

EXPERT INSIGHT

Deep Reinforcement Learning Hands-On

A practical and easy-to-follow guide to RL from Q-learning and DQNs to PPO and RLHF

Third Edition

Maxim Lapan

packt

Deep Reinforcement Learning Hands-On

Third Edition

A practical and easy-to-follow guide to RL from
Q-learning and DQNs to PPO and RLHF

Maxim Lapan



Deep Reinforcement Learning Hands-On

Third Edition

Copyright © 2024 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Lead Senior Publishing Product Manager: Bhavesh Amin

Acquisition Editor – Peer Reviews: Swaroop Singh

Project Editor: Amisha Vathare

Development Editor: Shruti Menon

Copy Editor: Safis Editing

Technical Editor: Kushal Sharma

Indexer: Hemangini Bari

Proofreader: Safis Editing

Presentation Designer: Pranit Padwal

Developer Relations Marketing Executive: Anamika Singh

First published: June 2018

Second edition: January 2020

Third edition: October 2024

Production reference: 1071124

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK.

ISBN 978-1-83588-270-2

www.packtpub.com

To my loving family and friends.

- Maxim Lapan

Contributors

About the author

Maxim Lapan is a deep learning and machine learning enthusiast. His background and 20 years of work expertise as a software developer and a systems architect covers everything from low-level Linux kernel driver development to performance optimization and the design of distributed applications working on thousands of servers. With extensive work experience in big data, machine learning, deep learning, and large parallel distributed HPC and non-HPC systems, he has the ability to explain complex concepts using simple words and clear examples. His current area of interest is the practical applications of deep learning, such as deep natural language processing, large language models, and deep reinforcement learning.

Maxim lives in Gronau (North Rhine-Westphalia), Germany, with his family.

I'd like to thank my family: my wife, Olga, and my children, Ksenia, Julia, and Fedor, for their patience and support. The third edition was written during a turbulent time for our family, and without you, this would not have been doable.

I also want to thank my business partner, Arnout. Thanks for your patience and support!

About the reviewers

Daniel Armstrong is an applied data scientist specializing in natural language processing, conversational AI, and computer vision. With experience in the finance and consumer goods industries, he develops AI-powered solutions that enhance customer experiences and streamline operations. He is well versed in machine learning, deep learning, and knowledge management. Currently, Daniel is focusing on LLM-powered agents that integrate structured dialogue models and logical reasoning frameworks to build context-aware systems for automation and decision-making.

Michael Yurushkin holds a PhD in computer science. He is an expert in machine learning and software development, and is passionate about creating products using cutting-edge technology. Michael is an associate professor at Southern Federal University (SFedU), where he teaches deep learning with applications in computer vision and natural language processing, while also supervising student research. Michael enjoys playing chess and currently holds the title of Candidate Master.

Join our community on Discord

Read this book alongside other users, deep learning experts, and the author himself.

Ask questions, provide solutions to other readers, chat with the author via Ask Me Anything sessions, and much more.

Scan the QR code or visit the link to join the community:

<https://packt.link/r1>



Table of Contents

Preface	xxi
<hr/>	
Part I: Introduction to RL	1
<hr/>	
Chapter 1: What Is Reinforcement Learning?	3
<hr/>	
Supervised learning	4
Unsupervised learning	4
Reinforcement learning	5
Complications in RL	6
RL formalisms	7
Reward • 8	
The agent • 9	
The environment • 10	
Actions • 10	
Observations • 11	
The theoretical foundations of RL	14
Markov decision processes • 14	
<i>The Markov process</i> • 14	
<i>Markov reward processes</i> • 19	
<i>Adding actions to MDP</i> • 22	
Policy • 25	
Summary	26
<hr/>	
Chapter 2: OpenAI Gym API and Gymnasium	27
<hr/>	
The anatomy of the agent	28

Hardware and software requirements	31
The OpenAI Gym API and Gymnasium	33
The action space • 34	
The observation space • 34	
The environment • 36	
Creating an environment • 38	
The CartPole session • 40	
The random CartPole agent	43
Extra Gym API functionality	44
Wrappers • 44	
Rendering the environment • 47	
More wrappers • 49	
Summary	49
Chapter 3: Deep Learning with PyTorch	51
Tensors	52
The creation of tensors	52
Scalar tensors • 55	
Tensor operations • 56	
GPU tensors • 56	
Gradients	58
Tensors and gradients • 60	
NN building blocks	63
Custom layers	64
Loss functions and optimizers	67
Loss functions • 68	
Optimizers • 68	
Monitoring with TensorBoard	70
TensorBoard 101 • 71	
Plotting metrics • 72	
GAN on Atari images	74
PyTorch Ignite	80
Ignite concepts • 81	
GAN training on Atari using Ignite • 82	
Summary	85

Chapter 4: The Cross-Entropy Method	87
The taxonomy of RL methods	88
The cross-entropy method in practice	89
The cross-entropy method on CartPole	91
The cross-entropy method on FrozenLake	101
The theoretical background of the cross-entropy method	109
Summary	110
Part II: Value-based methods	111
Chapter 5: Tabular Learning and the Bellman Equation	113
Value, state, and optimality	114
The Bellman equation of optimality	116
The value of the action	118
The value iteration method	121
Value iteration in practice	123
Q-iteration for FrozenLake	130
Summary	132
Chapter 6: Deep Q-Networks	133
Real-life value iteration	134
Tabular Q-learning	135
Deep Q-learning	140
Interaction with the environment • 142	
SGD optimization • 143	
Correlation between steps • 143	
The Markov property • 144	
The final form of DQN training • 144	
DQN on Pong	145
Wrappers • 146	
The DQN model • 151	
Training • 153	
Running and performance • 164	
Your model in action • 167	
Things to try	169

Summary	170
Chapter 7: Higher-Level RL Libraries 171	
Why RL libraries?	172
The PTAN library	172
Action selectors • 174	
The agent • 176	
<i>DQNAgent</i> • 177	
<i>PolicyAgent</i> • 179	
Experience source • 180	
<i>Toy environment</i> • 181	
<i>The ExperienceSource class</i> • 182	
<i>The ExperienceSourceFirstLast Class</i> • 184	
Experience replay buffers • 186	
The TargetNet class • 188	
Ignite helpers • 190	
The PTAN CartPole solver	190
Other RL libraries	193
Summary	194
Chapter 8: DQN Extensions 195	
Basic DQN	196
Common library • 196	
Implementation • 202	
Hyperparameter tuning • 203	
Results with common parameters • 207	
Tuned baseline DQN • 209	
N-step DQN	210
Implementation • 213	
Results • 213	
Hyperparameter tuning • 215	
Double DQN	215
Implementation • 216	
Results • 218	
Hyperparameter tuning • 219	

Noisy networks	219
Implementation • 220	
Results • 223	
Hyperparameter tuning • 225	
Prioritized replay buffer	225
Implementation • 226	
Results • 231	
Hyperparameter tuning • 232	
Dueling DQN	233
Implementation • 235	
Results • 236	
Hyperparameter tuning • 237	
Categorical DQN	237
Implementation • 240	
Results • 246	
Hyperparameter tuning • 248	
Combining everything	248
Results • 249	
Hyperparameter tuning • 251	
Summary	252
Chapter 9: Ways to Speed Up RL	253
Why speed matters	253
Baseline	256
The computation graph in PyTorch	258
Several environments	261
Playing and training in separate processes	264
Tweaking wrappers	268
Benchmark results	271
Summary	272
Chapter 10: Stocks Trading Using RL	273
Why trading?	273
Problem statement and key decisions	274
Data	276

The trading environment	277
Models	285
Training code	287
Results	288
The feed-forward model • 288	
The convolution model • 293	
Things to try	294
Summary	295
Part III: Policy-based methods	297
Chapter 11: Policy Gradients	299
Values and policy	299
Why the policy? • 300	
Policy representation • 301	
Policy gradients • 302	
The REINFORCE method	302
The CartPole example • 304	
Results • 308	
Policy-based versus value-based methods • 310	
REINFORCE issues	310
Full episodes are required • 310	
High gradient variance • 311	
Exploration problems • 312	
High correlation of samples • 312	
Policy gradient methods on CartPole	313
Implementation • 313	
Results • 316	
Policy gradient methods on Pong	319
Implementation • 320	
Results • 321	
Summary	321
Chapter 12: Actor-Critic Method: A2C and A3C	323
Variance reduction	324

CartPole variance	325
Advantage actor-critic (A2C)	328
A2C on Pong • 331	
Results • 338	
Asynchronous Advantage Actor-Critic (A3C)	342
Correlation and sample efficiency • 342	
Adding an extra “A” to A2C • 343	
A3C with data parallelism • 346	
<i>Results</i> • 346	
A3C with gradient parallelism • 347	
<i>Implementation</i> • 347	
<i>Results</i> • 353	
Summary	353
Chapter 13: The TextWorld Environment	355
Interactive fiction	355
The environment	359
Installation • 360	
Game generation • 360	
Observation and action spaces • 361	
Extra game information • 362	
The deep NLP basics	364
Recurrent Neural Networks (RNNs) • 365	
Word embedding • 367	
The Encoder-Decoder architecture • 368	
Transformers • 369	
Baseline DQN	369
Observation preprocessing • 371	
Embeddings and encoders • 376	
The DQN model and the agent • 378	
Training code • 379	
Training results • 380	
Tweaking observations	382
Tracking visited rooms • 382	
Relative actions • 384	

Objective in observation • 386	
Transformers	387
ChatGPT	389
Setup • 389	
Interactive mode • 389	
ChatGPT API • 392	
Summary	395
 Chapter 14: Web Navigation 397	
The evolution of web navigation	397
Browser automation and RL	398
Challenges in browser automation	399
The MiniWoB benchmark	400
MiniWoB++	401
Installation • 401	
Actions and observations • 402	
Simple example • 402	
The simple clicking approach	406
Grid actions • 406	
The RL part of our implementation • 409	
The model and training code • 410	
Training results • 410	
Simple clicking limitations • 412	
Adding text description	414
Implementation • 414	
Results • 419	
Human demonstrations	420
Recording the demonstrations • 421	
Training with demonstrations • 423	
Results • 425	
Things to try	427
Summary	427

Part IV: Advanced RL	429
<hr/>	
Chapter 15: Continous Action Space	431
<hr/>	
Why a continuous space?	432
The action space • 432	
Environments • 433	
The A2C method	435
Implementation • 436	
Results • 440	
Using models and recording videos • 441	
Deep deterministic policy gradients	442
Exploration • 444	
Implementation • 445	
Results and video • 450	
Distributional policy gradients	453
Architecture • 453	
Implementation • 454	
Results • 457	
Things to try	459
Summary	459
<hr/>	
Chapter 16: Trust Region Methods	461
<hr/>	
Environments	462
The A2C baseline	463
Implementation • 463	
Results • 465	
Video recording • 468	
PPO	469
Implementation • 470	
Results • 474	
TRPO	476
Implementation • 477	
Results • 479	
ACKTR	481
Implementation • 481	

Results • 482	
SAC	483
Implementation • 484	
Results • 486	
Overall results	489
Summary	489
Chapter 17: Black-Box Optimizations in RL	491
Black-box methods	491
Evolution strategies	492
Implementing ES on CartPole • 493	
CartPole results • 499	
ES on HalfCheetah • 500	
Implementing ES on HalfCheetah • 501	
HalfCheetah results • 505	
Genetic algorithms	507
GA on CartPole • 508	
GA tweaks • 510	
<i>Deep GA</i> • 510	
<i>Novelty search</i> • 510	
GA on HalfCheetah • 511	
<i>Implementation</i> • 511	
<i>Results</i> • 514	
Summary	515
Chapter 18: Advanced Exploration	517
Why exploration is important	518
What's wrong with ϵ -greedy?	518
Alternative ways of exploration	522
Noisy networks • 522	
Count-based methods • 523	
Prediction-based methods • 524	
MountainCar experiments	524
DQN + ϵ -greedy • 525	
DQN + noisy networks • 527	

DQN + state counts • 528	
PPO method • 530	
PPO + Noisy Networks • 532	
PPO + state counts • 533	
PPO + network distillation • 534	
Comparison of methods • 537	
Atari experiments	537
DQN + ϵ -greedy • 538	
DQN + noisy networks • 539	
PPO • 539	
Summary	539
<hr/>	
Chapter 19: Reinforcement Learning with Human Feedback	541
<hr/>	
Reward functions in complex environments	542
Theoretical background	544
Method overview • 544	
RLHF and LLMs • 546	
RLHF experiments	547
Initial training using A2C • 548	
Labeling process • 552	
Reward model training • 556	
Combining A2C with the reward model • 560	
Fine-tuning with 100 labels • 563	
The second round of the experiment • 564	
The third round of the experiment • 565	
Overall results • 567	
Summary	567
<hr/>	
Chapter 20: AlphaGo Zero and MuZero	569
<hr/>	
Comparing model-based and model-free methods	570
Model-based methods for board games	571
The AlphaGo Zero method	572
Overview • 572	
MCTS • 573	
Self-play • 575	

Training and evaluation • 576	
Connect 4 with AlphaGo Zero	576
The game model • 578	
Implementing MCTS • 580	
The model • 585	
Training • 588	
Testing and comparison • 588	
Results • 588	
MuZero	591
High-level model • 592	
Training process • 593	
Connect 4 with MuZero	594
Hyperparameters and MCTS tree nodes • 594	
Models • 598	
MCTS search • 601	
Training data and gameplay • 604	
MuZero results	609
MuZero and Atari	610
Summary	611
Chapter 21: RL in Discrete Optimization	613
The Rubik’s cube and discrete optimization	614
Optimality and God’s number	615
Approaches to cube solving	616
Actions • 617	
States • 618	
The training process	621
The NN architecture • 622	
The training • 623	
The model application	624
Results	626
The code outline	628
Cube environments • 629	
Training • 633	
The search process • 634	

The experiment results	635
The 2×2 cube • 637	
The 3×3 cube • 639	
Further improvements and experiments	641
Summary	641
Chapter 22: Multi-Agent RL	643
What is multi-agent RL?	644
Getting started with the environment	645
An overview of MAgent • 645	
Installing MAgent • 646	
Setting up a random environment • 646	
Deep Q-network for tigers	652
Understanding the code • 652	
Training and results • 656	
Collaboration by the tigers	659
Training both tigers and deer	661
The battle environment	662
Summary	663
Other Books You May Enjoy	671
Index	675

Preface

This book is on **reinforcement learning (RL)**, which is a subfield of **machine learning (ML)**; it focuses on the general and challenging problem of learning optimal behavior in complex environments. The learning process is driven only by the reward value and observations obtained from the environment. This model is very general and can be applied to many practical situations, from playing games to optimizing complex manufacturing processes. We largely focus on **deep RL** in this book, which is RL that leverages **deep learning (DL)** methods.

Due to its flexibility and generality, the field of RL is developing very quickly and attracting lots of attention, both from researchers who are trying to improve existing methods or create new methods and from practitioners interested in solving their problems in the most efficient way.

Why I wrote this book

There is a lot of ongoing research activity in the RL field all around the world. New research papers are being published almost every day, and a large number of DL conferences, such as **Neural Information Processing Systems (NeurIPS)** or the **International Conference on Learning Representations (ICLR)**, are dedicated to RL methods. There are also several large research groups focusing on the application of RL methods to robotics, medicine, multi-agent systems, and others.

However, although information about the recent research is widely available, it is too specialized and abstract to be easily understandable. Even worse is the situation surrounding the practical aspect of RL, as it is not always obvious how to make the step from an abstract method described in its mathematics-heavy form in a research paper to a working implementation solving an actual problem.

This makes it hard for somebody interested in the field to get a clear understanding of the methods and ideas behind papers and conference talks.

There are some very good blog posts about various aspects of RL that are illustrated with working examples, but the limited format of a blog post allows authors to describe only one or two methods, without building a complete structured picture and showing how different methods are related to each other in a systematic way. This book was written as an attempt to fill this obvious gap in practical and structured information about RL methods and approaches.

The approach

A key aspect of the book is its orientation to practice. Every method is implemented for various environments, from the very trivial to the quite complex. I've tried to make the examples clean and easy to understand, which was made possible by the expressiveness and power of PyTorch. On the other hand, the complexity and requirements of the examples are oriented to RL hobbyists without access to very large computational resources, such as clusters of **graphics processing units (GPUs)** or very powerful workstations. This, I believe, will make the fun-filled and exciting RL domain accessible to a much wider audience than just research groups or large artificial intelligence companies. On the other hand, this is still deep RL, so access to a GPU is highly recommended, as computation speed up will make experimentations much more convenient (waiting for several weeks for a single optimization to complete is not very fun). Approximately half of the examples in the book will benefit from being run on a GPU.

In addition to traditional medium-sized examples of environments used in RL, such as Atari games or continuous control problems, this book contains several chapters (10, 13, 14, 19, 20, and 21) that contain larger projects, illustrating how RL methods can be applied to more complicated environments and tasks. These examples are still not full-sized, real-life projects (they would occupy a separate book on their own), but just larger problems illustrating how the RL paradigm can be applied to domains beyond the well-established benchmarks.

Another thing to note about the examples in Parts 1, 2, and 3 of the book is that I've tried to make them self-contained, with the source code shown in full. Sometimes this has led to the repetition of code pieces (for example, the training loop is very similar in most of the methods), but I believe that giving you the freedom to jump directly into the method you want to learn is more important than avoiding a few repetitions. All examples in the book are available on GitHub at <https://github.com/PacktPublishing/Deep-Reinforcement-Learning-Hands-On-3E/>, and you're welcome to fork them, experiment, and contribute.

Besides the source code, several chapters (15, 16, 19, and 22) are accompanied by video recordings of the trained model. All these recordings are available in the following YouTube playlist: <https://youtube.com/playlist?list=PLMVwuZENsfJmjP1BuFy5u7c3uStMTJYz7>.

Who this book is for

This book is ideal for machine learning engineers, software engineers, and data scientists looking to learn and apply deep RL in practice. It assumes familiarity with Python, calculus, and ML concepts. With practical examples and high-level overviews, it's also suitable for experienced professionals looking to deepen their understanding of advanced deep RL methods and apply them across industries, such as gaming and finance.

What this book covers

Chapter 1, What Is Reinforcement Learning?, contains an introduction to RL ideas and the main formal models.

Chapter 2, OpenAI Gym API and Gymnasium, introduces the practical aspects of RL, using the open source library Gym and its descendant, Gymnasium.

Chapter 3, Deep Learning with PyTorch, gives you a quick overview of the PyTorch library.

Chapter 4, The Cross-Entropy Method, introduces one of the simplest methods in RL to give you an impression of RL methods and problems.

Chapter 5, Tabular Learning and the Bellman Equation, this chapter opens Part 2 of the book, devoted to value-based family of methods.

Chapter 6, Deep Q-Networks, describes **deep Q-networks (DQNs)**, an extension of the basic value-based methods, allowing us to solve complicated environments.

Chapter 7, Higher-Level RL Libraries, describes the library PTAN, which we will use in the book to simplify the implementations of RL methods.

Chapter 8, DQN Extensions, gives a detailed overview of a modern extension to the DQN method, to improve its stability and convergence in complex environments.

Chapter 9, Ways to Speed up RL Methods, provides an overview of ways to make the execution of RL code faster.

Chapter 10, Stocks Trading Using RL, is the first practical project and focuses on applying the DQN method to stock trading.

Chapter 11, Policy Gradients, opens Part 3 of the book and introduces another family of RL methods that is based on direct policy optimisation.

Chapter 12, The Actor-Critic Method: A2C and A3C, describes one of the most widely used policy-based method in RL.

Chapter 13, The TextWorld Environment, covers the application of RL methods to interactive fiction games.

Chapter 14, Web Navigation, is another long project that applies RL to web page navigation using the MiniWoB++ environment.

Chapter 15, Continuous Action Space, opens the *advanced RL* part of the book and describes the specifics of environments using continuous action spaces and various methods (widely used in robotics).

Chapter 16, Trust Regions, is yet another chapter about continuous action spaces describing the trust region set of methods: PPO, TRPO, ACKTR and SAC.

Chapter 17, Black-Box Optimization in RL, shows another set of methods that don't use gradients in their explicit form.

Chapter 18, Advanced Exploration, covers different approaches that can be used for better exploration of the environment — a very important aspect of RL.

Chapter 19, Reinforcement Learning with Human Feedback, introduces and implements recent approach to guide the process of learning by giving human feedback. This method is widely used in training **large language models (LLMs)**. In this chapter, we'll implement RLHF pipeline from scratch and check its efficiency.

Chapter 20, AlphaGo Zero and MuZero, describes the AlphaGo Zero method and its evolution into MuZero, and applies both these methods to the game Connect 4.

Chapter 21, RL in Discrete Optimization, describes the application of RL methods to the domain of discrete optimization, using the Rubik's cube as an environment.

Chapter 22, Multi-Agent RL, introduces a relatively new direction of RL methods for situations with multiple agents.

To get the most out of this book

This book is suitable for you if you're using a machine with at least 32 GB of RAM. A GPU is not strictly required, but an Nvidia GPU is highly recommended. The code has been tested on Linux and macOS. For more details on the hardware and software requirements, refer to Chapter 2.

All the chapters in this book that describe RL methods have the same structure: in the beginning, we discuss the motivation of the method, its theoretical foundation, and the idea behind it. Then, we follow several examples of the method applied to different environments with the full source code.

You can use the book in different ways:

- To quickly become familiar with a particular method, you can read only the introductory part of the relevant chapter

- To get a deeper understanding of the way the method is implemented, you can read the code and the explanations accompanying it
- To gain a deeper familiarity with the method (which I believe is the best way to learn) you can try to reimplement the method and make it work, using the provided source code as a reference point

Whichever approach you choose, I hope the book will be useful for you!

Changes in the third edition

In comparison to the second edition of this book (published in 2020), there are several major changes made to the book's contents in this new edition:

- All the dependencies of code examples have been updated to the recent versions or replaced with better alternatives. For example, OpenAI Gym is not supported anymore, but we have the Farama Foundation Gymnasium fork. Another example is the MiniWoB++ library, which has replaced the MiniWoB and Universe environment.
- A new chapter on RLHF has been included, and the MuZero method has been added to the chapter on AlphaGo Zero.
- There are lots of small fixes and improvements — most of the figures have been redrawn to make them clearer and more easily understandable.

To better meet book volume limitations, several chapters were rearranged, which I hope made the book more consistent and easier to read.

Download the example code files

The code bundle for the book is hosted on GitHub at <https://github.com/PacktPublishing/Deep-Reinforcement-Learning-Hands-On-Third-Edition>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: <https://packt.link/gbp/9781835882702>.

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: “For the reward table, it is represented as a tuple with [State, Action, State] and for the transition table, it is written as [State, Action].”

A block of code is set as follows:

```
import typing as tt
import gymnasium as gym
from collections import defaultdict, Counter
from torch.utils.tensorboard.writer import SummaryWriter

ENV_NAME = "FrozenLake-v1"
GAMMA = 0.9
TEST_EPISODES = 20
```

