# ilpyt: Imitation Learning Research Code Base in PyTorch

**Amanda C. Vu**
The MITRE Corporation
McLean, VA 22102
amandavu@mitre.org

**Alex Tapley**
The MITRE Corporation
McLean, VA 22102
atapley@mitre.org

**Brett Bissey**
The MITRE Corporation
McLean, VA 22102
bbissey@mitre.org

## Abstract

Imitation learning, or learning by example, is an intuitive way to teach new behaviors to autonomous systems. With the parallel growth of deep reinforcement learning research [1, 2], a rich taxonomy of imitation learning algorithms ranging from behavioral cloning and inverse reinforcement learning algorithms to model-based and model-free algorithms has emerged [3]. In this paper, we present *ilpyt*, a research code base which implements a variety of imitation learning and reinforcement learning algorithm families in a shared infrastructure. It contains implementations of popular deep imitation learning algorithms, written in a modular fashion for easy user customization and novel implementations. The provided algorithm implementations were done in Python using PyTorch [4], and the overall library organization is inspired by the popular reinforcement learning research library, rlpyt [5]. This white paper summarizes the key features and basic usage of the *ilpyt* library, as well as benchmark results for the implemented algorithms in several representative OpenAI Gym environments [6]. *ilpyt* is available for download at `https://github.com/mitre/ilpyt`.

## 1 Introduction

Imitation learning, or learning by example, is an intuitive way to teach new behaviors to autonomous systems. In imitation learning, an expert demonstrates a new behavior to the robot; in turn, the robot learns the target task by observing and imitating the expert. This paradigm of *learning by demonstration* offers a compelling alternative to reinforcement learning and traditional control methods, which become increasingly challenging and computationally expensive as the desired behavior's complexity increases. In such cases, learning by demonstration provides an intuitive way for human experts to transfer their knowledge to robotic systems [7].

There exists a rich variety of imitation learning algorithms, each employing different assumptions, learning methods, and ways of incorporating expert demonstrations. At a high level, imitation learning algorithms can be categorized into behavioral cloning and inverse reinforcement learning methods, and model-free and model-based methods [3]. In this white paper, we present *ilpyt*, a research code base for imitation learning algorithm development, which unifies the different families of imitation learning algorithms under a single architecture. ilpyt, which was heavily inspired by the popular reinforcement learning research library, rlpyt [5], provides imitation learning algorithm implementations in Python using PyTorch [4] and provides compatibility with OpenAI Gym environments [6]. The library is structured for modular implementations of imitation learning and reinforcement learning algorithms which are easy to use, customize, and provide the basis for novel implementations. ilpyt is available at `https://github.com/mitre/ilpyt`.

Preprint. Under review.

### 1.1 Key Features and Algorithms

Key features of the ilpyt library include:

- Modular, extensible framework for training, evaluating, and testing imitation learning (and reinforcement learning) algorithms.
- Simple algorithm API which exposes train and test methods, allowing for quick library setup and use (a basic usage of the library requires less than ten lines of code to have a fully functioning train and test pipeline).
- A modular infrastructure for easy modification and reuse of existing components for novel algorithm implementations.
- Parallel and serialization modes, allowing for faster, optimized operations or serial operations for debugging.
- Compatibility with the OpenAI Gym environment interface for access to many existing benchmark learning environments, as well as the flexibility to create custom environments.

Implemented imitation learning algorithms include:

- Behavioral Cloning (BC) [8]
- Dataset Aggregation (DAgger) [9]
- Generative Adversarial Imitation Learning (GAIL) [10]
- Guided Cost Learning (GCL) [11]
- Apprenticeship Learning (AppL) [12]

In addition, ilpyt supports the implementation of reinforcement learning baselines. We provide the following reinforcement learning algorithm baselines:

- Deep Q-Networks (DQN) [1]
- Advantage Actor Critic (A2C) [13]
- Proximal Policy Optimization (PPO) [14]

The following OpenAI Gym environments are supported:

- Environments with observations spaces of type `Box` (1D and 3D)
- Environments with action spaces of type `Box` (1D), `Discrete`

This covers the vast majority of the 776 OpenAI Gym Environments. The library currently doesn't support less common observation space types such as `Discrete` and `Tuple` types, which accounts for 17 of the OpenAI Gym environments.

