Topic F - Visually-Guided Robotic Manipulation

Matias Etcheverry
Ecole Normale Supérieure Paris-Saclay
matias.etcheverry9@gmail.com

February 15, 2023

Abstract

Picking and placing objects in unstructured setting is a key issue in robotic manipulation. Classic systems use pose estimation of known objects as well as scripted planning. In this work, we propose to review a novel type of algorithm proposed by Zeng et al. [7]: transporter network. We will test this model on 2 simulated tasks, thanks to the Ravens framework. Moreover, we propose an ablation study consisting of removing depth information used by the network. We show that transporter networks reach state of the art results, while maintaining an excellent sample efficients but still requires exact depth features.

1. Introduction

Machine learning has lead to the emergence of multiple end-to-end models able to perform complex manipulation of real world objects. The objective is to compute control policies from one or more camera pointing towards the workspace. *Form2Fit* proposed by Zakka et al. [6] is able to generate policies, using a network matching the source and the destination of an object. Zeng et al. [7] proposed a simple neural architecture, the transporter network, which extracts simple visual features and is able to perform a wide variety of pick and place tasks with a few training demonstrations. In this work, we will start by an introduction on transporter networks and experimenting on 2 simulated tasks. We then propose an ablation study consisting in deleting the depth features given to the network. Our work is available on GitHub[1].

2. Transporter networks

Transporter networks introduced by [7] are capable of achieving state-of-the-art success rate with objects in new configurations, using as few as 10 expert demonstrations. In this work, transporter networks are used in a simulated environment.

2.1. Problem

We consider the pick-and-place problem, where a robot picks an object from the location $\mathcal{T}_{\text{pick}}$ and places it at $\mathcal{T}_{\text{place}}$. These locations are 2D coordinates[2]. Let o be the observation of the environment. Typically, this observation is made of RGB and depths images from multiple views directed towards the workspace. We want the robot to learn 2 functions:

$$f_{\text{pick}}(o) = \mathcal{T}_{\text{pick}} \qquad f_{\text{place}}(o, \mathcal{T}_{\text{pick}}) = \mathcal{T}_{\text{place}} \tag{1}$$

The placing action highly depends on the picking action. Intuitively, picking an object by the side won't result in the same place action as picking it by the top.

Orthographic projection Preprocessing the observation of the environment into spatially consistent features allows the model to learn faster. This setting also allows for powerful data augmentation techniques. Thus, the authors proposed an orthographic projection of the workspace as a preprocessing step, applied on the observations of the environment. This projection is then fed to the transporter network, as it contains color and depth information.

Learning picking A 43-layer Resnet [3] is used in order to select the picking action with the highest sucess rate. The output of the picking model is a dense pixel-wise map which highlights the picking success. While a dataset made of 10 expert demonstrations may exhibit a significant inductive bias, this picking network exploits translationally equivariance property of fully convolutional networks to generalize against unseen configurations.

Learning pick-conditioned placing Placing is learned through transporting: a partial crop $o[\mathcal{T}_{\text{pick}}]$ centered on $\mathcal{T}_{\text{pick}}$ is overlapped on another partial crop $o[\tau]$ centered on a candidate place pose τ.

More precisely, the pick-conditioned placing model is a fully convolutional networks which outputs two dense feature maps: the query features $\psi(o)$ and the key features $\phi(o)$. The backbone of this model is also a 43-layer Resnet. The cross-correlation between the partial query crop $\psi(o[\mathcal{T}_{\text{pick}}])$ and the key features of each candidate place pose is computed via $\psi(o[\mathcal{T}_{\text{pick}}]) * \phi(o)[\tau]$. $\mathcal{T}_{\text{place}}$ is the candidate place pose having the highest cross-correlation with the pick crop:

$$\mathcal{T}_{\text{place}} = \arg\max_{\tau} \psi(o[\mathcal{T}_{\text{pick}}]) * \phi(o)[\tau] \tag{2}$$

[1] https://github.com/MatiasEtcheve/RECVIS-transporter-networks

[2] Further work proposed by the authors extends to 3D locations.

Fig. 1 shows how to learn the placing in 4 steps, from RGB and depth images. A top-down view is reconstructed beforehand and then feed to the placing network, which outputs 2 heatmaps. These heatmaps are then cropped and overlapped to find the best candidate place pose.

