An Efficient Image-to-Image Translation HourGlass-based Architecture for Object Pushing Policy Learning

Marco Ewerton, Angel Martínez-González, Jean-Marc Odobez

***Abstract*— Humans effortlessly solve pushing tasks in everyday life but unlocking these capabilities remains a challenge in robotics because physics models of these tasks are often inaccurate or unattainable. State-of-the-art data-driven approaches learn to compensate for these inaccuracies or replace the approximated physics models altogether. Nevertheless, approaches like Deep Q-Networks (DQNs) suffer from local optima in large state-action spaces. Furthermore, they rely on well-chosen deep learning architectures and learning paradigms. In this paper, we propose to frame the learning of pushing policies (where to push and how) by DQNs as an image-to-image translation problem and exploit an Hourglass-based architecture. We present an architecture combining a predictor of which pushes lead to changes in the environment with a state-action value predictor dedicated to the pushing task. Moreover, we investigate positional information encoding to learn position-dependent policy behaviors. We demonstrate in simulation experiments with a UR5 robot arm that our overall architecture helps the DQN learn faster and achieve higher performance in a pushing task involving objects with unknown dynamics.**

I. Introduction

Pushing is an important motor skill that can serve different purposes like moving objects that are too large, too heavy, too distant to be grasped, or in heavily cluttered scenarios. As such, several types of pushes can help to solve many manipulation tasks. For example, pushes can be used to isolate objects, reorient them to improve the chances of grasping, bring them closer or put them into a container. Pushing can also aid perception by improving object segmentation. It is interesting to note that 25 out of the 50 manipulation tasks of the Meta-World benchmark involve pushing [1]. Numerous works have investigated pushing tasks from several perspectives, e.g., creating a dataset of pushes [2] or studying and modeling their dynamics [3], [4], [5]. However, deciding whether to push or not, which push to perform, where, and how to do it to help manipulation remains an open problem.

To address this later problem, Zeng et al. [6] proposed an interesting approach relying on a model-free deep reinforcement learning (RL) method (DQN) to synergistically perform pushing and grasping. In their learning approach, RGB-D images are mapped using two Fully Convolutional Networks (FCNs) to a discretized action space. The first FCN maps RGB-D images to the utility of pushing 10 cm to the right at each pixel. The second FCN maps the images to the utility of grasping at each pixel. By presenting these networks with the input RGB-D image rotated at 16 different angles, the networks can compute the utilities of pushes and grasps with different orientations. For training, Q-learning [7] is used, relying on positive rewards for successful grasps and for pushes that produce detectable changes to the environment, resulting in the synergistic selection of pushes and grasps. This seminal work and idea can also be found in other papers studying skill synergies, e.g., between grasping and throwing [8], or suction and grasping [9].

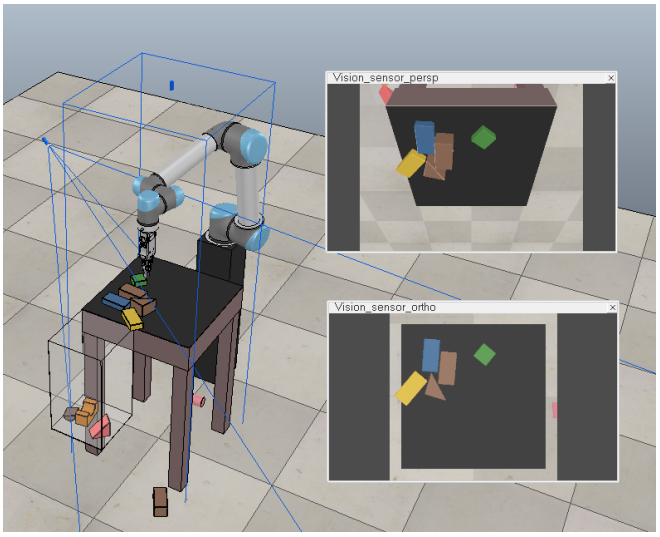

Fig. 1. Push into Box task. The robot receives depth images as inputs and has to decide where to push so that all the objects land inside the box. The input depth images are orthographic projections of the images captured by the camera on the left. Training a Deep Neural Network to process the scene and decide which push to perform presents challenges, since the best pushes depend on object and box positions, as well as on the physical properties of the objects and of the table surface, which are not known by the learning agent.

Despite their successes, these approaches present several shortcomings in their network architectures. First, they all rely on image classification network architectures (DenseNet [6], ResNet [8], [9]) which progressively and drastically downscale input images before classification. In these works, bilinear upscaling is used in the last layers of their proposed networks, after the DenseNet or ResNet modules, considerably reducing the sampling accuracy of the action space when applying the policy. Even if in theory every pixel position could be sampled, selecting the action corresponding to the maximum state-action value leads to only selecting a

The authors are with the Idiap Research Institute, CH-1920, Martigny, Switzerland {marco.ewerton, angel.martinez, jean-marc.odobez}@idiap.ch

The research leading to these results has received funding from the Swiss National Science Foundation through the HEAP project (Human-Guided Learning and Benchmarking of Robotic Heap Sorting, ERA-net CHIST-ERA).

