A Multitask and Kernel approach for Learning to Push Objects with a Target-Parameterized Deep Q-Network

Marco Ewerton, Michael Villamizar, Julius Jankowski, Sylvain Calinon, Jean-Marc Odobez

***Abstract*— Pushing is an essential motor skill involved in several manipulation tasks, and has been an important research topic in robotics. Recent works have shown that Deep Q-Networks (DQNs) can learn pushing policies (when, where to push, and how) to solve manipulation tasks, potentially in synergy with other skills (e.g. grasping). Nevertheless, DQNs often assume a fixed setting and task, which may limit their deployment in practice. Furthermore, they suffer from sparse-gradient backpropagation when the action space is very large, a problem exacerbated by the fact that they are trained to predict state-action values based on a single reward function aggregating several facets of the task, rendering the model training challenging. To address these issues, we propose a multi-head target-parameterized DQN to learn robotic manipulation tasks, in particular pushing policies, and make the following contributions: i) we show that learning to predict different reward and task aspects can be beneficial compared to predicting a single value function where reward factors are not disentangled; ii) we study several alternatives to generalize a policy by encoding the target parameters either into the network layers or visually in the input; iii) we propose a kernelized version of the loss function, allowing to obtain better, faster and more stable training performance. Extensive experiments on simulations validate our design choices, and we show that our architecture learned on simulated data can achieve high performance in a real-robot setup involving a Franka Emika robot arm and unseen objects.**

I. Introduction

Solving manipulations tasks often requires the composition of several elementary motor skills. Pushing is one of them and plays an important role, as illustrated by the Meta-World robotic manipulation benchmark, in which half of the tasks involves pushing [1]. It can be used to isolate objects [2], reorient them to improve grasping [3], bring them closer, or put them into a container [4]. Pushing can also aid perception by improving object segmentation [5]. Hence, numerous works have investigated pushing tasks by creating datasets of pushing gestures [6], studying and modeling their dynamics [7], [8], [9], or, more recently, investigating when to push, which push to perform, where [4], [5] and how it can help manipulation in synergy with other skills [3].

Related work and motivation. There has been a large amount of work addressing the learning of pushing policies [3], [9], [4], [5], [10], [11]. In [3], a model-free deep reinforcement learning (RL) method (DQN) was adopted, in which images (color and depth) are mapped to a discretized action space composed of pushing and grasping actions at different locations and orientations. Q-learning [12] is used to compute the expected utilities of actions, relying on positive rewards for pushes leading to further desired goals, like moving and successfully grasping objects [3], or improving the effectiveness of object visual segmentation to increase their graspability [5]. However, these methods suffer from several limitations that we address in this paper.

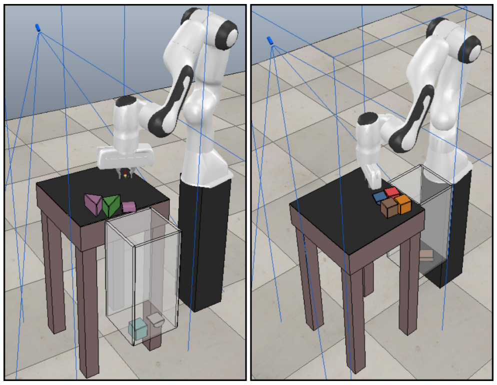

Fig. 1. Learning to push objects into a box having an arbitrary position around the table. We propose a deep-learning approach using as input the depth image and the position of the box (target parameters) and predicting the best push (position and orientation) to move objects towards the box.

First, an objective when learning policies is to make them general and versatile to allow their exploitation in more diverse environments and situations. Such an objective can be achieved by being more robust to changes (in object shapes, illumination, gripper size), but also through the definition of *target parameters* specifying task settings [9], [10]. For instance, in [9], the target is defined by the region where to push and concentrate objects (carrot pieces).

In this paper, we address this problem by investigating the learning of target-parameterized DQN models, focusing on the learning of pushing policies, where the parameters are the position and possible size of the box around the table, as shown in Fig. 1. This relates to the Goal-Conditioned RL problem [13], [14], [15] in which the Q (or Value) function is parametrized by a goal. However, most of the time, the goals and states are strongly correlated (e.g. end position in a navigation task), which differs from our situation where the goal can be more complex (position and size) and characterize the task rather than being a goal in itself.

One issue that arises when using a target-parameterized Deep Neural Network (DNN) approach for manipulation (outside the DQN context) is how to encode these parameters in the DNN. Two main trends can be found in the literature.

In the first one, parameters are introduced after some processing in the *bottleneck* part of the network. The main idea

The authors are with the Idiap Research Institute in Switzerland. Email: firstname.lastname@idiap.ch. The research was funded by the Swiss National Science Foundation through the HEAP project (Human-Guided Learning and Benchmarking of Robotic Heap Sorting, ERA-net CHIST-ERA).

is that the bottleneck captures the essential characteristics of the input data (e.g. visual scene) which can then be combined with task information to produce the desired output. Such an approach can be found in methods that encode actions as task parameters and measure their impact on input data [16]. A similar method is tested in [9] for pushing but underperforms the proposed switch-linear model, as authors observed that their deep model was often stuck in loops, selecting actions that do not cause any visual change. This issue is easily solved with our approach. In [17], an equivariant network is proposed where the state of the robot is inserted into the bottleneck of it. The Cliport method of Shridhar et al. [18] is another recent and impressive work exploiting this approach. Leveraging the pretrained “Clip” DNN model connecting text and vision [19], visual scene components are encoded in the bottleneck and merged with the semantics arising from the textual task description to select the relevant visual information and decide on the task to apply.

