My Tool Box

Comprehensive Research Toolkit and Environment Library

User Manual and Documentation

Dongming Wang
Research Toolkit Development

June 30, 2025

Contents

Chapter 1

Introduction

1.1 Overview

My Tool Box is a comprehensive research toolkit designed for reinforcement learning, neural network development, and environment simulation. This project consists of two main components:

- **Toolkit Project**: A unified Python package containing neural network architectures, plotting utilities, and research tools
- **Environment Library Project**: A collection of specialized reinforcement learning environments for various research domains

1.2 Project Structure

The project is organized as follows:

1.3 Key Features

1.3.1 Toolkit Project Features

- **Neural Networks**: Comprehensive collection of policy networks, value networks, and Q-networks
- **Architectures**: Support for MLP, CNN, RNN, Transformer, and hybrid architectures
- **Plotting Tools**: Advanced visualization utilities for research results
- **Parameter Management**: Flexible parameter configuration and management
- **Training Utilities**: Built-in training loops and evaluation tools

1.3.2 Environment Library Features

- **Multi-Agent Environments**: Collaborative and competitive scenarios
- **Physics Simulations**: Realistic physics-based environments

- **Communication Networks**: Wireless and message passing environments
- **Complex Systems**: Kuramoto oscillators and lattice-based systems
- **Modular Design**: Easy to extend and customize environments

1.4 Target Audience

This toolkit is designed for:

- **Researchers**: Working in reinforcement learning, multi-agent systems, and neural networks
- **Students**: Learning advanced machine learning concepts
- **Developers**: Building custom environments and algorithms
- **Engineers**: Implementing practical AI solutions

1.5 Prerequisites

Before using this toolkit, you should have:

- Python 3.7 or higher
- Basic knowledge of reinforcement learning concepts
- Familiarity with PyTorch and TensorFlow
- Understanding of neural network architectures

1.6 Quick Start

To get started quickly:

1. Install the toolkit: `pip install -e toolkit_project/`
2. Install the environment library: `pip install -e envlib_project/`
3. Run the example scripts in the `examples/` directory
4. Explore the documentation for detailed usage

1.7 Documentation Organization

This manual is organized as follows:

- **Chapters 1-2**: Introduction and installation
- **Chapters 3-5**: Toolkit components (neural networks, plotting, overview)

- **Chapters 6-10**: Environment library components
- **Chapters 11-13**: Examples, troubleshooting, and advanced usage
- **Appendices**: API reference, configuration, and benchmarks

1.8 Getting Help

If you encounter issues or need help:

- Check the troubleshooting chapter (Chapter 12)
- Review the example code in the `examples/` directory
- Examine the test files for usage patterns
- Consult the individual README files in each component

```
My_Tool_Box/
 toolkit_project/
    toolkit/
        neural_toolkit/     # Neural network components
        plotkit/           # Plotting utilities
        parakit/           # Parameter management
 envlib_project/
    env_lib/
        pistonball_env/    # Multi-agent physics environment
        kos_env/           # Kuramoto oscillator environment
        wireless_comm_env/ # Wireless communication environment
        ajlatt_env/        # Agent-based lattice environment
        linemsg_env/       # Linear message passing environment
 manual/                    # This documentation
```

Figure 1.1: Project directory structure

Chapter 2

Installation and Setup

2.1 System Requirements

2.1.1 Operating System

The toolkit supports the following operating systems:

- Linux (Ubuntu 18.04+, CentOS 7+)
- macOS (10.14+)
- Windows (10+)

2.1.2 Python Requirements

- Python 3.7 or higher
- pip package manager
- virtual environment (recommended)

2.1.3 Hardware Requirements

- **Minimum**: 4GB RAM, 2GB free disk space
- **Recommended**: 8GB+ RAM, 5GB+ free disk space
- **GPU**: Optional but recommended for neural network training

2.2 Environment Setup

2.2.1 Creating a Virtual Environment

It's recommended to use a virtual environment to avoid dependency conflicts:

[language=bash, caption=Creating virtual environment] Create virtual environment python -m venv my$_t oolbox_e nv$

Activate virtual environment On Linux/macOS: source my$_t oolbox_e nv/bin/activate$

On Windows: my$_{t}oolbox_{e}nv$

2.2.2 Installing Dependencies

Core Dependencies

The toolkit requires several core Python packages:

[language=bash, caption=Installing core dependencies] pip install torch>=1.9.0 pip install tensorflow>=2.6.0 pip install numpy>=1.20.0 pip install matplotlib>=3.5.0 pip install seaborn>=0.11.0 pip install scikit-learn>=1.0.0 pip install pandas>=1.3.0 pip install gym>=0.21.0

Optional Dependencies

For enhanced functionality, install these optional packages:

[language=bash, caption=Installing optional dependencies] For advanced plotting pip install plotly>=5.0.0 pip install bokeh>=2.4.0

For Jupyter notebook support pip install jupyter>=1.0.0 pip install ipywidgets>=7.6.0

For experiment tracking pip install wandb>=0.12.0 pip install tensorboard>=2.8.0

For parallel processing pip install joblib>=1.1.0 pip install multiprocessing-logging>=0.3.0

2.3 Installing the Toolkit

2.3.1 Installing Toolkit Project

Navigate to the toolkit project directory and install in development mode:

[language=bash, caption=Installing toolkit project] cd toolkit$_{p}rojectpipinstall - e.$

This installs the unified `toolkit` package containing:

- `neural_toolkit`: Neural network components
- `plotkit`: Plotting utilities
- `parakit`: Parameter management

2.3.2 Installing Environment Library

Install the environment library components:

[language=bash, caption=Installing environment library] cd envlib$_{p}rojectpipinstall - e.$

2.3.3 Installing Individual Environments

You can also install environments individually:

[language=bash, caption=Installing individual environments] Pistonball environment cd envlib$_{p}roject/env_{l}ib/p$
$e.$

Kuramoto oscillator environment cd envlib$_{p}roject/env_{l}ib/kos_{e}nvpipinstall - e.$

Wireless communication environment cd envlib$_project/env_lib/wireless_comm_envpipinstall - e.$

Agent-based lattice environment cd envlib$_project/env_lib/ajlatt_envpipinstall - e.$

Linear message passing environment cd envlib$_project/env_lib/linemsg_envpipinstall - e.$

2.4 Verification

2.4.1 Testing the Installation

Create a simple test script to verify the installation:

[language=python, caption=Test installation script] !/usr/bin/env python3

def test$_toolkit_imports() : """Testtoolkitimports"""try : importtoolkitprint("Toolkitimportedsuccessfully")$

from toolkit import neural$_toolkitprint("Neuraltoolkitimportedsuccessfully")$

from toolkit import plotkit print(" Plotkit imported successfully")

return True except ImportError as e: print(f" Import error: e") return False

def test$_envlib_imports() : """Testenvironmentlibraryimports"""try : importenv_libprint("Environmentlibrar$

Test individual environments from env$_libimportpistonball_envprint("Pistonballenvironmentimportedsuccessf$

from env$_libimportkos_envprint("Kuramotoenvironmentimportedsuccessfully")$

return True except ImportError as e: print(f" Import error: e") return False

if $_name_{=="}_{main_{":print("TestingMyToolBoxinstallation...")print()}}$

toolkit$_ok = test_toolkit_imports()print()envlib_ok = test_envlib_imports()$

if toolkit$_okandenvlib_ok : print("Allcomponentsinstalledsuccessfully!")else : print("Somecomponentsfailedt$

2.4.2 Running the Test

[language=bash, caption=Running installation test] python test$_installation.py$

2.5 Configuration

2.5.1 Environment Variables

Set these environment variables for optimal performance:

[language=bash, caption=Setting environment variables] For PyTorch export CUDA$_VISIBLE_DEVICES = 0UsefirstGPUexportOMP_NUM_THREADS = 4NumberofOpenMPthreads$

For TensorFlow export TF$_CPP_MIN_LOG_LEVEL = 2ReduceTensorFlowlogging$

For matplotlib (if using headless server) export MPLBACKEND=Agg

2.5.2 Configuration Files

Create configuration files for custom settings:

[language=python, caption=config.py example] Default configuration DEFAULT$_{C}ONFIG =$ $'neural_{t}oolkit' : 'device' :' cuda' if torch.cuda.is_{a}vailable() else' cpu',' default_{d}type' : torch.float32,' seed' : 42,' p$

2.6 Troubleshooting Installation

2.6.1 Common Issues

PyTorch Installation Issues

If you encounter PyTorch installation problems:

[language=bash, caption=Fixing PyTorch installation] Remove existing PyTorch pip uninstall torch torchvision torchaudio

Install PyTorch with CUDA support (if available) pip install torch torchvision torchaudio –index-url https://download.pytorch.org/whl/cu118

Or install CPU-only version pip install torch torchvision torchaudio –index-url https://download.pytorch.org/whl

TensorFlow Installation Issues

For TensorFlow problems:

[language=bash, caption=Fixing TensorFlow installation] Install TensorFlow with specific version pip install tensorflow==2.10.0

For GPU support pip install tensorflow-gpu==2.10.0

Permission Issues

If you encounter permission errors:

[language=bash, caption=Fixing permission issues] Use user installation pip install –user -e toolkit$_{p}roject/$

Or use sudo (not recommended) sudo pip install -e toolkit$_{p}roject/$

2.6.2 Getting Help

If installation issues persist:

1. Check the system requirements
2. Verify Python version compatibility
3. Ensure all dependencies are installed
4. Check for conflicting packages
5. Review the troubleshooting chapter

Chapter 3

Toolkit Overview

3.1 Introduction to the Toolkit

The `toolkit` package is a unified Python package that combines neural network components, plotting utilities, and parameter management tools into a single, cohesive research toolkit. It's designed to provide researchers and developers with a comprehensive set of tools for reinforcement learning and machine learning experiments.

3.2 Package Structure

The toolkit is organized into three main subpackages:

3.3 Neural Toolkit

The `neural_toolkit` subpackage provides comprehensive neural network components for reinforcement learning.

3.3.1 Network Architectures

Policy Networks

Policy networks implement different policy architectures:

- **MLPPolicyNetwork**: Multi-layer perceptron policy
- **CNPolicyNetwork**: Convolutional neural network policy
- **RNNPolicyNetwork**: Recurrent neural network policy
- **TransformerPolicyNetwork**: Transformer-based policy
- **HybridPolicyNetwork**: Combination of different architectures

Value Networks

Value networks for state-value and action-value estimation:

- **MLPValueNetwork**: MLP-based value network
- **CNValueNetwork**: CNN-based value network
- **RNNValueNetwork**: RNN-based value network
- **TransformerValueNetwork**: Transformer-based value network

Q-Networks

Q-networks for action-value function approximation:

- **MLPQNetwork**: MLP-based Q-network
- **CNQNetwork**: CNN-based Q-network
- **RNNQNetwork**: RNN-based Q-network
- **TransformerQNetwork**: Transformer-based Q-network
- **DuelingQNetwork**: Dueling architecture Q-network

3.3.2 State Encoders

State encoders transform raw observations into neural network inputs:

- **MLPEncoder**: Multi-layer perceptron encoder
- **CNNEncoder**: Convolutional neural network encoder
- **RNNEncoder**: Recurrent neural network encoder
- **TransformerEncoder**: Transformer-based encoder
- **HybridEncoder**: Combination of different encoders

3.3.3 Output Decoders

Output decoders transform network outputs into actions or values:

- **MLPDecoder**: Multi-layer perceptron decoder
- **CNNDecoder**: Convolutional neural network decoder
- **RNNDecoder**: Recurrent neural network decoder
- **TransformerDecoder**: Transformer-based decoder
- **VariationalDecoder**: Variational autoencoder decoder

3.3.4 Discrete Tools

Tools for tabular reinforcement learning methods:

- **QTable**: Q-learning table implementation
- **ValueTable**: Value iteration table
- **PolicyTable**: Policy table for discrete actions
- **DiscreteTools**: Utility functions for discrete RL

3.4 Plotkit

The `plotkit` subpackage provides advanced plotting utilities for research visualization.

3.4.1 Core Features

- **Training Plots**: Loss curves, reward plots, and convergence analysis
- **Performance Metrics**: Accuracy, precision, recall, and F1-score plots
- **Network Analysis**: Weight distributions, activation maps, and gradients
- **Environment Visualization**: State representations and action distributions
- **Comparison Plots**: Multi-algorithm and multi-environment comparisons

3.4.2 Plot Types

Training Visualization

[language=python, caption=Training plot example] from toolkit.plotkit import plot$_training_curves$

Plot training progress plot$_training_curves(rewards = rewards_history, losses = loss_history, title = "TrainingProgress", save_path = "training_curves.png")$

Performance Analysis

[language=python, caption=Performance analysis example] from toolkit.plotkit import plot$_performance_metrics$

Plot performance metrics plot$_performance_metrics(metrics = 'accuracy' : accuracy_history, 'precision' : precis$
$"PerformanceMetrics", save_path = "performance.png")$

Network Analysis

[language=python, caption=Network analysis example] from toolkit.plotkit import plot$_weight_distributions$

Plot weight distributions plot$_weight_distributions(model = policy_network, title = "WeightDistributions", save$
$"weights.png")$

3.5 Parakit

The `parakit` subpackage provides parameter management and configuration tools.

3.5.1 Features

- **Parameter Configuration**: JSON and YAML configuration files
- **Hyperparameter Management**: Grid search and random search utilities
- **Experiment Tracking**: Parameter logging and experiment management
- **Configuration Validation**: Parameter validation and type checking

3.6 Usage Examples

3.6.1 Basic Usage

[language=python, caption=Basic toolkit usage] import torch from toolkit.neural$_toolkitimportMLPPolicyNetw$

Create networks policy$_network = MLPPolicyNetwork(input_dim = 10, output_dim = 4, hidden_dims = [256, 256], activation =' relu')$

value$_network = MLPValueNetwork(input_dim = 10, hidden_dims = [256, 256], activation =' relu')$

Training loop rewards = [] losses = []

for episode in range(1000): Training code here episode$_reward = train_episode(policy_network, value_network)rew$

if episode loss = compute$_loss(policy_network, value_network)losses.append(loss)$

Plot results plot$_training_curves(rewards = rewards, losses = losses, title = "TrainingProgress", save_path = "training_results.png")$

3.6.2 Advanced Usage

[language=python, caption=Advanced toolkit usage] import torch from toolkit.neural$_toolkitimport(Transform$

Create complex network architecture encoder = CNNEncoder(input$_channels = 3, hidden_dims = [32, 64, 128], output_dim = 256)$

policy$_network = TransformerPolicyNetwork(input_dim = 256, output_dim = 10, d_model = 256, nhead = 8, num_layers = 6)$

value$_network = TransformerValueNetwork(input_dim = 256, d_model = 256, nhead = 8, num_layers = 6)$

Network analysis plot$_network_analysis(networks = 'encoder' : encoder,' policy' : policy_network,' value' : value_n$ $"NetworkArchitectureAnalysis", save_path = "network_analysis.png")$

3.7 Configuration

3.7.1 Default Configuration

The toolkit uses sensible defaults for most parameters:

[language=python, caption=Default configuration] DEFAULT$_CONFIG = 'device' :' cuda' if torch.cuda.is_available$

3.7.2 Custom Configuration

You can override default settings:

[language=python, caption=Custom configuration] from toolkit import set$_config$

Set custom configuration set$_config('device' :' cuda : 0',' dtype' : torch.float16,' activation' :' gelu',' hidden_dims$

3.8 Best Practices

3.8.1 Network Design

1. Start with simple architectures and gradually increase complexity
2. Use appropriate activation functions for your task
3. Consider using layer normalization for training stability
4. Monitor gradient flow and weight distributions
5. Use appropriate learning rates and optimizers

3.8.2 Training

1. Use appropriate batch sizes for your hardware
2. Monitor training progress with plots
3. Implement early stopping to prevent overfitting
4. Use learning rate scheduling for better convergence
5. Save model checkpoints regularly

3.8.3 Visualization

1. Plot training curves to monitor progress
2. Analyze network weights and activations
3. Compare different architectures and hyperparameters
4. Use appropriate plot types for different metrics
5. Save plots with descriptive filenames

3.9 Integration with Environments

The toolkit is designed to work seamlessly with the environment library:

[language=python, caption=Environment integration] import gym from toolkit.neural$_{t}oolkitimportMLPPolicy$

Create environment env = pistonball$_{e}nv.PistonballEnv()$

Create policy network policy$_{n}etwork = MLPPolicyNetwork(input_{d}im = env.observation_{s}pace.shape[0], outp$
$env.action_{s}pace.n, hidden_{d}ims = [256, 256])$

Training loop for episode in range(1000): obs = env.reset() done = False

while not done: action = policy$_{n}etwork.select_{a}ction(obs)obs, reward, done, info = env.step(action)$

```
toolkit/
 __init__.py                # Main package initialization
 neural_toolkit/            # Neural network components
    networks/              # Network architectures
    encoders/              # State encoders
    decoders/              # Output decoders
    discrete_tools/        # Tabular methods
    utils/                 # Utility functions
 plotkit/                   # Plotting utilities
    core.py                # Core plotting functions
    __main__.py            # Command-line interface
 parakit/                   # Parameter management
```

Figure 3.1: Toolkit package structure

Chapter 4

Neural Toolkit

4.1 Overview

The `neural_toolkit` is the core component of the toolkit, providing comprehensive neural network architectures and utilities for reinforcement learning. It includes policy networks, value networks, Q-networks, encoders, decoders, and discrete tools.

