

MimicKit: A Reinforcement Learning Framework for Motion Imitation and Control

XUE BIN PENG, Simon Fraser University and NVIDIA

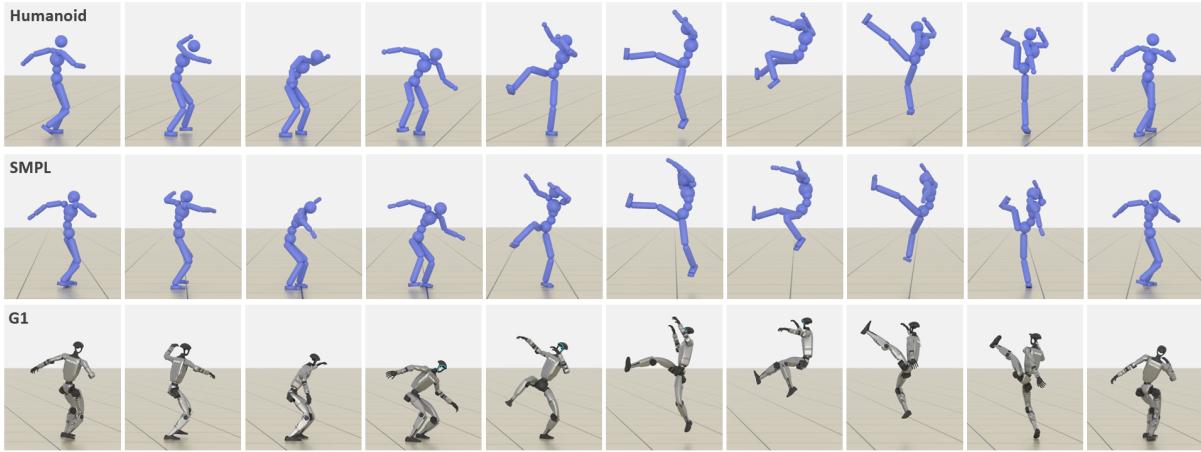


Fig. 1. MimicKit provides a suite motion imitation methods that can be used to train diverse simulated agents to perform highly dynamic and life-like motor skills. In this example, a variety of physically simulated humanoid characters are trained to perform a spinkick motion.

MimicKit is an open-source framework for training motion controllers using motion imitation and reinforcement learning. The codebase provides implementations of commonly-used motion-imitation techniques and RL algorithms. This framework is intended to support research and applications in computer graphics and robotics by providing a unified training framework, along with standardized environment, agent, and data structures. The codebase is designed to be modular and easily configurable, enabling convenient modification and extension to new characters and tasks. The open-source codebase is available at: <https://github.com/xbpeng/MimicKit>.

1 INTRODUCTION

Reinforcement-learning (RL) based motion imitation techniques have become a versatile and effective paradigm for constructing motion controllers that are able to produce agile, life-like behaviors for both simulated characters and robots in the real world. Although the many of the core ideas are conceptually simple, building effective motion imitation systems requires careful attention to numerous nuances and detailed design decisions that are often challenging to implement in practice. MimicKit is designed to lower the barrier for experimentation and reproducible research in this field by bringing together a suite of high-quality implementations of training methods and tools into a single unified and extensible framework.

2 BACKGROUND

In MimicKit, most models are trained using reinforcement learning, where an agent interacts with an environment according to a policy π in order to optimize a given objective [Sutton and Barto 2018]. At each time step t , the agent receives an observations \mathbf{o}_t of the environment, which provides partial information of the state \mathbf{s}_t of the underlying system. The agent responds by sampling an action from a policy $\mathbf{a}_t \sim \pi(\mathbf{a}_t | \mathbf{o}_t)$. The agent then

Author's address: Xue Bin Peng, xbpeng@sfu.ca, Simon Fraser University and NVIDIA.

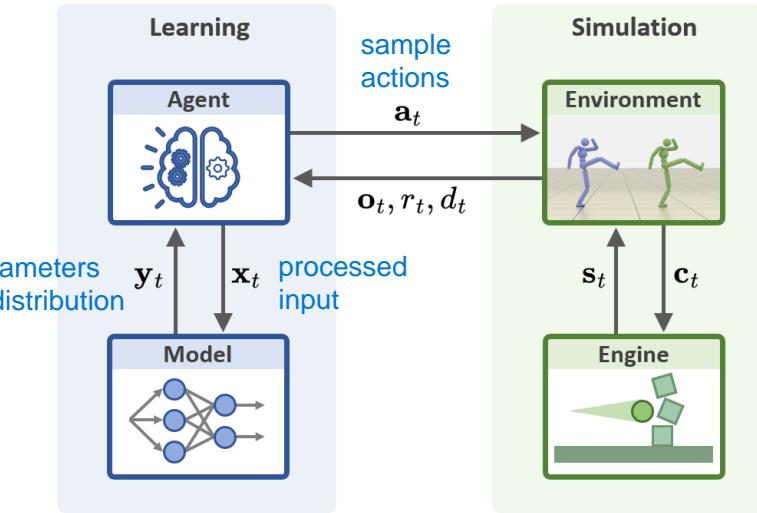


Fig. 2. Schematic overview of the MimicKit framework. The main components of the system are 1) the Agent, 2) the Model, 3) the Environment, and 4) the Engine. The learning algorithms are implemented primarily through the Agent and Model, while the Environment and Engine are responsible for simulating the desired task.

