Reinforcement Learning II

Leif Döring
Benedikt Wille

Universität Mannheim

Inhaltsverzeichnis

0 Eine kleine Einführung in die Praxis für Mathematiker **2**
0.1 Wer oder was sind Git und GitHub? 2
0.1.1 Einrichten von Git und GitHub 2
0.1.2 Die wichtigsten Git Befehle 3
0.2 Wie funktionieren (Python-)Repositories? 7
0.2.1 Python Moduling 7
0.2.2 Spezielle Dateien in Repositories 9
0.3 Die Stable Baselines Repositories 11
0.3.1 Installation der Stable Baselines Repositories 11
0.3.2 Stable Baselines 3 Zoo 13
0.3.3 Stable Baselines 3 und SB3 Contrib 21
0.4 Rechnen auf den Clustern des Landes BaWü 21

1 TBD **22**

Einleitung

Willkommen zum Kurs „Reinforcement Learning II“. Diese Vorlesung wird etwas interaktiver und praktischer ablaufen, als die meisten anderen Mathe-Module. Wir werden so weit wie möglich in die Welt des modernen Reinforcement Learnings abtauchen. Oft kann man bei den sehr forgeschrittenen Algorithmen nicht mehr allzu viel schöne saubere Mathe machen, auch wenn viele der Ideen mathematisch fundiert sind. Das Ziel des Kurses ist es daher, euch wo auch immer es geht die mathematischen Grundlagen mitzugeben, aber euch auch mit robustem praktischen Wissen auszustatten. Während des Kurses wollen wir uns gemeinsam überlegen, ob uns mithilfe unserer mathemnatischen Intuition vielleicht sogar Verbesserungen für bestehende Algorithmen einfallen und diese auch direkt auf den „interessanten“ Beispielen ausprobieren. Am Ende wird bestimmt etwas dabei sein, was funktioniert!

Zu Beginn werden wir euch einen Crashkurs in der Anwendung geben, damit alle in etwa auf dem gleichen Stand sind. Dabei werden wir uns die Nutzung von Git und GitHub, sowie die Nutzung von Python-Repositories anschauen. Anschließend werden wir uns mit den Stable Baselines Repositories beschäftigen, die eine Grundimplementation für viele wichtige Reinforcement Learning Algorithmen auf den interessanten gymnasium und ALE (Arcade Learning Environment; Atari Spiele) Environments bieten. Im Verlauf des Kurses werden wir uns dann Themen wie DQN (Deep Q-Networks), PPO (Proximal Policy Optimization), und ähnliche Algorithmen, die in Stable-baslines verbaut sind, anschauen.

Kapitel 0

Eine kleine Einführung in die Praxis für Mathematiker

Die folgenden Abschnitte geben eine bewusst klein gehaltene Einführung in die absolut notwendigen Tools, die wir für den Kurs benötigen. Soweit möglich werden Fachbegriffe vermieden und alles so grundlegend wie möglich erklärt. In vielen Fällen ist das LLM eurer Wahl bereits ausreichend, um euch weiterzuhelfen, sobald ihr die Grundlagen kennt. Speziell für die Arbeit mit Shell-Script (der Programmiersprache, die die Terminals verstehen und mit der wir auf dem Cluster arbeiten) werden wir nur das absolut notwendige lernen. Wer dennoch etwas tiefer in die Materie einsteigen möchte und nicht so sehr von LLMs abhängig sein möchte, empfehle ich diesen Open Source Kurs vom MIT. Da wir mit Macs arbeiten, kann es sein, dass manche Befehle auf Windows Computern etwas anders heißen. In diesem Fall hilft eine schnelle Internetsuche weiter. Wo es große Unterschiede gibt, wird eine entsprechende Anmerkung zu finden sein.

0.1 Wer oder was sind Git und GitHub?

Git ist ein sogenanntes Versionskontrollsystem. Es hilft uns dabei, Änderungen an Dateien und Verzeichnissen (i.e. Ordnern) zu verfolgen und zu verwalten. Anders als euer reguläres System an Ordnern und Dateien auf dem Laptop, speichert Git nicht nur den aktuellen Stand aller Dateien, sondern den kompletten Verlauf von Änderungen, sodass wir jederzeit zu früheren Versionen zurückkehren können und Änderungen ersichtlich sind. Das ist vergleichbar mit z.B. Word, wo man auch eine Art Versionshistorie hat, nur macht das Git eben für alle Dateien in einem Ordner gleichzeitig. Wenn wir Git für einen Ordner aktiviert haben (dazu gleich mehr), nennen wir das ein Git-Repository. Git ist ein Open-Source-Tool und kann kostenlos genutzt werden.

Besonders nützlich (und auch am meisten in diesem Kontext verwendet) ist Git in Verbindung mit GitHub. GitHub ist eine webbasierte Plattform, die Git-Repositories hostet und zusätzliche Funktionen für die Zusammenarbeit bietet. Ein Repository (kurz Repo) ist einfach ein Ordner, der von Git verwaltet wird. Auf GitHub können wir unsere Repos online speichern, teilen, und gemeinsam mit anderen daran arbeiten. Wenn mehrere Personen an demselben Repository arbeiten, werden alle Änderungen zentral online gespeichert und können von allen Mitgliedern, sobald sie von irgendwem gespeichert werden mit einem Befehl abgerufen werden. Das vermeidet Konflikte bei gleichzeitigen Änderungen und ihr müsst euch so nicht ständig einzelne Dateien hin und herschicken oder manuell alle Zeilen, die andere Leute modifiziert haben, bei euch abändern.

0.1.1 Einrichten von Git und GitHub

Als erstes müssen wir Git lokal installieren und einen GitHub-Account erstellen.

GitHub Account: Geht auf `https://github.com/`die Homepage von GitHub und erstellt einen Account, falls ihr noch keinen habt. Merkt euch eueren Benutzernamen und euer Passwort, ihr werdet beides später brauchen, um Git mit eurem GitHub Account zu verbinden.

Git installieren: Auf neueren Macs ist Git in der Regel bereits vorinstalliert. Öffnet ein Terminal und gebt den Befehl

```
git --version
```

ein. Wenn eine Versionsnummer angezeigt wird, ist Git bereits installiert. Falls nicht, könnt ihr Git über `https://git-scm.com/downloads`diesen Link herunterladen und installieren. Folgt den Anweisungen auf der Webseite für die Installation.

Git konfigurieren: Nachdem Git installiert ist, müsst ihr es mit eurem GitHub-Account verbinden. Gebt im Terminal die folgenden Befehle ein und ersetzt `your_username` und `your_email@example.com` durch eure eigenen Daten (die Email-Adresse und den Benutzernamen, die ihr bei der Registrierung bei GitHub verwendet habt):

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

Es gibt nun mehrere Wege um sicherzugehen, dass ihr nicht jedes Mal, wenn ihr etwas auf GitHub speichern wollt, euren Benutzernamen und euer Passwort eingeben müsst. Der einfachste Weg, und den den wir euch empfehlen würden, ist es, euch in VSCode mit eurem GitHub Account anzumelden. Dazu öffnet ihr VScode, klickt auf das Symbol für Quellcodeverwaltung (das Symbol mit den drei verbundenen Punkten, werden wir später noch häufiger verwenden) und dann auf „Anmelden bei GitHub“. Folgt den Anweisungen, um euch anzumelden. Alternativ könnt ihr auch einen sogenannten „Personal Access Token“ (PAT) erstellen, der als Passwort fungiert oder einen sogenannten SSH-Schlüssel nutzen. Beide Methoden sind etwas komplizierter einzurichten, aber es gibt viele Anleitungen im Internet, die euch dabei helfen können und man kommt da im allgemeinen auch mit LLMs gut zurecht.

0.1.2 Die wichtigsten Git Befehle

Wir wollen jetzt der Reihe nach die wichtigsten Git Befehle durchgehen, damit ihr einen Überblick bekommt, was man alles mit Git anstellen kann. Viele der Befehle werden wir später gar nicht mehr nutzen, da VSCode viele der Funktionen bereits auf Knopfdruck integriert hat. Es ist aber trotzdem gut, die Befehle zu kennen, falls mal etwas nicht so funktioniert wie es soll.

Ein neues Repository erstellen: Git trackt nicht automatisch alle Dateien auf eurem Computer, sondern nur von euch ausgewählte Ordner, die dann zu Repos werden. Wir können entweder lokal einen komplett neuen Ordner zu einem Git-Repository machen oder aber ein bestehendes Repository von GitHub herunterladen.

```
git init
```

Der Befehl `git init` initialisiert ein neues Git-Repository in dem aktuellen Verzeichnis. Das bedeutet, dass Git ab jetzt alle Änderungen an den Dateien in diesem Ordner verfolgt. Ihr könnt dieses Repo später auch auf GitHub hochladen, damit ihr es online speichern und mit anderen teilen könnt. Wir werden uns aber nicht allzu viel mit dieser Option beschäftigen, da wir in der Regel bestehende Repos von GitHub klonen werden. Auch hier sei wieder an LLMs oder Internet-Guides verwiesen.

```
git clone <repository_url>
```

Mit dem Befehl `git clone <repository_url>` können wir ein bestehendes Repository von GitHub (oder einem anderen Git-Server) auf unseren lokalen Computer herunterladen (i.e. klonen). Ersetzt `<repository_url>` durch die URL des Repositories, das ihr klonen möchtet. Achtung: der geklonte Ordner wird in den Ordner, in dem sich euer Terminal aktuell befindet, heruntergeladen. Wechselt also vorher in den Ordner, in dem ihr das Repo speichern möchtet. Das könnt ihr über den Befehl `cd <pfad/zum/ordner_name>` machen. Wenn ihr euch nicht sicher seid, auf welche Ordner ihr gerade zugriff habt, könnt ihr den Befehl `ls` (list) eingeben, um alle Ordner und Dateien im aktuellen Verzeichnis anzuzeigen. Im Zweifel könnt ihr mit dem Befehl `cd ..` immer einen Ordner nach oben wechseln.

Als kleine Übung könnt ihr ein TestRepo auf eurem GitHub Account erstellen und es dann mit dem `git clone` Befehl auf euren Computer herunterladen. Geht dazu auf GitHub, klickt auf „New Repository“, gebt dem Repo einen Namen (z.B. `TestRepo`) und klickt auf „Create Repository“. Kopiert dann die URL des Repositories (die URL, die ihr in der Adresszeile eures Browsers seht bzw. die ihr über das Grüne „Code“ Feld findet) und verwendet sie im Terminal mit dem `git clone` Befehl. Führt das ganze am besten in VSCode aus, dann könnt ihr das heruntergeladene Repo direkt anschauen und bearbeiten.

