Coding Guidelines

Leif Döring
Daniel Schmidt
Benedikt Wille

Universität Mannheim

Inhaltsverzeichnis

Kapitel 1

Git and GitHub

Git is what's known as a version control system. It helps us track and manage changes to files and directories (i.e., folders). Unlike your regular system of folders and files on a laptop, Git doesn't just store the current state of all files, it keeps a complete history of changes, allowing us to return to any previous version and see exactly what has been modified. You can think of it as similar to Word's version history feature, except Git does this for all files in a folder at once. When Git has been activated for a folder (more on that in a moment), we call it a Git repository. Git is an open-source tool and completely free to use.

Git becomes especially useful, and is most commonly used, in combination with GitHub. GitHub is a web-based platform that hosts Git repositories and adds extra features for collaboration. A repository (or "repo" for short) is simply a folder managed by Git. On GitHub, we can store our repositories online, share them, and work on them together with others. When multiple people work on the same repository, all changes are stored centrally online and can be retrieved by everyone with a single command as soon as someone saves them. This prevents conflicts from simultaneous changes and eliminates the need to constantly exchange individual files or manually incorporate edits that others have made.

1.1 Setting up Git and GitHub

First, we need to install Git locally and create a GitHub account.

GitHub Account: Go to die Homepage von GitHub and create an account if you don't already have one. Remember your username and password, as you will need both later to connect Git with your GitHub account.

Git installieren: On newer Macs, Git is usually preinstalled. Open a terminal and enter the command

```
git --version
```

If a version number is displayed, Git is already installed. If not, you can download Git from this link and install it by following the instructions on the website.

Configuring Git: Once Git is installed, you need to link it to your GitHub account. Enter the following commands in your terminal, replacing `your_username` and `your_email@example.com` with your actual data (the email address and username used when registering on GitHub):

```
git config --global user.name "your_username"
git config --global user.email "your_email@example.com"
```

There are several ways to avoid having to enter your username and password each time you push changes to GitHub. The simplest way, and the one we recommend, is to sign in to VS Code using your GitHub account. To do this, open VS Code, click on the Source Control icon (the symbol with three connected dots, which we will be using frequently), and then click on „Sign in to GitHub“. Follow the on-screen instructions to log in. Alternatively, you can create a so-called „Personal Access Token“ (PAT), which works as a password, or use an „SSH key“. Both methods are slightly more complicated to set up, but there are many tutorials online to guide you through the process, and LLMs are generally quite effective at helping with that.

1.2 The Most Important Git Commands

We will now go through the most important Git commands step by step to give you an overview of what you can do with Git. Many of these commands we won't use again later, since VSCode already integrates most functions with a single click. However, it's still good to know the commands in case something doesn't work as expected.

Creating a new repository: Git doesn't automatically track all files on your computer, but only selected folders that become repositories. We can either turn a completely new local folder into a Git repository or download an existing repository from GitHub.

```
git init
```

The command `git init` initializes a new Git repository in the current directory. This means Git will now track all changes to files in this folder. You can later upload this repository to GitHub to store it online and share it with others. However, we won't focus much on this option, as we will typically clone existing repositories from GitHub. For more details, refer to LLMs or online guides.

```
git clone <repository_url>
```

With the command `git clone <repository_url>`, we can download (i.e., clone) an existing repository from GitHub (or another Git server) to our local computer. Replace `<repository_url>` with the URL of the repository you want to clone. Note: the cloned folder will be downloaded into the folder where your terminal is currently located. So, first navigate to the folder where you want to store the repository using `cd <path/to/folder_name>`. If you're unsure which folders you have access to, use `ls` (list) to display all folders and files in the current directory. If needed, you can always move up one folder with `cd ..`.

As a small exercise, create a test repository on your GitHub account and then download it to your computer using the `git clone` command. Go to GitHub, click on „New Repository“, give the repository a name (e.g., `TestRepo`) and click on „Create Repository“. Copy the repository's URL (the one you see in your browser's address bar or find via the green „Code“ button) and use it in the terminal with the `git clone` command. It's best to do this in VSCode, so you can immediately view and edit the downloaded repository.

Tracking and saving changes: After creating or cloning a repository, we can make changes to the files. Git tracks these changes, and VSCode will also show them, but they are not yet saved (committed).