Any command-line input or output is written as follows:

```
>>> e.action_space
Discrete(2)
>>> e.observation_space
Box([-4.8000002e+00 -3.4028235e+38 -4.1887903e-01 -3.4028235e+38], [4.8000002e+00
3.4028235e+38 4.1887903e-01 3.4028235e+38], (4,), float32)
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: “The second term is called **cross-entropy**, which is a very common optimization objective in deep learning.”

Citations are represented using a condensed author–year format within square brackets, similar to [Sut88] or [Kro+11]. You can find the details of the corresponding paper in the *Bibliography* section at the end of the book.



Warnings or important notes appear like this.



Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email feedback@packtpub.com and mention the book's title in the subject of your message. If you have questions about any aspect of this book, please email us at questions@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit <http://www.packtpub.com/submit-errata>, click **Submit Errata**, and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packtpub.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit <http://authors.packtpub.com>.

Leave a Review!

Thank you for purchasing this book from Packt Publishing—we hope you enjoy it! Your feedback is invaluable and helps us improve and grow. Once you've completed reading it, please take a moment to leave an Amazon review; it will only take a minute, but it makes a big difference for readers like you.

Scan the QR code below to receive a free ebook of your choice.



<https://packt.link/Nz0WQ>

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere? Is your eBook purchase not compatible with the device of your choice?

Don't worry; with every Packt book, you now get a DRM-free PDF version of that book at no cost.

Read anywhere, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there! You can get exclusive access to discounts, newsletters, and great free content in your inbox daily.

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below:



<https://packt.link/free-ebook/9781835882702>

2. Submit your proof of purchase.
3. That's it! We'll send your free PDF and other benefits to your email address directly.

PART I

INTRODUCTION TO RL

1

What Is Reinforcement Learning?

The automatic learning of optimal decisions over time is a general and common problem that has been studied in many scientific and engineering fields. In our changing world, even problems that look like static input-output problems can become dynamic if time is taken into account. For example, imagine that you want to solve the simple supervised learning problem of pet image classification with two target classes—dog and cat. You gather the training dataset and implement the classifier using your favorite deep learning toolkit. After the training and validation, the model demonstrates excellent performance. Great! You deploy it and leave it running for a while. However, after a vacation at some seaside resort, you return to discover that dog grooming fashions have changed and a significant portion of your queries are now misclassified, so you need to update your training images and repeat the process again. Not so great!

This example is intended to show that even simple **machine learning (ML)** problems often have a hidden time dimension. This is frequently overlooked and might become an issue in a production system. This can be addressed by **reinforcement learning (RL)**, a subfield of ML, which is an approach that natively incorporates an extra dimension (which is usually time, but not necessarily) into learning equations. This places RL much closer to how people understand **artificial intelligence (AI)**.

In this chapter, we will discuss RL in more detail and you will become familiar with the following:

- How RL is related to and differs from other ML disciplines: **supervised and unsupervised learning**
- What the main RL formalisms are and how they are related to each other
- Theoretical foundations of RL: **Markov processes (MPs)**, **Markov reward processes (MRPs)**, and **Markov decision processes (MDPs)**

Supervised learning

You may be familiar with the notion of supervised learning, which is the most studied and well-known ML problem. Its basic question is, how do you automatically build a function that maps some input into some output when given a set of example pairs? It sounds simple in those terms, but the problem includes many tricky questions that computers have only recently started to address with some success. There are lots of examples of supervised learning problems, including the following:

- **Text classification:** Is this email message spam or not?
- **Image classification and object location:** Does this image contain a picture of a cat, dog, or something else?
- **Regression problems:** Given the information from weather sensors, what will be the weather tomorrow?
- **Sentiment analysis:** What is the customer satisfaction level of this review?

These questions may look different, but they share the same idea — we have many examples of input and desired output, and we want to learn how to generate the output for some future, currently unseen input. The name *supervised* comes from the fact that we learn from known answers provided by a “ground truth” data source.

Unsupervised learning

At the other extreme, we have the so-called unsupervised learning, which assumes no supervision and has no known labels assigned to our data. The main objective is to learn some hidden structure of the dataset at hand. One common example of such an approach to learning is the clustering of data. This happens when our algorithm tries to combine data items into a set of clusters, which can reveal relationships in data. For instance, you might want to find similar images or clients with common behavior patterns.

Another unsupervised learning method that is becoming more and more popular is **generative adversarial networks (GANs)**. When we have two competing **neural networks (NNs)**, the first network tries to generate fake data to fool the second network, while the second network tries to discriminate artificially generated data from data sampled from our dataset.

Over time, both networks become more and more skillful in their tasks by capturing subtle specific patterns in the dataset.

Reinforcement learning

RL is the third camp and lies somewhere in between full supervision and a complete lack of predefined labels. On the one hand, it uses many well-established methods of supervised learning, such as **deep neural networks** for function approximation, stochastic gradient descent, and backpropagation, to learn data representation. On the other hand, it usually applies them in a different way.

In the next two sections of the chapter, we will explore specific details of the RL approach, including assumptions and abstractions in its strict mathematical form. For now, to compare RL with supervised and unsupervised learning, we will take a less formal, but more easily understood, path.

Imagine that you have an **agent** that needs to take actions in some **environment**. Both “agent” and “environment” will be defined in detail later in this chapter. A robot mouse in a maze is a good example, but you can also imagine an automatic helicopter trying to perform a roll, or a chess program learning how to beat a grandmaster. Let’s go with the robot mouse for simplicity.

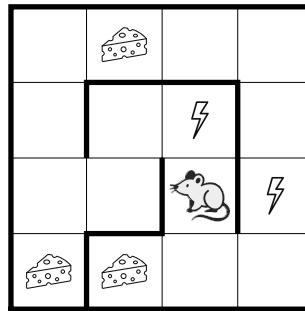


Figure 1.1: The robot mouse maze world

In this case, the *environment* is a maze with food at some points and electricity at others. The robot mouse is the *agent* that can take actions, such as turn left/right and move forward. At each moment, it can observe the full state of the maze to make a decision about the actions to take. The robot mouse tries to find as much food as possible while avoiding getting an electric shock whenever possible. These food and electricity signals stand as the reward that is given to the agent (robot mouse) by the environment as additional feedback about the agent’s actions. The **reward** is a very important concept in RL, and we will talk about it later in the chapter. For now, it is enough for you to know that the final goal of the agent is to maximize its reward as much as possible.

In our particular example, the robot mouse could suffer a slight electric shock as a short-term setback to get to a place with plenty of food in the long term — this would be a better result for the robot mouse than just standing still and gaining nothing.

We don't want to hard-code knowledge about the environment and the best actions to take in every specific situation into the robot mouse — it will take too much effort and may become useless even with a slight maze change. What we want is to have some magic set of methods that will allow our robot mouse to learn on its own how to avoid electricity and gather as much food as possible. RL is exactly this magic toolbox and it behaves differently from supervised and unsupervised learning methods; it doesn't work with predefined labels in the way that supervised learning does. Nobody labels all the images that the robot sees as *good* or *bad*, or gives it the best direction to turn in.

However, we're not completely blind as in an unsupervised learning setup — we have a reward system. The reward can be positive from gathering the food, negative from electric shocks, or neutral when nothing special happens. By observing the reward and relating it to the actions taken, our agent learns how to perform an action better, gather more food, and get fewer electric shocks. Of course, RL generality and flexibility comes with a price. RL is considered to be a much more challenging area than supervised or unsupervised learning. Let's quickly discuss what makes RL tricky.

Complications in RL

The first thing to note is that observations in RL depend on an agent's behavior and, to some extent, it is the result of this behavior. If your agent decides to do inefficient things, then the observations will tell you nothing about what it has done wrong and what should be done to improve the outcome (the agent will just get negative feedback all the time). If the agent is stubborn and keeps making mistakes, then the observations will give the false impression that there is no way to get a larger reward — *life is suffering* — which could be totally wrong.

In ML terms, this can be rephrased as *having non-IID data*. The abbreviation **iid** stands for **independent and identically distributed**, a requirement for most supervised learning methods.

The second thing that complicates our agent's life is that it needs to not only *exploit* the knowledge it has learned, but actively *explore* the environment, because maybe doing things differently will significantly improve the outcome. The problem is that too much exploration may also seriously decrease the reward (not to mention the agent can actually *forget* what it has learned before), so we need to find a balance between these two activities somehow. This exploration/exploitation dilemma is one of the open fundamental questions in RL. People face this choice all the time — should I go to an already known place for dinner or try this fancy new restaurant? How frequently should I change jobs? Should I study a new field or keep working in my area?

There are no universal answers to these questions.

The third complication lies in the fact that reward can be seriously delayed after actions. In chess, for example, one single strong move in the middle of the game can shift the balance. During learning, we need to discover such causalities, which can be tricky to discern during the flow of time and our actions.

However, despite all these obstacles and complications, RL has seen huge improvements in recent years and is becoming more and more active as a field of research and practical application.

Interested in learning more? Let's dive into the details and look at RL formalisms and play rules.

RL formalisms

Every scientific and engineering field has its own assumptions and limitations. Earlier in this chapter, we discussed supervised learning, in which such assumptions are the knowledge of input-output pairs. You have no labels for your data? You need to figure out how to obtain labels or try to use some other theory. This doesn't make supervised learning *good* or *bad*; it just makes it inapplicable to your problem.

There are many historical examples of practical and theoretical breakthroughs that have occurred when somebody tried to challenge rules in a creative way. However, we also must understand our limitations. It's important to know and understand game rules for various methods, as it can save you tons of time in advance. Of course, such formalisms exist for RL, and we will spend the rest of this book analyzing them from various angles.

The following diagram shows two major RL entities – **agent** and **environment** – and their communication channels – **actions**, **reward**, and **observations**:

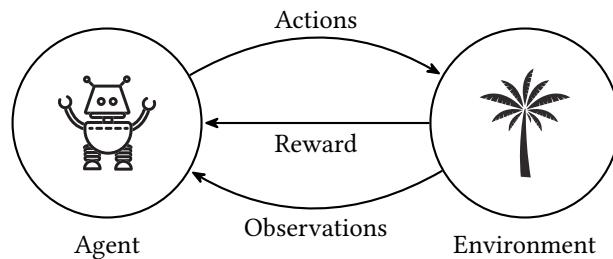


Figure 1.2: RL entities and their communication channels

We will discuss them in detail in the next few sections.

Reward

First, let's return to the notion of reward. In RL, it's just a scalar value we obtain periodically from the environment. As mentioned, reward can be positive or negative, large or small, but it's just a number. The purpose of reward is to tell our agent how well it has behaved. We don't define how frequently the agent receives this reward; it can be every second or once in an agent's lifetime, although it's common practice to receive rewards every fixed timestamp or at every environment interaction, just for convenience. In the case of once-in-a-lifetime reward systems, all rewards except the last one will be zero.

As I stated, the purpose of reward is to give an agent feedback about its success, and it's a central thing in RL. Basically, the term *reinforcement* comes from the fact that reward obtained by an agent should reinforce its behavior in a positive or negative way. Reward is local, meaning that it reflects the benefits and losses achieved by the agent so far. Of course, getting a large reward for some action doesn't mean that, a second later, you won't face dramatic consequences as a result of your previous decisions. It's like robbing a bank — it could look like a good idea until you think about the consequences.

What an agent is trying to achieve is the largest accumulated reward over its sequence of actions. To give you a better understanding of reward, here is a list of some concrete examples with their rewards:

- **Financial trading:** An amount of profit is a reward for a trader buying and selling stocks.
- **Chess:** Reward is obtained at the end of the game as a win, lose, or draw. Of course, it's up to interpretation. For me, for example, achieving a draw in a match against a chess grandmaster would be a huge reward. In practice, we need to specify the exact reward value, but it could be a fairly complicated expression. For instance, in the case of chess, the reward could be proportional to the opponent's strength.
- **Dopamine system in the brain:** There is a part of the brain (limbic system) that produces dopamine every time it needs to send a positive signal to the rest of the brain. High concentrations of dopamine lead to a sense of pleasure, which reinforces activities considered by this system to be *good*. Unfortunately, the limbic system is ancient in terms of the things it considers *good* — food, reproduction, and safety — but that is a totally different story!
- **Computer games:** They usually give obvious feedback to the player, which is either the number of enemies killed or a score gathered. Note in this example that reward is already accumulated, so the RL reward for arcade games should be the derivative of the score, that is, $+1$ every time a new enemy is killed, $-N$ if the player was killed by the enemy, and 0 at all other time steps.

- **Web navigation:** There are problems, with high practical value, that require the automated extraction of information available on the web. Search engines are trying to solve this task in general, but sometimes, to get to the data you're looking for, you need to fill in some forms or navigate through a series of links, or complete CAPTCHAs, which can be difficult for search engines to do. There is an RL-based approach to those tasks in which the reward is the information or the outcome that you need to get.
- **NN architecture search:** RL can be used for NN architecture optimization where the quality of models is crucial and people work hard to gain an extra 1% on target metrics. In this use case, the aim is to get the best performance metric on some dataset by tweaking the number of layers or their parameters, adding extra bypass connections, or making other changes to the NN architecture. The reward in this case is the performance (accuracy or another measure showing how accurate the NN predictions are).
- **Dog training:** If you have ever tried to train a dog, you know that you need to give it something tasty (but not too much) every time it does the thing you've asked. It's also common to reprimand your pet a bit (negative reward) when it doesn't follow your orders, although recent studies have shown that this isn't as effective as a positive reward.
- **School marks:** We all have experience here! School marks are a reward system designed to give pupils feedback about their studying.

As you can see from the preceding examples, the notion of reward is a very general indication of the agent's performance, and it can be found or artificially injected into lots of practical problems around us.

The agent

An agent is somebody or something who/that interacts with the environment by executing certain actions, making observations, and receiving eventual rewards for this. In most practical RL scenarios, the agent is our piece of software that is supposed to solve some problem in a more-or-less efficient way. For our initial set of six examples, the agents will be as follows:

- **Financial trading:** A trading system or a trader making decisions about order execution (buying, selling, or doing nothing).
- **Chess:** A player or a computer program.
- **Dopamine system:** The brain itself, which, according to sensory data, decides whether it was a good experience.
- **Computer games:** The player who enjoys the game or the computer program. (Andrej Karpathy once tweeted that “we were supposed to make AI do all the work and we play games but we do all the work and the AI is playing games!”).

- **Web navigation:** The software that tells the browser which links to click on, where to move the mouse, or which text to enter.
- **NN architecture search:** The software that controls the concrete architecture of the NN being evaluated.
- **Dog training:** You make decisions about the actions (feeding/reprimand), so, the agent is you. But in principle, your dog also could be seen as the agent — the dog is trying to maximize the reward (food and/or attention) by behaving properly. Strictly speaking, here we have a “multi-agent RL” setup, which is briefly discussed in *Chapter 22*.
- **School:** Student/pupil.

The environment

The environment is everything outside of an agent. In the most general sense, it's the rest of the universe, but this goes slightly overboard and exceeds the capacity of even tomorrow's computers, so we usually follow the general sense here.

The agent's communication with the environment is limited to reward (obtained from the environment), actions (executed by the agent and sent to the environment), and observations (some information besides the reward that the agent receives from the environment). We have discussed rewards already, so let's talk about actions and observations next. We will identify the environment for each of our examples when we discuss the observations.

Actions

Actions are things that an agent can do in the environment. Actions can, for example, be piece moves on the board (if it's a board game), or doing homework (in the case of school). They can be as simple as *move pawn one space forward* or as complicated as *build a profitable startup company*.

In RL, we distinguish between two types of actions — discrete or continuous. Discrete actions form the finite set of mutually exclusive things an agent can do, such as move left or right. Continuous actions have some value attached to them, such as a car's *turn the wheel* action having an angle and direction of steering. Different angles could lead to a different scenario a second later, so just *turn the wheel* is definitely not enough.

Giving concrete examples, let's look at the actions in our six scenarios:

- **Financial trading:** Actions are decisions to buy or sell stock. “Do nothing and wait” also is an action.
- **Chess:** Actions are valid piece moves according to the current board's position.
- **Dopamine system:** Actions are the things that you are doing.
- **Computer games:** Actions are pushing buttons. They could be also continuous, such as turning the steering wheel in an auto simulator.

- **Web navigation:** Actions could be mouse clicks, scrolling, and text typing.
- **NN architecture search:** Actions are changes in NN architecture, which could be discrete (count of layers in the network) or continuous (probability in the dropout layer).
- **Dog training:** Actions are everything you can do with your dog — giving a piece of tasty food, petting, even saying “good dog!” in a kind voice.
- **School:** Actions are marks and lots of more informal signals, like praising the successes or giving extra homework.

Observations

Observations of the environment form the second information channel for an agent, with the first being the *reward*. You may be wondering why we need a separate data source. The answer is convenience. Observations are pieces of information that the environment provides the agent with that indicate what’s going on around the agent.

Observations may be relevant to the upcoming reward (such as seeing a bank notification about being paid) or may not be. Observations can even include reward information in some vague or obfuscated form, such as score numbers on a computer game’s screen. Score numbers are just pixels, but potentially, we could convert them into reward values; it’s not a very complex task for a modern computer vision techniques.

On the other hand, reward shouldn’t be seen as a secondary or unimportant thing — reward is the main force that drives the agent’s learning process. If a reward is wrong, noisy, or just slightly off course from the primary objective, then there is a chance that training will go in the wrong direction.

It’s also important to distinguish between an environment’s state and observations. The state of an environment most of the time is *internal* to the environment and potentially includes every atom in the universe, which makes it impossible to measure everything about the environment. Even if we limit the environment’s state to be small enough, most of the time, it will be either not possible to get full information about it or our measurements will contain noise. This is completely fine, though, and RL was created to support such cases natively. To illustrate the difference, let’s return to our set of examples:

- **Financial trading:** Here, the environment is the whole financial market and everything that influences it. This is a huge list of things, such as the latest news, economic and political conditions, weather, food supplies, and Twitter/X trends. Even your decision to stay home today can potentially indirectly influence the world’s financial system (if you believe in the “butterfly effect”). However, our observations are limited to stock prices, news, and so on. We don’t have access to most of the environment’s state, which makes financial forecasting such a nontrivial thing.

- **Chess:** The environment here is your board *plus* your opponent, which includes their chess skills, mood, brain state, chosen tactics, and so on. Observations are what you see (your current chess position), but, at some levels of play, knowledge of psychology and the ability to read an opponent's mood could increase your chances.
- **Dopamine system:** The environment here is your brain *plus* your nervous system and your organs' states *plus* the whole world you can perceive. Observations are the inner brain state and signals coming from your senses.
- **Computer game:** Here, the environment is your computer's state, including all memory and disk data. For networked games, you need to include other computers *plus* all Internet infrastructure between them and your machine. Observations are a screen's pixels and sound only. These pixels are not a tiny amount of information (it has been estimated that the total number of possible moderate-size images (1024×768) is significantly larger than the number of atoms in our galaxy), but the whole environment state is definitely larger.
- **Web navigation:** The environment here is the Internet, including all the network infrastructure between the computer on which our agent works and the web server, which is a really huge system that includes millions and millions of different components. The observation is normally the web page that is loaded in the browser.
- **NN architecture search:** In this example, the environment is fairly simple and includes the NN toolkit that performs the particular NN evaluation and the dataset that is used to obtain the performance metric. In comparison to the Internet, this looks like a tiny toy environment. Observations might be different and include some information about testing, such as loss convergence dynamics or other metrics obtained from the evaluation step.
- **Dog training:** Here, the environment is your dog (including its hardly observable inner reactions, mood, and life experiences) and everything around it, including other dogs and even a cat hiding in a bush. Observations are signals from your senses and memory.
- **School:** The environment here is the school itself, the education system of the country, society, and the cultural legacy. Observations are the same as for the dog training example – the student's senses and memory.

This is our “mise en scène” and we will play around with it in the rest of this book. You will have already noticed that the RL model is extremely flexible and general, and it can be applied to a variety of scenarios. Let's now look at how RL is related to other disciplines, before diving into the details of the RL model.

There are many other areas that contribute or relate to RL. The most significant are shown in the following diagram, which includes six large domains heavily overlapping each other on the methods and specific topics related to decision-making (shown inside the inner circle).

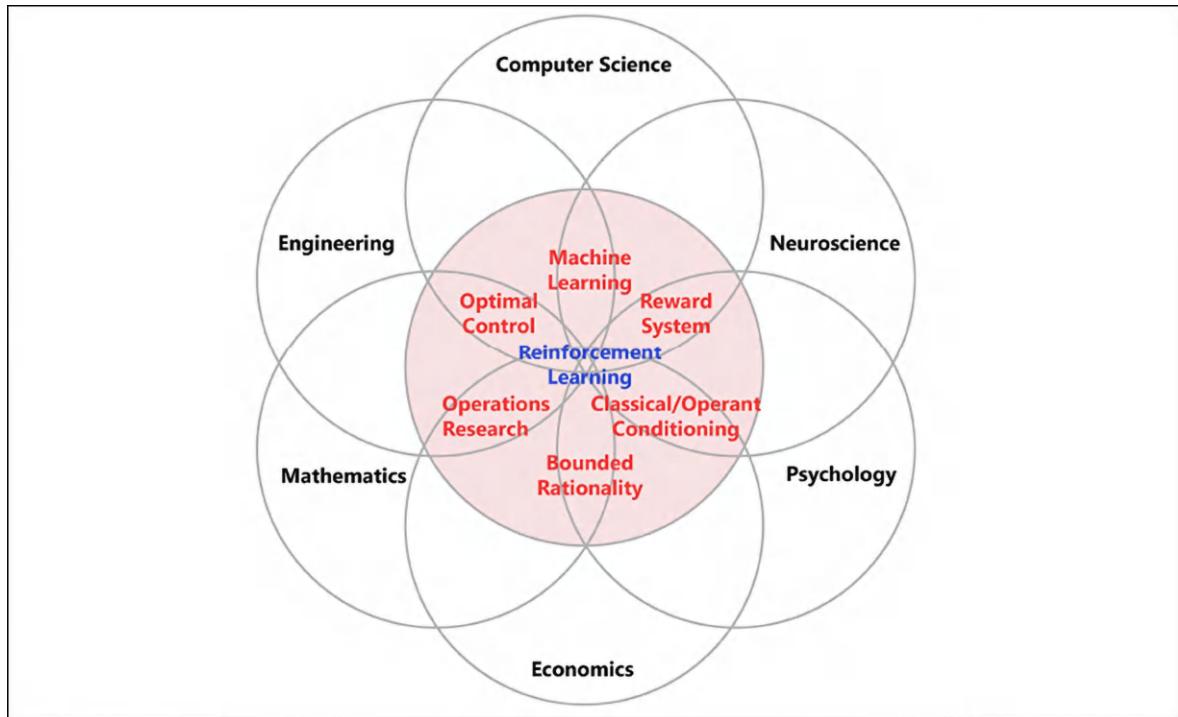


Figure 1.3: Various domains in RL

At the intersection of all those related, but still different, scientific areas sits RL, which is so general and flexible that it can take the best available information from these varying domains:

- **ML:** RL, being a subfield of ML, borrows lots of its machinery, tricks, and techniques from ML. Basically, the goal of RL is to learn how an agent should behave when it is given imperfect observational data.
- **Engineering (especially optimal control):** This helps with taking a sequence of optimal actions to get the best result.
- **Neuroscience:** We used the dopamine system as our example, and it has been shown that the human brain acts similarly to the RL model.
- **Psychology:** This studies behavior in various conditions, such as how people react and adapt, which is close to the RL topic.
- **Economics:** One of the important topics in economics is how to maximize reward in terms of imperfect knowledge and the changing conditions of the real world.
- **Mathematics:** This works with idealized systems and also devotes significant attention to finding and reaching the optimal conditions in the field of operations research.

In the next part of the chapter, you will become familiar with the theoretical foundations of RL, which will make it possible to start moving toward the methods used to solve the RL problem. The upcoming section is important for understanding the rest of the book.

The theoretical foundations of RL

In this section, I will introduce you to the mathematical representation and notation of the formalisms (reward, agent, actions, observations, and environment) that we just discussed. Then, using this as a knowledge base, we will explore the second-order notions of the RL language, including state, episode, history, value, and gain, which will be used repeatedly to describe different methods later in the book.

Markov decision processes

Before that, we will cover **Markov decision processes (MDPs)**, which will be described like a Russian matryoshka doll: we will start from the simplest case of a **Markov process (MP)**, then extend that with rewards, which will turn it into a **Markov reward process (MRP)**. Then, we will put this idea into an extra envelope by adding actions, which will lead us to an MDP.

MPs and MDPs are widely used in computer science and other engineering fields. So, reading this chapter will be useful for you not only for RL contexts but also for a much wider range of topics. If you're already familiar with MDPs, then you can quickly skim this chapter, paying attention only to the terminology definitions, as we will use them later on.

The Markov process

Let's start with the simplest concept in the Markov family: the MP, which is also known as the **Markov chain**. Imagine that you have some system in front of you that you can only observe. What you observe is called states, and the system can switch between states according to some laws of dynamics (most of the time unknown to you). Again, you cannot influence the system, but can only watch the states changing. All possible states for a system form a set called the **state space**. For MPs, we require this set of states to be finite (but it can be extremely large to compensate for this limitation). Your observations form a sequence of states or a **chain** (that's why MPs are also called Markov chains).

For example, looking at the simplest model of the weather in some city, we can observe the current day as sunny or rainy, which is our state space. A sequence of observations over time forms a chain of states, such as `[sunny, sunny, rainy, sunny, ...]`, and this is called **history**. To call such a system an MP, it needs to fulfill the **Markov property**, which means that the future system dynamics from any state have to depend on this state only. The main point of the Markov property is to make every observable state self-contained to describe the future of the system.

In other words, the Markov property requires the states of the system to be distinguishable from each other and unique. In this case, only one state is required to model the future dynamics of the system and not the whole history or, say, the last N states.

In the case of our toy weather example, the Markov property limits our model to represent only the cases when a sunny day can be followed by a rainy one with the same probability, regardless of the number of sunny days we've seen in the past. It's not a very realistic model as, from common sense, we know that the chance of rain tomorrow depends not only on the current conditions but on a large number of other factors, such as the season, our latitude, and the presence of mountains and sea nearby. It was recently proven that even solar activity has a major influence on the weather. So, our example is really naïve, but it's important to understand the limitations and make conscious decisions about them.

Of course, if we want to make our model more complex, we can always do this by extending our state space, which will allow us to capture more dependencies in the model at the cost of a larger state space. For example, if you want to capture separately the probability of rainy days during summer and winter, then you can include the season in your state.

In this case, your state space will be [*sunny+summer*, *sunny+winter*, *rainy+summer*, *rainy+winter*] and so on.

As your system model complies with the Markov property, you can capture transition probabilities with a **transition matrix**, which is a square matrix of the size $N \times N$, where N is the number of states in our model. Every cell in a row, i , and a column, j , in the matrix contains the probability of the system to transition from state i to state j .

For example, in our sunny/rainy example, the transition matrix could be as follows:

	Sunny	Rainy
Sunny	0.8	0.2
Rainy	0.1	0.9

In this case, if we have a sunny day, then there is an 80% chance that the next day will be sunny and a 20% chance that the next day will be rainy. If we observe a rainy day, then there is a 10% probability that the weather will become better and a 90% probability of the next day being rainy.

So, that's it. The formal definition of an MP is as follows:

- A set of states (S) that a system can be in
- A transition matrix (T), with transition probabilities, which defines the system dynamics

A useful visual representation of an MP is a graph with nodes corresponding to system states and edges, labeled with probabilities representing a possible transition from state to state. If the probability of a transition is 0, we don't draw an edge (there is no way to go from one state to another). This kind of representation is also widely used in finite state machine representation, which is studied in automata theory. For our sunny/rainy weather model, the graph is as shown here:

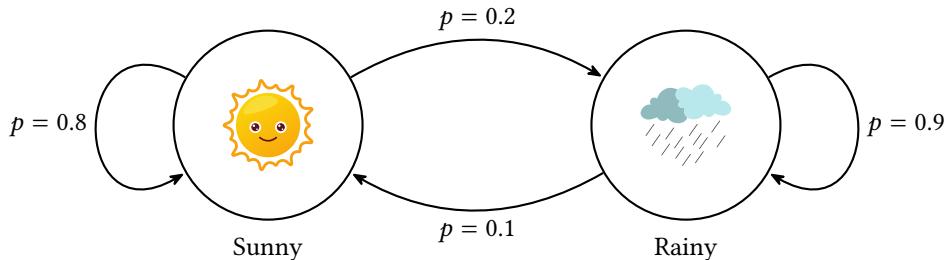


Figure 1.4: The sunny/rainy weather model

Again, we're talking about observation only. There is no way for us to influence the weather, so we just observe it and record our observations.

To give you a more complicated example, let's consider another model called *Office Worker* (Dilbert, the main character in Scott Adams' famous cartoons, is a good example). His state space in our example has the following states:

- **Home:** He's not at the office
- **Computer:** He's working on his computer at the office
- **Coffee:** He's drinking coffee at the office
- **Chat:** He's discussing something with colleagues at the office

The state transition graph is shown in the following figure:

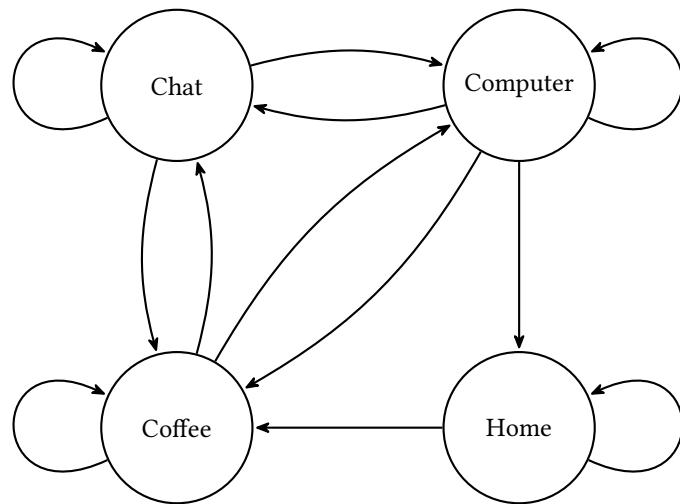


Figure 1.5: The state transition graph for our office worker

We assume that our office worker's weekday usually starts from the **Home** state and that he starts his day with **Coffee** without exception (no **Home** → **Computer** edge and no **Home** → **Chat** edge). The preceding diagram also shows that workdays always end (that is, going to the **Home** state) from the **Computer** state.

The transition matrix for the diagram above is as follows:

	Home	Coffee	Chat	Computer
Home	60%	40%	0%	0%
Coffee	0%	10%	70%	20%
Chat	0%	20%	50%	30%
Computer	20%	20%	10%	50%

The transition probabilities could be placed directly on the state transition graph, as shown in *Figure 1.6*.

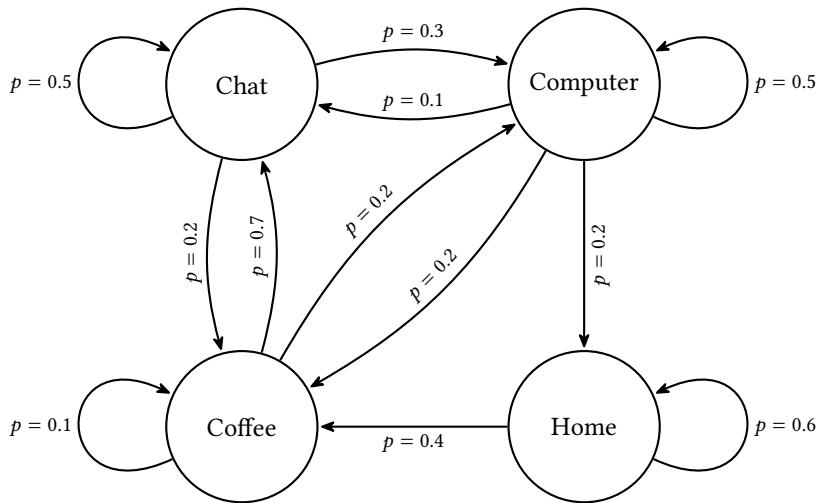


Figure 1.6: The state transition graph with transition probabilities

In practice, we rarely have the luxury of knowing the exact transition matrix. A much more real-world situation is when we only have observations of our system's states, which are also called **episodes**:

- **Home → Coffee → Coffee → Chat → Chat → Coffee → Computer → Computer → Home**
- **Computer → Computer → Chat → Chat → Coffee → Computer → Computer → Computer**
- **Home → Home → Coffee → Chat → Computer → Coffee → Coffee**

It's not complicated to estimate the transition matrix from our observations — we just count all the transitions from every state and normalize them to a sum of 1. The more observation data we have, the closer our estimation will be to the true underlying model.

It's also worth noting that the Markov property implies **stationarity** (which means, the underlying transition distribution for any state does not change over time). **Non-stationarity** means that there is some hidden factor that influences our system dynamics, and this factor is not included in observations. However, this contradicts the Markov property, which requires the underlying probability distribution to be the same for the same state regardless of the transition history.

It's important to understand the difference between the actual transitions observed in an episode and the underlying distribution given in the transition matrix. Concrete episodes that we observe are randomly sampled from the distribution of the model, so they can differ from episode to episode. However, the probability of the concrete transition to be sampled remains the same. If this is not the case, Markov chain formalism becomes non-applicable.

Now we can go further and extend the MP model to make it closer to our RL problems. Let's add rewards to the picture!

Markov reward processes

To introduce reward, we need to extend our MP model a bit. First, we need to add value to our transition from state to state. We already have probability, but probability is being used to capture the dynamics of the system, so now we have an extra scalar number without extra burden.

Rewards can be represented in various forms. The most general way is to have another square matrix, similar to the transition matrix, with a reward given for transitioning from state i to state j , which reside in row i and column j .

As mentioned, rewards can be positive or negative, large or small. In some cases, this representation is redundant and can be simplified. For example, if a reward is given for reaching the state regardless of the origin state, we can keep only (**state, reward**) pairs, which is a more compact representation. However, this is applicable only if the reward value depends solely on the target state, which is not always the case.

The second thing we're adding to the model is the discount factor γ (Greek letter "gamma"), which is a single number from 0 to 1 (inclusive). The meaning of this will be explained after the extra characteristics of our MRP have been defined.

As you will remember, we observe a chain of state transitions in an MP. This is still the case for a MRP, but for every transition, we have our extra quantity — reward. So now, all our observations have a reward value attached to every transition of the system.

For every episode, we define **return** at the time t as G_t :

$$G_t = R_{t+1} + \gamma R_{t+2} + \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

The γ in the preceding formula is very important in RL, and we will meet it a lot in the subsequent chapters. For now, think about it as a measure of how far into the future we look to estimate the future return. The closer its value is to 1, the more steps ahead of us we will take into account.

Now let's try to understand what the formula for **return** means. For every time point, we calculate return as a sum of subsequent rewards, but more distant rewards are multiplied by the discount factor raised to the power of the number of steps we are away from the starting point at t . The discount factor stands for the foresightedness of the agent. If $\gamma = 1$, then return, G_t , just equals a sum of all subsequent rewards and corresponds to the agent that has perfect visibility of any subsequent rewards. If $\gamma = 0$, G_t will be just immediate reward without any subsequent state and will correspond to absolute short-sightedness.

These extreme values are useful only in corner cases, and most of the time, γ is set to something in between, such as 0.9 or 0.99. In this case, we will look into future rewards, but not too far. The value of $\gamma = 1$ might be applicable in situations of short finite episodes.

This return quantity is not very useful in practice, as it was defined for every specific chain we observed from our MRP, so it can vary widely, even for the same state. However, if we go to the extreme and calculate the mathematical expectation of return for any state (by averaging a large number of chains), we will get a much more practical quantity, which is called the **value of the state**:

$$V(s) = \mathbb{E}[G|S_t = s]$$

This interpretation is simple—for every state, s , the value, $V(s)$, is the average (or expected) return we get by following the Markov reward process.

To represent this theoretical knowledge practically, let's extend our office worker (Dilbert) process with a reward and turn it into a **Dilbert reward process (DRP)**. Our reward values will be as follows:

- **Home → Home:** 1 (as it's good to be home)
- **Home → Coffee:** 1
- **Computer → Computer:** 5 (working hard is a good thing)
- **Computer → Chat:** -3 (it's not good to be distracted)
- **Chat → Computer:** 2
- **Computer → Coffee:** 1
- **Coffee → Computer:** 3
- **Coffee → Cofee:** 1
- **Coffee → Chat:** 2
- **Chat → Coffee:** 1
- **Chat → Chat:** -1 (long conversations become boring)

A diagram of this is shown in *Figure 1.7*.

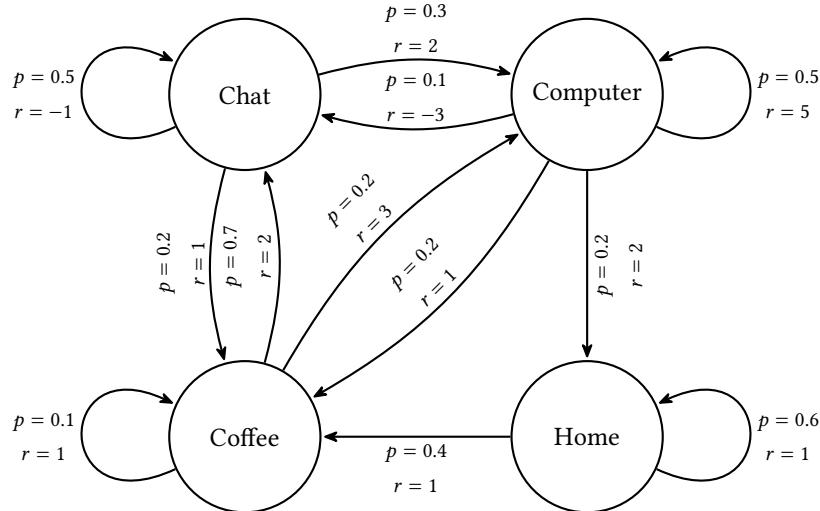


Figure 1.7: The state transition graph with transition probabilities and rewards

Let's return to our γ parameter and think about the values of states with different values of γ . We will start with a simple case: $\gamma = 0$. How do you calculate the values of states here? To answer this question, let's fix our state to **Chat**. What could the subsequent transition be? The answer is that it depends on chance. According to our transition matrix for the Dilbert process, there is a 50% probability that the next state will be **Chat** again, 20% that it will be **Coffee**, and 30% that it will be **Computer**. When $\gamma = 0$, our return is equal only to a value of the next immediate state. So, if we want to calculate the value of the **Chat** state in the preceding diagram, then we need to sum all transition values and multiply that by their probabilities:

$$\begin{aligned}
 V(\text{chat}) &= -1 \cdot 0.5 + 2 \cdot 0.3 + 1 \cdot 0.2 = 0.3 \\
 V(\text{coffee}) &= 2 \cdot 0.7 + 1 \cdot 0.1 + 3 \cdot 0.2 = 2.1 \\
 V(\text{home}) &= 1 \cdot 0.6 + 1 \cdot 0.4 = 1.0 \\
 V(\text{computer}) &= 5 \cdot 0.5 + (-3) \cdot 0.1 + 1 \cdot 0.2 + 2 \cdot 0.2 = 2.8
 \end{aligned}$$

So, **Computer** is the most valuable state to be in (if we care only about immediate reward), which is not surprising as **Computer** → **Computer** is frequent, has a large reward, and the ratio of interruptions is not too high.

Now a trickier question – what's the value when $\gamma = 1$? Think about this carefully.

The answer is that the value is infinite for all states. Our diagram doesn't contain **sink states** (states without outgoing transitions), and when our discount equals 1, we care about a potentially infinite number of transitions in the future. As you've seen in the case of $\gamma = 0$, all our values are positive in the short term, so the sum of the infinite number of positive values will give us an infinite value, regardless of the starting state.

This infinite result shows us one of the reasons to introduce γ into a MRP instead of just summing all future rewards. In most cases, the process can have an infinite (or large) amount of transitions. As it is not very practical to deal with infinite values, we would like to limit the horizon we calculate values for. Gamma with a value less than 1 provides such a limitation, and we will discuss this later in this book. On the other hand, if you're dealing with finite-horizon environments (for example, the tic-tac-toe game, which is limited by at most nine steps), then it will be fine to use $\gamma = 1$.



As another example, there is an important class of environments with only one step called **the multi-armed bandit MDP**. This means that on every step, you need to make a selection of one alternative action, which provides you with some reward and the episode ends.

You can learn more about bandit methods in the book Bandit Algorithms by Tor Lattimore and Csaba Szepesvari (<https://tor-lattimore.com/downloads/book/book.pdf>).

As I already mentioned about the MRP, γ is usually set to a value between 0 and 1. However, with such values, it becomes almost impossible to calculate them accurately by hand, even for MRPs as small as our Dilbert example, because it will require summing hundreds of values. Computers are good at tedious tasks such as this, and there are several simple methods that can quickly calculate values for MRPs for given transition and reward matrices. We will see and even implement one such method in *Chapter 5*, when we will start looking at Q-learning methods.

For now, let's put another layer of complexity around our Markov reward processes and introduce the final missing piece: **actions**.

Adding actions to MDP

You may already have ideas about how to extend our MDP to include actions. Firstly, we must add a set of actions (A), which has to be finite. This is our agent's **action space**. Secondly, we need to condition our transition matrix with actions, which basically means that our matrix needs an extra action dimension, which turns it into a cuboid of shape $|S| \times |S| \times |A|$, where S is our state space and A is an action space.

If you remember, in the case of MPs and MRPs, the transition matrix had a square form, with the source state in rows and target state in columns. So, every row, i , contained a list of probabilities to jump to every state, as shown in *Figure 1.8*.

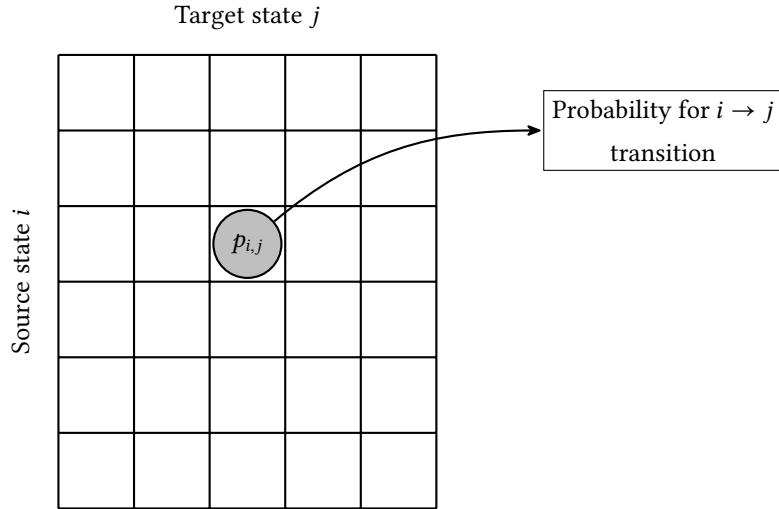


Figure 1.8: The transition matrix for the Markov process

In case of an MDP, the agent no longer passively observes state transitions, but can actively choose an action to take at every state transition. So, for every source state, we don't have a list of numbers, but we have a matrix, where the **depth** dimension contains actions that the agent can take, and the other dimension is what the target state system will jump to after actions are performed by the agent. The following diagram shows our new transition table, which became a cuboid with the source state as the *height* dimension (indexed by i), the target state as the *width* (j), and the action the agent can take as the *depth* (k) of the transition table:

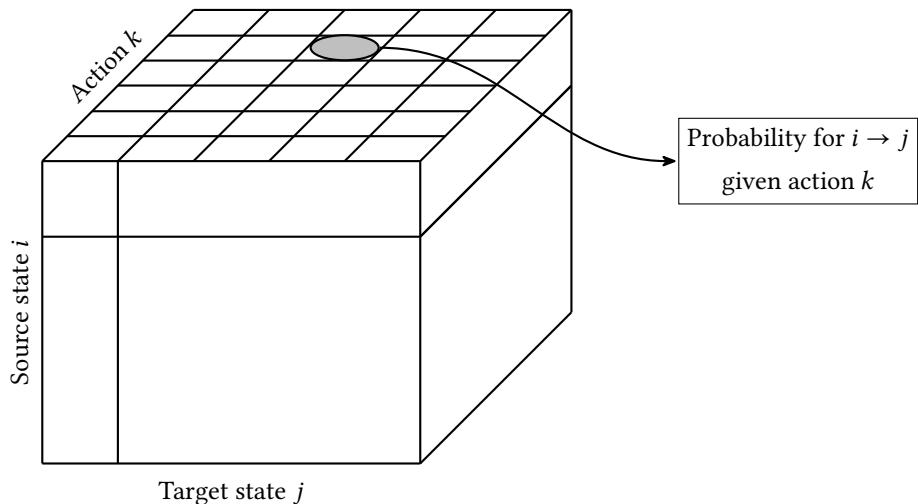


Figure 1.9: The transition probabilities for the MDP

So, in general, by choosing an action, the agent can affect the probabilities of the target states, which is a useful ability.

To give you an idea of why we need so many complications, let's imagine a small robot that lives in a 3×3 grid and can execute the actions *turn left*, *turn right*, and *go forward*. The state of the world is the robot's position plus orientation (up, down, left, and right), which gives us $3 \times 3 \times 4 = 36$ states (the robot can be at any location in any orientation).

Also, imagine that the robot has imperfect motors (which is frequently the case in the real world), and when it executes *turn left* or *turn right*, there is a 90% chance that the desired turn happens, but sometimes, with a 10% probability, the wheel slips and the robot's position stays the same. The same happens with *go forward* – in 90% of cases it works, but for therest (10%) the robot stays at the same position.

In *Figure 1.10*, a small part of a transition diagram is shown, displaying the possible transitions from the state $(1, 1)$, up, when the robot is in the center of the grid and facing up. If the robot tries to move forward, there is a 90% chance that it will end up in the state $(0, 1)$, up, but there is a 10% probability that the wheels will slip and the target position will remain $(1, 1)$, up.

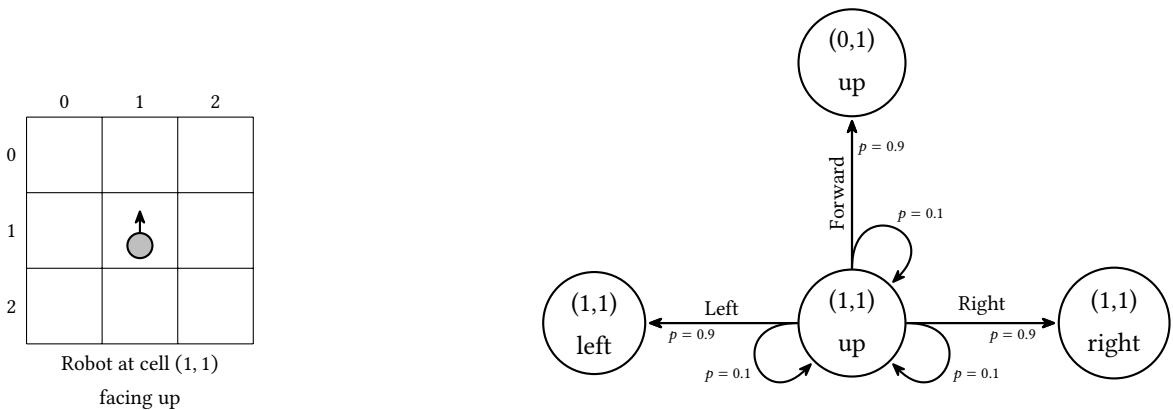


Figure 1.10: A grid world environment

To properly capture all these details about the environment and possible reactions to the agent's actions, the general MDP has a 3D transition matrix with the dimensions source state, action, and target state.

Finally, to turn our MRP into an MDP, we need to add actions to our reward matrix in the same way that we did with the transition matrix. Our reward matrix will depend not only on the state but also on the action. In other words, the reward the agent obtains will now depend not only on the state it ends up in but also on the action that leads to this state.

Now, with a formally defined MDP, we're finally ready to cover the most important thing for MDPs and RL: **policy**.

Policy

The simple definition of policy is that it is some set of rules that defines the agent's behavior. Even for fairly simple environments, we can have a variety of policies. For example, in the preceding example with the robot in the grid world, the agent can have different policies, which will lead to different sets of visited states. For example, the robot can perform the following actions:

- Blindly move forward regardless of anything
- Try to go around obstacles by checking whether that previous *forward* action failed
- Funnily spin around by always turning right to entertain its creator
- Choose an action randomly regardless of position and orientation, modeling a drunk robot in the grid world scenario

You may remember that the main objective of the agent in RL is to gather as much return as possible. So, again, different policies can give us different amounts of return, which makes it important to find a good policy. This is why the notion of policy is important.

Formally, policy is defined as the probability distribution over actions for every possible state:

$$\pi(a|s) = P[A_t = a | S_t = s]$$

This is defined as probability and not as a concrete action to introduce randomness into an agent's behavior. In section 3 of the book, we will talk about why this is important and useful. Deterministic policy is a special case of probabilistics with the needed action having 1 as its probability.

Another useful notion is that if our policy is fixed and not changing during training (i.e., when the policy always returns the same actions for the same states), then our MDP becomes a MRP, as we can reduce the transition and reward matrices with a policy's probabilities and get rid of the action dimensions.

Congratulations on getting to this stage! This chapter was challenging, but it was important for understanding subsequent practical material. After two more introductory chapters about OpenAI Gym and deep learning, we will finally start tackling this question — how do we teach agents to solve practical tasks?

Summary

In this chapter, you started your journey into the RL world by learning what makes RL special and how it relates to the supervised and unsupervised learning paradigms. We then learned about the basic RL formalisms and how they interact with each other, after which we covered MPs, MRPs, and MDPs. This knowledge will be the foundation for the material that we will cover in the rest of the book.

In the next chapter, we will move away from the formal theory to the practice of RL. We will cover the setup required and libraries, and then you will write your first agent.

Join our community on Discord

Read this book alongside other users, Deep Learning experts, and the author himself.

Ask questions, provide solutions to other readers, chat with the author via Ask Me Anything sessions, and much more.

Scan the QR code or visit the link to join the community.

<https://packt.link/rl>



2

OpenAI Gym API and Gymnasium

After talking so much about the theoretical concepts of **reinforcement learning (RL)** in *Chapter 1*, let's start doing something practical. In this chapter, you will learn the basics of Gymnasium, a library used to provide a uniform API for an RL agent and lots of RL environments. Originally, this API was implemented in the OpenAI Gym library, but it is no longer maintained. In this book, we'll use Gymnasium—a fork of OpenAI Gym implementing the same API. In any case, having a uniform API for environments removes the need to write boilerplate code and allows you to implement an agent in a general way without worrying about environment details.

You will also write your first randomly behaving agent and become more familiar with the basic concepts of RL that we have covered so far. By the end of the chapter, you will have an understanding of:

- The high-level requirements that need to be implemented to plug the agent into the RL framework
- A basic, pure-Python implementation of the random RL agent
- The OpenAI Gym API and its implementation – the Gymnasium library

The anatomy of the agent

As you learned in the previous chapter, there are several fundamental concepts in RL:

- **The agent:** A thing, or person, that takes an active role. In practice, the agent is some piece of code that implements some policy. Basically, this policy decides what action is needed at every time step, given our observations.
- **The environment:** Everything that is external to the agent and has the responsibility of providing observations and giving rewards. The environment changes its state based on the agent's actions.

Let's explore how both can be implemented in Python for a simple situation. We will define an environment that will give the agent random rewards for a limited number of steps, regardless of the agent's actions. This scenario is not very useful in the real world, but it will allow us to focus on specific methods in both the environment and agent classes.



Please note that the code snippets shown in this book are not full examples. You can find the full examples on the GitHub page: <https://github.com/PacktPublishing/Deep-Reinforcement-Learning-Hands-On-Third-Edition> and run them.

Let's start with the environment:

