Difference_DDPG_TD3

January 22, 2024

1 Twin-Delayed Deep Deterministic Policy Gradients

Authors: - Eshan Savla (Mtr-Nr. 91543) - Raphael Aberle (Mtr-Nr. 91386) - Leo Schäfer (Mtr-Nr. 91430)

Table of contents:

1.1 1. Introduction

This notebook is part of our final project submission of our project with "Twin-Delayed Deep Deterministic Policy Gradients in the course"Robot Programming" (RKIM121) from Prof. Dr.-Ing. Björn Hein and Gergely Sóti at the University of Applied Sciences in Karlsruhe.

This notebook aims to provide a comprehensive understanding of TD3 and its implementation. It includes explanations of the algorithm, code snippets, and discussions on benchmarking and performance.

Please note that this notebook assumes familiarity with reinforcement learning concepts and algorithms. If you are new to RL, it is recommended to first study the basics of DDPG before diving into TD3.

Let's get started! - - -

1.2 2. The path from DDPG to TD3

The TD3-Algorithm, being a direct successor to the DDPG-Algorithm, introduces several enhancements. Therefore, this chapter provides an overview of the motivations behind these extensions in

the TD3-Algorithm and points out the major differences between the two algorithms.

1.2.1 2.1. Motivation of TD3

The Deep Deterministic Policy Gradient (DPPG) deals with the folowing issues: - Overestimation Bias: DDPG tends to overestimate the Q-Values from the Q-Value function with it's one critic network - The Q-Value approximator sometimes develops an incorrect sharp peak for some actions - The policy will quickly exploit that peak leading to brittle and incorect behaviour - High sensitivity to hyperparameter: DDPG reacts strongly to changes in hyperparameters - Exploration is limited: Exploration is inherent difficult for the DDPG Agent - Limited robustness while learning: The learning is relatively unstable. Larger samples sizes are needed for stability.

1.2.2 2.2. Major differences bettween DDPG and TD3

The fundamental task of TD3 is to minimize the overestimation of Q-values and to generate more stable learning behavior. To achieve this, three modifications are proposed for TD3.

2.2.1. Clipped double Q-Learning

- TD3 learns two Q-functions instead of one (hence "twin")
- The smaller Q-value is used to form the targets in the Bellman error loss functions

$$y(r, s', d) = r + \gamma(1 - d) \min_{i=1,2}(Q\phi_{i,targ}(s', a'(s')))$$

2.2.2. Target policy smoothing

- The Target policy smoothing combats the exploitation of errors from the Q-Value approximator
- Clipped noise is added to the target action $a'(s')$ during the policy update process

$$a'(s') = clip(\mu_{\theta_{targ}}(s') + clip(\epsilon, -c, c), a_{Low}, a_{High}), \epsilon \sim N(0, \sigma)$$

- DDPG adds noise only in output action
- This addition of noise makes it harder for the policy to exploit Q-function errors by smoothing out Q along changes in action
- This results in a more robust and stable learning behavior
- For a higher-quality training reducing the scale of the noise of the target action over the course of training is an option

2.2.3. Delayed policy update

- TD3 updates the policy (and target networks) less frequently than the Q-function –> increases stability
- the original paper of the TD3-Alogrithm recommends one policy (and target) update for every two Q-function updates

2.2.4. Overview The resulting structure of the TD3-Algorithm is shown below:

1.3 3. Our TD3 implementation

For a better structure, the algorithm implementations are seperated into multiple classes and files, which are attached to this notebook. For more detailed explanations of the whole code, check the comments of the source code files.

1.3.1 3.1. Extension of TD3-Algorithm

Class description | Class | Description | |— |— | | | Actor | Contains the actor network for policy approximation | | Critic | Contains the critic network for Q-Value approximation | | Noise | Creates noise with the Ornstein-Uhlenbeck process | | ReplayBuffer | Contains the replay buffer for experience collection in Q-Learning | | DDPG | Contains the DDPG-Agent. Defines how the DDPG-Agent acts and learns | | TD3 | Contains the TD3-Agent. Defines how the TD3-Agent acts and learns |

Pseudocode The shown pseudocode gives an overview of the fundamental changes made for TD3 compared to DDPG.

Specific changes made for TD3-Agent The code snippets provided below illustrate the enhancements incorporated into the TD3 implementation. For clarity, each TD3 method that differs from the DDPG Algorithm is examined individually. The three significant modifications in the TD3 implementation, as compared to the DDPG implementation, are distinctly color-coded for each extension. The code presented primarily focuses on the crucial transitions from DDPG to TD3. For a more comprehensive understanding, please refer to the source code. - **Clipped double Q-Learning** - **Target policy smoothing** - **Delayed policy update**

Noise is incorporated into the target action to smooth the Q-value function, thereby mitigating the potential for exploiting errors in the Q-value function. Enhancing the quality of training is achieved by reducing the scale of noise. To counteract overestimation, the smallest calculated Q-value is chosen for the computation of critic loss.

The actor network and the target networks are only updated every n-step defined by the policy_freq parameter.