One interesting aspect of the bottleneck approach is that it can potentially encode any type of abstract task information. However, when this information has a direct visual counterpart, encoding the target parameters in the visual domain can have several advantages. First, it can be merged early on with scene information, avoiding the potential loss of relevant information (esp. spatial) in the bottleneck compression. Secondly, being visual, it avoids the cumbersome parametric description of shape components and can leverage pretrained deep learning visual processing architectures. Such a visual encoding principle has been successfully exploited with a Transporter Network [20] that relies on a learning from demonstration paradigm and has shown impressive results. This framework was extended for designing Goal-Conditioned Transporter Networks solving rearrangement tasks [21], [22], but these methods relied on Hindsight Experience Replay or visual foresight, as well as human demonstration. None of the above works have investigated visual encoding in the context of RL and target-parameterized DQNs, nor do they investigate target-parameter encoding in the bottleneck as an alternative.

Reinforcement learning issues. DQN methods applied to large action spaces suffer from sparse reward and gradient backpropagation. For instance, in our case, the action space comprises more than 800000 pushing actions (224×224 starting locations and 16 orientations), making learning challenging as each training experience only provides information (a reward) for a single action (the one simulated by the experience). However, when dealing with visual data, it is often the case that similar actions would have similar effects and would result in similar rewards, as illustrated in Fig. 2. In this paper, we exploit this continuity property to propose the use of a kernel loss that compares the Q-value network predictions for different actions with the target value computed from the action that has been performed, hence resulting in better gradient computation in backpropagation.

Finally, in RL, the total reward is often the linear or nonlinear combination of multiple individual rewards representing different facets of the task. For instance, in [3], the same network has to predict (without distinction) the Q-value depending on a single reward signal “triggered” either when pushing actions alter the scene or when grasping is successful. In this paper, we argue that such an approach is losing information related to the reward structure. Thus, following a multi-task paradigm [23], we propose a multi-head architecture, presented in Fig. 3, which distinguishes between the different aspects of the reward and implicitly of the task, allowing to exploit informative losses and to improve the training of the network.

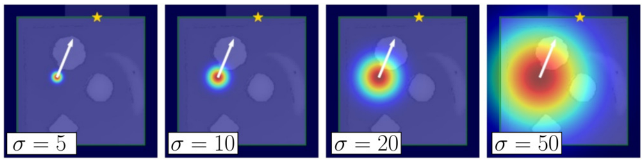

Fig. 2. Kernel loss illustration for a given experience corresponding to the displayed push (white arrow). The kernel loss assumes that pushing at nearby locations and in the same direction would result in similar outcomes. The Gaussian kernel displays the extent of this assumption for different values of the standard deviation σ, with the weight value at a location weighting the loss computed for the push starting in that location.

Contributions. We address the task of pushing objects into a box whose location is only known at test time (see Fig. 1). To address it, we follow the approach of [3], [4] and extend it in several ways, leading to the following contributions:

- We propose a target-parameterized DQN approach to learn push-into-the-box policies that can handle the variability in the location of the target box at test time;
- We investigate the use of a kernelized version of the typical Q-learning loss that makes better use of each experience and leads to less noisy and more informative gradient computation in the backpropagation algorithm;
- We propose a multi-task learning approach, where a single network learns to predict multiple relevant reward aspects by minimizing the associated training losses.

We assess the validity of our design choices through extensive experiments and ablation studies on simulated data and real experiments with a Franka Emika robot, and show that (i) our model can efficiently learn pushing policies with target parameters, (ii) the different target encodings are performing rather similarly, (iii) our kernel approach is beneficial, and (iv) our multi-task RL approach is performing the best compared to alternative architectures.

II. Approach

A. Overview

Fig. 3a shows our overall approach. The core is a multi-head network taking as input the position of the box and a depth image and predicting for each possible pushing action an estimate of its state-action value Q as well as whether it is a valid push (i.e. a push leading to an actual scene change). These two outputs are estimated using two network heads which are further combined using the Hadamard product to refine Q-value pushing predictions and avoid spurious and incorrect actions. In the test phase, the agent applies the push that maximizes the masked Q-value predictions, producing a new environmental state.