Änderungen verfolgen und speichern: Nachdem wir ein Repository erstellt oder geklont haben, können wir Änderungen an den Dateien vornehmen. Git verfolgt diese Änderungen und VSCode zeigt euch die Änderungen auch an, aber sie sind noch nicht gespeichert (committed). Fügt als Übung zwei Dateien zu eurem TestRepo hinzu, mit denen wir die Grundlegenden Git Befehle ausprobieren können. Sobald ihr die Dateien erstellt habt (sofern ihr das Repo mit Git verbunden habt vorher), wird euch VSCode mit einem Grünen „U“ neben dem Dateinamen anzeigen, dass die Dateien noch nicht commited (gespeichert) sind.

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

Um Dateien abzuspeichern, müssen wir zuerst spezifizieren, welche Dateien wir überhaupt speichern wollen (das nennt man “zur Stage hinzufügen„). Das machen wir mit dem Befehl `git add <datei_name1> <datei_name2> ....` Ersetzt `<datei_name1>` und `<datei_name2>` durch die Namen der Dateien. Wenn sich Dateien in Unterordner befinden, müsst ihr den Pfad zum Dateinamen mit angeben (z.B. `ordner/datei.txt`). Alternativ könnt ihr auch `git add .` eingeben, um alle geänderten Dateien im aktuellen Verzeichnis und allen Unterverzeichnissen hinzuzufügen.

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

Um alle gestage-ten (zur Speicherung ausgewählten) Änderungen tatsächlich zu speichern, verwenden wir den Befehl `git commit -m "Beschreibung der Änderungen"`. Ersetzt `"Beschreibung der Änderungen"` durch eine kurze Beschreibung dessen, was ihr geändert habt. Diese Beschreibung hilft euch und anderen, später nachzuvollziehen, was in diesem Commit (dem Speichervorgang) geändert wurde.

```
git push
```

Nun habt ihr eure Änderungen lokal gespeichert, aber sie sind noch nicht auf GitHub hochgeladen. Dafür müssen wir sie auf GitHub hochladen (pushen). Das machen wir mit dem Befehl `git push`. Wenn ihr das erste Mal pusht, werdet ihr möglicherweise nach eurem GitHub-Benutzernamen und Passwort (oder dem Personal Access Token, falls ihr das eingerichtet habt) gefragt. Nach dem ersten Mal sollte Git eure Anmeldedaten speichern, sodass ihr sie nicht jedes Mal eingeben müsst.

Jetzt kennen wir den typischen Zyklus der Arbeit mit Git: Änderungen vornehmen, Dateien zur Stage hinzufügen, Änderungen committen und schließlich die Änderungen auf GitHub pushen.

```
git pull
git fetch
```

Wenn ihr nun mit anderen Personen zusammenarbeitet, kann es sein, dass andere Personen Änderungen am Repository vorgenommen und diese auf GitHub gepusht haben, nachdem ihr das erste Mal das Repo heruntergeladen habt. Um sicherzustellen, dass ihr die neuesten Änderungen habt, bevor ihr eure eigenen Änderungen pusht, verwendet ihr den Befehl `git pull` oder `git fetch`. Diese Befehle laden die neuesten Änderungen von GitHub herunter. `git pull` lädt die Änderungen herunter und integriert sie direkt in eure lokale Kopie des Repositories, während `git fetch` nur die Änderungen herunterlädt, ohne sie zu integrieren. In der Regel ist es einfacher, `git pull` zu verwenden, wenn ihr sicher seid, dass ihr keine Konflikte habt (i.e. jemand anderes hat dieselbe Code-Zeile modifiziert, aber auf eine andere Art und Weise als ihr das getan habt). So müsst ihr nicht anschließend die Änderungen noch einbauen lassen (auch dazu, siehe LLM/Online-Guides). Wie man potenzielle Konflikte löst und die unterschiedlichen commits (gespeicherten Versionen) zusammenführt werden wir gleich noch lernen.

Wenn wir nun unsere Änderungen gespeichert haben, könnt ihr ausprobieren, was passiert, wenn ihr eine der Dateien lokal löscht und in die andere etwas reinschreibt (sie modifiziert):

Wechselt in das Menü mit den drei verbundenen Punkten (die Git-Ansicht) in VSCode. Dort wird VSCode euch mit einem roten „D“ und einem hellgelben „M“ neben den Dateinamen anzeigen, dass eine Datei deleted (also gelöscht) ist und die andere modified (modifiziert) wurde. Generell findet ihr hier oben eine Übersicht über alle Änderungen, die ihr vorgenommen habt. Ihr habt hier direkt eine Vielzahl an Optionen, Git zu benutzen, ohne gleich die Kommandozeile eures Terminals zu verwenden.

Wenn ihr über dem Namen einer geänderten Datei hovert, von links nach recht: Datei öffnen (öffnet die Datei in zwei Ansichten nebeneinander, die eine zeigt die ursprüngliche Version, die andere die modifizierte), Änderungen verwerfen (also die Datei wieder in den ursprünglichen Zustand zurückversetzen), und Datei zur Stage hinzufügen (also für den Commit vormerken).

Außerdem habt ihr oben über den blauen Button mehrere Optionen, schritte direkt zu überspringen und keine Zeile Git Code ins Terminal zu schreiben. In jedem Fall könnt ihr eine Commit-Message ins Textfeld schreiben (wenn ihr das nicht tut müsst ihr im nächsten Schritt manuell in die Datei, die zum tracken der Änderungen genutzt wird die Message schreiben): Commit (also alle gestageten Änderungen oder wenn nichts gestaget ist alle Änderungen commiten), Commit and push (dasselbe, aber Änderungen auf GitHub hochladen), und Commit and Sync (dasselbe, aber vorher noch die neuesten Änderungen von GitHub herunterladen und integrieren). Falls ihr keine Änderungen haben solltet, wird euch, sollte es auf GitHub Änderungen geben, eine entsprechende Meldung angezeigt und ihr könnt git pull über den blauen Button laufen lassen.

Versucht mal, alle Änderungen die ihr vorgenommen habt zu committen (noch nicht auf GitHub pushen). Schaut euch nun im Menü unten die Historie an. Dort könnt ihr alle eure Commits und die Commits von anderen sehen inklusive wer sie durchgeführt hat, ebenso wie wo ihr (der Blaue marker) und die auf GitHub gespeicherte Version (der lila Marker) befindet. Ihr solltet sehen, dass ihr einen Commit vor der auf GitHub seid und wenn ihr eure Änderungen pusht, werdet ihr sehen, wie der lila Marker nach vorne wandert.

Ihr könnt prinzipiell auch Änderungen rückgängig machen, nachdem ihr oder andere sie commited haben. Wenn es soweit kommt, lasst euch am besten einfach vom LLM eurer Wahl helfen.

Branches und Zusammenführen von Änderungen: In Git können wir sogenannte Branches (Zweige) erstellen, um verschiedene Versionen unseres Codes zu entwickeln, ohne die Hauptversion (den „main“ oder „master“ Branch) zu beeinflussen. Das ist besonders nützlich, wenn mehrere Personen an demselben Projekt arbeiten oder wenn wir neue Funktionen ausprobieren möchten,

ohne den stabilen Code zu gefährden.

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

Mit dem Befehl `git branch <branch_name>` können wir einen neuen Branch erstellen (ohne einen branch namen zu setzen zeigt der Befehl einfach die Namen aller aktuellen Branches an). Ersetzt `<branch_name>` durch den Namen des neuen Branches. Mit dem Befehl `git checkout <branch_name>` wechseln wir (checken ihn aus) zu dem neu erstellten Branch, sodass alle Änderungen, die wir vornehmen, nur in diesem Branch gespeichert werden. Wenn ein neuer Branch erstellt wird, basiert er standardmäßig auf dem aktuellen Branch, in dem ihr euch befindet. Das bedeutet, dass der neue Branch eine Kopie des Codes und der Historie des aktuellen Branches zum Zeitpunkt der Erstellung enthält. Im Git-Menü von VSCode wird der neue Branch als Abzweigung vom Hauptbranch angezeigt (das stellt auch das Logo im Menü dar) und ihr bekommt angezeigt, wo ihr euch gerade befindet.

```
git merge <branch_name>
```

Wenn ihr mit den Änderungen in eurem Branch zufrieden seid und sie in den Hauptbranch integrieren möchtet, könnt ihr den Befehl `git merge <branch_name>` verwenden. Ersetzt `<branch_name>` durch den Namen des Branches, den ihr zusammenführen möchtet. Stellt sicher, dass ihr euch im Hauptbranch befindet, bevor ihr den Merge-Befehl ausführt, denn der Merge integriert die Änderungen des angegebenen Branches in den aktuellen Branch.

Wenn es Konflikte gibt (also wenn dieselbe Code-Zeile in beiden Branches unterschiedlich geändert wurde), wird Git euch darauf hinweisen und ihr müsst die Konflikte manuell lösen. VSCode bietet dafür eine benutzerfreundliche Oberfläche, um die Unterschiede zu sehen und zu entscheiden, welche Änderungen beibehalten werden sollen. Dazu könnt ihr über jeder Änderung anklicken, welche von beiden Versionen ihr benutzen wollt, um eine gemergte Version zu erstellen.

Dasselbe gilt im Übrigen auch, wenn ihr `git pull` verwendet und es Konflikte gibt, denn die Version auf GitHub wird dann mit eurem aktuellen Branch gemergt. Sollte es hier zu Konflikten kommen, könnt ihr diese ebenfalls manuell in VSCode lösen. Hier gibt es allerdings die Alternative, das automatisch durchführen zu lassen, was in der Regel auch gut funktioniert. Dafür gibt es drei konfigurationen, die ihr vorher (oder nach einem gefailten Pull vor dem nächsten Pull) in der Kommandozeile eingeben könnt. Das Terminal wird euch automatisch die möglichen gängigen Befehle anzeigen, wenn ihr `git pull` eingebt und es Konflikte gibt. Ansonsten könnt ihr auch das LLM eurer Wahl fragen.

Weitere nützliche Befehle und GitHub Funktionalitäten:
Auf GitHub gibt es noch viele weitere nützliche Funktionen, die die Zusammenarbeit erleichtern. Ich würde euch empfehlen, euch etwas Zeit zu nehmen und ein wenig durchzuklicken im Dialog mit einem LLM. Ihr könnt z.B. Issues (Probleme) erstellen, um Bugs oder Aufgaben zu verfolgen, Pull Requests (Anfragen zum Zusammenführen von Änderungen) erstellen, um Änderungen vorzuschlagen und zu diskutieren, und vieles mehr.