Further remarks:

- Action space fix
  - We've changed the implementation of the action space to accept values of multiple dimensions as required by the MuJoCo "Ant-v3" environment.
  - Neglecting this, we were only able to train one leg.
  - For reference, please review line 168 in td3.py.
- Training fix
  - Insted of using the done parameter, we are using the parameters terminated and truncated to stop the training.
  - Consequently, terminated and truncated are mapped to the done parameter with a logical "or" connection.
  - Neglecting this, our episode durations have exceeded 1000 timesteps and would falsify the results.
  - For reference, please review training sections in main_td3.py and the function compute_avg_return in functions.py.
- Training budget usage

  - Instead of using epochs and training steps, we've changed the implementation to use training steps only.
  - Neglecting this, an early stop of an epoch (e.g. earlier than 1000 steps) would lead to a earlier overall training stop (e.g. earlier than 1 million timesteps).
  - Consequently, it enables us to use the complete training budget.
  - For reference, please review training sections in main_td3.py.
- Delayed training start
  - By integrating a delayed learning start (e.g. starting at 10.000 timesteps), we've optimized the training process as the first experienced are not exploited directly.
  - Instead, we are collecting first experiences to stabilize the training convergence.
  - For reference, please review training sections in main_td3.py.

1.3.2 3.2. Hyperparameters

The hyperparameters are one way of influencing the algorithm and model quality.

General parameters: - Discount factor ( ) -> immediate rewards > future rewards - value range: 0 - 1 => higher values emphasize long-term rewards - Learning-rate (lr) -> learning rate for neural network optimizer - Determines the size of steps taken during the optimization process during updates - -greedy ( ) -> random action with -probability for exploration - Update weight ( ) -> how much target network is updated - Determines the rate of updating the target networks. It is slowly to reduce the variance in learning and prevent overly influence of most recent experiences

TD3 specific parameters: - Target action noise ( ) -> smoothing q-function by adding noise to action - It combats overfitting and prevents the learning process to become too reliant on current policy estimes - Update frequency -> Update rate of actor and target networks - Usually, the policy and target networks get updated every second update of the critic network

During the implementation of TD3, we've integrated further parameters to improve the model results.

Further parameters: - Noise reduction -> reduce noise over time steps - The noise gets reduces over ascending timesteps in a e-function. - Min noise factor -> max noise reduction - Introduced to maintain a minimum noise within higher time steps.

1.4 4. Test environment - Ant-v3

To do a proper benchmark of different algorithms, an environment is required.

Consequently, this chapter describes the used test environment "Ant-v3" from MuJoCo.

MuJoCo stands for Multi Joint dynamics with Contact. It is a physics engine designed for simulating and controlling the dynamics of rigid body systems.

- Action Space - An ant with 4 legs - 2 joints in each leg sum up to 8 joints in total - For each joint, it is possible to apply a continuous torque from -1 to 1 Nm.

- Observation Space - 27 – 29 (extended) observations - Positions of the torso (x, y, z) - velocities (in x-, y-, z-direction) - angles (around x, y, z) - angular velocities (around the x-, y-, z-axis)

- Reward - An important element in Reinforcement Learning is the reward definition. - In Ant-v3, the reward is defined as: reward = healthy_reward + forward_reward - ctrl_cost - contact_cost

```
- healthy_reward: Is received every timesteps the torso is within the healthy_z_range (defaul
- forward_reward: Motion in x-direcetion is rewarded (aim: as fast as possible)
- ctrl_cost: Negative reward is received if the taken actions are to large (ctrl_cost_weight =
- contact_cost: Negative reward is received if the external contact force is too large (jumpi
```

- Episode end - Any state space value is no longer finite - Truncation: The z-coordinate of the torso is not in the defined interval (default: [0,2:1]) - Termination: The episode duration reaches 1000 timesteps

For further informations, please review: https://www.gymlibrary.dev/environments/mujoco/ant/

1.5 5. Usage of RL TD3 algorithm

Disclaimer:

We've experienced that the code execution in the notebook is quite slow and could lead to problems with MuJoCo. For better performance please execute the files from the repository directly. Furthermore, the user experience can be improved by using a gpu as our comparison has shown that the training is 7% faster. (the explicit difference is hardware dependent)

1.5.1 5.1. Training

For a better overview of hyperparameter sets used for training the hydra library is used, in order to load hyperparameter sets from config files. A new config file for changing the hyperparameters could be created by copying one of the existing ones in the config folder and ajust it's values.

[ ]:
```
import hydra
from hydra.utils import instantiate
import matplotlib.pyplot as plt
import pandas as pd
import os
import sys
import timeit
from tqdm import tqdm
import matplotlib.pyplot as plt

sys.path.insert(0, '../src') # DO NOT CHANGE

import gymnasium as gym
import numpy as np
import tensorflow as tf
import tensorflow_addons as tfa
from td3 import TD3Agent
from replay_buffer import ReplayBuffer
from functions import compute_avg_return
```

The config file that should be used for training could be specified like this:

[ ]:
```
config_name = "config"
with hydra.initialize(config_path="../configs/", job_name="td3_config"):
    cfg = hydra.compose(config_name=config_name) # Option to test multiple␣
↪configurations: Change the config name to your desired config file
```

This is the main function for training the agent.

