool points seems to be sufficient for ToolFlowNet to imitate the demonstration data. Indeed, the version with PourWater even outperforms the one with the usual amount of tool points with 0.883 normalized performance versus 0.795.

This result should be interpreted with some nuance. First, we assume these 10 points are always available in the point cloud $\mathbf{P}$, even if they are occluded, which is in contrast to the standard experimental setup in this work where we use the observable point cloud, and hence, parts of the tool can be occluded. For example, in PourWater, the tool box frequently occludes itself, and when it gets close to the target box, the target box can also occlude parts of the tool box. Second, in order to get 5 complete Behavioral Cloning runs as per our evaluation metric in Appendix A.2.2, we had to run the 10 tool point case for PourWater 8 times. Of the 8 initial runs, 3 crashed due to an ill-conditioned matrix input to Singular Value Decomposition (SVD). This may suggest that extra tool points can add robustness to the SVD procedure and thus to ToolFlowNet, as we have never encountered this error in other experiments.

B.10 Number of Training Demonstrations

Method	# Demos	ScoopBall 4D	PourWater 3D
ToolFlowNet	10	0.620±0.22	0.256±0.07
ToolFlowNet	50	**0.975±0.11**	0.477±0.05
ToolFlowNet †	100	**1.152±0.07**	**0.795±0.05**

†Results are directly from Table 1.

Table S12: Performance of ToolFlowNet as a function of the number of training data demonstrations.

We standardize on 100 training demonstrations for simulation experiments for all tasks and demonstrations with the exception of the 6DoF ScoopBall task where we use 25 demonstrations. This is mainly due to the different tool which makes the task easier for policy learning. Here, we investigate the performance of ToolFlowNet as a function of the number of training data demonstrations for ScoopBall 4D and PourWater 3D. See Table S12 for the results, which indicate that while performance decreases with fewer demonstrations (as expected), ToolFlowNet can still be more sample efficient than alternative methods. In particular, for ScoopBall 4D, using *just 10 demonstrations* leads to a normalized success rate of 0.620, which outperforms other baselines from Table 1.

B.11 Baselines: Local vs Global Coordinates for Axis-Angle Rotations

Method	Frame	ScoopBall 4D	ScoopBall 6D	PourWater 3D	PourWater 6D	Average
PCL Direct Vector (MSE)†	Local	0.544±0.03	0.848±0.05	0.530±0.08	0.402±0.04	0.581
PCL Direct Vector (MSE)	Global	0.519±0.08	0.824±0.04	0.459±0.04	0.167±0.08	0.492
PCL Dense Transformation (MSE)†	Local	0.519±0.07	0.824±0.06	0.539±0.05	0.344±0.03	0.556
PCL Dense Transformation (MSE)	Global	0.646±0.09	0.824±0.03	0.494±0.05	0.216±0.02	0.545
ToolFlowNet †	N/A	**1.152±0.07**	**0.952±0.02**	**0.795±0.05**	**0.667±0.03**	**0.892**

†These results are directly from Table 1.

Table S13: Normalized success rates on the four task and action space combinations explored in the paper. We compare baseline methods of PCL Direct Vector and PCL Dense Transformation based on whether the target rotation (in axis-angle format) is expressed in local versus global coordinates. See Section B.11 for details.

For the baseline methods of PCL Direct Vector (MSE) and PCL Dense Transformation (MSE), we supervise the models with a 6D target vector, where 3 of the dimensions are for the 3D axis-angle rotation representation. The axis-angle is represented with respect to the local tool frame, centered at the ladle tip (for ScoopBall) or the bottom center part of the box (for PourWater).

Concurrent work which studies learning from point clouds has shown how the choice of coordinate frame for the points matter [75]. Motivated by this, we explore whether the baseline methods will improve when we adjust the frame for the axis-angle values, testing *global* axis-angle values with respect to the world frame. We only test with the MSE loss, and do not test the Point Matching loss, as the results from Table 1 showed that using the MSE loss for the baselines resulted in significantly better success rates.