executes the action, which leads to a new state s_{t+1} , sampled according to the dynamics of the environment $s_{t+1} \sim p(s_{t+1}|s_t, a_t)$. The agent in turn receives a scalar reward $r_t = r(s_t, a_t, s_{t+1})$, and a new observation o_{t+1} of the next state s_{t+1} . The agent's objective is to learn a policy that maximizes its expected discounted return $J(\pi)$,

$$J(\pi) = \mathbb{E}_{p(\tau|\pi)} \left[\sum_{t=0}^{T-1} \gamma^t r_t \right], \quad \text{the further from the final step, (1) the bigger discount}$$

where $p(\tau|\pi)$ represents the likelihood of a trajectory $\tau = \{o_0, a_0, r_0, o_1, \dots, o_{T-1}, a_{T-1}, r_{T-1}, o_T\}$ under π . T denotes the time horizon of a trajectory, and $\gamma \in [0, 1]$ is a discount factor. Each trajectory corresponds to one *episode* of interactions between the agent and the environment.

3 SYSTEM OVERVIEW

A schematic overview of the MimicKit framework is provided in Figure 2. The core components of MimicKit consist of: 1) the Agent, 2) the Model, 3) the Environment, and 4) the Engine. The learning algorithms are implemented primarily through the Agent and the Model, while the Environment and Engine are responsible for simulating the desired task. These components are designed to be modular and composable, enabling users to combine different learning algorithms, model architectures, characters, tasks, and simulators. The simulations are implemented using vectorized environments, which can be massively parallelized using GPU simulators for high-throughput data collection during training. The environments and learning algorithms are designed to be character-agnostic, enabling the overall system to be easily configured to support characters with different morphologies, including humanoid and non-humanoid characters, such as quadrupedal robots.

3.1 Agent

The Agent class is responsible for implementing the learning algorithm and managing data recorded through interactions with the environment. Implementations for a suite of different agents are provided in [mimickit/learning/](#).

At each timestep t , the Agent receives an observation \mathbf{o}_t from the Environment. This observations is processed into an input \mathbf{x}_t for the Model, which can include pre-processing steps such as observation normalization. The Model is then queried with the processed input \mathbf{x}_t , which produces an output \mathbf{y}_t . The Model outputs \mathbf{y}_t may specify parameters of an action distribution, value function predictions, discriminator predictions, or other quantities required by the learning algorithm. The Agent then extracts an action \mathbf{a}_t from the model outputs, and applies \mathbf{a}_t to the Environment. The Environment in turn transitions to a new state \mathbf{s}_{t+1} and provides the Agent with the next observations \mathbf{o}_{t+1} , reward r_t , and a done flag d_t . The done flag d_t indicates if the current episode has been terminated.

In each iteration, the Agent repeats this interaction loop with the Environment until a designated number of timesteps has been collected. The data collected through these interactions are stored in an experience buffer implemented in `mimickit/learning/experience_buffer.py`. Once a sufficiently large batch of data has been collected, the Agent then uses the data to update its Model. The agent configuration files, located in `data/agents/`, are used to specify the type of Agent to use for training, as well as its associated hyperparameters.

3.2 Model

While the Agent implements the learning procedure, the Model is responsible for implementing the underlying neural network architecture used in the learning process. Each Agent is paired with a corresponding Model, located in `mimickit/learning/`. For an actor-critic algorithm, such as PPO [Schulman et al. 2017], the model may contain multiple neural networks, one for the policy (i.e. actor) and the value function (i.e. critic). For methods such as AMP, an additional network might be constructed for the discriminator. The Agent can query the Model's various networks with the appropriate input \mathbf{x}_t , and the Model returns the corresponding output \mathbf{y}_t . The Model's network architectures are specified through the `model` field in the agent configuration file.

3.3 Environment

The Environment implements the task-specific logic necessary to simulated a desired task. This class is used to define the interface through which the agent observes and interacts with its surrounding environment. At each timestep t , the Environment constructs the observation \mathbf{o}_t based on the state \mathbf{s}_t of the world determined by the Engine. The observation \mathbf{o}_t can contain proprioceptive information on the character's body, information on the configuration of surrounding objects, as well as task-specific information, such as the target locations and steering commands.

Upon receiving the action \mathbf{a}_t from the Agent, the Environment processes \mathbf{a}_t into a command \mathbf{c}_t . The command is then applied to the Engine to update the state of the underlying system, which can be modeled by a simulator or correspond to a real-world system. The environment update is performed through a step function:

```
obs, r, done, info = self._env.step(action)
```

The step function returns a new observation \mathbf{o}_{t+1} and a reward r_t for the state transition. Furthermore, an `info` dictionary can be used to store additional information from the Environment, such as auxiliary observations for a critic or discriminator. Finally, the done flag d_t indicates if the current episode has terminated. The done flag can assume 4 different values, as defined in `mimickit/envs/base_env.py`, depending on the conditions under which an episode was terminated. The different done flags include:

- `NULL`: The episode has not been terminated.
- `FAIL`: The episode terminated due to a failure, such as the character falling down. This flag can be used to apply a terminal penalty when calculating returns during training.

