Tango Wiki

July 31, 2020

Wiki

Welcome to the Robot-task-planning wiki! This implementation contains all the models mentioned in the paper for next-action prediction.

Introduction

In order to effectively perform activities in realistic environments like a home or a factory, robots often require tools. Determining if a tool could be effective in accomplishing a task can be helpful in scoping the search for feasible plans. This work aims at learning such "commonsense" to generalize and adapt to novel settings where one or more known tools are missing with the presence of unseen tools and world scenes. Towards this objective, we crowd-source a data set of humans instructing a robot in a Physics simulator perform tasks involving multi-step object interactions with symbolic state changes. The data set is used to supervise a learner that predicts the use of an object as a tool in a plan achieving the agent's goal. The model encodes the agent's environment, including metric and semantic properties, using gated graph convolutions and incorporates goal- conditioned spatial attention to predict the optimal tool to use. Compared to the baseline Tango performs much better in action prediction and plan execution on the generalization dataset on the home and factory domains respectively.

Navigating this wiki

This wiki consists of the following sections:

1. Environment Setup
2. Working with the simulator
3. Creating the dataset
4. Training your own models
5. Replicating the results in the pape

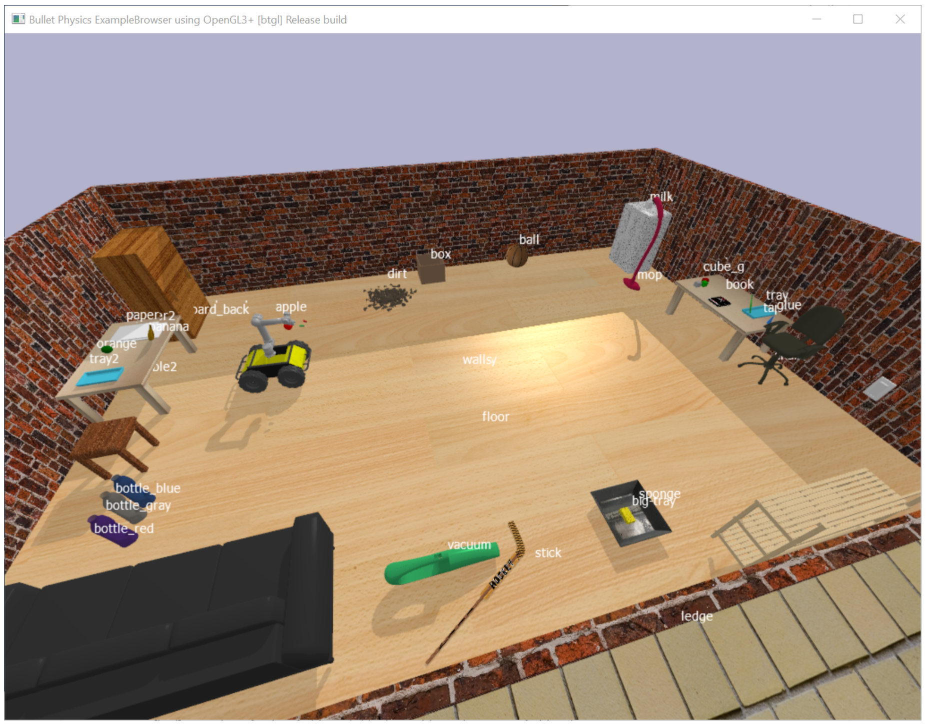


Figure 1: Simulation environment

Directory Structure

Folder/File	Utility
app.py	This is the main file to run the data collection platform. This file starts the web server at the default port 5000 on localhost. It starts an instance of PyBullet on the system and exposes the simulator to the user at the appropriate website.
train.py	This is the main file used to train and evaluate all the models as mentioned in the paper.

