Guide for recreating experiments and using this benchmarking platform.

These experiments have been created and tested using Ubuntu 16.04 with python>=3.6, support for alternative systems is not currently provided.

Running Experiments in Simulation

It is recommended that you run experiments in simulation before setting up any physical experiments, this will ensure that algorithm logic works.

Setting up

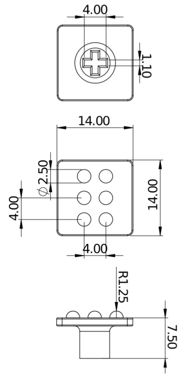
```
# requires python>=3.6
# It is recommended that you use a virtual environment for this set up

# clone and install the repo (this may take a while)
git clone https://github.com/ac-93/braille_rl.git
cd braille_rl
pip install -e .

# install Spinningup from openAI
git clone https://github.com/openai/spinningup.git
cd spinningup
pip install -e .

# install common robot interface
git clone https://github.com/jlloyd237/cri.git
cd cri
python setup.py install

# install video stream processor
git clone https://github.com/jlloyd237/vsp.git
cd vsp
python setup.py install

# install python3-v4l2capture
git clone https://github.com/atareao/python3-v4l2capture.git
cd python3-v4l2capture
python setup.py install

# test the installation by running a training script in simulation, from the base direct
python algos/dd_dqn_algo/train_discrete_model.py
```

Recreating Experiments

All experiment parameters are set in the `train_(discrete/cont)_model.py` scripts located in the corresponding algorithm directories. This will save configurations and if specified trained models to the `saved_models` directory. An example configuration for each of the possible environments and algorithm combinations for seed 1 is given in the `saved_models` directory. It is recommended to use the simulated environment for fast prototyping. These training script examples can be copied and run in any directory. Training can also be stopped and resumed from a saved model using the `resume_training.py` scripts, this is helpful for running longer experiments on a physical robot.

Running Experiments on Physical Platform

Keyboard



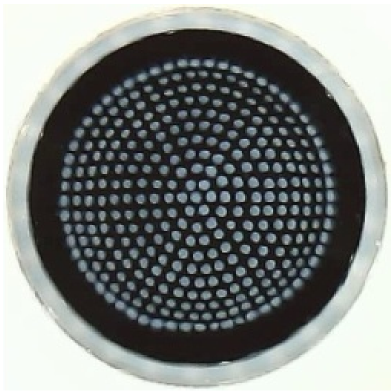
The keyboard used for these experiments is the DREVO Excalibur 84 Key Mechanical Keyboard with Cherry MX Black switches (available here).

The 3d printable keycaps are designed to fit on Cherry MX switches, the Black switches have been used as they offer a relatively stiff switch (requiring 60N for actuation) making the task slightly easier.

We printed the keycaps using a high precision 3d printer (Stratasys Objet260 Connex), this ensures that the features to be interpreted are accurate and consistent.

Ideally, recreating these experiments will use the same keyboard or the same key switches, if these components are unavailable a similar experiment can be created using alternatives but differences should be noted. If using an alternative keyboard be sure that evdev accesses the right keyboard by specifying the correct keyboard name in `ur5GymEnv.py` files. Additionally, keys are currently presumed to be spaces 19mm apart for discrete tasks with safety limits that are specific to this keyboard, a tap depth actuation point of 2mm is also assumed. These will also need to be adjusted to match alternative keyboards.

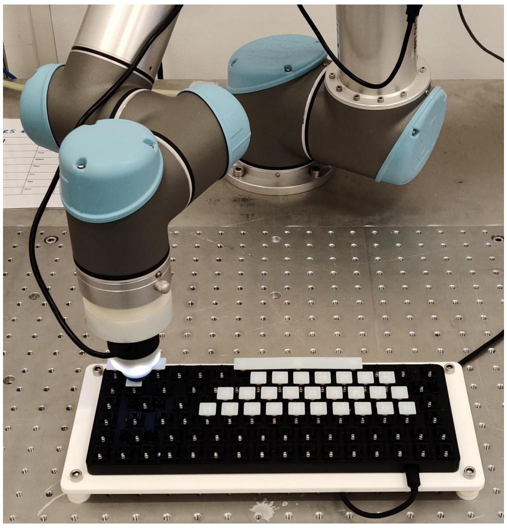
Configuring Sensor



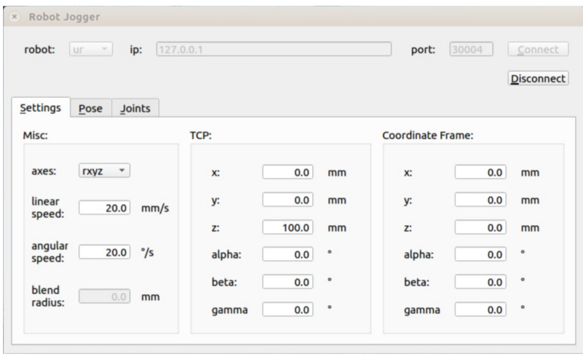
The tactile sensor used in this work is a modified version of the BRL tactile fingertip (TacTip), for more details on this sensor check https://www.bristolroboticslab.com/tactile-robotics. This sensor uses a standard USB webcam as the source for gathering tactile images. This is reliant on the Video Stream Processor (VSP) library created by co-author John LLoyd, check the Github repo for more detail.