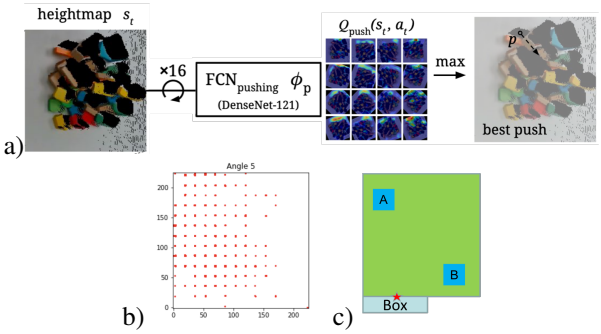

Fig. 2. a) Architecture used in [6] (similar in [8], [9]), in which the same image classification network followed by upsampling is applied to 16 rotations of the same input image. b) Sampling locations of pushes for one direction when applying a policy learned with this architecture, showing the important loss in action space accuracy when applying the learned policy. c) Fully convolutional networks are translation (position) invariant. Tasks, however, may require different outputs depending on object positions. In the example, the cube in A should be pushed vertically, whereas the cube in B should be pushed horizontally.

position at the lowest resolution, as illustrated in Fig. 2b.

Secondly, as the *same* network is applied to rotated images, the network is agnostic about orientation-specific information (it does not know which rotated image it is processing). Although this is not a problem when selecting which push better breaks objects apart for grasping [6], it can be problematic when handling task direction asymmetries, as is the case in the task we are interested in, namely, pushing objects into a box, as illustrated in Fig. 1.

Thirdly, as the chosen networks are fully convolutional, they are translation invariant, meaning that, for instance, when processing a single object on a table, they will likely predict very similar state-action values around this object whatever its position on the table. Such networks cannot handle task-dependent position asymmetries, as in our task. See Fig. 2c.

Finally, in [6], one network has to learn at the same time which pushes are valid (i.e., alter the scene) as well as those which are effective for the specific manipulation task (in their case, for making grasping possible). However, given the same input image and the same action parameterization, the set of valid pushes should be common to all tasks involving pushing (such as isolating objects, aligning them, bringing them closer). There is no need to learn from scratch to detect valid pushes whenever a new task involving pushing is learned.

Approach and contributions.

In this paper, we address the task of pushing objects into a box whose environment setup is shown in Fig. 1. To address this task, we adopted a similar DQN learning framework as in [6] to learn a pushing policy using deep convolutional neural network (CNN) agents, which entails several challenges, as discussed above, due to orientation and position asymmetries. In this context, we first argue that this is essentially an *image-to-image translation* problem where deep CNN agents predict the state-action value for all possible pushes given an input depth image. In this view, we propose to rely on an *Hourglass architecture* that is much more suitable than image classification networks to address such problems and is also computationally more efficient than common U-Networks [10] used for image-to-image translation tasks. We also solve the orientation asymmetry challenge by predicting state-action values for all orientations in one pass. Moreover, we propose to provide as input to the network the position of each pixel with respect to the box in addition its depth, which could better allow the network to make decisions based on the object location.

Secondly, in contrast to [6], we introduce a *masking method* that improves the sampling efficiency by constraining the set of possible actions to those effectively leading to changes in the environment. Our driving motivation is to exploit the collaboration between two different CNN agents, each focusing on different aspects of the task: pushing spot detection and task-specific pushing state-action value prediction.

Indeed, decoupling the task with two different agents brings several benefits. First, as our pushing spot detection agent has been trained to *identify* valid regions for pushing, it increases the sampling efficiency of the state-action value prediction agent since action sampling only needs to be conducted in a much smaller set. Second, training each agent independently allows for exploiting prior information regarding the task at hand and potentially prevents them from getting trapped in local minima as often observed with end-to-end agents. Finally, this formulation allows us to use different architectures for the decoupled tasks, possibly increasing the complexity for the more difficult one. In that context, our contributions are as follows:

- We propose to exploit the Hourglass CNN architecture in DQNs to learn object pushing policies, which leads to more efficient and more accurate high-resolution state-action value prediction and push selection.
- We investigate the use of positional information as network input to allow position-dependent prediction behavior.
- We propose a decoupled approach for solving the pushing task, introducing a pushing spot detection agent that works as a masking mechanism by identifying pushing actions leading to effective changes. Such an agent, once trained, can be exploited in any other manipulation task requiring pushes for other objectives.

We demonstrate in simulated experiments with a UR5 robot arm that our approach helps our DQN learn a pushing task faster and achieve better performance than the state-of-the-art.