We show the results in Table S13, which also compares against ToolFlowNet. Overall, we find that the choice of coordinate frame for expressing the rotation does not make a significant difference in our tasks. There is a slight boost towards using rotations expressed with respect to the local tool frame, but both baselines remain worse compared to ToolFlowNet.

B.12 Baselines: 4D, 6D, 9D, and 10D Rotation Representations

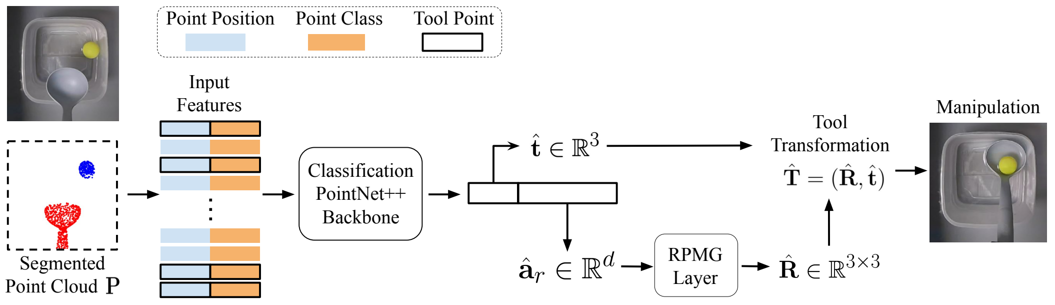
Method	Rotation	ScoopBall 4D	ScoopBall 6D	PourWater 3D	PourWater 6D	Average
PCL Direct Vector (MSE)†	3D	0.544±0.03	0.848±0.05	0.530±0.08	0.402±0.04	0.581
PCL Direct Vector (MSE)	4D	0.203±0.05	0.280±0.16	0.177±0.16	0.059±0.05	0.180
PCL Direct Vector (MSE)	6D	0.304±0.03	0.576±0.14	0.212±0.19	0.059 ±0.04	0.288
PCL Direct Vector (MSE)	9D	0.405±0.11	0.320±0.12	0.132±0.12	0.147±0.09	0.251
PCL Direct Vector (MSE)	10D	0.215±0.09	0.176±0.16	0.079±0.07	0.079±0.07	0.137
ToolFlowNet †	N/A	**1.152±0.07**	**0.952±0.02**	**0.795±0.05**	**0.667±0.03**	**0.892**

†These results are directly from Table 1.

Table S14: Experiments comparing normalized test-time success rates of PCL Direct Vector (MSE) with different rotation representations. The 3D rotation represents local axis-angle which we used for results in Table 1. See Section B.12 for details.

Prior work [60, 61, 68] has demonstrated that regressing to rotations using deep neural networks is challenging with 3D rotation representations such as axis-angle, which we use as our default (non-flow based) rotation representation. We thus perform experiments to check whether using alternative rotation representations can improve performance of the PCL Direct Vector MSE baseline. We test using 4D rotations (quaternions), 6D rotations [60], 9D rotations (rotation matrices) [54], and 10D rotations [61].

To implement this, we use a classification PointNet++ network which takes in the same segmented point cloud as input. Instead of the output as a vector in $\mathbb{R}^6$, as is the case for the PCL Direct Vector method, the output is a vector in $\mathbb{R}^{3+d}$, split into the translation prediction $\hat{\mathbf{t}} \in \mathbb{R}^3$ and a d-dimensional rotation vector $\hat{\mathbf{a}}_r \in \mathbb{R}^d$. We then pass the $\hat{\mathbf{a}}_r$ vector through the RPMG layer [68] to produce a (predicted) rotation matrix $\hat{\mathbf{R}} \in \mathbb{R}^{3\times 3}$. During backpropagation, the RPMG layer