[ ]:
```
def main_training(load_replay_buffer:bool = True):
    """
    The main function for running the TD3 training algorithm.

    Parameters:
        - load_replay_buffer (bool): Whether to load the replay buffer from a␣
↪checkpoint. Default is True.

    Returns:
        - None
    """
    physical_devices = tf.config.list_physical_devices('GPU')
    for device in physical_devices:
        tf.config.experimental.set_memory_growth(device, True)
    replay_buffer = instantiate(cfg.ReplayBuffer)
    env = gym.make('Ant-v3', max_episode_steps=1000,autoreset=True,␣
↪render_mode='rgb_array') #human
    agent = TD3Agent(env.action_space, env.observation_space.shape[0],gamma=cfg.
↪TD3Agent.gamma,tau=cfg.TD3Agent.tau, epsilon=cfg.TD3Agent.epsilon,␣
↪noise_clip=cfg.TD3Agent.noise_clip, policy_freq=cfg.TD3Agent.policy_freq)
    if type(cfg.TD3Agent.use_checkpoint_timestamp) == bool and cfg.TD3Agent.
↪use_checkpoint_timestamp:
        print("Loading most recent checkpoint")
        agent.load_weights(use_latest=True)
        if load_replay_buffer:
            replay_buffer.load(agent.save_dir)
    elif not cfg.TD3Agent.use_checkpoint_timestamp:
        print("No checkpoint loaded. Starting from scratch.")
    else:
        print("Loading weights from timestamp: ", cfg.TD3Agent.
↪use_checkpoint_timestamp)
        agent.load_weights(load_dir=os.path.join(cfg.TD3Agent.weights_path, cfg.
↪TD3Agent.use_checkpoint_timestamp+'/'), use_latest=False)
        if load_replay_buffer:
            replay_buffer.load(agent.save_dir)
    total_timesteps = cfg.Training.start
    returns = list()
    actor_losses = list()
```

```
    critic1_losses = list()
    critic2_losses = list()
    evals_dir = None
    first_training = True
    eval_count = 0
    with tqdm(total=cfg.Training.timesteps, desc="Timesteps", leave=True) as␣
↪pbar:
        while total_timesteps <= cfg.Training.timesteps:
            obs, _ = env.reset()
            # gather experience
            agent.noise_output_net.reset()
            agent.noise_target_net.reset()

            ep_actor_loss = 0
            ep_critic1_loss = 0
            ep_critic2_loss = 0
            steps = 0
            for j in range(1000):
                steps += 1
                action = agent.act(np.array([obs]),␣
↪random_action=(total_timesteps < cfg.Training.start_learning)) # i < 1 no␣
↪policy for first episode
                # execute action

                # Patching terminated/truncated state behaviour based on issue:
                # https://github.com/Farama-Foundation/Gymnasium/pull/101
                # and
                # https://github.com/openai/gym/issues/3102
                new_obs, r, terminated, truncated, info = env.step(action)
                done = terminated or truncated
                if steps >= 1000:
                    episode_truncated = not done or info.get("TimeLimit.
↪truncated", False)
                    info["TimeLimit.truncated"] = episode_truncated
                    # truncated may have been set by the env too
                    truncated = truncated or episode_truncated
                    done = terminated or truncated
                replay_buffer.put(obs, action, r, new_obs, done)
                obs = new_obs
                if done:
                    break

            total_timesteps += steps

            if total_timesteps >= cfg.Training.start_learning:
            # Learn from the experiences in the replay buffer.
                for s in range(cfg.Training.batch_size):
```

```
                    s_states, s_actions, s_rewards, s_next_states, s_dones =␣
↪replay_buffer.sample(cfg.Training.sample_size, cfg.Training.unbalance)
                    actor_l, critic1_l, critic2_l = agent.learn(s_states,␣
↪s_actions, s_rewards, s_next_states, s_dones,s, total_timesteps)
                    ep_actor_loss += actor_l
                    ep_critic1_loss += critic1_l
                    ep_critic2_loss += critic2_l
                if eval_count % 25 == 0 or first_training:
                    first_training = False
                    avg_return, _, _, _, _ = compute_avg_return(env, agent,␣
↪num_episodes=5, max_steps=1000, render=False)
                    print(
                        f'Timestep {total_timesteps}, actor loss {ep_actor_loss␣
↪/ steps}, critic 1 loss {ep_critic1_loss / steps}, critic 2 loss␣
↪{ep_critic2_loss/steps} , avg return {avg_return}')
                    agent.save_weights()
                    replay_buffer.save(agent.save_dir)
                if evals_dir is None:
                    evals_dir = '../evals/'+ agent.save_dir.split('/')[-2] + "/"
                    os.makedirs(evals_dir, exist_ok=True)   # create folder if␣
↪not existing yet
                returns.append(avg_return)
                actor_losses.append(tf.get_static_value(ep_actor_loss) / steps)
                critic1_losses.append(tf.get_static_value(ep_critic1_loss) /␣
↪steps)
                critic2_losses.append(tf.get_static_value(ep_critic2_loss) /␣
↪steps)
                df = pd.DataFrame({'returns': returns, 'actor_losses':␣
↪actor_losses, 'critic1_losses': critic1_losses, 'critic2_losses':␣
↪critic2_losses})
                plot_losses = df.drop("returns", axis=1, inplace=False).
↪plot(title='TD3 losses', figsize=(10, 5))
                plot_losses.set(xlabel='Epochs', ylabel='Loss')
                plot_losses.get_figure().savefig(evals_dir+'losses_td3.png')

                returns_df = pd.DataFrame({'returns': returns})
                plot_returns = returns_df.plot(title='TD3 returns',␣
↪figsize=(10, 5))
                plot_returns.set(xlabel='Epochs', ylabel='Returns')
                plot_returns.get_figure().savefig(evals_dir+'returns_td3.png')
                plt.close('all')
                df.to_csv(evals_dir+'td3_results.csv', index=True)
                eval_count += 1
            pbar.update(steps)
```

```
    agent.save_weights()
    replay_buffer.save(agent.save_dir)

    env.close()
```