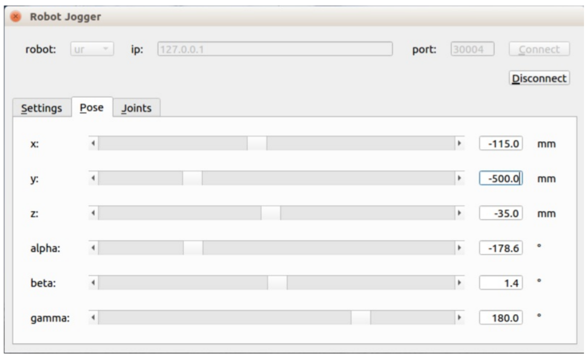
If recreating this work with alternative tactile sensors the files that will need to be modified are `envs/robot/cont_ur5_braille_env/ur5_w_tactip.py` and `envs/robot/disc_ur5_braille_env/ur5_w_tactip.py`. These files will need to be modified such that the `get_observation()` function returns a current observation from your specific tactile sensor. If you are using an optical tactile sensor where the images are used as an observation then the VSP library should work as long as the correct device_path is set in the previously mentioned files. (**Note:** *USB cameras use frame buffers of fixed size, ensure that enough frames are processed to cycle this buffer on each iteration such that the observation is from the current time step and not lagging behind.*) To allow for more experimentation the image processing is applied later in the algorithms sections. The `algos\rl_utils.py` file contains a `process_image_observation()` function where cropping, resizing, thresholding, shifting, etc can be applied to the raw images gathered from the sensor.

Configuring Robot Arm



In this work the control of a the robot arm leverages the Common Robot Interface (CRI) framework developed by co-author John Lloyd, check the [Github repo]() for more detail. This allows for common robotic arm utilities to be used across several different robot arms. The arms currently supported are ABB robot controllers, and UR controllers that implement the Real-Time Data Exchange (RTDE) protocol. The functionality that we use in this project consists of relatively simple tasks such as setting up workspaces and linear moves, unsupported robot arms can be used for this project but control code will have to be re-implemented by the user. Ideally, this framework will grow and offer support for simple functionality across a broad range of robot arms. A useful robot-jogger tool is also provided with the CRI framework, this allows you to specify robot arm settings and find relative positions in joint or world coordinates.





When configuring the robot arm we use mm and degree units throughout. Depending on the tasks, two files will need to be edited for different robotic arm and sensor setups. These are `envs/robot/disc_ur5_braille_env/ur5_w_tactip.py` for the discrete action tasks and `envs/robot/`

`cont_ur5_braille_env/ur5_w_tactip.py` for the continuous action tasks. In both cases

```
self.robot_tcp  = [x, y, z, alpha, beta, gamma] # tool center point
self.base_frame = [x, y, z, alpha, beta, gamma] # origin of arm base
self.home_pose  = [x, y, z, alpha, beta, gamma] # safe position of arm
```

needs to be specified. An optional `self.sensor_angle = theta` can be used to orientate the sensor.

Depending on both the task and the action space different origin points for the work frames needs to be set up, in all cases `self.work_frame = [x, y, z, alpha, beta, gamma]` but these value are chosen according to:

Task	Description
`Disc-Arrows`	3.5mm above center of DOWN arrow
`Disc-Alpha`	3.5mm above center of Q Button
`Cont-Arrows`	3.5mm above center of box covering all arrow keys
`Cont-Alpha`	3.5mm above center of box covering all alphabet keys

The robot-jogger tool is useful for finding these work frame positions.

Additionally 'tap_move' and 'press_move' can also be adjusted slightly to better suit the specific setup. In the discrete setting be sure to check that each movement gathers a tactile observation without activating a button and in the continuous setting the tap depth range allows for a range of actions that do not activate the button.

(**Note:** *If using the robot jogger be sure to specify a matching tool center point in the code.*)

Evaluating and Comparison of Trained Models

There are three comparisons that will offer the most insight when comparing between different sensors or different algorithms. These are

- Accuracy of trained model e.g. how many miss-presses will a trained agent make when typing example key sequences using the `evaluate_model.py` scripts.
- Efficiency of trained model e.g. how many steps taken to complete these same key sequences.
- Sample efficiency over training, this can be measured by the number of training steps required to reach asymptotic performance. In this work we use the first epoch to acheive 95% of the maximum episodic return acheived throughout training, this can be found by running the 'plot_progress' function.

Evaluation scripts are provided in the algorithm sub directories, this is done per algorithm due to slight difference in the saved models. These scripts can be called with a saved model directory, saved model number and seed. e.g. `run_evaluation(model_dir='saved_models/sim/discrete/arrows/dd_dqn/dd_dqn_s2/', model_save_name='tf1_save', seed=1)`. This will test the accuracy and efficiency of the trained model when typing a series of sample sequences as done in the paper. A confusion matrix will also be shown, this is to give some insight into the types of inaccuracies made by the trained model. This can be used to compare between either new/adjusted algorithms or new sensors.

The sample efficiency can be found by running the functions found in 'plot_progress.py' and providing the 'progress.txt' file found in saved_model directories. This will return graphs of training curves similar to those used in the paper for comparison. Additionally the first epoch in which an average return of >= 95% of the max return for that training run will be displayed. This is a measure of sample efficiency and can be used along with the other metrics for comparison.