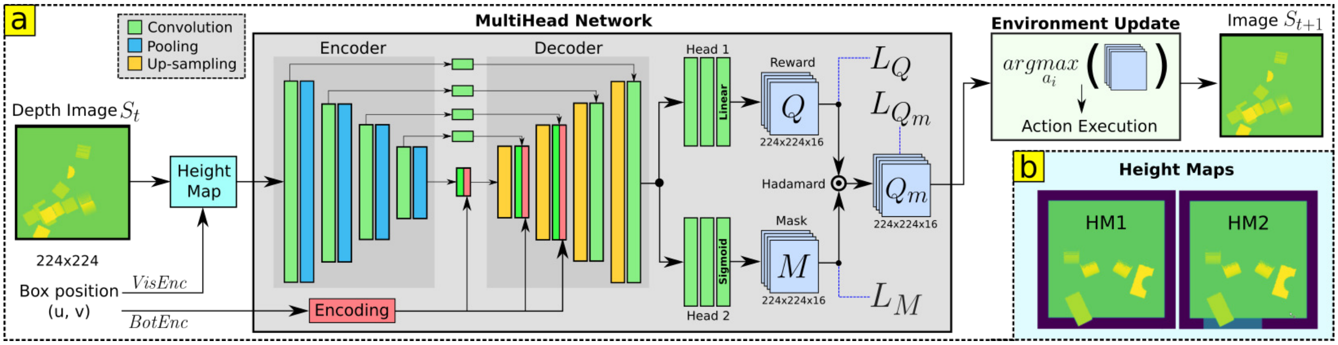

Fig. 3. *(a)* Overall scheme of the proposed multi-head network for pushing objects into a box. The inputs are the target parameters (box position) and a depth image which is processed to produce a heightmap. The network encodes the target parameters either visually in the heightmap (*VisEnc* approach, see point b), or as feature layers incorporated in the network bottleneck (*BotEnc* approach). The push Q-value (or reward for a discount factor $\gamma = 0$), Q, and masks, M, are predicted by two network heads. They are combined to obtain a more accurate prediction of the Q-value, Q_m. In the test phase, the action (push position and orientation) that maximizes the masked Q_m value is executed by the robot, leading to the next state. *(b)* The different depth image representations (heightmaps) tested in our work. In HM1, the table and floor are set to diferent height values, having the floor a lower height than the table. In HM2, the box is visually encoded by adding it at a depth falling in between those of the table and of the floor.

B. *Modeling*

Heightmap. The input depth image acquired by the depth sensor is processed to represent the heightmap of the scene as a 224×224 tensor. We consider two alternatives (see Fig. 3b). The first heightmap (HM1) is obtained by setting two different depth (height) values to the table and floor in the simulator. The second heightmap (HM2) is obtained by adding the box at its planned location p^{box} with an intermediate height (blue area). We have decided to use a heightmap instead of a simple binary occupancy representation of the objects because a heightmap allows for inferring the shape of an object seen from above (e.g. triangle, half cylinder, rectangle). In contrast, a binary occupancy representation may represent all these objects as rectangles depending on their orientations. Moreover, the height of each pixel is used to determine the height of the push and is important to deal with stacked objects, for example.

State-space. The state $s_t = (\mathbf{x}_t, p_t^{\text{box}})$ of the environment at time step t is represented by the heightmap $\mathbf{x}_t$ and the coordinates $p_t^{\text{box}} = (u_t, v_t)$ of the middle point of the side of the box in contact with the table.

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. The action set $\mathcal{A}$ is thus the set of all these possible pushes and is of size $224 \times 224 \times 16$. In our setup, all the pushes in the action set are feasible within the workspace of the robot arm.

Network architecture. We follow the network proposed in [4] based on the Hourglass architecture [24]. However, rather than having two networks working in parallel and trained separately, which has a large computational cost, we propose an efficient multi-head network that is trained to predict the Q-values and the masks.

More specifically, the network comprises an encoder, which creates a feature representation of the depth image in the bottleneck [25], and a decoder. The encoder applies convolutional layers and downsampling operations resulting in a stack of $14 \times 14 \times 64$ feature maps in the bottleneck (prior to the concatenation of target parameter layers, see below). The decoder uses convolutional layers and upsampling operations to return feature maps with the same resolution as the input (224×224). In addition, it integrates information from all resolutions thanks to skip connections coming from the different resolution blocks in the encoder. This allows for combining the different semantic levels while maintaining the localization accuracy of the network prediction. Finally, our network comprises two heads dedicated to the prediction of the Q-value Q and mask M tensors. They consist of convolutional layers connected to the output of the Hourglass network. These head outputs are merged via elementwise multiplication to get the final 'masked' output Q_m.

Target parameters encoding. We study two different ways of encoding the target parameters (box location), $p_t^{\text{box}} = (u_t, v_t)$, into the network. The first one is to encode the parameters in the bottleneck feature layers, using a nonlinear (*BotEnc*) encoding. There u_t and v_t are given as inputs to a fully connected (FC) layer, whose output is reshaped to a $14 \times 14 \times 2$ tensor that further goes through a convolutional layer whose $14 \times 14 \times 8$ tensor output is concatenated with the bottleneck feature maps. Note that following [18], we also concatenate this target tensor to other higher-resolution layers through simple upsampling. When the *BotEnc* parameter encoding is used, the heightmap HM1, which does not visually represent the box, is given to the network as input.

The second target parameterization consists of a visual encoding (*VisEnc*), which is simply achieved by using the HM2 heightmap representation which by construction includes the box location information. With this approach, target information can be combined with the visual scene already in the early network layers.

C. *Rewards*

Our rewards are defined as in [4], and summarized below.