Eine wichtige Funktion ist, dass ihr fremde Repositories „forken“ könnt. Das bedeutet, dass ihr eine Kopie eines fremden Repositories in eurem eigenen GitHub Account erstellt. Ihr könnt dann Änderungen an dieser Kopie vornehmen, ohne das Original zu beeinflussen. Wenn ihr der Meinung seid, dass eure Änderungen nützlich sind, könnt ihr einen Pull Request erstellen, um die Besitzer des Original-Repositories zu bitten, eure Änderungen zu übernehmen. Außerdem könnt ihr Änderungen im originalen Repository übernehmen, indem ihr einen sogenannten „Upstream“ einrichtet und dann regelmäßig die neuesten Änderungen von dort in euer geforktes Repository zieht. Auch hier hilft euch das LLM eurer Wahl bestimmt weiter.

```
git rebase <branch_name>
```

Rebase ist eine weitere Möglichkeit, Änderungen von einem Branch in einen anderen zu integrieren. Im Gegensatz zum Merge, der die Historie beider Branches beibehält und einen neuen Commit erstellt, um die Änderungen zusammenzuführen, nimmt Rebase die Änderungen des angegebenen Branches und „spielt“ sie auf den aktuellen Branch ab. Das führt zu einer geraderen und saubereren Historie, da es so aussieht, als ob alle Änderungen nacheinander vorgenommen wurden. Allerdings kann Rebase die Historie verändern, was problematisch sein kann, wenn mehrere Personen an demselben Branch arbeiten. Daher wird Rebase oft für lokale Branches verwendet, die noch nicht mit anderen geteilt wurden.

Ansonsten bieten Git und GitHub noch viele weitere Funktionen, die je nach Bedarf genutzt werden können. Es lohnt sich, die Dokumentation zu lesen oder Tutorials anzuschauen, um ein tieferes Verständnis zu bekommen.

0.2 Wie funktionieren (Python-)Repositories?

Jetzt haben wir gelernt, wie ihr mit Git und GitHub umgehen könnt. Der nächste Schritt ist es, zu verstehen, wie (Python-)Repositories strukturiert sind und wie ihr sie effektiv nutzen könnt. Dazu schauen wir uns erst etwas genauer an, wie das Moduling in Python funktioniert und wie ihr es nutzen könnt, um in Repos zu arbeiten und anschließend, welche Arten von anderen Dateien sich in Repos typischerweise befinden werden, welchem Zweck sie dienen, und wie sie funktionieren.

0.2.1 Python Moduling

PYTHONPATH Damit Python weiß, wo es nach Modulen suchen soll, die ihr importieren möchtet, verwendet es eine Umgebungsvariable namens `PYTHONPATH`. Diese Variable enthält eine Liste von Verzeichnissen, in denen Python nach Modulen sucht. Wenn ihr ein Modul importiert, durchsucht Python diese Verzeichnisse in der Reihenfolge, in der sie in `PYTHONPATH` aufgeführt sind. Wenn ihr Python auf eurem Rechner zuerst installiert, dann ladet ihr im Prinzip einfach einen Ordner herunter, dessen Unterordner einmal die Python-Grundversion enthält und einen Ordner namens `Lib`, in dem alle Standardmodule und installierte Pakete als Ordner liegen. Zur Variable PYTHONPATH wird nun automatisch sowohl der Ordner, in dem die Python-Version liegt, als auch der Ordner, in dem sich die Module befinden, hinzugefügt. Wenn ihr nun in VSCode oder im Terminal einen Ordner öffnet und dort eine Python-Datei ausführt, wird automatisch der Ordner, in dem sich die Datei befindet, zu PYTHONPATH hinzugefügt. Um euch anzeigen zu lassen, welche Ordner aktuell in PYTHONPATH sind, könnt ihr im Terminal den Befehl `echo $PYTHONPATH` eingeben. Wenn ihr wissen wollt, wo eure aktuelle python-Installation liegt, könnt ihr den Befehl `which python` eingeben.

```
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 ...
```

Wenn ihr nun Module (also Ordner mit Python-Code) oder Funktionen aus Modulen (bzw. Dateien) in eurem Python-Code verwenden möchtet, könnt ihr sie mit dem `import` Befehl importieren. Ersetzt `<modul_name>` durch den Namen des Moduls, das ihr importieren möchtet. Wenn sich das Modul in einem Unterordner befindet, müsst ihr den Pfad zum Modul mit Punkten getrennt angeben (z.B. `from ordner.unterordner.modul import funktion`). Ihr könnt mit dem Befehl `from <modul_name> import <funktion_name> <classe_name> <object_name> ...` auch nur bestimmte Funktionen, Klassen oder Objekte aus einem Modul importieren, anstatt das gesamte Modul zu importieren. Das kann euren Code gegebenenfalls deutlich schneller machen.

Dabei wird Python in allen Ordnern suchen, die in PYTHONPATH definiert sind. Achtet also darauf, dass ihr immer relativ zum PYTHONPATH den Pfad zum Modul angebt, das ihr importieren möchtet. Wenn `<modul_name>` weder als Ordner noch als .py-Datei in einem dieser Ordner liegt, wird Python einen Fehler anzeigen, dass das Modul nicht gefunden werden konnte.

Besonders müsst ihr darauf achten, wenn ihr in einem Repository arbeitet, das aus mehreren Unterordnern besteht. Wenn ihr z.B. eine Datei in einem Unterordner ausführt und versucht, ein Modul aus einem anderen Unterordner zu importieren, wird das Modul im Allgemeinen nicht gefunden werden, da der aktuelle PYTHONPATH nur den Unterordner, aber nicht einen anderen Unterordner im selben Repository enthält. In solchen Fällen müsst ihr entweder den PYTHONPATH anpassen (indem ihr den Ordner, der das Modul enthält, hinzufügt) oder relative Importe verwenden (indem ihr den Pfad zum Modul relativ zum aktuellen Verzeichnis angebt). Dabei könnt ihr eine Ebene nach oben mit `..` vor dem Modulnamen bzw. Pfad wechseln.

Standardmäßig müsst ihr, wenn ihr keine fertigen Module, sondern eigenen Code in einem Repository nutzen möchtet, immer den Pfad zu einer .py-Datei angeben, aus der ihr importieren möchtet. Das könnt ihr umgehen, indem ihr in dem Ordner, in dem sich die .py-Datei befindet, eine Datei namens `__init__.py` erstellt. Diese Datei kann leer sein, aber ihre Existenz signalisiert Python, dass der Ordner als Modul behandelt werden soll. Danach könnt ihr den Ordnernamen als Modulnamen verwenden und Funktionen, Klassen oder Objekte daraus importieren. In dieser Datei könnt ihr auch globale Einstellungen für das Modul vornehmen, z.B. welche Funktionen oder Klassen standardmäßig importiert werden sollen, wenn das Modul importiert wird oder Initialisierungscode schreiben, der ausgeführt wird, wenn das Modul importiert wird oder globale Variablen definieren, die im gesamten Modul verfügbar sind.

Virtual Environments Wenn ihr mit Python arbeitet, ist es oft nützlich, sogenannte virtuelle Umgebungen (virtual environments) zu verwenden. Eine virtuelle Umgebung ist ein isolierter Bereich auf eurem Computer, in dem ihr eine eigene Python-Installation und eigene Pakete verwalten könnt, ohne dass sie mit der globalen Python-Installation auf eurem Computer in Konflikt geraten. Das ist besonders nützlich, wenn ihr an mehreren Projekten arbeitet, die unterschiedliche Versionen von Python oder unterschiedlichen Paketen benötigen. Dabei wird der PYTHONPATH automatisch so angepasst, dass nur die Pakete und Module in der virtuellen Umgebung verfügbar sind. Wenn ihr eine virtuelle Umgebung in VSCode erstellt, wird euch VSCode automatisch fragen, ob ihr die virtuelle Umgebung als Interpreter für das aktuelle Projekt verwenden möchtet. Bestätigt das, damit ihr sicherstellt, dass ihr die Pakete und Module in der virtuellen Umgebung verwendet. Ihr könnt nach dem aktivieren (dazu gleich mehr) mit dem Befehl `which python` überprüfen, ob ihr den richtigen Interpreter verwendet. Dabei sollte der Pfad, der euch angezeigt wird, zum virtual environment zeigen.

Bevor ihr eine virutelle Umgebung erstellt, solltet ihr pyenv installieren. pyenv ist ein Tool, mit dem ihr verschiedene Versionen von Python auf eurem Computer verwalten könnt. Für Mac könnt ihr das ganz einfach mit Homebrew machen, indem ihr den Befehl `brew install pyenv` im Terminal eingebt oder über die Guides auf dieser Seite. Für Windows gibt es keine “offizielle„ Version von pyenv,aber hier ist eine inoffizielle Version mit denselben features. Denkt aber immer daran, dass es nicht 100% dasselbe ist. Folgt dann den Anweisungen auf der pyenv GitHub Seite, um pyenv einzurichten.

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

Mit den Befehlen oben könnt ihr alle Python-Version, die ihr benötigt, installieren. Ersetzt `<python_version>` durch die gewünschte Version (z.B. `3.9.7`). Mit dem Befehl `pyenv global <python_version>` könnt ihr die globale Python-Version festlegen, die auf eurem Computer verwendet wird. Mit dem Befehl `pyenv local <python_version>` könnt ihr die Python-Version für

das aktuelle Verzeichnis, in dem sich euer Terminal befindet, auf dem ihr den Befehl durchführt, festlegen. Das ist besonders nützlich, wenn ihr an einem Projekt arbeitet, das eine bestimmte Python-Version benötigt, wie wir es später haben werden.