Point Position
Point Class
Tool Point
Input Features
Segmented Point Cloud $\mathbf{P}$
Classification PointNet++ Backbone
$\hat{\mathbf{t}} \in \mathbb{R}^3$
$\hat{\mathbf{a}}_r \in \mathbb{R}^d$
RPMG Layer
$\hat{\mathbf{R}} \in \mathbb{R}^{3\times3}$
Tool Transformation
$\hat{\mathbf{T}} = (\hat{\mathbf{R}}, \hat{\mathbf{t}})$
Manipulation

Figure S6: PCL Direct Vector baseline, adjusted to test different rotation representations. With a standard segmented point cloud as input, it uses a classification PointNet++ to output a single vector, split into a translation $\hat{\mathbf{t}}$ and a rotation $\hat{\mathbf{a}}_r$ component. For $\hat{\mathbf{a}}_r \in \mathbb{R}^d$, we test 4D, 6D, 9D, and 10D rotation representations ($d \in \{4, 6, 9, 10\}$), and use an RPMG layer to project $\hat{\mathbf{a}}_r$ to a rotation matrix. See Section B.12 for details.

produces gradients through the rotation representation, by taking gradients on the $SO(3)$ manifold for the rotation representations. To reduce the chances of implementation errors, we directly reuse the layer from the RPMG [68] code.[6] See Figure S6 for a visualization of the architecture.

The RPMG layer introduces two hyperparameters, λ and τ. Following the RPMG paper, we fix $\lambda = 0.01$ and adjust τ throughout training by initially setting it to $\tau_{\text{init}} = 0.05$ and then increasing it to $\tau_{\text{converge}} = 0.25$ at the end of 500 Behavioral Cloning training epochs.

To train this model with the RPMG layer, we optimize the sum of translation and rotation losses. For translation, we use mean-square error, and for rotation, we follow the RPMG paper and minimize the Frobenius norm of the difference between the predicted and ground truth rotations: $\|\hat{\mathbf{R}} - \mathbf{R}^*\|_F$. We apply equal weight to the translation and rotation losses.

We show the Behavioral Cloning results with different rotation representations in Table S14, and compare with the standard 3D axis-angle rotation representation and ToolFlowNet. The results suggest that none of the alternative rotation representations offer performance benefits. We have also tried using the point matching loss instead of adding separate MSE and Frobenius norm losses, but the results were worse and we do not report them.

[6] https://github.com/jychen18/RPMG

C Physical Experiments

In this section, we discuss our physical experiments in more detail and present new results with more general starting configurations.

C.1 Physical Setup

The experimental setup consists of a Rethink Sawyer robot. We attach a standard consumer ladle to its gripper to make it feasible for the Sawyer to scoop a yellow floating ping-pong ball. We use Shining 3D EinScan-Pro to obtain the mesh of the ladle. We then convert this mesh to a point cloud that we query tool points from, while collecting ground-truth demonstrations and while running inference. A custom designed, 3D printed tool holder made of ABS plastic attaches the ladle to the end-effector.

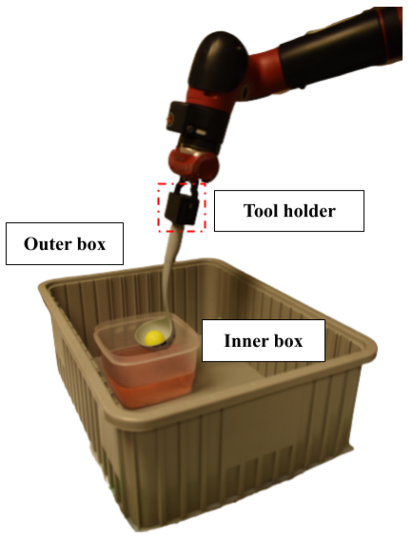


Figure S7: Closeup of inner and outer box and the tool holder.