4.2 Network Architectures

4.2.1 Policy Networks

Policy networks map states to action probabilities or continuous actions.

MLPPolicyNetwork

Multi-layer perceptron policy network for discrete and continuous action spaces.

[language=python, caption=MLP Policy Network] from toolkit.neural$_{t}oolkitimportMLPPolicyNetwork$

Discrete action space policy$_{n}etwork = MLPPolicyNetwork(input_{d}im = 10, output_{d}im = 4, Numberofactionshidden_{d}ims = [256, 256], activation =' relu', dropout = 0.1, layer_{n}orm = False)$

Continuous action space policy$_{n}etwork = MLPPolicyNetwork(input_{d}im = 10, output_{d}im = 2, Actiondimensionshidden_{d}ims = [256, 256], activation =' relu', action_{t}ype =' continuous', action_{s}td = 1.0)$

CNPolicyNetwork

Convolutional neural network policy for image-based observations.

[language=python, caption=CNN Policy Network] from toolkit.neural$_{t}oolkitimportCNPolicyNetwork$

policy$_{n}etwork = CNPolicyNetwork(input_{c}hannels = 3, output_{d}im = 4, conv_{d}ims = [32, 64, 128], fc_{d}ims = [256, 256], kernel_{s}izes = [3, 3, 3], strides = [2, 2, 2], activation =' relu')$

RNNPolicyNetwork

Recurrent neural network policy for sequential decision making.

[language=python, caption=RNN Policy Network] from toolkit.neural$_{t}oolkitimportRNNPolicyNetwork$

policy$_{n}etwork = RNNPolicyNetwork(input_{d}im = 10, output_{d}im = 4, hidden_{d}im = 256, num_{l}ayers = 2, rnn_{t}ype =' lstm',' lstm' or' gru' fc_{d}ims = [256], activation =' relu')$

TransformerPolicyNetwork

Transformer-based policy network for complex sequential patterns.

[language=python, caption=Transformer Policy Network] from toolkit.neural$_{t}oolkitimportTransformerPolicy$

policy$_{n}etwork = TransformerPolicyNetwork(input_{d}im = 10, output_{d}im = 4, d_{m}odel = 256, nhead = 8, num_{l}ayers = 6, dim_{f}eedforward = 1024, dropout = 0.1, fc_{d}ims = [256], activation =' relu')$

4.2.2 Value Networks

Value networks estimate state values or state-action values.

MLPValueNetwork

Multi-layer perceptron value network.

[language=python, caption=MLP Value Network] from toolkit.neural$_{t}oolkitimportMLPValueNetwork$

value$_{n}etwork = MLPValueNetwork(input_{d}im = 10, hidden_{d}ims = [256, 256], activation =' relu', dropout = 0.1, layer_{n}orm = False)$

CNValueNetwork

Convolutional neural network value network.

[language=python, caption=CNN Value Network] from toolkit.neural$_{t}oolkitimportCNValueNetwork$

value$_{n}etwork = CNValueNetwork(input_{c}hannels = 3, conv_{d}ims = [32, 64, 128], fc_{d}ims = [256, 256], kernel_{s}izes = [3, 3, 3], strides = [2, 2, 2], activation =' relu')$

RNNValueNetwork

Recurrent neural network value network.

[language=python, caption=RNN Value Network] from toolkit.neural$_{t}oolkitimportRNNValueNetwork$

value$_{n}etwork = RNNValueNetwork(input_{d}im = 10, hidden_{d}im = 256, num_{l}ayers = 2, rnn_{t}ype =' lstm', fc_{d}ims = [256], activation =' relu')$

TransformerValueNetwork

Transformer-based value network.

[language=python, caption=Transformer Value Network] from toolkit.neural$_{t}oolkitimportTransformerValueN$

value$_n$$etwork = TransformerValueNetwork(input_dim = 10, d_model = 256, nhead = 8, num_layers = 6, dim_feedforward = 1024, dropout = 0.1, fc_dims = [256], activation =' relu')$

4.2.3 Q-Networks

Q-networks estimate action-value functions for Q-learning algorithms.

MLPQNetwork

Multi-layer perceptron Q-network.

[language=python, caption=MLP Q-Network] from toolkit.neural$_toolkitimportMLPQNetwork$

q$_n$$etwork = MLPQNetwork(input_dim = 10, output_dim = 4, Numberofactionshidden_dims = [256, 256], activation =' relu', dropout = 0.1)$

DuelingQNetwork

Dueling architecture Q-network with separate value and advantage streams.

[language=python, caption=Dueling Q-Network] from toolkit.neural$_toolkitimportDuelingQNetwork$

q$_n$$etwork = DuelingQNetwork(input_dim = 10, output_dim = 4, hidden_dims = [256, 256], value_hidden_dims = [256], advantage_hidden_dims = [256], activation =' relu')$

4.3 State Encoders

State encoders transform raw observations into neural network inputs.

4.3.1 MLPEncoder

Multi-layer perceptron encoder for vector observations.

[language=python, caption=MLP Encoder] from toolkit.neural$_toolkitimportMLPEncoder$

encoder = MLPEncoder(input$_d$$im = 100, output_dim = 256, hidden_dims = [512, 256], activation =' relu', dropout = 0.1, layer_norm = False)$

4.3.2 CNNEncoder

Convolutional neural network encoder for image observations.

[language=python, caption=CNN Encoder] from toolkit.neural$_toolkitimportCNNEncoder$

encoder = CNNEncoder(input$_c$$hannels = 3, output_dim = 256, conv_dims = [32, 64, 128, 256], fc_dims = [512], kernel_sizes = [3, 3, 3, 3], strides = [2, 2, 2, 2], activation =' relu')$

4.3.3 RNNEncoder

Recurrent neural network encoder for sequential observations.

[language=python, caption=RNN Encoder] from toolkit.neural$_toolkitimportRNNEncoder$

encoder = RNNEncoder(input$_{d}im = 10, output_{d}im = 256, hidden_{d}im = 256, num_{l}ayers = 2, rnn_{t}ype =' lstm', fc_{d}ims = [256], activation =' relu')$

4.3.4 TransformerEncoder

Transformer-based encoder for complex sequential patterns.

[language=python, caption=Transformer Encoder] from toolkit.neural$_{t}oolkitimportTransformerEncoder$

encoder = TransformerEncoder(input$_{d}im = 10, output_{d}im = 256, d_{m}odel = 256, nhead = 8, num_{l}ayers = 6, dim_{f}eedforward = 1024, dropout = 0.1, fc_{d}ims = [256], activation =' relu')$

4.4 Output Decoders

Output decoders transform network outputs into actions or values.

4.4.1 MLPDecoder

Multi-layer perceptron decoder.

[language=python, caption=MLP Decoder] from toolkit.neural$_{t}oolkitimportMLPDecoder$

decoder = MLPDecoder(latent$_{d}im = 256, output_{d}im = 10, hidden_{d}ims = [256, 128], activation =' relu', dropout = 0.1)$

4.4.2 CNNDecoder

Convolutional neural network decoder for image generation.

[language=python, caption=CNN Decoder] from toolkit.neural$_{t}oolkitimportCNNDecoder$

decoder = CNNDecoder(latent$_{d}im = 256, output_{c}hannels = 3, fc_{d}ims = [512, 256], conv_{d}ims = [256, 128, 64, 32], kernel_{s}izes = [3, 3, 3, 3], strides = [2, 2, 2, 2], initial_{s}ize = 4)$

4.4.3 RNNDecoder

Recurrent neural network decoder for sequential generation.

[language=python, caption=RNN Decoder] from toolkit.neural$_{t}oolkitimportRNNDecoder$

decoder = RNNDecoder(latent$_{d}im = 256, output_{d}im = 10, hidden_{d}im = 256, num_{l}ayers = 2, rnn_{t}ype =' lstm', max_{s}eq_{l}en = 100, fc_{d}ims = [256], activation =' relu')$

4.4.4 TransformerDecoder

Transformer-based decoder for complex sequential generation.

[language=python, caption=Transformer Decoder] from toolkit.neural$_{t}oolkitimportTransformerDecoder$

decoder = TransformerDecoder(latent$_{d}im = 256, output_{d}im = 10, d_{m}odel = 256, nhead = 8, num_{l}ayers = 6, dim_{f}eedforward = 1024, max_{s}eq_{l}en = 100, dropout = 0.1, fc_{d}ims = [256], activation =' relu')$

4.5 Discrete Tools

Discrete tools provide tabular reinforcement learning methods.

4.5.1 QTable

Q-learning table for discrete state-action spaces.

[language=python, caption=Q-Table Usage] from toolkit.neural$_t oolkitimportQTable$

Create Q-table q$_t able = QTable(state_s pace_s ize = 100, action_s pace_s ize = 4, initial_v alue = 0.0)$

Get Q-value q$_v alue = q_t able.get_v alue(state = 0, action = 1)$

Update Q-value q$_t able.update_v alue(state = 0, action = 1, value = 0.5, learning_r ate = 0.1)$

Get best action best$_a ction = q_t able.get_m ax_a ction(state = 0)$

Epsilon-greedy policy action = q$_t able.get_p olicy(state = 0, epsilon = 0.1)$

4.5.2 ValueTable

Value table for discrete state spaces.

[language=python, caption=Value Table Usage] from toolkit.neural$_t oolkitimportValueTable$

Create value table value$_t able = ValueTable(state_s pace_s ize = 100, initial_v alue = 0.0)$

Get value value = value$_t able.get_v alue(state = 0)$

Update value value$_t able.update_v alue(state = 0, value = 0.5, learning_r ate = 0.1)$

Get all values values = value$_t able.get_v alues()$

4.5.3 PolicyTable

Policy table for discrete state-action spaces.

[language=python, caption=Policy Table Usage] from toolkit.neural$_t oolkitimportPolicyTable$

Create policy table policy$_t able = PolicyTable(state_s pace_s ize = 100, action_s pace_s ize = 4)$

Get policy probability prob = policy$_t able.get_v alue(state = 0, action = 1)$

Update policy policy$_t able.update_v alue(state = 0, action = 1, value = 0.3, learning_r ate = 0.1)$

Sample action action = policy$_t able.get_p olicy(state = 0)$

Get policy probabilities probs = policy$_t able.get_p olicy_p robs(state = 0)$

4.5.4 DiscreteTools

Utility functions for discrete reinforcement learning.

[language=python, caption=Discrete Tools Usage] from toolkit.neural$_t oolkitimportDiscreteTools$

Q-learning update DiscreteTools.q$_l earning_u pdate(q_t able = q_t able, state = 0, action = 1, reward = 1.0, next_s tate = 2, gamma = 0.99, alpha = 0.1)$

SARSA update DiscreteTools.sarsa$_update(q_table = q_table, state = 0, action = 1, reward = 1.0, next_state = 2, next_action = 3, gamma = 0.99, alpha = 0.1)$

Expected SARSA update DiscreteTools.expected$_sarsa_update(q_table = q_table, state = 0, action = 1, reward = 1.0, next_state = 2, policy_table = policy_table, gamma = 0.99, alpha = 0.1)$

Epsilon-greedy policy action = DiscreteTools.epsilon$_greedy_policy(q_table = q_table, state = 0, epsilon = 0.1)$

Softmax policy action = DiscreteTools.softmax$_policy(q_table = q_table, state = 0, temperature = 1.0)$

4.6 Training Examples

4.6.1 PPO Training

[language=python, caption=PPO Training Example] import torch import torch.optim as optim from toolkit.neural$_toolkitimportMLPPolicyNetwork, MLPValueNetwork$

Create networks policy$_network = MLPPolicyNetwork(input_dim = 10, output_dim = 4, hidden_dims = [256, 256], activation =' relu')value_network = MLPValueNetwork(input_dim = 10, hidden_dims = [256, 256], activation =' relu')$

Optimizers policy$_optimizer = optim.Adam(policy_network.parameters(), lr = 0.001)value_optimizer = optim.Adam(value_network.parameters(), lr = 0.001)$

Training loop for episode in range(1000): Collect experience states, actions, rewards, next$_states, dones = collect_experience()$

Compute advantages advantages = compute$_advantages(states, rewards, value_network)$

PPO update for $_inrange(10) : MultipleepochsPolicylosslog_probs = policy_network.get_log_probs(states, actions)$ $torch.exp(log_probs - old_log_probs)surr1 = ratio * advantagessurr2 = torch.clamp(ratio, 0.8, 1.2) * advantagespolicy_loss = -torch.min(surr1, surr2).mean()$

Value loss value$_pred = value_network(states)value_loss = torch.nn.functional.mse_loss(value_pred, returns)$

Update networks policy$_optimizer.zero_grad()policy_loss.backward()policy_optimizer.step()$

value$_optimizer.zero_grad()value_loss.backward()value_optimizer.step()$

4.6.2 DQN Training

[language=python, caption=DQN Training Example] import torch import torch.optim as optim from toolkit.neural$_toolkitimportMLPQNetworkfromcollectionsimportdequeimportrandom$

Create Q-network q$_network = MLPQNetwork(input_dim = 10, output_dim = 4, hidden_dims = [256, 256], activation =' relu')target_network = MLPQNetwork(input_dim = 10, output_dim = 4, hidden_dims = [256, 256], activation =' relu')target_network.load_state_dict(q_network.state_dict())$

Replay buffer replay$_buffer = deque(maxlen = 10000)$

Optimizer optimizer = optim.Adam(q$_network.parameters(), lr = 0.001)$

Training loop for episode in range(1000): state = env.reset() done = False

while not done: Epsilon-greedy action selection if random.random() < epsilon: action = env.action$_space.sample()else : action = q_network.select_action(state)$

Take action $\text{next}_{s}tate, reward, done, _{=} env.step(action)$

Store experience $\text{replay}_{b}uffer.append((state, action, reward, next_{s}tate, done))$

Sample batch if $\text{len}(\text{replay}_{b}uffer) >= batch_{s}ize : batch = random.sample(replay_{b}uffer, batch_{s}ize)states, acti$
$zip(*batch)$

Convert to tensors states = torch.FloatTensor(states) actions = torch.LongTensor(actions) rewards = torch.FloatTensor(rewards) $\text{next}_{s}tates = torch.FloatTensor(next_{s}tates)dones = torch.BoolTensor(d$

Compute Q-values $\text{current}_{qv}alues = q_{n}etwork(states).gather(1, actions.unsqueeze(1))next_{qv}alues = target_{n}etwork(next_{s}tates).max(1)[0].detach()target_{qv}alues = rewards+(gamma*next_{qv}alues* dones)$

Compute loss loss = $\text{torch.nn.functional.mse}_{l}oss(current_{qv}alues.squeeze(), target_{qv}alues)$

Update network $\text{optimizer.zero}_{g}rad()loss.backward()optimizer.step()$

state = $\text{next}_{s}tate$

Update target network if episode $\text{target}_{n}etwork.load_{s}tate_{d}ict(q_{n}etwork.state_{d}ict())$

4.7 Network Utilities

4.7.1 Weight Initialization

[language=python, caption=Weight Initialization] from $\text{toolkit.neural}_{t}oolkitimportinitialize_{w}eights$

Initialize network weights $\text{initialize}_{w}eights(policy_{n}etwork, method =' xavier_{u}niform')initialize_{w}eights(value_{n}$
$orthogonal')$

4.7.2 Model Saving and Loading

[language=python, caption=Model Persistence] import torch

Save model torch.save('policy$_{s}tate_{d}ict' : policy_{n}etwork.state_{d}ict(),' value_{s}tate_{d}ict' : value_{n}etwork.state_{d}ict(),'$
$policy_{o}ptimizer.state_{d}ict(),' value_{o}ptimizer_{s}tate_{d}ict' : value_{o}ptimizer.state_{d}ict(),' episode' : episode,' reward_{h}$
$reward_{h}istory,' checkpoint.pth')$

Load model checkpoint = torch.load('checkpoint.pth') $\text{policy}_{n}etwork.load_{s}tate_{d}ict(checkpoint['policy_{s}tate_{d}ict'])$
$checkpoint['episode']reward_{h}istory = checkpoint['reward_{h}istory']$

4.8 Best Practices

4.8.1 Network Architecture

1. Start with simple architectures and gradually increase complexity
2. Use appropriate activation functions (ReLU for most cases, GELU for transformers)
3. Consider using layer normalization for training stability
4. Use dropout for regularization
5. Monitor gradient flow and weight distributions

4.8.2 Training

1. Use appropriate learning rates (typically 0.001 for Adam)
2. Implement learning rate scheduling
3. Use gradient clipping for stability
4. Monitor training progress with plots
5. Save checkpoints regularly

4.8.3 Hyperparameter Tuning

1. Use grid search or random search for hyperparameter optimization
2. Start with broad ranges and narrow down
3. Use cross-validation when possible
4. Monitor multiple metrics (reward, loss, convergence)
5. Document all hyperparameter settings

Chapter 5

Plotkit

5.1 Overview

The `plotkit` subpackage provides comprehensive plotting utilities for research visualization. It's designed to make it easy to create publication-quality plots for reinforcement learning experiments, neural network analysis, and performance evaluation.