As an exercise, add two files to your test repository so we can try out the basic Git commands. Once you've created the files (assuming the repository is already connected to Git), VSCode will show a green „U“next to the filenames, indicating that the files are not yet committed (saved).

```
git add <datei_name1> <datei_name2>
```

To save files, we first need to specify which files we want to save (this is called „adding to the stage“). We do this with the command `git add <file_name1> <file_name2> ...`. Replace `<file_name1>` and `<file_name2>` with the names of the files. If files are in subfolders, include the path (e.g., `folder/file.txt`). Alternatively, you can use `git add .` to add all changed files in the current directory and all subdirectories.

```
git commit -m "Beschreibung der Änderungen"
```

To actually save all staged (selected for saving) changes, we use the command `git commit -m „Description of changes"`. Replace `„Description of changes"` with a short description of what you changed. This description helps you and others later understand what was modified in this commit (the save operation).

```
git push
```

Now your changes are saved locally, but not yet uploaded to GitHub. To upload them (push), use the command `git push`. The first time you push, you may be prompted for your GitHub username and password (or Personal Access Token if you've set one up). After the first time, Git should remember your credentials so you don't have to enter them every time.
Now you know the typical Git workflow: make changes, add files to the stage, commit the changes, and finally push them to GitHub.

```
git pull
git fetch
```

When working with others, it's possible that someone else has made changes to the repository and pushed them to GitHub after you first downloaded it. To ensure you have the latest changes before pushing your own, use `git pull` or `git fetch`. These commands download the latest changes from GitHub. `git pull` downloads the changes and integrates them directly into your local copy of the repository, while `git fetch` only downloads them without integrating. Usually, `git pull` is simpler if you're sure there are no conflicts (i.e., someone else modified the same line of code differently from you). This way, you don't have to manually merge the changes afterward (see LLMs/online guides for that). We'll learn how to resolve potential conflicts and merge different commits soon.

If you've now saved your changes, try what happens when you delete one of the files locally and modify the other by adding something to it:

Switch to the menu with the three connected dots (the Git view) in VSCode. There, VSCode will show a red „D“ and a light yellow „M“ next to the filenames, indicating that one file is deleted and the other modified. Generally, you'll find an overview of all changes you've made here. You have a variety of options to use Git directly without opening the terminal.

Hovering over a changed file's name from left to right: Open file (opens the file in two side-by-side views, one showing the original version and the other the modified one), Discard changes (reverts the file to its original state), and Stage changes (marks the file for commit).

Additionally, above the blue button, you have several options to skip steps without writing any Git code in the terminal. In any case, you can write a commit message in the text field (if you don't, you'll have to manually add it to the tracking file in the next step): Commit (commits all staged changes, or all changes if nothing is staged), Commit and push (same, but uploads changes to GitHub), and Commit and Sync (same, but first downloads and integrates the latest changes from GitHub). If you have no changes, and there are changes on GitHub, you'll see a message, and you can run `git pull` via the blue button.

Try committing all the changes you've made (but don't push to GitHub yet). Now look at the history in the bottom menu. There, you can see all your commits and those from others, including

who made them, as well as where you are (the blue marker) and the version stored on GitHub (the purple marker). You should see that you're one commit ahead of GitHub, and when you push your changes, you'll see the purple marker move forward.

You can also undo changes after they've been committed by you or others. If it comes to that, just ask your LLM of choice for help.

Branches and Merging Changes: In Git, we can create so-called branches to develop different versions of our code without affecting the main version (the main „or master" branch). This is especially useful when multiple people are working on the same project or when we want to test new features without risking the stable code.

```
git branch (<branch_name>)
git checkout <branch_name>
```

With the command `git branch <branch_name>`, we can create a new branch (without specifying a branch name, the command simply lists all current branches). Replace `<branch_name>` with the name of the new branch. With the command `git checkout <branch_name>`, we switch (check it out) to the newly created branch, so all changes we make are saved only in this branch. When a new branch is created, it is based by default on the current branch you are in. This means the new branch contains a copy of the code and history from the current branch at the time of creation. In VSCode's Git menu, the new branch is shown as a branch off the main branch (which the menu logo also represents) and shows you where you currently are.

```
git merge <branch_name>
```

If you are satisfied with the changes in your branch and want to integrate them into the main branch, you can use the command `git merge <branch_name>`. Replace `<branch_name>` with the name of the branch you want to merge. Make sure you are on the main branch before executing the merge command, because the merge integrates the changes from the specified branch into the current branch.

If there are conflicts (i.e., if the same code line was changed differently in both branches), Git will notify you and you must resolve the conflicts manually. VSCode provides a user-friendly interface to see the differences and decide which changes to keep. You can click above each change to select which of the two versions you want to use to create a merged version.