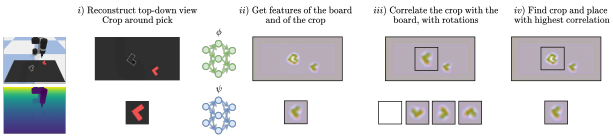


Figure 1. Learning the pick-conditioned placing within 4 steps

Training In order to train the model, we first create a dataset of n expert demonstrations where each demonstration is made of a list of (observation, pick action, place action) tuple, which depicts the best actions to perform, given a certain observation of the environment. A robotic task may be expressed as a sequence of pick-and-place actions. Thus, we feed our model random observation-action pairs at any time in the sequences. While solving Eq. (1) leads to stateless solutions, experiments lead to models learning sequences. Finally, the training loss is simply the cross-entropy between the labelled pick and place actions and the predictions.

2.2. Experiments

We execute experiments in simulated settings to evaluate the performances of the transporter networks.

Simulation Ravens [5] is the framework used to train the transporter networks. This framework has 10 built-in tasks, which are all represented as Markovian: the best action to perform at a given timestep only depends on the current state. 3 RGB-Depth cameras directed to the workspace, where the robot manipulates objects, are used as inputs to our model. The framework also proposes a list of observation-action-reward tuples as labels. In order to create the datasets, built-in scripted oracles provide expert demonstrations per task by randomly sampling the distribution of successful picking and placing actions. While this framework was especially designed to evaluate the transporter networks, it can be used with active learning, reinforcement learning and active perception.

For now, we will focus on 2 different tasks: *block-insertion* and *manipulating-rope*. The former consists in moving a red L-shaped object into a gray fixture. The *manipulating-rope* task consists in moving a rope so that if fits an incomplete perimeter of a square. Fig. 2 shows the inputs and the labels on the *block-insertion* task.

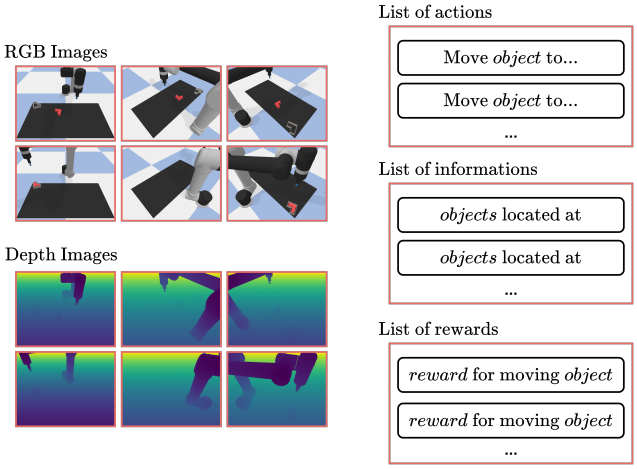


Figure 2. Inputs (left) and labels (rights) for the *block-insertion* task.

Evaluation metrics In order to evaluate the performances of the model, we use 2 different metrics. Intuitively, the binary success rate should be reported. However, this metric only relates if the robot succeeded without telling how good it was if it failed. Thus, we instead store the total reward after the last performed action. This reward may differ depending on the task being performed:

1. for pose tasks, like *block-insertion*, the reward is binary. it equals 1 if object place is less than 1 cm and 15° away from target pose, and 0 otherwise. Thus, the reward is exactly the same as the binary success rate for the *block-insertion* task.

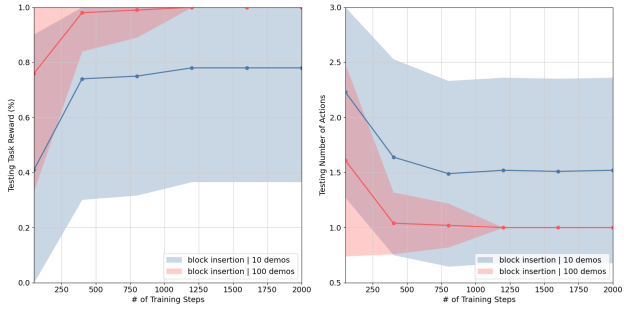
2. for zone tasks, like *manipulating-rope*, the space is discretized in 2 cm^3 voxels. The total reward is the fraction of total voxels in the target zone. For such tasks, it is exponentially more difficult to have a reward $r = 0.9$ than a reward $r = 0.8$.