- **SUCC** : The episode terminated due to successfully completing a task, such as the character successfully reaching the target location. This flag can be used to apply a terminal bonus when calculating returns during training.
- **TIME** : The episode is terminated due to a time limit, but should in principle continue after the last timestep of the episode. In the event that a trajectory is truncated due to time, bootstrapping with a value function can be used to estimate future returns, as if the trajectory had continued after the last timestep. This enables the learning algorithms to emulate infinite-horizon MDPs given finite-length trajectories.

The configuration of the Environment is specified through environment configuration files located in `data/envs/`.

3.4 Engine

While the Environment implements the high-level logic for simulating a particular task, the low-level simulation of the world is delegated to an Engine. The Engine class, implemented in `mimickit/engines/engine.py`, provides a unified API that abstracts away the low-level details of how an Environment is simulated. Different Engines can be constructed for different physics engines and real-world robotic systems. This enables a specific task and environment to be instantiated through different underlying simulators and physical robots. MimicKit currently only supports IsaacGym [Makoviychuk et al. 2021], but additional Engines will be introduced in the future to support other physics simulators, as well as deployment on real robots.

At each timestep, the Engine receives the command c_t from the Environment, and returns an updated state s_{t+1} . The representation of c_t depends on the control modes that are supported by a specific Engine. For example, the IsaacGym Engine, implemented in `mimickit/engines/isaac_gym_engine.py` supports the following control modes:

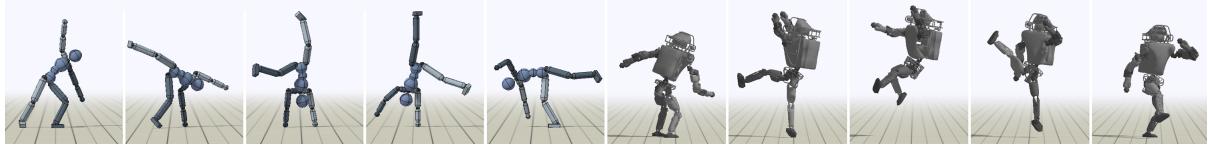
- **none** : Commands have no effect on the simulation. This mode can be useful for visualization and debugging.
- **pos** : Commands specify target rotations for PD controllers, which support both 1D revolute joints and 3D spherical joints.
- **vel** : Commands specify target velocities for each joint.
- **torque** : Commands directly specify torques for each joint.
- **pd_1d** : Commands specify target rotations for 1D revolute joints. This control mode can only be applied to morphologies that solely consist of 1D revolute joints, and does not support 3D spherical joints. This control mode is best suited for simulating robots that only contain 1D revolute joints.

The configuration of the engine can be specified through the `engine` field in the environment configuration file. The environment file can be used to specify the type of Engine to use for simulation, along with parameters such as the control model, control frequency, simulation frequency, etc.

4 METHODS

MimicKit provides a suite of motion imitation methods for training controllers. These methods offer different characteristics and trade-offs, and an appropriate method should be selected based on the requirements of a target application. Example argument files are provided in the arguments directory `args/` for using the various methods.

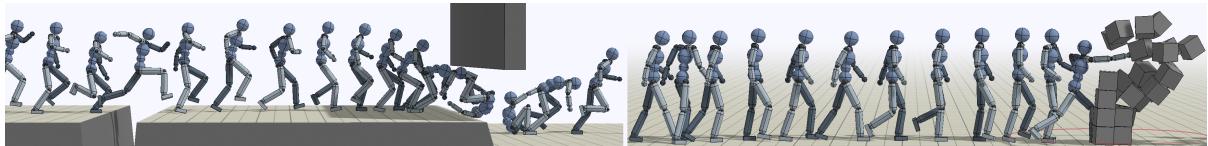
4.1 DeepMimic



DeepMimic is a simple RL-based motion tracking method [Peng et al. 2018], which trains a tracking controller to follow target reference motions. This method is very general and reliable, and has been successfully applied to train controllers for a wide range of behaviors. DeepMimic is often a good starting point before considering more sophisticated techniques, and can be a highly effective method for applications that require precise replication of a target reference motion. However, a key limitation of DeepMimic is that the motion tracking objective used during training often leads to inflexible policies that are restricted to closely following a given reference motion. This can limit the agent's ability to modify and adapt behaviors in the dataset as necessary to perform new tasks.

Example arguments for running DeepMimic are provided in `args/deepmimic_humanoid_ppo_args.txt`. The reference motion data used for training can be specified using the `motion_file` in the environment configuration file `data/envs/deepmimic_humanoid_env.yaml`.

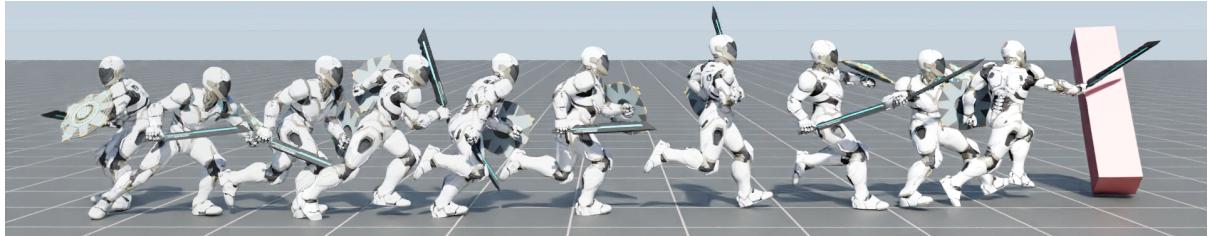
4.2 Adversarial Motion Priors (AMP)