```
class Environment:  
    def __init__(self):  
        self.steps_left = 10
```

In the preceding code, we allowed the environment to initialize its internal state. In our case, the state is just a counter that limits the number of time steps that the agent is allowed to take to interact with the environment.

The `get_observation()` method is supposed to return the current environment's observation to the agent. It is usually implemented as some function of the internal state of the environment:

```
def get_observation(self) -> List[float]:  
    return [0.0, 0.0, 0.0]
```

If you're curious about what is meant by `-> List[float]`, that's an example of Python type annotations, which were introduced in Python 3.5. You can find out more in the documentation at <https://docs.python.org/3/library/typing.html>. In our example, the observation vector is always zero, as the environment basically has no internal state.

The `get_actions()` method allows the agent to query the set of actions it can execute:

```
def get_actions(self) -> List[int]:  
    return [0, 1]
```

Normally, the set of actions does not change over time, but some actions can become impossible in different states (for example, not every move is possible in any position of the tic-tac-toe game). In our simplistic example, there are only two actions that the agent can carry out, which are encoded with the integers 0 and 1.

The following method signals the end of the episode to the agent:

```
def is_done(self) -> bool:  
    return self.steps_left == 0
```

As you saw in *Chapter 1*, the series of environment-agent interactions is divided into a sequence of steps called episodes. Episodes can be finite, like in a game of chess, or infinite, like the Voyager 2 mission (a famous space probe that was launched over 46 years ago and has traveled beyond our solar system). To cover both scenarios, the environment provides us with a way to detect when an episode is over and there is no way to communicate with it anymore.

The `action()` method is the central piece in the environment's functionality:

```
def action(self, action: int) -> float:  
    if self.is_done():  
        raise Exception("Game is over")  
    self.steps_left -= 1  
    return random.random()
```

It does two things – handles an agent's action and returns the reward for this action. In our example, the reward is random and its action is discarded. Additionally, we update the count of steps and don't continue the episodes that are over.

Now, when looking at the agent's part, it is much simpler and includes only two methods: the constructor and the method that performs one step in the environment:

```
class Agent:  
    def __init__(self):  
        self.total_reward = 0.0
```

In the constructor, we initialize the counter that will keep the total reward accumulated by the agent during the episode.

The `step()` function accepts the environment instance as an argument:

```
def step(self, env: Environment):
    current_obs = env.get_observation()
    actions = env.get_actions()
    reward = env.action(random.choice(actions))
    self.total_reward += reward
```

This function allows the agent to perform the following actions:

- Observe the environment
- Make a decision about the action to take based on the observations
- Submit the action to the environment
- Get the reward for the current step

For our example, the agent is dull and ignores the observations obtained during the decision-making process about which action to take. Instead, every action is selected randomly. The final piece is the glue code, which creates both classes and runs one episode:

```
if __name__ == "__main__":
    env = Environment()
    agent = Agent()
    while not env.is_done():
        agent.step(env)
    print("Total reward got: %.4f" % agent.total_reward)
```

You can find the full code in this book's GitHub repository at <https://github.com/PacktPublishing/Deep-Reinforcement-Learning-Hands-On-Third-Edition> in the `Chapter02/01_agent_anatomy.py` file. It has no external dependencies and should work with any relatively modern Python version. By running it several times, you'll get different amounts of reward gathered by the agent. The following is an output I got on my machine:

```
Chapter02$ python 01_agent_anatomy.py
Total reward got: 5.8832
```

The simplicity of the preceding code illustrates the important basic concepts of the RL model. The environment could be an extremely complicated physics model, and an agent could easily be a large **neural network (NN)** that implements the latest RL algorithm, but the basic pattern will stay the same – at every step, the agent will take some observations from the environment, do its calculations, and select the action to take. The result of this action will be a reward and a new observation.

You may ask, if the pattern is the same, why do we need to write it from scratch? What if it is already implemented by somebody and could be used as a library? Of course, such frameworks exist, but before we spend some time discussing them, let's prepare your development environment.

Hardware and software requirements

The examples in this book were implemented and tested using Python version 3.11. I assume that you're already familiar with the language and common concepts such as virtual environments, so I won't cover in detail how to install packages and how to do this in an isolated way. The examples will use the previously mentioned Python type annotations, which will allow us to provide type signatures for functions and class methods.

Nowadays, there are lots of ML and RL libraries available, but in this book, I tried to keep the list of dependencies to a minimum, giving a preference to our own implementation of methods over the blind import of third-party libraries.

The external libraries that we will use in this book are open source software, and they include the following:

- **NumPy**: This is a library for scientific computing and implementing matrix operations and common functions.
- **OpenCV Python bindings**: This is a computer vision library and provides many functions for image processing.
- **Gymnasium** from the Farama Foundation (<https://farama.org>): This is a maintained fork of the OpenAI Gym library (<https://github.com/openai/gym>) and an RL framework that has various environments that can be communicated with in a unified way.
- **PyTorch**: This is a flexible and expressive **deep learning (DL)** library. A short crash course on it will be given in *Chapter 3*.
- **PyTorch Ignite**: This is a set of high-level tools on top of PyTorch used to reduce boilerplate code. It will be covered briefly in *Chapter 3*. The full documentation is available here: <https://pytorch-ignite.ai/>.

- **PTAN:** (<https://github.com/Shmuma/ptan>): This is an open source extension to the OpenAI Gym API that I created to support modern deep RL methods and building blocks. All classes used will be described in detail together with the source code.

Other libraries will be used for specific chapters; for example, we will use Microsoft TextWorld to play text-based games, PyBullet and MuJoCo for robotic simulations, Selenium for browser-based automation problems, and so on. Those specialized chapters will include installation instructions for those libraries.

A significant portion of this book (Parts 2, 3, and 4) is focused on the modern deep RL methods that have been developed over the past few years. The word “deep” in this context means that DL is heavily used. You may be aware that DL methods are computationally hungry. One modern **graphics processing unit (GPU)** can be 10 to 100 times faster than even the fastest multiple **central processing unit (CPU)** systems. In practice, this means that the same code that takes one hour to train on a system with a GPU could take from half a day to one week even on the fastest CPU system. It doesn’t mean that you can’t try the examples from this book without having access to a GPU, but it will take longer. To experiment with the code on your own (the most useful way to learn anything), it is better to get access to a machine with a GPU. This can be done in various ways:

- Buying a modern GPU suitable for CUDA and supported by the PyTorch framework.
- Using cloud instances. Both Amazon Web Services and Google Cloud Platform can provide you with GPU-powered instances.
- Google Colab offers free GPU access to its Jupyter notebooks.

The instructions on how to set up the system are beyond the scope of this book, but there are plenty of manuals available on the Internet. In terms of an **operating system (OS)**, you should use Linux or macOS. Windows is supported by PyTorch and Gymnasium, but the examples in the book were not fully tested on the Windows OS.

To give you the exact versions of the external dependencies that we will use throughout the book, here is a `requirements.txt` file (please note that it was tested on Python 3.11; different versions might require you to tweak the dependencies or not work at all):

```
[text]
gymnasium[atari]==0.29.1
gymnasium[classic-control]==0.29.1
gymnasium[accept-rom-license]==0.29.1
moviepy==1.0.3
numpy<2
opencv-python==4.10.0.84
torch==2.5.0
```

```
torchvision==0.20.0
pytorch-ignite==0.5.1
tensorboard==2.18.0
mypy==1.8.0
ptan==0.8.1
stable-baselines3==2.3.2
torchrl==0.6.0
ray[tune]==2.37.0
pytest
```

All the examples in the book were written and tested with PyTorch 2.5.0, which can be installed by following the instructions on the <https://pytorch.org> website (normally, that's just the `conda install pytorch torchvision -c pytorch` or even just `pip install torch` command, depending on your OS).

Now, let's go into the details of the OpenAI Gym API, which provides us with tons of environments, from trivial to challenging ones.

The OpenAI Gym API and Gymnasium

The Python library called Gym was developed by OpenAI (www.openai.com). The first version was released in 2017 and since then, lots of environments were developed or adopted to this original API, which became a de facto standard for RL.

In 2021, the team that developed OpenAI Gym moved the development to Gymnasium (github.com/Farama-Foundation/Gymnasium) – the fork of the original Gym library. Gymnasium provides the same API and is supposed to be a “drop-in replacement” for Gym (you can write `import gymnasium as gym` and most likely your code will work).



Examples in this book are using Gymnasium, but in the text, I'll use “Gym” for brevity. In rare cases when the difference does matter, I'll use “Gymnasium.”

The main goal of Gym is to provide a rich collection of environments for RL experiments using a unified interface. So, it is not surprising that the central class in the library is an environment, which is called `Env`. Instances of this class expose several methods and fields that provide the required information about its capabilities. At a high level, every environment provides these pieces of information and functionality:

- A set of actions that is allowed to be executed in the environment. Gym supports both discrete and continuous actions, as well as their combination.
- The shape and boundaries of the observations that the environment provides the agent with.

- A method called `step` to execute an action, which returns the current observation, the reward, and a flag indicating that the episode is over.
- A method called `reset`, which returns the environment to its initial state and obtains the first observation.

Let's now talk about these components of the environment in detail.

The action space

As mentioned, the actions that an agent can execute can be discrete, continuous, or a combination of the two.

Discrete actions are a fixed set of things that an agent can do, for example, directions in a grid like left, right, up, or down. Another example is a push button, which could be either pressed or released. Both states are mutually exclusive and this is the main characteristic of a discrete action space, where only one action from a finite set of actions is possible at a time.

A **continuous action** has a value attached to it, for example, a steering wheel, which can be turned at a specific angle, or an accelerator pedal, which can be pressed with different levels of force. A description of a continuous action includes the boundaries of the value that the action could have. In the case of a steering wheel, it could be from -720 degrees to 720 degrees. For an accelerator pedal, it's usually from 0 to 1.

Of course, we are not limited to a single action; the environment could take multiple actions, such as pushing multiple buttons simultaneously or steering the wheel and pressing two pedals (the brake and the accelerator). To support such cases, Gym defines a special container class that allows the nesting of several action spaces into one unified action.

The observation space

As discussed in *Chapter 1*, observations are pieces of information that an environment provides the agent with, on every timestamp, besides the reward. Observations can be as simple as a bunch of numbers or as complex as several multidimensional tensors containing color images from several cameras. An observation can even be discrete, much like action spaces. An example of a discrete observation space is a lightbulb, which could be in two states – on or off – given to us as a Boolean value.

So, you can see the similarity between actions and observations, and that is how they have been represented in Gym's classes. Let's look at a class diagram:

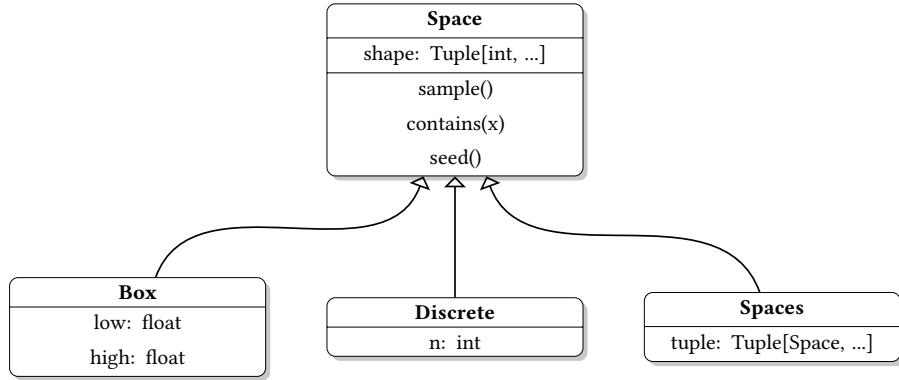


Figure 2.1: The hierarchy of the Space class in Gym

The basic abstract Space class includes one property and three methods that are relevant to us:

- `shape`: This property contain the shape of the space, identical to NumPy arrays.
- `sample()`: This returns a random sample from the space.
- `contains(x)`: This checks whether the argument, `x`, belongs to the space's domain.
- `seed()`: This method allows us to initialize a random number generator for the space and all subspaces. This is useful if you want to get reproducible environment behavior across several runs.

All these methods are abstract and reimplemented in each of the Space subclasses:

- The `Discrete` class represents a mutually exclusive set of items, numbered from 0 to `n-1`. If needed, you can redefine the starting index with the optional constructor argument `start`. The value `n` is a count of the items our `Discrete` object describes. For example, `Discrete(n=4)` can be used for an action space of four directions to move in [left, right, up, or down].
- The `Box` class represents an n -dimensional tensor of rational numbers with intervals [`low`, `high`]. For instance, this could be an accelerator pedal with one single value between 0.0 and 1.0, which could be encoded by `Box(low=0.0, high=1.0, shape=(1,), dtype=np.float32)`. Here, the `shape` argument is assigned a tuple of length 1 with a single value of 1, which gives us a one-dimensional tensor with a single value. The `dtype` parameter specifies the space's value type, and here, we specify it as a NumPy 32-bit float. Another example of `Box` could be an Atari screen observation (we will cover lots of Atari environments later), which is an RGB (red, green, and blue) image of size 210×160 : `Box(low=0, high=255, shape=(210, 160, 3), dtype=np.uint8)`. In this case, the `shape` argument is a tuple of three elements: the first dimension is the height of the image, the second is the width, and the third equals 3, which all correspond to three color planes for red, green, and blue, respectively. So, in total, every observation is a three-dimensional tensor with 100,800 bytes.

- The final child of Space is a Tuple class, which allows us to combine several Space class instances together. This enables us to create action and observation spaces of any complexity that we want. For example, imagine we want to create an action space specification for a car. The car has several controls that can be changed at every timestamp, including the steering wheel angle, brake pedal position, and accelerator pedal position. These three controls can be specified by three float values in one single Box instance. Besides these essential controls, the car has extra discrete controls, like a turn signal (which could be off, right, or left) or horn (on or off). To combine all of this into one action space specification class, we can use the following code:

```
Tuple(spaces=(
    Box(low=-1.0, high=1.0, shape=(3,), dtype=np.float32),
    Discrete(n=3),
    Discrete(n=2)
))
```

This flexibility is rarely used; for example, in this book, you will see only the Box and Discrete actions and observation spaces, but the Tuple class can be handy in some cases.

There are other Space subclasses defined in Gym, for example, Sequence (representing variable-length sequences), Text (strings), and Graph (where space is a set of nodes with connections between them). But the three that we have described are the most useful ones.

Every environment has two members of type Space: the `action_space` and `observation_space`. This allows us to create generic code that could work with any environment. Of course, dealing with the pixels of the screen is different from handling discrete observations (as in the former case, we may want to preprocess images with convolutional layers or with other methods from the computer vision toolbox); so, most of the time, this means optimizing the code for a particular environment or group of environments, but Gym doesn't prevent us from writing generic code.

The environment

The environment is represented in Gym by the Env class, which has the following members:

- `action_space`: This is the field of the Space class and provides a specification for allowed actions in the environment.
- `observation_space`: This field has the same Space class, but specifies the observations provided by the environment.
- `reset()`: This resets the environment to its initial state, returning the initial observation vector and the dict with extra information from the environment.

- `step()`: This method allows the agent to take the action and returns information about the outcome of the action:
 - The next observation
 - The local reward
 - The end-of-episode flag
 - The flag indicating a truncated episode
 - A dictionary with extra information from the environment

This method is a bit complicated; we will look at it in detail later in this section.

There are extra utility methods in the Env class, such as `render()`, which allows us to obtain the observation in a human-friendly form, but we won't use them. You can find the full list in Gym's documentation, but let's focus on the core Env methods: `reset()` and `step()`.

As `reset` is much simpler, we will start with it. The `reset()` method has no arguments; it instructs an environment to reset into its initial state and obtain the initial observation. Note that you have to call `reset()` after the creation of the environment. As you may remember from *Chapter 1*, the agent's communication with the environment may have an end (like a "Game Over" screen). Such sessions are called episodes, and after the end of the episode, an agent needs to start over. The value returned by this method is the first observation of the environment.

Besides the observation, `reset()` returns the second value – the dictionary with extra environment-specific information. Most standard environments return nothing in this dictionary, but more complicated ones (like TextWorld, an emulator for interactive-fiction games; we'll take a look at it later in the book) might return additional information that doesn't fit into standard observation.

The `step()` method is the central piece in the environment's functionality. It does several things in one call, which are as follows:

- Telling the environment which action we will execute in the next step
- Getting the new observation from the environment after this action
- Getting the reward the agent gained with this step
- Getting the indication that the episode is over
- Getting the flag which signals an episode truncation (when time limit is enabled, for example)
- Getting the dict with extra environment-specific information

The first item in the preceding list (action) is passed as the only argument to the `step()` method, and the rest are returned by this method. More precisely, this is a tuple (Python tuple and not the `Tuple` class we discussed in the previous section) of five elements (`observation`, `reward`, `done`, `truncated`, and `info`).

They have these types and meanings:

- **observation**: This is a NumPy vector or a matrix with observation data.
- **reward**: This is the float value of the reward.
- **done**: This is a Boolean indicator, which is `True` when the episode is over. If this value is `True`, we have to call `reset()` in the environment, as no more actions are possible.
- **truncated**: This is a Boolean indicator, which is `True` when the episode is truncated. For most environments, this is a `TimeLimit` (which is a way to limit length of episodes), but might have different meaning in some environments. This flag is separated from `done`, because in some scenarios it might be useful to distinguish situations “agent reached the end of episode” and “agent has reached the time limit of the environment.” If `truncated` is `True`, we also have to call `reset()` in the environment, the same as with the `done` flag.
- **info**: This could be anything environment-specific with extra information about the environment. The usual practice is to ignore this value in general RL methods.

You may have already got the idea of environment usage in an agent’s code – in a loop, we call the `step()` method with an action to perform until the `done` or `truncated` flags become `True`. Then, we can call `reset()` to start over. There is only one piece missing – how we create Env objects in the first place.

Creating an environment

Every environment has an unique name of the `EnvironmentName-vN` form, where `N` is the number used to distinguish between different versions of the same environment (when, for example, some bugs get fixed or some other major changes are made). To create an environment, the `gymnasium` package provides the `make(name)` function, whose only argument is the environment’s name in string form.

At the time of writing, Gymnasium version 0.29.1 (being installed with the `[atari]` extension) contains 1,003 environments with different names. Of course, all of these are not unique environments, as this list includes all versions of an environment. Additionally, the same environment can have different variations in the settings and observations spaces. For example, the Atari game Breakout has these environment names:

- **Breakout-v0, Breakout-v4**: The original Breakout with a random initial position and direction of the ball.
- **BreakoutDeterministic-v0, BreakoutDeterministic-v4**: Breakout with the same initial placement and speed vector of the ball.
- **BreakoutNoFrameskip-v0, BreakoutNoFrameskip-v4**: Breakout with every frame displayed to the agent. Without this, every action is executed for several consecutive frames.
- **Breakout-ram-v0, Breakout-ram-v4**: Breakout with the observation of the full Atari emulation memory (128 bytes) instead of screen pixels.

- **Breakout-ramDeterministic-v0, Breakout-ramDeterministic-v4:** Memory observation with the same initial state.
- **Breakout-ramNoFrameskip-v0, Breakout-ramNoFrameskip-v4:** Memory observation without frame skipping.

In total, there are 12 environments for a single game. In case you've never seen it before, here is a screenshot of its gameplay:



Figure 2.2: The gameplay of Breakout

Even after the removal of such duplicates, Gymnasium comes with an impressive list of 198 unique environments, which can be divided into several groups:

- **Classic control problems:** These are toy tasks that are used in optimal control theory and RL papers as benchmarks or demonstrations. They are usually simple, with low-dimension observation and action spaces, but they are useful as quick checks when implementing algorithms. Think about them as the “MNIST for RL” (MNIST is a handwriting digit recognition dataset from Yann LeCun, which you can find at <http://yann.lecun.com/exdb/mnist/>).
- **Atari 2600:** These are games from the classic game platform from the 1970s. There are 63 unique games.
- **Algorithmic:** These are problems that aim to perform small computation tasks, such as copying the observed sequence or adding numbers.

- **Box2D:** These are environments that use the Box2D physics simulator to learn walking or car control.
- **MuJoCo:** This is another physics simulator used for several continuous control problems.
- **Parameter tuning:** This is RL being used to optimize NN parameters.
- **Toy text:** These are simple grid world text environments.

Of course, the total number of RL environments supporting the Gym API is much larger. For example, The Farama Foundation maintains several repositories related to special RL topics like multi-agent RL, 3D navigation, robotics, and web automation. In addition, there are lots of third-party repositories. To get the idea, you can check out https://gymnasium.farama.org/environments/third_party_environments in the Gymnasium documentation.

But enough theory! Let's now look at a Python session working with one of Gym's environments.

The CartPole session

Let's apply our knowledge and explore one of the simplest RL environments that Gym provides.