## 2 Benchmarking Performance

This section presents benchmark results (Table 1) of the ilpyt algorithm implementations against a subset of the OpenAI Gym Environments (LunarLander-v2, LunarLanderContinuous-v2, MountainCar-v0, MountainCarContinuous-v0, CartPole-v0). In all the imitation learning algorithm results, algorithms were trained with 100 successful expert demonstrations. The expert demonstrations were generated using a heuristic agent policy [15]. Reinforcement learning algorithms were trained for 10,000 iterations before stopping. All test results shown are averaged over 100 separate test trials.

The expert demonstrations used, as well as a model zoo of all the trained algorithms, are available at the ilpyt repository.

| | Environment (Threshold) | | | | |
|---|---|---|---|---|---|
| | **CartPole -v0** | **MountainCar -v0** | **MountainCar Continuous-v0** | **LunarLander -v2** | **LunarLander Continuous-v2** |
| **Threshold** | 200 | -110 | 90 | 200 | 200 |
| **Expert (Mean/Std)** | 200.00 / 0.00 | -98.71 / 7.83 | 93.36 / 0.05 | 268.09 / 21.18 | 283.83 / 17.70 |
| **BC (Mean/Std)** | 200.00 / 0.00 | -100.800 / 13.797 | 93.353 / 0.113 | 244.295 / 97.765 | 285.895 / 14.584 |
| **DAgger (Mean/Std)** | 200.00 / 0.00 | -102.36 / 15.38 | 93.20 / 0.17 | 230.15 / 122.604 | 285.85 / 14.61 |
| **GAIL (Mean/Std)** | 200.00 / 0.00 | -104.31 / 17.21 | 79.78 / 6.23 | 201.88 / 93.82 | 282.00 / 31.73 |
| **GCL (Mean/Std)** [2] | 200.00 / 0.00 | - [1] | - [1] | 212.321 / 119.933 | 255.414 / 76.917 |
| **AppL(Mean/Std)** | 200.00 / 0.00 | -108.60 / 22.843 | - [3,5] | - [4] | - [3,4,5] |
| **DQN (Mean/Std)** | - | - | - | 281.96 / 24.57 | - |
| **A2C (Mean/Std)** | - | | - | 201.26 / 62.52 | - |
| **PPO (Mean/Std)** | - | - | - | 249.72 / 75.05 | - |

Table 1: Performance of algorithms on selected OpenAI Gym environments over 100 trials. Expert episodes were recorded using heuristic agents. An environment is considered solved when the agent exceeds the threshold averaged over 100 episodes. Dashes indicate that no evaluation was done for this environment and algorithm combination.

[1] GCL does not perform well in the MountainCar environments; we suspect due to sparse rewards.
[2] GCL has noisy training, possibly due to its three-net architecture; for example, it could be the case that a network with a lower overall loss may perform worse in evaluation than a network with a higher total loss but lower loss for one particular network. Because of the noisiness during learning, we evaluate models saved at intermediate steps throughout training.
[3] At times, AppL works better when we invert the learned reward function during training, such as in the MountainCar-v0 environment. We suspect this is because of the negative reward structure of the environment. In the paper [12], the environments tested produce positive rewards only, so the effect of negative reward functions may require more research.
[4] AppL was unable to solve the LunarLander environments. We believe this is due to the linear formulation of the learned reward function, which is not sufficiently complex to capture non-linear relationships in our observations.
[5] We also found that AppL could not solve the continuous environments. Similar to the above, the paper [12] only performs experiments in environments with a discrete action space, so using apprenticeship learning to solve continuous action space environments will require more research.
[6] GAIL is extremely sensitive to its hyperparameters. Thorough hyperparameter searches must be conducted to find good configurations. The configuration for MountainCarContinuous-v0 is close, but requires further tuning.

## 3 Implementation and Usage Details

The following is a conceptual overview of ilpyt. For more details on practical code usage, please refer to the provided examples in the repository.