The same applies when you use `git pull` and there are conflicts, because the version on GitHub is then merged with your current branch. If conflicts occur here, you can also resolve them manually in VSCode. However, there is also the option to let it resolve automatically, which usually works well. For this, there are three configurations that you can enter in the command line beforehand (or after a failed pull before the next pull). The terminal will automatically show you the common possible commands when you enter `git pull` and there are conflicts. Otherwise, you can also ask the LLM of your choice.

Other Useful Commands and GitHub Features:

GitHub offers many more useful features that make collaboration easier. I recommend taking some time to explore by clicking around, perhaps in dialog with an LLM. For example, you can create Issues to track bugs or tasks, create Pull Requests to suggest and discuss changes, and much more.

An important feature is that you can „fork" other people's repositories. This means creating a copy of someone else's repository in your own GitHub account. You can then make changes to this copy without affecting the original. If you think your changes are useful, you can create a Pull Request to ask the original repository owners to incorporate your changes. You can also incorporate changes from the original repository by setting up a so-called „upstream" and

regularly pulling the latest changes from there into your forked repository. Your LLM of choice can definitely help you with that as well.

```
git rebase <branch_name>
```

Rebase is another way to integrate changes from one branch into another. Unlike merge, which preserves the history of both branches and creates a new commit to combine the changes, rebase takes the changes from the specified branch and „replays“ them on the current branch. This results in a straighter and cleaner history, as it appears as if all changes were made sequentially. However, rebase can alter the history, which can be problematic if multiple people are working on the same branch. Therefore, rebase is often used for local branches that haven't yet been shared with others.

Otherwise, Git and GitHub offer many more features that can be used as needed. It's worth reading the documentation or watching tutorials to gain a deeper understanding.

Kapitel 2

How do (Python-)Repositories work?

Now we have learned how to work with Git and GitHub. The next step is to understand how (Python) repositories are structured and how you can use them effectively. To do this, we will first take a closer look at how moduling works in Python and how you can use it to work in repositories, and then examine what kinds of other files are typically found in repositories, what purpose they serve, and how they function.

2.1 Python Moduling

PYTHONPATH Python uses an environment variable called `PYTHONPATH` to know where to look for modules you want to import. This variable contains a list of directories where Python searches for modules. When you import a module, Python searches these directories in the order they are listed in `PYTHONPATH`. When you first install Python on your computer, you essentially download a folder whose subfolders contain the Python base version and a folder called `Lib`, where all standard modules and installed packages are stored as folders. The `PYTHONPATH` variable automatically includes both the folder where the Python version is located and the folder containing the modules. If you now open a folder in VSCode or the terminal and run a Python file from there, the folder containing the file is automatically added to `PYTHONPATH`. To display which folders are currently in `PYTHONPATH`, enter the command `echo $PYTHONPATH` in the terminal. To find out where your current Python installation is located, use `which python`.

```
import <modul_name>
from <modul_name> import <funktion_name> <classe_name> <object_name> ...
import <upper_folder>.<lower_folder>.<modul_name>
from <upper_folder>.<lower_folder>.<modul_name> import ...
```

If you want to use modules (i.e., folders with Python code) or functions from modules (i.e., files) in your Python code, you can import them using the `import` statement. Replace `<module_name>` with the name of the module you want to import. If the module is in a subfolder, specify the path to the module separated by dots (e.g., `from folder.subfolder.module import function`). You can also import only specific functions, classes, or objects from a module using `from <module_name> import <function_name> <class_name> <object_name> ...`, instead of importing the entire module. This can make your code significantly faster in some cases.

Python will search for them in all folders defined in `PYTHONPATH`. So make sure to always specify the path to the module relative to `PYTHONPATH`. If `<module_name>` exists neither as a folder nor as a .py file in one of these folders, Python will display an error that the module could not be found.

You need to pay particular attention to this when working in a repository with multiple subfolders. For example, if you run a file from one subfolder and try to import a module from another subfolder, the module generally won't be found, because the current `PYTHONPATH` only includes the subfolder but not other subfolders in the same repository. In such cases, you either need to adjust `PYTHONPATH` (by adding the folder containing the module) or use relative imports (by specifying the path to the module relative to the current directory). You can move up one level using `..` before the module name or path.

By default, if you want to use your own code in a repository rather than pre-made modules, you always need to specify the path to a .py file you want to import from. You can bypass this by creating a file named `__init__.py` in the folder containing the .py file. This file can be empty, but its presence tells Python to treat the folder as a module. Afterward, you can use the folder name as the module name and import functions, classes, or objects from it. In this file, you can also set global configurations for the module, such as which functions or classes should be imported by default when the module is imported, write initialization code that runs when the module is imported, or define global variables available throughout the module.

