Homework 4

Due: April 28, 2022

Introduction

When you design learning algorithms for robots, how to represent a robot's observation input and action outputs often play a decisive role in the algorithm's learning efficiency and generalization ability. In this homework, we explore a specific type of action representation, Visual Affordance (also called Spatial Action Map): a state-of-the-art algorithm for visual robotic pack-and-place tasks. This method played a critical role for the MIT-Princeton team's victory in the 2017 Amazon Robotics Challenge.

In **Problem 1 and 2**, implement and train a Visual Affordance model with manually labeled data. In **Problem 3**, implement a method that further improves the performance of your model on unseen novel objects. In **Problem 4**, implement Action Regression, an alternative action representation and explore its difference with Visual Affordance.

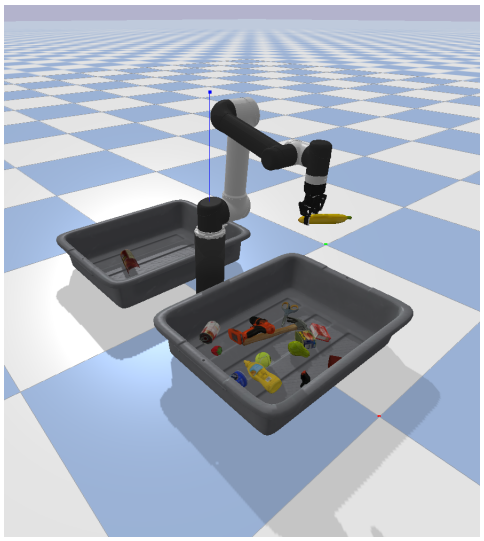

Figure 1: UR5 robot moving an object to another bin

You will submit:

1) Code: edit the provided python files and fill in the TODOs;

2) Report, hw4_report.pdf, with your answers to the write-up questions;

3) Videos: Submit screen recordings of your completed tasks. You can use any screen

recording software as you desire. One choice is OBS or QuickTime Player for Mac users.

See Section Submission instructions for more details.

Install Required Packages

Your same virtual environment from Homework 3 should work for this homework. If you don't have it already, do the following to install: Install python interpreter and dependencies using miniforge (also works for Apple Silicon, comparing to miniconda). Please download the corresponding Mambaforge-OS-arch.sh/exe file and follow the installation instructions. You can skip this part if you already installed mamba in WH3.

After installation and initialization (i.e. conda init), launch a **new** terminal and run

```
mamba env create -f environment_gpu.yaml
```

if you have an Nvidia GPU on your computer, or

```
mamba env create -f environment_cpu.yaml
```

otherwise, inside the unzipped homework zip. This will create a new environment named "comsw4733_hw4", which can be activated by running

```
mamba activate comsw4733_hw4
```

Please do not introduce any other dependencies/packages without taking prior permission from TAs.

Notes on PyBullet: This assignment uses PyBullet simulation engine extensively. We highly recommend to read the *Introduction* section in PyBullet API documentation to get some working knowledge of the engine. To interact with the simulation GUI, you can change the camera viewpoint by zooming in/out or pressing *ctrl* key and dragging the cursor at the same time. Pressing *g* key will toggle the image windows on left side.

Visual Affordance

Two key assumptions in this homework:

- 1. The robot arm's image observations come from a top-down camera, and the entire workspace is visible.
- 2. The robot performs only top-down grasping, where the pose of the gripper is reduced to 3 degrees of freedom (2D translation and 1D rotation).

Under these assumptions, we can easily align actions with image observations (hence the name spatial-action map). Visual Affordance is defined as a per-pixel value between 0 and 1 that represents whether the pixel (or the action directly mapped to this pixel) is graspable. Using this representation, the transnational degrees of freedom are naturally encoded in the 2D pixel coordinates. To encode the rotational degree of freedom, we rotate the image observation in the opposite direction of gripper rotation before passing it to the network, effectively simulating a wrist-mounted camera that rotates with the gripper.

Problem 1: Generate Training Data (5 points)

In this problem, you will get familiar with the data annotation pipeline and generate training data for the subsequent problems.

(a) Complete the class GraspLabeler (2 points)

Go to *pick_labeler.py*, update the pixel coordinate *self.coord* in the mouse callback *GraspLabeler.callback*, as well as update the gripper orientation *self.angle* in *GraspLabeler.__call__*. Note that in this homework, we follow the OpenCV convention for pixel coordinate and rotation angle (Fig. 2).