Folder/File	Utility
husky__ur5.py	This is the main file for the PyBullet simulator. It is responsible for loading the PyBullet simulator, running the appropriate action and sending the appropriate exceptions wherever applicable. It also checks if the goal specified has been completed.
src/GNN/CONSTANTS.py	This file contains the global constant used by the training pipeline like the number of epochs, hidden size used etc.
src/GNN/dataset__utils.py	This file contains the Dataset class, which can be used to process the dataset in any form as required by the training pipeline.
src/GNN/*models	These contain the different PyTorch models that were used for training the system.
src/datapoint.py	This contains the datapoint class. All datapoints found in the dataset are an instance of this class.
jsons/embeddings	These contain the files corresponding to fasttext and conceptnet embeddings.
jsons/*__goals	These contain the goals which can be completed by the robot in the factory and the home domain.
jsons/*__worlds	These contain the different world instances in the home and factory domain.
jsons/*.json	These are configuration files for the simulator and the webapp. These define the different actions possible in the simulator, the different objects present in the simulator, states possible, readable form of actions to show on the webapp etc.

Folder/File	Utility
models/*	This folder contains the stl and urdf files for all models which are loaded by the physics simulator used i.e Pybullet.
templates/*	These are the templates that are used by the webapp to load the different tutorial webpages along with the actual data collection platform.
dataset/*	These are files corresponding to training, test and generalization datasets.

License

Environment Setup

To set up your environment, follow the steps below:

```
$ python3 -m venv commonsense_tool
$ source commonsense_tool/bin/activate
(commonsense_tool) $ git clone https://github.com/reail-iitd/Robot-task-planning.git
(commonsense_tool) $ cd Robot-task-planning
(commonsense_tool) $ pip3 install -r requirements.txt
```

We used a system with 32GB RAM, i7 Octa-core CPU and 8GB graphics RAM to run our simulator. The models were trained using the same system. The simulator and training pipeline has been tested on Windows, MacOS and Ubuntu 16.04.

Known Errors

If you are using >=32GB RAM, graphics card with >=8GB GRAM and Windows 10 or above, or MacOS 10.15 or above or Ubuntu 16 or above you should not get any errors. However, in some cases, it is possible to run the setup even if you do not have these exact requirements.

1. **MacOS 10.14 or below:** You may get a multi-processing error for which you may add the following lines in *app.py* after the line `inp = "jsons/input_home.json"`:

```
mp.set_start_method('spawn')
queue_from_webapp_to_simulator = mp.Queue()
queue_from_simulator_to_webapp = mp.Queue()
queue_for_error = mp.Queue()
queue_for_execute_to_stop = mp.Queue()
queue_for_execute_is_ongoing = mp.Queue()
```

Working with the simulator

Tutorial on how to work with the simulator.

Prerequisite: Follow the instructions on the Environment Setup Section.

File Structure

The simulator is all coded in `husky_ur5.py` which then interfaces with the web-app `app.py` to create a crowd-sourcing platform. However, in this wiki we only discuss how to work with the simulator. `The goal specifications for $DOMAIN are encoded in ‘jsons/$DOMAIN_goals/.json’. The scene specifications for $DOMAIN are encoded in jsons/$DOMAIN_worlds/.json’`. The goal specifications are declarative and are encoded as a list of constraints that need to be valid. The scene specifications are encoded as a list of objects names with initial position, orientation and possible states (as a list of discrete positions and orientations).

Sample plan inputs for the simulator are encoded in `jsons/input_$DOMAIN.json` which is a list of symbolic actions. Possible symbolic actions and their decomposition to sub-symbolic actions are given in `src/actions.py`. A sample plan file is shown below.

```
{
    "actions":[
        {
            "name": "changeState",
            "args": ["cupboard", "open"]
        },
        {
            "name": "pickNplaceAonB",
            "args": ["apple", "box"]
        },
        {
            "name": "pickNplaceAonB",
            "args": ["banana", "box"]
        },
        {
            "name": "pickNplaceAonB",
            "args": ["orange", "box"]
        },
        {
            "name": "pickNplaceAonB",
            "args": ["box", "cupboard"]
        }
    ]
}
```