Pushing spot detection reward. This reward refers to the premise that *effective* pushing actions should at least produce *changes*. Hence, given the state $s_t = (\mathbf{x}_t, p_t^{\text{box}})$ and the executed action a_t leading to s_{t+1}, we count the amount of local pixel changes as $c_t = \sum_p I_{|\mathbf{x}_t(p) - \mathbf{x}_{t+1}(p)| \geq 1mm}$, where I_v is the indicator function, equal to 1 if v is true, and 0 otherwise, and $\mathbf{x}_t(p)$ refers to the height at the pixel p. Accordingly, the *PushMask* change reward is given by

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

where τ_{mask} is a threshold that is set to 450 in practice.

Push-into-box reward. This reward incentivizes pushes that move objects towards and into the box, and hence reduce the average distance between object pixels and the box. Accordingly, a measure of pushing effectiveness is:

$$\Delta d_t^M = d_t^M - d_{t+1}^M \text{ with } d_t^M = \frac{1}{|\mathcal{O}_t|} \sum_{p \in \mathcal{O}_t} d_t(p), \tag{2}$$

where $d_t(p) = \|p - p_t^{\text{box}}\|$ denotes the distance between the pixel p and the box center p^{box}, $\mathcal{O}_t$ is the set of pixels on objects, obtained by thresholding the scene depth image, and $|\mathcal{O}_t|$ denotes its cardinality. Accordingly, we define the distance reward as $R_t^d = \max\left(0, \Delta d_t^M\right)$, measuring how much objects move closer to the box. Finally, denoting by N_t^{box} and N_t^{floor} the number of objects falling into the box or on the ground, as provided by the environment simulator, we define the push-into box reward R^p as the sum of R_t^d with the box reward $R_t^{\text{box}} = 10 \times N_t^{\text{box}}$, unless one or more objects fall to the ground ($N_t^{\text{floor}} > 0$). Formally,

$$R^p\left(a_t, s_t, s_{t+1}\right) = \begin{cases} 0 & \text{if } N_t^{\text{floor}} > 0 \\ R_t^d + R_t^{\text{box}} & \text{otherwise} \end{cases}. \tag{3}$$

D. Training protocol

In this section, we indicate the approach and losses used to train our network. We first introduce the Q-learning training procedure, then present our approach relying on multi-task Reinforcement learning (RL). Finally, we formalize our Kernel approach and show how losses can be weighted by a Gaussian kernel to speed up the training and improve the accuracy of the predictions under continuity assumptions.

RL Q-Learning. In simulation, a set of B experiences e_i, which can be expressed as triplets $e_i = (s_t^i, a_t^i, s_{t+1}^i)$, is generated through exploration. Accordingly, the network learns to minimize the discrepancy between the state-action value $Q(s_t^i, a_t^i)$ predicted by the network and the target value:

$$y\left(e_i\right) = R^p\left(e_i\right) + \gamma Q\left(s_{t+1}^i, \arg\max_{a'} Q\left(a', s_{t+1}^i\right)\right), \tag{4}$$

which can be computed from the definition of the reward and the optimal action in the next state ($\arg\max_{a'} Q\left(a', s_{t+1}^i\right)$). In Deep Q-Networks (DQNs) [26], this latter term can be estimated by applying the target network to the state s_{t+1}^i of the current experience. More precisely, the discrepancy can be formalized as the loss:

$$L_Q(e_i) = L_{mse}(Q\left(s_t^i, a_t^i\right), y\left(e_i\right)), \tag{5}$$

where L_{mse} denotes the mean squared error. Such a loss can be computed over mini-batches and used to learn the network weight via SGD algorithms and backpropagation.

Multi-Task RL. In [4], two networks, *PushReward* and *PushMask* were trained independently and their output combined. Each of them had a specific loss function, but there was no loss for the final result, although this is our target. In this paper, we propose to rely on a single network trained with a multi-task loss comprising partial losses for the two heads (the Q-value and the mask) and a loss related to the final prediction (masked Q-values). The motivations are to allow sharing parameters involved in predicting the mask and Q-value, which results in a simpler and more efficient architecture, and provide more supervision (more losses). Accordingly, we define the loss for the Mask head as:

$$L_M(e_i) = L_{bce}\left(M\left(s_t^i, a_t^i\right), R^c\left(a_t^i, s_t^i, s_{t+1}^i\right)\right) \tag{6}$$

where M refers to the outcome of the mask head and L_{bce} is the binary-cross entropy loss. Similarly to Eq. 5, we define a loss for the masked Q-values Q_m (see Fig. 3) as:

$$L_{Q_m}(e_i) = L_{mse}(Q_m\left(s_t^i, a_t^i\right), y\left(e_i\right)). \tag{7}$$

Finally, the total loss is a linear combination of the three different losses

$$L(e_i) = w_1 L_Q(e_i) + w_2 L_{Q_m}(e_i) + w_3 L_M(e_i), \tag{8}$$

where $w_1 = 1$, $w_2 = 1$, and $w_3 = 100$ are the weights associated to the partial losses, which were chosen to make the terms in Eq. 8 lie in the same range.