Virtual Environments When working with Python, it's often useful to use so-called virtual environments. A virtual environment is an isolated space on your computer where you can manage your own Python installation and packages without conflicting with the global Python installation on your computer. This is especially helpful when working on multiple projects that require different Python versions or packages. The `PYTHONPATH` is automatically adjusted so that only packages and modules in the virtual environment are available. When you create a virtual environment in VSCode, it will ask if you want to use the virtual environment as the interpreter for the current project. Confirm this to ensure you use the packages and modules from the virtual environment. After activation (more on that soon), you can check with `which python` if you're using the correct interpreter. The path shown should point to the virtual environment.

Before creating a virtual environment, you should install pyenv. pyenv is a tool for managing different Python versions on your computer. On Mac, you can do this easily with Homebrew by entering `brew install pyenv` in the terminal or following the guides on this page. For Windows, there is no „official“ version of pyenv, but here is an unofficial version with the same features. But always remember it's not 100% the same. Then follow the instructions on the pyenv GitHub page to set up pyenv.

```
pyenv install <python_version>
pyenv global <python_version>
pyenv local <python_version>
```

With the commands above, you can install any Python version you need. Replace `<python_version>` with the desired version (e.g., `3.9.7`). With `pyenv global <python_version>`, you can set the global Python version used on your computer. With `pyenv local <python_version>`, you can set the Python version for the current directory where your terminal is located when you run the command. This is especially useful for projects requiring a specific Python version, as we'll see later.

```
python -m venv <env_name>
source <env_name>/bin/activate
deactivate
```

To create a virtual environment, use the command `python -m venv <env_name>`. Replace `<env_name>` with the name you want for the virtual environment (e.g., `venv`). This creates a new folder with the virtual environment's name in the current directory, containing the isolated Python installation and packages. VSCode will automatically ask if you want to use the virtual environment as the interpreter for the current project. Confirm this to ensure you use the packages and modules from the virtual environment. Normally, VSCode also automatically activates the

virtual environment in your terminal, and you'll see the environment name in parentheses before the prompt. If not, you can manually activate it with `source <env_name>/bin/activate`. When the virtual environment is active, its name appears in parentheses before the terminal prompt. To deactivate the virtual environment and return to the global Python installation, enter `deactivate`.

Installing Packages In Python, additional functions and tools are often provided in so-called packages. Packages are collections of modules offering specific functionalities. We typically use the package manager `pip`, which comes standard with Python, to install them.

```
pip install <package_name> <version_specifier>
pip install -r requirements.txt
```

You can install packages manually with `pip install <package_name> <version_specifier>`, specifying a particular version (using ==, <, >, <=, >=). Replace `<package_name>` with the package name and `<version_specifier>` with an optional version (e.g., `==1.2.3` for a specific version or `>=1.2.3` for a minimum version). If you omit the version, the latest version is installed automatically. Alternatively, if your project has a file named `requirements.txt` (common in most repositories), you can install all listed packages with `pip install -r requirements.txt`. This file lists the packages and versions required for the project.

2.2 Special Files in Repositories

We will mainly focus here on files you will find in the Stable Baselines repositories. You won't necessarily need all of them, but it helps to know them and understand what they are for. Of course, there are many other files that can be used in repositories, but this includes the absolute standard files that exist.

.gitkeep If you create an empty folder in a repository, you'll quickly notice that Git doesn't track (i.e., save) this folder. This is because Git doesn't track empty folders by default. However, if you want to keep an empty folder in your repository (e.g., because you plan to store files in it later), you can create a file named `.gitkeep` in this folder. This file can be empty, but its existence signals to Git that the folder is not empty and should therefore be tracked.

.gitignore In many repositories, you'll find a file named `.gitignore`. This file contains a list of files and folders that Git should ignore. This is useful for excluding temporary files, log files, result files, or other files that shouldn't be versioned from the repository. This keeps the repository clean and organized, and you don't waste storage space on GitHub and on other people's computers working with the repository (plus, it's faster when Git doesn't have to track unnecessary files). The syntax in the .gitignore file is relatively simple: Each line contains a pattern applied to files or folders. For example, the entry `*.log` ignores all files with the .log extension, and the entry `temp/` ignores the temp folder and all its contents. LLMs can help you with this as well.

.python-version If you've created a virtual environment with pyenv, the file `.python-version` is automatically created. This file contains the Python version that should be used for the project. When working in a directory containing this file, you should also use the specified Python version. pyenv automatically activates the specified Python version when you use the `pyenv local` command. This ensures you always use the correct Python version for the project without having to switch it manually.