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

Um nun eine virtuelle Umgebung zu erstellen, verwendet ihr den Befehl `python -m venv <env_name>`. Ersetzt `<env_name>` durch den Namen, den ihr der virtuellen Umgebung geben möchtet (z.B. `venv`). Das erstellt einen neuen Ordner mit dem Namen der virtuellen Umgebung im aktuellen Verzeichnis, der die isolierte Python-Installation und die Pakete enthält. VSCode wird euch automatisch fragen, ob ihr die virtuelle Umgebung als Interpreter für das aktuelle Projekt verwenden möchtet. Bestätigt das, damit ihr sicherstellt, dass ihr die Pakete und Module in der virtuellen Umgebung verwendet. Normalerweise aktiviert VSCode die virtuelle Umgebung auch automatisch in eurem Terminal, dann seht ihr vor der Eingabezeile den Namen der virtuellen Umgebung in Klammern. Falls das nicht der Fall ist, könnt ihr die virtuelle Umgebung manuell aktivieren, indem ihr den Befehl `source <env_name>/bin/activate` eingebt. Wenn die virtuelle Umgebung aktiviert ist, wird der Name der Umgebung in Klammern vor der Eingabezeile im Terminal angezeigt. Um die virtuelle Umgebung zu deaktivieren und zur globalen Python-Installation zurückzukehren, könnt ihr den Befehl `deactivate` eingeben.

Pakete installieren In Python werden zusätzliche Funktionen und Werkzeuge oft in sogenannten Paketen bereitgestellt. Pakete sind Sammlungen von Modulen, die bestimmte Funktionalitäten bieten. Um Pakete zu installieren, verwenden wir in der Regel den Paketmanager `pip`, der standardmäßig mit Python installiert wird.

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

Ihr könnt entweder manuell mittels `pip install <package_name> <version_specifier>` Pakete mit einer bestimmten Version (ihr könnt ==, <, >, <=, >= verwenden) installieren. Ersetzt `<package_name>` durch den Namen des Pakets, das ihr installieren möchtet, und `<version_specifier>` durch eine optionale Versionsangabe (z.B. `==1.2.3` für eine bestimmte Version oder `>=1.2.3` für eine Mindestversion). Wenn ihr die Version weglasst, wird automatisch die neueste Version des Pakets installiert. Alternativ, falls euer Projekt eine Datei namens `requirements.txt` enthält (was in den meisten Repositories der Fall sein wird), könnt ihr alle Pakete, die in dieser Datei aufgelistet sind, mit dem Befehl `pip install -r requirements.txt` installieren. Diese Datei enthält eine Liste von Paketen und deren Versionen, die für das Projekt benötigt werden.

0.2.2 Spezielle Dateien in Repositories

Wir werden hier vor allem auf Dateien eingehen, die ihr in den Stable Baselines Repositories finden werdet. Nicht alle werdet ihr unbedingt brauchen, aber es hilft, sie zu kennen und zu wissen wofür sie da sind. Es gibt natürlich noch viele weitere Dateien, die in Repositories verwendet werden können, aber hier enthalten sind auch die absoluten Standard-Dateien, die es so gibt.

.gitkeep Wenn ihr in einem Repo einen leeren Ordner erstellt, werdet ihr recht schnell merken, dass Git diesen Ordner nicht trackt (also nicht speichert). Das liegt daran, dass Git standardmäßig keine leeren Ordner verfolgt. Wenn ihr jedoch einen leeren Ordner in eurem Repository behalten möchtet (z.B. weil ihr plant, später Dateien darin zu speichern), könnt ihr eine Datei namens `.gitkeep` in diesem Ordner erstellen. Diese Datei kann leer sein, aber ihre Existenz signalisiert

Git, dass der Ordner nicht leer ist und daher verfolgt werden soll.

.gitignore In vielen Repositories werdet ihr eine Datei namens `.gitignore` finden. Diese Datei enthält eine Liste von Dateien und Ordnern, die Git ignorieren soll. Das ist nützlich, um temporäre Dateien, Log-Dateien, Ergebnisdateien, oder andere Dateien, die nicht versioniert werden sollen, aus dem Repository auszuschließen. So kann das Repository sauber und übersichtlich bleiben und ihr verschwendet keinen Speicherplatzauf GitHub und auf den Computern anderer Leute, die mit dem Repository arbeiten (außerdem ist es natürlich auch schneller, wenn Git nicht unnötig viele Dateien tracken muss). Die Syntax in der .gitignore Datei ist relativ einfach: Jede Zeile enthält ein Muster, das auf Dateien oder Ordner angewendet wird. Zum Beispiel ignoriert der Eintrag `*.log` alle Dateien mit der Endung .log, und der Eintrag `temp/` ignoriert den Ordner temp und alle seine Inhalte. Auch hier werden euch LLMs weiterhelfen können.

.python-version Wenn ihr ein Virtual Environment mit pyenv erstellt habt, wird automatisch die Datei `.python-version` erstellt. Diese Datei enthält die Python-Version, die für das Projekt verwendet werden soll. Wenn ihr in einem Verzeichnis arbeitet, in dem sich diese Datei befindet, solltet ihr auch mit der angegebenen Python-Version arbeiten. pyenv aktiviert automatisch die angegebene Python-Version, wenn ihr den Befehl `pyenv local` verwendet. Das stellt sicher, dass ihr immer die richtige Python-Version für das Projekt verwendet, ohne dass ihr sie manuell wechseln müsst.

.readthedocs.yaml Diese Datei wird verwendet, um die Dokumentation für das Projekt zu konfigurieren, wenn ihr Read the Docs (eine Website für Guides zu Repositories) verwendet, um die Dokumentation zu hosten. In dieser Datei könnt ihr verschiedene Einstellungen vornehmen, z.B. welche Version von Python verwendet werden soll, welche Abhängigkeiten installiert werden müssen, und wie die Dokumentation gebaut werden soll.

LICENSE In dieser Datei wird die Lizenz für das Projekt angegeben. Die Lizenz legt fest, wie der Code verwendet, modifiziert, und verteilt werden darf. Es ist wichtig, die Lizenz zu verstehen, bevor ihr den Code in eurem eigenen Projekt verwendet oder ändert. GitHub stellt verschiedenen Standard-Lizenzen zur Verfügung, die für Open-Source Arbeit gedacht sind und zeigt bei Projekten auf der Hauptseite auch unter anderem die Lizenz an. In den Stable Baselines Repositories wird die MIT-Lizenz verwendet, die sehr permissiv ist und es euch erlaubt, den Code fast ohne Einschränkungen, aber auch ohne Garantien zu verwenden.

Makefile Ein Makefile ist eine Datei, die Anweisungen für das `make` Tool enthält. Make ist ein Build-Automatisierungstool, das ursprünglich für die Kompilierung von Programmen in C und C++ entwickelt wurde, aber auch für andere Aufgaben verwendet werden kann. In einem Makefile könnt ihr verschiedene „Targets“ definieren, die bestimmte Aufgaben ausführen, wenn ihr den Befehl `make <target>` im Terminal eingebt. Zum Beispiel könnte ein Target namens `install` Anweisungen enthalten, um alle Abhängigkeiten zu installieren, und ein Target namens `test` könnte Anweisungen enthalten, um alle Tests auszuführen. Makefiles sind besonders nützlich, um wiederkehrende Aufgaben zu automatisieren und sicherzustellen, dass sie immer auf die gleiche Weise ausgeführt werden. Stable Baselines hat ein Makefile mit einigen Befehlen, aber ihr werdet das voraussichtlich nicht wirklich brauchen.

pyproject.toml Diese Datei ist eine relativ neue Möglichkeit, Metadaten und Abhängigkeiten für Python-Projekte zu definieren. Sie wird von verschiedenen Tools verwendet, um Informationen über das Projekt zu erhalten, z.B. welche Version von Python verwendet werden soll, welche Abhängigkeiten installiert werden müssen, und wie das Projekt gebaut werden soll. In den Stable Baselines Repositories wird die Datei verwendet, um die Abhängigkeiten des Projekts zu definieren. Ein Tool, mit dem ihr diese Datei nutzen könnt, ist `poetry`. Wir werden aber später die dependecies anders installieren.

README.md Die Datei `README.md` ist eine sehr wichtige Datei in jedem Repository. Sie soll

eine Beschreibung des Projekts, Anweisungen zur Installation und Verwendung, und andere wichtige Informationen enthalten. Die Datei ist im sogenannten Markdown-Format geschrieben, das eine einfache Möglichkeit bietet, Text zu formatieren und Links, Bilder, und andere Elemente hinzuzufügen (siehe LLM für genaue Funktionsweise). Wenn ihr ein Repository auf GitHub öffnet, wird der Inhalt der README-Datei automatisch auf der Hauptseite des Repositories angezeigt. Es ist eine gute Praxis, eine ausführliche README-Datei zu erstellen, damit andere Personen (und ihr selbst in der Zukunft) verstehen können, worum es in dem Projekt geht und wie es verwendet werden kann.

requirements.txt Diese Datei enthält eine Liste von Paketen und deren Versionen, die für das Projekt benötigt werden. Ihr könnt alle Pakete, die in dieser Datei aufgelistet sind, mit dem Befehl `pip install -r requirements.txt` installieren. Das ist besonders nützlich, wenn ihr an einem Projekt arbeitet, das viele Abhängigkeiten hat, da ihr so alle benötigten Pakete mit einem einzigen Befehl installieren könnt. Wenn ihr bereits ein Repo erstellt habt, das noch keine requirements.txt Datei hat, könnt ihr eine solche Datei auch selbst erstellen, indem ihr den Befehl `pip freeze > requirements.txt` eingebt. Das erstellt eine Datei, die alle aktuell in eurer (virtuellen) Umgebung installierten Pakete und deren Versionen auflistet.

.sh-Dateien Dateien mit der Endung `.sh` sind Shell-Skripte, die Anweisungen enthalten, die in einer Unix-Shell (wie Bash; im Prinzip wie euer Terminal am Computer) ausgeführt werden können. Diese Skripte können verwendet werden, um wiederkehrende Aufgaben zu automatisieren, z.B. das Einrichten einer Umgebung, das Ausführen von Tests, oder das Starten eines Programms. Wenn wir später auf dem Cluster rechnen, müssen wir Shell-Skripte verwenden, um sie in die Warteschlange zu schicken und können nicht mehr wie gewohnt unseren Code einfach mit dem grünen Dreieck rechts oben in VSCode laufen lassen. Die Syntax ist mehr oder weniger wie in der Kommandozeile, aber es gibt einige Unterschiede und zusätzliche Funktionen, die in Shell-Skripten verwendet werden können. Wenn wir uns mit dem Rechnen auf den Clustern beschäftigen, werden wir nur die absoluten Basics, die wir brauchen, lernen. Auch hier wird auf LLMs verwiesen, aber auch nochmal auf den MIT Open-Source Guide vom Anfang des Kapitels.