(b) Label training data (3 points)

On your **local compute**, run:

```
python3 pick_labeler.py
```

If your *GrapsLabeler* is implemented correctly, the script will launch a PyBullet GUI, then load the robot and objects. An OpenCV window will pop up (Fig. 3), where

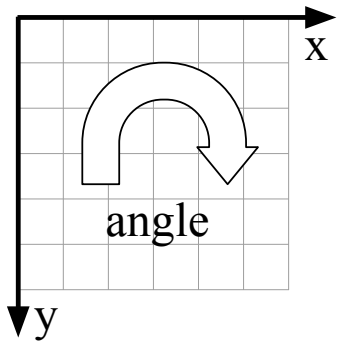

Figure 2: OpenCV pixel coordinate and rotation angle convention

Figure 3: Labeling GUI. The window might look slightly different across operating systems.

you can click left mouse button to select grasping location and use A and D keys to rotate the grasping orientation. To confirm the grasp pose, press Q or *Enter*.

Note: if the label GUI does not show up, it might be hidden behind other windows (e.g. you might find a blank icon on your MacOS Dock). After you confirm (Q or Enter), the robot will try to pick the object using the confirmed grasping pose. If grasping failed, you will be prompted to try again. If there is any discrepancy between the robot's behavior and the labeling GUI's visualization, check if there's a bug in your code.

Label 5 training objects with 12 attempts each. This usually take around 5 minutes. You might notice one of the objects, namely YcbMediumClamp, shows up similar to a small dot and makes it hard identify a grasping pose: try to click its center and position the gripper in any reasonable orientation.

Problem 2: Implement Visual Affordance model (50 points)

In this problem, implement training related features for the Visual Affordance model. The model uses the *MiniUNet* architecture from Homework 2 and 3 (except the last layer). However, its training process will be quite different.

(a) AffordanceDataset (10 points)

Read and understand *RGBDataset* in *train.py*. Then, complete *AffordanceDataset* in *affordance_model.py*. In your report, **answer** why do we use *get_gaussian_scoremap* to generate the affordance target, instead of a one-hot pixel image. Hint: if you are not sure, try to train without it after you finished implementating this problem.

(b) Data augmentation (5 points)

Complete *AugumentedDataset* class and *get_center_angle* function in *train.py*. In your report, **explain** in English what transformations are done in *self.aug_pipeline*.

(d) Training (5 points)

Training on your local CPU should be sufficient for this homework. If you want to use a Google Cloud GPU instance, upload your repo, including the data directory.

Execute the following command to start training your model:

```
python3 train.py --model affordance --augmentation
```

If the training pipeline is implemented correctly, this script should train the AffordanceModel for 101 epochs until convergence. On a decent GPU (faster than Nvidia GTX 1070), it should finish in around 1 minute. In your report, **report** your training loss and test(validation) loss. Include a image from data/affordance/training_vis. It should look like Fig. 4.

(e) Grasp prediction (10 points)

In *affordance_model.py* implement *AffordanceModel.predict_grasp* for both prediction and visualization. It should look like Fig. 5 but it doesn't have to be pixel perfect.

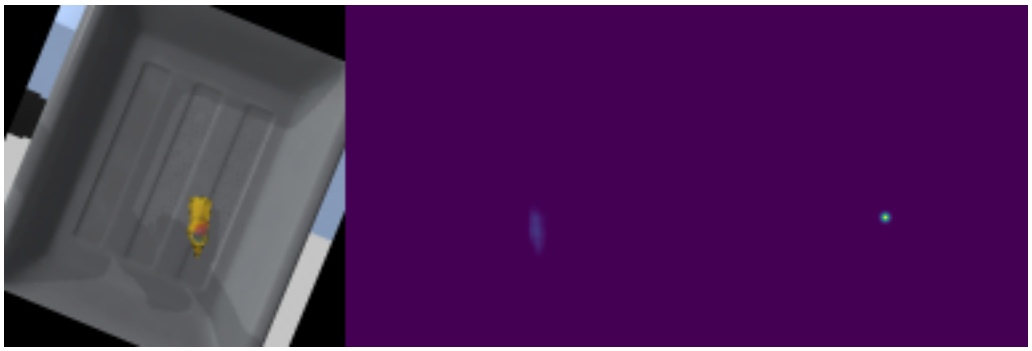

Figure 4: Example for training visualization. From left to right: input, prediction, target.