### 3.1 Conceptual Structure

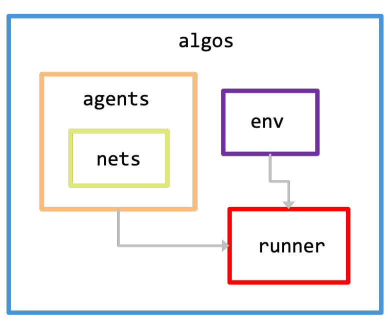


Figure 1: Conceptual structure diagram for the ilpyt library, illustrating the relationships between the different modular components of the library.

A conceptual structure and flow of the ilpyt framework is shown in Figure 1. At a high-level, the `algorithm` orchestrates the training and testing of our `agent` in a particular `environment`. During training or testing, a `runner` will execute the `agent` and `environment` in a loop to collect (state, action, reward, next state) transitions. The `agent` can then use this batch of transitions to update its `network` and move towards an optimal action policy.

Pseudo-code for the `algorithm` train loop is shown below (some steps omitted, for clarity):

```
def train(algo, num_episodes, batch_size, save_path):
    for ep in num_episodes:
        batch = algo.runner.generate_episode()
        loss_dict = algo.agent.update(batch)
        algo.log(ep, loss_dict)
        algo.agent.save(save_path)

def generate_episode():
    batch = []
    state = runner.env.reset()
    for t in batch_size:
        action = runner.agent.step(state)
        next_state, reward, done, info = runner.env.step(action)
        batch.add(state, next_state, reward, done, info)
    return batch
```

### 3.2 Code Structure

The following tree summarizes the ilpyt repository code structure.

```
ilpyt
├── docs
├── examples
├── ilpyt
│   ├── agents
│   ├── algos
│   ├── envs
│   ├── nets
│   ├── runners
│   └── utils
└── tests
```

`docs` Contains automatically generated documentation using pdoc3.

`examples` Contains examples of ilpyt library usage.

`tests` Contains unit tests for all implemented algorithms. Four representative OpenAI Gym environments were chosen to run the tests.

`ilpyt` Contains the main repository source code.

`algos` The algorithms are the highest level of the code. They coordinate the `agent` and `environment` via the `runner` for the main train and test functionality. The API requires `init`, `train`, and `test` methods.

`envs` ilpyt implements two wrappers for the OpenAI Gym environments: `SubProcVecEnv` and `DummyVecEnv`. They produce parallelized and serialized vectorized Gym environments for high-throughput training and were adapted from the OpenAI Baselines repository [16].

`agents` The agents coordinate the policy learning and execution. The API requires `init`, `step` and `update` methods, where `step` ingests a state and outputs an action, and `update` ingests a batch of transitions to update the agent policy weights.

`runner` The runner coordinates the interaction between the `agent` and the `environment`. It collects transitions (state, action, reward, next state) over specified intervals of time. We can have the runner generate a collection of transitions for us by calling `generate_batch` (specify number of steps) and `generate_episodes` (specify number of episodes).

`nets` The networks are extensions of PyTorch's `torch.nn.Module` class. The API requires `init`, `forward`, and `get_action` methods.

### 3.3 Custom Algorithms and Environments

To implement a new algorithm, the user simply has to extend the existing interfaces for each of the modular components. In most cases, the user will only have to extend the `algorithm` and `agent` interfaces, but further customization to the `net`, `runner`, etc. modules is available. The modular implementation of each component allows for easy code reuse; for example, the agent generator used in the GAIL algorithm can be easily swapped between PPOAgent, DQNAgent, or A2Cagent. In a similar way, new algorithm implementations can utilize existing implemented classes as building blocks or extend the class interfaces for more customization.

Adding a custom environment to ilpyt is as simple as extending the OpenAI Gym Environment interface and registering it within your local gym environment registry.