Running the simulator

To run a simulator with a world scene specification $SCENE_PATH, goal specification $GOAL_PATH and input plan $INPUT_PATH, use the following command:

```
python3 husky_ur5.py --world $SCENE_PATH --goal $GOAL_PATH
 --input $INPUT_PATH --display tp
```

For example, if you want to run the simulator with a custom plan specified in `jsons/input_home.json` on world_home0 and goal of transporting fruits to cupboard, use the following command:

```
python3 husky_ur5.py --world jsons/home_worlds/world_home0.json --goal
jsons/home_goals/goal2-fruits-cupboard.json --input jsons/input_home.json --display tp
```

This command will open an OpenGL simulator window. Click on the window and press **G** to minimize the debugging tabs. Use `mouse scroll` to zoom in or out, use `Ctrl+Scroll click` to pan, and `Ctrl+Left click` to rotate the camera.

Note: Without the `--display tp` option, the simulator will save a sequence of first-person images in the `logs/` directory.

The plan execution, when finished will close the simulator and save the datapoint as `test.datapoint` file.

Adding a new object in a scene

To add a new object to the scene, follow the steps below: 1. Create or download an object (OBJ) file and save it and corresponding material (MTL) texture file in `models/meshes/$OBJECT_NAME/`. For example, if you want to create an *apple* object, create two new files `models/meshes/apple/10162_Apple_v01_l3.obj` and `models/meshes/apple/10162_Apple_v01_l3.mtl`. Make sure the linking of MTL file inside the OBJ file is relative and not absolute. Examples of online repositories of 3D objects are Free3D and CGTrader. 2. Create a Unified Robotic Description Format (URDF) file for the object. For example, if you want to create an *apple* object, create a new file `models/urdf/apple.urdf`. The URDF file needs to specify contact, visual, geometric and collision details where you need to mention the path of the OBJ file and scale. The URDF file for the *apple* object is shown below:

```
<?xml version="1.0" encoding="utf-8"?>
<robot name="apple">
  <link name="base_link">
    <contact>
      <lateral_friction value="0.5"/>
      <rolling_friction value="0.0"/>
      <contact_cfm value="0.0"/>
      <contact_erp value="1.0"/>
    </contact>
    <inertial>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <mass value="5"/>
      <inertia ixx="0" ixy="0" ixz="0" iyy="0" iyz="0" izz="0"/>
    </inertial>
    <visual>
      <origin rpy="1.57 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="models/meshes/apple/10162_Apple_v01_l3.obj"
             scale="0.0007 0.0007 0.0007"/>
      </geometry>
      <material name="tray_material">
        <color rgba="1 0.03 0 1"/>
      </material>
    </visual>
    <collision>
      <origin rpy="1.57 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="models/meshes/apple/10162_Apple_v01_l3.obj"
             scale="0.0006 0.0007 0.0007"/>
      </geometry>
    </collision>
  </link>
</robot>
```

3. Add the specifications of the object in the `jsons/objects.json` file. Specify the name, URDF file path, list of constraints which might include “noise” if Gaussian noise is to be added to position when placing in the scene and “horizontal” if object needs to be kept horizontal at all times. Add tolerance in distance, constraint link (default is -1), constraint c_pos i.e. relative position from object center when another object is placed on this object. Add constraint pos i.e. the relative position from object center when this object is placed on another. Add distance to be kept from UR5 gripper when constrained to it. Add object properties like “Movable”, “Surface”, “Printable”, “Switchable”, etc. For *apple* object the specs are given below.

```
    {
        "name": "apple",
        "urdf": "models/urdf/apple.urdf",
        "constraints": ["noise"],
        "tolerance": 1.2,
        "constraint_link": -1,
        "constraint_cpos": [[0, 0, -0.05]],
        "constraint_pos": [],
        "ur5_dist": [0.3,0,0],
        "properties": ["Movable"],
        "size": [0.2, 0.2, 0.2]
    },
```

4. Add the object in the scene. For example, in world_home0 the description of *apple* object is given below.

```
{
    "name": "apple",
    "position": [-1, -4.5, 0.7],
    "orientation": [],
    "states": []
},
```

5. Add ConceptNet Embedding of this object in `jsons/embeddings/conceptnet.vectors` and FastText Embedding in `jsons/embeddings/fasttext.vectors`.