An overhead Microsoft Kinect Azure camera continuously queries depth and RGB images of the scene, which we use to generate point clouds $\mathbf{P}$. Given the distinct yellow color of the target, we can segment the target points from the point cloud using HSV thresholding on the Kinect's RGB images. This gives us one of the two segmentation classes. At each time step, ROS's `tf` functionality queries the transformation between the Sawyer's base frame and the end-effector link. We apply this transformation to the scanned 3D model of the ladle to obtain a transformed model of the tool. We sample points from this transformed tool model to obtain the tool points. Through this technique, we obtain the second segmentation class, pertaining to the tool points. When collecting demonstrations, we track the changes in the pose of the ladle at consecutive time-steps to derive the tool flow. These form the observation-action pairs to train ToolFlowNet.

At the start of each demonstration and each test-time trial, we drop a yellow ping-pong ball in a translucent box in Figure S7 which contains water.

The water contains red food coloring to provide better color contrast for accurate HSV segmentation of the ping-pong ball. We tape the inner box within a larger box, which is the outer, gray box in Figure S7; this helps to contain spills and to prevent the smaller box from sliding. The gray box we use is from MSC Industrial Direct Co. and is a 100 Lb Load Capacity Gray Polypropylene Dividable Container with dimensions 22.5 inches long, 17.5 inches wide, and 8 inches tall.

C.2 Experiment Details

The demonstrations only describe translation motions, and we use the Sawyer's impedance controller to avoid end-effector rotations. We will test the model's performance in the physical environment with rotations in future work. One author of this paper collected all the training demonstrations.

During initial physical tests with collecting demonstration data, we noticed that ground-truth translations were roughly 2 mm to 3 mm in magnitude, which could result in small and jerky robot motions at test time from a learned policy. Thus, we compose the ground truth actions until their magnitude is at least 1 cm. These composed actions then become the ground truth training targets for the point cloud observations at each time step. Figure S8 depicts the variable composing method. In this example for the flow at time step t and $t+1$, we compose n actions to generate the flow $\mathbf{F}'_{\mathrm{t}}$ and $\mathbf{F}'_{\mathrm{t+1}}$ respectively, which both have a magnitude of at least 1 cm. While training ToolFlowNet on the physical experiment data, we scale the ground truth actions so that their values roughly lie in the

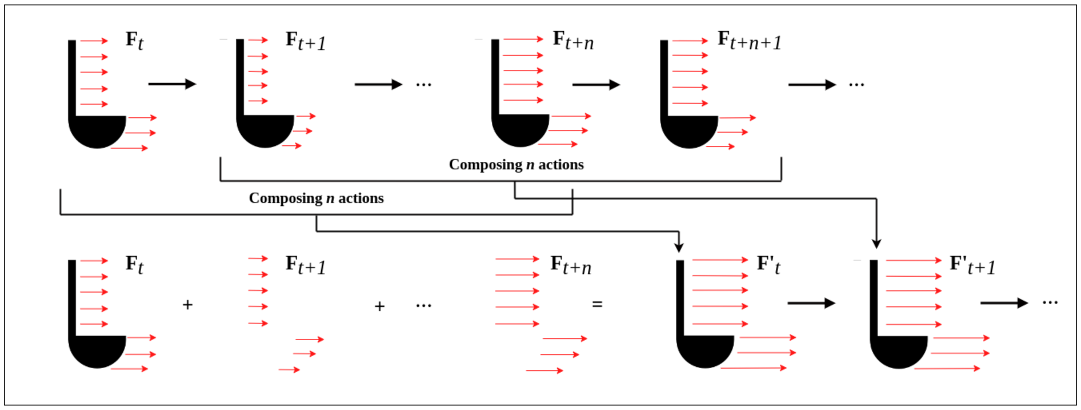


Figure S8: Visualization of the variable composing technique used in the (translation-only) physical experiments. We compose the ground-truth action targets until the composed flow vectors are at least 1 cm. In the figure, for time-step t, n actions are composed together to generate the variably composed flow represented as $\mathbf{F}'_{\mathrm{t}}$, which replaces the flow $\mathbf{F}_{\mathrm{t}}$, which has a magnitude less than 1 cm. Similarly, for time step $t+1$, n actions are composed together to generate the new flow, $\mathbf{F}'_{\mathrm{t+1}}$, replacing the original flow, $\mathbf{F}_{\mathrm{t+1}}$.