```
$ python
>>> import gymnasium as gym
>>> e = gym.make("CartPole-v1")
```

Here, we have imported the `gymnasium` package and created an environment called `CartPole`. This environment is from the classic control group and its gist is to control the platform with a stick attached to its bottom part (see the following figure).

The trickiness is that this stick tends to fall right or left and you need to balance it by moving the platform to the right or left at every step.



Figure 2.3: The CartPole environment

The observation of this environment is four floating-point numbers containing information about the x coordinate of the stick's center of mass, its speed, its angle to the platform, and its angular speed. Of course, by applying some math and physics knowledge, it won't be complicated to convert these numbers into actions when we need to balance the stick, but our problem is different – how do we learn how to balance this system *without knowing* the exact meaning of the observed numbers and only by getting the reward? The reward in

this environment is 1, and it is given on every time step. The episode continues until the stick falls, so to get a more accumulated reward, we need to balance the platform in a way to avoid the stick falling.

This problem may look difficult, but in just two chapters, we will write the algorithm that will easily solve CartPole in minutes, without any idea about what the observed numbers mean. We will do it only by trial and error and using a bit of RL magic.

But now, let's continue with our session.

```
>>> obs, info = e.reset()
>>> obs
array([ 0.02100407,  0.02762252, -0.01519943, -0.0103739 ], dtype=float32)
>>> info
{}
```

Here, we reset the environment and obtained the first observation (we always need to reset the newly created environment). As I said, the observation is four numbers, so no surprises here. Let's now examine the action and observation space of the environment:

```
>>> e.action_space
Discrete(2)
>>> e.observation_space
Box([-4.8000002e+00 -3.4028235e+38 -4.1887903e-01 -3.4028235e+38], [4.8000002e+00
3.4028235e+38 4.1887903e-01 3.4028235e+38], (4,), float32)
```

The `action_space` field is of the `Discrete` type, so our actions will be just 0 or 1, where 0 means pushing the platform to the left and 1 is pushing to the right. The observation space is of `Box(4,)`, which means a vector of four numbers. The first list shown in the `observation_space` field is the low bound and the second is the high bound of parameters.

If you're curious, you can peek at the source code of the environment in the Gymnasium repository in the `cartpole.py` file at https://github.com/Farama-Foundation/Gymnasium/blob/main/gymnasium/envs/classic_control/cartpole.py#L40.

Documentation strings of the `CartPole` class provide all the details, including semantics of observation:

- **Cart position:** Value in $-4.8 \dots 4.8$ range
- **Cart velocity:** Value in $-\infty \dots \infty$ range
- **Pole angle:** Value in radians in $-0.418 \dots 0.418$ range
- **Pole angular velocity:** Value in $-\infty \dots \infty$ range

Python uses `float32` maximum and minimum values to represent infinity, which is why some entries in boundary vectors have values of scale 10^{38} . All those internal details are interesting to know, but absolutely not needed to solve the environment using RL methods. Let's go further and send an action to the environment:

```
>>> e.step(0)
(array([-0.01254663, -0.22985364, -0.01435183,  0.24902613], dtype=float32), 1.0, False,
False, {})
```

Here, we pushed our platform to the left by executing the action 0 and got the tuple of five elements:

- A new observation, which is a new vector of four numbers
- A reward of 1.0
- The done flag with value `False`, which means that the episode is not over yet and we are more or less okay with balancing the pole
- The truncated flag with value `False`, meaning that the episode was not truncated
- Extra information about the environment, which is an empty dictionary

Next, we will use the `sample()` method of the `Space` class on the `action_space` and `observation_space`.

```
>>> e.action_space.sample()
0
>>> e.action_space.sample()
1
>>> e.observation_space.sample()
array([-4.05354548e+00, -1.13992760e+38, -1.21235274e-01,  2.89040989e+38],
      dtype=float32)
>>> e.observation_space.sample()
array([-3.6149189e-01, -1.0301251e+38, -2.6193827e-01, -2.6395525e+36],
      dtype=float32)
```

This method returned a random sample from the underlying space, which in the case of our `Discrete` action space means a random number of 0 or 1, and for the observation space means a random vector of four numbers. The random sample of the observation space is not very useful, but the sample from the action space could be used when we are not sure how to perform an action.

This feature is especially handy because you don't know any RL methods yet, but we still want to play around with the Gym environment. Now that you know enough to implement your first randomly behaving agent for CartPole, let's do it.

The random CartPole agent

Although the environment is much more complex than our first example in section 2, the code of the agent is much shorter. This is the power of reusability, abstractions, and third-party libraries!

So, here is the code (you can find it in `Chapter02/02_cartpole_random.py`):

```
import gymnasium as gym

if __name__ == "__main__":
    env = gym.make("CartPole-v1")
    total_reward = 0.0
    total_steps = 0
    obs, _ = env.reset()
```

Here, we created the environment and initialized the counter of steps and the reward accumulator. On the last line, we reset the environment to obtain the first observation (which we will not use, as our agent is stochastic):

```
while True:
    action = env.action_space.sample()
    obs, reward, is_done, is_trunc, _ = env.step(action)
    total_reward += reward
    total_steps += 1
    if is_done:
        break

print("Episode done in %d steps, total reward %.2f" % (total_steps, total_reward))
```

In the preceding loop, after sampling a random action, we asked the environment to execute it and return to us the next observation (`obs`), the `reward`, the `is_done`, and the `is_trunc` flags. If the episode is over, we stop the loop and show how many steps we have taken and how much reward has been accumulated. If you start this example, you will see something like this (not exactly, though, due to the agent's randomness):

```
Chapter02$ python 02_cartpole_random.py
Episode done in 12 steps, total reward 12.00
```

On average, our random agent takes 12 to 15 steps before the pole falls and the episode ends. Most of the environments in Gym have a “reward boundary,” which is the average reward that the agent should gain during 100 consecutive episodes to “solve” the environment. For CartPole, this boundary is 195, which means that, on average, the agent must hold the stick for 195 time steps or longer. Using this perspective, our random agent’s performance looks poor. However, don’t be disappointed; we are just at the beginning, and soon you will solve CartPole and many other much more interesting and challenging environments.

Extra Gym API functionality

What we have discussed so far covers two-thirds of the Gym core API and the essential functions required to start writing agents. The rest of the API you can live without, but it will make your life easier and the code cleaner. So, let’s briefly cover the rest of the API.

Wrappers

Very frequently, you will want to extend the environment’s functionality in some generic way. For example, imagine an environment gives you some observations, but you want to accumulate them in some buffer and provide to the agent the N last observations. This is a common scenario for dynamic computer games, when one single frame is just not enough to get the full information about the game state. Another example is when you want to be able to crop or preprocess an image’s pixels to make it more convenient for the agent to digest, or if you want to normalize reward scores somehow. There are many such situations that have the same structure – you want to “wrap” the existing environment and add some extra logic for doing something. Gym provides a convenient framework for this – the `Wrapper` class.

The class structure is shown in *Figure 2.4*.

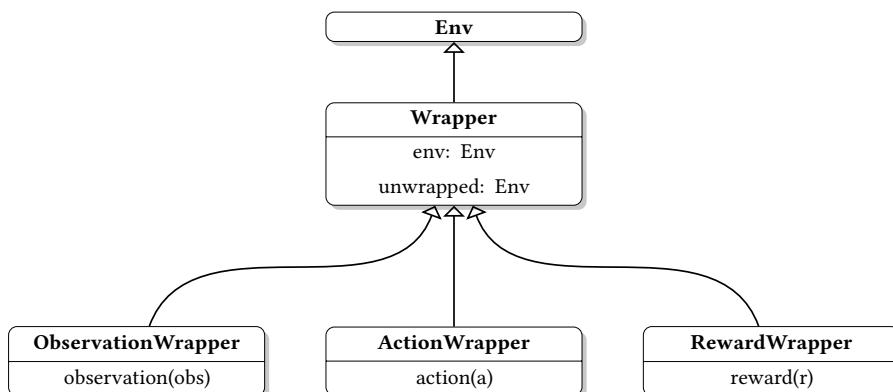


Figure 2.4: The hierarchy of the `Wrapper` classes in Gym

The `Wrapper` class inherits the `Env` class. Its constructor accepts the only argument – the instance of the `Env` class to be “wrapped.” To add extra functionality, you need to redefine the methods you want to extend, such as `step()` or `reset()`. The only requirement is to call the original method of the superclass. To simplify accessing the environment being wrapped, `Wrapper` has two properties: `env`, of the immediate environment we’re wrapping (which could be another wrapper as well), and property `unwrapped`, which is an `Env` without any wrappers.

To handle more specific requirements, such as a `Wrapper` class that wants to process only observations from the environment, or only actions, there are subclasses of `Wrapper` that allow the filtering of only a specific portion of information. They are as follows:

- `ObservationWrapper`: You need to redefine the `observation(obs)` method of the parent. The `obs` argument is an observation from the wrapped environment, and this method should return the observation that will be given to the agent.
- `RewardWrapper`: This exposes the `reward(rew)` method, which can modify the reward value given to the agent, for example, scale it to the needed range, add a discount based on some previous actions, or something like this.
- `ActionWrapper`: You need to override the `action(a)` method, which can tweak the action passed to the wrapped environment by the agent.

To make it slightly more practical, let’s imagine a situation where we want to intervene in the stream of actions sent by the agent and, with a probability of 10%, replace the current action with a random one. It might look like an unwise thing to do, but this simple trick is one of the most practical and powerful methods for solving the exploration/exploitation problem that we mentioned in *Chapter 1*. By issuing the random actions, we make our agent explore the environment and from time to time drift away from the beaten track of its policy. This is an easy thing to do using the `ActionWrapper` class (a full example is in `Chapter02/03_random_action_wrapper.py`):

```
import gymnasium as gym
import random

class RandomActionWrapper(gym.ActionWrapper):
    def __init__(self, env: gym.Env, epsilon: float = 0.1):
        super(RandomActionWrapper, self).__init__(env)
        self.epsilon = epsilon
```

Here, we initialized our wrapper by calling a parent’s `__init__` method and saving `epsilon` (the probability of a random action).

The following is a method that we need to override from a parent's class to tweak the agent's actions:

```
def action(self, action: gym.core.WrapperActType) -> gym.core.WrapperActType:
    if random.random() < self.epsilon:
        action = self.env.action_space.sample()
        print(f"Random action {action}")
    return action
return action
```

Every time we roll the die, and with the probability of `epsilon`, we sample a random action from the action space and return it instead of the action the agent has sent to us. Note that using `action_space` and wrapper abstractions, we were able to write abstract code, which will work with any environment from Gym. We also print the message on the console, just to illustrate that our wrapper is working. In the production code, this won't be necessary, of course.

Now it's time to apply our wrapper. We will create a normal CartPole environment and pass it to our `Wrapper` constructor:

```
if __name__ == "__main__":
    env = RandomActionWrapper(gym.make("CartPole-v1"))
```

From here on, we will use our wrapper as a normal `Env` instance, instead of the original `CartPole`. As the `Wrapper` class inherits the `Env` class and exposes the same interface, we can nest our wrappers as deep as we want. This is a powerful, elegant, and generic solution.

Here is almost the same code as in the random agent, except that every time, we issue the same action, 0, so our agent is dull and does the same thing:

```
obs = env.reset()
total_reward = 0.0

while True:
    obs, reward, done, _, _ = env.step(0)
    total_reward += reward
    if done:
        break

print(f"Reward got: {total_reward:.2f}")
```

By running the code, you should see that the wrapper is indeed working:

```
Chapter02$ python 03_random_action_wrapper.py
Random action 0
Random action 0
Reward got: 9.00
```

We should move on now and look at how you can render your environment during execution.

Rendering the environment

Another possibility that you should be aware of is rendering the environment. It is implemented with two wrappers: `HumanRendering` and `RecordVideo`.

Those two classes replace the original `Monitor` wrapper in the OpenAI Gym library, which was removed. This class was able to record the information about your agent's performance in a file, with an optional video recording of your agent in action.

With the Gymnasium library, you have two classes to check what's going on inside the environment. The first one is `HumanRendering`, which opens a separate graphical window in which the image from the environment is being shown interactively. To be able to render the environment (CartPole in our case), it has to be initialized with the `render_mode="rgb_array"` argument. This argument tells the environment to return pixels from its `render()` method, which is being called by the `HumanRendering` wrapper.

So, to use the `HumanRenderer` wrapper, you need to change the random agent's code (the full code is in `Chapter02/04_cartpole_random_monitor.py`):

```
if __name__ == "__main__":
    env = gym.make("CartPole-v1", render_mode="rgb_array")
    env = gym.wrappers.HumanRendering(env)
```

If you start the code, the window with environment rendering will appear. As our agent cannot balance the pole for too long (10-30 steps max), the window will disappear quite quickly, once the `env.close()` method is called.

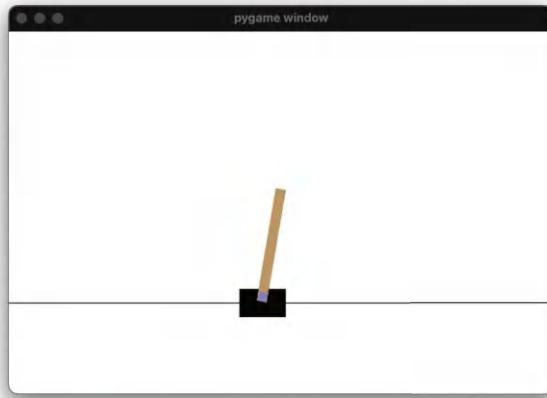


Figure 2.5: CartPole environment rendered by HumanRendering

Another wrapper that might be useful is `RecordVideo`, which captures the pixels from the environment and produces a video file of your agent in action. It is used in the same way as the human renderer, but requires an extra argument specifying the directory to store video files. If the directory doesn't exist, it will be created:

```
if __name__ == "__main__":
    env = gym.make("CartPole-v1", render_mode="rgb_array")
    env = gym.wrappers.RecordVideo(env, video_folder="video")
```

After starting the code, it reports the name of the video produced:

```
Chapter02$ python 04_cartpole_random_monitor.py
Moviepy - Building video Chapter02/video/rl-video-episode-0.mp4.
Moviepy - Writing video Chapter02/video/rl-video-episode-0.mp4

Moviepy - Done !
Moviepy - video ready Chapter02/video/rl-video-episode-0.mp4
Episode done in 30 steps, total reward 30.00
```

This wrapper is especially useful in situations when you're running your agent on a remote machine without the GUI.

More wrappers

Gymnasium provides lots of other wrappers, which we'll use in the upcoming chapters. It can do standardized preprocessing of Atari game images, do reward normalization, stack observation frames, do vectorization of an environment, do time limiting and much more.

The full list of available wrappers is available in the documentation, <https://gymnasium.farama.org/api/wrappers/>, and in the source code.

Summary

You have started to learn about the practical side of RL! In this chapter, we experimented with Gymnasium, with its tons of environments to play with. We studied its basic API and created a randomly behaving agent.

You also learned how to extend the functionality of existing environments in a modular way and became familiar with a way to render our agent's activity using wrappers. This will be heavily used in the upcoming chapters.

In the next chapter, we will do a quick DL recap using PyTorch, which is one of the most widely used DL toolkits.

3

Deep Learning with PyTorch

In the previous chapter, you became familiar with open source libraries, which provided you with a collection of **reinforcement learning (RL)** environments. However, recent developments in RL, and especially its combination with **deep learning (DL)**, now make it possible to solve much more challenging problems than ever before. This is partly due to the development of DL methods and tools. This chapter is dedicated to one such tool, PyTorch, which enables us to implement complex DL models with just a bunch of lines of Python code.

The chapter doesn't pretend to be a complete DL manual, as the field is very wide and dynamic; however, we will cover:

- The PyTorch library specifics and implementation details (assuming that you are already familiar with DL fundamentals)
- Higher-level libraries on top of PyTorch, with the aim of simplifying common DL problems
- The PyTorch Ignite library, which will be used in some examples



All of the examples in this chapter were updated for the latest(at the time of writing) PyTorch 2.3.1, which has changes in comparison to version 1.3.0, which was used in the second edition of this book. If you are using the old PyTorch, consider upgrading. Throughout this chapter, we will discuss the differences that are present in the latest version.

Tensors

A **tensor** is the fundamental building block of all DL toolkits. The name sounds rather mystical, but the underlying idea is that a tensor is just a multi-dimensional array. Using the analogy of school math, one single number is like a point, which is zero-dimensional, while a vector is one-dimensional like a line segment, and a matrix is a two-dimensional object. Three-dimensional number collections can be represented by a cuboid of numbers, but they don't have a separate name in the same way as a *matrix*. We can keep the term "tensor" for collections of higher dimensions.

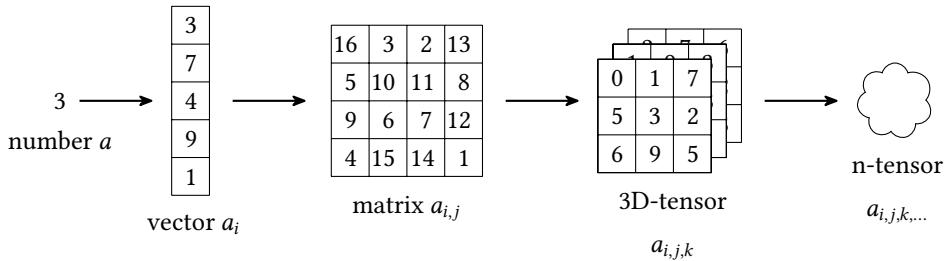


Figure 3.1: Going from a single number to an n -dimensional tensor

Another thing to note about tensors used in DL is that they are only partially related to tensors used in *tensor calculus* or *tensor algebra*. In DL, a tensor is any multi-dimensional array, but in mathematics, a tensor is a mapping between vector spaces, which might be represented as a multi-dimensional array in some cases, but has much more semantical payload behind it. Mathematicians usually frown at anybody who uses well-established mathematical terms to name different things, so be warned!

The creation of tensors

As we'll deal with tensors everywhere in this book, we need to be familiar with basic operations on them, and the most basic is how to create one. There are several ways to do this, and your choice might influence code readability and performance.

If you are familiar with the NumPy library (and you should be), then you already know that its central purpose is the handling of multi-dimensional arrays in a generic way. Even though in NumPy, such arrays aren't called tensors, they are, in fact, tensors. Tensors are used very widely in scientific computations as generic storage for data. For example, a color image could be encoded as a 3D tensor with the dimensions of width, height, and color plane.

Apart from dimensions, a tensor is characterized by the type of its elements. There are 13 types supported by PyTorch:

- Four float types: 16-bit, 32-bit, and 64-bit. 16-bit float has two variants: `float16` has more bits for precision while `bfloat16` has larger exponent part
- Three complex types: 32-bit, 64-bit, and 128-bit
- Five integer types: 8-bit signed, 8-bit unsigned, 16-bit signed, 32-bit signed, and 64-bit signed
- Boolean type

There are also four “quantized number” types, but they are using the preceding types, just with different bit representation and interpretation.

Tensors of different types are represented by different classes, with the most commonly used being `torch.FloatTensor` (corresponding to a 32-bit float), `torch.ByteTensor` (an 8-bit unsigned integer), and `torch.LongTensor` (a 64-bit signed integer). You can find names of other tensor types in the documentation.

There are three ways to create a tensor in PyTorch:

- By calling a constructor of the required type.
- By asking PyTorch to create a tensor with specific data for you. For example, you can use the `torch.zeros()` function to create a tensor filled with zero values.
- By converting a NumPy array or a Python list into a tensor. In this case, the type will be taken from the array’s type.

To give you examples of these methods, let’s look at a simple session:

```
$ python
>>> import torch
>>> import numpy as np
>>> a = torch.FloatTensor(3, 2)
>>> a
tensor([[0., 0.],
        [0., 0.],
        [0., 0.]])
```

Here, we imported both PyTorch and NumPy and created a new float tensor tensor of size 3×2 . As you can see, PyTorch initializes memory with zeros, which is a different behaviour from previous versions. Before, it just allocated memory and kept it uninitialized, which is slightly faster but less safe (as it might introduce tricky bugs and security issues). But you shouldn’t rely on this behaviour, as it might change again (or behave differently on different hardware backends) and always initialize the contents of the tensor. To do so, you can either use one of the tensor construct operators:

```
>>> torch.zeros(3, 4)
tensor([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

Or you can call the tensor modification method:

```
>>> a.zero_()
tensor([[0., 0.],
       [0., 0.],
       [0., 0.]])
```

There are two types of operation for tensors: **inplace** and **functional**. Inplace operations have an underscore appended to their name and operate on the tensor's content. After this, the object itself is returned. The functional equivalent creates a copy of the tensor with the performed modification, leaving the original tensor untouched. Inplace operations are usually more efficient from a performance and memory point of view, but modification of an existing tensor (especially if it is shared in different pieces of code) might lead to hidden bugs.

Another way to create a tensor by its constructor is to provide a Python iterable (for example, a list or tuple), which will be used as the contents of the newly created tensor:

```
>>> torch.FloatTensor([[1,2,3],[3,2,1]])
tensor([[1., 2., 3.],
       [3., 2., 1.]])
```

Here, we are creating the same tensor with zeros from the NumPy array:

```
>>> n = np.zeros(shape=(3, 2))
>>> n
array([[0., 0.],
       [0., 0.],
       [0., 0.]])
>>> b = torch.tensor(n)
>>> b
tensor([[0., 0.],
       [0., 0.],
       [0., 0.]], dtype=torch.float64)
```

The `torch.tensor` method accepts the NumPy array as an argument and creates a tensor of appropriate shape from it. In the preceding example, we created a NumPy array initialized by zeros, which created a double (64-bit float) array by default. So, the resulting tensor has the `DoubleTensor` type (which is shown in the example with the `dtype` value). Usually, in DL, double precision is not required and it adds an extra memory and performance overhead. Common practice is to use the 32-bit float type, or even the 16-bit float type, which is more than enough. To create such a tensor, you need to specify explicitly the type of NumPy array:

```
>>> n = np.zeros(shape=(3, 2), dtype=np.float32)
>>> torch.tensor(n)
tensor([[0., 0.],
        [0., 0.],
        [0., 0.]])
```

As an option, the type of the desired tensor could be provided to the `torch.tensor` function in the `dtype` argument. However, be careful, since this argument expects to get a PyTorch type specification and not the NumPy one. PyTorch types are kept in the `torch` package, for example, `torch.float32`, `torch.uint8`, and so on.

```
>>> n = np.zeros(shape=(3,2))
>>> torch.tensor(n, dtype=torch.float32)
tensor([[0., 0.],
        [0., 0.],
        [0., 0.]])
```

A note on compatibility

The `torch.tensor()` method and explicit PyTorch type specification were added in the 0.4.0 release, and this is a step toward the simplification of tensor creation. In previous versions, the `torch.from_numpy()` function was a recommended way to convert NumPy arrays, but it had issues with handling the combination of the Python list and NumPy arrays. This `from_numpy()` function is still present for backward compatibility, but it is deprecated in favor of the more flexible `torch.tensor()` method.

Scalar tensors

Since the 0.4.0 release, PyTorch has supported zero-dimensional tensors that correspond to scalar values (on the left of *Figure 3.1*). Such tensors can be the result of some operations, such as summing all values in a tensor. Previously, such cases were handled by the creation of a one-dimensional tensor (also known as vector) with a single dimension equal to one.

This solution worked, but it wasn't very simple, as extra indexation was needed to access the value. Now, zero-dimensional tensors are natively supported and returned by the appropriate functions, and they can be created by the `torch.tensor()` function. To access the actual Python value of such a tensor, we can use the special `item()` method:

```
>>> a = torch.tensor([1,2,3])
>>> a
tensor([1, 2, 3])
>>> s = a.sum()
>>> s
tensor(6)
>>> s.item()
6
>>> torch.tensor(1)
tensor(1)
```

Tensor operations

There are lots of operations that you can perform on tensors, and there are too many to list them all. Usually, it's enough to search in the PyTorch documentation at <http://pytorch.org/docs/>. I need to mention that there are two places to look for operations:

- The `torch` package: The function usually accepts the tensor as an argument.
- The `tensor` class: The function operates on the called tensor.

Most of the time, tensor operations in PyTorch are trying to correspond to their NumPy equivalent, so if there is some not-very-specialized function in NumPy, then there is a good chance that PyTorch will also have it. Examples are `torch.stack()`, `torch.transpose()`, and `torch.cat()`. This is very convenient, as NumPy is a very widely used library (especially in the scientific community), so your PyTorch code becomes readable by anyone familiar with NumPy without looking into the documentation.

GPU tensors

PyTorch transparently supports CUDA GPUs, which means that all operations have two versions — CPU and GPU — that are automatically selected. The decision is made based on the type of tensors that you are operating on.

Every tensor type that I mentioned is for CPU and has its GPU equivalent. The only difference is that GPU tensors reside in the `torch.cuda` package, instead of just `torch`. For example, `torch.FloatTensor` is a 32-bit float tensor that resides in CPU memory, but `torch.cuda.FloatTensor` is its GPU counterpart.



In fact, under the hood, PyTorch supports not just CPU and CUDA; it has a notion of *backend*, which is an abstract computation device with memory. Tensors could be allocated in the backend's memory and computations could be performed on them. For example, on Apple hardware, PyTorch supports **Metal Performance Shaders (MPS)** as a backend called `mps`. In this chapter, we focus on CPU and GPU as the mostly used backends, but your PyTorch code could be executed on much more fancier hardware without major modifications.

To convert from CPU to GPU, there is a tensor method, `to(device)`, that creates a copy of the tensor to a specified device (this could be CPU or GPU). If the tensor is already on the device, nothing happens and the original tensor will be returned. The device type can be specified in different ways. First of all, you can just pass a string name of the device, which is “cpu” for CPU memory or “cuda” for GPU. A GPU device could have an optional device index specified after the colon; for example, the second GPU card in the system could be addressed by “cuda:1” (the index is zero-based).

Another slightly more efficient way to specify a device in the `to()` method is by using the `torch.device` class, which accepts the device name and optional index. To access the device that your tensor is currently residing in, it has a `device` property:

```
>>> a = torch.FloatTensor([2,3])
>>> a
tensor([2., 3.])
>>> ca = a.to('cuda')
>>> ca
tensor([2., 3.], device='cuda:0')
```

Here, we created a tensor on the CPU, then copied it to GPU memory. Both copies can be used in computations, and all GPU-specific machinery is transparent to the user:

```
>>> a+1
tensor([3., 4.])
>>> ca + 1
tensor([3., 4.], device='cuda:0')
>>> ca.device
device(type='cuda', index=0)
```



The `to()` method and `torch.device` class were introduced in 0.4.0. In previous versions, copying between CPU and GPU was performed by separate tensor methods, `cpu()` and `cuda()`, respectively, which required adding the extra lines of code to explicitly convert tensors into their CUDA versions. In newer PyTorch versions, you can create a desired `torch.device` object at the beginning of the program and use `to(device)` on every tensor that you're creating. The old methods in the tensor, `cpu()` and `cuda()`, are still present and might be handy if you want to ensure that a tensor is in CPU or GPU memory regardless of its original location.

Gradients

Even with transparent GPU support, all of this dancing with tensors isn't worth bothering without one "killer feature" — the automatic computation of gradients. This functionality was originally implemented in the Caffe toolkit and then became the de facto standard in DL libraries.

Earlier, computing gradients manually was extremely painful to implement and debug, even for the simplest **neural network (NN)**. You had to calculate derivatives for all your functions, apply the chain rule, and then implement the result of the calculations, praying that everything was done right. This could be a very useful exercise for understanding the nuts and bolts of DL, but it wasn't something that you wanted to repeat over and over again by experimenting with different NN architectures.

Luckily, those days have gone now, much like programming your hardware using a soldering iron and vacuum tubes! Now, defining an NN of hundreds of layers requires nothing more than assembling it from predefined building blocks or, in the extreme case of you doing something fancy, defining the transformation expression manually.

All gradients will be carefully calculated for you, backpropagated, and applied to the network. To be able to achieve this, you need to define your network architecture using DL library primitives. In *Figure 3.2*, I have outlined the direction of the data and gradients flow during the optimization process:

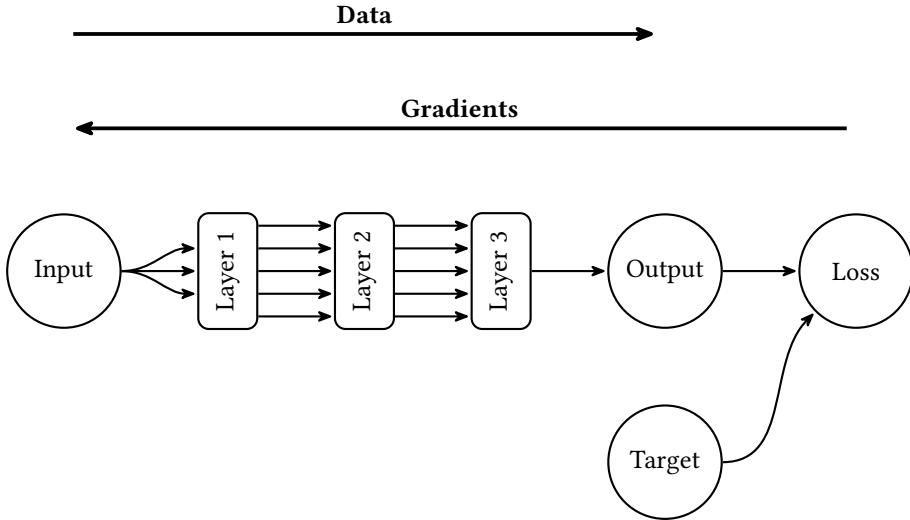


Figure 3.2: Data and gradients flowing through the NN

What can make a fundamental difference is how your gradients are calculated. There are two approaches:

- **Static graph:** In this method, you need to define your calculations in advance and it won't be possible to change them later. The graph will be processed and optimized by the DL library before any computation is made. This model is implemented in TensorFlow (versions before 2.0), Theano, and many other DL toolkits.
- **Dynamic graph:** You don't need to define your graph in advance exactly as it will be executed; you just need to execute operations that you want to use for data transformation on your actual data. During this, the library will record the order of the operations performed, and when you ask it to calculate gradients, it will unroll its history of operations, accumulating the gradients of the network parameters. This method is also called notebook gradients and it is implemented in PyTorch, Chainer, and some others.

Both methods have their strengths and weaknesses. For example, a static graph is usually faster, as all computations can be moved to the GPU, minimizing the data transfer overhead. Additionally, in a static graph, the library has much more freedom in optimizing the order that computations are performed in or even removing parts of the graph.

On the other hand, although a dynamic graph has a higher computation overhead, it gives a developer much more freedom. For example, they can say, "For this piece of data, I can apply this network two times, and for this piece of data, I'll use a completely different model with gradients clipped by the batch mean".

Another very appealing strength of the dynamic graph model is that it allows you to express your transformation more naturally and in a more “Pythonic” way. In the end, it’s just a Python library with a bunch of functions, so just call them and let the library do the magic.