Execute the training with the current loaded config:

[ ]:
```
elapsed_time = timeit.timeit(lambda: main_training(load_replay_buffer=True),␣
 ↪number=1)
minutes, seconds = divmod(elapsed_time, 60)
print(f"The main function ran for {int(minutes)} minutes and {seconds:.2f}␣
 ↪seconds.")
```

1.5.2 5.2. Enjoy

After training or by loading pretrained models, the agent can be evaluated by the main_enjoy-function.

[ ]:
```
from ddpg import DDPGAgent
from td3 import TD3Agent
import os
import gymnasium as gym
from functions import compute_avg_return
import pandas as pd
```

[ ]:
```
def main_enjoy(agent_type:str, load_dir:str=None, use_latest:str=True, render:
 ↪bool=True, num_episodes:int=150):  #defaults: agent_type="td3",␣
 ↪load_dir=None, use_latest=True, render_mode=None
    """
    This function allows you to enjoy a trained agent in the environment.

    Parameters:
        agent_type (str): Specify the agent type you want to enjoy. Options:␣
 ↪"ddpg" or "td3"
        load_dir (str, optional): Defaults to None.
        use_latest (str, optional): Defaults to True.
        render (bool, optional): True  => render the environment visually //␣
 ↪False => Run enjoy without environement and agent rendering. Defaults to␣
 ↪True.
    Returns:
        - None
    """
    render_mode = "human" if render else "rgb_array"
    env = gym.make(id='Ant-v3', autoreset=True, render_mode = render_mode)  #␣
 ↪create the environment
                                                                            ␣
 ↪# id = Environment ID
```

```
␣
↪# autoreset=True => automatically reset the environment after an episode is␣
↪done
␣
↪#render_mode='human' => render the environment visually //␣
↪render_mode='rgb_array' => render the environment as an array to collect␣
↪results
   if agent_type == "ddpg":   #if you want to enjoy a DDPG agent
       from ddpg_config import cfg as ddpg_cfg, config_name
       #create a DDPGAgent with the same parameters as the one used for␣
↪training and configurations specified in yaml file (loaded via Hydra)
       agent = DDPGAgent(env.action_space, env.observation_space.
↪shape[0],gamma=ddpg_cfg.DDPGAgent.gamma,tau=ddpg_cfg.DDPGAgent.tau,␣
↪epsilon=0)

   elif agent_type == "td3":  #if you want to enjoy a TD3 agent
       #create a TD3Agent with the same parameters as the one used for␣
↪training and configurations specified in yaml file (loaded via Hydra)
       from td3_config import cfg as td3_cfg, config_name
       agent = TD3Agent(env.action_space, env.observation_space.
↪shape[0],gamma=td3_cfg.TD3Agent.gamma,tau=td3_cfg.TD3Agent.tau, epsilon=0)
   else: #handling wrong agent type specification
       raise ValueError("Invalid agent type")

   agent.load_weights(load_dir=load_dir, use_latest=use_latest) #load the␣
↪weights of the agent
   obs, _ = env.reset() #reset the environment and get the initial observation

   avg_return, avg_return_stddev, episode_no, returns, stddevs =␣
↪compute_avg_return(env, agent, num_episodes=num_episodes, max_steps=1000,␣
↪render=False) #compte the average return and specify the to be evaluated␣
↪number of episodes
   print(f"Average return: {avg_return}, Standard deviation:␣
↪{avg_return_stddev}")

   #To get a unique benchmark result, we save the results in a csv file
   time_stamp = agent.save_dir.split("/")[-2]  #get timestamp
   user_name = os.getlogin()                   #get username
   df = pd.DataFrame({"file": [config_name], "time_stamp": [time_stamp],␣
↪"user_name": [user_name], "agent_type": [agent_type], "avg_return":␣
↪[avg_return], "return_stddev": [avg_return_stddev], "episode_no":␣
↪[episode_no], "returns": [returns], "stddevs": [stddevs]}) #create pandas DF
   df.to_csv("../benchmarks_new.csv", mode="a", header=False, index=False)
   env.close()
```

Execute the enjoy here, by specifying the following parameters: - the training algorithm - the directory of the trained model for the evaluation - a flag, whether the function should just use the

latest_trained model

```
[ ]: main_enjoy("td3", render=False)  #speficy the algorithm/ agent type ("ddpg" or␣
     ↪"td3"), specify the path to the model you want to evaluate. For more␣
     ↪information: see Mapping_mod-conf.md,  True if you want to use the latest␣
     ↪checkpoint
```