0.3 Die Stable Baselines Repositories

Die Grundidee der stable baselines Repos ist es, einheitliche Implementierungen von Algorithmen sowie eine organisierte Art und Weise, diese zu testen und zu vergleichen, zu bieten. Gerade im Deep RL (RL mit Neuronalen Netzen statt Tabular) gibt es eine große Anzahl an Hyperparametern und verschiedene Arten der Implementierung machen einen echten Unterschied. Es gibt drei zu stable baselines gehörige Repos. Die ersten beiden sind python packages und enthalten Implementierungen von einigen Basisalgorithmen (Stable baselines 3), sowie später von anderen Autoren beigesteuerte Algorithmen (sb3 contrib), die von den Stable Baselines Autoren geprüft wurden. Das dritte Repo, Stable Baselines 3 Zoo enthält Funktionen, um die Algorithmen mit vorgemerkten Hyperparametern laufen zu lassen, plots über die Performance zu erhalten, und vieles mehr. Da wir für das Erklären der Algorithmen diese erstmal kennen müssen, und auch ein wenig mehr über spezielle Tricks im Deep Setting wissen müssen, beschäftigen wir uns erst einmal mit dem Zoo. Im Skript wird die Erklärung der Algorithmen-Implementierung direkt im Anschluss kommen, in der Vorlesung allerdings nicht. Wir werden auch auf einer abgespeckten fork des Zoos arbeiten, daher werden wir nicht alle original-features besprechen, aber diese können auf dem originalen GitHub Repo im README und/oder der Dokumentation nachgelesen werden.

0.3.1 Installation der Stable Baselines Repositories

Wir arbeiten auf einer eigenen fork des Zoos, die eine etwas abgespeckte Version enthält. Wenn ihr Zugang dazu haben wollt, schreibt uns einfach eine Mail. Sobald wir euch ins Team eingeladen haben, könnt ihr das Repo klonen und darauf arbeiten. Erstellt bitte als erstes einen eigenen

branch, auf dem ihr arbeiten werdet, damit ihr nicht aus Versehen den Hauptbranch oder die Arbeit anderer Leute verändert. Das könnt ihr entweder in VSCode machen, oder mit dem Befehl `git checkout -b <branch_name>`. Ersetzt `<branch_name>` durch den Namen, den ihr dem Branch geben möchtet und benutzt den Befehl `git checkout <branch_name>`, um zu dem Branch zu wechseln. VSCode wir euch auch anzeigen, in welchem Branch ihr euch gerade befindet.

pyenv Bevor wir die Repositories klonen, wollen wir pyenv als Versionskontrollsystem herunterladen. Für MacOS könnt ihr das ganz einfach mit Homebrew machen, indem ihr den Befehl `brew install pyenv` im Terminal eingebt oder über die Guides auf der pyenv GitHub Seite. Nach der Installation müsst ihr einige Schritte ausführen, um sicherzugehen, dass pyenv korrekt eingerichtet ist. Folgt den Anweisungen auf der pyenv GitHub Seite, um pyenv zu initialisieren und sicherzustellen, dass es in eurem Terminal funktioniert. Mit dem Befehl `pyenv which` könnt ihr überprüfen, ob pyenv korrekt installiert ist. Für Windows-User gibt es eine inoffizielle Version von pyenv namens pyenv-win. Denkt aber immer daran, dass es nicht 100% dasselbe ist. Folgt dann den Anweisungen auf der pyenv GitHub Seite, um pyenv bei euch einzurichten.

Installiert nun die Python Version 3.9.19 oder die am besten dazu passende Version (pyenv-win scheint diese Version z.B. nicht zu haben). Das könnt ihr mit dem Befehl `pyenv install 3.9.19` machen. Danach könnt ihr mit dem Befehl `pyenv global 3.9.19` die globale Python Version auf 3.9.19 setzen, sodass ihr immer diese Version verwendet, wenn ihr Python benutzt. Alternativ könnt ihr auch mit dem Befehl `pyenv local 3.9.19` die Python Version nur für das aktuelle Verzeichnis setzen, in dem sich euer Terminal befindet, wenn ihr den Befehl eingebt. Dabei müsst ihr darauf achten, dass ihr euch in dem Verzeichnis (Ordner) befindet, in dem ihr das Repository geklont habt. Wenn alles geklappt hat, könnt ihr mit dem Befehl `which python` überprüfen, ob ihr in eurem aktuellen Verzeichnis die richtige Python Version verwendet. Der Pfad, der euch angezeigt wird, sollte zu der Python Version 3.9.19 zeigen.

venv erstellen Nachdem ihr pyenv installiert und die richtige Python Version ausgewählt habt, könnt ihr eine virtuelle Umgebung für das Projekt erstellen. Achtet erneut darauf, dass sich euer Terminal in dem Verzeichnis befindet, in dem ihr bereits das Repo geklont habt. Gebt den Befehl `python -m venv venv` ein, um eine virtuelle Umgebung mit dem Namen `venv` zu erstellen. Das erstellt einen neuen Ordner namens `venv` in eurem aktuellen Verzeichnis, der die isolierte Python-Installation und die Pakete enthält. VSCode wird euch automatisch fragen, ob ihr die virtuelle Umgebung als Interpreter für das aktuelle Projekt verwenden möchtet. Bestätigt das, damit ihr sicherstellt, dass ihr die Pakete und Module in der virtuellen Umgebung verwendet. Normalerweise aktiviert VSCode die virtuelle Umgebung auch automatisch in eurem Terminal, dann seht ihr vor der Eingabezeile den Namen der virtuellen Umgebung in Klammern. Falls das nicht der Fall ist, könnt ihr die virtuelle Umgebung manuell aktivieren, indem ihr den Befehl `source venv/bin/activate` eingebt. Wenn die virtuelle Umgebung aktiviert ist, wird der Name der Umgebung in Klammern vor der Eingabezeile im Terminal angezeigt. Falls VSCode euch nicht dazu auffordert, die virtuelle Umgebung zu verwenden oder ihr das Fenster aus Versehen weggeklickt habt, ihr aber nicht immer händisch das venv aktivieren wollt, könnt ihr auch in der unteren linken Ecke von VSCode auf die Einstellungen klicken (das Zahnrad) und dann auf „Command Palette...“ gehen (Shortcut MacOS: ⇧+⌘+P). Gebt dort „Python: Select Interpreter“ ein und wählt den Interpreter aus dem venv Ordner aus. Ihr könnt nun mit dem Befehl `which python` erneut überprüfen, ob ihr den richtigen Interpreter verwendet und mit `python -version`, ob ihr die richtige Python Version verwendet.

Dependencies installieren Nun könnt ihr alle benötigten Pakete und deren Versionen, die in der Datei `requirements.txt` aufgelistet sind, installieren. Bevor ihr direkt die Ansammlung installieren könnt, müsst ihr allerdings noch ein Paket namens `swig` installieren. Das macht ihr mit dem Befehl `pip install swig`. Danach könnt ihr mit dem Befehl `pip install -r requirements.txt` alle Pakete, die in der Datei `requirements.txt` aufgelistet sind, installieren. Das installiert direkt alle benötigten Pakete und deren Versionen in eurer virtuellen Umgebung. Falls ihr mit poetry arbeiten wollt, könnt ihr alternativ auch den Befehl `poetry install` verwen-

den, um die Abhängigkeiten zu installieren. Solltet ihr bei Paketen Fehler bekommen, geht in die Datei `requirements.txt` und klammert die problematischen Pakete aus, indem ihr eine Raute (#) vor die Zeile setzt. Danach könnt ihr den Befehl erneut ausführen. Fragt am besten ein LLM eurer Wahl, wenn ihr nicht wisst, wie ihr die Probleme mit einzelnen Paketen lösen könnt, wir würden auch nichts anderes machen! Tatsächlich haben wir das Paket `pybullet_envs_gymnasium` auch nicht installiert bekommen (scheint ein Fehler mit Kompatibilität zu MacOS zu sein), aber das brauchen wir nicht unbedingt, da es nur zusätzliche Umgebungen liefert. Wenn ihr Umgebungen mit `bullet` im Namen verwenden wollt, müsst ihr das Paket allerdings installieren.

0.3.2 Stable Baselines 3 Zoo

Jetzt, wo wir hoffentlich alle das Stable Baselines 3 Zoo Repo geklont und eingerichtet haben, können wir uns dessen Funktionen anschauen.

Grundlegende Funktionen zum Trainieren, Testen, und Plotten

train.py Im Ordner rl_zoo3 befindet sich eine Datei namens `train.py`. Diese Datei enthält die Funktion train, um Algorithmen zu trainieren. Da eine Initialisierungsdatei namens `train.py` direkt im Repo vorhanden ist, die diese Funktion aufruft, könnt ihr die Funktion ausführen, indem ihr im Terminal den Befehl `python train.py` eingebt.

Die Funktion akzeptiert verschiedene Argumennte, die ihr im Terminal über sogenannte „flags“ angeben könnt. Flags sind im Prinzip Namen für Parameter, die ihr übergeben könnt. Es gibt short flags (mit einem Bindestrich, z.B. `-f`) und long flags (mit zwei Bindestrichen, z.B. `-flag`). Wenn ihr ein Flag eingebt, wird der Wert, der dem Flag folgt, als Wert für diesen Parameter verwendet. Wenn ihr z.B. den Befehl `python train.py -algo ppo` eingebt, wird der Wert `ppo` als Wert für das Flag `-algo` verwendet.

In der Datei `train.py` im Ordner rl_zoo3 könnt ihr nachschauen, welche Flags ihr verwenden könnt und was sie bedeuten. In der Datei `run_example.sh` findet ihr ein Beispiel dafür, wie ihr die Funktion verwenden könnt (kopiert Zeile 5 in euer Terminal). Alternativ könnt ihr mit dem Befehl `bash run_example.sh` das Skript ausführen, das die Funktion mit den angegebenen Flags aufruft. Probiert das am besten einmal aus. Ihr bekommt am Anfang der Ausgabe eine Übersicht über die verwendeten Parameter und sobald das Training losgeht Zwischenstände wie beobachteter score angezeigt. Die wichtigsten Flags der train-funktion sind:

- `-algo`: Der Algorithmus, der verwendet werden soll (z.B. `ppo`, `a2c`, `dqn`, etc.).
- `-env`: Die Umgebung, in der der Algorithmus trainiert werden soll (z.B. `CartPole-v1`, `MountainCar-v0`, etc.).
- `-n-timesteps/-n`: Die Anzahl der Zeitschritte, die der Algorithmus trainiert werden soll.
- `-log-interval`: Das Intervall (in Episoden), in dem die Trainingsfortschritte geloggt werden sollen. Das sind die Informationen, die ihr im Terminal seht, wenn ihr den Code laufen lasst.
- `-eval-freq`: Das Intervall (in Zeitschritten), nach dem der Algorithmus evaluiert werden soll.
- `-eval-episodes`: Die Anzahl der Episoden, die für die Evaluation verwendet werden sollen.
- `-n-eval-envs`: Die Anzahl der Umgebungen, die für die Evaluation verwendet werden sollen.