(f) Evaluation on the training set (10 points)

When your training is done, the model with the lowest loss should be saved as checkpoint data/affordance/best.ckpt (if using cloud compute, download data/affordance/best.ckpt back to your *local computer*).

Execute the following command to evaluate your model on the same set of objects used in the training dataset:

```
python3 eval.py --model affordance --task pick_training
```

If your prediction code is implemented correctly, it should have a success rate better than 70%.

Record a video of the terminal and PyBullet GUI.

In your report, **report** your success rate and include data/affordance/eval_pick_training_vis/YcbMustar
It should look similiar to Fig. 5.

(g) Evaluation on the heldout objects (10 points)

Execute the following command to evaluate your model on a novel set of objects that were **not** included in the training dataset:

```
python3 eval.py --model affordance --task pick_testing
```

Your model should do slightly worse than on the training set, expect around 50% success rate. In your report, **report** your success rate and include an image from data/affordance/eval_pick_testing_vis/. Describe how the image might be different or similar to the visualizations from the previous part.

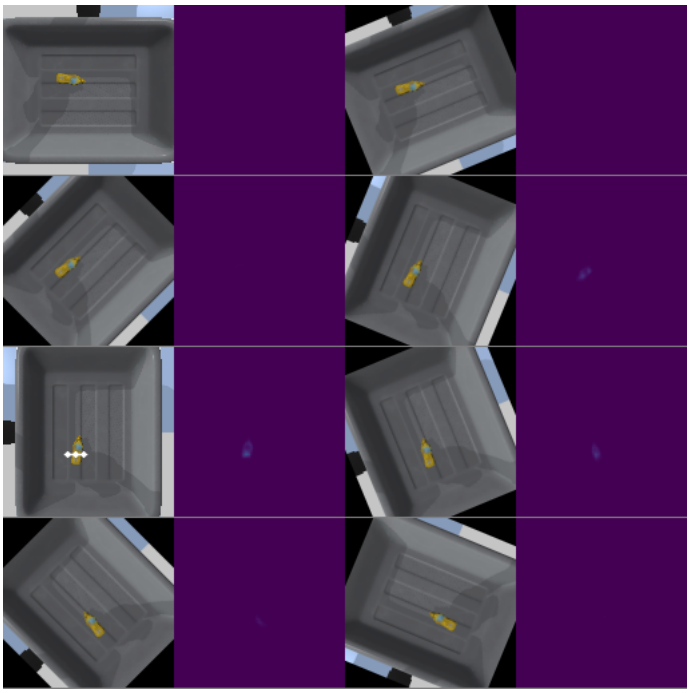

Figure 5: Example prediction visualization.

(h) Evaluation on mixed objects (5 points)

Execute:

```
python3 eval.py --model affordance --task empty_bin --seed 2
```

This command will load 15 objects into a bin, and the model tries to move all of them into another bin within 25 attempts. If your implementation is correct, the model should be able to pick up at least 8 objects, with a few more left. **Record** a video of the terminal and PyBullet GUI. In your report, **report** how many objects are left.

(i) Write-up question

Now that you have evaluated the model, in your report, try to **explain** why is this method is sample efficient, i.e. performs well on both seen and unseen objects despite given only 60 images to train.

Problem 3: Improve Test-time Performance of your Visual Affordance model (10 points)

As you may have noticed, the trained model does not succeed 100% at test time. In this problem, you will implement a method to improve the performance. Recall that, in the visual affordance model, the spatial-action map formulation is essentially looking at an image observation and finding the best pixel to act on. So if an action turns out to fail after executing the grasp, the robot can try again and select the **next best** pixel/action. See Figure 6 for an illustration.

Implement this idea by editing the TODOs in `affordance_model.py`, so that the model can hold a list of past actions. During evaluation, make the model attempt to grasp an object multiple times by setting the command line argument `--n_past_action 8` and keep track of the failed pixels' actions, such that for each new attempt, it avoids those before selecting a new action.

When you are done, evaluate the model again on training, testing, and mixed-object data like Problem 2, except now you perform multiple attempts when trying to grasp each object.