5.2 Core Features

5.2.1 Training Visualization

Plotkit provides extensive tools for visualizing training progress:

- **Loss Curves**: Plot training and validation losses over time
- **Reward Plots**: Visualize episode rewards and cumulative rewards
- **Convergence Analysis**: Monitor algorithm convergence
- **Learning Curves**: Track learning progress across episodes

5.2.2 Performance Metrics

Comprehensive performance analysis tools:

- **Accuracy Metrics**: Plot accuracy, precision, recall, and F1-score
- **Policy Analysis**: Visualize policy distributions and action probabilities
- **Value Function Analysis**: Plot value function estimates
- **Q-Value Analysis**: Visualize Q-value distributions

5.2.3 Network Analysis

Deep analysis of neural network behavior:

- **Weight Distributions**: Analyze weight distributions across layers
- **Activation Maps**: Visualize activation patterns
- **Gradient Analysis**: Monitor gradient flow and vanishing/exploding gradients
- **Attention Maps**: Visualize attention patterns in transformer networks

5.2.4 Environment Visualization

Tools for understanding environment dynamics:

- **State Representations**: Visualize state spaces and transitions
- **Action Distributions**: Plot action selection patterns
- **Trajectory Visualization**: Show agent trajectories through environments
- **Multi-Agent Interactions**: Visualize interactions between agents

5.3 Basic Usage

5.3.1 Training Curves

Plot training progress with multiple metrics:

[language=python, caption=Training Curves Example] from toolkit.plotkit import plot$_training_curvesimportnu$

Generate sample data episodes = np.arange(1000) rewards = np.random.normal(0, 1, 1000).cumsum() losses = np.exp(-episodes / 200) + 0.1 * np.random.randn(1000)

Plot training curves plot$_training_curves(episodes = episodes, rewards = rewards, losses = losses, title = "TrainingProgress", xlabel = "Episode", ylabel = "Value", save_path = "training_curves.png",$ $True)$

5.3.2 Performance Metrics

Visualize multiple performance metrics:

[language=python, caption=Performance Metrics Example] from toolkit.plotkit import plot$_performance_metrics$

Generate sample metrics episodes = np.arange(1000) accuracy = 0.5 + 0.4 * (1 - np.exp(-episodes / 200)) precision = 0.6 + 0.3 * (1 - np.exp(-episodes / 150)) recall = 0.4 + 0.5 * (1 - np.exp(-episodes / 250))

Plot performance metrics plot$_performance_metrics(episodes = episodes, metrics = 'Accuracy' : accuracy,' Pre$ $"PerformanceMetrics", xlabel = "Episode", ylabel = "Score", save_path = "performance_metrics.png", show$ $True)$

5.3.3 Network Analysis

Analyze neural network weights and activations:

[language=python, caption=Network Analysis Example] from toolkit.plotkit import plot$_weight_distributionsfro$

Create a network network = MLPPolicyNetwork(input$_{d}im = 10, output_{d}im = 4, hidden_{d}ims =$ $[256, 256]$)

Plot weight distributions plot$_{w}eight_{d}istributions(model = network, title = "WeightDistributions", save_{p}ath =$ $"weight_{d}istributions.png", show_{p}lot = True, bins = 50)$

5.4 Advanced Plotting

5.4.1 Multi-Environment Comparison

Compare performance across different environments:

[language=python, caption=Multi-Environment Comparison] from toolkit.plotkit import plot$_{m}ulti_{e}nvironment$

Generate data for different environments episodes = np.arange(1000) env1$_{r}ewards = np.random.normal(0, 1, 10$ $np.random.normal(0.5, 1, 1000).cumsum()env3_{r}ewards = np.random.normal(-0.5, 1, 1000).cumsum()$

Plot comparison plot$_{m}ulti_{e}nvironment_{c}omparison(episodes = episodes, environment_{d}ata =$ $'Environment1' : env1_{r}ewards,' Environment2' : env2_{r}ewards,' Environment3' : env3_{r}ewards, title =$ $"Multi-EnvironmentPerformanceComparison", xlabel = "Episode", ylabel = "CumulativeReward", save_{p}$ $"multi_{e}nv_{c}omparison.png", show_{p}lot = True)$

5.4.2 Algorithm Comparison

Compare different algorithms on the same environment:

[language=python, caption=Algorithm Comparison] from toolkit.plotkit import plot$_{a}lgorithm_{c}omparisonimpo$

Generate data for different algorithms episodes = np.arange(1000) ppo$_{r}ewards = np.random.normal(0, 1, 1000)$ $np.random.normal(0.2, 1, 1000).cumsum()a2c_{r}ewards = np.random.normal(-0.1, 1, 1000).cumsum()$

Plot comparison plot$_{a}lgorithm_{c}omparison(episodes = episodes, algorithm_{d}ata = 'PPO' : ppo_{r}ewards,' DQN'$ $"AlgorithmPerformanceComparison", xlabel = "Episode", ylabel = "CumulativeReward", save_{p}ath =$ $"algorithm_{c}omparison.png", show_{p}lot = True)$

5.4.3 Hyperparameter Analysis

Analyze the effect of different hyperparameters:

[language=python, caption=Hyperparameter Analysis] from toolkit.plotkit import plot$_{h}yperparameter_{a}nalysis$

Generate data for different hyperparameters episodes = np.arange(1000) lr$_{0}01_{r}ewards = np.random.normal(0,$ $np.random.normal(0.3, 1, 1000).cumsum()lr_{1r}ewards = np.random.normal(-0.2, 1, 1000).cumsum()$

Plot analysis plot$_{h}yperparameter_{a}nalysis(episodes = episodes, hyperparameter_{d}ata = 'LR = 0.001' : lr_{0}01_{r}e$ $"LearningRate", title = "LearningRateAnalysis", xlabel = "Episode", ylabel = "CumulativeReward", save_{}$ $"hyperparameter_{a}nalysis.png", show_{p}lot = True)$

5.5 Specialized Plots

5.5.1 Policy Visualization

Visualize policy distributions and action probabilities:

[language=python, caption=Policy Visualization] from toolkit.plotkit import plot$_policy_analysisimportnumpyas$

Generate sample policy data states = np.arange(100) action$_probs = np.random.dirichlet([1, 1, 1, 1], size =$ 100)

Plot policy analysis plot$_policy_analysis(states = states, action_probabilities = action_probs, action_names =$ $['Up','Down','Left','Right'], title = "PolicyAnalysis", xlabel = "State", ylabel = "ActionProbability", save_p$ $"policy_analysis.png", show_plot = True)$

5.5.2 Value Function Visualization

Visualize value function estimates:

[language=python, caption=Value Function Visualization] from toolkit.plotkit import plot$_value_functionimport$

Generate sample value function data states = np.arange(100) values = np.sin(states / 10) + 0.1 * np.random.randn(100)

Plot value function plot$_value_function(states = states, values = values, title = "ValueFunction", xlabel =$ $"State", ylabel = "Value", save_path = "value_function.png", show_plot = True)$

5.5.3 Q-Value Analysis

Analyze Q-value distributions and action-value functions:

[language=python, caption=Q-Value Analysis] from toolkit.plotkit import plot$_{qv}alue_analysisimportnumpyasnp$

Generate sample Q-value data states = np.arange(50) actions = np.arange(4) q$_values = np.random.randn(50, 4)$

Plot Q-value analysis plot$_{qv}alue_analysis(states = states, actions = actions, q_values = q_values, action_names =$ $['Up','Down','Left','Right'], title = "Q - ValueAnalysis", xlabel = "State", ylabel = "Q -$ $Value", save_path = "q_value_analysis.png", show_plot = True)$

5.6 Command Line Interface

Plotkit provides a command-line interface for quick plotting:

5.6.1 Basic CLI Usage

[language=bash, caption=Basic CLI Usage] Plot training curves from CSV file python -m toolkit.plotkit –input training$_data.csv --typetraining_curves --outputtraining_plot.png$

Plot performance metrics python -m toolkit.plotkit –input metrics.csv –type performance –output performance$_plot.png$

Plot network analysis python -m toolkit.plotkit –input model.pth –type network$_analysis --outputnetwork_plot.png$

5.6.2 Advanced CLI Options

[language=bash, caption=Advanced CLI Options] Custom plot with specific options python -m toolkit.plotkit –input data.csv –type training$_curves --outputplot.png --title"CustomTitle" --xlabel"Episodes" --ylabel"Reward" --styleseaborn --figsize128 --dpi300$

5.7 Configuration

5.7.1 Plot Styles

Plotkit supports multiple plot styles:

[language=python, caption=Plot Style Configuration] from toolkit.plotkit import set$_plot_style$

Set plot style set$_plot_style('seaborn-v0_8')Modern, cleanstyleset_plot_style('ggplot')Rggplotstyleset_plot_style('class$

5.7.2 Color Schemes

Customize color schemes for different plot types:

[language=python, caption=Color Scheme Configuration] from toolkit.plotkit import set$_color_scheme$

Set color scheme set$_color_scheme('viridis')Perceptuallyuniformset_color_scheme('plasma')Highcontrastset_colo$

5.7.3 Figure Settings

Configure figure appearance:

[language=python, caption=Figure Configuration] from toolkit.plotkit import configure$_figure$

Configure figure settings configure$_figure(figsize = (12, 8), dpi = 300, style =' seaborn - v0_8', rc_params = 'font.size' : 12,' axes.titlesize' : 14,' axes.labelsize' : 12,' xtick.labelsize' : 10,' ytick.labelsize'$

5.8 Export Options

5.8.1 Image Formats

Plotkit supports multiple image formats:

[language=python, caption=Export Formats] from toolkit.plotkit import save$_plot$

Save in different formats save$_plot('plot.png', dpi = 300)High-resolutionPNGsave_plot('plot.pdf', dpi = 300)VectorPDFsave_plot('plot.svg', dpi = 300)ScalableSVGsave_plot('plot.jpg', dpi = 300)JPEGformatsave_p$ $300)TIFFformat$

5.8.2 Batch Export

Export multiple plots at once:

[language=python, caption=Batch Export] from toolkit.plotkit import batch$_export_plots$

Define plots to export plots$_config = ['type' :' training_curves',' data' : training_data,' output' :' training_curves.p$

Export all plots batch$_export_plots(plots_config, output_dir =' plots/')$

5.9 Integration with Other Tools

5.9.1 Integration with Neural Toolkit

Seamless integration with neural network components:

[language=python, caption=Neural Toolkit Integration] from toolkit.neural$_{t}oolkitimportMLPPolicyNetworkf$

Create and analyze network network = MLPPolicyNetwork(input$_{d}im = 10, output_{d}im = 4)plot_{n}etwork_{a}nalysis$ $network_{a}nalysis.png')$

5.9.2 Integration with Environment Library

Visualize environment interactions:

[language=python, caption=Environment Integration] from env$_{l}ibimportpistonball_{e}nvfromtoolkit.plotkitimpo$

Create environment env = pistonball$_{e}nv.PistonballEnv()$

Analyze environment plot$_{e}nvironment_{a}nalysis(env = env, num_{e}pisodes = 100, save_{p}ath =' environment_{a}nalysis.png')$

5.10 Best Practices

5.10.1 Plot Design

1. Use clear, descriptive titles and labels
2. Choose appropriate color schemes for accessibility
3. Use consistent formatting across related plots
4. Include error bars and confidence intervals when appropriate
5. Use log scales for data spanning multiple orders of magnitude

5.10.2 Data Visualization

1. Plot raw data alongside smoothed curves
2. Use multiple metrics to provide comprehensive analysis
3. Include baseline comparisons when possible
4. Use appropriate plot types for different data types
5. Consider the audience when choosing visualization complexity

5.10.3 Export and Sharing

1. Use high-resolution formats for publications
2. Include source data or code when sharing plots
3. Use consistent naming conventions for files
4. Document plot generation parameters
5. Consider file size for web sharing

5.11 Troubleshooting

5.11.1 Common Issues

Memory Issues

For large datasets, use data sampling:

[language=python, caption=Memory Management] Sample data for plotting sampled$_{e}pisodes = episodes[:: 10] Every 10 the pisode sampled_{r}ewards = rewards[:: 10]$

plot$_{t}raining_{c}urves(episodes = sampled_{e}pisodes, rewards = sampled_{r}ewards, save_{p}ath = "training_{c}urves.pn$

Style Issues

Reset plot style if encountering display problems:

[language=python, caption=Style Reset] import matplotlib.pyplot as plt

Reset to default style plt.style.use('default') plt.rcParams.update(plt.rcParamsDefault)

Export Issues

Ensure proper file permissions and paths:

[language=python, caption=Export Troubleshooting] import os

Create output directory if it doesn't exist os.makedirs('plots', exist$_{o}k = True$)

Use absolute paths for reliability save$_{p}ath = os.path.abspath('plots/training_{c}urves.png') plot_{t}raining_{c}urves(re$
$rewards, save_{p}ath = save_{p}ath)$

Chapter 6

Environment Library Overview

6.1 Introduction

The Environment Library (`env_lib`) is a comprehensive collection of specialized reinforcement learning environments designed for research in multi-agent systems, physics simulations, communication networks, and complex systems. Each environment is carefully designed to provide realistic, challenging scenarios for testing and developing reinforcement learning algorithms.

6.2 Library Structure

The environment library is organized into five main environments:

6.3 Environment Categories

6.3.1 Multi-Agent Environments

Environments designed for studying multi-agent interactions:

- **Pistonball Environment**: Collaborative physics-based environment where agents must work together to move a ball
- **Agent-based Lattice Environment**: Grid-based environment for studying spatial agent interactions

6.3.2 Physics Simulations

Realistic physics-based environments:

- **Pistonball Environment**: 2D physics simulation with multiple pistons and a ball
- **Kuramoto Oscillator Environment**: Complex systems simulation of coupled oscillators

6.3.3 Communication Networks

Environments for studying communication and coordination:

- **Wireless Communication Environment**: Realistic wireless network simulation
- **Linear Message Passing Environment**: Simplified message passing between agents

6.3.4 Complex Systems

Environments for studying emergent behavior and complex dynamics:

- **Kuramoto Oscillator Environment**: Study synchronization phenomena
- **Agent-based Lattice Environment**: Emergent behavior in spatial systems

6.4 Common Interface

All environments follow a consistent interface based on the OpenAI Gym standard:

[language=python, caption=Standard Environment Interface] import gym from env$_{l}ibimportpistonball_{e}nv$

Create environment env = pistonball$_{e}nv.PistonballEnv()$

Reset environment observation = env.reset()

Take action action = env.action$_{s}pace.sample()Randomactionobservation, reward, done, info = env.step(action)$

Get environment information print(f"Observation space: env.observation$_{s}pace")print(f"Actionspace : env.action_{s}pace")print(f"Numberofagents : env.num_{a}gents")$

6.5 Environment Features

6.5.1 Multi-Agent Support

Most environments support multiple agents with different interaction patterns:

[language=python, caption=Multi-Agent Usage] Get agent information num$_{a}gents = env.num_{a}gentsagent_{i}ds =$ $env.agent_{i}ds$

Multi-agent step actions = agent$_{i}d : env.action_{s}pace.sample()foragent_{i}dinagent_{i}dsobservations, rewards, don$ $env.step(actions)$

Check if episode is done episode$_{d}one = all(dones.values())$

6.5.2 Observation Spaces

Environments provide various observation types:

- **Vector Observations**: Numerical state representations
- **Image Observations**: Visual representations (RGB arrays)
- **Multi-Modal Observations**: Combinations of different observation types
- **Agent-Specific Observations**: Different observations for different agents

6.5.3 Action Spaces

Different action space types are supported:

- **Discrete Actions**: Finite set of possible actions
- **Continuous Actions**: Real-valued action vectors
- **Multi-Dimensional Actions**: Actions with multiple components
- **Hierarchical Actions**: Actions with different levels of abstraction

6.5.4 Reward Systems

Sophisticated reward mechanisms:

- **Individual Rewards**: Agent-specific reward signals
- **Global Rewards**: Environment-wide reward signals
- **Sparse Rewards**: Rewards only at specific events
- **Dense Rewards**: Continuous reward signals
- **Shaped Rewards**: Reward shaping for learning efficiency

6.6 Environment Configuration

6.6.1 Parameter Configuration

All environments support extensive parameter configuration:

[language=python, caption=Environment Configuration] Configure environment parameters config = 'num$_agents' : 4,' max_steps' : 1000,' reward_scale' : 1.0,' observation_type' :' vector',' render_mode' :' rgb_array'$

env = pistonball$_env.PistonballEnv(**config)$

6.6.2 Environment Wrappers

Use wrappers to modify environment behavior:

[language=python, caption=Environment Wrappers] from env$_libimportpistonball_envfromenv_lib.wrappersimpo$

Create base environment env = pistonball$_env.PistonballEnv()$

Add observation wrapper env = ObservationWrapper(env, observation$_type =' normalized')$

Add reward wrapper env = RewardWrapper(env, reward$_scale = 0.1)$

6.7 Integration with Toolkit

6.7.1 Neural Network Integration

Seamless integration with neural toolkit components:

[language=python, caption=Neural Network Integration] from toolkit.neural$_{t}oolkitimportMLPPolicyNetwork$

Create environment env = pistonball$_{e}nv.PistonballEnv()$

Create policy network policy$_{n}etwork = MLPPolicyNetwork(input_{d}im = env.observation_{s}pace.shape[0], outp$
$env.action_{s}pace.n, hidden_{d}ims = [256, 256])$

Training loop for episode in range(1000): obs = env.reset() done = False

while not done: action = policy$_{n}etwork.select_{a}ction(obs)obs, reward, done, info = env.step(action)$