Since version 2.0, PyTorch introduced the `torch.compile` function, which speeds up PyTorch code by JIT-compiling the code into optimized kernels. This is an evolution of the *TorchScript* and *FX Tracing* compiling methods from earlier versions.

From a historical perspective, this is highly amusing how originally radically different approaches of TensorFlow (static graph) and PyTorch (dynamic graph) are fusing into each other over time. Nowadays, PyTorch supports `compile()` and TensorFlow has “eager execution mode”.

Tensors and gradients

PyTorch tensors have a built-in gradient calculation and tracking machinery, so all you need to do is convert the data into tensors and perform computations using the tensor methods and functions provided by `torch`. Of course, if you need to access underlying low-level details, you always can, but most of the time, PyTorch does what you’re expecting.

There are several attributes related to gradients that every tensor has:

- `grad`: A property that holds a tensor of the same shape containing computed gradients.
- `is_leaf`: Equals `True` if this tensor was constructed by the user and `False` if the object is a result of function transformation (in other words, have a parent in the computation graph).
- `requires_grad`: Equals `True` if this tensor requires gradients to be calculated. This property is inherited from leaf tensors, which get this value from the tensor construction step (`torch.zeros()` or `torch.tensor()` and so on). By default, the constructor has `requires_grad=False`, so if you want gradients to be calculated for your tensor, then you need to explicitly say so.

To make all of this gradient-leaf machinery clearer, let’s consider this session:

```
>>> v1 = torch.tensor([1.0, 1.0], requires_grad=True)
>>> v2 = torch.tensor([2.0, 2.0])
```

Here, we created two tensors. The first requires gradients to be calculated and the second doesn’t.

Next, we have added both vectors element-wise (which is vector [3, 3]), doubled every element, and summed them together:

```
>>> v_sum = v1 + v2
>>> v_sum
tensor([3., 3.], grad_fn=<AddBackward0>)
>>> v_res = (v_sum*2).sum()
>>> v_res
tensor(12., grad_fn=<SumBackward0>)
```

The result is a zero-dimensional tensor with the value 12. Okay, so far this is just a simple math. Now let's look at the underlying graph that our expressions created:

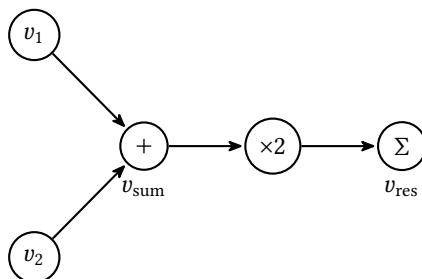


Figure 3.3: Graph representation of the expression

If we check the attributes of our tensors, then we will find that $v1$ and $v2$ are the only leaf nodes and every variable, except $v2$, requires gradients to be calculated:

```
>>> v1.is_leaf, v2.is_leaf
(True, True)
>>> v_sum.is_leaf, v_res.is_leaf
(False, False)
>>> v1.requires_grad
True
>>> v2.requires_grad
False
>>> v_sum.requires_grad
True
>>> v_res.requires_grad
True
```

As you can see, the property `requires_grad` is sort of “sticky”: if one of the variables involved in computations has it set to `True`, all subsequent nodes also have it. This is logical behaviour, as we normally need gradients to be calculated for all intermediate steps in our computation. But “calculation” doesn't mean they will be preserved in the `.grad` field.

For memory efficiency, gradients are stored only for leaf nodes with `requires_grad=True`. If you want to keep gradients in the non-leaf nodes, you need to call their `retain_grad()` method, which tells PyTorch to keep the gradients for non-leaf node.

Now, let's tell PyTorch to calculate the gradients of our graph:

```
>>> v_res.backward()  
>>> v1.grad  
tensor([2., 2.])
```

By calling the `backward` function, we asked PyTorch to calculate the numerical derivative of the `v_res` variable with respect to any variable that our graph has. In other words, what influence do small changes to the `v_res` variable have on the rest of the graph? In our particular example, the value of 2 in the gradients of `v1` means that by increasing any element of `v1` by one, the resulting value of `v_res` will grow by two.

As mentioned, PyTorch calculates gradients only for leaf tensors with `requires_grad=True`. Indeed, if we try to check the gradients of `v2`, we get nothing:

```
>>> v2.grad
```

The reason for that is efficiency in terms of computations and memory. In real life, our network can have millions of optimized parameters, with hundreds of intermediate operations performed on them. During gradient descent optimization, we are not interested in gradients of any intermediate matrix multiplication; the only things we want to adjust in the model are gradients of loss with respect to model parameters (weights). Of course, if you want to calculate the gradients of input data (it could be useful if you want to generate some adversarial examples to fool the existing NN or adjust pretrained word embeddings), then you can easily do so by passing `requires_grad=True` on tensor creation.

Basically, you now have everything needed to implement your own NN optimizer. The rest of this chapter is about extra, convenient functions, which will provide you with higher-level building blocks of NN architectures, popular optimization algorithms, and common loss functions. However, don't forget that you can easily reimplement all of these bells and whistles in any way that you like. This is why PyTorch is so popular among DL researchers — for its elegance and flexibility.



Compatibility

Support of gradients calculation in tensors is one of the major changes in PyTorch 0.4.0. In previous versions, graph tracking and gradients accumulation were done in a separate, very thin class, `Variable`. This worked as a wrapper around the tensor and automatically saved the history of computations in order to be able to backpropagate. This class is still present in 2.2.0 (available in `torch.autograd`), but it is deprecated and will go away soon, so new code should avoid using it. From my perspective, this change is great, as the `Variable` logic was really thin, but it still required extra code and the developer's attention to wrap and unwrap tensors. Now, gradients are a built-in tensor property, which makes the API much cleaner.

NN building blocks

In the `torch.nn` package, you will find tons of predefined classes providing you with the basic functionality blocks. All of them are designed with practice in mind (for example, they support mini-batches, they have sane default values, and the weights are properly initialized). All modules follow the convention of *callable*, which means that the instance of any class can act as a function when applied to its arguments. For example, the `Linear` class implements a feed-forward layer with optional bias:

```
>>> l = nn.Linear(2, 5)
>>> v = torch.FloatTensor([1, 2])
>>> l(v)
tensor([-0.1039, -1.1386,  1.1376, -0.3679, -1.1161], grad_fn=<ViewBackward0>)
```

Here, we created a randomly initialized feed-forward layer, with two inputs and five outputs, and applied it to our float tensor. All classes in the `torch.nn` packages inherit from the `nn.Module` base class, which you can use to implement your own higher-level NN blocks. You will see how you can do this in the next section, but, for now, let's look at useful methods that all `nn.Module` children provide. They are as follows:

- `parameters()`: This function returns an iterator of all variables that require gradient computation (that is, module weights).
- `zero_grad()`: This function initializes all gradients of all parameters to zero.
- `to(device)`: This function moves all module parameters to a given device (CPU or GPU).
- `state_dict()`: This function returns the dictionary with all module parameters and is useful for model serialization.
- `load_state_dict()`: This function initializes the module with the state dictionary.

The whole list of available classes can be found in the documentation at <http://pytorch.org/docs>.

Now, I should mention one very convenient class that allows you to combine other layers into the pipe: `Sequential`. The best way to demonstrate `Sequential` is through an example:

```
>>> s = nn.Sequential(  
... nn.Linear(2, 5),  
... nn.ReLU(),  
... nn.Linear(5, 20),  
... nn.ReLU(),  
... nn.Linear(20, 10),  
... nn.Dropout(p=0.3),  
... nn.Softmax(dim=1))  
>>> s  
Sequential(  
    (0): Linear(in_features=2, out_features=5, bias=True)  
    (1): ReLU()  
    (2): Linear(in_features=5, out_features=20, bias=True)  
    (3): ReLU()  
    (4): Linear(in_features=20, out_features=10, bias=True)  
    (5): Dropout(p=0.3, inplace=False)  
    (6): Softmax(dim=1)  
)
```

Here, we defined a three-layer NN with softmax on output, applied along dimension 1 (dimension 0 is batch samples), **rectified linear unit (ReLU)** nonlinearities, and dropout. Let's push something through it:

```
>>> s(torch.FloatTensor([[1,2]]))  
tensor([0.0847, 0.1145, 0.1063, 0.1458, 0.0873, 0.1063, 0.0864, 0.0821, 0.0894,  
      0.0971]), grad_fn=<SoftmaxBackward0>)
```

So, our mini-batch of one vector successfully traversed through the network!

Custom layers

In the previous section, I briefly mentioned the `nn.Module` class as a base parent for all NN building blocks exposed by PyTorch. It's not just a unifying parent for the existing layers — it's much more than that. By subclassing the `nn.Module` class, you can create your own building blocks, which can be stacked together, reused later, and integrated into the PyTorch framework flawlessly.

At its core, the `nn.Module` provides quite rich functionality to its children:

- It tracks all submodules that the current module includes. For example, your building block can have two feed-forward layers used somehow to perform the block's transformation. To keep track of (register) the submodule, you just need to assign it to the class's field.

- It provides functions to deal with all parameters of the registered submodules. You can obtain a full list of the module's parameters (`parameters()` method), zero its gradients (`zero_grads()` method), move to CPU or GPU (`to(device)` method), serialize and deserialize the module (`state_dict()` and `load_state_dict()`), and even perform generic transformations using your own callable (`apply()` method).
- It establishes the convention of `Module` application to data. Every module needs to perform its data transformation in the `forward()` method by overriding it.
- There are some more functions, such as the ability to register a hook function to tweak module transformation or gradients flow, but they are more for advanced use cases.

These functionalities allow us to nest our submodels into higher-level models in a unified way, which is extremely useful when dealing with complexity. It could be a simple one-layer linear transformation or a 1001-layer **residual NN (ResNet)** monster, but if they follow the conventions of `nn.Module`, then both of them could be handled in the same way. This is very handy for code reusability and simplification (by hiding non-relevant implementation details).

To make our life simpler, when following the above conventions, PyTorch authors simplified the creation of modules through careful design and a good dose of Python magic. So, to create a custom module, we usually have to do only two things – register submodules and implement the `forward()` method.

Let's look at how this can be done for our `Sequential` example from the previous section, but in a more generic and reusable way (the full sample is `Chapter03/01_modules.py`). The following is our module class that inherits `nn.Module`:

```
[python]
class OurModule(nn.Module):
    def __init__(self, num_inputs, num_classes, dropout_prob=0.3):
        super(OurModule, self).__init__()
        self.pipe = nn.Sequential(
            nn.Linear(num_inputs, 5),
            nn.ReLU(),
            nn.Linear(5, 20),
            nn.ReLU(),
            nn.Linear(20, num_classes),
            nn.Dropout(p=dropout_prob),
            nn.Softmax(dim=1)
        )
```

In the constructor, we pass three parameters: the input size, the output size, and the optional dropout probability. The first thing we need to do is call the parent's constructor to let it initialize itself.

In the second step in the preceding code, we create an already familiar `nn.Sequential` with a bunch of layers and assign it to our class field named `pipe`. By assigning a `Sequential` instance to our object's field, we will automatically register this module (`nn.Sequential` inherits from `nn.Module`, as does everything in the `nn` package). To register it, we don't need to call anything, we just need to assign our submodules to fields. After the constructor finishes, all those fields will be registered automatically. If you really want to, there is a function in `nn.Module` to register submodules called `add_module()`. It might be useful if your module can have variable number of layers and they need to be created programmatically.

Next, we must override the `forward` function with our implementation of data transformation:

```
[python]
def forward(self, x):
    return self.pipe(x)
```

As our module is a very simple wrapper around the `Sequential` class, we just need to ask `self.pipe` to transform the data. Note that to apply a module to the data, we need to call the module as a callable (that is, pretend that the module instance is a function and call it with the arguments) and not use the `forward()` function of the `nn.Module` class. This is because `nn.Module` overrides the `__call__()` method, which is being used when we treat an instance as callable. This method does some `nn.Module` magic and calls our `forward()` method. If we call `forward()` directly, we will intervene with the `nn.Module` duty, which can give wrong results.

So, that's what we need to do to define our own module. Now, let's use it:

```
if __name__ == "__main__":
    net = OurModule(num_inputs=2, num_classes=3)
    print(net)
    v = torch.FloatTensor([[2, 3]])
    out = net(v)
    print(out)
    print("Cuda's availability is %s" % torch.cuda.is_available())
    if torch.cuda.is_available():
        print("Data from cuda: %s" % out.to('cuda'))
```

We create our module, providing it with the desired number of inputs and outputs, then we create a tensor and ask our module to transform it, following the same convention of using it as callable. After that, we print our network's structure (`nn.Module` overrides `__str__()` and `__repr__()`) to represent the inner structure in a nice way. The last thing we show is the result of the network's transformation.

The output of our code should look like this:

```
Chapter03$ python 01_modules.py
OurModule(
    (pipe): Sequential(
        (0): Linear(in_features=2, out_features=5, bias=True)
        (1): ReLU()
        (2): Linear(in_features=5, out_features=20, bias=True)
        (3): ReLU()
        (4): Linear(in_features=20, out_features=3, bias=True)
        (5): Dropout(p=0.3, inplace=False)
        (6): Softmax(dim=1)
    )
)
tensor([[0.3297, 0.3854, 0.2849]], grad_fn=<SoftmaxBackward0>)
Cuda's availability is False
```

Of course, everything that was said about the dynamic nature of PyTorch is still true. The `forward()` method is called for every batch of data, so if you want to do some complex transformations based on the data you need to process, like hierarchical softmax or a random choice of network to apply, then nothing can stop you from doing so. The count of arguments to your module is also not limited by one parameter. So, if you want, you can write a module with multiple required parameters and dozens of optional arguments, and it will be fine.

Next, we need to get familiar with two important pieces of the PyTorch library that will simplify our lives: loss functions and optimizers.

Loss functions and optimizers

The network that transforms input data into output is not the only thing we need for training. We also define our learning objective, which has to be a function that accepts two arguments — the network’s output and the desired output. Its responsibility is to return to us a single number — how close the network’s prediction is from the desired result. This function is called the **loss function**, and its output is the **loss value**. Using the loss value, we calculate gradients of network parameters and adjust them to decrease this loss value, which pushes our model to better results in the future. Both the loss function and the method of tweaking a network’s parameters by gradient are so common and exist in so many forms that both of them form a significant part of the PyTorch library. Let’s start with loss functions.

Loss functions

Loss functions reside in the `nn` package and are implemented as an `nn.Module` subclass. Usually, they accept two arguments: output from the network (prediction) and desired output (ground-truth data, which is also called the label of the data sample). At the time of writing, PyTorch 2.3.1 contains over 20 different loss functions and, of course, nothing stops you from writing any custom function you want to optimize.

The most commonly used standard loss functions are:

- `nn.MSELoss`: The mean square error between arguments, which is the standard loss for regression problems.
- `nn.BCELoss` and `nn.BCEWithLogits`: Binary cross-entropy loss. The first version expects a single probability value (usually it's the output of the `Sigmoid` layer), while the second version assumes raw scores as input and applies `Sigmoid` itself. The second way is usually more numerically stable and efficient. These losses (as their names suggest) are frequently used in binary classification problems.
- `nn.CrossEntropyLoss` and `nn.NLLLoss`: Famous “maximum likelihood” criteria that are used in multi-class classification problems. The first version expects raw scores for each class and applies `LogSoftmax` internally, while the second expects to have log probabilities as the input.

There are other loss functions available and you are always free to write your own `Module` subclass to compare the output and target. Now, let's look at the second piece of the optimization process.

Optimizers

The responsibility of the basic optimizer is to take the gradients of model parameters and change these parameters in order to decrease the loss value. By decreasing the loss value, we are pushing our model toward the desired output, which can give us hope for better model performance in the future. Changing parameters may sound simple, but there are lots of details here and the optimizer procedure is still a hot research topic. In the `torch.optim` package, PyTorch provides lots of popular optimizer implementations, and the most widely known are as follows:

- `SGD`: A vanilla stochastic gradient descent algorithm with an optional momentum extension
- `RMSprop`: An optimizer proposed by Geoffrey Hinton
- `Adagrad`: An adaptive gradients optimizer
- `Adam`: A quite successful and popular combination of both `RMSprop` and `Adagrad`

All optimizers expose the unified interface, which makes it easy to experiment with different optimization methods (sometimes the optimization method can really make a difference in convergence dynamics and the final result). On construction, you need to pass an iterable of tensors, which will be modified during the optimization process.

The usual practice is to pass the result of the `params()` call of the upper-level `nn.Module` instance, which will return an iterable of all leaf tensors with gradients.

Now, let's discuss the common blueprint of a training loop:

```
1  for batch_x, batch_y in iterate_batches(data, batch_size=N):
2      batch_x_t = torch.tensor(batch_x)
3      batch_y_t = torch.tensor(batch_y)
4      out_t = net(batch_x_t)
5      loss_t = loss_function(out_t, batch_y_t).
6      loss_t.backward()
7      optimizer.step()
8      optimizer.zero_grad()
```

Usually, you iterate over your data over and over again (one iteration over a full set of examples is called an *epoch*). Data is usually too large to fit into CPU or GPU memory at once, so it is split into batches of equal size. Every batch includes data samples and target labels, and both of them have to be tensors (lines 2 and 3).

You pass data samples to your network (line 4) and feed its output and target labels to the loss function (line 5). The result of the loss function shows the “badness” of the network result relative to the target labels. As input to the network and the network’s weights are tensors, all transformations of your network are nothing more than a graph of operations with intermediate tensor instances. The same is true for the loss function — its result is also a tensor of one single loss value.

Every tensor in this computation graph remembers its parent, so to calculate gradients for the whole network, all you need to do is call the `backward()` function on a loss function result (line 6). The result of this call will be the unrolling of the graph of the performed computations and the calculating of gradients for every leaf tensor with `require_grad=True`. Usually, such tensors are our model’s parameters, such as the weights and biases of feed-forward networks, and convolution filters. Every time a gradient is calculated, it is accumulated in the `tensor.grad` field, so one tensor can participate in a transformation multiple times and its gradients will be properly summed together. For example, one single **recurrent neural network (RNN)** cell could be applied to multiple input items.

After the `loss.backward()` call is finished, we have the gradients accumulated, and now it’s time for the optimizer to do its job — it takes all gradients from the parameters we have passed to it on construction and applies them. All this is done with the method `step()` (line 7).

The last, but not least, piece of the training loop is our responsibility to zero gradients of parameters. This can be done by calling `zero_grad()` on our network, but, for our convenience, the optimizer also exposes such a call, which does the same thing (line 8). Sometimes, `zero_grad()` is placed at the beginning of the training

loop, but it doesn't matter much.

The preceding scheme is a very flexible way to perform optimization and it can fulfill the requirements even in sophisticated research. For example, you can have two optimizers tweaking the options of different models on the same data (and this is a real-life scenario from **generative adversarial network (GAN)** training).

So, we are done with the essential functionality of PyTorch required to train NNs. This chapter ends with a practical medium-size example to demonstrate all the concepts covered, but before we get to it, we need to discuss one important topic that is essential for an NN practitioner — monitoring the learning process.

Monitoring with TensorBoard

If you have ever tried to train an NN on your own, then you will know how painful and uncertain it can be. I'm not talking about following the existing tutorials and demos, when all the hyperparameters are already tuned for you, but about taking some data and creating something from scratch. Even with modern DL high-level toolkits, where all best practices, such as proper weights initialization; optimizers' betas, gammas, and other options set to sane defaults; and tons of other stuff hidden under the hood, there are still lots of decisions that you have to make, hence lots of things that could go wrong. As a result, your code almost never works from the first run, and this is something that you should get used to.

Of course, with practice and experience, you will develop a strong understanding of the possible causes of problems, but this needs input data about what's going on inside your network. So, you need to be able to peek inside your training process somehow and observe its dynamics. Even small networks (such as tiny MNIST tutorial networks) could have hundreds of thousands of parameters with quite nonlinear training dynamics.

DL practitioners have developed a list of things that you should observe during your training, which usually includes the following:

- Loss value, which normally consists of several components like base loss and regularization losses. You should monitor both the total loss and the individual components over time.
- Results of validation on training and test datasets.
- Statistics about gradients and weights.
- Values produced by the network. For example, if you are solving a classification problem, you definitely want to measure the entropy of predicted class probabilities. In the case of a regression problem, raw predicted values can give tons of data about the training.
- Learning rates and other hyperparameters, if they are adjusted over time.

The list could be much longer and include domain-specific metrics, such as word embedding projections, audio samples, and images generated by GANs. You also may want to monitor values related to training speed, like how long an epoch takes, to see the effect of your optimizations or problems with hardware.

To cut a long story short, you need a generic solution to track lots of values over time and represent them for analysis, preferably developed especially for DL (just imagine looking at such statistics using an Excel spreadsheet). Luckily, such tools exist, and we will explore them next.

TensorBoard 101

When the first edition of this book was written, there wasn't too much choice for NN monitoring. As time has passed by and new people and companies have become involved with the pursuit of ML and DL, more new tools have appeared, for example, MLflow <https://mlflow.org/>. In this book, we will still focus on the TensorBoard utility from TensorFlow, but you might consider trying other alternatives.

From the first public version, TensorFlow included a special tool called TensorBoard, which was developed to solve the problem we are talking about – how to observe and analyze various NN characteristics during and after the training. TensorBoard is a powerful, generic solution with a large community and it looks quite pretty:

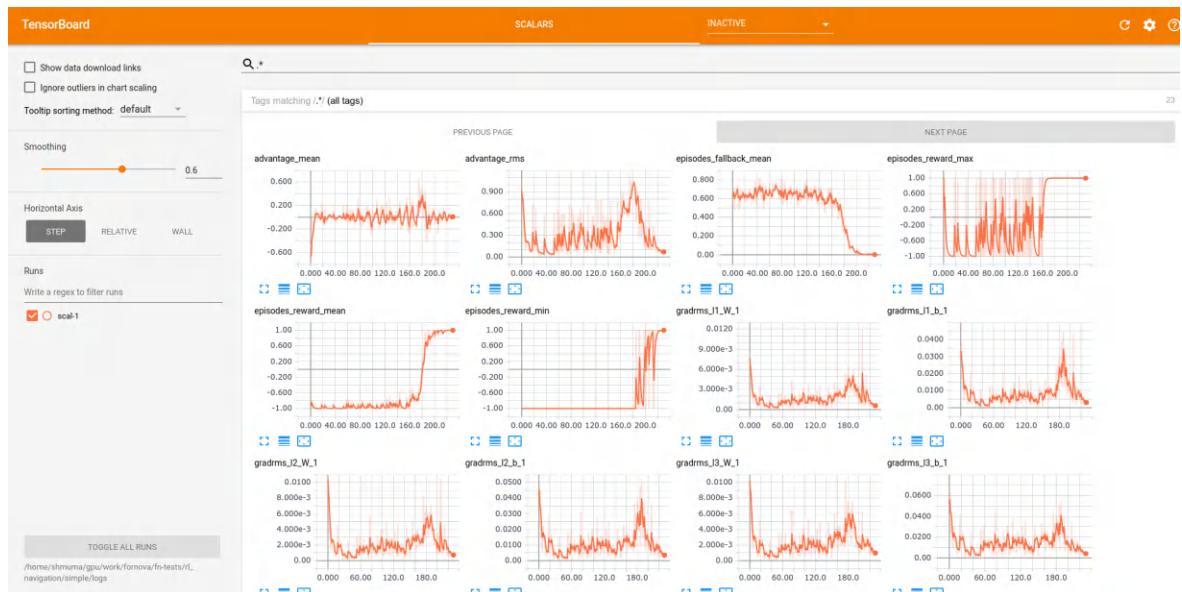


Figure 3.4: The TensorBoard web interface (for better visualization, refer to <https://packt.link/gbp/9781835882702>)

From the architecture point of view, TensorBoard is a Python web service that you can start on your computer, passing it the directory where your training process will save values to be analyzed. Then, you point your browser to TensorBoard's port (usually 6006), and it shows you an interactive web interface with values updated in real time, as shown in *Figure 3.4*. It's nice and convenient, especially when your training is performed on a remote machine somewhere in the cloud.

Originally, TensorBoard was deployed as a part of TensorFlow, but after some time, it has been moved to a separate project (it's still being maintained by Google) and it has its own package name. However, TensorBoard still uses the TensorFlow data format, so we will need to write this data from our PyTorch program. Several years ago, it required third-party libraries to be installed, but nowadays, PyTorch already comes with support of this data format (available in the `torch.utils.tensorboard` package).

Plotting metrics

To give you an impression of how simple is to use TensorBoard, let's consider a small example that is not related to NNs, but is just about writing values into TensorBoard (the full example code is in `Chapter03/02_tensorboard.py`).

In the following code, we import the required packages, create a writer of data, and define functions that we are going to visualize:

```
import math
from torch.utils.tensorboard.writer import SummaryWriter

if __name__ == "__main__":
    writer = SummaryWriter()
    funcs = {"sin": math.sin, "cos": math.cos, "tan": math.tan}
```

By default, `SummaryWriter` will create a unique directory in the `runs` directory for every launch, to be able to compare different rounds of training. The name of the new directory includes the current date and time, and the hostname. To override this, you can pass the `log_dir` argument to `SummaryWriter`. You can also add a suffix to the name of the directory by passing a `comment` argument, for example, to capture different experiments' semantics, such as `dropout=0.3` or `strong_regularisation`.

Next, we loop over angle ranges in degrees:

```
for angle in range(-360, 360):
    angle_rad = angle * math.pi / 180
    for name, fun in funcs.items():
        val = fun(angle_rad)
        writer.add_scalar(name, val, angle)

writer.close()
```

Here, we convert the angle ranges into radians and calculate our functions' values. Every value is added to the writer using the `add_scalar` function, which takes three arguments: the name of the parameter, its value, and the current iteration (which has to be an integer).

The last thing we need to do after the loop is close the writer. Note that the writer does a periodical flush (by default, every two minutes), so even in the case of a lengthy optimization process, you will still see your values. If you need to flush `SummaryWriter` data explicitly, it has the `flush()` method.

The result of running this will be zero output on the console, but you will see a new directory created inside the `runs` directory with a single file. To look at the result, we need to start TensorBoard:

```
Chapter03$ tensorboard --logdir runs
TensorFlow installation not found - running with reduced feature set.
Serving TensorBoard on localhost; to expose to the network, use a proxy or pass
--bind_all
TensorBoard 2.15.1 at http://localhost:6006/ (Press CTRL+C to quit)
```

If you are running TensorBoard on a remote server, you will need to add the `--bind_all` command-line option to make it accessible from other machines. Now you can open `http://localhost:6006` in your browser to see something like this:

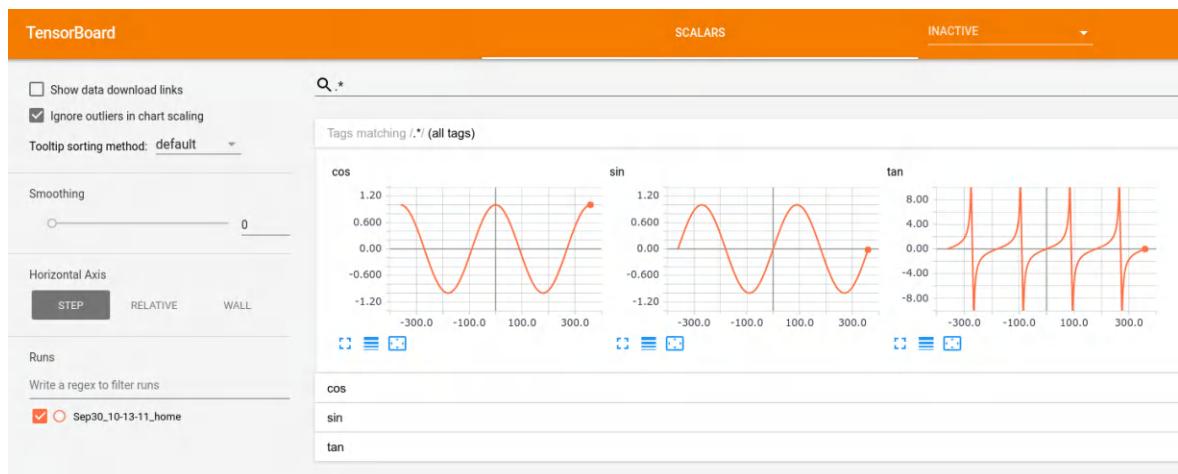


Figure 3.5: Plots produced by the example (for better visualization, refer to <https://packt.link/gbp/9781835882702>)

The graphs are interactive, so you can hover over them with your mouse to see the actual values and select regions to zoom in and look at details. To zoom out, double-click inside the graph. If you run your program several times, then you will see several items in the **Runs** list on the left, which can be enabled and disabled in any combination, allowing you to compare the dynamics of several optimizations. TensorBoard allows you to analyze not only scalar values but also images, audio, text data, and embeddings, and it can even show you the structure of your network. Refer to the documentation of TensorBoard for all those features.

Now, it's time to combine everything you learned in this chapter and look at a real NN optimization problem using PyTorch.

GAN on Atari images

Almost every book about DL uses the MNIST dataset to show you the power of DL, which, over the years, has made this dataset extremely boring, like a fruit fly for genetic researchers. To break this tradition, and add a bit more fun to the book, I've tried to avoid well-beaten paths and illustrate PyTorch using something different. I briefly referred to **generative adversarial networks (GANs)** earlier in the chapter. In this example, we will train a GAN to generate screenshots of various Atari games.

The simplest GAN architecture is this: we have two NNs where the first works as a “cheater” (it is also called the *generator*), and the other as a “detective” (another name is the *discriminator*). Both networks compete with each other — the generator tries to generate fake data, which will be hard for the discriminator to distinguish from your dataset, and the discriminator tries to detect the generated data samples. Over time, both networks improve their skills — the generator produces more and more realistic data samples, and the discriminator invents more sophisticated ways to distinguish the fake items.

Practical usage of GANs includes image quality improvement, realistic image generation, and feature learning. In our example, practical usefulness is almost zero, but it will be a good showcase about everything we learned about PyTorch so far.

So, let's get started. The whole example code is in the file `Chapter03/03_atari_gan.py`. Here, we will look at only the most significant pieces of code, without the import section and constants declaration. The following class is a wrapper around a Gym game:

```
class InputWrapper(gym.ObservationWrapper):
    """
    Preprocessing of input numpy array:
    1. resize image into predefined size
    2. move color channel axis to a first place
    """
    def __init__(self, *args):
        super(InputWrapper, self).__init__(*args)
        old_space = self.observation_space
        assert isinstance(old_space, spaces.Box)
        self.observation_space = spaces.Box(
            self.observation(old_space.low), self.observation(old_space.high),
            dtype=np.float32
        )
```

```
def observation(self, observation: gym.core.ObsType) -> gym.core.ObsType:
    # resize image
    new_obs = cv2.resize(
        observation, (IMAGE_SIZE, IMAGE_SIZE))
    # transform (w, h, c) -> (c, w, h)
    new_obs = np.moveaxis(new_obs, 2, 0)
    return new_obs.astype(np.float32)
```

The preceding class includes several transformations:

- Resize the input image from 210×160 (the standard Atari resolution) to a square size of 64×64
- Move the color plane of the image from the last position to the first, to meet the PyTorch convention of convolution layers that input a tensor with the shape of the channels, height, and width
- Cast the image from bytes to float

Then, we define two `nn.Module` classes: `Discriminator` and `Generator`. The first takes our scaled color image as input and, by applying five layers of convolutions, converts it into a single number passed through a `Sigmoid` nonlinearity. The output from `Sigmoid` is interpreted as the probability that `Discriminator` thinks our input image is from the real dataset.