Unlike DeepMimic, which trains a controllers to closely track a given reference motion, AMP is an adversarial distribution-matching method that aims to imitate the overall behavioral distribution (i.e. style) depicted in a dataset of motion clips [Peng et al. 2021], without explicitly tracking any specific motion clip. AMP provides more versatility than tracking-based methods, providing the agent with more flexibility to compose and adapt behaviors in the dataset in order to perform new tasks. However, a key drawback of distribution-matching methods, such as AMP, is that they are more prone to converging to local optima, especially for challenging, highly dynamics motions. Therefore AMP may struggle more to closely replicate challenging behaviors, compared to tracking-based methods, such as DeepMimic.

Example arguments for using AMP to imitate individual motion clips, without auxiliary tasks, are provided in `args/amp_humanoid_args.txt`. An example for training an AMP model with auxiliary tasks is provided in `args/amp_location_humanoid_args.txt`.

4.3 Adversarial Skill Embeddings (ASE)



ASE is an adversarial methods for training reusable generative controllers [Peng et al. 2022]. This method combines adversarial imitation learning with a mutual information-based skill discovery objective to learn latent skill embeddings. Points in the latent space can be mapped to diverse behaviors by the ASE controller. Once trained, the ASE controller can be reused to perform new tasks by training task-specific high-level controllers to select skills from the learned latent space. Example arguments for training ASE models are provided in `args/ase_humanoid_args.txt`.

4.4 Adversarial Differential Discriminator (ADD)



ADD is an adversarial motion tracking method that uses a differential discriminator to automatically learn adaptive motion tracking objectives [Zhang et al. 2025]. This method can mitigate the manual effort required to design and tune tracking reward functions for different characters and motions. Example arguments for training ADD models are provided in `args/add_humanoid_args.txt`.

Most of the methods in MimicKit are implemented using proximal policy optimization (PPO) as the underlying RL algorithm [Schulman et al. 2017]. PPO is currently the most commonly-used RL algorithm for motion control tasks, and can be effectively scaled with high-throughput GPU simulators. However, since PPO is an on-policy algorithm [Sutton and Barto 2018], it can be notoriously sample inefficient. Our framework provides an off-policy algorithm, Advantage-Weighted Regression (AWR) [Peng et al. 2019], as an alternative to PPO for settings that may require off-policy RL algorithms.

5 INSTRUCTIONS

In this section, we provide starter instructions for installing MimicKit, training models, testing models, and an overview of basic tools to assist in common workflows.

5.1 Installation

MimicKit can be installed by following the steps below:

- (1) MimicKit utilizes NVIDIA IsaacGym for high-performance physics simulation. IsaacGym installation instructions can be found at: <https://developer.nvidia.com/isaac-gym>.
- (2) Next install the dependencies from `requirements.txt`:

```
pip install -r requirements.txt
```

- (3) Download assets and motion data from the `data` repository, then extract the contents into the `data` directory `data/`.

After completing these steps, MimicKit should be ready for use.

5.2 Training

To train a model, a typical training command will be as follows:

```
python mimickit/run.py --mode train --num_envs 4096 \
--env_config data/envs/deepmimic_humanoid_env.yaml \
--agent_config data/agents/deepmimic_humanoid_ppo_agent.yaml \
--visualize true --log_file output/log.txt \
--out_model_file output/model.pt
```

The arguments consist of

- `--mode` selects either `train` or `test` mode.
- `--num_envs` specifies the number of parallel environments used for simulation.
- `--env_config` specifies the configuration file for the environment.
- `--agent_config` specifies configuration file for the agent.
- `--visualize` enables visualization. Rendering should be disabled for faster training.
- `--log_file` specifies the output log file, which will keep track of statistics during training.
- `--out_model_file` specifies the output model file, which contains the model parameters.
- `--logger` specifies the logger used to record training statistics. The options are TensorBoard `tb` or `wandb`.

Instead of specifying all arguments through the command line, arguments can also be loaded from an argument file `arg_file`:

```
python mimickit/run.py --arg_file args/deepmimic_humanoid_ppo_args.txt
```

The arguments in `arg_file` are treated the same as command line arguments. When using an argument file, additional command line arguments can be included to override the arguments in the `arg_file`. A library of arguments are provided in the arguments directory `args/` for training models and using various tools.

5.3 Distributed Training

The standard training command will train a model using a single process. To accelerate training, distributed training with multi-CPU or multi-GPU can be used with the following command:

```
python mimickit/run.py --arg_file args/deepmimic_humanoid_ppo_args.txt \
--num_workers 2 --device cuda:0
```

where `--num_workers` specifies the number of worker processes used to parallelize training. `--device` specifies the device used for training, which can be either `cpu` or `cuda:0`. When training with multiple GPUs, the

number of worker processes used to parallelize training must be less than or equal to the number of GPUs available on the system.

5.4 Testing

During training, the latest model parameters will be saved to a checkpoint `.pt` file, specified by `--out_model_file`. A typical command to test a trained model will be as follows:

```
python rl_forge/run.py --arg_file args/deepmimic_humanoid_ppo_args.txt \
--num_envs 4 \
--visualize true \
--mode test \
--model_file data/models/deepmimic_humanoid_spinkick_model.pt
```

`--mode test` specifies that the code should be run in testing mode. `--model_file` specifies the `.pt` file that contains the parameters of the trained model. Pretrained models are provided in `data/models/`, and the corresponding training log files are available in `data/logs/`.