- `-f`: Der Pfad zum Verzeichnis, in dem die Trainings- und Evaluationsmodelle gespeichert werden sollen. Standardmäßig ist `./logs` eingestellt, aber wir wollen typischerweise nach Projekten sortierte Unterordner benutzen.
- `-seed`: Der Zufallsseed, der verwendet werden soll, um die Ergebnisse reproduzierbar zu machen.
- `-optimize`: Wenn dieses Flag gesetzt ist, werden die Hyperparameter des Algorithmus optimiert, bevor das Training beginnt. Mehr dazu später.
- `-params/-hyperparams`: Die Hyperparameter, die für das Training verwendet werden sollen. Ihr könnt entweder den Namen einer JSON-Datei angeben, die die Hyperparameter enthält (werden wir wahrscheinlich nicht brauchen), oder ihr könnt einzelne Hyperparameter direkt im Terminal angeben (z.B. `-hyperparams learning_rate=0.001 batch_size=64`).

Die train-Funktion speichert eure Daten in folgender Ordnerstruktur ab:

```
Speicherpfad/
└── Algo_Name/
    └── Env_Name_Nummer/
        ├── Env_Name/
        │   ├── args.yml
        │   ├── command.txt
        │   └── config.yml
        ├── Nummer.monitor.csv
        ├── Env_Name.zip
        ├── best_model.zip
        └── evaluations.npz
```

Die Dateien args und configs enthalten die verwendeten Parameter im sogenannten YAML-Dateiformat (pythoneigen, einträge mit richtiger Syntax können einfacher in Python-Objekte zurückgewandelt werden) und command.txt den Befehl, den ihr ins Terminal eingegeben habt. Die einzelnen Monitor-Dateien enthalten Werte, die in der Evaluation beobachtet wurden. Atari-Environments werden standardmäßig z.B. auf 8 parallelen Environments evaluiert, daher werden 8 monitor-dateien im csv-Dateiformat ausgegeben. Jede dieser Dateien enthält im Header den Zeitpunkt, zu dem das Monitoring angefangen hat und die ID des Environments, sowie tripel aus rewards (r), Episodenlängen (l), und Zeit (t), die beim Monitoring gebraucht wurde. Die zip-Dateien enthalten das Model zum Endzeitpunkt und das beste im Training gesehene Modell. Diese Dateien sind sehr groß, da wir mit Neuronalen Netzen arbeiten. Daher solltet ihr diese, sofern ihr sie nicht für die enjoy-Funktion (kommt gleich) braucht, am besten löschen oder später im Cluster nur auf sogenannten `workspaces` speichern, nicht aber auf eurem Laptop. Die npz-Datei enthält die aggregierten Ergebnisse von den Metriken, die ihr geloggt habt. Mehr dazu, wie man weitere Metriken außer standardmäßig den scores, die ihr zum evaluationszeitpunkt erhalten habt, loggt, kommt später.

enjoy.py Auch diese Funktion befindet sich im Ordner rl_zoo3 und hat eine Initialisierungsdatei namens `enjoy.py`, die die Funktion direkt aufruft. Mit dieser Funktion könnt ihr ein bereits trainiertes Modell laden und es in der angegebenen Umgebung ausführen lassen und beobachten, wie euer Agent sich verhält. Das ist nützlich, um zu sehen, wie gut das Modell nach dem Training performt. Die Funktion akzeptiert ebenfalls verschiedene Flags, die ihr im Terminal angeben könnt. Die wichtigsten Flags der enjoy-Funktion sind:

- `-env`: Die Umgebung, in der das Modell ausgeführt werden soll (z.B. `CartPole-v1`, `MountainCar-v0`, etc.).
- `-f`: Der Pfad zum Verzeichnis, in dem das trainierte Modell gespeichert ist. Befinden sich hier mehrere Modelle, wird automatisch das neuste verwendet. Ihr könnt aber auch eine spezielle id angeben, die das Modell eindeutig identifiziert.

- `-algo`: Der Algorithmus, der verwendet wurde, um das Modell zu trainieren (z.B. `ppo`, `a2c`, `dqn`, etc.).
- `-n-timesteps`: Die Anzahl der Episoden, die das Modell ausführen soll.
- `-seed`: Der Zufallsseed, der verwendet werden soll, um die Ergebnisse reproduzierbar zu machen.

Die Plotfunktionen befinden sich im Ordner .rl_zoo3/plots und haben keine Initialisierungsdateien. Daher müsst ihr immer den vollen Pfad angeben, wenn ihr `python rl_zoo3/plots/<name.py>` nutzen wollt im Terminal.

all_plots Diese Plotfunktion aggregiert die Daten von allen Runs in einem Bestimmten Directory, sortiert sie nach Algorithmen, Environments, und Metriken, und speichert diese gesammelt ab. Dabei ist es wichtig, dass der Ordner, den ihr als Input übergebt, die richtige Struktur hat. Die Funktion train.py speichert bereits die richtige Struktur ab, sodass ihr für ein Projekt am besten gesammelt den Pfad angebt, den ihr auch der train-Funktion übergeben habt. Zu beachten ist, dass immer alle Runs, die sich auf dem übergebenen Pfad befinden genommen werden, ihr also darauf achten müsst, dass keine ungewollten runs darin enthalten sind. Die Ergebnisse werden dabei aus der Datei `evaluations.npz` in den jeweiligen Ordnern der Runs geholt. Die wichtigsten Flags sind:

- `-algos/-a`: Die Algorithmen, deren Daten aggregiert werden sollen.
- `-env/-e`: Die Environments, deren Daten aggregiert werden sollen.
- `-exp-folders/-f`: Die Ordner, in denen nach den zu aggregierenden Daten gesucht werden soll.
- `-max-timesteps/-max und -min-timesteps/-min`: Die maximalen und minimalen Zeitschritte, die ein Experiment gelaufen sein muss, um inkludiert zu werden. Das ist besonders nützlich, um abgebrochene Experimente aus den Daten herauszufiltern.
- `-output/-o`: Der Pfad (inklusive Namen der Datei, die erstellt werden soll), in den die Output-Datei gespeichert werden sollen. Dabei ist zu beachten, dass der Pfad bereits existieren muss, anders als der Pfad in der train-Funktion.

Die all_plots-Funktion gibt neben der .pkl-Datei, die abgespeichert wird und die aggregierten Daten enthält ebenfalls je Environment einen Plot mit den Mittelwerten und Konfidenzintervallen allenr Algorithmen aus, die gefunden wurden. Dieser muss aber separat abgespeichert werden.

plot_from_file Diese Plotfunktion benutzt die Daten, die durch die Funktion all_plots aggregiert wurden, um weitere Plots daraus zu machen. Wichtig dabei ist, dass standardmäßig immer alle Daten verwendet werden, ihr also bestimmte algorithmen, environments, oder geloggte Metriken, die ihr nicht plotten wollt, per flags ausschließen müsst. Es gibt auch die Option, verschiedene Plots aus dem Paket RLiable erstellen zu lassen. Die Funktionen des Pakets werden in der Übernächsten Sektion nochmal erklärt. Die wichtigsten Flags sind:

- `-input/-i`: Der Pfad zu der .pkl-Datei, die die aggregierten Daten enthält, aus denen geplottet werden soll.
- `-output/-o`: Der Pfad (inklusive Namen der Datei, die erstellt werden soll), in den der Output-plot gespeichert werden soll. Dabei ist zu beachten, dass der Pfad bereits existieren muss, anders als der Pfad in der train-Funktion.
- `-labels/-l`: Die Labels, die die Kurven haben sollen. Dabei ist zu beachten, dass die Reihenfolge der geplotteten Kurven der Reihenfolge entspricht, die durch die Flag `-algos` der all_plots Funktion festgelegt wird.

- `-boxplot/-b`: Gibt zusätzlich einen Boxplot über die Sensitivität der Daten aus (siehe Erklärung zu RLiable).
- `-rliable/-r`: Gibt zusätzlich einen Plot über die RLiable-Metriken und die Performance Profiles aus (siehe Erklärung zu RLiable).
- `-versus/-vs`: Gibt zusätzlich den Probability of Improvement Plot aus RLiable aus (siehe Erklärung zu RLiable).
- `-iqm/-iqm`: Gibt zusätzlich den sample efficiency Plot aus RLiable aus (siehe Erklärung zu RLiable).

Die plot_from_file Funktion gibt standardmäßig ein Balkendiagramm mit Konfidenzintervallen über die Scores der Algorithmen auf den Environments aus, welches auch abgespeichert wird, sowie dieselben Plots, die auch all_plots ausgibt (ohne sie abzuspeichern). Zusätzlich können noch Plots aus dem RLiable Paket erstellt werden (welche auch nicht abgespeichert werden).

plot_train Diese Plotfunktion benutzt die Daten aus den Monitor-Dateien, um die Performance des Agenten während des Trainings zu zeigen. Die wichtigsten Flags sind:

- `-algo/-a`: Der Algorithmus, dessen Trainings-Performance geplottet werden soll.
- `-env/-e`: Die Environments, die inkludiert werden sollen.
- `-exp-folder/-f`: Die Ordner, aus denen Dateien inkludiert werden sollen.
- `-x-axis/-x`: Die Wahl, ob man nach Schritten, Episoden, oder Zeit plotten möchte.
- `--y-axis/-y` Die Wahl, ob man Success, Rewards, oder Episodenlänge plotten möchte.

Die plot_train Funktion gibt den spezifizierten Plot für alle gewählten Environments aus, speichert aber nichts ab.

Es lohnt sich bei allen diesen Funktionen, einmal durchzugehen und sich anzuschauen, welche Optionen ihr alles angeben könnt. Die meisten Flags beinhalten eine Erklärung, aus der eigentlich sofort hervorgeht, was sie machen. Die Flags sind am Anfang der Funktion definiert, wo ihr sonst deren Parameter finden würdet.

Python Debugging nutzen, damit wir nicht alles ins Terminal eingeben müssen

Es ist ziemlich lästig, wenn wir jedes Mal den gesamten Befehl inklusive aller Flags im Terminal angeben müssen, obwohl sich in der Regel in euren Workflows nur wenige Flags häufig ändern werden. Dafür gibt es zwei Lösungen.