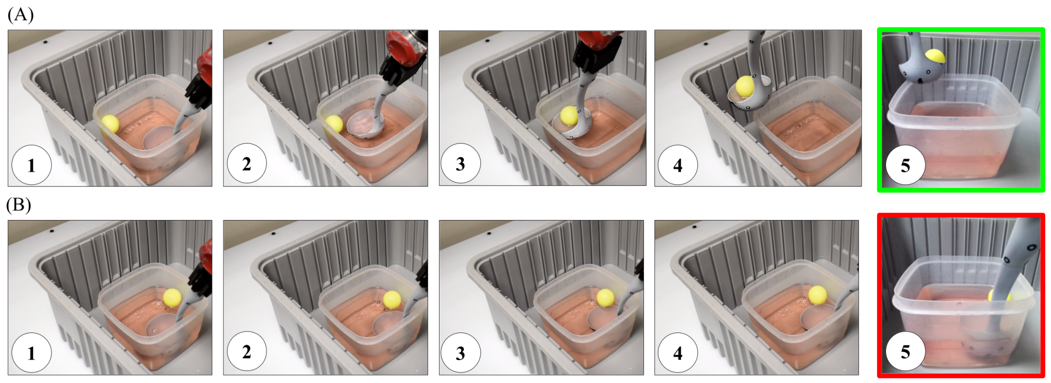


Figure S9: Subsampled frames from the testing trials during the physical experiments. Frame 1 shows the starting location of the target. The alternative camera view in frame 5 shows the height to which the robot lifts the target at the end of the trial. A) Frames from a scooping success, where the model successfully locates the ball (frame 2) and then manages to raise it above the top of the inner, translucent box (frame 4). Frame 5 shows the side view, where it is apparent that the robot has lifted the target, well over the top of the inner box. B) Frames from a scooping failure, where the robot was not able to locate or lift the ball to the top of the inner box due to collisions with the bottom right corner of the inner box. Frame 5 in the bottom row shows the ladle still submerged in the water.

range of -1 to +1, similar to the protocol followed for the simulation experiments (see Section B.3). At test time, we downscale the actions predicted by the network with the same factor.

The Sawyer is controlled by a computationally lightweight computer, which lacks the ability to run GPU intensive inference using the trained ToolFlowNet model. Furthermore, the Sawyer is controlled using ROS 1, which runs on Python 2, whereas we train ToolFlowNet using Python 3. At each time step, we therefore send the point cloud observations to a separate, more powerful GPU-enabled machine with Python 3 to run inference using ToolFlowNet and generate the necessary action commands. To interface the control computer with the GPU-enabled machine, we utilize Python bindings from ZeroMQ [76], called `pyzmq` to create a SSH tunnel between the two machines.

C.2.1 Experiment Protocol

We judge the performance of ToolFlowNet on whether the Sawyer successfully scoops the ping-pong ball (i.e., the target) out of the water without colliding with the rest of the experimental setup.

In Section 4, we report the success rate of ToolFlowNet in experiments where the target was dropped at some arbitrary location inside the smaller inner box. At the start of each trial, we initialize the ladle such that its bowl is just under the surface of the water. The robot then executes actions predicted by ToolFlowNet, in order to scoop the ball out of the water. The robot scoops the ball in an average of 17 time steps.

In Section 4, all the failures occur when the Sawyer repeatedly pushes its ladle against the walls of the inner box. Subsampled frames from a successful trial and a collision failure are shown in Figure S9, rows (A) and (B), respectively. Collision failures are better conveyed through videos and can be found on the project website: `https://tinyurl.com/toolflownet`.