1.5.3 5.3 Evaluation

For evaluation purposes, we save average returns and losses in cvs files while training. For the benachmark tests the return in each episode and the standard deviation of the returns is saved in a csv file. Both files can be evaluated using the evaluate.py file

```
[ ]: import pandas as pd
     import numpy as np
     import matplotlib.pyplot as plt
     import ast

     from functions import flatten
     import datetime
     import numpy as np
     import datetime
     from scipy import stats
     import os
```

```
[ ]: def evaluate_enjoy(data_path_csv:str, plot_type: str = 'bar', plot_avgs:bool =␣
     ↪False, plot_timeseries:bool = False, **kwargs):
         """
         This function allows you to evaluate the enjoy phase of a trained model␣
     ↪based on the benchmark results of the different agents.

         Parameters:
             data_path_csv (str, optional): _description_. Specify the path to the␣
     ↪csv file
             plot_type (str, optional): _description_. Specify the plot_type for the␣
     ↪graph. Default = 'bar'.
             only_avgs (bool, optional): _description_. Default = False.
             plot_title (str, optional): _description_. Specify plot tile. Default =␣
     ↪None
             x_axis_title (str, optional): _description_. Specify title of x axis.␣
     ↪Default = None
             y_axis_title (str, optional): _description_. Specify title of y axis.␣
     ↪Default = None
         Returns:
             - None
         """
```

```
    plot_title = kwargs.get("plot_title", "Returns per Episode for different␣
↪configurations")
    x_axis_title = kwargs.get("x_axis_title", "Episode")
    y_axis_title = kwargs.get("y_axis_title", "Returns per Episode")
    # Read data from CSV file
    data = pd.read_csv(data_path_csv)

    # Convert the string representation of lists into actual lists of floats
    data['returns'] = data['returns'].apply(ast.literal_eval)
    data['stddevs'] = data['stddevs'].apply(ast.literal_eval)
    data['episode_no'] = data['episode_no'].apply(ast.literal_eval)

    if isinstance(data['episode_no'], str):
        episode_no = ast.literal_eval(data['episode_no'])

    data['stddevs'] = data['stddevs'].apply(flatten)

    # Calculate the mean of the lists in 'stddevs' column
    data['stddevs'] = data['stddevs'].apply(np.mean)

    # Group the data and calculate the mean standard deviation per episode over␣
↪all experiments
    mean_stddev_per_episode = data.groupby(['time_stamp', 'config_name',␣
↪'user_name', 'agent_type'])['stddevs'].mean()

    # Update y1 and y2 values
    data['y1'] = data.apply(lambda row: np.mean(row['returns']) +␣
↪mean_stddev_per_episode[(row['time_stamp'], row['config_name'],␣
↪row['user_name'], row['agent_type'])], axis=1)
    data['y2'] = data.apply(lambda row: np.mean(row['returns']) -␣
↪mean_stddev_per_episode[(row['time_stamp'], row['config_name'],␣
↪row['user_name'], row['agent_type'])], axis=1)

    # Create a linegraph with x-axis = episode, y-axis=returns per episode
    timestamp = datetime.datetime.now().strftime("%Y-%m-%d_%H-%M")
    if plot_type.lower() == 'line':
        fig, ax = plt.subplots()
        for i, row in data.iterrows():
            episode_no = row['episode_no']
            returns = row['returns']
            stddevs = row['stddevs']

            # Convert the list into a string representation
            episode_no_str = str(episode_no)
            # Convert the string representation of episode numbers into actual␣
↪list of floats
```

```
            episode_no = ast.literal_eval(episode_no_str)

            # Plot the returns per episode
            label = f' {row["agent_type"]}-{row["config_name"]}'
            label_means = f'{row["agent_type"]}-{row["config_name"]}-mean'

            if plot_timeseries:
                ax.plot(episode_no, returns, label=label)
            if plot_avgs:
                ax.plot(episode_no, [row["avg_return"] for i in␣
↪range(len(returns))], label=label_means)

            # Add a subplot using fill_between
            if plot_timeseries:
                ax.fill_between(episode_no, np.array(returns) + np.
↪array(stddevs), np.array(returns) - np.array(stddevs), alpha=0.3)
            if plot_avgs:
                ax.fill_between(episode_no, np.array([row["avg_return"] for i␣
↪in range(len(returns))]) + np.array([row["avg_return_stddev"] for i in␣
↪range(len(returns))]), np.array([row["avg_return"] for i in␣
↪range(len(returns))]) - np.array([row["avg_return_stddev"] for i in␣
↪range(len(returns))]), alpha=0.3)

        # Set the labels and title and legend
        ax.set_xlabel(x_axis_title)
        ax.set_ylabel(y_axis_title)
        ax.set_title(plot_title)
        ax.legend()

        # Save the plot with the unique timestamp in the file name
        fig.savefig(f'{timestamp}_returns_per_episode.png')

    if plot_type.lower() == 'bar':
        # Create a barchart with x-axis = configuration, y-axis=average return
        color=['green', 'blue', 'blue', 'gold', 'gold', 'darkred',  'darkred',␣
↪'darkgreen', 'darkorchid', 'darkorchid', 'cadetblue']
        X = data['config_name']
        Y = data['avg_return']
        t = type(Y)
        yerr = data['avg_return_stddev']

        fig, ax = plt.subplots() #create figure and axes
        #set the size of the figure
        fig.set_figheight(10)
        fig.set_figwidth(25)
        ax.bar(X, Y, yerr=yerr, align='center', color=color[0:len(X)],␣
↪ecolor='black', capsize=10)
```

```
        #set the labels (including ticks and it's parameter) and title
        ax.set_title(plot_title, fontsize=25)
        ax.set_ylabel(y_axis_title, fontsize=25)
        ax.set_xlabel(x_axis_title, fontsize=25)
        ax.set_xticklabels(X, rotation=45, fontsize=30)
        ax.tick_params(axis='y', labelsize=30)
        fig.tight_layout()
        fig.savefig(f'{timestamp}_avg_return_per_config.png') #save the figure
```