6.7.2 Plotting Integration

Visualize environment behavior with plotkit:

[language=python, caption=Plotting Integration] from toolkit.plotkit import plot$_{e}nvironment_{a}nalysisfromen$

Create environment env = pistonball$_{e}nv.PistonballEnv()$

Analyze environment plot$_{e}nvironment_{a}nalysis(env = env, num_{e}pisodes = 100, save_{p}ath ='$
$environment_{a}nalysis.png')$

6.8 Performance Considerations

6.8.1 Computational Efficiency

- **Vectorized Environments**: Support for parallel environment execution
- **Optimized Physics**: Efficient physics calculations
- **Memory Management**: Careful memory usage for large-scale experiments
- **GPU Acceleration**: GPU support for physics simulations where applicable

6.8.2 Scalability

- **Variable Agent Counts**: Support for different numbers of agents
- **Configurable Complexity**: Adjustable environment complexity
- **Parallel Execution**: Support for multiple environment instances
- **Distributed Training**: Compatible with distributed training frameworks

6.9 Best Practices

6.9.1 Environment Selection

1. Choose environments appropriate for your research question

2. Start with simpler environments and gradually increase complexity
3. Consider the computational requirements of each environment
4. Match environment characteristics to your algorithm capabilities

6.9.2 Configuration

1. Use appropriate observation and action spaces for your algorithms
2. Configure reward scales to match your learning rates
3. Set reasonable episode lengths for your training setup
4. Use environment wrappers to standardize interfaces

6.9.3 Training

1. Monitor environment performance and agent behavior
2. Use appropriate exploration strategies for each environment
3. Consider the multi-agent nature when designing algorithms
4. Validate results across multiple environment seeds

6.10 Development and Extension

6.10.1 Custom Environments

Guidelines for creating custom environments:

[language=python, caption=Custom Environment Template] import gym from gym import spaces import numpy as np

class CustomEnv(gym.Env): def $_{init}{}_{(self,**kwargs):super().}{}_{init}{}_{()}$

Define observation and action spaces self.observation$_space = spaces.Box(low = -np.inf, high = np.inf, shape = (10,), dtype = np.float32) self.action_space = spaces.Discrete(4)$

Initialize environment state self.reset()

def reset(self): Reset environment to initial state self.state = np.zeros(10) return self.state

def step(self, action): Execute action and return (observation, reward, done, info) Implementation here pass

def render(self, mode='human'): Render environment (optional) pass

6.10.2 Environment Testing

Comprehensive testing framework:

[language=python, caption=Environment Testing] def test$_environment() : """Testenvironmentfunctionality"$
$pistonball_env.PistonballEnv()$

Test reset obs = env.reset() assert obs.shape == env.observation$_{s}pace.shape$

Test step action = env.action$_{s}pace.sample()obs, reward, done, info = env.step(action)assertobs.shape == env.observation_{s}pace.shapeassertisinstance(reward, (int, float))assertisinstance(done, bool)$

Test episode completion step$_{c}ount = 0whilenotdoneandstep_{c}ount < 1000 : action = env.action_{s}pace.sample()o$ $env.step(action)step_{c}ount+ = 1$

print("Environment test passed!")

6.11 Troubleshooting

6.11.1 Common Issues

Import Errors

If you encounter import errors:

[language=bash, caption=Fixing Import Issues] Install environment in development mode cd envlib$_{p}roject/env_{l}ib/pistonball_{e}nvpipinstall - e.$

Check installation python -c "import pistonball$_{e}nv; print('Importsuccessful')$"

Performance Issues

For performance problems:

[language=python, caption=Performance Optimization] Use vectorized environments from env$_{l}ibimportpistonb$ $pistonball_{e}nv.PistonballEnv(num_{e}nvs = 4)$

Disable rendering during training env = pistonball$_{e}nv.PistonballEnv(render_{m}ode = None)$

Use appropriate observation types env = pistonball$_{e}nv.PistonballEnv(observation_{t}ype =' vector')$

Memory Issues

For memory problems:

[language=python, caption=Memory Management] Limit episode length env = pistonball$_{e}nv.PistonballEnv(m$ 500)

Use smaller observation spaces env = pistonball$_{e}nv.PistonballEnv(observation_{t}ype =' minimal')$

Clear environment state env.close() del env

6.12 Future Development

6.12.1 Planned Features

- **Additional Environments**: More specialized environments for specific research domains
- **Enhanced Physics**: More realistic physics simulations
- **Better Visualization**: Improved rendering and visualization tools
- **Performance Optimization**: Further optimization for large-scale experiments

6.12.2 Contributing

Guidelines for contributing to the environment library:

1. Follow the existing code style and conventions
2. Include comprehensive tests for new environments
3. Document all parameters and interfaces
4. Provide example usage and tutorials
5. Ensure compatibility with the toolkit components

```
env_lib/
 __init__.py            # Main package initialization
 pistonball_env/        # Multi-agent physics environment
 kos_env/               # Kuramoto oscillator environment
 wireless_comm_env/     # Wireless communication environment
 ajlatt_env/            # Agent-based lattice environment
 linemsg_env/           # Linear message passing environment
```

Figure 6.1: Environment library structure

Chapter 7

Pistonball Environment

7.1 Overview

The Pistonball Environment is a multi-agent physics-based environment where agents control pistons to collaboratively move a ball. This environment is designed to study cooperative behavior, coordination, and emergent strategies in multi-agent systems.

7.2 Environment Description

7.2.1 Physics Simulation

The environment simulates a 2D physics world with:

- **Multiple Pistons**: Agents control individual pistons that can move up and down
- **Ball Physics**: A ball that bounces off pistons and walls
- **Gravity**: Realistic gravitational effects
- **Collision Detection**: Accurate collision handling between ball and pistons

7.2.2 Objective

The goal is to move the ball from one side of the environment to the other by coordinating piston movements. Agents must learn to:

- **Coordinate Timing**: Move pistons at the right moments
- **Maintain Ball Momentum**: Keep the ball moving in the desired direction
- **Avoid Obstacles**: Prevent the ball from getting stuck
- **Work Together**: Coordinate with other agents for optimal performance

7.3 Installation

7.3.1 Basic Installation

[language=bash, caption=Basic Installation] cd envlib$_{project/env_lib/pistonball_envpipinstall-e.}$

7.3.2 Dependencies

The environment requires:

- **PyGame**: For rendering and physics simulation
- **NumPy**: For numerical computations
- **Gym**: For the environment interface

7.4 Basic Usage

7.4.1 Creating the Environment

[language=python, caption=Basic Environment Creation] from env$_libimportpistonball_env$

Create environment with default settings env = pistonball$_env.PistonballEnv()$

Get environment information print(f"Number of agents: env.num$_agents")print(f"Observationspace :$ $env.observation_space")print(f"Actionspace : env.action_space")$

7.4.2 Running a Simple Episode

[language=python, caption=Simple Episode] import numpy as np

Reset environment observations = env.reset()

Run episode done = False total$_reward = 0$

while not done: Random actions for all agents actions = agent$_id : env.action_space.sample()foragent_idinenv.ag$

Take step observations, rewards, dones, infos = env.step(actions)

Accumulate reward total$_reward+ = sum(rewards.values())$

Check if episode is done done = all(dones.values())

print(f"Episode completed with total reward: total$_reward")$

7.5 Configuration Options

7.5.1 Environment Parameters

[language=python, caption=Environment Configuration] Configure environment config = 'num$_agents' :$ $4, Numberofpistons/agents'max_steps' : 1000, Maximumstepsperepisode'ball_speed' : 5.0, Initialballspeed'gra$ $0.5, Gravitystrength'piston_speed' : 2.0, Pistonmovementspeed'reward_scale' : 1.0, Rewardscalingfactor'obser$ $vector', Observationtype'render_mode' :' rgb_array'Renderingmode$

env = pistonball$_env.PistonballEnv(**config)$

7.5.2 Parameter Descriptions

7.6 Observation Spaces

7.6.1 Vector Observations

Vector observations provide numerical state information:

[language=python, caption=Vector Observations] Vector observation space observation$_{s}pace = spaces.Box(low = -np.inf, high = np.inf, shape = (observation_{d}im,), dtype = np.float32)$

Observation components observation = [ball$_x, ball_y, Ballpositionball_vx, ball_vy, Ballvelocitypiston1_y, piston1_vy,$

7.6.2 Image Observations

Image observations provide visual representations:

[language=python, caption=Image Observations] Image observation space observation$_{s}pace = spaces.Box(low = 0, high = 255, shape = (height, width, 3), dtype = np.uint8)$

RGB image of the environment observation = env.render(mode='rgb$_{a}rray')$

7.6.3 Multi-Modal Observations

Combination of vector and image observations:

[language=python, caption=Multi-Modal Observations] Multi-modal observation space observation$_{s}pace = spaces.Dict('vector' : spaces.Box(low = -np.inf, high = np.inf, shape = (vector_{d}im,)), 'image' : spaces.Box$

7.7 Action Spaces

7.7.1 Discrete Actions

Simple discrete action space:

[language=python, caption=Discrete Actions] Discrete action space action$_{s}pace = spaces.Discrete(3)$

Actions: 0 = no movement, 1 = move up, 2 = move down actions = 'agent$'_0 : 1, Moveup'agent'_1 : 2, Movedown'agent'_2 : 0, Nomovement'agent'_3 : 1Moveup$

7.7.2 Continuous Actions

Continuous action space for smoother control:

[language=python, caption=Continuous Actions] Continuous action space action$_{s}pace = spaces.Box(low = -1.0, high = 1.0, shape = (1,), dtype = np.float32)$

Actions: -1 = move down, 0 = no movement, 1 = move up actions = 'agent$'_0 : [0.5], Moveup'agent'_1 : [-0.3], Movedownslightly'agent'_2 : [0.0], Nomovement'agent'_3 : [1.0]Moveupfully$

7.8 Reward System

7.8.1 Reward Components

The reward system includes multiple components:

- **Ball Progress**: Reward for ball moving toward the goal
- **Collision Reward**: Reward for successful ball-piston collisions
- **Coordination Bonus**: Bonus for coordinated movements
- **Time Penalty**: Small penalty per step to encourage efficiency

7.8.2 Reward Calculation

[language=python, caption=Reward Calculation] def calculate$_reward(self, ball_progress, collisions, coordinatio$
$"""Calculaterewardforthecurrentstate"""$

Base reward from ball progress progress$_reward = ball_progress * self.reward_scale$

Collision reward collision$_reward = collisions * 0.1$

Coordination bonus coordination$_bonus = coordination * 0.05$

Time penalty time$_penalty = -0.001$

Total reward total$_reward = progress_reward+collision_reward+coordination_bonus+time_penalty$

return total$_reward$

7.9 Advanced Features

7.9.1 Multi-Agent Coordination

The environment supports sophisticated multi-agent interactions:

[language=python, caption=Multi-Agent Coordination] Get agent information agent$_ids = env.agent_idsnum_age$
$env.num_agents$

Agent-specific observations agent$_observations = foragent_idinagent_ids : agent_observations[agent_id] =$
$observations[agent_id]$

Coordinated actions def coordinated$_policy(observations) : """Policythatconsidersotheragents""" actions =$

for agent$_idinagent_ids : Considerotheragents'positionsother_positions = [observations[other_id][:$
$2]forother_idinagent_idsifother_id! = agent_id]$

Make decision based on coordination actions[agent$_id] = decide_action(observations[agent_id], other_positions)$

return actions

7.9.2 Physics Customization

Customize physics parameters for different scenarios:

[language=python, caption=Physics Customization] Custom physics configuration physics$_config =$ $'gravity' : 0.3, Reducedgravity'friction' : 0.1, Ballfriction'restitution' : 0.8, Ballbounciness'piston_mass' : 1.0$

env = pistonball$_env.PistonballEnv(physics_config = physics_config)$

7.10 Training Examples

7.10.1 PPO Training

[language=python, caption=PPO Training Example] import torch from toolkit.neural$_toolkitimportMLPPolicy$

Create environment env = pistonball$_env.PistonballEnv(num_agents = 4)$

Create networks for each agent policy$_networks = value_networks =$

for agent$_idinenv.agent_ids : policy_networks[agent_id] = MLPPolicyNetwork(input_dim = env.observation_space$ $env.action_space.n, hidden_dims = [256, 256])value_networks[agent_id] = MLPValueNetwork(input_dim =$ $env.observation_space.shape[0], hidden_dims = [256, 256])$

Training loop for episode in range(1000): observations = env.reset() episode$_rewards = agent_id : 0foragent_idine$

done = False while not done: Get actions from policy networks actions = for agent$_idinenv.agent_ids :$ $action_probs = policy_networks[agent_id](observations[agent_id])actions[agent_id] = torch.multinomial(action_pr$

Take step observations, rewards, dones, infos = env.step(actions)

Accumulate rewards for agent$_idinenv.agent_ids : episode_rewards[agent_id]+ = rewards[agent_id]$

done = all(dones.values())

Log episode results avg$_reward = sum(episode_rewards.values())/len(episode_rewards)print(f"Episodeepisode$ $Averagereward = avg_reward : .2f")$

7.10.2 MADDPG Training

[language=python, caption=MADDPG Training Example] import torch from toolkit.neural$_toolkitimportMLPP$

Create environment env = pistonball$_env.PistonballEnv(num_agents = 4)$

Create MADDPG networks actors = critics =

for agent$_idinenv.agent_ids : actors[agent_id] = MLPPolicyNetwork(input_dim = env.observation_space.shape[0$ $env.action_space.n, hidden_dims = [256, 256])critics[agent_id] = MLPQNetwork(input_dim =$ $env.observation_space.shape[0] * env.num_agents, output_dim = env.action_space.n, hidden_dims =$ $[256, 256])$

Training loop for episode in range(1000): observations = env.reset() episode$_rewards = agent_id : 0foragent_idine$

done = False while not done: Get actions from actors actions = for agent$_idinenv.agent_ids :$ $action_probs = actors[agent_id](observations[agent_id])actions[agent_id] = torch.multinomial(action_probs, 1).ite$

Take step next$_observations, rewards, dones, infos = env.step(actions)$

Update critics with global state global$_state = torch.cat([observations[agent_id]foragent_idinenv.agent_ids])$

for agent$_idinenv.agent_ids : Criticupdate(simplified)q_values = critics[agent_id](global_state)target_q =$ $rewards[agent_id] + 0.99 * q_values.max()...trainingcodehere$

observations = next$_observationsforagent_idinenv.agent_ids : episode_rewards[agent_id]+ = rewards[agent_id]$

done = all(dones.values())

avg$_reward = sum(episode_rewards.values())/len(episode_rewards)print(f"Episodeepisode : Averagereward =$ $avg_reward : .2f")$

7.11 Visualization and Analysis

7.11.1 Environment Rendering

[language=python, caption=Environment Rendering] import matplotlib.pyplot as plt

Render environment env = pistonball$_env.PistonballEnv(render_mode =' rgb_array')$

Run episode with rendering observations = env.reset() done = False

while not done: actions = agent$_id : env.action_space.sample()foragent_idinenv.agent_idsobservations, rewards,$ $env.step(actions)$

Render current state frame = env.render() plt.imshow(frame) plt.axis('off') plt.show()

done = all(dones.values())

7.11.2 Performance Analysis

[language=python, caption=Performance Analysis] from toolkit.plotkit import plot$_training_curvesimportnumpy$

Collect training data episode$_rewards = []episode_lengths = []$

for episode in range(100): observations = env.reset() episode$_reward = 0episode_length = 0$

done = False while not done: actions = agent$_id : env.action_space.sample()foragent_idinenv.agent_idsobservatio$ $env.step(actions)$

episode$_reward+ = sum(rewards.values())episode_length+ = 1done = all(dones.values())$

episode$_rewards.append(episode_reward)episode_lengths.append(episode_length)$

Plot results plot$_training_curves(episodes = np.arange(100), rewards = episode_rewards, losses =$ $episode_lengths, title = "PistonballTrainingProgress", save_path = "pistonball_training.png")$

7.12 Troubleshooting

7.12.1 Common Issues

Rendering Issues

If you encounter rendering problems:

[language=python, caption=Rendering Troubleshooting] Disable rendering for training env = pistonball$_env.PistonballEnv(render_mode = None)$

Use headless rendering env = pistonball$_env.PistonballEnv(render_mode =' rgb_array')$

Check display settings import os os.environ['SDL$_VIDEODRIVER'] =' dummy'$

Performance Issues

For performance optimization:

[language=python, caption=Performance Optimization] Reduce physics complexity env = pistonball$_env.Piston$
$'gravity' : 0.1,' friction' : 0.05)$

Use vector observations instead of images env = pistonball$_env.PistonballEnv(observation_type ='$
$vector')$

Limit episode length env = pistonball$_env.PistonballEnv(max_steps = 500)$

Training Issues

For training problems:

[language=python, caption=Training Troubleshooting] Adjust reward scaling env = pistonball$_env.PistonballEn$
0.1)

Use simpler action space env = pistonball$_env.PistonballEnv(action_type =' discrete')$

Increase exploration epsilon = 0.1 Epsilon-greedy exploration

7.13 Best Practices

7.13.1 Environment Configuration

1. Start with fewer agents (2-3) for initial experiments
2. Use vector observations for faster training
3. Adjust reward scaling to match your learning rates
4. Set appropriate episode lengths for your training setup

7.13.2 Training Strategies

1. Use centralized training with decentralized execution
2. Implement experience replay for multi-agent learning
3. Consider using parameter sharing between agents
4. Monitor coordination metrics during training

7.13.3 Evaluation

1. Evaluate on multiple random seeds
2. Measure both individual and team performance
3. Analyze coordination patterns in successful episodes
4. Compare against baseline policies

Parameter	**Type**	**Description**
num_agents	int	Number of pistons/agents (default: 4)
max_steps	int	Maximum steps per episode (default: 1000)
ball_speed	float	Initial ball velocity (default: 5.0)
gravity	float	Gravity strength (default: 0.5)
piston_speed	float	Piston movement speed (default: 2.0)
reward_scale	float	Reward scaling factor (default: 1.0)
observation_type	str	Observation type (default: 'vector')
render_mode	str	Rendering mode (default: 'rgb$_{a}rray'$)

Table 7.1: Environment parameters

Chapter 8

Kuramoto Oscillator Environment

8.1 Overview

The Kuramoto Oscillator Environment (`kos_env`) is a specialized environment for studying synchronization phenomena in complex systems. It implements the Kuramoto model, which describes the dynamics of coupled oscillators and is widely used in physics, biology, and engineering.