Adding a goal specification

To add a goal specification, create a goal file, say `jsons/home_goals/goal1_milk_fridge.json`. Specify the declarative goal predicates as a list. An example is shown below. The implementation of goal checking can be seen in the `checkGoal()` function in `src/utils.py`.

```
{
    "goals":[
        {
            "object": "milk",
            "target": "fridge",
            "state": "",
            "position": "",
            "tolerance": 0
        },
        {
            "object": "fridge",
            "target": "",
            "state": "close",
            "position": "",
            "tolerance": 0
        }
    ],
    "text": "Put the milk carton inside the fridge.",
    "goal-objects": ["milk", "fridge"],
        "goal-vector": [float list of 300 size]
}
```

The goal vector can be calculated as the average of the embeddings of all words in goal "text". For this, we use ConceptNet Embeddings. To do this follow the steps below (here *fname* is the file path for the extracted file and *text* is the text for the new goal):

```
$ python3
>>> from src.extract_vectors import *
>>> data = load_all_vectors(fname)
>>> list(form_goal_vec(data, text))
```

So for the extracted file path `numberbatch-en.txt` and new goal text “put milk in cupboard”, use the following:

```
$ python3
>>> from src.extract_vectors import *
>>> data = load_all_vectors("numberbatch-en.txt")
>>> list(form_goal_vec(data, "put milk in cupboard"))
```

Visualizing datapoints

To visualize datapoints, use the `analyze.py` code. There are many visualizations possible using this file. 1. To visualize complete datapoints use the `runDatapoint()` method. An example is given below. This method works only when the datapoint does not have any errors in the plan. Note, due to stochasticity in the simulator, a plan working earlier might not work again as the object positions and constraints may act randomly in different runs.

```
python analyze.py  "runDatapoint('dataset/home/goal1-milk-fridge/world_home0/7')"
```

2. To visualize object-centric graphs of a datapoint, use the `drawGraph()` method. An example is given below. This method saves a sequence of graphs including that of the initial state and world state after every action.

```
python analyze.py  "drawGraph('dataset/home/goal1-milk-fridge/world_home0/7')"
```

A sample graph is shown below. The *agent node* is shown in green, *tools* is black and *objects with states* in red. The relations of *Close* are shown in green (only populated for agent), *On* in black, *Inside* in red and *Stuck to* in blue. The legend for node IDs to objects are given below:

Home: 0: floor, 1: walls, 2: door, 3: fridge, 4: cupboard, 5: husky, 6: table, 7: table2, 8: couch, 9: big-tray, 10: book, 11: paper, 12: paper2, 13: cube gray, 14: cube green, 15: cube red, 16: tray, 17: tray2, 18: light, 19: bottle blue, 20: bottle gray, 21: bottle red, 22: box, 23: apple, 24: orange, 25: banana, 26: chair, 27: ball, 28: stick, 29: dumpster, 30: milk, 31: shelf, 32: glue, 33: tape, 34: stool, 35: mop, 36: sponge, 37: vacuum, 38: dirt.

Factory: 0: floor warehouse, 1: 3D printer, 2: assembly station, 3: blow dryer, 4: board, 5: box, 6: brick, 7: coal, 8: crate green, 9: crate peach, 10: crate red, 11: cupboard, 12: drill, 13: gasoline, 14: generator, 15: glue, 16: hammer, 17: ladder, 18: lift, 19: long shelf, 20: mop, 21: nail, 22: oil, 23: paper, 24: part1, 25: part2, 26: part3, 27: platform, 28: screw, 29: screwdriver, 30: spraypaint, 31: stick, 32: stool, 33: table, 34: tape, 35: toolbox, 36: trolley, 37: wall warehouse, 38: water, 39: welder, 40: wood, 41: wood cutter, 42: worktable, 43: ramp, 44: husky, 45: tray.