II. Related work

For pushing, robots often rely on physics models. These models are computable but are only approximations of physical phenomena. For instance, Yu et al. [2] present a dataset of planar pushing experiments to study how reliable these models are, benchmark motion prediction methods, or for model learning. It shows that pushing can be seen as a stochastic process even if a highly precise manipulator is used to perform the pushes, highlighting the importance of learning to tackle such tasks. This is investigated in [4]

which, as in [8], uses the concept of Residual Physics, i.e., augments analytical models with data-driven techniques to compensate for the imperfections of the models. In [4], the trained neural networks not only correct the model predictions but also provide distributions over possible outcomes of actions. Hogan et al. [3] further address the problem of pushing an object on a plane along a desired trajectory or to pass through a sequence of via points. The proposed method consists of using Model Predictive Control and a family of mode sequences that have been designed by the authors. The main limitation of [3] is the necessity of hand-designing a specific family of mode sequences for each task the robot has to solve. In addition, all the above works only deal with single objects.

As mentioned in the introduction, Zeng et al. [6] use DQNs to synergistically perform pushing and grasping, an idea which has also been employed to explore synergies between other movements such as grasping and throwing of arbitrary objects [8]. In the latter, a perception module computes a feature representation of an RGB-D image, and a simple physics model is used to estimate the release velocity necessary to throw an object at a certain target position. Both pieces of information (feature representation and estimated release velocity) are fed into the grasping and throwing modules. The grasping module computes the utility of a grasp at each pixel. The throwing module computes, for each possible grasp, the residual on top of the estimated velocity to account for phenomena not accounted for by the simple physics model. The perception, grasping, and throwing modules are fully convolutional networks, and rewards are defined based on whether or not the thrown object landed in the correct box. Both papers [6], [8] suffer from limitations in their network architectures.

The methods proposed in [6] and [8] succeed in learning to push, grasp and throw objects in part due to the discretization of the action space, which simplifies the reinforcement learning problem. In contrast, other methods do not require this assumption [11], [12]. For instance, [11] uses Logic-Geometric Programming [13] and Multi-Bound Tree Search [14] to find and optimize a sequence of continuous actions to solve tasks involving the usage of tools and the manipulation of objects. Notwithstanding its success, some assumptions have been made (e.g. that each object has a sphere-swept convex geometry) to make this optimization problem numerically solvable, which may be difficult to hold in real-world applications. We thus resorted to the DQN framework of [6] to solve our task.

Finally, Suh and Tedrake [5] propose a switch-linear model to deal with the problem of pushing little carrot pieces towards a certain region. They do not deal with variable height and their images are binary, which is limiting when working with arbitrary heaps of objects. Nevertheless, they show that a switch-linear model can surpass the performance of deep learning models for this task because the tested deep models can get stuck in a loop, predicting actions that do not cause any change in the actual scene. The method we propose allows us to handle this issue by introducing a module specialized in identifying pushes that lead to changes in the environment. Once trained, this module can be used in conjunction with other networks to address any pushing task.

III. Approach

A. Overview

The proposed system is described in Fig. 3. It takes as input a depth image, complements this information with positional information, and processes the result with the *Push-into-Box* architecture (termed *PushInBox*), which delivers an estimate of the state-action value for each possible robot pushing action, as parameterized by its position and orientation. At run time, the action having the highest state-action value is executed, and the new scene is observed.

Looking more in detail, the *PushInBox* network itself is composed of two HourGlass [15] subnetworks: the *PushMask* subnetwork, whose aim is to spot all the valid pushes, i.e., the pushes which result in scene changes; and the *PushReward* subnetwork, whose aim is to predict the state-action value Q that results from the reward function that we have defined to tackle the pushing task at hand.

First, we provide a more formal description of the different components of our framework, then introduce the different rewards involved in the learning, and finally the approach required for training the global network.

B. Modeling

State-space and network input. We represent the state s_t of the environment at time step t by its sensed depth image $\mathbf{I}_t$ with shape 224×224. Additionally, the input image can be complemented with 2D positional information of each point on the surface of the table with respect to the box. The image $\mathbf{I}_t$ is given as input to both *PushMask* and *PushReward* subnetworks.

Action-space. The robot can perform a pushing action of length $10cm$ at any (x, y) position in the image and in any one of 16 directions o. Hence, the action set $\mathcal{A}$ is the set of all the possible (x, y, o) pushes and is of dimension $224 \times 224 \times 16$. Each subnetwork outputs one scalar for each action. Therefore, the output of each subnetwork is composed of 16 layers of size 224×224.