8.2 Theoretical Background

8.2.1 Kuramoto Model

The Kuramoto model describes the dynamics of N coupled oscillators:

$$\frac{d\theta_i}{dt} = \omega_i + \frac{K}{N} \sum_{j=1}^{N} \sin(\theta_j - \theta_i) + \eta_i(t) \tag{8.1}$$

where:

- θ_i is the phase of oscillator i
- ω_i is the natural frequency of oscillator i
- K is the coupling strength
- $\eta_i(t)$ is noise

8.2.2 Synchronization Order Parameter

The degree of synchronization is measured by the order parameter:

$$r = \left| \frac{1}{N} \sum_{j=1}^{N} e^{i\theta_j} \right| \tag{8.2}$$

where $r = 1$ indicates perfect synchronization and $r = 0$ indicates complete desynchronization.

8.3 Environment Features

8.3.1 Core Components

- **Oscillator Dynamics**: Realistic implementation of Kuramoto equations
- **Multiple Oscillators**: Configurable number of oscillators
- **Coupling Control**: Adjustable coupling strength
- **Noise Injection**: Configurable noise levels
- **Synchronization Metrics**: Real-time synchronization measurement

8.3.2 Observation Space

The environment provides rich observations:

[language=python, caption=Observation Components] observation = [phase$_1, phase_2, ..., phase_N, Oscillatorpha$

8.3.3 Action Space

Actions control the coupling strength and external forcing:

[language=python, caption=Action Space] Continuous actions action$_space = spaces.Box(low = [0.0, -1.0], [coupling_strength, external_force]high = [10.0, 1.0], dtype = np.float32)$

8.4 Installation and Setup

8.4.1 Basic Installation

[language=bash, caption=Install Kuramoto Environment] cd envlib$_project/env_lib/kos_envpipinstall-e.$

8.4.2 Dependencies

The environment requires:

- **NumPy**: For numerical computations
- **SciPy**: For ODE integration
- **Matplotlib**: For visualization (optional)

8.5 Basic Usage

8.5.1 Creating the Environment

[language=python, caption=Basic Environment Creation] from env$_libimportkos_env$

Create environment with default settings env = kos$_{e}nv.KuramotoEnv(num_{o}scillators = 10, coupling_{s}trength =$ $1.0, noise_{s}trength = 0.1, max_{s}teps = 200)$

Get environment information print(f"Number of oscillators: env.num$_{o}scillators")print(f"Observationspace :$ $env.observation_{s}pace")print(f"Actionspace : env.action_{s}pace")$

8.5.2 Running a Simple Episode

[language=python, caption=Simple Episode] import numpy as np

Reset environment observations = env.reset()

Run episode done = False total$_{r}eward = 0synchronization_{h}istory = []$

while not done: Random action action = env.action$_{s}pace.sample()$

Take step next$_{o}bservations, reward, done, info = env.step(action)$

Extract synchronization order$_{p}arameter = next_{o}bservations[env.num_{o}scillators]synchronization_{h}istory.appe$

observations = next$_{o}bservationstotal_{r}eward+ = reward$

print(f"Episode completed with total reward: total$_{r}eward")print(f"Finalsynchronization :$ $synchronization_{h}istory[-1] : .3f")$

8.6 Configuration Options

8.6.1 Environment Parameters

[language=python, caption=Environment Configuration] Configure environment config = 'num$_{o}scillators' :$ $20, Numberofoscillators'coupling_{s}trength' : 2.0, Initialcouplingstrength'noise_{s}trength' : 0.05, Noiselevel'ma$ $500, Maximumstepsperepisode'dt' : 0.01, Timestepforintegration'frequency_{d}istribution' :' uniform', Freque$ $[-1.0, 1.0], Frequencyrange'reward_{t}ype' :' synchronization'Rewardtype$

env = kos$_{e}nv.KuramotoEnv(**config)$

8.6.2 Parameter Descriptions

8.7 Reward System

8.7.1 Reward Types

The environment supports different reward schemes:

[language=python, caption=Reward Types] Synchronization reward def synchronization$_{r}eward(order_{p}paramete$ $1.0) : returnorder_{p}arameter - target_{s}ync$

Energy efficiency reward def energy$_{r}eward(order_{p}arameter, coupling_{s}trength) : sync_{b}enefit =$ $order_{p}arameterenergy_{c}ost = coupling_{s}trength**2returnsync_{b}enefit - 0.1*energy_{c}ost$

Multi-objective reward def multi$_{o}bjective_{r}eward(order_{p}arameter, coupling_{s}trength, action) :$ $sync_{r}eward = order_{p}arameterenergy_{p}enalty = 0.1*coupling_{s}trength**2action_{s}moothness =$ $-0.01*np.sum(action**2)returnsync_{r}eward + energy_{p}enalty + action_{s}moothness$

8.8 Advanced Features

8.8.1 Multiple Integration Methods

The environment supports different ODE integration methods:

[language=python, caption=Integration Methods] Configure integration method env = kos$_e nv.KuramotoEnv(i$
$rk4', Runge{-}Kutta4thorderintegration_m ethod =' euler', Eulermethod(faster)integration_m ethod ='$
$scipy', SciPyintegratordt = 0.01)$

8.8.2 Frequency Distributions

Different frequency distributions can be used:

[language=python, caption=Frequency Distributions] Uniform distribution env = kos$_e nv.KuramotoEnv(frequ$
$uniform', frequency_r ange = [-1.0, 1.0])$

Normal distribution env = kos$_e nv.KuramotoEnv(frequency_d istribution =' normal', frequency_m ean =$
$0.0, frequency_s td = 0.5)$

Lorentzian distribution env = kos$_e nv.KuramotoEnv(frequency_d istribution =' lorentzian', frequency_c enter =$
$0.0, frequency_w idth = 0.5)$

8.9 Training Examples

8.9.1 PPO Training for Synchronization

[language=python, caption=PPO Training Example] import torch import torch.optim as optim
import numpy as np from toolkit.neural$_t oolkitimportMLPPolicyNetwork, MLPValueNetworkfromenv_l ibim$

Create environment env = kos$_e nv.KuramotoEnv(num_o scillators = 15, coupling_s trength =$
$1.0, noise_s trength = 0.1, max_s teps = 300)$

Create networks policy$_n etwork = MLPPolicyNetwork(input_d im = env.observation_s pace.shape[0], output_d im$
$env.action_s pace.shape[0], hidden_d ims = [128, 128], activation =' relu', action_t ype =' continuous')$

value$_n etwork = MLPValueNetwork(input_d im = env.observation_s pace.shape[0], hidden_d ims =$
$[128, 128], activation =' relu')$

Optimizers policy$_o ptimizer = optim.Adam(policy_n etwork.parameters(), lr = 0.001)value_o ptimizer =$
$optim.Adam(value_n etwork.parameters(), lr = 0.001)$

Training loop num$_e pisodes = 1000episode_r ewards = []synchronization_h istory = []$

for episode in range(num$_e pisodes) : observations = env.reset()episode_r eward = 0episode_s ync =$
$0done = False$

Collect episode data states, actions, rewards, values, log$_p robs = [], [], [], [], []$

while not done: Get action from policy action$_m ean = policy_n etwork(torch.FloatTensor(observations))action_d$
$torch.distributions.Normal(action_m ean, 0.1)action = action_d ist.sample()log_p rob = action_d ist.log_p rob(action)$

Get value estimate value = value$_n etwork(torch.FloatTensor(observations))$

Take action next$_o bservations, reward, done, info = env.step(action.detach().numpy())$

Store data states.append(observations) actions.append(action.detach().numpy()) rewards.append(reward)
values.append(value.item()) log$_p robs.append(log_p rob.item())$

Track synchronization order$_parameter = next_observations[env.num_oscillators]episode_sync+ = order_parameter$

observations = next$_observationsepisode_reward+ = reward$

Convert to tensors states = torch.FloatTensor(states) actions = torch.FloatTensor(actions) rewards = torch.FloatTensor(rewards) values = torch.FloatTensor(values) old$_log_probs = torch.FloatTensor(log_pr$

Compute advantages advantages = compute$_gae(rewards, values, gamma = 0.99, gae_lambda = 0.95)returns = advantages + values$

Normalize advantages advantages = (advantages - advantages.mean()) / (advantages.std() + 1e-8)

PPO update for $_inrange(10) : MultipleepochsGetcurrentpolicyandvalueaction_means = policy_network(states)$ $torch.distributions.Normal(action_means, 0.1)new_log_probs = action_dist.log_prob(actions).sum(dim = 1)entropy = action_dist.entropy().mean()$

current$_values = value_network(states).squeeze()$

Compute ratios ratio = torch.exp(new$_log_probs - old_log_probs)$

Compute surrogate losses surr1 = ratio * advantages surr2 = torch.clamp(ratio, 0.8, 1.2) * advantages policy$_loss = -torch.min(surr1, surr2).mean()$

value$_loss = torch.nn.functional.mse_loss(current_values, returns)$

Total loss total$_loss = policy_loss + 0.5 * value_loss - 0.01 * entropy$

Update networks policy$_optimizer.zero_grad()value_optimizer.zero_grad()total_loss.backward()policy_optimizer.ste$

Log results episode$_rewards.append(episode_reward)synchronization_history.append(episode_sync/env.max_steps$

if episode avg$_reward = np.mean(episode_rewards[-100 :])avg_sync = np.mean(synchronization_history[-100 :])print(f"Episodeepisode : Reward = avg_reward : .2f, Sync = avg_sync : .3f")$

Plot results plot$_training_curves(episodes = np.arange(num_episodes), rewards = episode_rewards, losses = synchronization_history, title = "KuramotoOscillatorTraining", save_path = "kuramoto_training.png")$

def compute$_gae(rewards, values, gamma, gae_lambda) : """ComputeGeneralizedAdvantageEstimation"""adv$ $torch.zeros_like(rewards)last_advantage = 0$

for t in reversed(range(len(rewards))): if t == len(rewards) - 1: next$_value = 0else : next_value = values[t + 1]$

delta = rewards[t] + gamma * next$_value - values[t]advantages[t] = delta + gamma * gae_lambda * last_advantagelast_advantage = advantages[t]$

return advantages

8.9.2 Multi-Agent Control

Training multiple agents to control different groups of oscillators:

[language=python, caption=Multi-Agent Control] import torch import torch.optim as optim import numpy as np from toolkit.neural$_toolkitimportMLPPolicyNetworkfromenv_libimportkos_env$

Create environment with multiple oscillator groups env = kos$_env.KuramotoEnv(num_oscillators = 20, coupling_strength = 1.0, noise_strength = 0.1, max_steps = 300)$

Create agents for different oscillator groups num$_agents = 4oscillators_per_agent = env.num_oscillators//num_age$

agents = optimizers =

for i in range(num$_agents) : agents[f'agent'_i] = MLPPolicyNetwork(input_dim = oscillators_per_agent+ 2, Phases+order_param+couplingoutput_dim = 2, [coupling_strength, external_force]hidden_dims = [64, 64], activation =' relu', action_type =' continuous')optimizers[f'agent'_i] = optim.Adam(agents[f'agent'_i].p$ 0.001)

Training loop num$_episodes = 500episode_rewards = []$

for episode in range(num$_episodes) : observations = env.reset()episode_reward = 0done = False$

while not done: Get actions from all agents actions = for i in range(num$_agents) : agent_id = f'agent'_istart_idx = i * oscillators_per_agentend_idx = start_idx + oscillators_per_agent$

Agent-specific observation agent$_obs = np.concatenate([observations[start_idx : end_idx], Phases[observations[e$ 1]]$Couplingstrength])$

action$_mean = agents[agent_id](torch.FloatTensor(agent_obs))action_dist = torch.distributions.Normal(action_$ $action_dist.sample()actions[agent_id] = action.detach().numpy()$

Combine actions (average coupling, sum forces) combined$_action = np.array([np.mean([actions[agent_id][0]fora$

Take step next$_observations, reward, done, info = env.step(combined_action)$

Update observations observations = next$_observationsepisode_reward+ = reward$

episode$_rewards.append(episode_reward)$

if episode avg$_reward = np.mean(episode_rewards[-50 :])print(f"Episodeepisode : Averagereward = avg_reward : .2f")$

8.10 Visualization and Analysis

8.10.1 Phase Evolution Visualization

[language=python, caption=Phase Evolution] import matplotlib.pyplot as plt import numpy as np from env$_libimportkos_env$

Create environment env = kos$_env.KuramotoEnv(num_oscillators = 10, max_steps = 1000)$

Run simulation observations = env.reset() phase$_history = []sync_history = []$

done = False while not done: action = env.action$_space.sample()Randomactionsobservations, reward, done, inf$ $env.step(action)$

Store phases and synchronization phases = observations[:env.num$_oscillators]order_parameter = observations[env.num_oscillators]$

phase$_history.append(phases.copy())sync_history.append(order_parameter)$

Convert to arrays phase$_history = np.array(phase_history)sync_history = np.array(sync_history)$

Plot phase evolution fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(12, 8))

Phase trajectories for i in range(env.num$_oscillators) : ax1.plot(phase_history[:, i], label = f'Oscillatori + 1')ax1.$

Synchronization ax2.plot(sync$_history,' r-', linewidth = 2)ax2.set_title('SynchronizationOrderParameter')ax$

plt.tight$_layout()plt.savefig('kuramoto_evolution.png', dpi = 300, bbox_inches =' tight')plt.show()$

8.10.2 Parameter Space Analysis

[language=python, caption=Parameter Analysis] import numpy as np import matplotlib.pyplot as plt from env$_libimportkos_env$

Analyze synchronization vs coupling strength coupling$_strengths = np.linspace(0, 5, 50) final_sync =$ []

for K in coupling$_strengths : env = kos_env.KuramotoEnv(num_oscillators = 20, coupling_strength = K, noise_strength = 0.1, max_steps = 500)$

observations = env.reset() done = False

while not done: action = env.action$_space.sample() observations, reward, done, info = env.step(action)$

Final synchronization final$_sync.append(observations[env.num_oscillators])$

Plot results plt.figure(figsize=(10, 6)) plt.plot(coupling$_strengths, final_sync, 'b-', linewidth = 2) plt.xlabel('CouplingStrength(K)') plt.ylabel('FinalSynchronization') plt.title('SynchronizationvsCoupling$ $300, bbox_inches =' tight') plt.show()$

8.11 Research Applications

8.11.1 Synchronization Control

The environment is useful for studying:

- **Optimal Control**: Finding optimal coupling strategies
- **Robustness**: Studying synchronization under noise
- **Network Effects**: Analyzing different network topologies
- **Multi-Scale Dynamics**: Understanding hierarchical synchronization

8.11.2 Extensions and Modifications

[language=python, caption=Custom Extensions] Custom frequency distribution class CustomKuramotoEnv(kos
$def_{init_{(self, **kwargs): super().}}{}_{init_{(**kwargs)}}$

def generate$_frequencies(self) : Customfrequencygenerationreturnnp.random.exponential(scale = 0.5, size = self.num_oscillators)$

def custom$_reward(self, order_parameter, coupling_strength) : Customrewardfunctionsync_reward = order_parameterenergy_cost = 0.1 * coupling_strength * *2returnsync_reward - energy_cost$

8.12 Best Practices

8.12.1 Environment Configuration

1. Start with fewer oscillators (10-20) for initial experiments
2. Use appropriate coupling strength ranges (0-5)
3. Set reasonable noise levels (0.01-0.1)

4. Choose appropriate episode lengths (200-500 steps)

8.12.2 Training Strategies

1. Use continuous action spaces for smooth control
2. Implement reward shaping for better learning
3. Monitor both synchronization and energy efficiency
4. Consider multi-objective optimization

8.12.3 Analysis

1. Track phase evolution over time
2. Analyze synchronization transitions
3. Study parameter sensitivity
4. Compare different control strategies

Parameter	**Type**	**Description**
num_oscillators	int	Number of oscillators (default: 10)
coupling_strength	float	Initial coupling strength (default: 1.0)
noise_strength	float	Noise level (default: 0.1)
max_steps	int	Maximum steps per episode (default: 200)
dt	float	Time step for integration (default: 0.01)
frequency_distribution	str	Distribution type (default: 'uniform')
frequency_range	list	Frequency range (default: [-1.0, 1.0])
reward_type	str	Reward type (default: 'synchronization')

Table 8.1: Environment parameters

Chapter 9

Examples and Tutorials

9.1 Overview

This chapter provides comprehensive examples and tutorials for using the My Tool Box. These examples demonstrate how to combine the toolkit components with the environment library to create complete reinforcement learning experiments.