.readthedocs.yaml This file is used to configure the documentation for the project when using Read the Docs (a website for hosting repository guides). In this file, you can make various settings, such as which Python version to use, which dependencies need to be installed, and how the documentation should be built.

LICENSE This file specifies the license for the project. The license determines how the code may be used, modified, and distributed. It's important to understand the license before using or modifying the code in your own project. GitHub provides various standard licenses designed for open-source work and displays the license on the main page of projects, among other things. The Stable Baselines repositories use the MIT license, which is very permissive and allows you to use the code with almost no restrictions, but also without warranties.

Makefile A Makefile is a file containing instructions for the `make` tool. Make is a build automation tool originally developed for compiling programs in C and C++, but it can also be used for other tasks. In a Makefile, you can define various "targets" that execute specific tasks when you enter the command `make <target>` in the terminal. For example, a target named `install` might contain instructions to install all dependencies, and a target named `test` might contain instructions to run all tests. Makefiles are especially useful for automating repetitive tasks and ensuring they are always executed the same way. Stable Baselines has a Makefile with some commands, but you probably won't really need it.

pyproject.toml This file is a relatively new way to define metadata and dependencies for Python projects. It is used by various tools to obtain information about the project, such as which Python version to use, which dependencies need to be installed, and how the project should be built. In the Stable Baselines repositories, the file is used to define the project's dependencies. One tool you can use this file with is `poetry`. However, we will install the dependencies differently later.

README.md The `README.md` file is a very important file in every repository. It should contain a description of the project, installation and usage instructions, and other important information. The file is written in so-called Markdown format, which provides a simple way to format text and add links, images, and other elements (see LLM for exact functionality). When you open a repository on GitHub, the contents of the README file are automatically displayed on the repository's main page. It's good practice to create a detailed README file so that other people (and yourself in the future) can understand what the project is about and how it can be used.

requirements.txt This file contains a list of packages and their versions required for the project. You can install all packages listed in this file with the command `pip install -r requirements.txt`. This is especially useful when working on a project with many dependencies, as you can install all required packages with a single command. If you've already created a repository that doesn't have a requirements.txt file yet, you can create one yourself by entering `pip freeze > requirements.txt`. This creates a file listing all currently installed packages and their versions in your (virtual) environment.

.sh files Files with the `.sh` extension are shell scripts containing instructions that can be executed in a Unix shell (like Bash; basically like your terminal on the computer). These scripts can be used to automate repetitive tasks, such as setting up an environment, running tests, or starting a program. When we later compute on the cluster, we will need to use shell scripts to submit them to the queue and can no longer simply run our code with the green triangle in the top right of VSCode as usual. The syntax is more or less like in the command line, but there are some differences and additional features that can be used in shell scripts. When we deal with computing on clusters, we will only learn the absolute basics we need. LLMs are referenced here as well, but also the MIT Open-Source Guide from the beginning of the chapter.

Kapitel 3

The Stable Baselines3 Zoo Repository

The core idea of the Stable Baselines repositories is to provide uniform implementations of algorithms as well as an organized way to test and compare them. Particularly in Deep RL (RL with neural networks instead of tabular methods), there are a large number of hyperparameters and different implementation approaches make a real difference. There are three repositories belonging to Stable Baselines. The first two are Python packages containing implementations of basic algorithms (Stable Baselines 3), as well as algorithms contributed later by other authors (sb3 contrib) that were reviewed by the Stable Baselines authors. The third repository, Stable Baselines 3 Zoo, contains functions to run the algorithms with pre-configured hyperparameters, generate performance plots, and much more. Since it will be part of the exercises to explore the Stable Baselines3 repository, we will focus on the Zoo. We will not discuss all features of the Zoo but you can find an extensive guide on the GitHub repository in the README and/or readthedocs documentation.

3.1 Basic Functions for Training, Testing, and Plotting

train.py In the `rl_zoo3` folder, there is a file called `train.py`. This file contains the `train` function, which is used to train algorithms. Since there is an initialization file named `train.py` directly in the repository that calls this function, you can execute the function by entering the command `python train.py` in the terminal.

The function accepts various arguments that you can specify in the terminal using so-called "flags". Flags are essentially names for parameters that you can pass. There are short flags (with a hyphen, e.g., `-f`) and long flags (with two hyphens, e.g., `--flag`). When you enter a flag, the value that follows the flag is used as the value for that parameter. For example, if you enter the command `python train.py -algo ppo`, the value `ppo` will be used as the value for the `-algo` flag.