Network architecture. We use an Hourglass architecture for each of our subnetworks to produce accurate and high-resolution predictions. Each of them comprises a U-Net-like network [10] that processes image features in an encoding-decoding fashion, from high-resolution to low-resolution and back to high-resolution. The image is first processed by a set of convolution filters to extract a feature representation of the image. In the downward part of the network, a set of residual blocks r_d progressively filters the features from the previous resolution and downsamples the result, until it reaches the bottleneck layers, which correspond to an encoding of the image at a small resolution (128 layers at a 14×14 resolution in our case). The upward part of the network comprises a set of residual blocks r_u, each of them processing the

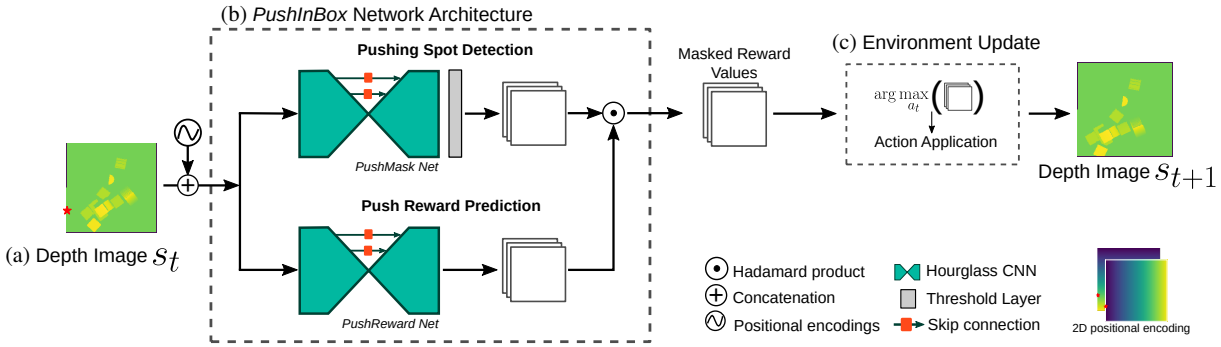

Fig. 3. Proposed push into box agent collaboration pipeline. (a) The environment state s_t (a depth image of the scene) is complemented with positional information and used as input to the *PushInBox* network. This input is processed by a *PushReward* network predicting the state-action value Q for each possible push, and by the *PushMask* predicting whether a push may lead to a scene change. (b) The pushing state-action values are masked by the outputs of the *PushMask* through elementwise multiplication to only select relevant pushes. (c) The environment is updated by applying the push that maximizes the masked state-action value maps and produces a new environment state s_{t+1}.

output from the previous (lower) resolution followed by upsampling. Each r_u also integrates information from the same resolution thanks to skip connections coming from the same resolution blocks in the downward part, allowing to combine the different semantic levels while maintaining the localization accuracy of the network prediction.

C. Rewards

We introduce reward functions specific to each subtask, i.e., valid pushing spot detection and pushing objects into the box.

Pushing spot detection reward. We design this reward under the premise that pushing actions should produce *effective changes*. More precisely, given the environment state at time t, s_t (the depth image $\mathbf{I}_t$), and the executed action a_t leading to the next state s_{t+1} (image $\mathbf{I}_{t+1}$), we define a notion of change by counting the amount of local pixel changes

$$c_t = \sum_p I_{(|\mathbf{I}(p)_t - \mathbf{I}(p)_{t+1}| >= 1cm)}, \tag{1}$$

where I_v is the indicator function, equal to 1 if v is true, and 0 otherwise. Accordingly, we define the *PushMask* change reward

$$R^c\left(a_t, s_t, s_{t+1}\right) = \begin{cases} 0 & \text{if } c_t < \tau_{mask} \\ 1 & \text{otherwise} \end{cases}, \tag{2}$$

where τ_{mask} is a threshold which we set to 1000 in practice.

Push-into-box reward. We define an effective push as that which moves the objects towards the box, and hence will minimize the average distance between object pixels and the box. Accordingly, our measure of pushing effectiveness is defined as:

$$\Delta d_t^M = d_t^M - d_{t+1}^M \text{ with } d_t^M = \frac{1}{|\mathcal{O}_t|} \sum_{p \in \mathcal{O}_t} d_t(p), \tag{3}$$

where p is a pixel position in the image, $d_t(p) = \|p - p_{box}\|$ denotes the distance between pixel p and the center of the edge of the box in contact with the table p_{box}, $\mathcal{O}_t$ is the set of objects' pixels obtained by thresholding the scene depth image, and $|\mathcal{O}_t|$ denotes its cardinality. Finally we define the push-into box reward R^p as the sum of a distance reward $R_t^d = \max\left(0, \Delta d_t^M\right)$ measuring how much objects move closer to the box, as well as a reward $R_t^{ib} = 10 \times N_t^{box}$ proportional to the number of objects N_t^{box} falling into the box, unless one or more objects fall to the ground ($N_t^{gro} > 0$) where N_t^{box} and N_t^{gro} are provided by the environment simulator. More formally,

$$R^p\left(a_t, s_t, s_{t+1}\right) = \begin{cases} 0 & \text{if } N_t^{gro} > 0 \\ R_t^d + R_t^{ib} & \text{otherwise} \end{cases}. \tag{4}$$

D. Training protocol

In this section, we indicate the protocol followed to train our architecture, which involves both supervised training and reinforcement learning steps. The main steps are as follows:

1) Train *PushMask* using appropriate data. The parameters of this network are assumed to be frozen afterward.
2) Train *PushReward* using an offline dataset D_{rew}.
3) Train the whole *PushInBox* using online reinforcement learning, taking advantage of the *PushMask* to improve training. In this step, the parameters of *PushMask* are frozen.