5.5 Visualizing Training Logs

During training, When using the TensorBoard logger during training, a TensorBoard `events` file will be saved the same output directory as the log file. The log can be viewed with:

```
tensorboard --logdir=output/ --port=6006 --bind_all
```

In addition to visualizing training statistics with the runtime loggers, output log `.txt` file can also be visualized using the plotting script `tools/plot_log/plot_log.py`. Examples of learning curves generated by `plot_log.py` are shown in Figure 5.

6 MOTION DATA

Most of the methods implemented in MimicKit utilize motion data to guide the training process. Example motion clips are provided in `data/motions/`. The `motion_file` field in the environment configuration file can be used to specify the reference motion clip used for training and testing. In addition to imitating individual motion clips, `motion_file` can also specify a dataset file, located in `data/datasets/`, which will train a model to imitate a dataset containing multiple motion clips.

The `view_motion` environment can be used to visualize motion clips:

```
python mimickit/run.py --mode test --arg_file args/view_motion_humanoid_args.txt \
--visualize true
```

Motion clips are represented by the Motion class implemented in `mimickit/anim/motion.py`. Each motion clip is stored in a `.pkl` file. Each frame in a motion specifies the pose of the character according to [root position (3D), root rotation (3D), joint rotations], where 3D rotations are specified using 3D exponential maps [Grassia 1998]. Joint rotations are recorded in the order that the joints are specified in the `.xml` file (i.e. depth-first traversal of the kinematic tree). For example, in the case of the Humanoid character `data/assets/humanoid.xml`, each frame is represented as:

- (1) root position (3D)
- (2) root rotation (3D)
- (3) abdomen (3D)
- (4) neck (3D)
- (5) right_shoulder (3D)
- (6) right_elbow (1D)
- (7) left_shoulder (3D)
- (8) left_elbow (1D)
- (9) right_hip (3D)
- (10) right_knee (1D)
- (11) right_ankle (3D)
- (12) left_hip (3D)
- (13) left_knee (1D)
- (14) left_ankle (3D)

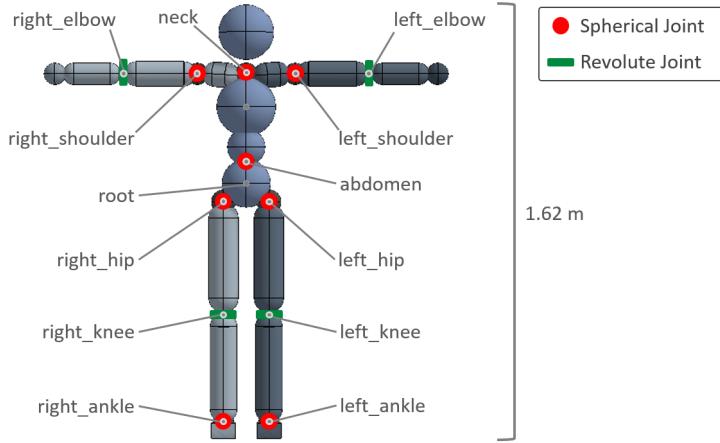


Fig. 3. Simulated Humanoid character.

The rotations of 3D joints are represented using 3D exponential maps, and the rotations of 1D joints are represented using 1D rotation angles.

7 EXPERIMENTS

To evaluate the framework’s effectiveness to reproduce diverse naturalistic motions, we apply the methods implemented in MimicKit on motion imitation tasks with a diverse suite of motions, ranging from common everyday behaviors, such as walking and running, to highly dynamic and athletic behaviors, such as acrobatics and martial arts. Our experiments assess both the quantitative tracking performance of the learned controllers and the qualitative fidelity of the resulting motions.

7.1 Motion Imitation

All experiments are conducted using the IsaacGym physics simulator. Each task involves a simulated humanoid character trained to imitate reference motion clips recorded via motion capture of live actors. We compare policies trained using three representative algorithms implemented within MimicKit: DeepMimic [Peng et al. 2018], AMP [Peng et al. 2021], and ADD [Zhang et al. 2025]. Separate policies are trained for each motion clip. In order to compare different methods under similar settings, we disable pose error termination used in Peng et al. [2018] during training, which terminates an episode if the character’s pose deviates significantly from the reference. Pose error termination is not applicable to distribution matching techniques such as AMP, where the policy is not synchronized with the reference motion. During training and evaluation, early termination is triggered only when the character makes undesired contact with the ground.

Snapshots of the behaviors learned by policies trained using various methods implemented in MimicKit are shown in Figure 4. Our framework is able to effectively train policies for a wide range of challenging and highly dynamics behaviors with a diverse cast of simulated characters, including a humanoid character modeled after the SMPL body model [Loper et al. 2015], a Unitree G1 humanoid robot, and a Unitree Go2 quadrupedal robot. Despite the significant differences in morphology among these character, the same underlying training framework can be applied with minimal modifications, highlighting the modularity and generality of our system.