.sh-Dateien Ein Umweg ist, Shell-Script-Dateien zu verwenden. Ein Beispiel ist die Datei `run_example.sh`, die ihr im Repo findet. Ihr könnt euch eigene .sh-Dateien erstellen und diese über den Befehl `bash <Datei_Name.sh>` ausführen. Gerade wenn ihr viele Funktionen hintereinander ausführen wollt, ist das sehr nützlich (ihr könnt z.B. loops über algorithmen oder seeds einbauen und die Ergebnisse direkt plotten lassen). Für die sonstige Syntax lasst euch wie immer vom LLM eurer Wahl beraten. Typischerweise werden interessantere Runs länger Dauern, daher ist es, wenn ihr mit der train-Funktion arbeitet definitiv sinnvoll, eine .sh-Datei zu nutzen. Auf dem Cluster dürft ihr längere Rechenvorhaben nicht in eurem Terminal ausführen, sondern müsst eure Jobs in eine Warteschlange schicken. Daher müssen wir dort immer .sh-Dateien verwenden. Mehr dazu und zur speziellen Syntax im nächsten Kapitel.

Python Debugging und die workspace-Datei Option zwei ist, den python-eigenen Debugger zu verwenden. Das ist sinnvoll vor allem wenn ihr Fehler in eurem Code finden wollt (mehr dazu gleich, wenn wir besprechen, wie wir eigene Algorithmen hinzufügen), dafür ist der Debugger

schließlich gedacht, oder wenn ihr Plotfunktionen öfter hintereinander mit fast denselben Flags aufrufen wollt, um die Plots nach und nach zu verschönern.

Um anzugeben, welche Konfigurationen ihr zum Debuggen (bzw. zum ausführen im Falle der Plot-Funktionen oder der enjoy-Funktion) zur Verfügung haben wollt, gibt es die code-workspace Datei. Damit der Debugger korrekt aktiviert ist, müsst ihr, anstatt `Open Folder...`, um in VSCode in euer Repo zu wechseln, `Open Workspace From File...` auswählen, zu eurem Repo navigieren, und dann die .code-workspace-Datei anklicken. Dann habt ihr alle spezifizierten Konfigurationen unter dem Dreieck mit dem Käfer (Bug!) in der linken Menüleiste unter dem GitHub-Symbol zur Verfügung. Wenn ihr dort hin-navigiert, könnt ihr oben aus einem Drop-Down Menü zwischen euren Konfigurationen wählen und diese ausführen.

Die generelle Syntax der Workspace-Datei könnt ihr wieder in Zusammenarbeit mit einem LLM lernen, die einzelnen Konfigurationen sehen aber folgendermaßen aus und müssen in der Liste `"configurations"` enthalten sein (Kommas nicht vergessen!):

```
{
    "name": "\<Name der Konfiguration\>",
    "type": "debugpy",
    "request": "launch",
    "program": "${workspaceFolder}/\<auszuführende\_funktion.py\>",
    "args": [
        "--\<Flag1\>", "\<Wert1\>", "\<Wert2\>",
        "--\<Flag2\>", "\<Wert1\>",
    ],
    "console": "integratedTerminal",
    "justMyCode": false,
    "pythonArgs": [
        "-Xfrozen_modules=off"
    ],
    "env": {
        "PYTHONPATH": "${workspaceFolder}",
        "PYDEVD_DISABLE_FILE_VALIDATION": "1"
    }
}
```

Das sieht erstmal recht unübersichtlich aus, aber im Prinzip müsst ihr nur spezifizieren, welchen Namen eure Konfiguration haben soll, welche Funktion aufgerufen werden soll, und welche Flags mit welchen Parametern genutzt werden sollen. In unserem Repo stehen euch schon einige Beispielkonfigurationen zur Verfügung. Ihr könnt einfach eine der Konfiguration, die halbwegs passen kopieren und basierend darauf nur den Namen und die Werte für die Flags anpassen. Der erste Block an Konfigurationen, die mit `Ex:` anfangen, hat Beispielkonfigurationen, die aufeinander abgestimmt sind und euch zeigen, wie ein Workflow aussehen könnte. Dann kommt eine Konfiguration, die wir für das testen von eigenen Algorithmen und vor allem auch, um zu schauen, ob sie ohne Fehler durchlaufen, benutzen können. Die restlichen, die mit `Dynamic:` anfangen, könnt ihr benutzen, um euch KOnfigurationen rauskopieren oder diese direkt zu ändern. Es gibt auch Syntax, die ermöglicht, euch beim ausführen der Konfiguration zu prompten, fehlende Werte einzugeben, bevor es losgeht (deshalb heißen sie dynamic!), probiert das einfach mal aus.

Die Debug-Konsole kann aber noch viel mehr als einfach Programme ausführen. Die grundlegende Idee ist, dass die Ausführung nicht nur gestoppt werden kann, wenn ein Fehler passiert (die Debug Konsole stoppt tatsächlich vor dem Fehler, wichtiges Detail damit ihr noch Zugriff auf alle Parameter direkt davor habt), sondern erlaubt es euch auch, sogenannte Breakpoints zu setzen, an denen das Programm angehalten wird.So lassen sich Variableninhalte überprüfen, Schleifenverhalten nachvollziehen oder Fehlerquellen Schritt für Schritt eingrenzen. Breakpoints könnt ihr setzen, indem ihr euren Cursor neben eine Code-Zeile bewegt, und dort den Kreis, der erscheint, anklickt. Sobald er rot ist, ist der Breakpoint gesetzt.

Darüber hinaus bietet das Debug-Tool die Möglichkeit, Programme schrittweise auszuführen.

Man kann eine Funktion Zeile für Zeile durchlaufen, über Funktionsaufrufe hinweg springen oder in deren Inneres wechseln. Das könnt ihr über das kleine Fenster, das erscheint, sobald ihr den Debugger ausführt, machen. Von links nach recht sind die Funktionen: Stoppen/Weiterlaufen, Nächste Zeile ausführen, in eine Funktion gehen (z.B. wenn ihr eine funktion aufruft in der Zeile, in deren Source-Code wechseln), aus einer Funktion gehen, Neustart, Abbruch.

Gleichzeitig werden in einer übersichtlichen Seitenleiste einige Funktionen bereitgestellt. Ganz unten wird euch angezeigt, welche Breakpoints gerade gesetzt sind und ihr könnt sie aktivieren und deaktivieren. Darüber wird der sogenannte Call Stack angezeigt. Das ist die hierarchische Struktur, wo euer Code gerade ist. Wenn ihr train.py aufruft (das wird typischerweise ganz unten sein), ruft diese Funktion dann den Code, der zum gewünschten Algorithmus gehört, auf (eine Ebene wird obendrauf auf dem Call-Stapel gelegt), der wiederum z.B. die Funktionen für die Policies aufruft, usw., wodurch nach und nach ein Stapel an Funktionen, die aufgerufen wurden und jeweils neue Funktionen aufrufen, entsteht. So ist klar ersichtlich ist, aus welchem Kontext eine Funktion aufgerufen wurde und in welcher Code-Zeile von welcher Unterfunktion ihr euch befindet. Auch komplexe Programme lassen sich damit strukturiert analysieren. Darüber befinden sich die Abschnitte Watch und Variables. In Variables werden alle Variablen, die der Code zu dem Zeipunkt definiert hat mitsamt ihren Werten übersichtlich angezeigt. Sie sind aufgeteilt in globale (hierarchisch über der aktuellen Ebene) und lokale (aktuell im Code-Abschnitt) Variablen. Bei großen Programmen sind dort typischerweise sehr viele Variablen zu finden, weshalb ihr mit einem Rechtsklick Dinge zum Watch bereich hinzufügen könnt (die übrigens auch nach dem Beenden der Session dort bleiben). Wenn ihr iterierte Objekte (Listen, Dictionaries, etc.) habt, könnt ihr die Werte, die angezeigt werden aufklappen, um mehr Details zu sehen. Ihr könnt auch Teile eines solchen Objektes zu Watch hinzufügen.

Schließlich gibt es noch die Debug Konsole, die sich in der Leiste neben dem Terminal befindet. In ihr habt ihr Zugriff auf alle definierten Variablen, könnt damit rechnen, oder neue Variablen daraus definieren (die dann auch in der Seitenleiste auftauchen).

Was machen die anderen Dateien und Ordner?

Es gibt nun noch einige Dateien im Repo, über die wir nicht gesprochen haben.

RLiable zugehörige Dateien Die Idee von RLiable ist es, wenn man wenige Experimente machen kann, trotzdem statistisch sinnvolle Aussagen zu liefern. Das hat den Hintergrund, dass es sehr lange üblich war, große Experimente, wie Atari-Spiele, mit nur 3 oder 5 Seeds (entspricht 3 bzw. 5 Wiederholungen im Spiel, N in Monte Carlo!) präsentiert hat. Das ist natürlich nicht sinnvoll. Das Paper zu RLiable postuliert daher zwei Ansätze. Zum einen möchte man Metriken verwenden, die weniger von Ausreißern dominiert werden. Dazu wird der Median und das Interquantilsmittel (IQM; Nur der Mittelwert der Werte zwischen dem 1. und 3. Quantil) vorgeschlagen. Zum anderen wird vorgeschlagen, die Scores von verschiedenen Spielen zu normalisieren, um sie vergleichbar zu machen. So kann man quasi aus N Spielen mit je 3 oder 5 Seeds, ein Spiel mit N*3 bzw. N*5 Seeds machen. Letzteres scheint zumindest (wenn es tatsächlich ein Spiel wäre) statistisch sinnvoller. Dazu wird bei Atari typischerweise mit 0$^=$random player und 1$^=$human expert player (die Idee war zu schauen, ob man human performance erreichen kann). Daraus ergibt sich eine weitere Metrik: Optimality Gap, also der Anteil an Runs, der nicht Performance größer gleich 1 erreicht.

Prinzipiell kann man natürlich normalisieren wie man will. Bei den MuJoCo-Environments beispielsweise gibt es keine wirklich sinnvolle Art der Normierung außer vielleicht 0$^=$kleinster beobachteter Wert und 1$^=$größter beobachteter Wert. Die Normalisierungswerte für jedes Environment kann man in der Datei `rl_zoo3/plots/score_normalization.py` anpassen. Dort findet ihr für Atari die Werte der Human Benchmark und für einige der MuJoCo-Environments Werte, die uns sinnvoll erschienen aus Beobachtungen.