`Generator` takes as input a vector of random numbers (latent vector) and, by using the “transposed convolution” operation (it is also known as deconvolution), converts this vector into a color image of the original resolution. We will not look at those classes here as they are lengthy and not very relevant to our example; you can find them in the complete example file.

As input, we will use screenshots from several Atari games played simultaneously by a random agent. *Figure 3.6* is an example of what the input data looks like.

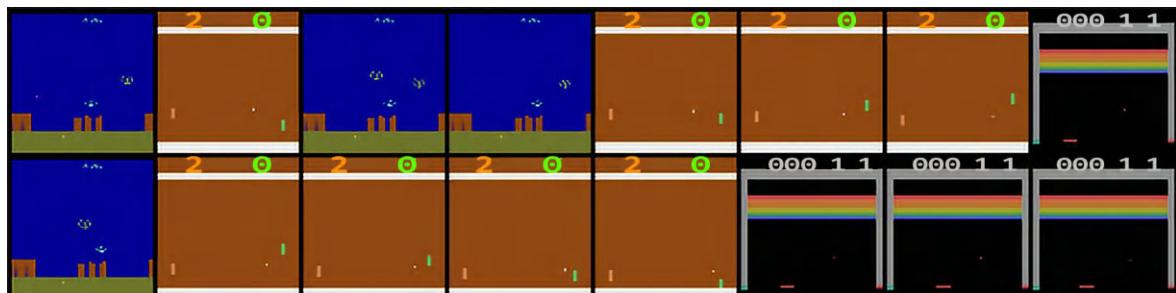


Figure 3.6: Sample screenshots from three Atari games

Images are combined in batches that are generated by the following function:

```
def iterate_batches(envs: tt.List[gym.Env],
                   batch_size: int = BATCH_SIZE) -> tt.Generator[torch.Tensor, None,
                                                 None]:
    batch = [e.reset()[0] for e in envs]
    env_gen = iter(lambda: random.choice(envs), None)

    while True:
        e = next(env_gen)
        action = e.action_space.sample()
        obs, reward, is_done, is_trunc, _ = e.step(action)
        if np.mean(obs) > 0.01:
            batch.append(obs)
        if len(batch) == batch_size:
            batch_np = np.array(batch, dtype=np.float32)
            # Normalising input to [-1..1]
            yield torch.tensor(batch_np * 2.0 / 255.0 - 1.0)
            batch.clear()
        if is_done or is_trunc:
            e.reset()
```

This function infinitely samples the environment from the provided list, issues random actions, and remembers observations in the batch list. When the batch becomes of the required size, we normalize the image, convert it to a tensor, and yield from the generator. The check for the non-zero mean of the observation is required due to a bug in one of the games to prevent the flickering of an image.

Now, let's look at our `main` function, which prepares models and runs the training loop:

```
if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("--dev", default="cpu", help="Device name, default=cpu")
    args = parser.parse_args()

    device = torch.device(args.dev)
    envs = [
        InputWrapper(gym.make(name))
        for name in ('Breakout-v4', 'AirRaid-v4', 'Pong-v4')
    ]
    shape = envs[0].observation_space.shape
```

Here, we process the command-line arguments (which could be only one optional argument, `-dev`, which specifies the device to use for computations) and create our environment pool with a wrapper applied. This environment array will be passed to the `iterate_batches` function later to generate training data.

In the following piece, we create our classes — a summary writer, both networks, a loss function, and two optimizers:

```
net_discr = Discriminator(input_shape=shape).to(device)
net_gener = Generator(output_shape=shape).to(device)

objective = nn.BCELoss()
gen_optimizer = optim.Adam(params=net_gener.parameters(), lr=LEARNING_RATE,
                           betas=(0.5, 0.999))
dis_optimizer = optim.Adam(params=net_discr.parameters(), lr=LEARNING_RATE,
                           betas=(0.5, 0.999))
writer = SummaryWriter()
```

Why do we need two optimizers? It's because that's the way that GANs get trained: to train the discriminator, we need to show it both real and fake data samples with appropriate labels (1 for real and 0 for fake). During this pass, we update only the discriminator's parameters.

After that, we pass both real and fake samples through the discriminator again, but this time, the labels are 1s for all samples and we update only the generator's weights. The second pass teaches the generator how to fool the discriminator and confuse real samples with the generated ones.

We then define arrays, which will be used to accumulate losses, iterator counters, and variables with the true and fake labels. We also store the current timestamp to report the time elapsed after 100 iterations of training:

```
gen_losses = []
dis_losses = []
iter_no = 0

true_labels_v = torch.ones(BATCH_SIZE, device=device)
fake_labels_v = torch.zeros(BATCH_SIZE, device=device)
ts_start = time.time()
```

At the beginning of the following training loop, we generate a random vector and pass it to the Generator network:

```
for batch_v in iterate_batches(envs):
    # fake samples, input is 4D: batch, filters, x, y
    gen_input_v = torch.FloatTensor(BATCH_SIZE, LATENT_VECTOR_SIZE, 1, 1)
    gen_input_v.normal_(0, 1)
    gen_input_v = gen_input_v.to(device)
    batch_v = batch_v.to(device)
    gen_output_v = net_gener(gen_input_v)
```

We then train the discriminator by applying it two times, once to the true data samples in our batch and once to the generated ones:

```
dis_optimizer.zero_grad()
dis_output_true_v = net_discr(batch_v)
dis_output_fake_v = net_discr(gen_output_v.detach())
dis_loss = objective(dis_output_true_v, true_labels_v) + \
           objective(dis_output_fake_v, fake_labels_v)
dis_loss.backward()
dis_optimizer.step()
dis_losses.append(dis_loss.item())
```

In the preceding code, we need to call the `detach()` function on the generator's output to prevent gradients of this training pass from flowing into the generator (`detach()` is a method of tensor, which makes a copy of it without connection to the parent's operation, i.e., detaching the tensor from the parent's graph).

Now it's the generator's training time:

```
gen_optimizer.zero_grad()
dis_output_v = net_discr(gen_output_v)
gen_loss_v = objective(dis_output_v, true_labels_v)
gen_loss_v.backward()
gen_optimizer.step()
gen_losses.append(gen_loss_v.item())
```

We pass the generator's output to the discriminator, but now we don't stop the gradients. Instead, we apply the objective function with True labels. It will push our generator in the direction where the samples that it generates make the discriminator confuse them with the real data.

That was the code related to training, and the next couple of lines report losses and feed image samples to TensorBoard:

```

iter_no += 1
if iter_no % REPORT_EVERY_ITER == 0:
    dt = time.time() - ts_start
    log.info("Iter %d in %.2fs: gen_loss=% .3e, dis_loss=% .3e",
             iter_no, dt, np.mean(gen_losses), np.mean(dis_losses))
    ts_start = time.time()
    writer.add_scalar("gen_loss", np.mean(gen_losses), iter_no)
    writer.add_scalar("dis_loss", np.mean(dis_losses), iter_no)
    gen_losses = []
    dis_losses = []
if iter_no % SAVE_IMAGE_EVERY_ITER == 0:
    img = vutils.make_grid(gen_output_v.data[:64], normalize=True)
    writer.add_image("fake", img, iter_no)
    img = vutils.make_grid(batch_v.data[:64], normalize=True)
    writer.add_image("real", img, iter_no)

```

The training of this example is quite a lengthy process. On a GTX 1080Ti GPU, 100 iterations take about 2.7 seconds. At the beginning, the generated images are completely random noise, but after 10k–20k iterations, the generator becomes more and more proficient at its job and the generated images become more and more similar to the real game screenshots.



It also worth noting the performance improvement in software libraries. In the first and second editions of the book, exactly the same example ran much slower on the same hardware I have. On GTX 1080Ti, 100 iterations took around 40 seconds. Now, with PyTorch 2.2.0 on exactly the same GPU, 100 iterations take 2.7 seconds. So, instead of 3–4 hours, it now takes about 30 minutes to get good generated images.

My experiments gave the following images after 40k–50k of training iterations (about half an hour on a 1080 GPU):



Figure 3.7: Sample images produced by the generator network

As you can see, our network was able to reproduce the Atari screenshots quite well. In the next section, we'll look at how we can simplify our code by using the add-on PyTorch library, Ignite.

PyTorch Ignite

PyTorch is an elegant and flexible library, which makes it a favorite choice for thousands of researchers, DL enthusiasts, industry developers, and others. But flexibility has its own price: too much code to be written to solve your problem. Sometimes, this is very beneficial, such as when implementing some new optimization method or DL trick that hasn't been included in the standard library yet. Then you just implement the formulas using Python and PyTorch magic will do all the gradient and backpropagation machinery for you. Another example is in situations when you have to work on a very low level, fiddling with gradients, optimizer details, and the way your data is transformed by the NN.

However, sometimes you don't need this flexibility, which happens when you work on routine tasks, like the simple supervised training of an image classifier. For such tasks, standard PyTorch might be at too low a level when you need to deal with the same code over and over again. The following is a non-exhaustive list of topics that are an essential part of any DL training procedure, but require some code to be written:

- Data preparation and transformation, and the generation of batches
- Calculation of training metrics, like loss values, accuracy, and F1-scores
- Periodical testing of the model being trained on the test and validation datasets
- Model checkpointing after some number of iterations or when a new best metric is achieved
- Sending metrics into a monitoring tool like TensorBoard
- Hyperparameters change over time, like a learning rate decrease/increase schedule
- Writing training progress messages on the console

They are all doable using only PyTorch, of course, but it might require you to write a significant amount of code. As those tasks occur in any DL project, it quickly becomes cumbersome to write the same code over and over again. The normal approach to solving the issue is to write the functionality once, wrap it into a library, and reuse it later. If the library is open source and of good quality (easy to use, provides a good level of flexibility, written properly, and so on), it will become popular as more and more people use it in their projects. This process is not DL-specific; it happens everywhere in the software industry.

There are several libraries for PyTorch that simplify the solving of common tasks: `ptlearn`, `fastai`, `ignite`, and some others. The current list of "PyTorch ecosystem projects" can be found here: <https://pytorch.org/ecosystem>.

It might be appealing to start using those high-level libraries from the beginning, as they allow you to solve common problems with just a couple of lines of code, but there is some danger here.

If you only know how to use high-level libraries without understanding low-level details, you might get stuck on problems that can't be solved solely by standard methods. In the very dynamic field of ML, this happens very often.

The main focus of this book is to ensure that you understand RL methods, their implementation, and their applicability, so we will use an incremental approach. In the beginning, we will implement methods using only PyTorch code, but with more progress, examples will be implemented using high-level libraries. For RL, this will be the small library written by me: PTAN (<https://github.com/Shmuma/ptan/>), and it will be introduced in *Chapter 7*.

To reduce the amount of DL boilerplate code, we will use a library called PyTorch Ignite: <https://pytorch-ignite.ai>. In this section, a small overview of Ignite will be given, then we will check the Atari GAN example once it has been rewritten using Ignite.

Ignite concepts

At a high level, Ignite simplifies the writing of the training loop in PyTorch DL. Earlier in this chapter (in the *Loss functions and optimizers* section, you saw that the minimal training loop consists of:

- Sampling a batch of training data
- Applying an NN to this batch to calculate the loss function—the single value we want to minimize
- Running backpropagation of the loss to get gradients on the network's parameters in respect to the loss
- Asking the optimizer to apply the gradients to the network
- Repeating until we are happy or bored of waiting

The central piece of Ignite is the `Engine` class, which loops over the data source, applying the processing function to the data batch. In addition to that, Ignite offers the ability to provide functions to be called at specific conditions of the training loop. Those conditions are called `Events` and could be at the:

- Beginning/end of the whole training process
- Beginning/end of a training epoch (iteration over the data)
- Beginning/end of a single batch processing

In addition to that, custom events exist and allow you to specify your function to be called every N events, for example, if you want to do some calculations every 100 batches or every second epoch.

A very simplistic example of Ignite in action is shown in the following code block:

```
from ignite.engine import Engine, Events

def training(engine, batch):
```

```
optimizer.zero_grad()
x, y = prepare_batch()
y_out = model(x)
loss = loss_fn(y_out, y)
loss.backward()
optimizer.step()
return loss.item()

engine = Engine(training)
engine.run(data)
```

This code is not runnable as it misses lots of details, like the data source, model, and optimizer creation, but it shows the basic idea of Ignite usage. The main benefit of Ignite is in the ability it provides to extend the training loop with existing functionality. You want the loss value to be smoothed and written in TensorBoard every 100 batches? No problem! Add two lines and it will be done. You want to run model validation every 10 epochs? Okay, write a function to run a test and attach it to the `Engine` instance, and it will be called.

A description of the full Ignite functionality is beyond the scope of the book, but you can read the documentation on the official website: <https://pytorch-ignite.ai>.

GAN training on Atari using Ignite

To give you an illustration of Ignite, let's change the example of GAN training on Atari images. The full example code is available in `Chapter03/04_atari_gan_ignite.py`; here, I will just show code that differs from the previous section.

First, we import several Ignite classes:

```
from ignite.engine import Engine, Events
from ignite.handlers import Timer
from ignite.metrics import RunningAverage
from ignite.contrib.handlers import tensorboard_logger as tb_logger
```

The `Engine` and `Events` classes have already been outlined. The package `ignite.metrics` contains classes related to working with the performance metrics of the training process, such as confusion matrices, precision, and recall. In our example, we will use the class `RunningAverage`, which provides a way to smooth time series values. In the previous example, we did this by calling `np.mean()` on an array of losses, but `RunningAverage` provides a more convenient (and mathematically more correct) way of doing this. In addition, we import the TensorBoard logger from the Ignite `contrib` package (the functionality of which is contributed by others). We'll also use the `Timer` handler, which provides a simple way to calculate time elapsed between certain events.

As a next step, we need to define our processing function:

```
def process_batch(trainer, batch):
    gen_input_v = torch.FloatTensor(BATCH_SIZE, LATENT_VECTOR_SIZE, 1, 1)
    gen_input_v.normal_(0, 1)
    gen_input_v = gen_input_v.to(device)
    batch_v = batch.to(device)
    gen_output_v = net_gener(gen_input_v)

    # train discriminator
    dis_optimizer.zero_grad()
    dis_output_true_v = net_discr(batch_v)
    dis_output_fake_v = net_discr(gen_output_v.detach())
    dis_loss = objective(dis_output_true_v, true_labels_v) + \
               objective(dis_output_fake_v, fake_labels_v)
    dis_loss.backward()
    dis_optimizer.step()

    # train generator
    gen_optimizer.zero_grad()
    dis_output_v = net_discr(gen_output_v)
    gen_loss = objective(dis_output_v, true_labels_v)
    gen_loss.backward()
    gen_optimizer.step()

    if trainer.state.iteration % SAVE_IMAGE_EVERY_ITER == 0:
        fake_img = vutils.make_grid(gen_output_v.data[:64], normalize=True)
        trainer.tb.writer.add_image("fake", fake_img, trainer.state.iteration)
        real_img = vutils.make_grid(batch_v.data[:64], normalize=True)
        trainer.tb.writer.add_image("real", real_img, trainer.state.iteration)
        trainer.tb.writer.flush()
    return dis_loss.item(), gen_loss.item()
```

This function takes the data batch and does an update of both the discriminator and generator models on this batch. This function can return any data to be tracked during the training process; in our case, it will be two loss values for both models. In this function, we can also save images to be displayed in TensorBoard.

After this is done, all we need to do is create an Engine instance, attach the required handlers, and run the training process:

```
engine = Engine(process_batch)
tb = tb_logger.TensorboardLogger(log_dir=None)
engine.tb = tb
RunningAverage(output_transform=lambda out: out[1]).\
    attach(engine, "avg_loss_gen")
```

```
RunningAverage(output_transform=lambda out: out[0]).\
    attach(engine, "avg_loss_gen")

handler = tb_logger.OutputHandler(tag="train", metric_names=['avg_loss_gen',
    'avg_loss_dis'])
tb.attach(engine, log_handler=handler, event_name=Events.ITERATION_COMPLETED)

timer = Timer()
timer.attach(engine)
```

In the preceding code, we create our engine, passing our processing function and attaching two `RunningAverage` transformations for our two loss values. Being attached, every `RunningAverage` produces a so-called “metric”—a derived value kept around during the training process. The names of our smoothed metrics are `avg_loss_gen` for smoothed loss from the generator, and `avg_loss_dis` for smoothed loss from the discriminator. Those two values will be written in TensorBoard after every iteration.

We also attach the timer, which, being created without any constructor arguments, acts as a simple manually-controlled timer (we call its `reset()` method manually), but can work in a more flexible way with different configuration options.

The last piece of code attaches another event handler, which will be our function, and is called by the `Engine` on every iteration completion:

```
@engine.on(Events.ITERATION_COMPLETED)
def log_losses(trainer):
    if trainer.state.iteration % REPORT_EVERY_ITER == 0:
        log.info("%d in %.2fs: gen_loss=%f, dis_loss=%f",
                 trainer.state.iteration, timer.value(),
                 trainer.state.metrics['avg_loss_gen'],
                 trainer.state.metrics['avg_loss_dis'])
        timer.reset()

engine.run(data=iterate_batches(envs))
```

It will write a log line with an iteration index, time taken and values of smoothed metrics. The final line starts our engine, passing the already defined function as the data source (the `iterate_batches` function is a generator, returning the normal iterator over batches, so, it will be perfectly fine to pass its output as a data argument).

And that's it. If you run the `Chapter03/04_atari_gan_ignite.py` example, it will work the same way as our previous example, which might not be very impressive for such a small example, but in real projects, Ignite usage normally pays off by making your code cleaner and more extensible.

Summary

In this chapter, you saw a quick overview of PyTorch's functionality and features. We talked about basic fundamental pieces, such as tensors and gradients, and you saw how an NN can be made from the basic building blocks, before learning how to implement those blocks yourself.

We discussed loss functions and optimizers, as well as the monitoring of training dynamics. Finally, you were introduced to PyTorch Ignite, a library used to provide a higher-level interface for training loops. The goal of the chapter was to give a very quick introduction to PyTorch, which will be used later in the book.

In the next chapter, we are ready to start dealing with the main subject of this book: RL methods.

4

The Cross-Entropy Method

In the last chapter, you learned about PyTorch. In this chapter, we will wrap up *Part 1* of this book and you will become familiar with one of the **reinforcement learning (RL)** methods: *cross-entropy*.

Despite the fact that it is much less famous than other tools in the RL practitioner's toolbox, such as **deep Q-network (DQN)** or **advantage actor-critic (A2C)**, the cross-entropy method has its own strengths. Firstly, the cross-entropy method is really simple, which makes it an easy method to follow. For example, its implementation on PyTorch is less than 100 lines of code.

Secondly, the method has good convergence. In simple environments that don't require you to learn complex, multistep policies and that have short episodes with frequent rewards, the cross-entropy method usually works very well. Of course, lots of practical problems don't fall into this category, but sometimes they do. In such cases, the cross-entropy method (on its own or as part of a larger system) can be the perfect fit.

In this chapter, we will cover:

- The practical side of the cross-entropy method
- How the cross-entropy method works in two environments in Gym (the familiar CartPole and the grid world of FrozenLake)
- The theoretical background of the cross-entropy method. This section is optional and requires a bit of knowledge of probability and statistics, but if you want to understand why the method works, then you can delve into it.

The taxonomy of RL methods

The cross-entropy method falls into the **model-free**, **policy-based**, and **on-policy** categories of methods. These notions are new, so let's spend some time exploring them.

All the methods in RL can be classified into various groups:

- Model-free or model-based
- Value-based or policy-based
- On-policy or off-policy

There are other ways that you can taxonomize RL methods, but, for now, we are interested in the above three. Let's define them, as the specifics of your problem can influence your choice of a particular method.

The term **model-free** means that the method doesn't build a model of the environment or reward; it just directly connects observations to actions (or values that are related to actions). In other words, the agent takes current observations and does some computations on them, and the result is the action that it should take. In contrast, **model-based** methods try to predict what the next observation and/or reward will be. Based on this prediction, the agent tries to choose the best possible action to take, very often making such predictions multiple times to look more and more steps into the future.

Both classes of methods have strong and weak sides, but usually pure model-based methods are used in deterministic environments, such as board games with strict rules. On the other hand, model-free methods are usually easier to train because it's hard to build good models of complex environments with rich observations. All of the methods described in this book are from the model-free category, as those methods have been the most active area of research for the past few years. Only recently have researchers started to mix the benefits from both worlds (for example, in *Chapter 20*, we'll take a look at the AlphaGo Zero and MuZero methods, which use a model-based approach to board games and Atari).

Looking at this from another angle, **policy-based** methods directly approximate the policy of the agent, that is, what actions the agent should carry out at every step. The policy is usually represented by a probability distribution over the available actions. Alternatively, the method could be **value-based**. In this case, instead of the probability of actions, the agent calculates the value of every possible action and chooses the action with the best value. These families of methods are equally popular, and we will discuss value-based methods in the next part of the book. Policy methods will be the topic of *Part 3*.

The third important classification of methods is **on-policy** versus **off-policy**. We will discuss this distinction in more depth in *Parts 2* and *3* of the book, but, for now, it is enough to explain off-policy as the ability of the method to learn from historical data (obtained by a previous version of the agent, recorded by human demonstration, or just seen by the same agent several episodes ago). On the other hand, on-policy methods require fresh data for training, generated from the policy we're currently updating. They cannot be trained on old historical data because the result of the training will be wrong. This makes such methods much less data-efficient (you need much more communication with the environment), but in some cases, this is not a problem (for example, if our environment is very lightweight and fast, so we can quickly interact with it).

So, our cross-entropy method is model-free, policy-based, and on-policy, which means the following:

- It doesn't build a model of the environment; it just says to the agent what to do at every step
- It approximates the policy of the agent
- It requires fresh data obtained from the environment

The cross-entropy method in practice

The explanation of the cross-entropy method can be split into two unequal parts: practical and theoretical. The practical part is intuitive in nature, while the theoretical explanation of *why* the cross-entropy method works and what happens, is more sophisticated.

You may remember that the central and trickiest thing in RL is the agent, which tries to accumulate as much total reward as possible by communicating with the environment. In practice, we follow a common **machine learning (ML)** approach and replace all of the complications of the agent with some kind of nonlinear trainable function, which maps the agent's input (observations from the environment) to some output. The details of the output that this function produces may depend on a particular method or a family of methods (such as value-based or policy-based methods), as described in the previous section. As our cross-entropy method is policy-based, our nonlinear function (**neural network (NN)**) produces the *policy*, which basically says for every observation which action the agent should take. In research papers, policy is denoted as $\pi(a|s)$, where a are actions and s is the current state.

This is illustrated in the following diagram:

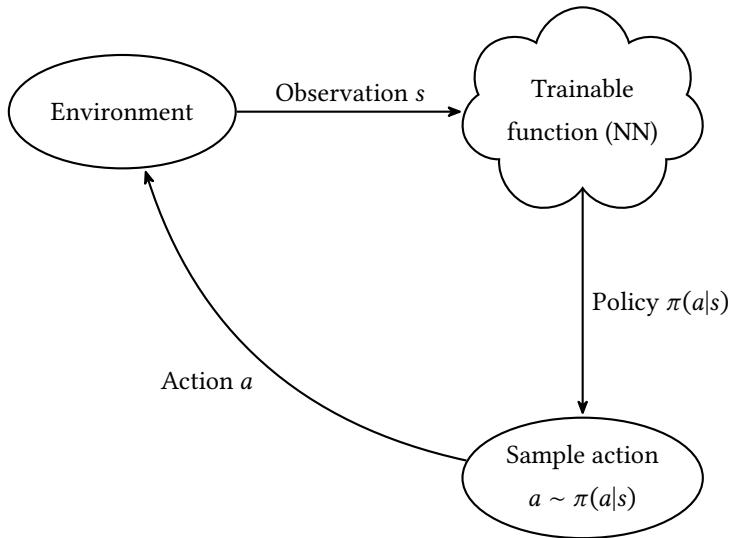


Figure 4.1: A high-level approach to policy-based RL

In practice, the policy is usually represented as a probability distribution over actions, which makes it very similar to a classification problem, with the number of classes being equal to the number of actions we can carry out.

This abstraction makes our agent very simple: it needs to pass an observation from the environment to the NN, get a probability distribution over actions, and perform random sampling using the probability distribution to get an action to carry out. This random sampling adds randomness to our agent, which is a good thing because at the beginning of the training, when our weights are random, the agent behaves randomly. As soon as the agent gets an action to issue, it fires the action to the environment and obtains the next observation and reward for the last action. Then the loop continues, as shown in *Figure 4.1*.

During the agent's lifetime, its experience is presented as episodes. Every episode is a sequence of observations that the agent has got from the environment, actions it has issued, and rewards for these actions. Imagine that our agent has played several such episodes. For every episode, we can calculate the total reward that the agent has claimed. It can be discounted or not discounted; for simplicity, let's assume a discount factor of $\gamma = 1$, which just means an undiscounted sum of all local rewards for every episode.

This total reward shows how good this episode was for the agent. It is illustrated in *Figure 4.2*, which contains four episodes (note that different episodes have different values for o_i , a_i , and r_i):

Episode 1:	$[o_1, a_1, r_1]$	$[o_2, a_2, r_2]$	$[o_3, a_3, r_3]$	$[o_4, a_4, r_4]$	$R = r_1 + r_2 + r_3 + r_4$
Episode 2:	$[o_1, a_1, r_1]$	$[o_2, a_2, r_2]$	$[o_3, a_3, r_3]$		$R = r_1 + r_2 + r_3$
Episode 3:	$[o_1, a_1, r_1]$	$[o_2, a_2, r_2]$			$R = r_1 + r_2$
Episode 4:	$[o_1, a_1, r_1]$	$[o_2, a_2, r_2]$	$[o_3, a_3, r_3]$		$R = r_1 + r_2 + r_3$

Figure 4.2: Example episodes with their observations, actions, and rewards

Every cell represents the agent’s step in the episode. Due to randomness in the environment and the way that the agent selects actions to take, some episodes will be better than others. The core of the cross-entropy method is to throw away bad episodes and train on better ones. So, the steps of the method are as follows:

1. Play N episodes using our current model and environment.
2. Calculate the total reward for every episode and decide on a reward boundary. Usually, we use a percentile of all rewards, such as the 50th or 70th.
3. Throw away all episodes with a reward below the boundary.
4. Train on the remaining “elite” episodes (with rewards higher than the boundary) using observations as the input and issued actions as the desired output.
5. Repeat from step 1 until we become satisfied with the result.

So, that’s the cross-entropy method’s description. With the preceding procedure, our NN learns how to repeat actions, which leads to a larger reward, constantly moving the boundary higher and higher. Despite the simplicity of this method, it works well in basic environments, it’s easy to implement, and it’s quite robust against changing hyperparameters, which makes it an ideal baseline method to try. Let’s now apply it to our CartPole environment.

The cross-entropy method on CartPole

The whole code for this example is in `Chapter04/01_cartpole.py`. Here, I will show only the most important parts. Our model’s core is a one-hidden-layer NN, with **rectified linear unit (ReLU)** and 128 hidden neurons (which is absolutely arbitrary; you can try to increase or decrease this constant – we’ve left this as an exercise for you).

Other hyperparameters are also set almost randomly and aren't tuned, as the method is robust and converges very quickly. We define constants at the top of the file:

```
import typing as tt
import torch
import torch.nn as nn
import torch.optim as optim

HIDDEN_SIZE = 128
BATCH_SIZE = 16
PERCENTILE = 70
```

As shown in the preceding code, the constants include the count of neurons in the hidden layer, the count of episodes we play on every iteration (16), and the percentile of each episode's total rewards that we use for elite episode filtering. We will take the 70th percentile, which means that we will keep the top 30% of episodes sorted by reward.

There is nothing special about our NN; it takes a single observation from the environment as an input vector and outputs a number for every action we can perform:

```
class Net(nn.Module):
    def __init__(self, obs_size: int, hidden_size: int, n_actions: int):
        super(Net, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(obs_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, n_actions)
        )

    def forward(self, x: torch.Tensor):
        return self.net(x)
```

The output from the NN is a probability distribution over actions, so a straightforward way to proceed would be to include softmax nonlinearity after the last layer. However, in the code, we don't apply softmax to increase the numerical stability of the training process. Rather than calculating softmax (which uses exponentiation) and then calculating cross-entropy loss (which uses a logarithm of probabilities), we will use the `nn.CrossEntropyLoss` PyTorch class later, which combines softmax and cross-entropy into a single, more numerically stable expression. `CrossEntropyLoss` requires raw, unnormalized values from the NN (also called logits). The downside of this is that we need to remember to apply softmax every time we need to get probabilities from our NN's output.

Next, we will define two helper dataclasses:

```
@dataclass
class EpisodeStep:
    observation: np.ndarray
    action: int

@dataclass
class Episode:
    reward: float
    steps: tt.List[EpisodeStep]
```

The purpose of these dataclasses is as follows:

- `EpisodeStep`: This will be used to represent one single step that our agent made in the episode, and it stores the observation from the environment and what action the agent performed. We will use episode steps from elite episodes as training data.
- `Episode`: This is a single episode stored as a total undiscounted reward and a collection of `EpisodeStep`.

Let's look at a function that generates batches with episodes:

```
def iterate_batches(env: gym.Env, net: Net, batch_size: int) ->
    tt.Generator[tt.List[Episode], None, None]:
    batch = []
    episode_reward = 0.0
    episode_steps = []
    obs, _ = env.reset()
    sm = nn.Softmax(dim=1)
```

The preceding function accepts the environment (the `Env` class instance from the Gym library), our NN, and the count of episodes it should generate on every iteration. The `batch` variable will be used to accumulate our batch (which is a list of `Episode` instances). We also declare a reward counter for the current episode and its list of steps (the `EpisodeStep` objects). Then we reset our environment to obtain the first observation and create a softmax layer, which will be used to convert the NN's output to a probability distribution of actions. That's our preparations complete, so we are ready to start the environment loop:

```
while True:
    obs_v = torch.tensor(obs, dtype=torch.float32)
    act_probs_v = sm(net(obs_v.unsqueeze(0)))
    act_probs = act_probs_v.data.numpy()[0]
```

At every iteration, we convert our current observation to a PyTorch tensor and pass it to the NN to obtain action probabilities. There are several things to note here:

- All `nn.Module` instances in PyTorch expect a batch of data items, and the same is true for our NN, so we convert our observation (which is a vector of four numbers in CartPole) into a tensor of size 1×4 (to achieve this, we call the `unsqueeze(0)` function on our tensor, which adds an extra dimension at the zero position of the shape).
- As we haven't used nonlinearity at the output of our NN, it outputs raw action scores, which we need to feed through the softmax function.
- Both our NN and the softmax layer return tensors that track gradients, so we need to unpack this by accessing the `tensor.data` field and then converting the tensor into a NumPy array. This array will have the same two-dimensional structure as the input, with the batch dimension on axis 0, so we need to get the first batch element to obtain a one-dimensional vector of action probabilities.

Now that we have the probability distribution of actions, we can use it to obtain the actual action for the current step:

```
action = np.random.choice(len(act_probs), p=act_probs)
next_obs, reward, is_done, is_trunc, _ = env.step(action)
```

Here, we sample the distribution using NumPy's function `random.choice()`. After this, we will pass this action to the environment to get our next observation, our reward, the indication of the episode ending, and the truncation flag. The last value returned by the `step()` function is extra information from the environment and is discarded.

The reward is added to the current episode's total reward, and our list of episode steps is also extended with an `(observation, action)` pair:

```
episode_reward += float(reward)
step = EpisodeStep(observation=obs, action=action)
episode_steps.append(step)
```

Note that we save the observation that was used to choose the action, but not the observation returned by the environment as a result of the action. These are the tiny, but important, details that you need to keep in mind.