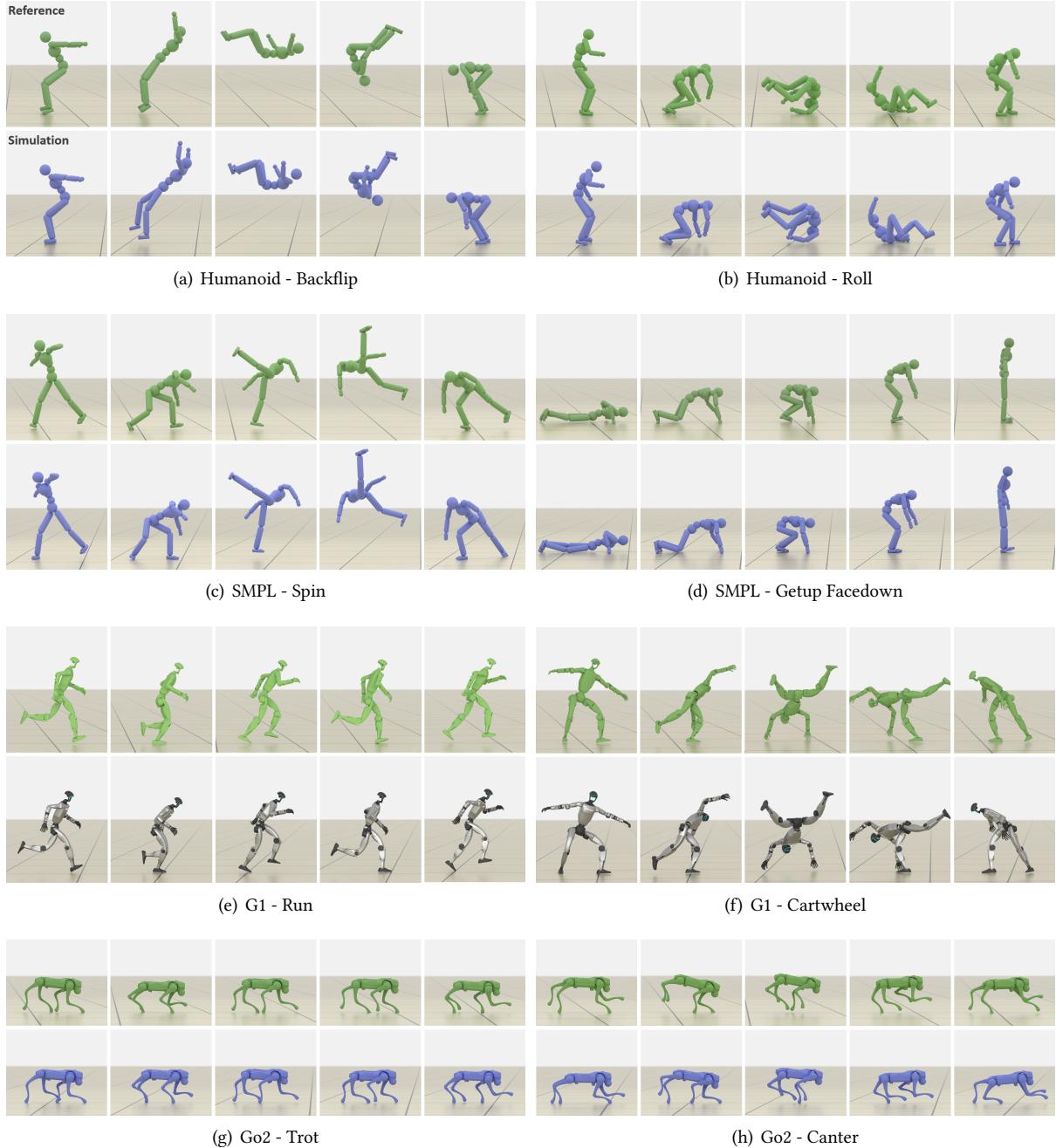


Fig. 4. Snapshots of physically simulated characters performing skills learned by imitating motion data recorded from real-life actors. The methods implemented in MimicKit can be applied to train policies for a diverse cast of simulated characters and skills.

Motion	Length	Position Tracking Error [m]			DoF Velocity Tracking Error [rad/s]		
		AMP	DeepMimic	ADD	AMP	DeepMimic	ADD
Run	0.80s	0.163 ± 0.008	0.013 ± 0.002	0.165 ± 0.017	2.811 ± 0.048	0.584 ± 0.054	0.478 ± 0.007
Jog	0.83s	0.120 ± 0.007	0.021 ± 0.000	0.024 ± 0.004	2.017 ± 0.052	0.575 ± 0.007	0.507 ± 0.010
Sideflip	2.44s	0.387 ± 0.011	0.138 ± 0.004	0.145 ± 0.006	2.276 ± 0.014	1.118 ± 0.034	1.350 ± 0.049
Crawl	2.93s	0.050 ± 0.006	0.027 ± 0.000	0.028 ± 0.002	0.646 ± 0.089	0.430 ± 0.006	0.283 ± 0.002
Roll	2.00s	0.141 ± 0.031	0.115 ± 0.132	0.152 ± 0.005	1.576 ± 0.318	0.994 ± 0.051	1.330 ± 0.101
Getup Face-down	3.03s	0.096 ± 0.018	0.023 ± 0.001	0.022 ± 0.001	0.838 ± 0.029	0.433 ± 0.008	0.325 ± 0.005
Spinkick	1.28s	0.064 ± 0.010	0.078 ± 0.062	0.025 ± 0.000	1.453 ± 0.327	1.222 ± 0.233	0.774 ± 0.007
Cartwheel	2.71s	0.076 ± 0.006	0.144 ± 0.153	0.017 ± 0.000	0.722 ± 0.020	0.659 ± 0.160	0.317 ± 0.002
Backflip	1.75s	0.267 ± 0.015	0.111 ± 0.054	0.062 ± 0.001	2.243 ± 0.113	1.103 ± 0.024	0.878 ± 0.013
Dance A	1.62s	0.065 ± 0.009	0.065 ± 0.029	0.028 ± 0.007	0.895 ± 0.108	0.830 ± 0.090	0.428 ± 0.014
Walk	0.96s	0.132 ± 0.021	0.009 ± 0.001	0.009 ± 0.001	1.394 ± 0.123	0.286 ± 0.005	0.213 ± 0.003