3. To print the complete datapoint use the `printDatapoint()` method. An example is shown below.

```
python analyze.py  "printDatapoint('dataset/home/goal1-milk-fridge/world_home0/7')"
```

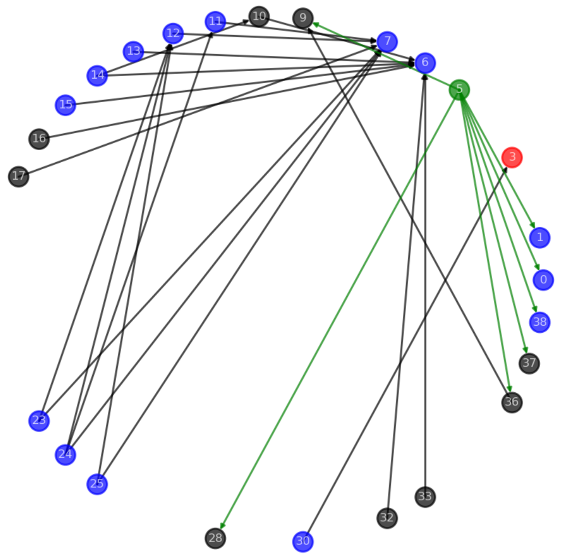


Figure 2: Sample graph

Creating the dataset

This page gives a tutorial on how to create the dataset using the crowd-sourcing application.

Prerequisite: Follow the instructions on the Environment Setup Section.

Setup Web Interface for Data Collection

To execute the website that is needed for data collection, use the following command:

```
$ python3 app.py --randomize
```

The website can now be accessed by using the link (http://0.0.0.0:5000/ on MacOS/Linux or http://localhost:5000/ on Windows). For all the arguments that can be provided, look at the help description.

```
$ python3 app.py --help
```

The web-interface can now be used to create a dataset, wherein all datapoints would be saved in the `dataset/` directory.

Tutorial for Human Teachers

In order to familiarize the humans giving us the data for collection, a tutorial was constructed and is the first thing which is shown when the website is launched. These videos have been put in the `static/` folder in the repository.

Dataset

Here is how the dataset structure should look like:

```
dataset
 home
 factory
 test
     home
     factory
```

The dataset is organized as follows. We have 8 different goals and 10 different world instances for both the domains, home and factory. Each domain has 8 directories corresponding to the goals possible for the domain. These goals itself, contain directories for the 10 different world instances. Each goal for each world instance in a particular domain thus has a number of different human demonstrations, and these are saved in the form of a .datapoint file for each plan. This is a pickled instance of the Datapoint class found in `src/datapoint.py` and contains all the information needed about the plan. Refer to the class for more information.

This is an example of basic datapoint file.

```
World = world_home1
Goal = goal1-milk-fridge
Symbolic actions:
{'name': 'changeState', 'args': ['fridge', 'open']}
{'name': 'pickNplaceAonB', 'args': ['milk', 'fridge']}
{'name': 'changeState', 'args': ['fridge', 'close']}
```

Training your own models

This page gives a tutorial on how to train the models mentioned in the Tango paper.

Prerequisite: Follow the instructions on the Environment Setup Section.

Pre-trained models: This code already has all trained models included in the `trained_models` folder.

Model Training

All the models mentioned in the paper can be trained through the command

```
$ python3 train.py $DOMAIN action $MODEL_NAME train
```

Here $DOMAIN can be home/factory.

MODEL_NAME specifies the specific PyTorch model that you want to train. Look at `src/GNN/models.py` (ToolNet) or `src/GNN/action_models.py` (Tango) to specify the name. They are specified here for reference.