[ ]:
```
def evaluate_training(training_data_path_csv, **kwargs):
    """
    Evaluate the training by plotting the actor and critic losses over time.

    Parameters:
        training_data_path_csv (str): The file path to the training data in CSV␣
↪format.
        plot_title (str, optional): The title of the plot. Default is None.␣
↪Actor or critic will be inserted to title automatically
        x_axis_title (str, optional): The title of the x-axis. Default is None.
        y_axis_title (str, optional): The title of the y-axis. Default is None.

    Returns:
        None
    """
    plot_title = kwargs.get("plot_title", "Losses over Time")
    x_axis_title = kwargs.get("x_axis_title", "Timesteps x 1e5")
    y_axis_title = kwargs.get("y_axis_title", "Loss")
    # Load the training data from the CSV file
    training_data = pd.read_csv(training_data_path_csv)

    plt.figure(figsize=(15,10))     #specify figure size

    # Plot the actor loss over time
    X = np.array(list(range(1000, 1000001, int(1000000/
↪len(training_data['actor_losses'])))))[-len(training_data['actor_losses']):])
    plt.plot(X, training_data['actor_losses'], color = "blue",label='Actor␣
↪Loss')

    # Set labels and title
    plt.xlabel(x_axis_title, fontsize=30)
    plt.ylabel(y_axis_title, fontsize=30)
    #plt.title('Actor Loss over Time', fontsize=30)

    # Visualize a trend line
    x = training_data['Unnamed: 0']
    y = training_data['actor_losses']
    z = np.polyfit(X, y, 1)
```

```
    p = np.poly1d(z)
    plt.plot(X, p(X), "r--", label='Trend Line', linewidth=6)
    plt.xticks(range(0,1000000, 100000), fontsize=30) #set ticks for x-axis␣
↪labels
    plt.ticklabel_format(style='sci', axis='x', scilimits=(0,0)) #format the␣
↪labels in scientific notation
    plt.tick_params(axis='y', labelsize=30) #set ticks for y-axis labels and␣
↪labelsize

    # Add legend
    plt.legend(fontsize=30)
    plt.title('Actor ' + plot_title, fontsize=30)
    plt.tight_layout()

    # Save plot
    timestamp = datetime.datetime.now().strftime("%Y-%m-%d-%H-%M") #create␣
↪unigque timestamp
    filename = f"{training_data_path_csv}_{timestamp}_actor_losses.png"
    plt.savefig(filename)

    # Plot of critic loss
    plt.figure(figsize=(15,10)) #define figure size

    # Plot the critics loss of critic 1 over time
    x1 = training_data['Unnamed: 0']
    y1 = training_data['critic1_losses']
    plt.plot(X, y1, label='Critic loss', color='red')
    plt.xticks(range(0,1000000, 100000), fontsize=30) #set ticks for x-axis␣
↪labels
    plt.tick_params(axis='y', labelsize=30) #set ticks for y-axis labels and␣
↪labelsize
    plt.ticklabel_format(style='sci', axis='x', scilimits=(0,0)) #format the␣
↪labels in scientific notation

    # Plot the critics loss of critic 2 over time
    x2 = training_data['Unnamed: 0']
    y2 = training_data['critic2_losses']
    plt.plot(X, y2, label='Critic loss', color='#1f77ba')
    plt.xticks(range(0,1000000, 100000), fontsize=30)
    plt.tick_params(axis='y', labelsize=30)
    plt.ticklabel_format(style='sci', axis='x', scilimits=(0,0))

    # Calculate and plot trend line for critics losses
    # Critic 2
    slope2, intercept2, _, _, _ = stats.linregress(X, y2)
    plt.plot(X, intercept2 + slope2*X, 'r--', label='Trend line 2', linewidth=6)
```

```
    plt.xticks(range(0,1000000, 100000), fontsize=30)
    plt.ticklabel_format(style='sci', axis='x', scilimits=(0,0))
    plt.tick_params(axis='y', labelsize=30)

    # Critic 1
    slope1, intercept1, _, _, _ = stats.linregress(X, y1)
    plt.plot(X, intercept1 + X*slope1, 'r--', label='Trend line 1', linewidth=6)
    plt.xticks(range(0,1000000, 100000), fontsize=30)
    plt.tick_params(axis='y', labelsize=30)
    plt.ticklabel_format(style='sci', axis='x', scilimits=(0,0))

    # Set labels and title
    plt.xlabel(x_axis_title, fontsize=30)
    plt.ylabel(y_axis_title, fontsize=30)
    plt.title('Critic ' + plot_title, fontsize=30)

    # Add a legend
    plt.legend(fontsize=30)
    plt.tight_layout()

    # Save the plot
    plt.savefig(f"{training_data_path_csv}_{timestamp}_critic_losses.png")
```

Execute the following code to view the training evaluation:

```
[ ]: # Set the path to the csv file to evaluate the training results
training_data_path_csv = '../models/td3_best/td3_results.csv'
evaluate_training(training_data_path_csv)
```

Execute the following code to view the enjoy evaluation:

```
[ ]: #Set the path to the csv file to evaluate the benchmark results
data_path_csv = '../benchmarks/benchmarks_td3_test_hp.csv'
evaluate_enjoy(data_path_csv=data_path_csv)
```

1.6 6. Benchmark

This chapter deals with the development of our TD3 implementation driven by various parameter tests. Furthermore, we will benchmark the implementation to untrained models, basic ground truth models and the stable baselines model.