9.2 Quick Start Examples

9.2.1 Basic PPO Training

A simple PPO training example with the pistonball environment:

[language=python, caption=Basic PPO Training] import torch import torch.optim as optim import numpy as np from toolkit.neural$_toolkitimportMLPPolicyNetwork, MLPValueNetworkfromenv_libimpo$

Set random seeds torch.manual$_seed(42)np.random.seed(42)$

Create environment env = pistonball$_env.PistonballEnv(num_agents = 2, max_steps = 500)$

Create networks policy$_network = MLPPolicyNetwork(input_dim = env.observation_space.shape[0], output_dim$ $env.action_space.n, hidden_dims = [128, 128], activation =' relu')$

value$_network = MLPValueNetwork(input_dim = env.observation_space.shape[0], hidden_dims =$ $[128, 128], activation =' relu')$

Optimizers policy$_optimizer = optim.Adam(policy_network.parameters(), lr = 0.001)value_optimizer =$ $optim.Adam(value_network.parameters(), lr = 0.001)$

Training parameters num$_episodes = 1000gamma = 0.99gae_lambda = 0.95clip_ratio = 0.2value_loss_coef =$ $0.5entropy_coef = 0.01$

Training loop episode$_rewards = []episode_lengths = []$

for episode in range(num$_episodes) : observations = env.reset()episode_reward = 0episode_length =$ 0

Collect episode data states, actions, rewards, values, log$_probs = [], [], [], [], []$

done = False while not done: Get action from policy action$_probs = policy_network(torch.FloatTensor(observat$ $torch.distributions.Categorical(action_probs)action = action_dist.sample()log_prob = action_dist.log_prob(action)$

Get value estimate value = value$_network(torch.FloatTensor(observations))$

Take action next$_observations, reward, done, info = env.step(action.item())$

Store data states.append(observations) actions.append(action.item()) rewards.append(reward) values.append(value.item()) log$_probs.append(log_prob.item())$

observations = next$_observationsepisode_reward+ = rewardepisode_length+ = 1$

Convert to tensors states = torch.FloatTensor(states) actions = torch.LongTensor(actions) rewards = torch.FloatTensor(rewards) values = torch.FloatTensor(values) old$_log_probs = torch.FloatTensor(log_pr$

Compute advantages advantages = compute$_gae(rewards, values, gamma, gae_lambda)returns = advantages + values$

Normalize advantages advantages = (advantages - advantages.mean()) / (advantages.std() + 1e-8)

PPO update for $_inrange(10) : MultipleepochsGetcurrentpolicyandvalueaction_probs = policy_network(states)a$ $torch.distributions.Categorical(action_probs)new_log_probs = action_dist.log_prob(actions)entropy = action_dist.entropy().mean()$

current$_values = value_network(states).squeeze()$

Compute ratios ratio = torch.exp(new$_log_probs - old_log_probs)$

Compute surrogate losses surr1 = ratio * advantages surr2 = torch.clamp(ratio, 1 - clip$_ratio, 1 + clip_ratio) * advantagespolicy_loss = -torch.min(surr1, surr2).mean()$

value$_loss = torch.nn.functional.mse_loss(current_values, returns)$

Total loss total$_loss = policy_loss + value_loss_coef * value_loss - entropy_coef * entropy$

Update networks policy$_optimizer.zero_grad()value_optimizer.zero_grad()total_loss.backward()policy_optimizer.ste$

Log results episode$_rewards.append(episode_reward)episode_lengths.append(episode_length)$

if episode avg$_reward = np.mean(episode_rewards[-100 :])print(f"Episodeepisode : Averagereward = avg_reward : .2f")$

Plot results plot$_training_curves(episodes = np.arange(num_episodes), rewards = episode_rewards, losses = episode_lengths, title = "PPOTrainingProgress", save_path = "ppo_training.png")$

def compute$_gae(rewards, values, gamma, gae_lambda) : """ComputeGeneralizedAdvantageEstimation"""adv$ $torch.zeros_like(rewards)last_advantage = 0$

for t in reversed(range(len(rewards))): if t == len(rewards) - 1: next$_value = 0else : next_value = values[t + 1]$

delta = rewards[t] + gamma * next$_value - values[t]advantages[t] = delta + gamma * gae_lambda * last_advantagelast_advantage = advantages[t]$

return advantages

9.2.2 Multi-Agent DQN Training

Training multiple agents with DQN in the pistonball environment:

[language=python, caption=Multi-Agent DQN Training] import torch import torch.optim as optim import numpy as np from collections import deque import random from toolkit.neural$_toolkitimportMLPQN$

Set random seeds torch.manual$_seed(42)np.random.seed(42)$

Create environment env = pistonball$_env.PistonballEnv(num_agents = 3, max_steps = 500)$

Create Q-networks for each agent q$_networks = target_networks = optimizers =$

for agent$_i dinenv.agent_i ds : q_n etworks[agent_i d] = MLPQNetwork(input_d im = env.observation_s pace.shape[0]$ $env.action_s pace.n, hidden_d ims = [128, 128], activation =' relu')target_n etworks[agent_i d] = MLPQNetwork(i$ $env.observation_s pace.shape[0], output_d im = env.action_s pace.n, hidden_d ims = [128, 128], activation ='$ $relu')target_n etworks[agent_i d].load_s tate_d ict(q_n etworks[agent_i d].state_d ict())optimizers[agent_i d] =$ $optim.Adam(q_n etworks[agent_i d].parameters(), lr = 0.001)$

Replay buffers replay$_b uffers = agent_i d : deque(maxlen = 10000)foragent_i dinenv.agent_i ds$

Training parameters num$_e pisodes = 1000batch_s ize = 32gamma = 0.99epsilon_s tart = 1.0epsilon_e nd =$ $0.01epsilon_d ecay = 0.995target_u pdate_f req = 100$

epsilon = epsilon$_s tartepisode_r ewards = []$

for episode in range(num$_e pisodes) : observations = env.reset()episode_r eward = 0done = False$

while not done: Select actions actions = for agent$_i dinenv.agent_i ds : ifrandom.random() <$ $epsilon : actions[agent_i d] = env.action_s pace.sample()else : withtorch.no_g rad() : q_v alues =$ $q_n etworks[agent_i d](torch.FloatTensor(observations[agent_i d]))actions[agent_i d] = q_v alues.argmax().item()$

Take step next$_o bservations, rewards, dones, infos = env.step(actions)$

Store experience for agent$_i dinenv.agent_i ds : replay_b uffers[agent_i d].append((observations[agent_i d], actions[ag$

Update observations observations = next$_o bservationsepisode_r eward+ = sum(rewards.values())done =$ $all(dones.values())$

Train networks for agent$_i dinenv.agent_i ds : iflen(replay_b uffers[agent_i d]) >= batch_s ize : Samplebatchbatch =$ $random.sample(replay_b uffers[agent_i d], batch_s ize)states, actions, rewards, next_s tates, dones =$ $zip(*batch)$

Convert to tensors states = torch.FloatTensor(states) actions = torch.LongTensor(actions) rewards = torch.FloatTensor(rewards) next$_s tates = torch.FloatTensor(next_s tates)dones = torch.BoolTensor(d$

Compute Q-values current$_{qv} alues = q_n etworks[agent_i d](states).gather(1, actions.unsqueeze(1))next_{qv} alues =$ $target_n etworks[agent_i d](next_s tates).max(1)[0].detach()target_{qv} alues = rewards + (gamma *$ $next_{qv} alues * \ dones)$

Compute loss loss = torch.nn.functional.mse$_l oss(current_{qv} alues.squeeze(), target_{qv} alues)$

Update network optimizers[agent$_i d].zero_g rad()loss.backward()optimizers[agent_i d].step()$

Update target networks if episode for agent$_i dinenv.agent_i ds : target_n etworks[agent_i d].load_s tate_d ict(q_n etworks[$

Decay epsilon epsilon = max(epsilon$_e nd, epsilon * epsilon_d ecay)$

Log results episode$_r ewards.append(episode_r eward)$

if episode avg$_r eward = np.mean(episode_r ewards[-100 :])print(f"Episodeepisode : Averagereward =$ $avg_r eward : .2f, Epsilon = epsilon : .3f")$

Plot results plot$_t raining_c urves(episodes = np.arange(num_e pisodes), rewards = episode_r ewards, title =$ $"Multi - AgentDQNTraining", save_p ath = "multi_a gent_d qn.png")$

9.3 Advanced Examples

9.3.1 Transformer-based Policy Training

Using transformer networks for sequential decision making:

[language=python, caption=Transformer Policy Training] import torch import torch.optim as optim import numpy as np from toolkit.neural$_t oolkitimportTransformerPolicyNetwork, TransformerValueN$

Create environment env = pistonball$_{e}nv.PistonballEnv(num_{a}gents = 2, max_{s}teps = 200)$

Create transformer networks policy$_{n}etwork = TransformerPolicyNetwork(input_{d}im = env.observation_{s}pace$ $env.action_{s}pace.n, d_{m}odel = 128, nhead = 8, num_{l}ayers = 4, dim_{f}eedforward = 512, dropout = 0.1, fc_{d}ims = [128], activation =' relu')$

value$_{n}etwork = TransformerValueNetwork(input_{d}im = env.observation_{s}pace.shape[0], d_{m}odel = 128, nhead = 8, num_{l}ayers = 4, dim_{f}eedforward = 512, dropout = 0.1, fc_{d}ims = [128], activation =' relu')$

Optimizers policy$_{o}ptimizer = optim.Adam(policy_{n}etwork.parameters(), lr = 0.0001)value_{o}ptimizer = optim.Adam(value_{n}etwork.parameters(), lr = 0.0001)$

Training loop num$_{e}pisodes = 500episode_{r}ewards = []$

for episode in range(num$_{e}pisodes) : observations = env.reset()episode_{r}eward = 0done = False$

Store sequence data state$_{s}equence = []action_{s}equence = []reward_{s}equence = []$

while not done: Add to sequence state$_{s}equence.append(observations)$

Get action from transformer policy if len(state$_{s}equence) > 1 : Usesequencefortransformersequence_{t}ensor = torch.FloatTensor(state_{s}equence).unsqueeze(0)action_{p}robs = policy_{n}etwork(sequence_{t}ensor)action = torch.multinomial(action_{p}robs.squeeze(-1), 1).item()else : Randomactionforfirststepaction = env.action_{s}pace.sample()$

action$_{s}equence.append(action)$

Take step next$_{o}bservations, reward, done, info = env.step(action)reward_{s}equence.append(reward)$

observations = next$_{o}bservationsepisode_{r}eward+ = reward$

Train on episode sequence if len(state$_{s}equence) > 1 : states = torch.FloatTensor(state_{s}equence)actions = torch.LongTensor(action_{s}equence)rewards = torch.FloatTensor(reward_{s}equence)$

Compute advantages values = value$_{n}etwork(states).squeeze()advantages = compute_{a}dvantages(rewards, valu$

Policy loss action$_{p}robs = policy_{n}etwork(states)action_{d}ist = torch.distributions.Categorical(action_{p}robs.squee$ $action_{d}ist.log_{p}rob(actions)policy_{l}oss = -(log_{p}robs * advantages).mean()$

Value loss value$_{l}oss = torch.nn.functional.mse_{l}oss(values, rewards)$

Update networks policy$_{o}ptimizer.zero_{g}rad()policy_{l}oss.backward()policy_{o}ptimizer.step()$

value$_{o}ptimizer.zero_{g}rad()value_{l}oss.backward()value_{o}ptimizer.step()$

episode$_{r}ewards.append(episode_{r}eward)$

if episode avg$_{r}eward = np.mean(episode_{r}ewards[-50 :])print(f"Episodeepisode : Averagereward = avg_{r}eward : .2f")$

Plot results plot$_{t}raining_{c}urves(episodes = np.arange(num_{e}pisodes), rewards = episode_{r}ewards, title = "TransformerPolicyTraining", save_{p}ath = "transformer_{t}raining.png")$

def compute$_{a}dvantages(rewards, values, gamma = 0.99) : """Computeadvantagesforsequence"""advantages$ $torch.zeros_{l}ike(rewards)last_{a}dvantage = 0$

for t in reversed(range(len(rewards))): if t == len(rewards) - 1: next$_{v}alue = 0else : next_{v}alue = values[t + 1]$

delta = rewards[t] + gamma * next$_{v}alue - values[t]advantages[t] = delta + gamma * last_{a}dvantagelast_{a}dvantage$ $advantages[t]$

return advantages

9.4 Environment-Specific Examples

9.4.1 Kuramoto Oscillator Environment

Training agents to control oscillator synchronization:

[language=python, caption=Kuramoto Oscillator Training] import torch import torch.optim as optim import numpy as np from toolkit.neural$_{t}oolkitimportMLPPolicyNetwork, MLPValueNetworkfromen$

Create environment env = kos$_{e}nv.KuramotoEnv(num_{o}scillators = 10, coupling_{s}trength = 1.0, noise_{s}trength = 0.1, max_{s}teps = 200)$

Create networks policy$_{n}etwork = MLPPolicyNetwork(input_{d}im = env.observation_{s}pace.shape[0], output_{d}im$ $env.action_{s}pace.n, hidden_{d}ims = [128, 128], activation =' relu')$

value$_{n}etwork = MLPValueNetwork(input_{d}im = env.observation_{s}pace.shape[0], hidden_{d}ims = [128, 128], activation =' relu')$

Optimizers policy$_{o}ptimizer = optim.Adam(policy_{n}etwork.parameters(), lr = 0.001)value_{o}ptimizer = optim.Adam(value_{n}etwork.parameters(), lr = 0.001)$

Training loop num$_{e}pisodes = 500episode_{r}ewards = []synchronization_{m}etrics = []$

for episode in range(num$_{e}pisodes) : observations = env.reset()episode_{r}eward = 0episode_{s}ync = 0done = False$

while not done: Get action from policy action$_{p}robs = policy_{n}etwork(torch.FloatTensor(observations))action_{d}$ $torch.distributions.Categorical(action_{p}robs)action = action_{d}ist.sample()$

Take step next$_{o}bservations, reward, done, info = env.step(action.item())$

Compute synchronization metric sync$_{m}etric = compute_{s}ynchronization(observations)episode_{s}ync+ = sync_{m}etric$

observations = next$_{o}bservationsepisode_{r}eward+ = reward$

episode$_{r}ewards.append(episode_{r}eward)synchronization_{m}etrics.append(episode_{s}ync/env.max_{s}teps)$

if episode avg$_{r}eward = np.mean(episode_{r}ewards[-100 :])avg_{s}ync = np.mean(synchronization_{m}etrics[-100 :])print(f"Episodeepisode : Reward = avg_{r}eward : .2f, Sync = avg_{s}ync : .3f")$

Plot results plot$_{t}raining_{c}urves(episodes = np.arange(num_{e}pisodes), rewards = episode_{r}ewards, losses = synchronization_{m}etrics, title = "KuramotoOscillatorTraining", save_{p}ath = "kuramoto_{t}raining.png")$

def compute$_{s}ynchronization(observations) : """Computesynchronizationmetricfromoscillatorphases"""pha$ $observations[: env.num_{o}scillators]AssumingfirstNvaluesarephasesmean_{p}hase = np.mean(phases)sync = np.abs(np.mean(np.exp(1j * phases)))returnsync$

9.5 Visualization Examples

9.5.1 Training Progress Visualization

Comprehensive training visualization:

[language=python, caption=Training Visualization] import numpy as np import matplotlib.pyplot as plt from toolkit.plotkit import (plot$_{t}raining_{c}urves, plot_{p}erformance_{m}etrics, plot_{n}etwork_{a}nalysis)fromtoo$

Create environment and network env = pistonball$_{e}nv.PistonballEnv(num_{a}gents = 2)policy_{n}etwork = MLPPolicyNetwork(input_{d}im = env.observation_{s}pace.shape[0], output_{d}im = env.action_{s}pace.n, hidden_{d}ims$ $[128, 128])$

Simulate training data episodes = np.arange(1000) rewards = np.random.normal(0, 1, 1000).cumsum() losses = np.exp(-episodes / 200) + 0.1 * np.random.randn(1000) accuracy = 0.5 + 0.4 * (1 - np.exp(-episodes / 300))

Plot training curves plot$_training_curves(episodes = episodes, rewards = rewards, losses = losses, title = "TrainingProgress", save_path = "training_curves.png")$

Plot performance metrics plot$_performance_metrics(episodes = episodes, metrics = 'Accuracy' : accuracy, 'Rew$
$"PerformanceMetrics", save_path = "performance_metrics.png")$

Plot network analysis plot$_network_analysis(model = policy_network, title = "PolicyNetworkAnalysis", save_pat$
$"network_analysis.png")$

Create comprehensive dashboard fig, axes = plt.subplots(2, 2, figsize=(15, 10))

Training curves axes[0, 0].plot(episodes, rewards, 'b-', label='Reward') axes[0, 0].plot(episodes, losses, 'r-', label='Loss') axes[0, 0].set$_title('TrainingProgress')axes[0,0].legend()axes[0,0].grid(True)$

Performance metrics axes[0, 1].plot(episodes, accuracy, 'g-', label='Accuracy') axes[0, 1].set$_title('PerformanceM$

Reward distribution axes[1, 0].hist(rewards, bins=50, alpha=0.7, color='blue') axes[1, 0].set$_title('RewardDistri$

Moving average window = 50 moving$_avg = np.convolve(rewards, np.ones(window)/window, mode = '$
$valid')axes[1,1].plot(episodes[window-1 :], moving_avg, 'purple', label = f'window-episodemovingaverage')a$

plt.tight$_layout()plt.savefig('training_dashboard.png', dpi = 300, bbox_inches = 'tight')plt.show()$

9.6 Best Practices

9.6.1 Code Organization

1. Separate environment creation, network creation, and training loops
2. Use configuration dictionaries for hyperparameters
3. Implement proper logging and checkpointing
4. Create reusable training functions

9.6.2 Performance Optimization

1. Use appropriate batch sizes for your hardware
2. Implement experience replay for sample efficiency
3. Use vectorized environments when possible
4. Monitor GPU memory usage

9.6.3 Experiment Management

1. Use consistent random seeds for reproducibility
2. Save all hyperparameters and configurations
3. Implement proper evaluation protocols
4. Document all experimental results

Chapter 10

Troubleshooting

10.1 Common Issues and Solutions

This chapter provides solutions to common problems encountered when using the My Tool Box. Each section covers specific issues with detailed explanations and step-by-step solutions.