(a) Evaluate on training objects

Run:

```
python3 eval.py --model affordance --task pick_training
                    --n_past_actions 8
```

(b) Evaluate on testing objects

Run:

```
python3 eval.py --model affordance --task empty_bin --n_past_actions 8
```

(c) Evaluate on mixed objects

Run:

```
python3 eval.py --model affordance --task empty_bin --seed 2
```

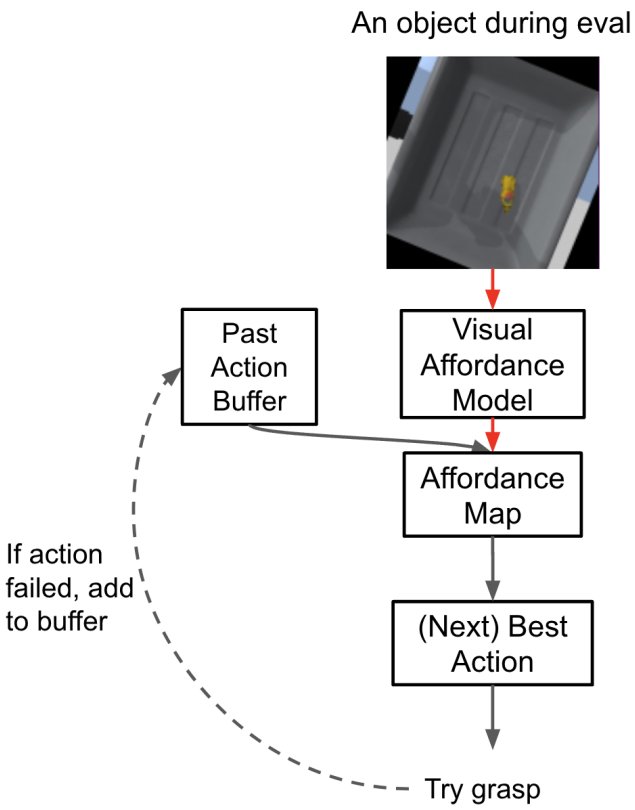

Figure 6: Illustration for the test-time improvement idea in Problem 3: add a buffer of past failed actions to the model, such that it selects the next-best actions when making a new attempt.

(d) Write-up question

In your report, discuss how the performance from each evaluation run is different from Problem 2.

Problem 4: Alternative method: Action Regression (20 points)

In the visual affordance formulation above, aligning the action with images allow us to effectively discretize the action space. In contrast, in this problem we explore an action regression approach, where a model outputs the action as a vector. This is the most common action representation prior to the popularization of visual affordance. More specifically for this homework, we will will predict the vector (x,y,angle) with each dimension normalized to between 0 and 1.

(a) Training (5 points)

In *action_regression_model.py*, complete *ActionRegressionDataset.__getitem__*, *recover_action* and *ActionRegressionModel.get_criterion*. Train your model using the command

```
python3 train.py --model action_regression --augmentation
```

In your report, **report** your training loss and test(validation) loss. Include a image from data/action_regression/training_vis.

(b) Evaluation on training objects (5 points)

In *action_regression_model.py*, complete *ActionRegressionModel.predict_grasp*. Execute:

```
python3 eval.py --model action_regression --task pick_training
```

Record a video of the terminal and PyBullet GUI. In your report, **report** your success rate and include data/action_regression/eval_pick_training_vis/YcbMustardBottle_2.png.

(c) Evaluation on all objects (10 points)

Execute:

```
python3 eval.py --model action_regression --task empty_bin
```

Record a video of the terminal and PyBullet GUI. In your report, **report** how many objects are left. Also **explain** why is this method perform worse comparing to Visual Affordance.

Submission instructions

- Generate separate URLs for each video containing solution to problem 2(f), 2(g), 3(b), 3(c) and 4 (see instructions for generating video URL at the bottom) in order. Place the URL in *url_main.txt* file.
- Compile all written answers and the video URL to *hw4_report.pdf*.
- Put the report, all python code, as well as the assets and data directory to a folder named **[Your-UNI]_hw4** and then upload the compressed directory with name **[Your-UNI]_hw4.zip**. For example, cc4617_hw4.zip.

Please make sure that you adhere to these submission instructions. **TAs can deduct up to 10 points for not following these instructions properly.**

Please note that for all students, we will look at both the code and the video for grading. If significant discrepancies are found between the submitted videos and the results from manually running the code, we will report the student to Academic Committee. **We will also randomly sample a fair proportion of students and ask them for a homework interview**.