Kernel loss. In the above, each experience e_i provides only gradient feedback for the action a_t^i actually performed in that experience. This is highly inefficient and leads to high noise during training, even with large batch sizes. As motivated earlier, to improve training, we assume that for a given experience e_i, an action a close to a_t^i would most likely produce a state similar to the next state s_{t+1}^i, and hence would produce a similar target value $y\left(e_i\right)$. Accordingly, we can compare the network prediction for that action a with $y\left(e_i\right)$, and define the following action loss:

$$l_K(e_i, a) = L_{mse}(Q\left(s_t^i, a\right), y\left(e_i\right)). \tag{9}$$

The losses associated to all actions can then be combined to define the loss associated to a given experience as follows:

$$L_K(e_i) = \frac{\sum_a w(a, e_i) l_K(e_i, a)}{\sum_a w(a, e_i)}, \tag{10}$$

where $w(a, e_i)$ is a weighting factor giving more importance to actions close to a_t^i. In practice, we use a Gaussian kernel centered at the experienced action start position $p_{a_t^i}$, i.e., w is 0 for actions a whose pushing direction is different from the one of a_t^i, and equal to $\mathcal{N}(p_a; p_{a_t^i}, \Sigma)$ otherwise.

III. Experiments

A. Network Training

To conduct experiments, we relied on an offline RL approach [27] in which we first collected a dataset D_{off} of experiences and used this fixed dataset to learn the different models. The dataset has been collected using CoppeliaSim [28]. The objects used in our simulations are the same as in [3]. Although suboptimal, as applying online RL and collecting data according to the learned policy would probably result in better performance, we preferred this option as it allowed to have better control of the learning conditions and to achieve a fairer comparison of the different alternatives. Hence, for the experiments we have used a discount factor γ

TABLE I

DATASET D_{OFF}

Total number of pushes	82458
- leading to changes	51189 (62.1%)
- where at least one object fell to the ground	8232 (10.0%)
- where at least one object fell into the box	969 (1.2%)

equal to zero. In the future, we will address other tasks and investigate further the choice of the discount factor.

Experience dataset. To obtain our dataset D_{off}, pushes have been sequentially performed in scenarios generated by tossing five randomly selected objects onto the table. Approximately 50% of these pushes were sampled uniformly at random from the set of all possible pushes, while the other 50% pushes were sampled to likely produce changes based on Canny edge detection heuristics [29]. By sampling pushes in these two manners, we aimed at obtaining a more informative dataset than through pure random sampling. Table I summarizes the resulting statistics. Note in particular that the number of actual objects falling in the box is very low, making the task challenging.

Training protocol. We train our networks from scratch using the dataset D_{off}. Our models have been trained for 30 epochs using Adam optimizer and early-stopping criterion to avoid overfitting. The training data comprises 85% of D_{off}, while 15% has been used for validation. Models minimizing the best validation loss have been used for the subsequent evaluations. If not stated otherwise, the Gaussian kernel loss for training has a standard deviation $\sigma = 5$.

B. Experimental protocol

Evaluation procedure. We created a dataset of 300 random scenarios with exactly five objects each. For each model under evaluation, we applied the corresponding policy until either there was no more objects on the table, or the policy did not produce any change five times in a row.

Evaluation metrics. The performance is measured by the average numbers of (i) objects successfully pushed into the box (N_{ObjB}) (ii) objects that fell on the ground (N_{ObjG}) (iii) objects left on the table (N_{ObjT}) and (iv) pushes (N_{Act}) needed to process each scenario, providing some measure of the efficiency of pushes.

Tested models. To evaluate the benefits of our framework and compare to the state-of-the-art, we considered the models below. Note that the same HourGlass-based architecture as in Fig. 3 is used (to the exception of the stated modeling aspects) to allow a fair comparison.

- SHead. This is the single head approach in DQNs, in which a single network is used to predict the Q-value [3]. We use the same architecture as in Fig. 3, keeping only the Head 1 and the associated reward loss L_Q. This network represents the state-of-the-art [3]. Note that in [3] a DenseNet backbone was used, but was shown to underperform w.r.t. our HourGlass-based architecture [4].
- TNets. It corresponds to the approach proposed in [4] (but generalized to allow for target parameters), in which two networks are trained separately: one to predict the push reward, the other to predict changes. The outputs of both networks are elementwise multiplied at test time to produce masked push reward predictions. In practice, the architecture of Fig. 3 is repeated twice, using the relevant prediction head and training losses for each network (so L_Q and L_M).,
- MHead. It corresponds to our multi-head network trained by minimizing the loss defined in Eq. 8;

TABLE II

COMPARISON OF NETWORKS AND TARGET-PARAMETERS ENCODINGS. OUR PROPOSED NETWORK IS IN LIGHTGRAY.

Model	Encoding	**N_{ObjB} (%)**	**N_{ObjG} (%)**	**N_{ObjT} (%)**	**N_{Act}**
SHead [3]	*BotEnc*	4.05 (81.0)	0.62 (12.4)	0.34 (6.8)	11.7
SHead [3]	*VisEnc*	4.21 (84.2)	0.58 (11.6)	0.21 (4.2)	10.3
TNets [4]	*BotEnc*	4.30 (86.0)	0.67 (13.4)	0.03 (0.0)	11.1
TNets [4]	*VisEnc*	4.39 (87.8)	0.53 (10.6)	0.09 (0.2)	9.7
MHead	*BotEnc*	4.47 (89.3)	0.52 (10.5)	0.01 (0.2)	10.1
MHead	*VisEnc*	**4.50 (90.1)**	0.47 (9.3)	0.03 (0.6)	16.8
MHead	Combined	4.47 (89.4)	0.53 (10.6)	0.00 (0.0)	9.8

C. Results

Network architecture. Table II provides our results. We can first notice that whatever the target encoding, SHead [3] performs worse than both TNets and MHead, which both predict a mask to filter out spurious predictions provided by the reward head through the Hadamard product. This aspect can be observed by the fact that the number of objects left on the table N_{ObjT} in SHead is reduced significantly, and is illustrated in Fig. 4 where the robot with SHead selects a push on the table border with no object.