Jetzt können wir auch die zusätzlichen Plots verstehen in der plot_from_file Funktion. Der Boxplot gibt die Sensitivitätsanalyse für das Zusammenwerfen der Environments. Die RLiable Metriken umfassen Mean, Meadian, IQM, und Optimality Gap für alle Algorithmen, gemittelt über alle Spiele und die Performance Profiles, zu welchem Zeitpunkt welcher Anteil der Seeds (über alle Spiele) noch nicht Score 1 erreicht hat. Der Probability of Improvement Plot enthält Vergleiche zwischen allen inkludierten Algorithmen, welcher Anteil an Runs besser war. Schließlich plottet der IQM sample efficiency Plot die sample efficiency gegen das Interquantilmittel.

Hyperparameter Dateien Im Ordner Hyperparameter befinden sich .yml-Dateien, in dem die Hyperparameter festgelegt sind, die standardmäßig benutzt werden. Stable baselines stellt für viele Environments bereits getunete Parameter bereit und ansonsten standard-parameter, die für Benchmarks sinnvoll sind. Die Syntax könnt ihr euch wie immer vom LLM erklären lassen.

Sonstige Dateien in rl_zoo3 Ansonsten gibt es noch einige weitere Dateien im Ordner rl_zoo3. Von oben nach unten machen sie folgendes:

- `__init__.py`: Legt fest, welche Objekte bekannt sind. Müsst ihr eigentlich nie anfassen.
- `benchmark.py`: Hier könnt ihr die Stable Baselines3 benchmark einmal komplett durchtrainieren lassen und später eure Algorithmen miteinbeziehen. Aufgrund von begrenzter Rechenkapazität werden wir das aber ziemlich sicher nicht benutzen.
- `callbacks.py`: Regelt den switch zwischen Evaluations- und Trainingsschleife. Müsst ihr eigentlich nie anfassen.
- `cli.py`: Command line interface, erklärt, was das Modell „kennt“. Müsst ihr eigentlich nie anfassen.
- `exp_manager.py`: Erstellt Ordner, sogenannte Tensorboards (machen wir nicht), Parallelisieren von Prozessen, Konfigurationen prüfen, einstellen, und ordnen, speichern von Dateien. Müsst ihr eigentlich nie anfassen.
- `gym_patches.py`: Behebt einige inkompatible Probleme mit dem gym Paket. Müsst ihr nie anfassen.
- `hyperparams_opt.py`: Hier könnt ihr festlegen, auf welche Art und Weise der optimierer, den ihr in der train Funktion nutzen könnt, Parameter zufällig zieht und welche davon er zufällig ziehen soll. Wird im nächsten Abschnitt genauer erklärt. Damit werden wir vermutlich arbeiten.
- `import_envs.py`: Importiert weitere environments, wenn ihr in der train Funktion angebt, welche ihr zusätzlich haben wollt. Damit werden wir vermutlich nicht arbeiten.
- `load/push_to_hub`: Ermöglicht, auf hubs, die Modelle sammeln und speichern, hochzuladen oder Modelle herunterladen. Stable Baselines stellt z.B. ein Hub mit vielen trainierten Modellen zur Verfügung. Damit werden wir vermutlich nicht arbeiten.
- `py.typed`: Wird für die type-erkennung gebracht. Müsst ihr nie anfassen.
- `record_training/video.py`: Damit könnt ihr während des Trainings Videos eurer Agenten erstellen und abspeichern. record_video wird dabei verwendet, wenn ihr es in Kombination mit dem benchmarking oder parallelen environments verwenden wollt. Damit werden wir vermutlich nicht arbeiten.
- `utils.py`: Verschiedenste tools. Müsst ihr eigentlich nie anfassen.
- `version.txt`: Aktuelle Version. Müsst ihr nie anfassen.
- `wrappers.py`: Enthält sogenannte wrapper, die Standardkonfigurationen für Environments definieren. Müsst ihr selten anfassen, werden wir aber eventuell nutzen.

Eigene Algorithmen hinzufügen

Im nächsten Abschnitt erst werden wir die genauere Funktionsweise der Algorithmen in Stable Baselines kennenlernen. Daher können wir an dieser Stelle natürlich noch keinen wirklich eigenen Algorithmus hinzufügen. Wir werden also einfach Code von einem der Stable Baselines Algorithmen nehmen und uns dann schauen, was wir machen müssen, damit alles problemlos funktioniert. Typischerweise werden wir ohnehin alle unsere Algorithmen basierend auf Stable Baselines Code und Algorithmen coden, um Vergleichbarkeit zu gewährleisten, wobei wir dann einfache Tricks einfügen.

Ordner erstellen/kopieren und Namen setzen Im Ordner `rl_zoo3` befindet sich ein Ordner namens `custom_algos`, in den wir die Ordner kopieren werden, die wir für die Algorithmen brauchen. Dazu können wir auf dem Pfad `venv/lib` den Unterordner `stable_baselines3/ppo` finden und nach `custom_algos` kopieren. Dann müssen wir den Namen ändern in den Namen des neuen Algorithmus (Kleinbuchstaben-Schreibweise) und im Ordner die Datei, die ppo.py heißt zum den Namen des neuen Algorithmusses ändern. Die Datei policies.py importiert Standardpolicies aus dem stable_baselines paket. Wenn wir später noch die policies, die dann neuronale Netze sein werden anpassen wollen, müssen wir den Code aus den jeweiligen Dateien holen. Zum Glück wird euch der Pfad zu den Dateien ja direkt angezeigt! In der Datei, die nun den Namen eures Algorithmusses tragen sollte, müsst ihr nun 7 Mal den Namen umändern, wo aktuell PPO stehen sollte. Drei mal in der Zeile, die mit Self startet ganz oben, einmal in der Klassendefinition direkt darunter, und drei Mal ganz am Ende der Datei in der Funktion `learn`. In allen drei Fällen müsst ihr Großschreibung verwenden. Schließlich müsst ihr in der Datei ___init___.py noch die Pfade ändern, die ihr zum Importieren der Objekte braucht. Das macht ihr, indem ihr folgende Zeilen stattdessen einfügt:

```
from rl\_zoo.custom\_algos.\<algo\_name\>.policies import CnnPolicy, ...
rl\_zoo.custom\_algos.\<algo\_name\>.\<algo\_name\> import \<ALGONAME\>
```

und in der Liste `__all__` den Namen PPO durch den eures Algos austauscht.

Verpflichtende Änderungen Als erstes müsst ihr in der ___init___.py-Datei des Ordners `custom_algos` die Objekte aus dem neu eingefügten Ordner importieren. Für einen Algorithmus namens `EXAMPLE` ist ein Beispiel dafür auskommentiert. Beachtet wieder die Großschreibung. Als nächstes müsst ihr in der Datei utils.py die Algorithmen importieren. Dazu müsst ihr in die Liste in der Zeile unter `# custom algos` den Namen eures Algos schreiben. Vergesst nicht das Komma danach und beachtet, dass am Anfang nur ein Platzhalter drin ist. Zudem müsst ihr im dictionary `ALGOS`, das sich direkt darunter befindet die Kleinschreibweise zur Großschreibweise mappen. Schließlich müsst ihr noch eine .yml-Datei im Ordner hyperparams hinzufügen, die den Namen eures Algos in Kleinschreibweise trägt. In der Regel kopiert ihr einfach die Datei des Algos, auf dem euer neuer Algo basiert, und benennt sie um.

Optionale Änderungen Die folgenden Änderungen sind nur dann notwendig, wenn ihr bestimmte Funktionen korrekt mit euren Algos nutzen wollt.

Stable Baselines3 unterscheidet Offline- und Onlinealgos als Klassen (Achtung: offline und online sind etwas anders als in der Vorlesung definiert). Manche Funktionen müssen für Off-Policy Algos anders agieren. Daher finden sich in den Dateien benchmark.py, enjoy.py, und record_video.py Listen namens off_policy_algos, in die ihr euren Algo eintragen müsst, um deren Funktionen zu nutzen.

Außerdem müsst ihr, falls ihr den optimierer in der train-Funktion nutzen wollt, noch in der Datei hyperparams_opt.py festlegen, welche Parameter und auf welche Art und Weise der Optimierer die Parameter ausprobieren soll. Für die Oberklassen Onpolicy- und Offpolicyalgorithmen stehen converter Funktionen für einige Hyperparameter, die sie alle gemein haben und bei denen man nicht einfach einen Wert übergibt, zur Verfügung. Außerdem hat jeder der

vorimplementierten Algorithmen eine eigene Funktion namens `sample_<algo_name>_params`, in der mit der trail methode festgelegt wird, welche parameter wie gezogen werden sollen (z.B. gibt trial.suggest_float("<parameter_name>", <lower_value>, <upper_value>) eine float-zahl zwischen lower_value und upper_value). Ihr müsst dann in den Dictionaries `HYPERPARAMS_SAMPLER` und `HYPERPARAMS_CONVERTER` den namen eures neuen Algorithmus (in Kleinschreibweise) der sampler- bzw. Converterfunktion, die ihr nutzen wollt, zuordnen.

Testen Wenn ihr nun alle Schritte befolgt habt, könnt ihr testen, ob euer Algorithmus läuft und das macht, was er sollte. Ihr könnt beispielsweise mit dem Befehl `python train.py <neuer_algo_name>` schauen, ob er losläuft oder bereits am Anfang Fehler auftreten. Ihr könnt aber auch die Debug-Konfiguration `Dev-run: train dev run` nutzen. Damit seht ihr, ob euer Code auch bis zum Ende durchläuft. Außerdem könnt ihr natürlich auch schauen, ob einzelne Code-Abschnitte auch das machen, was sie sollen. Generell ist es empfehlenswert, wenn euer Code auf einem Algo beruht, gerade wenn dieser ein Spezialfall eures neuen Algos ist, die Ergebnisse eures neuen Codes mit dem eigentlichen Algo zu vergleichen, um kleine Implementierungsfehler zu vermeiden.

An dieser Stelle noch eine kleine Anmerkung: Der Workspace enthält noch einen Ordner custom_envs, der für das Ablegen neuer Environments gedacht ist. Diese Funktion werden wir sehr wahrscheinlich nicht nutzen, aber das Einfügen funktioniert recht ähnlich zum oben beschriebenen Vorgang.

0.3.3 Stable Baselines 3 und SB3 Contrib

0.4 Rechnen auf den Clustern des Landes BaWü

Kapitel 1

TBD