MODEL_NAME	Name in paper
GGCN_Auto_Action	GGCN+Auto (Baseline)
GGCN_Metric_Attn_Aseq_L_Auto_Cons_C_Action	Tango
Final_GGCN_Action	- GGCN
Final_Attn_Action	- Goal-Conditioned Attn
Final_Cons_Action	- Constraints
Final_Auto_Action	- Autoregression
Final_Aseq_Action	- Temporal Action History
Final_L_Action	- Factored Likelihood

This command will train MODEL_NAME on the training dataset for `NUM_EPOCHS` epochs specified in `src/GNN/CONSTANTS.py`. It will save a checkpoint file `trained_models/DOMAIN/MODEL_NAME_EPOCH.ckpt` after the `EPOCH` epoch. In the end, it will output the epoch (say `N`) corresponding to the maximum policy accuracy using early stopping criteria. Rename the `trained_models/DOMAIN/MODEL_NAME_N.ckpt` file to `trained_models/DOMAIN/MODEL_NAME_Trained.ckpt` for testing. You may delete the other checkpoint files.

Sample Commands

To train the best model in **home** domain:

```
python3 train.py home action GGCN_Metric_Attn_Aseq_L_Auto_Cons_C_Action train
```

To train the best model in **factory** domain:

```
python3 train.py factory action GGCN_Metric_Attn_Aseq_L_Auto_Cons_C_Action train
```

To train the ablated **"- GGCN"** model in **home** domain:

```
python3 train.py home action Final_GGCN_Action train
```

Replicating results

This page explains how to replicate results in the Tango paper.

Prerequisite: Follow the instructions on the Environment Setup Section. You should already have all trained models stored in `./trained_models/home/` for home domain and `./trained_models/factory/` for factory domain.

Evaluation Metrics

We use two evaluation metrics: 1. **Action Prediction Accuracy:** This is the fraction of tool interactions predicted by the model, which matched the human demonstrated action `a` for a given state `s`. 2. **Plan Execution Accuracy:** This is the fraction of estimated plans that are successful, i.e., can be executed by the robot in simulation and attain the intended goal (with an upper bound of 50 actions in the plan).

These two metrics are calculated on the Test set and Generalization Test (Gen-Test) set.

Determine the accuracy of a model

Accuracy of any model can be obtained using the following command:

```
$ python3 train.py $DOMAIN action $MODEL_NAME $EXEC_TYPE
```

This command looks for the file `trained_models/DOMAIN/MODEL_NAME_Trained.ckpt` if you are using a pre-trained model, so make sure that the file corresponding to this command is present.

Here DOMAIN can be home/factory.

MODEL_NAME specifies the specific PyTorch model that you want to train. Look at `src/GNN/models.py` (ToolNet) or `src/GNN/action_models.py` (Tango) to specify the name. They are specified here for reference.

MODEL_NAME	Name in paper
GGCN_Auto_Action	GGCN+Auto (Baseline)
GGCN_Metric_Attn_Aseq_L_Auto_Cons_C_Action	Tango
Final_GGCN_Action	- GGCN
Final_Metric_Action	- Metric
Final_Attn_Action	- Goal-Conditioned Attn
Final_Cons_Action	- Constraints
Final_C_Action	- ConceptNet
Final_Auto_Action	- Autoregression
Final_Aseq_Action	- Temporal Action History
Final_L_Action	- Factored Likelihood

EXEC_TYPE can be as follows:

EXEC_TYPE	Meaning
accuracy	Determine the action prediction accuracy of the given model on the Test set
generalization	Calculate the plan execution accuracy of the given model on the GenTest set
policy	Calculate the plan execution accuracy of the given model on the Test set

Sample Commands

To find the action prediction accuracy of trained **"- GGCN"** model:

```
python3 train.py home action Final_GGCN_Action accuracy
```

To find the plan execution accuracy on Test Set of trained **"- GGCN"** model:

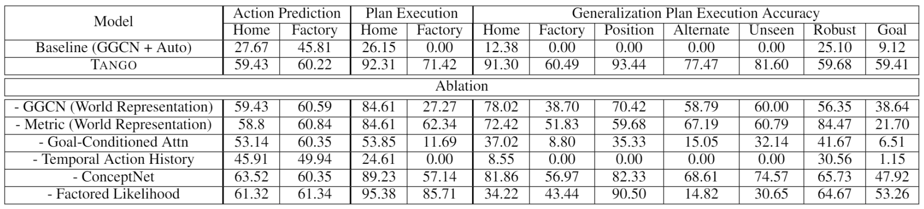
```
python3 train.py home action Final_GGCN_Action policy
```

To find the plan execution accuracy on GenTest Set of trained **"- GGCN"** model:

```
python3 train.py home action Final_GGCN_Action generalization
```

Replicating the results table

We now describe how to replicate the below table given in the Tango paper.