Before describing the specific aspects of how the *PushMask*, *PushReward*, and *PushInBox* models were trained, we introduce the basic elements of reinforcement learning.

Reinforcement learning (RL). During RL, the system simulates a set of B experiences, which can be expressed as triplets (s_t, a_t, s_{t+1}). For each experience, we can compute the expected state-action value $Q(a_t, s_t)$ as predicted by the network. The output $Q(a_t, s_t)$ should be as close as possible to the target value

$$y\left(a_t, s_t, s_{t+1}\right) = R\left(a_t, s_t, s_{t+1}\right) + \gamma Q\left(s_{t+1}, \arg\max_{a'}\left(Q\left(a', s_{t+1}\right)\right)\right), \tag{5}$$

which can be computed from the definition of the reward and $\arg\max_{a'}\left(Q\left(a', s_{t+1}\right)\right)$. The latter can be computed by applying the network to the state s_{t+1} of the current experience. In other words, for an experience i, the temporal difference error $\delta_t^i = L_e(Q\left(a_t^i, s_t^i\right), y\left(a_t^i, s_t^i, s_{t+1}^i\right))$ must be minimized, where L_e is the loss used to measure the discrepancy between the prediction and the target. The loss for each mini-batch of size B is $Loss = \sum_{i=1}^{B} \delta_t^i$, which can

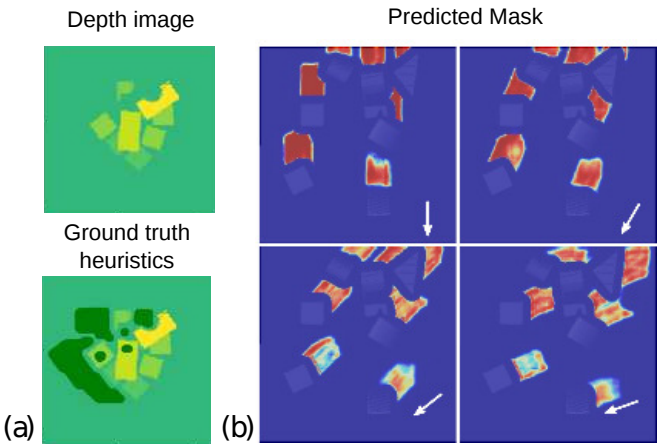

Fig. 4. (a) Example of generating ground truth with our proposed heuristics. Top: Input depth image; Bottom: generated ground truth push mask for pushes pointing to the right. (b) Example of generated valid pushing probability maps predicted by our *PushMask* for different pushing orientation possibilities on a sample image. The color red indicates start positions of pushes with a higher probability of leading to changes, while blue indicates a lower probability of leading to changes.

be optimized via gradient descent optimization algorithms and using backpropagation to compute the gradients. We relied on the Adam optimizer.

Training *PushMask*. To train the *PushMask* we first generate a training set D_m using a heuristics that identifies pixels close to the borders of objects as promising or not to push according to edge and orientation analysis. More precisely, we first generate a set of scenarios by tossing 10 randomly selected objects onto the table and generating their corresponding depth images. To compute the ground truth, we start by detecting edges using the Canny edge detector. Then, we label an edge pixel p to be a valid pushing position if $\mathbf{I}(p_\theta) - \mathbf{I}(p) > \delta$, where p_θ is the pixel p moved in a line along the pushing orientation possibility θ. The reasoning behind this procedure is that a push starting at p and passing through p_θ will probably meet an object border if p_θ is higher than p by a certain threshold δ, meaning that this push will probably produce some significant change. Fig. 4(a) shows an example of the ground truth generated with this heuristics for a given image.

Our goal is to train a network, *PushMask*, which identifies whether a given push a is leading to a change or not. In this regard, we have a binary classification problem for each a, and we thus rely on the binary cross-entropy loss L_e for training. The network is trained in two stages. In the first stage, we train *PushMask* using the dataset D_m, i.e., the ground truth masks generated by the described heuristics. However, as this heuristics does not account well for setup characteristics (gripper shape and size, friction, noise), we run a refinement training stage (in an RL fashion) that uses actual object pushes and Eq.(2) as ground truth for supervised learning. Fig. 4(b) illustrates our results on a sample image.

Training *PushReward* and *PushInBox*.

We divide the *PushInBox* training into two stages. In each stage, the network is trained to minimize the temporal difference errors δ_t^i.

In the first stage (offline RL) we train *PushReward* from scratch from a dataset D_{rew} of experiences obtained as follows: scenarios are generated at random, and the pushing actions are sampled uniformly at random 50% of the time, or using a trained *PushMask* otherwise. Specifically, in the latter case, we compute the forward pass of the *PushMask* on the current depth image and generate a binary mask of pushing actions. Then, we select pushing actions that lead to a change in the environment with probability greater than a certain threshold. These actions are executed in simulation and the corresponding experiences are recorded. We have chosen the probability threshold to select the actions based on the F-score obtained by different thresholds in a test set, as detailed in Section IV. Subsequently, *PushReward* is trained to minimize the temporal difference errors associated with the recorded experiences.