Comparing TNets and MHead, the latter performs better and obtains the best performance with *VisEnc* (N_{ObjB} = 4.50 (90.1%)). It shows that training a single network for multiple tasks is beneficial, which is particularly interesting since MHead is also significantly (twice) more efficient computationally as it uses one network instead of two.

Target-parameter encoding. First, we can notice that although visual encoding *VisEnc* produces the best results, the bottleneck encoding *BotEnc* results are competitive. It shows that even in the absence of a visual representation of the box position, the network can learn relevant features to succeed in the pushing task. Such an encoding can be desirable when it is cumbersome to visually encode all relevant features for the task at hand; for instance, to push objects close to a certain point irrespective of any occlusions of the target. The bottleneck encoding *BotEnc* could in principle also be used to tackle different tasks. For example, one could consider giving additional parameter values as input to the *BotEnc* module in order to determine if the robot should push only objects of a certain color (if RGBD images were given as input to the network) or shape into the box or sort different objects into different boxes.

Secondly, the inspection of the results and pushing simulations shows that the two encodings produce two different policy behaviors, as illustrated in Fig. 6. With the *BotEnc* encoding, the policy is more direct: it tends to group objects and then push them jointly towards the box, as this way of doing maximizes the reward at each step, based on the scene configuration. Although this behavior seems correct,

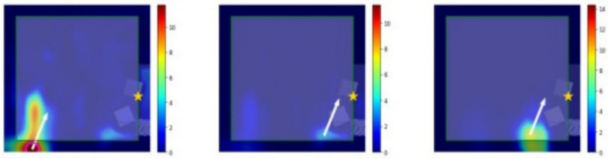

Fig. 4.Left: best push (white arrow) selected by the SHead network (using visual encoding), shown on the heatmap of pushes with high probability for the selected direction. Middle and right: heatmaps for the same direction as on the left (with arrows corresponding to the max for these heatmaps) for the TNets and MHead, showing that having a mask allows to filter out spurious predictions as the one shown in SHead.

TABLE III

IMPACT OF THE KERNEL SIZE AND LOSSES DURING TRAINING. THE DEFAULT CONFIGURATION ($\sigma = 5$, ALL LOSSES IN EQ. 8, I.E. L_{Q_m}, L_Q, AND L_M) IS SHOWN IN LIGHT GRAY.

Kernel	Encoding	$N_{\mathbf{ObjB}}$ (%)	$N_{\mathbf{ObjG}}$ (%)	$N_{\mathbf{ObjT}}$ (%)	$N_{\mathbf{Act}}$
None	*BotEnc*	4.26 (85.2)	0.74 (14.8)	0.00 (0.0)	9.7
None	*VisEnc*	4.41 (88.1)	0.59 (11.8)	0.01 (0.1)	14.8
$\sigma = 5$	*BotEnc*	4.47 (89.3)	0.52 (10.5)	0.01 (0.2)	10.1
$\sigma = 5$	*VisEnc*	4.50 (90.1)	0.47 (9.3)	0.03 (0.6)	16.8
$\sigma = 10$	*BotEnc*	4.45 (89.0)	0.54 (10.9)	0.01 (0.1)	10.2
$\sigma = 10$	*VisEnc*	**4.63 (92.7)**	0.32 (6.5)	0.04 (0.9)	18.8
$\sigma = 20$	*BotEnc*	3.93 (78.6)	0.50 (10.1)	0.56 (11.3)	13.9
$\sigma = 20$	*VisEnc*	4.01 (80.2)	0.43 (8.7)	0.56 (11.1)	21.7
Losses					
L_{Q_m}	*BotEnc*	4.32 (86.4)	0.67 (13.5)	0.01 (0.1)	9.4
L_{Q_m}	*VisEnc*	4.39 (87.8)	0.55 (11.0)	0.06 (1.1)	17.5
L_Q, L_M	*BotEnc*	4.37 (87.3)	0.63 (12.7)	0.00 (0.0)	9.9
L_Q, L_M	*VisEnc*	4.45 (89.0)	0.47 (9.33)	0.08 (1.6)	16.5

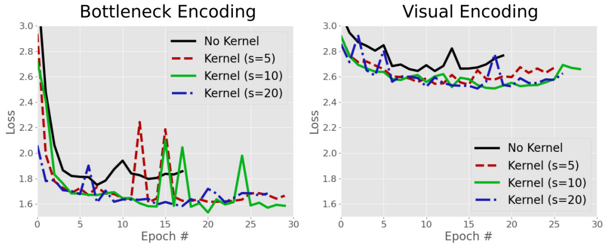

Fig. 5. Validation losses during the network training for the visual *VisEnc* and bottleneck *BotEnc* encodings and for different Gaussian kernel sizes.