The continuation of the code handles the situation when the current episode is over (in the case of CartPole, the episode ends when the stick has fallen down, despite our efforts, or when the time limit of the environment has been reached):

```
if is_done or is_trunc:  
    e = Episode(reward=episode_reward, steps=episode_steps)  
    batch.append(e)  
    episode_reward = 0.0  
    episode_steps = []  
    next_obs, _ = env.reset()  
if len(batch) == batch_size:  
    yield batch  
    batch = []
```

We append the finalized episode to the batch, saving the total reward (as the episode has been completed and we have accumulated all the rewards) and steps we have taken. Then we reset our total reward accumulator and clean the list of steps. After that, we reset our environment to start over.

If our batch has reached the desired count of episodes, we return it to the caller for processing using `yield`. Our function is a generator, so every time the `yield` operator is executed, the control is transferred to the outer iteration loop and then continues after the `yield` line. If you are not familiar with Python's generator functions, refer to the Python documentation: <https://wiki.python.org/moin/Generators>. After processing, we will clean up the batch.

The last, but very important, step in our loop is to assign an observation obtained from the environment to our current observation variable:

```
obs = next_obs
```

After that, everything repeats infinitely – we pass the observation to the NN, sample the action to perform, ask the environment to process the action, and remember the result of this processing.

One very important fact to understand in this function's logic is that the training of our NN and the generation of our episodes are performed *at the same time*. They are not completely in parallel, but every time our loop accumulates enough episodes (16), it passes control to this function caller, which is supposed to train the NN using gradient descent. So, when `yield` is returned, the NN will have different, slightly better (we hope) behavior. As you should remember from the beginning of the chapter, the cross-entropy method is from the on-policy class, so using fresh training data is important for the method to work properly.

Since training and data gathering happen in the same thread, proper synchronization isn't necessary. However, you should be aware of the frequent jumps between training the NN and using it.

Okay; now we need to define yet another function, and then we will be ready to switch to the training loop:

```
def filter_batch(batch: tt.List[Episode], percentile: float) -> \
    tt.Tuple[torch.FloatTensor, torch.LongTensor, float, float]:
    rewards = list(map(lambda s: s.reward, batch))
    reward_bound = float(np.percentile(rewards, percentile))
    reward_mean = float(np.mean(rewards))
```

This function is at the core of the cross-entropy method – from the given batch of episodes and percentile value, it calculates a boundary reward, which is used to filter elite episodes to train on. To obtain the boundary reward, we will use NumPy’s percentile function, which, from the list of values and the desired percentile, calculates the percentile’s value. Then, we will calculate the mean reward, which is used only for monitoring.

Next, we will filter off our episodes:

```
train_obs: tt.List[np.ndarray] = []
train_act: tt.List[int] = []
for episode in batch:
    if episode.reward < reward_bound:
        continue
    train_obs.extend(map(lambda step: step.observation, episode.steps))
    train_act.extend(map(lambda step: step.action, episode.steps))
```

For every episode in the batch, we will check that the episode has a higher total reward than our boundary and, if it has, we will populate lists of observations and actions that we will train on.

The following is the final step of the function:

```
train_obs_v = torch.FloatTensor(np.vstack(train_obs))
train_act_v = torch.LongTensor(train_act)
return train_obs_v, train_act_v, reward_bound, reward_mean
```

Here, we will convert our observations and actions from elite episodes into tensors, and return a tuple of four: observations, actions, the boundary of reward, and the mean reward. The last two values are not used in the training; we will write them into TensorBoard to check the performance of our agent.

Now, the final chunk of code that glues everything together, and mostly consists of the training loop, is as follows:

```

if __name__ == "__main__":
    env = gym.make("CartPole-v1")
    assert env.observation_space.shape is not None
    obs_size = env.observation_space.shape[0]
    assert isinstance(env.action_space, gym.spaces.Discrete)
    n_actions = int(env.action_space.n)

    net = Net(obs_size, HIDDEN_SIZE, n_actions)
    print(net)
    objective = nn.CrossEntropyLoss()
    optimizer = optim.Adam(params=net.parameters(), lr=0.01)
    writer = SummaryWriter(comment="-cartpole")

```

In the beginning, we create all the required objects: the environment, our NN, the objective function, the optimizer, and the summary writer for TensorBoard.

In the training loop, we iterate our batches (a list of `Episode` objects):

```

for iter_no, batch in enumerate(iterate_batches(env, net, BATCH_SIZE)):
    obs_v, acts_v, reward_b, reward_m = filter_batch(batch, PERCENTILE)
    optimizer.zero_grad()
    action_scores_v = net(obs_v)
    loss_v = objective(action_scores_v, acts_v)
    loss_v.backward()
    optimizer.step()

```

We perform filtering of the elite episodes using the `filter_batch` function. The result is tensors of observations and taken actions, the reward boundary used for filtering, and the mean reward. After that, we zero the gradients of our NN and pass observations to the NN, obtaining its action scores. These scores are passed to the objective function, which will calculate the cross-entropy between the NN output and the actions that the agent took. The idea of this is to reinforce our NN to carry out those elite actions that have led to good rewards. Then, we calculate gradients on the loss and ask the optimizer to adjust our NN.

The rest of the loop is mostly the monitoring of progress:

```

print("%d: loss=%.3f, reward_mean=%.1f, rw_bound=%.1f" % (
    iter_no, loss_v.item(), reward_m, reward_b))
writer.add_scalar("loss", loss_v.item(), iter_no)
writer.add_scalar("reward_bound", reward_b, iter_no)
writer.add_scalar("reward_mean", reward_m, iter_no)

```

On the console, we show the iteration number, the loss, the mean reward of the batch, and the reward boundary.

We also write the same values to TensorBoard to get a nice chart of the agent's learning performance.

The last check in the loop is the comparison of the mean rewards of our batch episodes:

```

if reward_m > 475:
    print("Solved!")
    break
writer.close()

```

When the mean reward becomes greater than 475, we stop our training. Why 475? In Gym, the CartPole-v1 environment is considered to be solved when the mean reward for the last 100 episodes is greater than 475. However, our method converges so quickly that 100 episodes are usually what we need. The properly trained agent can balance the stick for an infinitely long period of time (obtaining any amount of score), but the length of an episode in CartPole-v1 is limited to 500 steps (if you look in https://github.com/Farama-Foundation/Gymnasium/blob/main/gymnasium/envs/__init__.py where all the environments are registered, v1 of Cartpole has `max_episode_steps=500`). With all this in mind, we will stop training after the mean reward in the batch is greater than 475, which is a good indication that our agent knows how to balance the stick like a pro.

That's it. So, let's start our first RL training!

```

Chapter04$ ./01_cartpole.py
Net(
  (net): Sequential(
    (0): Linear(in_features=4, out_features=128, bias=True)
    (1): ReLU()
    (2): Linear(in_features=128, out_features=2, bias=True)
  )
)
0: loss=0.683, reward_mean=25.2, rw_bound=24.0
1: loss=0.669, reward_mean=34.3, rw_bound=39.0
2: loss=0.648, reward_mean=37.6, rw_bound=40.0
3: loss=0.647, reward_mean=41.9, rw_bound=43.0
4: loss=0.634, reward_mean=41.2, rw_bound=50.0
...
38: loss=0.537, reward_mean=431.8, rw_bound=500.0
39: loss=0.529, reward_mean=450.1, rw_bound=500.0
40: loss=0.533, reward_mean=456.4, rw_bound=500.0
41: loss=0.526, reward_mean=422.0, rw_bound=500.0
42: loss=0.531, reward_mean=436.8, rw_bound=500.0
43: loss=0.526, reward_mean=475.5, rw_bound=500.0
Solved!

```

It usually doesn't take the agent more than 50 batches to solve the problem. My experiments show something from 30 to 60 episodes, which is a really good learning performance (remember, we need to play only 16 episodes for every batch). TensorBoard shows that our agent is consistently making progress, pushing the upper boundary at almost every batch (there are some periods of rolling down, but most of the time it improves):

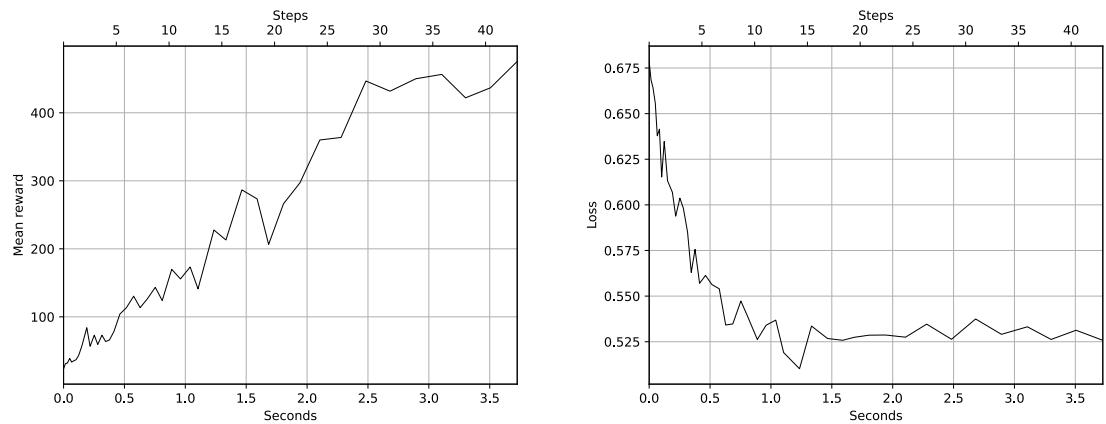


Figure 4.3: Mean reward (left) and loss (right) during the training

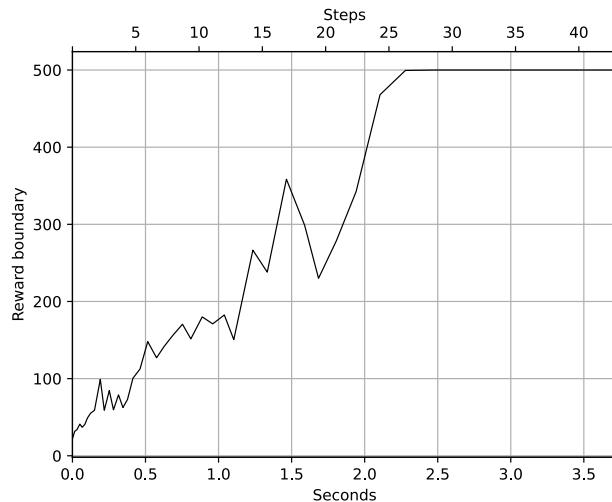


Figure 4.4: The reward boundary during the training

To monitor the training process, you can tweak the environment creation by setting the rendering mode in the CartPole environment and adding a RecordVideo wrapper:

```
env = gym.make("CartPole-v1", render_mode="rgb_array")
env = gym.wrappers.RecordVideo(env, video_folder="video")
```

During the training, it will create a video directory with a bunch of MP4 movies inside, allowing you to compare the progress of agent training:

```
Chapter04$ ./01_cartpole.py
Net(
    (net): Sequential(
        (0): Linear(in_features=4, out_features=128, bias=True)
        (1): ReLU()
        (2): Linear(in_features=128, out_features=2, bias=True)
    )
)
Moviepy - Building video Chapter04/video/rl-video-episode-0.mp4.
Moviepy - Writing video Chapter04/video/rl-video-episode-0.mp4
Moviepy - Done !
Moviepy - video ready Chapter04/video/rl-video-episode-0.mp4
Moviepy - Building video Chapter04/video/rl-video-episode-1.mp4.
Moviepy - Writing video Chapter04/video/rl-video-episode-1.mp4
...
...
```

The MP4 movies might look like the following:



Figure 4.5: CartPole episode movie

Let's now pause for a bit and think about what's just happened. Our NN has learned how to play the environment purely from observations and rewards, without any interpretation of observed values. The environment could easily not be a cart with a stick; it could be, say, a warehouse model with product quantities as an observation and money earned as the reward. Our implementation doesn't depend on environment-related details. This is the beauty of the RL model and, in the next section, we will look at how exactly the same method can be applied to a different environment from the Gym collection.

The cross-entropy method on FrozenLake

The next environment that we will try to solve using the cross-entropy method is FrozenLake. Its world is from the so-called grid world category, when your agent lives in a grid of size 4×4 and can move in four directions: up, down, left, and right. The agent always starts at the top left, and its goal is to reach the bottom-right cell of the grid. There are holes in the fixed cells of the grid and if you get into those holes, the episode ends and your reward is zero. If the agent reaches the destination cell, then it obtains a reward of 1.0 and the episode ends.

To make life more complicated, the world is slippery (it's a frozen lake after all), so the agent's actions do not always turn out as expected – there is a 33% chance that it will slip to the right or to the left. If you want the agent to move left, for example, there is a 33% probability that it will, indeed, move left, a 33% chance that it will end up in the cell above, and a 33% chance that it will end up in the cell below. As you will see at the end of the section, this makes progress difficult.



Figure 4.6: The FrozenLake environment rendered in human mode

Let's look at how this environment is represented in the Gym API:

```
>>> e = gym.make("FrozenLake-v1", render_mode="ansi")
>>> e.observation_space
Discrete(16)
>>> e.action_space
Discrete(4)
>>> e.reset()
(0, {'prob': 1})
>>> print(e.render())

SFFF
FHFH
FFFF
HFFG
```

Our observation space is discrete, which means that it's just a number from 0 to 15 inclusive. Obviously, this number is our current position in the grid. The action space is also discrete, but it can be from zero to three. Although the action space is similar to CartPole, the observation space is represented in a different way. To minimize the required changes in our implementation, we can apply the traditional one-hot encoding of discrete inputs, which means that the input to our network will have 16 float numbers and zero everywhere, except the index that we will encode (representing our current position on the grid).

As this transformation only affects the observation of the environment, it could be implemented as an `ObservationWrapper`, as we discussed in *Chapter 2*. Let's call it `DiscreteOneHotWrapper`:

```
class DiscreteOneHotWrapper(gym.ObservationWrapper):
    def __init__(self, env: gym.Env):
        super(DiscreteOneHotWrapper, self).__init__(env)
        assert isinstance(env.observation_space, gym.spaces.Discrete)
        shape = (env.observation_space.n, )
        self.observation_space = gym.spaces.Box(0.0, 1.0, shape, dtype=np.float32)

    def observation(self, observation):
        res = np.copy(self.observation_space.low)
        res[observation] = 1.0
        return res
```

With that wrapper applied to the environment, both the observation space and action space are 100% compatible with our CartPole solution (source code `Chapter04/02_frozenlake_naive.py`). However, by launching it, we can see that our training process doesn't improve the score over time:

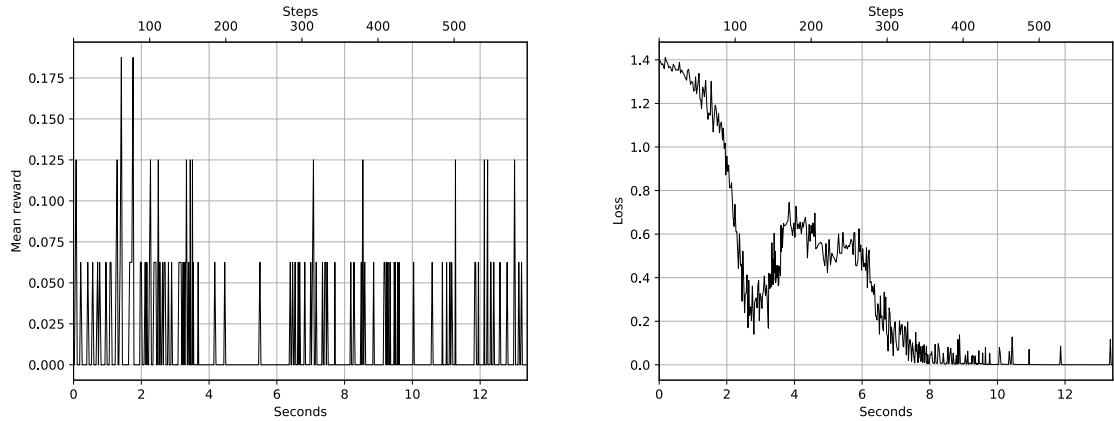


Figure 4.7: Mean reward (left) and loss (right) on the FrozenLake environment

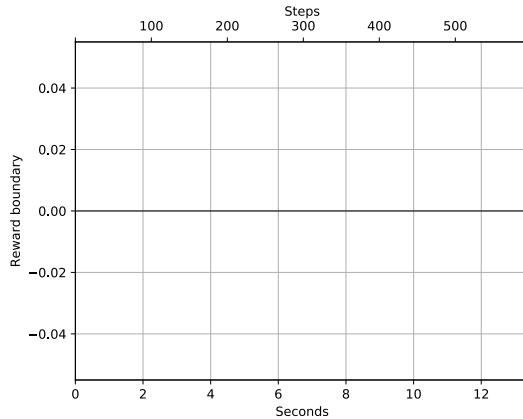


Figure 4.8: The reward boundary during the training (boring 0.0 all the time)

To understand what's going on, we need to look deeper at the reward structure of both environments. In CartPole, every step of the environment gives us the reward 1.0, until the moment that the pole falls. So, the longer our agent balanced the pole, the more reward it obtained. Due to randomness in our agent's behavior, different episodes were of different lengths, which gave us a pretty normal distribution of the episodes' rewards. After choosing a reward boundary, we rejected less successful episodes and learned how to repeat better ones (by training on successful episodes' data).

This is shown in the following diagram:

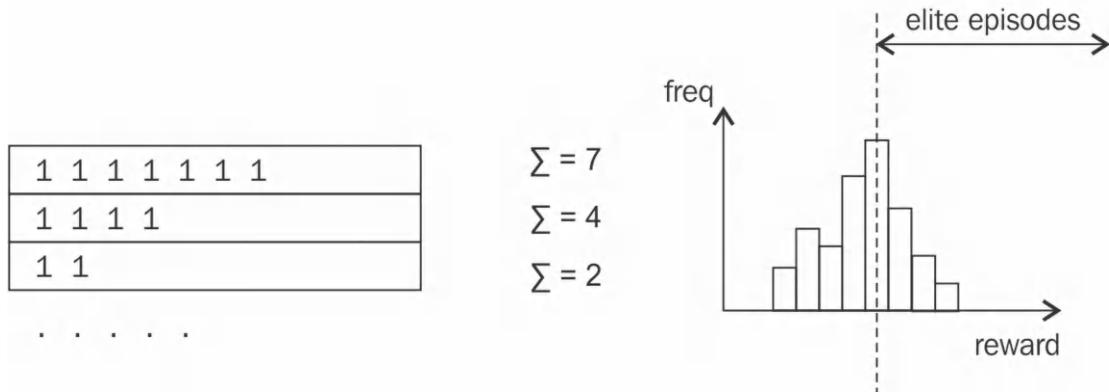


Figure 4.9: Distribution of the reward in the CartPole environment

In the FrozenLake environment, episodes and their rewards look different. We get the reward of 1.0 only when we reach the goal, and this reward says nothing about how good each episode was. Was it quick and efficient, or did we make four rounds on the lake before we randomly stepped into the final cell? We don't know; it's just a 1.0 reward and that's it. The distribution of rewards for our episodes is also problematic. There are only two kinds of episodes possible, with zero reward (failed) and one reward (successful), and failed episodes will obviously dominate at the beginning of the training, when the agent acts randomly. So, our percentile selection of elite episodes is totally wrong and gives us bad examples to train on. This is the reason for our training failure.

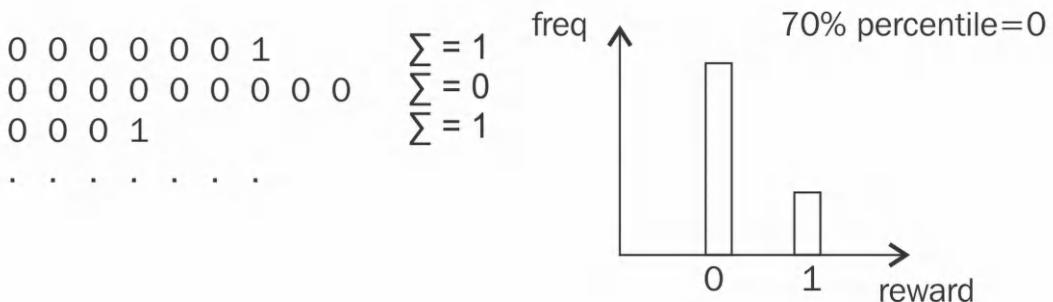


Figure 4.10: Reward distribution of the FrozenLake environment

This example shows us the limitations of the cross-entropy method:

- For training, our episodes have to be finite (in general, they could be infinite) and, preferably, short

- The total reward for the episodes should have enough variability to separate good episodes from bad ones
- It is beneficial to have an intermediate reward during the episode instead of having the reward at the end of the episode

Later in the book, you will become familiar with other methods that address these limitations. For now, if you are curious about how FrozenLake can be solved using the cross-entropy method, here is a list of tweaks of the code that you need to make (the full example is in `Chapter04/03_frozenlake_tweaked.py`):

- **Larger batches of played episodes:** In CartPole, it was sufficient to have 16 episodes on every iteration, but FrozenLake requires at least 100 just to get some successful episodes.
- **Discount factor applied to the reward:** To make the total reward for an episode depend on its length, and to add variety in episodes, we can use a discounted total reward with the discount factor $\gamma = 0.9$ or 0.95 . In this case, the reward for short episodes will be higher than the reward for long ones. This increases variability in reward distribution, which helps to avoid situations like the one shown in *Figure 4.10*.
- **Keeping elite episodes for a longer time:** In the CartPole training, we sampled episodes from the environment, trained on the best ones, and threw them away. In FrozenLake, a successful episode is a much rarer animal, so we need to keep them for several iterations to train on them.
- **Decreasing the learning rate:** This will give our NN time to average more training samples, as a smaller learning rate decreases the effect of new data on the model.
- **Much longer training time:** Due to the sparsity of successful episodes and the random outcome of our actions, it's much harder for our NN to get an idea of the best behavior to perform in any particular situation. To reach 50% successful episodes, about 5,000 training iterations are required.

To incorporate all these into our code, we need to change the `filter_batch` function to calculate the discounted reward and return elite episodes for us to keep:

```
def filter_batch(batch: tt.List[Episode], percentile: float) -> \
    tt.Tuple[tt.List[Episode], tt.List[np.ndarray], tt.List[int], float]:
    reward_fun = lambda s: s.reward * (GAMMA ** len(s.steps))
    disc_rewards = list(map(reward_fun, batch))
    reward_bound = np.percentile(disc_rewards, percentile)

    train_obs: tt.List[np.ndarray] = []
    train_act: tt.List[int] = []
    elite_batch: tt.List[Episode] = []

    for example, discounted_reward in zip(batch, disc_rewards):
        if discounted_reward > reward_bound:
```

```

    train_obs.extend(map(lambda step: step.observation, example.steps))
    train_act.extend(map(lambda step: step.action, example.steps))
    elite_batch.append(example)

    return elite_batch, train_obs, train_act, reward_bound

```

Then, in the training loop, we will store previous elite episodes to pass them to the preceding function on the next training iteration:

```

full_batch = []
for iter_no, batch in enumerate(iterate_batches(env, net, BATCH_SIZE)):
    reward_mean = float(np.mean(list(map(lambda s: s.reward, batch))))
    full_batch, obs, acts, reward_bound = filter_batch(full_batch + batch,
    PERCENTILE)
    if not full_batch:
        continue
    obs_v = torch.FloatTensor(obs)
    acts_v = torch.LongTensor(acts)
    full_batch = full_batch[-500:]

```

The rest of the code is the same, except that the learning rate decreased 10 times and the `BATCH_SIZE` was set to 100. After a period of patient waiting (the new version takes about 50 minutes to finish 10,000 iterations), you can see that the training of the model stopped improving at around 55% of solved episodes:

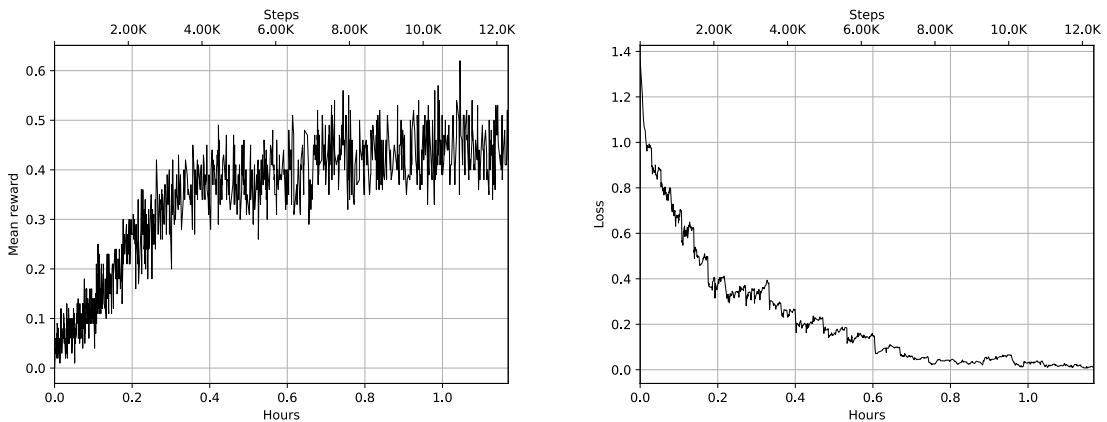


Figure 4.11: Mean reward (left) and loss (right) of the tweaked version

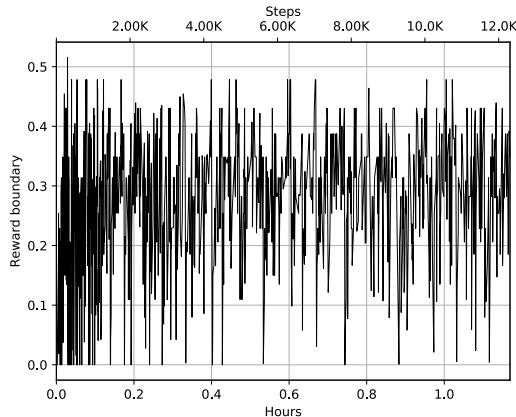


Figure 4.12: The reward boundary of the tweaked version

There are ways to address this (by applying entropy loss regularization, for example), but those techniques will be discussed in upcoming chapters.

The final point to note here is the effect of slipperiness in the FrozenLake environment. Each of our actions, with 33% probability, is replaced with the 90° rotated action (the *up* action, for instance, will succeed with a 0.33 probability, and there will be a 0.33 chance that it will be replaced with the *left* action and a 0.33 chance with the *right* action).

The nonslippery version is in `Chapter04/04_frozenlake_nonslippery.py`, and the only difference is in the environment creation:

```
env = DiscreteOneHotWrapper(gym.make("FrozenLake-v1", is_slippery=False))
```

The effect is dramatic! The nonslippery version of the environment can be solved in 120-140 batch iterations, which is 100 times faster than the noisy environment:

```
Chapter04$ ./04_frozenlake_nonslippery.py
2: loss=1.436, rw_mean=0.010, rw_bound=0.000, batch=1
3: loss=1.410, rw_mean=0.010, rw_bound=0.000, batch=2
4: loss=1.391, rw_mean=0.050, rw_bound=0.000, batch=7
5: loss=1.379, rw_mean=0.020, rw_bound=0.000, batch=9
6: loss=1.375, rw_mean=0.010, rw_bound=0.000, batch=10
7: loss=1.367, rw_mean=0.040, rw_bound=0.000, batch=14
8: loss=1.361, rw_mean=0.000, rw_bound=0.000, batch=14
9: loss=1.356, rw_mean=0.010, rw_bound=0.000, batch=15
...
```

```

134: loss=0.308, rw_mean=0.730, rw_bound=0.478, batch=93
136: loss=0.440, rw_mean=0.710, rw_bound=0.304, batch=70
137: loss=0.298, rw_mean=0.720, rw_bound=0.478, batch=106
139: loss=0.337, rw_mean=0.790, rw_bound=0.430, batch=65
140: loss=0.295, rw_mean=0.720, rw_bound=0.478, batch=99
142: loss=0.433, rw_mean=0.670, rw_bound=0.000, batch=67
143: loss=0.287, rw_mean=0.820, rw_bound=0.478, batch=114
Solved!

```

This is also evident from the following graphs:

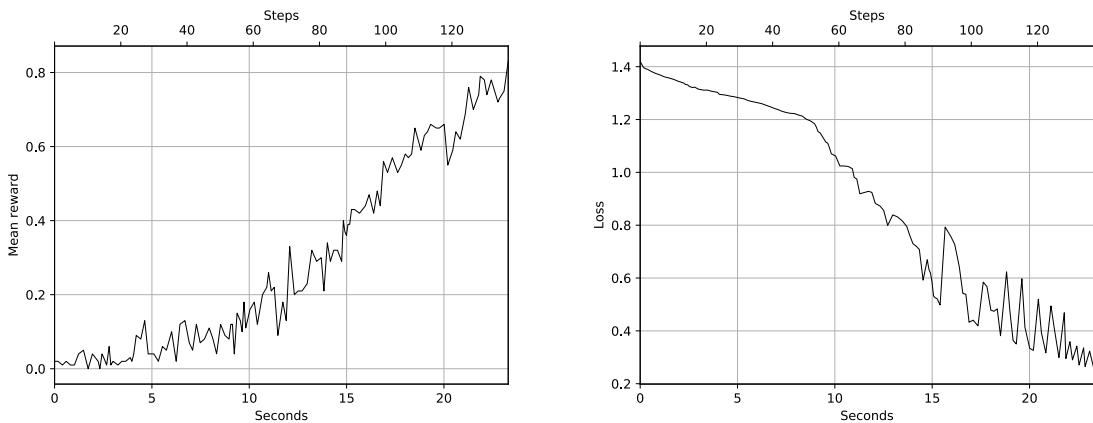


Figure 4.13: Mean reward (left) and loss (right) of the nonslippery version

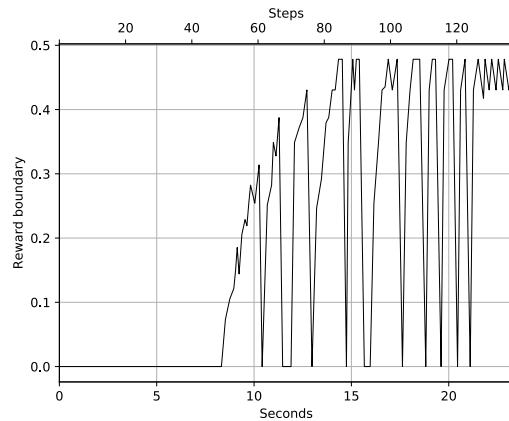


Figure 4.14: The reward boundary of the nonslippery version

The theoretical background of the cross-entropy method



This section is optional and is included for readers who want to understand why the method works. If you wish, you can refer to the original paper by Kroese, titled *Cross-entropy method*, [Kro+11].

The basis of the cross-entropy method lies in the importance sampling theorem, which states this:

$$\begin{aligned}\mathbb{E}_{x \sim p(x)}[H(x)] &= \int_x p(x)H(x)dx = \int_x q(x)\frac{p(x)}{q(x)}H(x)dx \\ &= \mathbb{E}_{x \sim q(x)}\left[\frac{p(x)}{q(x)}H(x)\right]\end{aligned}$$

In our RL case, $H(x)$ is a reward value obtained by some policy x , and $p(x)$ is a distribution of all possible policies. We don't want to maximize our reward by searching all possible policies; instead, we want to find a way to approximate $p(x)H(x)$ by $q(x)$, iteratively minimizing the distance between them. The distance between two probability distributions is calculated by **Kullback-Leibler (KL)** divergence, which is as follows:

$$\begin{aligned}KL(p_1(x) \| p_2(x)) &= \mathbb{E}_{x \sim p_1(x)} \log \frac{p_1(x)}{p_2(x)} \\ &= \mathbb{E}_{x \sim p_1(x)} [\log p_1(x)] - \mathbb{E}_{x \sim p_1(x)} [\log p_2(x)]\end{aligned}$$

The first term in KL is called entropy and it doesn't depend on $p_2(x)$, so it could be omitted during the minimization. The second term is called **cross-entropy**, which is a very common optimization objective in deep learning.