You can check the `train.py` file in the `rl_zoo3` folder to see which flags you can use and what they mean. In the README.md file under the section „Train an Agent“ you can find an example of how to use the function. If you try this out, at the beginning of the output, you will see an overview of the used parameters, and as soon as the training starts, you will see intermediate results such as the observed score. The most important flags of the `train` function are:

- `--algo`: The algorithm to be used (e.g., `ppo`, `a2c`, `dqn`, etc.).
- `--env`: The environment in which the algorithm should be trained (e.g., `CartPole-v1`, `MountainCar-v0`, etc.).

- `--n-timesteps/-n`: The number of time steps the algorithm should be trained for.
- `--log-interval`: The interval (in episodes) at which the training progress should be logged. This is the information you see in the terminal when you run the code.
- `--eval-freq`: The interval (in time steps) at which the algorithm should be evaluated.
- `--eval-episodes`: The number of episodes to be used for the evaluation.
- `--n-eval-envs`: The number of environments to be used for the evaluation.
- `-f`: The path to the directory where the training and evaluation models should be saved. By default, `./logs` is set, but we typically want to use subdirectories sorted by project.
- `--seed`: The random seed to be used to make the results reproducible.
- `-optimize`: If this flag is set, the algorithm's hyperparameters will be optimized before the training starts. More on this later.
- `-params/--hyperparams`: The hyperparameters to be used for the training. You can either specify the name of a JSON file containing the hyperparameters (which you probably won't need) or you can directly specify individual hyperparameters in the terminal (e.g., `--hyperparams learning_rate=0.001 batch_size=64`).

The `train` function saves your data in the following folder structure:

```
Storage_Path/
└── Algo_Name/
    └── Env_Name_Number/
        ├── Env_Name/
        │   ├── args.yml
        │   ├── command.txt
        │   └── config.yml
        ├── Number.monitor.csv
        ├── Env_Name.zip
        ├── best_model.zip
        └── evaluations.npz
```

The `args` and `configs` files contain the used parameters in the YAML file format (Python-native, entries with the right syntax can be more easily converted back into Python objects), and `command.txt` contains the command you entered into the terminal. The individual monitor files contain values observed during the evaluation. Atari environments are evaluated on 8 parallel environments by default, so 8 monitor files in CSV format are output. Each of these files contains the time at which the monitoring started and the ID of the environment, as well as triples of rewards (r), episode lengths (l), and the time (t) used for monitoring. The ZIP files contain the model at the end of the training and the best model seen during the training. These files are very large because we are working with neural networks. Therefore, you should preferably delete them if you don't need them anymore (if you want to run an agent on an environment to observe how it behaves you need the network). The `npz` file contains the aggregated results of the metrics you have logged. More on how to log additional metrics besides the standard scores you receive at the evaluation time will come later.

enjoy.py This function is also located in the `rl_zoo3` folder and has an initialization file named `enjoy.py` that calls the function directly. With this function, you can load a pre-trained model and let it run in the specified environment, observing how your agent behaves. This is useful to see how well the model performs after training. The function also accepts various flags that you can specify in the terminal.

The most important flags of the `enjoy` function are:

- `--env`: The environment in which the model should be executed (e.g., `CartPole-v1`, `MountainCar-v0`, etc.).
- `-f`: The path to the directory where the trained model is saved. If there are multiple models, the latest one will be used automatically. However, you can also specify a specific ID that uniquely identifies the model.
- `--algo`: The algorithm used to train the model (e.g., `ppo`, `a2c`, `dqn`, etc.).
- `--n-timesteps`: The number of episodes the model should execute.
- `--seed`: The random seed to be used to make the results reproducible.

The plotting functions are located in the `rl_zoo3/plots` folder and do not have initialization files. Therefore, you always have to specify the full path when you want to use `python rl_zoo3/plots/<name.py>` in the terminal.

all_plots.py This plotting function aggregates the data from all runs in a given directory, sorts them by algorithms, environments, and metrics, and saves them collectively. It is important that the folder you provide as input has the correct structure. The `train.py` function already saves the correct structure, so it's best to provide the path that you also passed to the `train` function for a project. Note that all runs found in the provided path will be taken, so you need to make sure that no unwanted runs are included. The results are retrieved from the `evaluations.npz` file in the respective run folders. The most important flags are:

- `--algos/-a`: The algorithms whose data should be aggregated.
- `--env/-e`: The environments whose data should be aggregated.
- `--exp-folders/-f`: The folders in which to search for the data to be aggregated.
- `--max-timesteps/-max` and `-min-timesteps/-min`: The maximum and minimum time steps an experiment must have run to be included. This is particularly useful for filtering out aborted experiments.
- `--output/-o`: The path (including the name of the file to be created) where the output file should be saved. Note that the path must already exist, unlike the path in the `train` function.