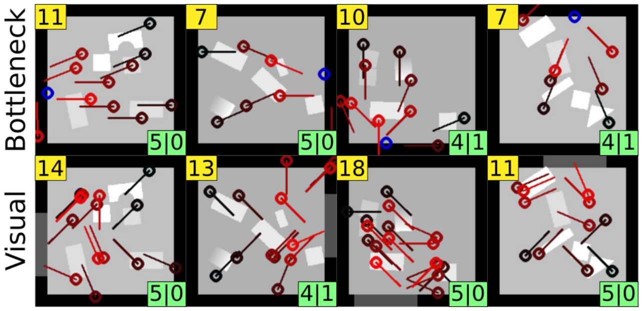

Fig. 6. Pushes done by MHead using *BotEnc* (Top) and *VisEnc* (bottom) for 4 testing scenarios. The numbers of pushes to complete the task are shown in yellow, while the number of objects pushed in the box or on the floor are shown in green. In the *BotEnc* case, the box position is indicated with a blue circle. Black arrows refers to pushes made at the beginning of the experiment, and red ones to those done at the end. The image behind corresponds to the initial layout of objects (first frame).

it is prone to pushing objects on the ground, resulting in a larger N_{ObjG}. On the other hand, with *VisEnc* the network adopts a more conservative behavior. While it also starts to pushes the objects farthest away from the box, it also tends to relocate them more often towards the table center or in front of the box until it is more confident to push them safely into the box. This is confirmed by Table II and Table III which show that *VisEnc* requires more actions to process a scene. This might be due to the visual representation of the box which includes both position and width, and may influence the learning, while with *BotEnc*, only the box position is specified. Further investigations and analysis are needed to corroborate these observations.

Kernel. Table III shows the performance of our MHead network in function of the kernel size used during training. We see that larger kernels lead to better performance (especially with the *BotEnc* encoding) until $\sigma = 20$, which is too large and breaks the underlying continuity assumption of the approach. The impact of the kernel size can be further assessed by looking at the validation loss[1] as shown in Fig. 5. We see that a much faster convergence can be achieved in the first epochs as more gradients are backpropagated, and that smaller validation loss can be obtained in the long run for a given network. Note that the validation loss is larger without the kernel, and shows faster overfitting (training is interrupted by the early-stopping criteria).

Finally, it is interesting to see that although the *VisEnc* encoding provides better task performance, it leads to worse validation losses than with *BotEnc*. This highlights the slight discrepancy between the reward and the actual task, esp. as offline RL training is used.

Loss definition. Table III further shows the results of the MHead network trained with only a subset of the losses. We see that using only the loss L_{Q_m} on the target Q-value (the masked Q-values) as done traditionally, or using only the losses on the two heads (L_Q, L_M), as implicitly done in [4], provides worse results than using all the three losses defined in Eq. 8, showing that more supervision is beneficial.

Real robot experiments. We tested on a real robot a policy learned in simulation (see Fig. 7). In contrast to the simulated setup, we deployed an eye-in-hand configuration of the vision system. To execute a push with the robot, we use a task-space trajectory tracking controller with inverse dynamics in the configuration space [30]. The pushing motion is designed to be similar to the behavior in the simulation while being dynamically feasible for the real robot.

We tested the performance of the learned policy in 6 different scenarios with 3 different box positions. We used 4 YCB objects [31] with shapes unseen during training and arbitrarily scattered on the tabletop in each scenario. In 4 scenarios, the robot managed to clean the table properly (all objects fell into the box). In the 2 others, the robot pushed 3 objects into the box, while the last one remained on the tabletop. In those cases, the robot got stuck in repeating a pushing action that did not move the remaining object, which could be caused by a mismatch between the objects used in the training and the real ones, or by the limitations of the dataset used for training, which did not have so many cases of objects falling into the box or to the ground.

[1] Note that to the contrary of the training loss, the validation loss is only evaluated at the performed action, so losses are comparable across models.

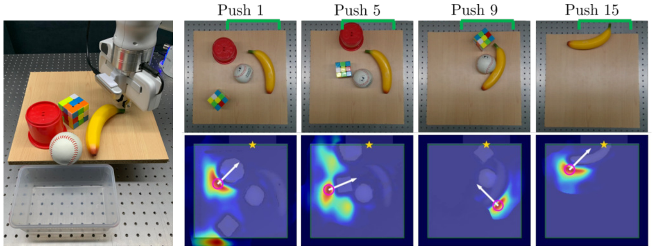

Fig. 7. Experimental setup with the Franka Emika robot arm. The objective of the robot is to push all objects on the tabletop into a box located at arbitrary positions around the table.

IV. Conclusion and Future Work

We have proposed an efficient target-parameterized multi-head network architecture for pushing objects into a box whose location can change at test time. We have shown that both a visual encoding as well as bottleneck layer encoding of the target parameters (box location) can lead to good performance, although the learned policies exhibit different behaviors. In addition, a kernelized version of the conventional Q-learning loss was proposed. It led to better, faster and more stable training thanks to a better use of each experience, resulting in less noisy and more informative gradient computation in the backpropagation algorithm. This approach is generic and can be applied for other robotics manipulation tasks. In the future, we want to explore tasks with sparser rewards and more challenging tasks such as sorting objects according to their shape or color. There it will be important to explore discount factors greater than zero to achieve policies that can consider future rewards when choosing their actions. In theses tasks, we expect that the faster learning of the kernel approach could be beneficial.