Finally, we also report the number of actions required to succeed a task. All the performance metrics are run on a test dataset made of 100 unknown situations, per task.

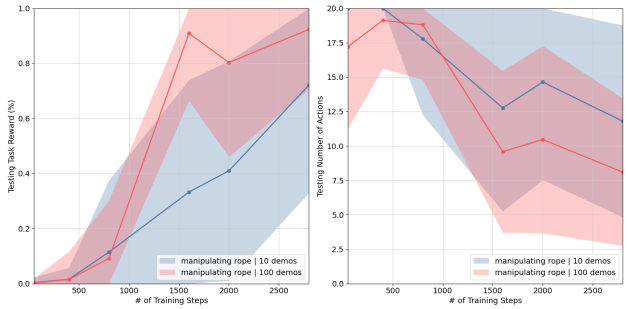
Results While the authors trained for 10 000 iterations, we limit ourselves to 2 000 iterations, because of resource limitations and already satisfying results. The authors also trained with up to 1 000 training expert demonstrations, and we will again limit us to 100 demonstrations for the same reasons.

Fig. 3 shows the curves of the evaluation metrics during the training phase. For the *block-insertion* task, the model is learning very fast, and reaches its final result within 750 iterations. For the *manipulating-rope* task, training for 2 800 iterations is not sufficient, but already leads to good results. The average reward for both tasks is greater when training

with more demonstrations. We also notice that the number of actions required to succeed a task is decreasing with respect to the training steps, which shows that the model learns optimal actions in each state. Fig. 4 shows the place position and the place rotation on the *block-insertion* task. It highlights the capibility of the transporter network to generalize with only 10 training examples, on unseen configurations.



(a) *Block-insertion*



(b) *Manipulating-rope*

Figure 3. Performance metrics on the 2 simulated tasks.

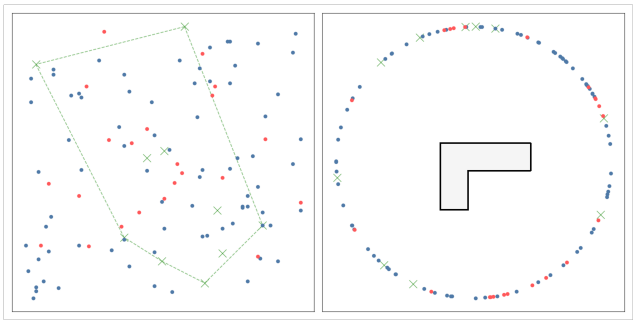


Figure 4. Place positions (left) and place rotations (right). Training examples (green crosses) and their convex hull, failed test examples (red points) and succeeded test examples (blue points). This plot highlights the generalization capability of the transporter network, which can performs greatly even outside of the convex hull of its training demonstrations.

Tab. 1 shows the average rewards on the test dataset. After training, our model reached roughly the same performances as stated by the authors. It is noticeable that the model achieves very good performance when using as few as 10 training examples. However, we observe a gap when training with 10 demonstrations on the *block-insertion* task: our model performs 20% worse compared to what the authors reported. This decrease is explained by the number of training demonstrations: ultimately, the model will be highly sensitive to the training examples, when training on a very few of them.

	Block-insertion		*Manipulating-rope*	
Training Demonstrations	10	100	10	100
Ours	78.0%	100%	72.2%	92.3%
Proposed	100%	100%	73.2%	85.4%

Table 1. Average reward on test dataset

Finally, it is interesting to inspect the behavior of the robot in simulated situations. Our GitHub page contains a few videos of the predicted policies.

3. Depth ablation

The transporter networks are using RGB-Depth images. An interesting ablation of these models is to delete the depth information, to see how the learning goes. This ablation study can be done through 2 methods: *soft* and *hard* deletion.