In addition to the `.pkl` file that is saved and contains the aggregated data, the `all_plots` function also outputs a plot with the means and confidence intervals of all the algorithms found for each environment, but this must be saved separately.

plot_from_file.py This plotting function uses the data aggregated by the `all_plots` function to create additional plots. It is important to note that by default, all data is used, so you have to exclude certain algorithms, environments, or logged metrics that you don't want to plot using flags. There is also an option to create various plots using the RLiable package. The functions of the package will be explained in the next section. The most important flags are:

- `--input/-i`: The path to the `.pkl` file containing the aggregated data to be plotted.
- `--output/-o`: The path (including the name of the file to be created) where the output plot should be saved. Note that the path must already exist, unlike the path in the `train` function.
- `--labels/-l`: The labels to be used for the curves. Note that the order of the plotted curves corresponds to the order set by the `--algos` flag of the `all_plots` function.
- `--boxplot/-b`: Outputs an additional boxplot on the sensitivity of the data (see explanation of RLiable).

The `plot_from_file` function outputs a bar chart with confidence intervals on the scores of the algorithms on the environments by default, which is also saved. It also outputs the same plots as `all_plots` (without saving them). Additionally, plots from the RLiable package can be created (which are also not saved).

plot_train.py This plotting function uses the data from the monitor files to show the agent's performance during training. The most important flags are:

- `--algo/-a`: The algorithm whose training performance should be plotted.
- `--env/-e`: The environments to be included.
- `--exp-folder/-f`: The folders from which files should be included.
- `--x-axis/-x`: The choice to plot by steps, episodes, or time.
- `--y-axis/-y`: The choice to plot success, rewards, or episode length.

The `plot_train` function outputs the specified plot for all selected environments, but does not save anything.

It's worth going through all these functions once and looking at the options you can specify. Most flags contain an explanation that immediately shows what they do. The flags are defined at the beginning of the function, where you would otherwise find their parameters.

3.2 Further relevant files and folders

Hyperparameter files In the `Hyperparameter` folder, there are `.yml` files in which the hyperparameters that are used by default are defined. Stable Baselines provides tuned parameters for many environments, as well as standard parameters that are suitable for benchmarks. You can have the LLM explain the syntax to you as usual.

Other files in the `rl_zoo3` folder There are also several other files in the `rl_zoo3` folder. From top to bottom, they do the following:

- `__init__.py`: Defines which objects are known. You most likely never have to touch this.
- `benchmark.py`: Here you can run the Stable Baselines3 benchmark completely and later include your algorithms. However, due to limited computing power, you will most likely not use this.
- `callbacks.py`: Regulates the switch between the evaluation and training loop. You most likely never have to touch this.
- `cli.py`: Command line interface, explains what the model "knows". You most likely never have to touch this.
- `exp_manager.py`: Creates folders, so-called Tensorboards (which we won't do), parallelizes processes, checks, sets, and orders configurations, saves files. You most likely never have to touch this.
- `gym_patches.py`: Fixes some incompatible issues with the gym package. You never have to touch this.
- `hyperparams_opt.py`: Here you can define how the optimizer, which you can use in the `train` function, randomly draws parameters and which ones it should draw randomly. This will be part of the RL 2 course.
- `import_envs.py`: Imports additional environments if you specify in the `train` function which ones you want to have. You will likely not work with this.

- `load/push_to_hub`: Allows uploading to hubs that collect and store models, or downloading models. Stable Baselines, for example, provides a hub with many trained models. You will likely not work with this.
- `py.typed`: Is brought for type recognition. You never have to touch this.
- `record_training/video.py`: With this, you can create and save videos of your agents during training. `record_video` is used when you want to use it in combination with benchmarking or parallel environments. You will likely not work with this.
- `utils.py`: Various tools. If you want to implement your own algorithms you will work with this.
- `version.txt`: Current version. You never have to touch this.
- `wrappers.py`: Contains so-called wrappers that define standard configurations for environments. You most likely don't have to touch this.

3.3 Adding Your Own Algorithms

Most of the time, when you want to add a new algorithm, it will be a version of an already existing one. Thus, you can simply take code from one of the Stable Baselines algorithms. Now, we will explain what you need to do in this case in order to make everything work smoothly.