### 3.4 Simple Usage

In addition to providing a great deal of customization and flexibility, ilpyt also allows for quick experimentation. A minimal train and test snippet for an imitation learning algorithm takes less than 10 lines of code in ilpyt. Here we are training a behavioral cloning algorithm for 10,000 epochs before testing the best policy for 100 episodes.

```
import ilpyt
from ilpyt.agents.imitation_agent import ImitationAgent
from ilpyt.algos.bc import BC

env = ilpyt.envs.build_env(env_id='LunarLander-v2',  num_env=16)
net = ilpyt.nets.choose_net(env)
agent = ImitationAgent(net=net, lr=0.0001)

algo = BC(agent=agent, env=env)
algo.train(num_epochs=10000, expert_demos='demos/LunarLander-v2/demos.pkl')
algo.test(num_episodes=100)
```

## 4 Conclusion

We believe that ilpyt can accelerate the development of imitation learning algorithms by providing several baseline imitation learning implementations, a simple API for quick library spin-up and easy usage, and a modular framework for quick customization and spin-off for new implementations. We hope that the offering of algorithms will grow as the field matures.

**Acknowledgments**

The authors acknowledge the contributions of Jason Ice and Daniel Lee, who helped build the ilpyt test harness.

## 5 References

## References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[2] O. Vinyals, I. Babuschkin, J. Chung, M. Mathieu, M. Jaderberg, W. Czarnecki, A. Dudzik, A. Huang, P. Georgiev, R. Powell, T. Ewalds, D. Horgan, M. Kroiss, I. Danihelka, J. Agapiou, J. Oh, V. Dalibard, D. Choi, L. Sifre, Y. Sulsky, S. Vezhnevets, J. Molloy, T. Cai, D. Budden, T. Paine, C. Gulcehre, Z. Wang, T. Pfaff, T. Pohlen, D. Yogatama, J. Cohen, K. McKinney,

O. Smith, T. Schaul, T. Lillicrap, C. Apps, K. Kavukcuoglu, D. Hassabis, and D. Silver, “Alphastar: Mastering the real-time strategy game starcraft ii,” *Nature*, vol. 575, p. 350–354, 2019.

[3] T. Osa, J. Pajarinen, and G. Neumann, *An Algorithmic Perspective on Imitation Learning*. Hanover, MA, USA: Now Publishers Inc., 2018.

[4] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, “Pytorch: An imperative style, high-performance deep learning library,” in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[5] A. Stooke and P. Abbeel, “rlpyt: A research code base for deep reinforcement learning in pytorch,” 2019.

[6] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, “Openai gym,” 2016.

[7] S. Schaal, “Is imitation learning the route to humanoid robots?” *Trends in Cognitive Sciences*, vol. 3, no. 6, pp. 233 – 242, 1999. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1364661399013273

[8] D. A. Pomerleau, *ALVINN: An Autonomous Land Vehicle in a Neural Network*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1989, p. 305–313.

[9] S. Ross, G. J. Gordon, and J. A. Bagnell, “A reduction of imitation learning and structured prediction to no-regret online learning,” in *AISTATS*, 2011.

[10] J. Ho and S. Ermon, “Generative adversarial imitation learning,” 2016.

[11] C. Finn, S. Levine, and P. Abbeel, “Guided cost learning: Deep inverse optimal control via policy optimization,” in *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ser. ICML’16. JMLR.org, 2016, p. 49–58.

[12] P. Abbeel and A. Y. Ng, “Apprenticeship learning via inverse reinforcement learning,” in *Proceedings of the Twenty-First International Conference on Machine Learning*, ser. ICML ’04. New York, NY, USA: Association for Computing Machinery, 2004, p. 1. [Online]. Available: https://doi.org/10.1145/1015330.1015430

[13] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, “Asynchronous methods for deep reinforcement learning,” in *ICML*, 2016.

[14] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms.” *CoRR*, vol. abs/1707.06347, 2017. [Online]. Available: http://dblp.uni-trier.de/db/journals/corr/corr1707.html#SchulmanWDRK17

[15] Z. Xiao, *Reinforcement Learning: Theory and Python Implementation*.

[16] P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, Y. Wu, and P. Zhokhov, “Openai baselines,” https://github.com/openai/baselines, 2017.