6.1. Untrained models In order to evaluate the performance of our models, their performance was compared to that of untrained ddpg and td3 agents. The following graph shows how these untrained agents perform in the same environment by carrying out entirely random actions along all joints.

Untrained returns	DDPG	TD3
average return	773	761
standard deviation	167	198

Considering only the empirical implications of these results without considering the qualitative aspects of the actions carried out by these untrained agents, suggests that carrying out entirely random actions seems to be more profitable than attempting to traverse the environment along its x axis as fast as possible.

1.6.1 6.2. Ground Truth

To compare the two algorithms, we've defined a basic parameter set, the so-called Ground Truth.

The below shown graphic gives an overview to the Ground Truth parameters:

6.2.1. Learning phase This chapter compares the learning phase of the ground truth models from DDPG and TD3.

Actor losses over time:

——————————— **DDPG GT** ———————————- |——————————— **TD3 GT** ———————————-

- Delayed policy updates lead to more stable actor training losses.
  - This gets visible by the comparison of the success and precessor actor losses of the two algorithm.
  - The relative difference is smaller in TD3 than in DDPG.

Critics losses over time:

——————————— **DDPG GT** ———————————- |——————————— **TD3 GT** ———————————-

- Double Q learning combats overestimation successfully
  - This gets visible in the critic training losses of TD3. By using the minimum of the two critic losses, the overestimation is tackled.
  - The arithmetic average difference is 0.92% leading to a successful improve.
- Added target policy noise smoothens exploitation of a possible q-function error

During the implementation, we've changed two further points:

- The training phase is now only based on the training steps, instead of a combination of epochs and steps. This ensures that the complete training budget is used.
- Otherwise a early episode stop would decrease the total amount of training steps.
- Our learning starts first after 10.000 training steps. The delayed learning start stabilizes and optimizes the training as the algorithm collected more first experience.

6.2.2. Trained models After a complete training process, you can observe the trained models interacting in the environment as e.g. shown in the videos below: ——————————— **DDPG GT** ———————————- |——————————— **TD3 GT** ———————————-

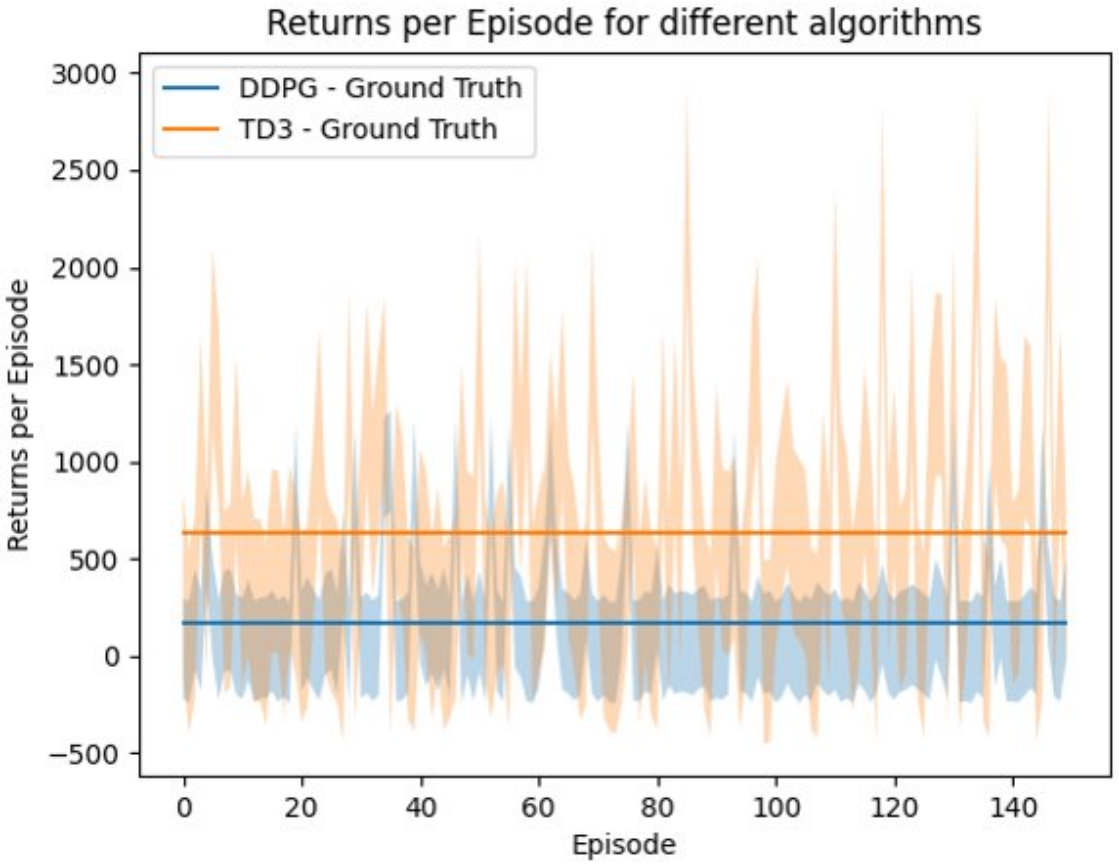
In order to quantify the quality of the trained models, the average return of an episode can be used.