Table 1. Motion tracking performance of the Humanoid character trained using AMP, DeepMimic, and ADD. Position (Eq. 2) and DoF Velocity tracking errors are averaged across 5 models initialized with different random seeds. For each model, errors are calculated using 4096 test episodes. Motion tracking methods, such as DeepMimic and ADD, are able to more accurately reproduce a given reference motion compared to distribution-matching methods, such as AMP.

To evaluate the performance of each policy, we measure the position tracking error e_t^{pos} , and DoF velocity tracking error e_t^{vel} , which provides an indicator of motion smoothness. The position tracking error e_t^{pos} measures the difference in the global root position and relative joint positions between the simulated character and the reference motion:

$$e_t^{\text{pos}} = \frac{1}{N^{\text{joint}} + 1} \left(\sum_{j \in \text{joints}} \left\| (\hat{\mathbf{x}}_t^j - \hat{\mathbf{x}}_t^{\text{root}}) - (\mathbf{x}_t^j - \mathbf{x}_t^{\text{root}}) \right\|_2 + \left\| \hat{\mathbf{x}}_t^{\text{root}} - \mathbf{x}_t^{\text{root}} \right\|_2 \right). \quad (2)$$

Here, \mathbf{x}_t^j and $\hat{\mathbf{x}}_t^j$ represent the 3D Cartesian position of joint j from the simulated character and the reference motion, respectively. N^{joint} denotes the number of joints in the character. The DoF velocity tracking error measures the differences in local angular velocities of each joint between the simulated character and the reference motion:

$$e_t^{\text{vel}} = \frac{1}{N^{\text{joint}} + 1} \sum_{j \in \text{joints}} \left\| \hat{\mathbf{q}}_t^j - \dot{\mathbf{q}}_t^j \right\|_2, \quad (3)$$

where $\dot{\mathbf{q}}_t^j$ and $\hat{\mathbf{q}}_t^j$ represent the local angular velocity of joint j from the simulated character and the reference motion.

Table 1 summarizes performance of the various methods. Performance statistics for each method are calculated across 5 models initialized with different random seeds. AMP exhibits poor tracking performance, since the policies are trained using a general distribution-matching objective. However, qualitatively AMP can still be effective at reproducing the general behaviors of a reference motion, despite not precisely tracking the motion clip. Motion tracking methods, such as DeepMimic and ADD are able to accurately track a wide variety of reference motions. However, there are important distinctions in the consistency of the results across training runs. Since