In the second stage (online RL) we further train *PushReward* in simulation, using the previously trained *PushReward* model as the starting point and exploring new experiences with an epsilon-greedy approach. We proceed as follows: given each new depth image experienced in simulation, we compute the forward pass of *PushReward* and *PushMask*. These predictions are then combined via the Hadamard product. After each new batch of experiences, the parameters of *PushReward* are optimized to minimize the squared error loss L_e associated with the experiences in this new batch.

The offline RL stage allows for an effective initialization of *PushReward* with pre-recorded experiences while the online RL stage further improves the network by exploring new experiences. In this work, for simplicity, *PushMask* is trained and its parameters are frozen before training *PushReward*. One could also consider training *PushMask* further together with *PushReward*.

IV. Experiments

A. Experimental protocol

Experimental protocol for *PushMask*. As this network is trained independently, we generated a test dataset to evaluate the performance of the different models. To that end, we generated a test set by randomly sampling 10240 scenarios involving 1 to 10 objects, along with random pushes. The ground-truth (GT) regarding these pushes has been determined as during training (i.e., thresholding the number of pixels that have changed after the push). This test set was used to produce precision-recall curves. The precision is defined as: out of the set of pushes identified by the system as valid (true), what is the proportion of pushes that are true (according to the GT). The recall corresponds to the proportion of true pushes correctly identified by the system.

Push-into-box experimental protocol. We created 100 random scenarios comprising exactly 10 objects. For each model under evaluation, we applied the corresponding policy, until either there was no more object on the table, or the policy

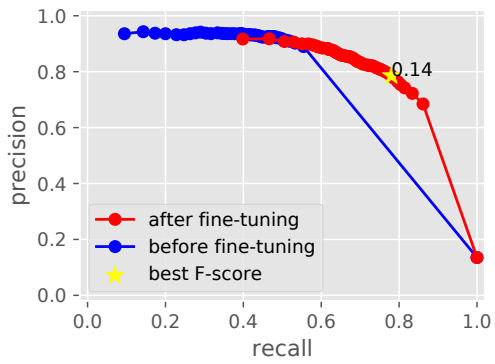

Fig. 5. Precision vs. recall curves of *PushMask* before fine-tuning (trained only based on a heuristics) vs. after fine-tuning with actual pushes. The model with probability threshold 0.14 obtained the best F-score.

did not produce any change 10 times in a row (implicitly implying that the model is pushing in the void).

Evaluation Metrics. We quantify the performance of our models by the mean and standard deviation of the number of objects successfully pushed into the box N_{ObjB}, number of objects that fell on the ground N_{ObjG}, and number of objects left on the table N_{ObjT}. Additionally, we measure the efficiency of our system by computing the ratio between the average number of objects successfully pushed into the box N_{ObjB} and the average number of executed actions N_{Act}.

Tested models. To evaluate the benefits of the different parts of our framework, we tested different algorithmic configurations. These elements are specified when reporting results in Table I.

- DENSENET architecture [6]: in this case, we used the architecture proposed by [6] as *PushReward* network, which is based on a DenseNet image recognition network pre-trained on ImageNet [16], with an improvement. Rather than predicting the reward for the 16 pushing directions by rotating the image 16 times and applying the DensetNet architecture, we modified the network so that it can predict the reward for all directions in the last stage of the network, as is done with our architecture.
- HGLASSN architecture: for the *PushReward* network.
- HGLASSN-POS architecture: the same as HGLASSN, but using an additional position encoding for the network input.
- MASK option: this indicates whether the *PushMask* network was exploited by the system, both during training in the online RL step and in testing.
- ONLINERL: whether the *PushReward* network was only trained with the offline dataset D_{rew} or if an additional RL fine-tuning with an epsilon-greedy exploration strategy was conducted (ONLINERL = Yes).

B. Results

***PushMask* results.** As shown in Fig. 5, training based only on the masks generated by the heuristics led to high precision but resulted in missing some of the pushes which produced changes (low recall). In this regard, the *PushMask* benefited from the fine-tuning with actual pushes. The fine-tuning led to a higher recall and a higher F-Score. The probability threshold of 0.14 to detect changes has achieved the highest F-Score and has been used in our further experiments involving the *PushMask*.

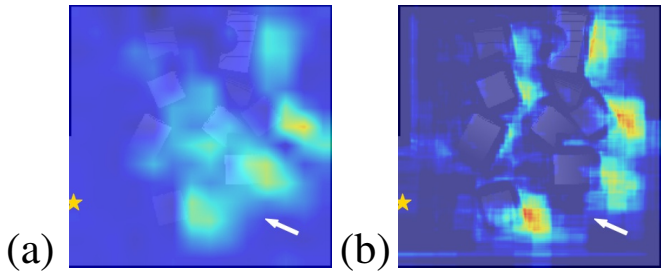

Fig. 6. Comparison of output state-action value maps for (a) Densenet, and (b) Hourglass architectures. The smoothing effects in (a) appear by producing the maps at very low resolution. A more precise localization of pushing action can be identified in the maps produced by the Hourglass network in higher resolution.