Initial steps The first time you create a new algorithm in this way, you need to go to the `rl_zoo3` folder and create a subfolder, in which you collect your algorithms. You may call it `custom_algos`. Then, create a `__init__.py` file inside the folder and create an empty list titled `__all__`. Furthermore, in the file `utils.py` inside the `rl_zoo3` folder, under the import statements, insert the following line:

```
from rl_zoo3.custom_algos import (
# This comment is here so you do not get an error initially,
# later the names of the algos will be here
)
```

Create/Copy Folder and Set Name If you followed the first step, in the `rl_zoo3` folder, there is now a folder called `custom_algos`, into which you will copy the folders you need for the algorithms. For this, find the subfolder of the algorithm you want to base your algorithm on. For example, if you based the algorithm on the PPO algorithm you can find the code under `stable_baselines3/ppo` on the path `venv/lib` and copy the whole folder to `custom_algos`. Then, change the name to the name of the new algorithm (lowercase) and in the folder, change the file that is called `ppo.py` to the name of the new algorithm. The `policies.py` file imports standard policies from the `stable_baselines` package. If you later want to adapt the policies (which in this course you likely will not), you have to get the code from the respective files. Fortunately, the path to the files is directly displayed to you! In the file that should now bear the name of your algorithm, you have to change the name 7 times, where it currently says `PPO`. Three times in the line starting with `Self` at the very top, once in the class definition right below it, and three times at the very end of the file in the `learn` function. In all three cases, you have to use uppercase. Finally, you have to change the paths in the `__init__.py` file that you need to import the objects. You do this by inserting the following lines instead:

```
from rl_zoo.custom_algos.<algo_name>.policies import CnnPolicy, ...
rl_zoo.custom_algos.<algo_name>.<algo_name> import <ALGONAME>
```

and in the `__all__` list, replace `PPO` with the name of your algo.

Mandatory Changes First, you have to import the objects from the newly inserted folder in the `__init__.py` file of the `custom_algos` folder. You can do this by writing

```
from .<algo_name> import <ALGONAME>
```

at the very top of the file and adding the name of your algo to the `__all__` list. This is necessary to make the objects available when you import the module. Next, you have to import the algorithms in the `utils.py` file. For this, you have to write the name of your algo in the list you created at the beginning. Don't forget the comma afterwards and note that there is only a placeholder there at the beginning. Furthermore, you have to map the lowercase to the uppercase in the `ALGOS` dictionary right below. Finally, you have to add a `.yml` file in the `hyperparams` folder that bears the name of your algo in lowercase. You may simply copy the file of the algorithm you based your algorithm on and change the hyperparameters as you wish. Just remember to give the new file the name of you algo in lowercase.

Optional Changes The following changes are only necessary if you want to use certain functions correctly with your algos.

Stable Baselines3 distinguishes offline and online algos as classes (caution: offline and online are defined somewhat differently than in the lecture). Some functions have to behave differently for off-policy algos. Therefore, there are lists called `off_policy_algos` in the files `benchmark.py`, `enjoy.py`, and `record_video.py`, in which you have to enter your new algo to use their functions. You can see in the class definition of your new algorithm, to which of the classes it belongs.

Furthermore, if you want to use the optimizer in the `train` function, you also have to define in the `hyperparams_opt.py` file which parameters the optimizer should try out and in what way. For the superclasses `Onpolicy` and `Offpolicyalgorithms`, there are converter functions available for some hyperparameters they all have in common and for which you cannot simply pass a value. Moreover, each of the pre-implemented algorithms has its own function called `sample_<algo_name>_params`, in which with the `trial` method it is defined how the parameters should be drawn (e.g. `trial.suggest_float(«parameter_name>", <lower_value>, <upper_value>)` returns a float number between `lower_value` and `upper_value`). You then have to assign the name of your new algorithm (in lowercase) to the sampler or converter function you want to use in the dictionaries `HYPERPARAMS_SAMPLER` and `HYPERPARAMS_CONVERTER`.

Testing If you have now followed all the steps, you can test whether your algorithm is running and doing what it should. You can, for example, use the command `python train.py <new_algo_name>` to see if it starts or if errors occur right at the beginning. You should also pass the argument `--n-timesteps 1000` to see if there are any errors occuring in-between or if your algorithm runs until the end. In general, it is recommended, if your code is based on an algo, especially if this is a special case of your new algo, to compare the results of your new code with the original algo to avoid small implementation errors.

At this point, a small note: In the same manner, you can create a `custom_envs` folder, which is intended for storing new environments. You will very likely not use this function, but if you want to in the future, the integration works quite similarly to the procedure described above.