Generating video URL

Upload video to your personal google drive account (**not Lionmail**). Generate shareable link such that **anyone with the link can view the file**. For testing, open the link in Incognito Mode / Private window. If done properly, you should be able to view the video without logging in.

You should also make sure that the *Modified* date is visible. For this, click on burger button (three vertical dots) on top-right. Click on *Details*. You should be able to see the *Modified* date (see Figure 7).

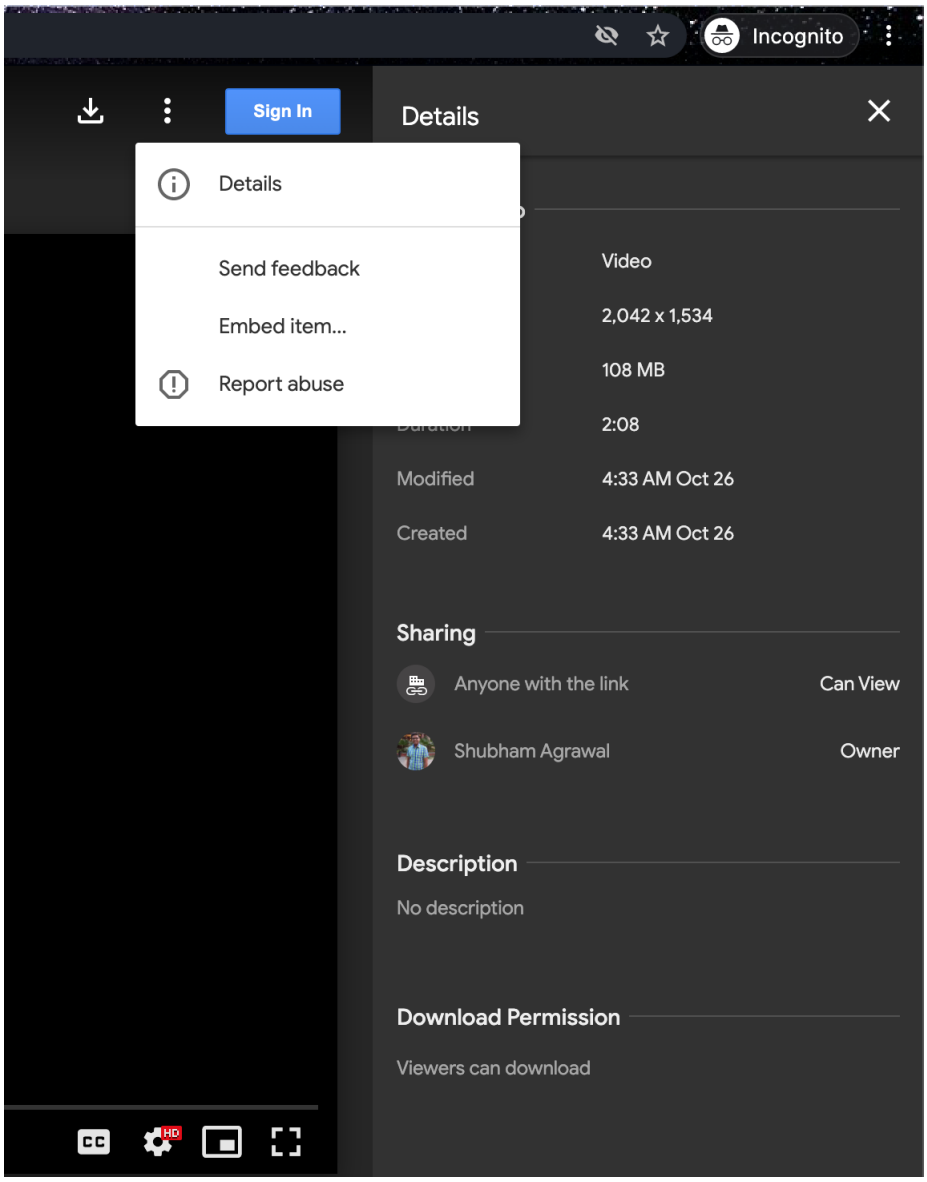

Figure 7: Video URL: should be accessible in Incognito / Private window. Modified date should be visible.