***PushInBox* results.** Our main quantitative results are summarized in Table I. The best result in each column is bold, the second-best, underlined. All the HourGlass-based models have surpassed the DenseNet-based ones in terms of the average number of objects correctly pushed into the box. Figs. 6 and 7 show state-action value predictions made by different models.

Comparing the HGLASSN models with and without the mask after online RL, we can observe that the mask only slightly improved the performance in terms of the average number of objects in the box and efficiency (fewer actions). Nevertheless, the mask has given the HGLASSN-POS model a significant boost in performance before online RL (8.05 objects on average in the box with the mask versus 6.71 without). We hypothesize that the mask helps to prevent models from sliding on the top of objects or pushing where there is no object.

The model with the highest performance overall has been the HGLASSN with mask and after online RL with discount factor $\gamma = 0$. By observing the behavior of this model during tests, we noticed that the main sources of problems preventing the model from pushing all the objects into the box are objects very close to the borders of the table, where the robot is unable to move them back to the center due to its limited workspace, and cylinders, which easily roll out of the table.

We had expected models with discount factor $\gamma = 0.4$ to perform better than the models with $\gamma = 0$, but this has not been the case. A possible explanation is that this task does not require learning complex successions of pushes to obtain higher expected future rewards.

Finally, while the model with linear position encoding performed best in offline learning, exhibiting a better behavior for carefully bringing objects in front of the box before pushing them into the box, its performance after online RL decreased. A closer look at the learning curve showed a relatively chaotic behavior beyond a certain number of epochs, suggesting that the selected optimizer and parameters

TABLE I

PUSH INTO THE BOX PERFORMANCE RESULTS OBTAINED BY TESTING OUR MODEL IN SCENARIOS WITH 10 OBJECTS

Architecture	ONLINERL	MASK	N_{ObjB} (std)	N_{ObjG} (std)	N_{ObjT} (std)	N_{Act}	$\frac{\text{avg.}N_{ObjB}}{\text{avg.}N_{Act}}$
DENSENET	No	No	4.56(2.1)	4.38(1.7)	1.06(2.2)	**18.0**(8.4)	0.25
DENSENET	Yes	No	5.38(3.2)	1.17(1.2)	3.45(3.9)	21.5(8.4)	0.25
HGLASSN	No	Yes	7.80(1.2)	2.15(1.1)	0.05(0.4)	23.7(4.8)	0.33
HGLASSN	No	No	7.76(1.4)	1.90(1.3)	0.34(1.2)	26.6(9.6)	0.29
HGLASSN-POS	No	Yes	8.05(1.7)	1.48(1.2)	0.47(1.6)	28.2(10.8)	0.29
HGLASSN-POS	No	No	6.71(2.6)	1.21(1.3)	2.08(3.0)	30.0(12.6)	0.22
HGLASSN, $\gamma = 0$	Yes	Yes	**8.96**(0.9)	1.01(0.9)	0.03(0.2)	21.5(4.6)	**0.42**
HGLASSN, $\gamma = 0$	Yes	No	8.94(1.0)	**1.00**(1.0)	0.06(0.4)	23.2(6.7)	0.39
HGLASSN, $\gamma = 0.4$	Yes	Yes	8.00(1.4)	1.98(1.4)	**0.02**(0.1)	20.6(6.9)	0.39
HGLASSN, $\gamma = 0.4$	Yes	No	7.61(1.6)	2.11(1.2)	0.28(1.4)	21.1(5.3)	0.36
HGLASSN-POS, $\gamma = 0$	Yes	Yes	6.66(1.8)	2.89(1.4)	0.45(1.8)	24.2(7.0)	0.28
HGLASSN-POS, $\gamma = 0.4$	Yes	Yes	6.59(1.7)	3.16(1.5)	0.25(1.3)	24.0(8.8)	0.27