3.1. Soft deletion

Originally, the transporter networks preprocess the RGB-Depth images into a top down projection of the workspace. This projection still contains both RGB and depth information, before being fed to the network. The soft deletion consists in deleting the depth information from this orthographic view, as shown by Fig. 5. This setting corresponds to the case where a RGB camera is perfectly set up above the workspace, without being occluded by the robot it self. This setting may not be accessible in real life problems, but allows to run the model with RGB only images.

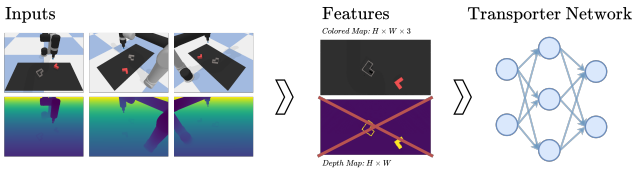


Figure 5. Soft deletion pipeline

3.2. Hard deletion

Thanks to Ravens, the transporter networks have the possibility to use exact depth images, from 3 different views.

However, in real life situations, the depth images are difficult to obtain as they require the use of specific cameras. They can lead to noisy results, which make difficult to top down reconstruction. Instead of simply deleting the depth information, we may want to reconstruct it. The hard deletion consists in estimating the depth images from the RGB images, as shown on image Fig. 6. In this case, we are analyzing the robustness of the transporter networks against noisy and possibly inconsistent depth images.

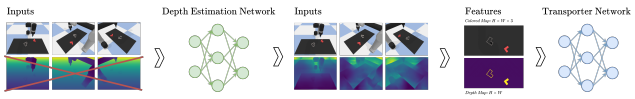


Figure 6. Hard deletion pipeline

Depth estimation model AdaBins [2] is a state-of-the-art algorithm to estimate depth from single RGB images[3]. It uses the best of depth classification and regression. Fig. 7 shows the 4 steps of AdaBins:

1. **Extract *decoded features***. It uses an encoder-decoder block inspired from [1]. This block outputs local features of the input image.

2. **Extract the bins**. The *decoded features* are fed to a transformer encoder. Its attention layers allow to learn global image's information, and ouputs an embedding vector which is then transformed into a N-dimensional vector. This vector corresponds to the size of N bins, which correspond to the labels of the depth classification. Intuitively, the depth value of a pixel is assigned to a specific bin.

3. **Compute range attention maps**. The output embedding vector of the transformer encoder is multiplied with the *decoded features*. Thus, the network integrates adaptive global information from the transformer into the local information of the *decoded features*. Values in the range attention maps are probabilities of a pixel of belonging to each depth bin.

4. **Hybrid regression**. Range attention maps are multiplied with the bins. This regression outputs a depth value for each pixel in the image.

Evaluation metrics AdaBins is run on the simulted RGB images obtained from Ravens. To evaluate the performances, we uses the relative absolute error (REL) and the

[3]I mentionned using BinsFormer [4] in my proposal. However, the official repository of AdaBins was much easier to manipulate and lead to better results.

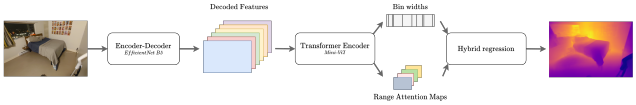


Figure 7. AdaBins algorithm

Root Mean Squared Error (RMSE). We also use the accuracy at threshold δ_1 defined as the percent of predictions verifying $\max(\frac{y}{\hat{y}}, \frac{\hat{y}}{y}) < 1.25$ with y and $\hat{y}$ being respectively the target and the predictions of AdaBins.

Results AdaBins was previously trained on the NYU dataset, which is composed of indoor images, whose depths range from 1cm to 10m. Fig. 8 shows an example of AdaBins applied on a simulated images while Tab. 2 shows the results obtained on NYU and Ravens simulated images. the AdaBins does not predict outliers but neither does it accurately predict depths. The high differences of performances between the NYU and Ravens images is due to their intrinsec design: AdaBins was trained only with real world images.