10.2 Installation Issues

10.2.1 Package Import Errors

Problem: ModuleNotFoundError

Symptoms: [language=python] ModuleNotFoundError: No module named 'toolkit' ModuleNotFoundError: No module named 'env$_{l}ib'$

Causes:

- Package not installed in development mode
- Wrong Python environment activated
- Missing dependencies

Solutions:

[language=bash, caption=Fix Import Errors] 1. Check current Python environment which python pip list | grep toolkit

2. Install in development mode cd toolkit$_{p}rojectpipinstall - e.$

cd ../envlib$_{p}rojectpipinstall - e.$

3. Verify installation python -c "import toolkit; print('Toolkit imported successfully')" python -c "import env$_{l}ib; print('Environmentlibraryimportedsuccessfully')$"

Problem: Version Conflicts

Symptoms: [language=python] ImportError: cannot import name 'X' from 'Y' VersionConflict: Package A requires B>=1.0, but C==0.9 is installed

Solutions:

[language=bash, caption=Resolve Version Conflicts] 1. Create fresh virtual environment python -m venv fresh$_envsourcefresh_env/bin/activateLinux/macOSorfresh_envWindows$

2. Install dependencies in correct order pip install torch>=1.9.0 pip install tensorflow>=2.6.0 pip install numpy>=1.20.0 pip install matplotlib>=3.5.0 pip install gym>=0.21.0

3. Install toolkit packages cd toolkit$_projectpipinstall - e.cd../envlib_projectpipinstall - e.$

10.3 Neural Network Issues

10.3.1 Memory Problems

Problem: CUDA Out of Memory

Symptoms: [language=python] RuntimeError: CUDA out of memory. Tried to allocate X MiB

Solutions:

[language=python, caption=Fix CUDA Memory Issues] 1. Reduce batch size batch$_size = 16Insteadof32or64$

2. Use gradient accumulation accumulation$_steps = 4foriinrange(0, len(data), batch_size) : batch = data[i : i + batch_size]loss = model(batch)loss = loss/accumulation_stepsloss.backward()$

if (i // batch$_size + 1)optimizer.step()optimizer.zero_grad()$

3. Clear cache import torch torch.cuda.empty$_cache()$

4. Use CPU if necessary device = 'cpu' if torch.cuda.is$_available()else'cpu'model = model.to(device)$

Problem: Model Not Learning

Symptoms:

- Loss not decreasing
- Rewards not improving
- Gradients are zero or NaN

Solutions:

[language=python, caption=Fix Learning Issues] 1. Check learning rate learning$_rate = 0.001Trydifferentvalu$ $0.01, 0.0001$

2. Add gradient clipping torch.nn.utils.clip$_grad_norm_(model.parameters(), max_norm = 1.0)$

3. Check for NaN values def check$_nan(model) : forname, paraminmodel.named_parameters() : iftorch.isnan(param).any() : print(f"NaNfoundinname")returnTruereturnFalse$

4. Use proper weight initialization def init$_weights(m) : ifisinstance(m, torch.nn.Linear) : torch.nn.init.xavier_uniform_(m.weight)m.bias.data.fill_(0.01)$

model.apply(init$_weights)$

5. Check reward scaling reward$_scale = 0.1Scalerewardstoreasonablerange$

10.3.2 Architecture Issues

Problem: Input/Output Dimension Mismatch

Symptoms: [language=python] RuntimeError: size mismatch, m1: [batch$_{s}ize, input_{d}im], m2 : [wrong_{d}im, hidden_{d}im]$

Solutions:

[language=python, caption=Fix Dimension Issues] 1. Check environment observation space print(f"Observation space: env.observation$_{s}pace")print(f"Actionspace : env.action_{s}pace")$

2. Create network with correct dimensions policy$_{n}etwork = MLPPolicyNetwork(input_{d}im = env.observation_{s}pace.shape[0], Useactualobservationdimoutput_{d}im = env.action_{s}pace.n, Useactualactiondim$
$[256, 256])$

3. Debug input shapes def debug$_{s}hapes(model, input_{d}ata) : print(f"Inputshape : input_{d}ata.shape")forname,$
$if hasattr(layer,' weight') : print(f"nameweightshape : layer.weight.shape")$

10.4 Environment Issues

10.4.1 Rendering Problems

Problem: Display/Window Issues

Symptoms: [language=python] pygame.error: No available video device SDL$_{V}IDEODRIVERerror$

Solutions:

[language=python, caption=Fix Rendering Issues] 1. Set display environment variables import os os.environ['SDL$_{V}IDEODRIVER'] =' dummy'os.environ['DISPLAY'] =': 0'$

2. Use headless rendering env = pistonball$_{e}nv.PistonballEnv(render_{m}ode =' rgb_{a}rray')$

3. Disable rendering for training env = pistonball$_{e}nv.PistonballEnv(render_{m}ode = None)$

4. Use matplotlib for display import matplotlib.pyplot as plt frame = env.render() plt.imshow(frame) plt.axis('off') plt.show()

Problem: Environment Not Resetting

Symptoms: [language=python] AttributeError: 'Environment' object has no attribute 'reset' TypeError: reset() takes 1 positional argument but 2 were given

Solutions:

[language=python, caption=Fix Reset Issues] 1. Check environment interface print(dir(env)) See available methods

2. Use correct reset method For newer gym versions observations = env.reset()

For older gym versions observations = env.reset(seed=42)

3. Handle multi-agent environments if hasattr(env, 'agent$_{i}ds') : observations = env.reset()observationsisadict$
$agent_{i}d : observationelse : observations = env.reset()observationsisasinglearray$

10.4.2 Performance Issues

Problem: Environment Too Slow

Symptoms:

- Training takes too long
- Low steps per second
- High CPU usage

Solutions:

[language=python, caption=Optimize Performance] 1. Use vectorized environments from gym.vector import make env = make('Pistonball-v0', num$_envs = 4)$

2. Disable unnecessary features env = pistonball$_env.PistonballEnv(render_mode = None, Disablerenderingobs$ $vector', Usevectorinsteadofimagemax_steps = 500 Limitepisodelength)$

3. Use multiprocessing import multiprocessing as mp from multiprocessing import Pool

def run$_episode(env_config) : env = pistonball_env.PistonballEnv(**env_config) Runepisodereturnepisode_rewar$

with Pool(processes=4) as pool: results = pool.map(run$_episode, [config] * 4)$

10.5 Training Issues

10.5.1 Convergence Problems

Problem: Training Not Converging

Symptoms:

- Rewards not increasing
- Loss oscillating
- Policy not improving

Solutions:

[language=python, caption=Fix Convergence Issues] 1. Adjust hyperparameters config = 'learning$_rate' :$ $0.0001, Trysmallerlearningrate'batch_size' : 64, Increasebatchsize'gamma' : 0.99, Discountfactor'gae_lambda'$ $0.95, GAEparameter'clip_ratio' : 0.2, PPOclipratio'value_loss_coef' : 0.5, Valuelosscoefficient'entropy_coef' :$ $0.01 Entropycoefficient$

2. Use learning rate scheduling from torch.optim.lr$_schedulerimportStepLRscheduler = StepLR(optimizer, ste$ $1000, gamma = 0.9)$

3. Implement early stopping best$_reward = -float('inf')patience = 100 no_improvement = 0$

for episode in range(num$_episodes) : episode_reward = train_episode()$

if episode$_reward > best_reward : best_reward = episode_rewardno_improvement = 0 Savebestmodeltorch.save(m$ $no_improvement+ = 1$

if no$_improvement >= patience : print(" Earlystoppingtriggered")break$

Problem: Exploding Gradients

Symptoms: [language=python] RuntimeError: Function 'AddBackward0' returned an invalid gradient at index 0 ValueError: gradients contain NaN values

Solutions:

[language=python, caption=Fix Gradient Issues] 1. Gradient clipping torch.nn.utils.clip$_grad_norm_(model.param$ 1.0)

2. Check for NaN in gradients def check$_gradients(model) : forname, paraminmodel.named_parameters() : ifparam.gradisnotNone : iftorch.isnan(param.grad).any() : print(f"NaNgradientinname")returnTrueret$

3. Use stable loss functions Instead of MSE for large values, use Huber loss loss$_fn = torch.nn.HuberLoss()$

4. Normalize inputs def normalize$_observations(obs) : return(obs - obs.mean())/(obs.std() + 1e-$ 8)

10.6 Multi-Agent Issues

10.6.1 Coordination Problems

Problem: Agents Not Coordinating

Symptoms:

- Individual agents perform well but team performance is poor
- Agents working against each other
- No emergent cooperative behavior

Solutions:

[language=python, caption=Improve Coordination] 1. Use centralized training with decentralized execution def centralized$_critic(observations, actions) : Concatenateallobservationsandactionsglobal_state$ $torch.cat([obsforobsinobservations.values()])global_actions = torch.cat([actforactinactions.values()])returnc$

2. Implement parameter sharing shared$_policy = MLPPolicyNetwork(input_dim, output_dim)policies =$ $agent_id : shared_policyforagent_idinagent_ids$

3. Use reward shaping for coordination def shaped$_reward(individual_reward, team_reward, coordination_metric)$ $returnindividual_reward + 0.1 * team_reward + 0.05 * coordination_metric$

4. Implement communication protocols def communicate$_observations(observations, communication_matrix) :$ $communicated_obs = foragent_id, obsinobservations.items() : Combineownobservationwithreceivedinformati$ $sum(communication_matrix[agent_id][other_id]*observations[other_id]forother_idinobservationsifother_id! =$ $agent_id)communicated_obs[agent_id] = torch.cat([obs, received_info])returncommunicated_obs$

10.6.2 Scaling Issues

Problem: Training Slow with Many Agents

Solutions:

[language=python, caption=Scale Multi-Agent Training] 1. Use asynchronous training import threading import queue def async$_training(agent_id, env, policy, queue): forepisodeinrange(episodes_per_agent): Trainagentepisode_data$ $train_episode(env, policy) queue.put((agent_id, episode_data))$

2. Implement experience replay with priority from collections import deque import random

class PrioritizedReplayBuffer: def $_{init}{}_{(self, capacity): self.buffer = deque(maxlen=capacity) self.priorities = deque(maxlen=capacity)}$

def add(self, experience, priority): self.buffer.append(experience) self.priorities.append(priority)

def sample(self, batch$_size): Samplebasedonprioritiesprobs = np.array(self.priorities)/sum(self.priorities)i$ $np.random.choice(len(self.buffer), batch_size, p = probs) return[self.buffer[i]foriinindices]$

3. Use hierarchical policies class HierarchicalPolicy: def $_{init}{}_{(self, high_level policy, low_level policies): self.high_level policy = high_level poli}$

def select$_action(self, observation, agent_id): High{-}leveldecisionhigh_level_action = self.high_level_policy(observ$ $levelexecutionlow_level_action = self.low_level_policies[agent_id](observation, high_level_action) returnlow_level_acti$

10.7 Debugging Tools

10.7.1 Logging and Monitoring

[language=python, caption=Debugging Setup] import logging import wandb import matplotlib.pyplot as plt

1. Set up logging logging.basicConfig(level=logging.INFO, format='handlers=[logging.FileHandler('training.lo logging.StreamHandler()])

2. Use Weights Biases for tracking wandb.init(project="my-toolbox-experiment") wandb.config.update("learning$_rate": 0.001, "batch_size": 32, "num_episodes": 1000)$

3. Monitor key metrics def log$_metrics(episode, reward, loss, accuracy): wandb.log("episode": episode, "rewar$ $Reward = reward: .2f, Loss = loss: .4f")$

4. Visualize training progress def plot$_training_progress(rewards, losses): fig, (ax1, ax2) = plt.subplots(1, 2, figsize = (12, 5))$

ax1.plot(rewards) ax1.set$_title('TrainingRewards') ax1.set_xlabel('Episode') ax1.set_ylabel('Reward')$

ax2.plot(losses) ax2.set$_title('TrainingLoss') ax2.set_xlabel('Episode') ax2.set_ylabel('Loss')$

plt.tight$_layout() plt.savefig('training_progress.png') plt.show()$

10.7.2 Model Inspection

[language=python, caption=Model Debugging] 1. Check model parameters def inspect$_model(model):$ $total_params = 0forname, paraminmodel.named_parameters(): print(f"name: param.shape, requires_grad =$ $param.requires_grad") total_params+ = param.numel() print(f"Totalparameters: total_params:,")$

2. Monitor gradients def monitor$_gradients(model): forname, paraminmodel.named_parameters():$ $ifparam.gradisnotNone: grad_norm = param.grad.norm().item() print(f"namegradientnorm:$ $grad_norm: .6f")$

3. Check for dead neurons def check$_dead_neurons(model, data_loader): activations = [] model.eval() withtorch.no$ $forbatchindata_loader: Getactivationsfromhiddenlayershidden_activations = model.get_hidden_activations(ba$

activations = torch.cat(activations, dim=0) dead$_neurons = (activations == 0).all(dim =$ $0) print(f"Deadneurons: dead_neurons.sum().item()/dead_neurons.numel()")$

4. Analyze weight distributions def analyze$_weights(model): weights = [] forname, paraminmodel.named_paran$

$if'weight'inname : weights.extend(param.data.flatten().tolist())$

plt.figure(figsize=(10, 6)) plt.hist(weights, bins=50, alpha=0.7) plt.title('Weight Distribution') plt.xlabel('Weight Value') plt.ylabel('Frequency') plt.show()

10.8 Getting Help

10.8.1 When to Seek Help

Seek help when you encounter:

- Issues not covered in this troubleshooting guide
- Unexpected behavior that persists after trying solutions
- Performance problems that affect research progress
- Bugs in the toolkit or environment code

10.8.2 How to Report Issues

When reporting issues, include:

[language=python, caption=Issue Report Template] System Information import sys import torch import numpy as np

print(f"Python version: sys.version") print(f"PyTorch version: torch.$_{version")print(f"NumPyversion:np._{version")print(f"CU}}$

Minimal Reproduction Code import toolkit from env$_libimportpistonball_env$

Your code here env = pistonball$_env.PistonballEnv()...restofthecodethatcausestheissue$

Error Message Paste the complete error traceback here

Expected Behavior Describe what you expected to happen

Actual Behavior Describe what actually happened

10.8.3 Resources

- **GitHub Issues**: Report bugs and request features
- **Documentation**: Check the main README files
- **Examples**: Review the example code in the repository
- **Community**: Join discussion forums and mailing lists

API Reference

.1 Neural Toolkit API

.1.1 Policy Networks

MLPPolicyNetwork

[language=python] class MLPPolicyNetwork(nn.Module): def $_{init}{}_{(self,input_dim,output_dim,hidden_dims=[256,256],activation='rel}$

Args: input$_d im(int) : Inputdimensionoutput_d im(int) : Outputdimensionhidden_d ims(list) : Hiddenlayerdimensionsactivation(str) : Activationfunctiondropout(float) : Dropoutratelayer_n orm(bool) : Uselayernormalizationaction_t ype(str) :' discrete' or' continuous' action_s td(float) : Standarddeviationforcont$

CNPolicyNetwork

[language=python] class CNPolicyNetwork(nn.Module): def $_{init}{}_{(self,input_channels,output_dim,conv_dims=[32,64,128],fc_dims=[256}$

Args: input$_c hannels(int) : Numberofinputchannelsoutput_d im(int) : Outputdimensionconv_d ims(list) : Convolutionallayerdimensionsfc_d ims(list) : Fullyconnectedlayerdimensionskernel_s izes(list) : Kernelsizesforconvlayersstrides(list) : Stridesforconvlayersactivation(str) : Activationfunctiondropout(f Dropoutrate$"""

RNNPolicyNetwork

[language=python] class RNNPolicyNetwork(nn.Module): def $_{init}{}_{(self,input_dim,output_dim,hidden_dim=256,num_layers=2,rnn_typ}$