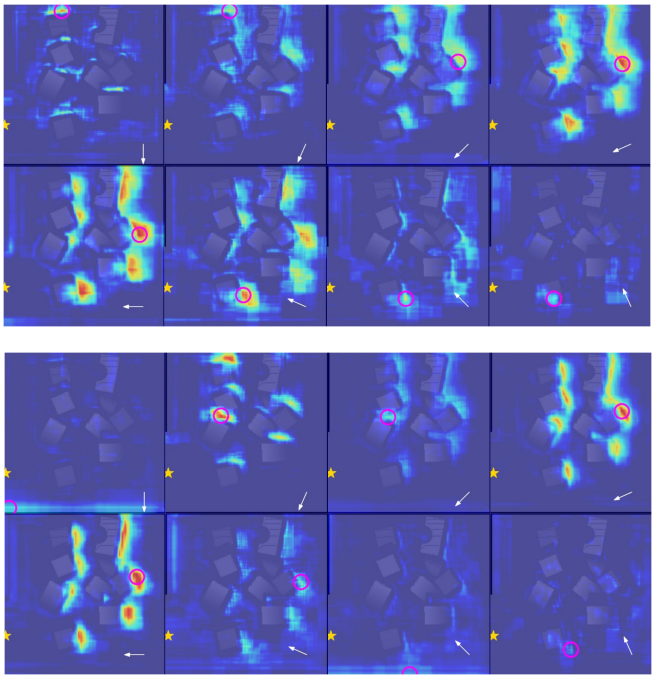

Fig. 7. Examples of state-action value map predictions for a set of pushing directions. a) HGLASSN architecture, using the *PushMask* network, and ONLINERL with $\gamma = 0$. b) Same as a), but with $\gamma = 0.4$. A white arrow shows the direction of the pushing action, the magenta circle center indicates the location of the maximum for that state-action value map and the golden star indicates the position of the center of the edge of the box in contact with the table.

are not optimal. This will be investigated in future work.

V. CONCLUSION

In this paper, we presented an approach for object pushing policy learning from visual data. We have addressed the task as an image-to-image translation and proposed to use Hourglass architectures to produce high-resolution state-action value maps given an input depth image. We divided the task into valid pushing spot detection and pushing state-action value prediction subtasks, and designed their rewards so that pushing actions effectively produce changes in the environment and solve the task at hand, in this case pushing objects into a box. Our results suggest that the use of the Hourglass architecture is better suitable for predicting more accurate pushing actions and is more efficient than other classification-based architectures such as DenseNet.

REFERENCES

[1] T. Yu, D. Quillen, Z. He, R. Julian, K. Hausman, C. Finn, and S. Levine, "Meta-world: A benchmark and evaluation for multi-task and meta reinforcement learning," PMLR, pp. 1094–1100, 2020.

[2] K.-T. Yu, M. Bauza, N. Fazeli, and A. Rodriguez, "More than a million ways to be pushed. A high-fidelity experimental dataset of planar pushing," in *IROS*. IEEE, 2016, pp. 30–37.

[3] F. R. Hogan and A. Rodriguez, "Feedback control of the pusher-slider system: A story of hybrid and underactuated contact dynamics," *Algorithmic Foundations of Robotics XII*, pp. 800–815, 2020.

[4] A. Ajay, J. Wu, N. Fazeli, M. Bauza, L. P. Kaelbling, J. B. Tenenbaum, and A. Rodriguez, "Augmenting physical simulators with stochastic neural networks: Case study of planar pushing and bouncing," in *IROS*. IEEE, 2018, pp. 3066–3073.

[5] H. Suh and R. Tedrake, "The surprising effectiveness of linear models for visual foresight in object pile manipulation," *arXiv preprint arXiv:2002.09093*, 2020.

[6] A. Zeng, S. Song, S. Welker, J. Lee, A. Rodriguez, and T. Funkhouser, "Learning synergies between pushing and grasping with self-supervised deep reinforcement learning," in *IROS*. IEEE, 2018, pp. 4238–4245.

[7] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[8] A. Zeng, S. Song, J. Lee, A. Rodriguez, and T. Funkhouser, "Tossingbot: Learning to throw arbitrary objects with residual physics," *T-RO*, vol. 36, no. 4, pp. 1307–1319, 2020.

[9] A. Zeng, S. Song, K.-T. Yu, E. Donlon, F. R. Hogan, M. Bauza, D. Ma, O. Taylor, M. Liu, E. Romo, *et al.*, "Robotic pick-and-place of novel objects in clutter with multi-affordance grasping and cross-domain image matching," in *ICRA*. IEEE, 2018, pp. 3750–3757.

[10] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *International Conference on Medical image computing and computer-assisted intervention*. Springer, 2015, pp. 234–241.

[11] M. Toussaint, K. Allen, K. A. Smith, and J. B. Tenenbaum, "Differentiable physics and stable modes for tool-use and manipulation planning." in *RSS*, 2018.

[12] R. Strudel, A. Pashevich, I. Kalevatykh, I. Laptev, J. Sivic, and C. Schmid, "Learning to combine primitive skills: A step towards versatile robotic manipulation," *ICRA*, pp. 4637–4643, 2020.

[13] M. Toussaint, "Logic-geometric programming: An optimization-based approach to combined task and motion planning," in *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[14] M. Toussaint and M. Lopes, "Multi-bound tree search for logic-geometric programming in cooperative manipulation domains," in *ICRA*. IEEE, 2017, pp. 4044–4051.

[15] A. Newell, K. Yang, and J. Deng, "Stacked hourglass networks for human pose estimation," in *ECCV*. Springer, 2016, pp. 483–499.

[16] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *CVPR*. IEEE, 2009, pp. 248–255.