Figure 8. Color image (right), ground-truth depth image (middle) and estimated depth image (left)

	REL ↓	RMSE ↓	δ_1 ↑
NYU	10.3%	0.364	90.3%
Ravens	35.7%	0.357	64.7%

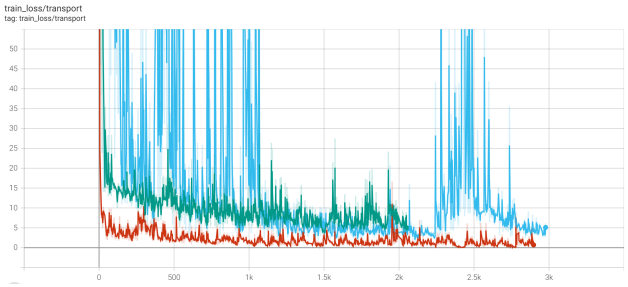
Table 2. Performances on NYU and Ravens

3.3. Experiments

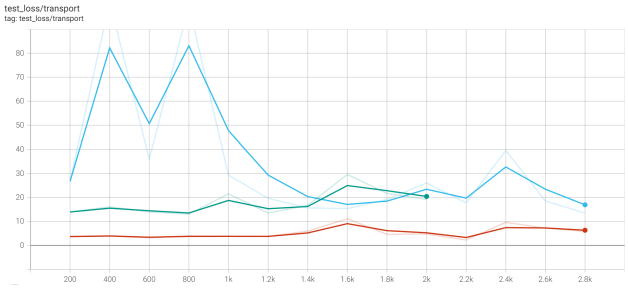
We perform soft and hard deletion on the pipeline. We keep the same evaluation metrics as in Sec. 2.2. A distinction is made on the type of the inputs: RGB-Depth images for the original transporter networks, RGB images for the soft deletion models and RGB-Estimated Depth images for the hard deletion models.

Fig. 9 shows the training and test losses, for the *block-insertion* task. Although the model learns efficiently whatever its input types, it fails to generalize on the test dataset.

Tab. 3 shows the average rewards on the test dataset, depending on the inputs fed to the model. We notice that both deletion methods lead to poor results. On the one hand, soft deletion exhibits the need of depth information for the



(a) Train loss on *Block-insertion*



(b) Test loss on *block-insertion*

Figure 9. Losses observed during the training phase. RGB-Depth images as input are in red, RGB images are in green and RGB-Estimated Depth images are in blue.

model to perform well. On the other hand, the poor success on hard deletion reveals the importance of exact depth information. Indeed, with noisy and inconsistent depth values between camera views, the reconstruction of the orthographic view of the workspace leads to sparse results, where the model can't learn anything from it.

RGB-Depth Images	RGB Images (soft deletion)	RGB-Estimated Depth Images (hard deletion)
100%	2%	3%

Table 3. Rewards obtained on the test dataset with different inputs

4. Conclusion & Future perspective

In this work, we started by training and running a transporter network on 2 simulated tasks. The excellent results led to ablation studies where we delete the depth information. We understand that this information is essential and needs to be accurate for the model to train. A future work consists in finetuning AdaBins to obtain better results on simulated images. Indeed, our GPU was not powerful enough to enable finetuning the model, within reasonable time. Special attention should be paid to this finetuning, because as the simulated images from ravens are all similar, it is very likely that the resulting AdaBins model will overfit. Moreover, we could rearrange Adabins' architecture so that it directly infer depth features from the 3 camera views instead of considering them independently.

References

[1] Ibraheem Alhashim and Peter Wonka. High quality monocular depth estimation via transfer learning, 2018. 4

[2] Shariq Farooq Bhat, Ibraheem Alhashim, and Peter Wonka. AdaBins: Depth estimation using adaptive bins. In *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, jun 2021. 4

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015. 1

[4] Zhenyu Li, Xuyang Wang, Xianming Liu, and Junjun Jiang. Binsformer: Revisiting adaptive bins for monocular depth estimation, 2022. 4

[5] Google Research. Ravens - transporter networks. https://github.com/google-research/ravens, 2020. 2

[6] Kevin Zakka, Andy Zeng, Johnny Lee, and Shuran Song. Form2fit: Learning shape priors for generalizable assembly from disassembly, 2019. 1

[7] Andy Zeng, Pete Florence, Jonathan Tompson, Stefan Welker, Jonathan Chien, Maria Attarian, Travis Armstrong, Ivan Krasin, Dan Duong, Ayzaan Wahid, Vikas Sindhwani, and Johnny Lee. Transporter networks: Rearranging the visual world for robotic manipulation, 2020. 1