Args: input$_d im(int) : Inputdimensionoutput_d im(int) : Outputdimensionhidden_d im(int) : Hiddendimensionnum_l ayers(int) : NumberofRNNlayersrnn_t ype(str) :' lstm' or' gru' fc_d ims(list) : Fullyconnectedlayerdimensionsactivation(str) : Activationfunctiondropout(float) : Dropoutrate$"""

TransformerPolicyNetwork

[language=python] class TransformerPolicyNetwork(nn.Module): def $_{init}{}_{(self,input_dim,output_dim,d_model=256,nhead=8,num_l}$

Args: input$_d im(int) : Inputdimensionoutput_d im(int) : Outputdimensiond_m odel(int) : Modeldimensionnhead Numberofattentionheadsnum_l ayers(int) : Numberoftransformerlayersdim_f eedforward(int) : Feedforwarddimensiondropout(float) : Dropoutratefc_d ims(list) : Fullyconnectedlayerdimensionsactivatio Activationfunction$"""

.1.2 Value Networks

MLPValueNetwork

[language=python] class MLPValueNetwork(nn.Module): def $_{init}{}_{(}{}^{self,input_dim,hidden_dims=[256,256],activation='relu',dropout=}$

Args: input$_dim(int):Inputdimensionhidden_dims(list):Hiddenlayerdimensionsactivation(str):$ $Activationfunctiondropout(float):Dropoutratelayer_norm(bool):Uselayernormalization$"""

CNValueNetwork

[language=python] class CNValueNetwork(nn.Module): def $_{init}{}_{(}{}^{self,input_channels,conv_dims=[32,64,128],fc_dims=[256,256],kernel}$

Args: input$_channels(int):Numberofinputchannelsconv_dims(list):Convolutionallayerdimensionsfc_dims(l$ $Fullyconnectedlayerdimensionskernel_sizes(list):Kernelsizesforconvlayersstrides(list):Stridesforconvl$ $Activationfunctiondropout(float):Dropoutrate$"""

RNNValueNetwork

[language=python] class RNNValueNetwork(nn.Module): def $_{init}{}_{(}{}^{self,input_dim,hidden_dim=256,num_layers=2,rnn_type='lstm',fc}$

Args: input$_dim(int):Inputdimensionhidden_dim(int):Hiddendimensionnum_layers(int):$ $NumberofRNNlayersrnn_type(str):'lstm'or'gru'fc_dims(list):Fullyconnectedlayerdimensionsactivation(s$ $Activationfunctiondropout(float):Dropoutrate$"""

TransformerValueNetwork

[language=python] class TransformerValueNetwork(nn.Module): def $_{init}{}_{(}{}^{self,input_dim,d_model=256,nhead=8,num_layers=6,dim}$

Args: input$_dim(int):Inputdimensiond_model(int):Modeldimensionnhead(int):Numberofattentionheadsn$ $Numberoftransformerlayersdim_feedforward(int):Feedforwarddimensiondropout(float):$ $Dropoutratefc_dims(list):Fullyconnectedlayerdimensionsactivation(str):Activationfunction$"""

.1.3 Q-Networks

MLPQNetwork

[language=python] class MLPQNetwork(nn.Module): def $_{init}{}_{(}{}^{self,input_dim,output_dim,hidden_dims=[256,256],activation='relu',dro}$

Args: input$_dim(int):Inputdimensionoutput_dim(int):Outputdimension(numberofactions)hidden_dims(list)$ $Hiddenlayerdimensionsactivation(str):Activationfunctiondropout(float):Dropoutrate$"""

DuelingQNetwork

[language=python] class DuelingQNetwork(nn.Module): def $_{init}{}_{(}{}^{self,input_dim,output_dim,hidden_dims=[256,256],value_hidden_dims}$

Args: input$_dim(int):Inputdimensionoutput_dim(int):Outputdimension(numberofactions)hidden_dims(list)$ $Sharedhiddenlayerdimensionsvalue_hidden_dims(list):Valuestreamhiddendimensionsadvantage_hidden_dims$ $Advantagestreamhiddendimensionsactivation(str):Activationfunctiondropout(float):Dropoutrate$"""

.1.4 State Encoders

MLPEncoder

[language=python] class MLPEncoder(nn.Module): def $_{init}{}_{(self,input_dim,output_dim,hidden_dims=[512,256],activation='relu',dropo}$

Args: input$_{d}im(int) : Inputdimensionoutput_{d}im(int) : Outputdimensionhidden_{d}ims(list) : Hiddenlayerdimensionsactivation(str) : Activationfunctiondropout(float) : Dropoutratelayer_{n}orm(bool) : Uselayernormalization$"""

CNNEncoder

[language=python] class CNNEncoder(nn.Module): def $_{init}{}_{(self,input_channels,output_dim,conv_dims=[32,64,128,256],fc_dims=[512],k}$

Args: input$_{c}hannels(int) : Numberofinputchannelsoutput_{d}im(int) : Outputdimensionconv_{d}ims(list) : Convolutionallayerdimensionsfc_{d}ims(list) : Fullyconnectedlayerdimensionskernel_{s}izes(list) : Kernelsizesforconvlayersstrides(list) : Stridesforconvlayersactivation(str) : Activationfunctiondropout(f$ $Dropoutrate$"""

RNNEncoder

[language=python] class RNNEncoder(nn.Module): def $_{init}{}_{(self,input_dim,output_dim,hidden_dim=256,num_layers=2,rnn_type='lstm'}$

Args: input$_{d}im(int) : Inputdimensionoutput_{d}im(int) : Outputdimensionhidden_{d}im(int) : Hiddendimensionnum_{l}ayers(int) : NumberofRNNlayersrnn_{t}ype(str) :' lstm'or'gru'fc_{d}ims(list) : Fullyconnectedlayerdimensionsactivation(str) : Activationfunctiondropout(float) : Dropoutrate$"""

TransformerEncoder

[language=python] class TransformerEncoder(nn.Module): def $_{init}{}_{(self,input_dim,output_dim,d_model=256,nhead=8,num_layers=6,}$

Args: input$_{d}im(int) : Inputdimensionoutput_{d}im(int) : Outputdimensiond_{m}odel(int) : Modeldimensionnhea$ $Numberofattentionheadsnum_{l}ayers(int) : Numberoftransformerlayersdim_{f}eedforward(int) : Feedforwarddimensiondropout(float) : Dropoutratefc_{d}ims(list) : Fullyconnectedlayerdimensionsactivatio$ $Activationfunction$"""

.1.5 Output Decoders

MLPDecoder

[language=python] class MLPDecoder(nn.Module): def $_{init}{}_{(self,latent_dim,output_dim,hidden_dims=[256,128],activation='relu',dropo}$

Args: latent$_{d}im(int) : Latentdimensionoutput_{d}im(int) : Outputdimensionhidden_{d}ims(list) : Hiddenlayerdimensionsactivation(str) : Activationfunctiondropout(float) : Dropoutrate$"""

CNNDecoder

[language=python] class CNNDecoder(nn.Module): def $_{init}{}_{(self,latent_dim,output_channels,fc_dims=[512,256],conv_dims=[256,128,64,3}$

Args: latent$_{d}im(int) : Latentdimensionoutput_{c}hannels(int) : Numberofoutputchannelsfc_{d}ims(list) : Fullyconnectedlayerdimensionsconv_{d}ims(list) : Convolutionallayerdimensionskernel_{s}izes(list) :$

$Kernelsizesforconvlayersstrides(list) : Stridesforconvlayersinitial_size(int) : Initialfeaturemapsize$"""

RNNDecoder

[language=python] class RNNDecoder(nn.Module): def $_{init}{}_{(self,latent_dim,output_dim,hidden_dim=256,num_layers=2,rnn_type='lstm}$

Args: latent$_dim(int) : Latentdimensionoutput_dim(int) : Outputdimensionhidden_dim(int) : Hiddendimensionnum_layers(int) : NumberofRNNlayersrnn_type(str) :' lstm'or'gru'max_seqlen(int) : Maximumsequencelengthfc_dims(list) : Fullyconnectedlayerdimensionsactivation(str) : Activationfunctio$

TransformerDecoder

[language=python] class TransformerDecoder(nn.Module): def $_{init}{}_{(self,latent_dim,output_dim,d_model=256,nhead=8,num_layers=6}$

Args: latent$_dim(int) : Latentdimensionoutput_dim(int) : Outputdimensiond_model(int) : Modeldimensionnhe$ $Numberofattentionheadsnum_layers(int) : Numberoftransformerlayersdim_feedforward(int) : Feedforwarddimensionmax_seqlen(int) : Maximumsequencelengthdropout(float) : Dropoutratefc_dims(list)$ $Fullyconnectedlayerdimensionsactivation(str) : Activationfunction$"""

.1.6 Discrete Tools

QTable

[language=python] class QTable: def $_{init}{}_{(self,state_spacesize,action_spacesize,initial_value=0.0):"""Q-learningtablefordiscretestate-actio}$

Args: state$_spacesize(int) : Sizeofstatespaceaction_spacesize(int) : Sizeofactionspaceinitial_value(float) : InitialQ - value$"""

def get$_value(self, state, action) : """GetQ - valueforstate - actionpair."""$

def update$_value(self, state, action, value, learning_rate = 0.1) : """UpdateQ - valueforstate - actionpair."""$

def get$_max_action(self, state) : """GetactionwithmaximumQ - valueforstate."""$

def get$_policy(self, state, epsilon = 0.1) : """Getepsilon - greedypolicyaction."""$

ValueTable

[language=python] class ValueTable: def $_{init}{}_{(self,state_spacesize,initial_value=0.0):"""Valuetablefordiscretestatespaces.}$

Args: state$_spacesize(int) : Sizeofstatespaceinitial_value(float) : Initialvalue$"""

def get$_value(self, state) : """Getvalueforstate."""$

def update$_value(self, state, value, learning_rate = 0.1) : """Updatevalueforstate."""$

def get$_values(self) : """Getallvalues."""$

PolicyTable

[language=python] class PolicyTable: def $_{init}{}_{(self,state_spacesize,action_spacesize):"""Policytablefordiscretestate-actionspaces.}$

Args: state$_spacesize(int) : Sizeofstatespaceaction_spacesize(int) : Sizeofactionspace$"""

def get$_value(self, state, action):"""Getpolicyprobabilityforstate-actionpair."""$

def update$_value(self, state, action, value, learning_rate = 0.1):"""Updatepolicyprobabilityforstate-actionpair."""$

def get$_policy(self, state):"""Sampleactionfrompolicy."""$

def get$_policy_probs(self, state):"""Getpolicyprobabilitiesforstate."""$

.2 Plotkit API

.2.1 Core Plotting Functions

plot_training_curves

[language=python] def plot$_training_curves(episodes, rewards, losses = None, title = "TrainingProgress", xla$
$"Episode", ylabel = "Value", save_path = None, show_plot = True):"""Plottrainingcurvesforrewardsandlos$

Args: episodes (array): Episode numbers rewards (array): Reward values losses (array, optional): Loss values title (str): Plot title xlabel (str): X-axis label ylabel (str): Y-axis label save$_path(str, optional): Pathtosaveplotshow_plot(bool): Whethertodisplayplot"""$

plot_performance_metrics

[language=python] def plot$_performance_metrics(episodes, metrics, title = "PerformanceMetrics", xlabel = "Episode", ylabel = "Score", save_path = None, show_plot = True):"""Plotmultipleperformancemetrics.$

Args: episodes (array): Episode numbers metrics (dict): Dictionary of metric arrays title (str): Plot title xlabel (str): X-axis label ylabel (str): Y-axis label save$_path(str, optional): Pathtosaveplotshow_plot(bool): Whethertodisplayplot"""$

plot_network_analysis

[language=python] def plot$_network_analysis(model, title = "NetworkAnalysis", save_path = None, show_plot = True, bins = 50):"""Analyzeandplotnetworkweightsandactivations.$

Args: model (nn.Module): Neural network model title (str): Plot title save$_path(str, optional): Pathtosaveplotshow_plot(bool): Whethertodisplayplotbins(int): Numberofhistogrambins"""$

plot_weight_distributions

[language=python] def plot$_weight_distributions(model, title = "WeightDistributions", save_path = None, show_plot = True, bins = 50):"""Plotweightdistributionsfornetworklayers.$

Args: model (nn.Module): Neural network model title (str): Plot title save$_path(str, optional): Pathtosaveplotshow_plot(bool): Whethertodisplayplotbins(int): Numberofhistogrambins"""$

.2.2 Specialized Plotting Functions

plot_policy_analysis

[language=python] def plot$_{p}olicy_{a}nalysis(states, action_{p}robabilities, action_{n}ames = None, title = "PolicyAnalysis", xlabel = "State", ylabel = "ActionProbability", save_{p}ath = None, show_{p}lot = True) : """Visualizepolicydistributionsandactionprobabilities.$

Args: states (array): State values action$_{p}robabilities(array) : Actionprobabilityarraysaction_{n}ames(list, option$ $Namesforactionstitle(str) : Plottitlexlabel(str) : X{-}axislabelylabel(str) : Y{-}axislabelsave_{p}ath(str, optiona$ $Pathtosaveplotshow_{p}lot(bool) : Whethertodisplayplot"""$

plot_value_function

[language=python] def plot$_{v}alue_{f}unction(states, values, title = "ValueFunction", xlabel = "State", ylabel = "Value", save_{p}ath = None, show_{p}lot = True) : """Visualizevaluefunctionestimates.$

Args: states (array): State values values (array): Value estimates title (str): Plot title xlabel (str): X-axis label ylabel (str): Y-axis label save$_{p}ath(str, optional) : Pathtosaveplotshow_{p}lot(bool) : Whethertodisplayplot"""$

plot_q_value_analysis

[language=python] def plot$_{qv}alue_{a}nalysis(states, actions, q_{v}alues, action_{n}ames = None, title = "Q - ValueAnalysis", xlabel = "State", ylabel = "Q - Value", save_{p}ath = None, show_{p}lot = True) : """AnalyzeQ - valuedistributionsandaction - valuefunctions.$

Args: states (array): State values actions (array): Action values q$_{v}alues(array) : Q{-}valuematrixaction_{n}ames($ $Namesforactionstitle(str) : Plottitlexlabel(str) : X{-}axislabelylabel(str) : Y{-}axislabelsave_{p}ath(str, optiona$ $Pathtosaveplotshow_{p}lot(bool) : Whethertodisplayplot"""$

.3 Environment Library API

.3.1 Pistonball Environment

PistonballEnv

[language=python] class PistonballEnv(gym.Env): def $_{init}{}_{(self, num_{a}gents=4, max_{s}teps=1000, ball_{s}peed=5.0, gravity=0.5, piston_{s}peed}$

Args: num$_{a}gents(int) : Numberofpistons/agentsmax_{s}teps(int) : Maximumstepsperepisodeball_{s}peed(float)$ $Initialballspeedgravity(float) : Gravitystrengthpiston_{s}peed(float) : Pistonmovementspeedreward_{s}cale(floa$ $Rewardscalingfactorobservation_{t}ype(str) : Observationtyperender_{m}ode(str) : Renderingmode"""$

def reset(self): """Reset environment to initial state."""

def step(self, actions): """Take action and return (observations, rewards, dones, infos)."""

def render(self, mode='human'): """Render environment."""

.3.2 Kuramoto Oscillator Environment

KuramotoEnv

[language=python] class KuramotoEnv(gym.Env): def $_{init}{}_{(self, num_oscillators=10, coupling_strength=1.0, noise_strength=0.1, max_ste}$

Args: num$_oscillators(int) : Numberofoscillatorscoupling_strength(float) : Initialcouplingstrengthnoise_stren$ $Noiselevelmax_steps(int) : Maximumstepsperepisodedt(float) : Timestepforintegrationfrequency_distributi$ $Distributiontypefrequency_range(list) : Frequencyrangereward_type(str) : Rewardtype$"""

def reset(self): """Reset environment to initial state."""

def step(self, action): """Take action and return (observation, reward, done, info)."""

def compute$_order_parameter(self) :$ """$Computesynchronizationorderparameter.$"""

.4 Utility Functions

.4.1 Weight Initialization

[language=python] def initialize$_weights(model, method =' xavier_uniform') :$ """$Initializemodelweights.$

Args: model (nn.Module): Neural network model method (str): Initialization method """

.4.2 Discrete Tools

[language=python] def q$_learning_update(q_table, state, action, reward, next_state, gamma = 0.99, alpha =$ $0.1) :$ """$Q - learningupdaterule.$"""

def sarsa$_update(q_table, state, action, reward, next_state, next_action, gamma = 0.99, alpha = 0.1) :$ """$SARSAupdaterule.$"""

def expected$_sarsa_update(q_table, state, action, reward, next_state, policy_table, gamma = 0.99, alpha =$ $0.1) :$ """$ExpectedSARSAupdaterule.$"""

def epsilon$_greedy_policy(q_table, state, epsilon = 0.1) :$ """$Epsilon - greedypolicy.$"""

def softmax$_policy(q_table, state, temperature = 1.0) :$ """$Softmaxpolicy.$"""

.5 Configuration

.5.1 Default Configuration

[language=python] DEFAULT$_CONFIG = 'device' :' cuda'iftorch.cuda.is_available()else'cpu',' dtype' : torch.f$

.5.2 Environment Configuration

[language=python] ENVIRONMENT$_CONFIG = 'pistonball' : 'num_agents' : 4,' max_steps' : 1000,' ball_speed' :$