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," in *Conference on Robot Learning (CoRL)*, 2020, pp. 1094–1100.

[2] A. Eitel, N. Hauff, and W. Burgard, "Learning to singulate objects using a push proposal network," in *Robotics research*. Springer, 2020, pp. 405–419.

[3] 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 *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, 2018, pp. 4238–4245.

[4] M. Ewerton, A. Martínez-González, and J.-M. Odobez, "An efficient image-to-image translation hourglass-based architecture for object pushing policy learning," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2021, pp. 3478–3484.

[5] B. Serhan, H. Pandya, A. Kucukyilmaz, and G. Neumann, "Push-to-see: Learning non-prehensile manipulation to enhance instance segmentation via deep q-learning," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2022.

[6] 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 *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2016, pp. 30–37.

[7] 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.

[8] 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 *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2018, pp. 3066–3073.

[9] H. Suh and R. Tedrake, "The surprising effectiveness of linear models for visual foresight in object pile manipulation," in *Workshop on Algorithmic Foundations of Robotics (WAFR)*, 2020.

[10] J. K. Li, W. S. Lee, and D. Hsu, "Push-net: Deep planar pushing for objects with unknown physical properties." in *Robotics: Science and Systems*, vol. 14, 2018, pp. 1–9.

[11] N. Dengler, D. Großklaus, and M. Bennewitz, "Learning goal-oriented non-prehensile pushing in cluttered scenes," in *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2022, pp. 1116–1122.

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

[13] L. P. Kaelbling, "Learning to achieve goals," in *IJCAI*, vol. 2, 1993, pp. 1094–8.

[14] E. Chane-Sane, C. Schmid, and I. Laptev, "Goal-conditioned reinforcement learning with imagined subgoals," in *International Conference on Machine Learning*. PMLR, 2021, pp. 1430–1440.

[15] M. Liu, M. Zhu, and W. Zhang, "Goal-conditioned reinforcement learning: Problems and solutions," *preprint, arXiv:2201.08299*, 2022.

[16] C. Finn and S. Levine, "Deep visual foresight for planning robot motion," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2017, pp. 2786–2793.

[17] D. Wang, R. Walters, X. Zhu, and R. Platt, "Equivariant q learning in spatial action spaces," in *Conference on Robot Learning*. PMLR, 2022, pp. 1713–1723.

[18] M. Shridhar, L. Manuelli, and D. Fox, "Cliport: What and where pathways for robotic manipulation," in *Conference on Robot Learning (CoRL)*, 2021.

[19] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, *et al.*, "Learning transferable visual models from natural language supervision," in *International Conference on Machine Learning (ICML)*, 2021, pp. 8748–8763.

[20] A. Zeng, P. Florence, J. Tompson, S. Welker, J. Chien, M. Attarian, T. Armstrong, I. Krasin, D. Duong, V. Sindhwani, and J. Lee, "Transporter networks: Rearranging the visual world for robotic manipulation," in *Conference on Robot Learning (CoRL)*, 2020.

[21] D. Seita, P. Florence, J. Tompson, E. Coumans, V. Sindhwani, K. Goldberg, and A. Zeng, "Learning to rearrange deformable cables, fabrics, and bags with goal-conditioned transporter networks," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2021, pp. 4568–4575.

[22] H. Wu, J. Ye, X. Meng, C. Paxton, and G. S. Chirikjian, "Transporters with visual foresight for solving unseen rearrangement tasks," in *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2022, pp. 10 756–10 763.

[23] Y. Zhang and Q. Yang, "An overview of multi-task learning," *National Science Review*, vol. 5, no. 1, pp. 30–43, 2018.

[24] A. Newell, K. Yang, and J. Deng, "Stacked hourglass networks for human pose estimation," in *European Conference on Computer Vision (ECCV)*. Springer, 2016, pp. 483–499.

[25] 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.

[26] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[27] S. Levine, A. Kumar, G. Tucker, and J. Fu, "Offline reinforcement learning: Tutorial, review," *and Perspectives on Open Problems*, vol. 5, 2020.

[28] E. Rohmer, S. P. N. Singh, and M. Freese, "V-rep: A versatile and scalable robot simulation framework," in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2013, pp. 1321–1326.

[29] M. Ewerton, S. Calinon, and J.-M. Odobez, "An attention mechanism for deep q-networks with applications in robotic pushing," in *Workshop on Emerging paradigms for robotic manipulation: from the lab to the productive world, ICRA*, 2021.

[30] J. Jankowski, M. Racca, and S. Calinon, "From key positions to optimal basis functions for probabilistic adaptive control," *IEEE Robotics and Automation Letters (RA-L)*, vol. 7, no. 2, pp. 3242–3249, 2022.

[31] B. Calli, A. Singh, J. Bruce, A. Walsman, K. Konolige, S. Srinivasa, P. Abbeel, and A. M. Dollar, "Yale-cmu-berkeley dataset for robotic manipulation research," *The International Journal of Robotics Research*, vol. 36, no. 3, pp. 261–268, 2017.