Our benchmarked uses a budget of 150,000 steps and evaluates 150 episodes (with max. 1.000 steps each) in total.

The result of this benchmark can be visualized as shown below:

The performance of the trained models can be quantified by the average return and standard deviation.

We've evaluated the average of 150 episodes, leading to the following result:



	DDPG	TD3
average return	170	631
standard deviation	259	470

Consequently, the applied algorithm changes lead to a remarkably increased performance in the average returns of factor ~3,7 by a standard deviation increase factor of ~1,8.

A comparison between the ground truth and the untrained model results lead to the suggestion, that the selected hyperparameters of the grount truths were suboptimal to maximize returns. It paves the way to carry out more detailed hyperparameters tests in order to improve the performance of our models.

1.6.2 6.3. TD3 Trained model - Hyperparameter tests

Compared to the ground truth, various isolated permutations of the hyperparameters have shown the following result:

A review of the graph suggests that an isolated increase of tau and of the learning rate have the biggest influence.

As the influence of the learning is higher than the one of tau, the following evaluation concentrates on models with a learning rate of 0.001.

Additional permutations of the noise have show that a combination of a higher learning rate and a higher noise lead to a remarkable increase in the average reward of a trained model. Increasing the tau value in this situation led to lower average returns

Building on this, the policy noise was reduced using a negative exponential function over time to provide adequate noise towards the initial and medial stages while ensuring a rapid drop in noise towards the final stages of training. This form of noise reduction led to drastic improvements in the average returns

1.6.3 6.4. Our TD3 Best vs. Stable Baselines

This section deals with the benchmark of our best TD3 compared to the one of stable baselines.

Stable baselines is a set of high-quality implementation of reinforcement learning algorithms in python. It is built on top of OpenAI Gym containing the Ant-v3 environment as well.

6.4.1. Learning phase By using our best TD3 parameter set, we can observe the following training process.

Compared to the ground truth, a similar trend in the losses was observed. It can be noted, that loss reduction for the actor was higher than that of the ground truths. Even the trendline of the critic losses displays a steeper gradient.

6.4.2. Trained models After a complete training process, you can observe the trained models interacting in the environment as e.g. shown in the videos below:

——————————— **TD3 Best** ———————————- |———————————- **Stable Baseline** ———————————-

In order to quantify the quality of the trained models, the average return and standard deviation of an episode can be used.

Our benchmarked uses a budget of 150,000 steps and evaluates 150 episodes (with max. 1.000 steps each) in total.

The result of this benchmark can be visualized as shown below:

Model	Average return	Standard deviation
DDPG Ground Truth	170	259
TD3 Ground Truth	631	470
TD3 Best model	5354	767
TD3 Stable Baselines	5813	589

A comparision of the performance of our best model with that from stable baselines shows, that our average returns lays behind by a margin of less than 500. Stable baselines refers to a library of pre-trained reinforcement learning models which can be used as benchmarks. The implementation of these algorithms is based off of the guidlines set by OpenAi.

Findings: - TD3 Ground Truth performs better than DDPG Ground Trouth - The tweaking of hyperparameters has an remarkable influence to the results of the algorithm. - Significant improvements within the basic algorithm are possible - TD3 Best model reaches nearly the performance of the stable baselines one - Reward shaping of healthy z-range would lead to increased returns and decreased standard deviations

---

1.7 7. Summary

- In general, TD3 achieved better results than DDPG through simple changes.
- The change of the training budget implementation (training steps only) enabled a complete usage of the training budget.
- Overall, a reduction in the TD3 training time (compared to DDPG) was noticed.
- While individual components contribute little to improvement(Fujimoto, et al. (2018)), the combination of all three proved to be a performance accelarator.
- Without any training, the untrained TD3 and DDPG model performed better than our ground truth trained models.
- Subsequent parameter tests proved that our ground truth parameters had been suboptimal.
- Thereby, a higher learning rate and target policy action noise with reduction over time greatly improved results.
- The result of our best TD3 model is slightly lower than the one of stable baseline. Considering the standard deviation, the perfomance of our best TD3 model is comparable to stable baselines.

Future potential: - Further in depth studies for TD3 parameter permutations could yield even better results. - More in depth studies for parameter permutations of DDPG could confirm sensitivity reduction as observed in other studies. - Our implementation could benefit from an optimization for replay buffer and parallel training to accelerate the training process.

1.8 8. Further material

Additional information can be found via the below mentioned links.

GitHub Repos: - TD3-RLE: https://github.com/eshan-savla/RLE-TD3.git - Stable Baselines: https://github.com/openai/baselines - DLR - RL Baselines Zoo: https://github.com/DLR-RM/rl-baselines3-zoo

Environment from MuJoCo: - Ant-v3: https://www.gymlibrary.dev/environments/mujoco/ant/

Useful information: - OpenAI Spinning Up: https://spinningup.openai.com/en/latest/