Model	Action Prediction		Plan Execution		Generalization Plan Execution Accuracy						
	Home	Factory	Home	Factory	Home	Factory	Position	Alternate	Unseen	Robust	Goal
Baseline (GGCN + Auto)	27.67	45.81	26.15	0.00	12.38	0.00	0.00	0.00	0.00	25.10	9.12
Tango	59.43	60.22	92.31	71.42	91.30	60.49	93.44	77.47	81.60	59.68	59.41
Ablation											
- GGCN (World Representation)	59.43	60.59	84.61	27.27	78.02	38.70	70.42	58.79	60.00	56.35	38.64
- Metric (World Representation)	58.8	60.84	84.61	62.34	72.42	51.83	59.68	67.19	60.79	84.47	21.70
- Goal-Conditioned Attn	53.14	60.35	53.85	11.69	37.02	8.80	35.33	15.05	32.14	41.67	6.51
- Temporal Action History	45.91	49.94	24.61	0.00	8.55	0.00	0.00	0.00	0.00	30.56	1.15
- ConceptNet	63.52	60.35	89.23	57.14	81.86	56.97	82.33	68.61	74.57	65.73	47.92
- Factored Likelihood	61.32	61.34	95.38	85.71	34.22	43.44	90.50	14.82	30.65	64.67	53.26

Figure 3: Results table

To replicate the *Action Prediction* columns (action prediction accuracy), run the following commands for all models for both domains. The value in the table is the accuracy corresponding to the test set.

```
python3 train.py $DOMAIN action $MODEL_NAME accuracy
```

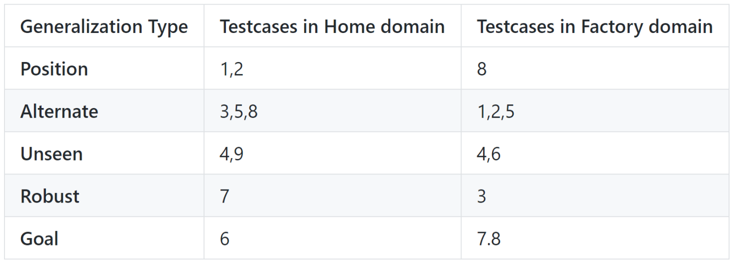
To replicate the *Plan Execution* columns (plan execution accuracy on the Test set), run the following commands for all models for both domains. The value in the table is the fraction of `Correct` plans (the first number in the output triple).

```
python3 train.py $DOMAIN action $MODEL_NAME policy
```

To replicate the *Generalizaiton Plan Execution Accuracy* columns (plan execution accuracy on the GenTest set), run the following commands for all models for both domains. The value in the table corresponding to the $DOMAIN is the fraction of `Correct` plans (the first number in the output triple).

```
python3 train.py $DOMAIN action $MODEL_NAME generalization
```

There are 9 test cases in the home domain and 8 in the factory domain. To obtain the table values corresponding to the different generalization types, take the average of the fraction of `Correct` plans corresponding to the test cases in both home and factory domains. For example, to calculate the **Position** accuracy take the average of the three numbers: fraction of correct plans in testcases 1 and 2 in home and testcase 8 in factory. Refer the table below for other generalization types.



Generalization Type	Testcases in Home domain	Testcases in Factory domain
Position	1,2	8
Alternate	3,5,8	1,2,5
Unseen	4,9	4,6
Robust	7	3
Goal	6	7.8

Figure 4: Generalization Types