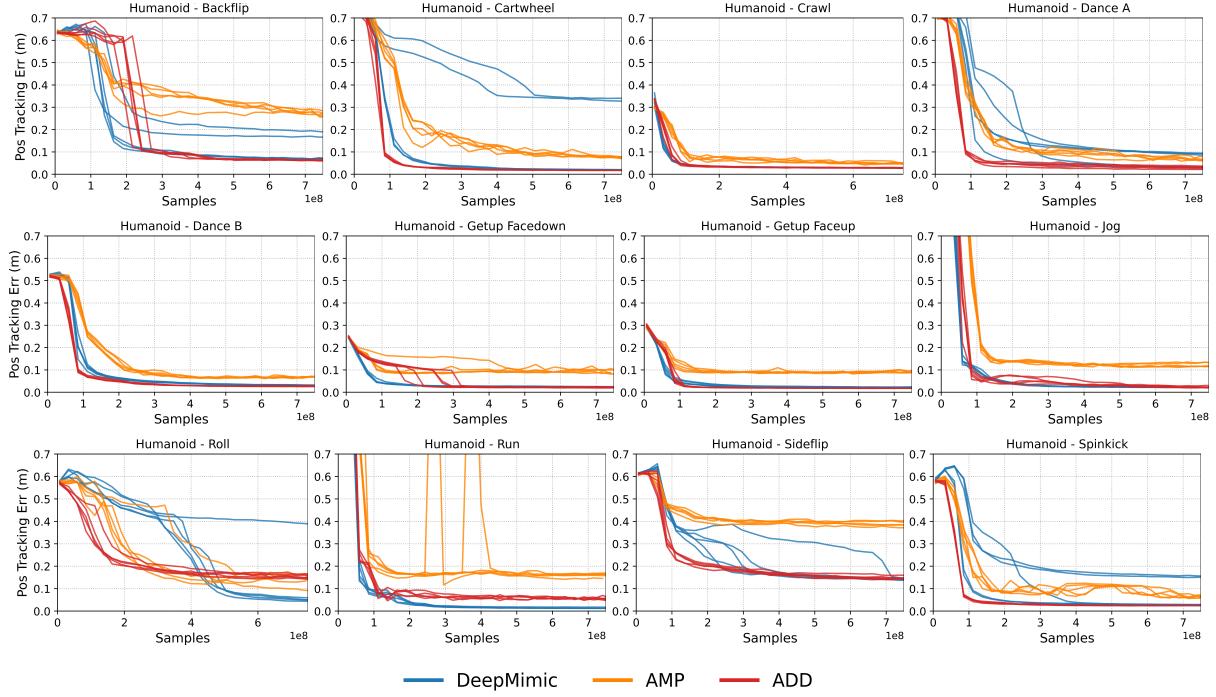


Fig. 5. Learning curves comparing the tracking performance with the simulated humanoid character trained with DeepMimic, AMP, and ADD. Five training runs initialized with different random seeds are shown for each method. In order to better compare methods under similar settings, policies are trained **without pose-error termination**. The standard configuration for tracking-based methods, such as DeepMimic and ADD, utilizes pose-error termination, which tends to produce better performance and more consistent results across training runs.

DeepMimic relies on a manually-designed reward function, it can be difficult to craft a general reward function that can effectively and consistently imitate a diverse variety of behaviors, in the absence of additional heuristics such as pose error termination. In contrast, ADD leverages a differential discriminator to automatically learn an adaptive reward function, which can lead to more consistent performance across diverse motions. However, we would like to note that when pose error termination is enabled during training, tracking accuracy and consistency across training runs generally improve substantially. Therefore, the default configuration for tracking-based methods generally incorporate pose error termination during training.

8 CONCLUSION

In this work, we introduced MimicKit, an open-source reinforcement learning framework for motion imitation and control. MimicKit unifies a suite of motion imitation methods for training motion controllers within a modular and extensible framework. We hope MimicKit will facilitate reproducible research in motor skill learning, and provide a convenient platform to accelerate progress in learning-based methods for motion control.

ACKNOWLEDGMENTS

We would like to thank all of the collaborators who have made valuable contributions to the development of MimicKit (ordered alphabetically): Yuxuan Mu, Yi Shi, Michael Xu, Dun Yang, and Ziyu Zhang. This work

was supported by NSERC (RGPIN-2015-04843), the National Research Council Canada (AI4D-166), and Sony Interactive Entertainment.

REFERENCES

- F. Sebastian Grassia. 1998. Practical Parameterization of Rotations Using the Exponential Map. *J. Graph. Tools* 3, 3 (March 1998), 29–48. <https://doi.org/10.1080/10867651.1998.10487493>
- Matthew Loper, Naureen Mahmood, Javier Romero, Gerard Pons-Moll, and Michael J. Black. 2015. SMPL: a skinned multi-person linear model. *ACM Trans. Graph.* 34, 6, Article 248 (Nov. 2015), 16 pages. <https://doi.org/10.1145/2816795.2818013>
- Viktor Makoviychuk, Lukasz Wawrzyniak, Yunrong Guo, Michelle Lu, Kier Storey, Miles Macklin, David Hoeller, Nikita Rudin, Arthur Allshire, Ankur Handa, and Gavriel State. 2021. Isaac Gym: High Performance GPU-Based Physics Simulation For Robot Learning. *CoRR* abs/2108.10470 (2021). arXiv:2108.10470 <https://arxiv.org/abs/2108.10470>
- Xue Bin Peng, Pieter Abbeel, Sergey Levine, and Michiel van de Panne. 2018. DeepMimic: Example-guided Deep Reinforcement Learning of Physics-based Character Skills. *ACM Trans. Graph.* 37, 4, Article 143 (July 2018), 14 pages. <https://doi.org/10.1145/3197517.3201311>
- Xue Bin Peng, Yunrong Guo, Lina Halper, Sergey Levine, and Sanja Fidler. 2022. ASE: Large-scale Reusable Adversarial Skill Embeddings for Physically Simulated Characters. *ACM Trans. Graph.* 41, 4, Article 94 (July 2022).
- Xue Bin Peng, Aviral Kumar, Grace Zhang, and Sergey Levine. 2019. Advantage-Weighted Regression: Simple and Scalable Off-Policy Reinforcement Learning. *CoRR* abs/1910.00177 (2019). arXiv:1910.00177 <https://arxiv.org/abs/1910.00177>
- Xue Bin Peng, Ze Ma, Pieter Abbeel, Sergey Levine, and Angjoo Kanazawa. 2021. AMP: Adversarial Motion Priors for Stylized Physics-Based Character Control. *ACM Trans. Graph.* 40, 4, Article 1 (July 2021), 15 pages. <https://doi.org/10.1145/3450626.3459670>
- John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. *CoRR* abs/1707.06347 (2017). arXiv:1707.06347 <http://arxiv.org/abs/1707.06347>
- Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA.
- Ziyu Zhang, Sergey Bashkirov, Dun Yang, Yi Shi, Michael Taylor, and Xue Bin Peng. 2025. Physics-Based Motion Imitation with Adversarial Differential Discriminators. In *SIGGRAPH Asia 2025 Conference Papers (SIGGRAPH Asia '25 Conference Papers)*.