Combining both formulas, we can get an iterative algorithm, which starts with $q_0(x) = p(x)$ and on every step improves. This is an approximation of $p(x)H(x)$ with an update:

$$q_{i+1}(x) = \arg \min_{q_i(x)} -\mathbb{E}_{x \sim q_i(x)} \frac{p(x)}{q_i(x)} H(x) \log q_{i+1}(x)$$

This is a generic cross-entropy method that can be significantly simplified in our RL case. We replace our $H(x)$ with an indicator function, which is 1 when the reward for the episode is above the threshold and 0 when the reward is below.

Our policy update will look like this:

$$\pi_{i+1}(a|s) = \arg \min_{\pi_{i+1}} -\mathbb{E}_{z \sim \pi_i(a|s)} [R(z) \geq \psi_i] \log \pi_{i+1}(a|s)$$

Strictly speaking, the preceding formula misses the normalization term, but it still works in practice without it. So, the method is quite clear: we sample episodes using our current policy (starting with some random initial policy) and minimize the negative log likelihood of the most successful samples and our policy.



If you are interested, refer to the book written by Reuven Rubinstein and Dirk P. Kroese [RK04] that is dedicated to this method. A shorter description can be found in the *Cross-entropy method* paper ([Kro+11]).

Summary

In this chapter, you became familiar with the cross-entropy method, which is simple but quite powerful, despite its limitations. We applied it to a CartPole environment (with huge success) and to FrozenLake (with much more modest success). In addition, we discussed the taxonomy of RL methods, which will be referenced many times during the rest of the book, as different approaches to RL problems have different properties, which influences their applicability.

This chapter ends the introductory part of the book. In the next part, we will switch to a more systematic study of RL methods and discuss the value-based family of methods. In upcoming chapters, we will explore more complex, but more powerful, tools of deep RL.

Join our community on Discord

Read this book alongside other users, Deep Learning experts, and the author himself.

Ask questions, provide solutions to other readers, chat with the author via Ask Me Anything sessions, and much more.

Scan the QR code or visit the link to join the community.

<https://packt.link/rl>



PART II

VALUE-BASED METHODS

5

Tabular Learning and the Bellman Equation

In the previous chapter, you became acquainted with your first **reinforcement learning (RL)** algorithm, the cross-entropy method, along with its strengths and weaknesses. In this new part of the book, we will look at another group of methods that has much more flexibility and power: Q-learning. This chapter will establish the required background shared by those methods.

We will also revisit the FrozenLake environment and explore how new concepts fit with this environment and help us to address issues related to its uncertainty.

In this chapter, we will:

- Review the value of the state and the value of the action, and learn how to calculate them in simple cases
- Talk about the Bellman equation and how it establishes the optimal policy if we know the values of states
- Discuss the value iteration method and try it on the FrozenLake environment
- Do the same for the Q-iteration method

Despite the simplicity of the environments in this chapter, it establishes the required preparation for deep Q-learning, which is a very powerful and generic RL method.

Value, state, and optimality

You may remember our definition of the value of the state from *Chapter 1*. This is a very important notion and the time has come to explore it further.

This whole part of the book is built around the value of the state and how to approximate it. We defined this value as an expected total reward (optionally discounted) that is obtainable from the state. In a formal way, the value of the state is given by

$$V(s) = \mathbb{E} \left[\sum_{t=0}^{\infty} r_t \gamma^t \right]$$

where r_t is the local reward obtained at step t of the episode.

The total reward could be discounted with $0 < \gamma < 1$ or not discounted (when $\gamma = 1$); it's up to us how to define it. The value is always calculated in terms of some policy that our agent follows. To illustrate this, let's consider a very simple environment with three states, as shown in *Figure 5.1*:

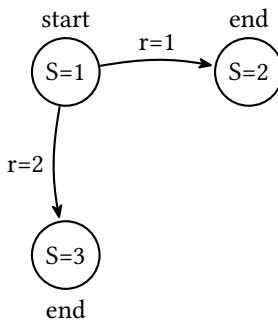


Figure 5.1: An example of an environment's state transition with rewards

1. The agent's initial state.
2. The final state that the agent is in after executing action “right” from the initial state. The reward obtained from this is 1.
3. The final state that the agent is in after action “down.” The reward obtained from this is 2.

The environment is always deterministic — every action succeeds and we always start from state 1. Once we reach either state 2 or state 3, the episode ends. Now, the question is, what's the value of state 1? This question is meaningless without information about our agent's behavior or, in other words, its policy. Even in a simple environment, our agent can have an infinite amount of behaviors, each of which will have its own value for state 1. Consider these examples:

- Agent always goes right

- Agent always goes down
- Agent goes right with a probability of 50% and down with a probability of 50%
- Agent goes right in 10% of cases and in 90% of cases executes the “down” action

To demonstrate how the value is calculated, let's do it for all the preceding policies:

- The value of state 1 in the case of the “always right” agent is **1.0** (every time it goes left, it obtains 1 and the episode ends)
- For the “always down” agent, the value of state 1 is **2.0**
- For the 50% right/50% down agent, the value is $1.0 \cdot 0.5 + 2.0 \cdot 0.5 = 1.5$
- For the 10% right/90% down agent, the value is $1.0 \cdot 0.1 + 2.0 \cdot 0.9 = 1.9$

Now, another question: what's the optimal policy for this agent? The goal of RL is to get as much total reward as possible. For this one-step environment, the total reward is equal to the value of state 1, which, obviously, is at the maximum at policy 2 (always down).

Unfortunately, such simple environments with an obvious optimal policy are not that interesting in practice. For interesting environments, the optimal policies are much harder to formulate and it's even harder to prove their optimality. However, don't worry; we are moving toward the point when we will be able to make computers learn the optimal behavior on their own.

From the preceding example, you may have a false impression that we should always take the action with the highest reward. In general, it's not that simple. To demonstrate this, let's extend our preceding environment with yet another state that is reachable from state 3. State 3 is no longer a terminal state but a transition to state 4, with a bad reward of -20. Once we have chosen the “down” action in state 1, this bad reward is unavoidable, as from state 3, we have only one exit to state 4. So, it's a trap for the agent, which has decided that “being greedy” is a good strategy.

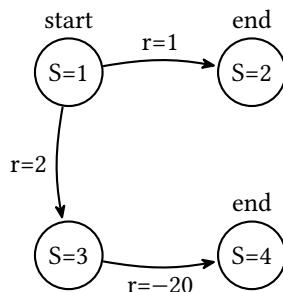


Figure 5.2: The same environment, with an extra state added

With that addition, our values for state 1 will be calculated this way:

- The “always right” agent is the same: **1.0**
- The “always down” agent gets $2.0 + (-20) = -18$
- The 50%/50% agent gets $0.5 \cdot 1.0 + 0.5 \cdot (2.0 + (-20)) = -8.5$
- The 10%/90% agent gets $0.1 \cdot 1.0 + 0.9 \cdot (2.0 + (-20)) = -16.1$

So, the best policy for this new environment is now policy 1: always go right. We spent some time discussing naïve and trivial environments so that you realize the complexity of this optimality problem and can appreciate the results of Richard Bellman better. Bellman was an American mathematician who formulated and proved his famous Bellman equation. We will talk about it in the next section.

The Bellman equation of optimality

To explain the Bellman equation, it's better to go a bit abstract. Don't be afraid; I'll provide concrete examples later to support your learning! Let's start with a deterministic case, when all our actions have a 100% guaranteed outcome. Imagine that our agent observes state s_0 and has N available actions. Every action leads to another state, $s_1 \dots s_N$, with a respective reward, $r_1 \dots r_N$. Also, assume that we know the values, V_i , of all states connected to state s_0 . What will be the best course of action that the agent can take in such a state?

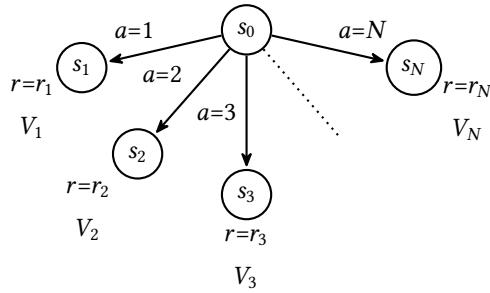


Figure 5.3: An abstract environment with N states reachable from the initial state

If we choose the concrete action, a_i , and calculate the value given to this action, then the value will be $V_0(a = a_i) = r_i + V_i$. So, to choose the best possible action, the agent needs to calculate the resulting values for every action and choose the maximum possible outcome. In other words, $V_0 = \max_{a \in 1 \dots N} (r_a + V_a)$. If we are using the discount factor, γ , we need to multiply the value of the next state by gamma: $V_0 = \max_{a \in 1 \dots N} (r_a + \gamma V_a)$.

This may look very similar to our greedy example from the previous section, and, in fact, it is. However, there is one difference: when we act greedily, we do not only look at the immediate reward for the action, but at the immediate reward plus the long-term value of the state. This allows us to avoid a possible trap with a large immediate reward but a state that has a bad value.

Bellman proved that with that extension, our behavior will get the best possible outcome. In other words, it will be optimal. So, the preceding equation is called the Bellman equation of value (for a deterministic case).

It's not very complicated to extend this idea for a stochastic case, when our actions have the chance of ending up in different states. What we need to do is calculate the expected value for every action, instead of just taking the value of the next state. To illustrate this, let's consider one single action available from state s_0 , with three possible outcomes:

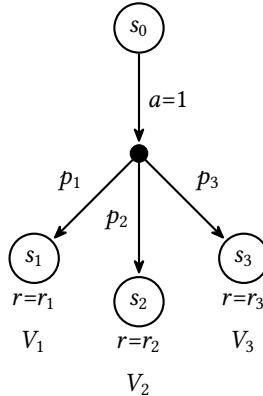


Figure 5.4: An example of the transition from the state in a stochastic case

Here, we have one action, which can lead to three different states with different probabilities. With probability p_1 , the action can end up in state s_1 , with p_2 in state s_2 , and with p_3 in state s_3 ($p_1 + p_2 + p_3 = 1$, of course). Every target state has its own reward (r_1, r_2 , or r_3). To calculate the expected value after issuing action 1, we need to sum all values, multiplied by their probabilities:

$$V_0(a = 1) = p_1(r_1 + \gamma V_1) + p_2(r_2 + \gamma V_2) + p_3(r_3 + \gamma V_3)$$

or, more formally

$$V_0(a) = \mathbb{E}_{s \sim S}[r_{s,a} + \gamma V_s] = \sum_{s \in S} p_{a,0 \rightarrow s}(r_{s,a} + \gamma V_s)$$

Here, $\mathbb{E}_{s \sim S}$ means taking the expected value over all states in our state space, S .

By combining the Bellman equation, for a deterministic case, with a value for stochastic actions, we get the Bellman optimality equation for a general case:

$$V_0 = \max_{a \in A} \mathbb{E}_{s \sim S}[r_{s,a} + \gamma V_s] = \max_{a \in A} \sum_{s \in S} p_{a,0 \rightarrow s}(r_{s,a} + \gamma V_s)$$

Note that $p_{a,i \rightarrow j}$ means the probability of action a , issued in state i , ending up in state j . The interpretation is still the same: the optimal value of the state corresponds to the action, which gives us the maximum possible expected immediate reward, plus the discounted long-term reward for the next state. You may also notice that this definition is recursive: the value of the state is defined via the values of the immediately reachable states. This recursion may look like cheating: we define some value, pretending that we already know it. However, this is a very powerful and common technique in computer science and even in math in general (proof by induction is based on the same trick). This Bellman equation is a foundation not only in RL but also in much more general dynamic programming, which is a widely used method for solving practical optimization problems.

These values not only give us the best reward that we can obtain, but they basically give us the optimal policy to obtain that reward: if our agent knows the value for every state, then it automatically knows how to gather this reward. Thanks to Bellman's optimality proof, at every state the agent ends up in, it needs to select the action with the maximum expected reward, which is a sum of the immediate reward and the one-step discounted long-term reward – that's it. So, those values are really useful to know. Before you get familiar with a practical way to calculate them, I need to introduce one more mathematical notation. It's not as fundamental as the value of the state, but we need it for our convenience.

The value of the action

To make our life slightly easier, we can define different quantities, in addition to the value of the state, $V(s)$, as the value of the action, $Q(s, a)$. Basically, this equals the total reward we can get by executing action a in state s and can be defined via $V(s)$. Being a much less fundamental entity than $V(s)$, this quantity gave a name to the whole family of methods called Q-learning, because it is more convenient. In these methods, our primary objective is to get values of Q for every pair of state and action:

$$Q(s, a) = \mathbb{E}_{s' \sim S}[r(s, a) + \gamma V(s')] = \sum_{s' \in S} p_{a,s \rightarrow s'}(r(s, a) + \gamma V(s'))$$

Q , for this state, s , and action, a , equals the expected immediate reward and the discounted long-term reward of the destination state. We also can define $V(s)$ via $Q(s, a)$:

$$V(s) = \max_{a \in A} Q(s, a)$$

This just means that the value of some state equals to the value of the maximum action we can execute from this state.

Finally, we can express $Q(s, a)$ recursively (which will be used in *Chapter 6*):

$$Q(s, a) = r(s, a) + \gamma \max_{a' \in A} Q(s', a')$$

In the last formula, the index on the immediate reward, (s, a) , depends on the environment details:

- If the immediate reward is given to us after executing a particular action, a , from state s , index (s, a) is used and the formula is exactly as shown above.
- But if the reward is provided for reaching some state, s' , via action a' , the reward will have the index (s', a') and will need to be moved into the max operator:

$$Q(s, a) = \max_{a' \in A}(r(s', a') + \gamma Q(s', a'))$$

That difference is not very significant from a mathematical point of view, but it could be important during the implementation of the methods. The first situation is more common, so we will stick to the preceding formula.

To give you a concrete example, let's consider an environment that is similar to FrozenLake, but has a much simpler structure: we have one initial state (s_0) surrounded by four target states, s_1, s_2, s_3, s_4 , with different rewards:

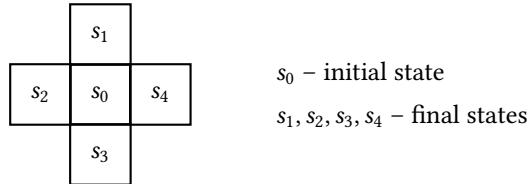


Figure 5.5: A simplified grid-like environment

Every action is probabilistic in the same way as in FrozenLake: with a 33% chance that our action will be executed without modifications, but with a 33% chance that we will slip to the left, relatively, of our target cell and a 33% chance that we will slip to the right.

For simplicity, we use discount factor $\gamma = 1$.

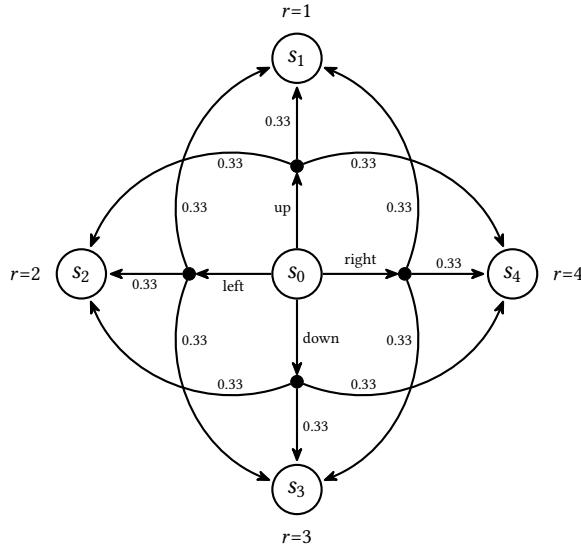


Figure 5.6: A transition diagram of the grid environment

Let's calculate the values of the actions to begin with. Terminal states $s_1 \dots s_4$ have no outbound connections, so Q for those states is zero for all actions. Due to this, the values of the terminal states are equal to their immediate reward (once we get there, our episode ends without any subsequent states): $V_1 = 1, V_2 = 2, V_3 = 3, V_4 = 4$.

The values of the actions for state 0 are a bit more complicated. Let's start with the "up" action. Its value, according to the definition, is equal to the expected sum of the immediate reward plus the long-term value for subsequent steps. We have no subsequent steps for any possible transition for the "up" action:

$$Q(s_0, up) = 0.33 \cdot V_1 + 0.33 \cdot V_2 + 0.33 \cdot V_4 = 0.33 \cdot 1 + 0.33 \cdot 2 + 0.33 \cdot 4 = 2.31$$

Repeating this for the rest of the s_0 actions results in the following:

$$Q(s_0, left) = 0.33 \cdot V_1 + 0.33 \cdot V_2 + 0.33 \cdot V_3 = 1.98$$

$$Q(s_0, right) = 0.33 \cdot V_4 + 0.33 \cdot V_1 + 0.33 \cdot V_3 = 2.64$$

$$Q(s_0, down) = 0.33 \cdot V_3 + 0.33 \cdot V_2 + 0.33 \cdot V_4 = 2.97$$

The final value for state s_0 is the maximum of those actions' values, which is 2.97.

Q -values are much more convenient in practice, as for the agent, it's much simpler to make decisions about actions based on Q than on V . In the case of Q , to choose the action based on the state, the agent just needs to calculate Q for all available actions using the current state and choose the action with the largest value of Q . To do the same using values of the states, the agent needs to know not only the values, but also the probabilities for transitions. In practice, we rarely know them in advance, so the agent needs to estimate transition probabilities for every action and state pair. Later in this chapter, you will see this in practice by solving the FrozenLake environment both ways. However, to be able to do this, we have one important thing still missing: a general way to calculate V_i and Q_i .

The value iteration method

In the simplistic example you just saw, to calculate the values of the states and actions, we exploited the structure of the environment: we had no loops in transitions, so we could start from terminal states, calculate their values, and then proceed to the central state. However, just one loop in the environment builds an obstacle in our approach. Let's consider such an environment with two states:

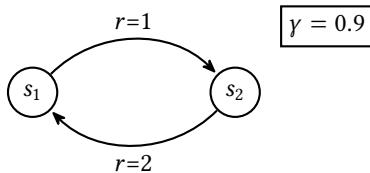


Figure 5.7: A sample environment with a loop in the transition diagram

We start from state s_1 , and the only action we can take leads us to state s_2 . We get the reward, $r = 1$, and the only transition from s_2 is an action, which brings us back to s_1 . So, the life of our agent is an infinite sequence of states $[s_1, s_2, s_1, s_2, \dots]$. To deal with this infinity loop, we can use a discount factor: $\gamma = 0.9$. Now, the question is, what are the values for both the states? The answer is not very complicated, in fact. Every transition from s_1 to s_2 gives us a reward of 1 and every back transition gives us 2. So, our sequence of rewards will be $[1, 2, 1, 2, 1, 2, 1, 2, \dots]$. As there is only one action available in every state, our agent has no choice, so we can omit the \max operation in formulas (there is only one alternative).

The value for every state will be equal to the infinite sum:

$$V(s_1) = 1 + \gamma(2 + \gamma(1 + \gamma(2 + \dots))) = \sum_{i=0}^{\infty} 1\gamma^{2i} + 2\gamma^{2i+1}$$

$$V(s_2) = 2 + \gamma(1 + \gamma(2 + \gamma(1 + \dots))) = \sum_{i=0}^{\infty} 2\gamma^{2i} + 1\gamma^{2i+1}$$

Strictly speaking, we can't calculate the exact values for our states, but with $\gamma = 0.9$, the contribution of every transition quickly decreases over time. For example, after 10 steps, $\gamma^{10} = 0.9^{10} \approx 0.349$, but after 100 steps, it becomes just 0.0000266. Due to this, we can stop after 50 iterations and still get quite a precise estimation:

```
>>> sum([0.9**[2*i] + 2*(0.9**[2*i+1]) for i in range(50)])
14.736450674121663
>>> sum([2*(0.9**[2*i]) + 0.9**[2*i+1] for i in range(50)])
15.262752483911719
```

The preceding example can be used to get the gist of a more general procedure called the **value iteration**. This allows us to numerically calculate the values of the states and values of the actions of **Markov decision processes (MDPs)** with known transition probabilities and rewards. The procedure (for values of the states) includes the following steps:

1. Initialize the values of all states, V_i , to some initial value (usually zero)
2. For every state, s , in the MDP, perform the Bellman update:

$$V_s \leftarrow \max_a \sum_{s'} p_{a,s \rightarrow s'} (r_{s,a,s'} + \gamma V_{s'})$$

3. Repeat step 2 for some large number of steps or until changes become too small

Okay, so that's the theory. In practice, this method has certain obvious limitations. First of all, our state space should be discrete and small enough to perform multiple iterations over all states. This is not an issue for FrozenLake-4x4 and even for FrozenLake-8x8 (it exists in Gym as a more challenging version), but for CartPole, it's not totally clear what to do. Our observation for CartPole is four float values, which represent some physical characteristics of the system. Potentially, even a small difference in those values could have an influence on the state's value. One of the solutions for that could be discretization of our observation's values; for example, we can split the observation space of CartPole into bins and treat every bin as an individual discrete state in space. However, this will create lots of practical problems, such as how large bin intervals should be and how much data from the environment we will need to estimate our values.

I will address this issue in subsequent chapters, when we get to the usage of neural networks in Q-learning.

The second practical problem arises from the fact that we rarely know the transition probability for the actions and rewards matrix. Remember the interface provided by Gym to the agent's writer: we observe the state, decide on an action, and only then do we get the next observation and reward for the transition. We don't know (without peeking into Gym's environment code) what the probability is of getting into state s_1 from state s_0 by issuing action a_0 . What we do have is just the history from the agent's interaction with the environment. However, in Bellman's update, we need both a reward for every transition and the probability of this transition. So, the obvious answer to this issue is to use our agent's experience as an estimation for both unknowns. Rewards could be used as they are. We just need to remember what reward we got on the transition from s_0 to s_1 using action a , but to estimate probabilities, we need to maintain counters for every tuple (s_0, s_1, a) and normalize them.

Now that you're familiar with the theoretical background, let's look at this method in practice.

Value iteration in practice

In this section, we will look at how the value iteration method will work for FrozenLake. The complete example is in `Chapter05/01_frozenlake_v_iteration.py`. The central data structures in this example are as follows:

- **Reward table:** A dictionary with the composite key “source state” + “action” + “target state.” The value is obtained from the immediate reward.
- **Transitions table:** A dictionary keeping counters of the experienced transitions. The key is the composite “state” + “action,” and the value is another dictionary that maps the “target state” into a count of times that we have seen it.
For example, if in state 0 we execute action 1 ten times, after three times, it will lead us to state 4 and after seven times to state 5. Then entry with the key `(0, 1)` in this table will be a dict with the contents `{4: 3, 5: 7}`. We can use this table to estimate the probabilities of our transitions.
- **Value table:** A dictionary that maps a state into the calculated value of this state.

The overall logic of our code is simple: in the loop, we play 100 random steps from the environment, populating the reward and transition tables. After those 100 steps, we perform a value iteration loop over all states, updating our value table. Then we play several full episodes to check our improvements using the updated value table. If the average reward for those test episodes is above the 0.8 boundary, then we stop training. During the test episodes, we also update our reward and transition tables to use all data from the environment.

Now let's come to the code. We first import the used packages and define constants. Then we define several type aliases. They are not necessary, but make our code more readable:

```

import typing as tt
import gymnasium as gym
from collections import defaultdict, Counter
from torch.utils.tensorboard.writer import SummaryWriter

ENV_NAME = "FrozenLake-v1"
GAMMA = 0.9
TEST_EPISODES = 20

```

For the FrozenLake environment, both observation and action spaces are of the Box class, so states and actions are represented by int values. We also define types for our reward and transition tables' keys. For the reward table, it is a tuple with [State, Action, State] and for the transition table it is [State, Action]:

```

State = int
Action = int
RewardKey = tt.Tuple[State, Action, State]
TransitKey = tt.Tuple[State, Action]

```

Then we define the Agent class, which will keep our tables and contain functions that we will be using in the training loop. In the class constructor, we create the environment that we will be using for data samples, obtain our first observation, and define tables for rewards, transitions, and values:

```

class Agent:
    def __init__(self):
        self.env = gym.make(ENV_NAME)
        self.state, _ = self.env.reset()
        self.rewards: tt.Dict[RewardKey, float] = defaultdict(float)
        self.transits: tt.Dict[TransitKey, Counter] = defaultdict(Counter)
        self.values: tt.Dict[State, float] = defaultdict(float)

```

The function play_n_random_steps is used to gather random experience from the environment and update the reward and transition tables. Note that we don't need to wait for the end of the episode to start learning; we just perform N steps and remember their outcomes. This is one of the differences between value iteration and the cross-entropy method, which can learn only on full episodes:

```

def play_n_random_steps(self, n: int):
    for _ in range(n):
        action = self.env.action_space.sample()
        new_state, reward, is_done, is_trunc, _ = self.env.step(action)
        rw_key = (self.state, action, new_state)

```

```

    self.rewards[rw_key] = float(reward)
    tr_key = (self.state, action)
    self.transits[tr_key][new_state] += 1
    if is_done or is_trunc:
        self.state, _ = self.env.reset()
    else:
        self.state = new_state

```

The next function (`calc_action_value()`) calculates the value of the action from the state using our transition, reward, and values tables. We will use it for two purposes: to select the best action to perform from the state and to calculate the new value of the state on value iteration.

We do the following:

1. We extract transition counters for the given state and action from the transition table. Counters in this table have a form of `dict`, with target states as the key and a count of experienced transitions as the value. We sum all counters to obtain the total count of times we have executed the action from the state. We will use this total value later to go from an individual counter to probability.
2. Then we iterate every target state that our action has landed on and calculate its contribution to the total action value using the Bellman equation. This contribution is equal to immediate reward plus discounted value for the target state. We multiply this sum to the probability of this transition and add the result to the final action value.

This logic is illustrated in the following diagram:

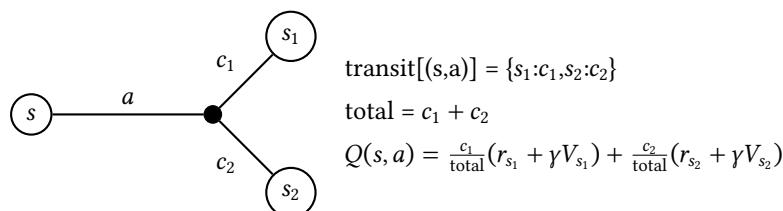


Figure 5.8: The calculation of the state's value

In the preceding diagram, we do a calculation of the value for state s and action a . Imagine that, during our experience, we have executed this action several times ($c_1 + c_2$) and it ends up in one of two states, s_1 or s_2 . How many times we have switched to each of these states is stored in our transition table as `dict {s1: c1, s2: c2}`.

Then, the approximate value for the state and action, $Q(s, a)$, will be equal to the probability of every state, multiplied by the value of the state. From the Bellman equation, this equals the sum of the immediate reward and the discounted long-term state value:

```
def calc_action_value(self, state: State, action: Action) -> float:
    target_counts = self.transits[(state, action)]
    total = sum(target_counts.values())
    action_value = 0.0
    for tgt_state, count in target_counts.items():
        rw_key = (state, action, tgt_state)
        reward = self.rewards[rw_key]
        val = reward + GAMMA * self.values[tgt_state]
        action_value += (count / total) * val
    return action_value
```

The next function uses the function that I just described to make a decision about the best action to take from the given state. It iterates over all possible actions in the environment and calculates the value for every action. The action with the largest value wins and is returned as the action to take. This action selection process is deterministic, as the `play_n_random_steps()` function introduces enough exploration. So, our agent will behave greedily in regard to our value approximation:

```
def select_action(self, state: State) -> Action:
    best_action, best_value = None, None
    for action in range(self.env.action_space.n):
        action_value = self.calc_action_value(state, action)
        if best_value is None or best_value < action_value:
            best_value = action_value
            best_action = action
    return best_action
```

The `play_episode()` function uses `select_action()` to find the best action to take and plays one full episode using the provided environment. This function is used to play test episodes, during which we don't want to mess with the current state of the main environment used to gather random data. So, we use the second environment passed as an argument. The logic is very simple and should already be familiar to you: we just loop over states accumulating the reward for one episode:

```
def play_episode(self, env: gym.Env) -> float:
    total_reward = 0.0
    state, _ = env.reset()
    while True:
        action = self.select_action(state)
        new_state, reward, is_done, is_trunc, _ = env.step(action)
        rw_key = (state, action, new_state)
        self.rewards[rw_key] = float(reward)
        tr_key = (state, action)
```

```

        self.transits[tr_key][new_state] += 1
        total_reward += reward
        if is_done or is_trunc:
            break
        state = new_state
    return total_reward

```

The final method of the Agent class is our value iteration implementation and it is surprisingly simple, thanks to the functions we already defined. What we do is just loop over all states in the environment, then for every state, we calculate the values for the states reachable from it, obtaining candidates for the value of the state. Then we update the value of our current state with the maximum value of the action available from the state:

```

def value_iteration(self):
    for state in range(self.env.observation_space.n):
        state_values = [
            self.calc_action_value(state, action)
            for action in range(self.env.action_space.n)
        ]
        self.values[state] = max(state_values)

```

That's all of our agent's methods, and the final piece is a training loop and the monitoring of the code:

```

if __name__ == "__main__":
    test_env = gym.make(ENV_NAME)
    agent = Agent()
    writer = SummaryWriter(comment="-v-iteration")

```

We create the environment that we will be using for testing, the Agent class instance, and the summary writer for TensorBoard:

```

iter_no = 0
best_reward = 0.0
while True:
    iter_no += 1
    agent.play_n_random_steps(100)
    agent.value_iteration()

```

The last two lines in the preceding code snippet are the key piece in the training loop. We first perform 100 random steps to fill our reward and transition tables with fresh data, and then we run value iteration over all states.

The rest of the code plays test episodes using the value table as our policy, then writes data into TensorBoard, tracks the best average reward, and checks for the training loop stop condition:

```

reward = 0.0
for _ in range(TEST_EPISODES):
    reward += agent.play_episode(test_env)
reward /= TEST_EPISODES
writer.add_scalar("reward", reward, iter_no)
if reward > best_reward:
    print(f"{iter_no}: Best reward updated {best_reward:.3} -> {reward:.3}")
    best_reward = reward
if reward > 0.80:
    print("Solved in %d iterations!" % iter_no)
    break
writer.close()

```

Okay, let's run our program:

```

Chapter05$ ./01_frozenlake_v_iteration.py
3: Best reward updated 0.0 -> 0.1
4: Best reward updated 0.1 -> 0.15
7: Best reward updated 0.15 -> 0.45
9: Best reward updated 0.45 -> 0.7
11: Best reward updated 0.7 -> 0.9
Solved in 11 iterations!

```

Our solution is stochastic, and my experiments usually required 10 to 100 iterations to reach a solution, but in all cases, it took less than a second to find a good policy that could solve the environment in 80% of runs. If you remember, about an hour was needed to achieve a 60% success ratio using the cross-entropy method, so this is a major improvement. There are two reasons for that.

First, the stochastic outcome of our actions, plus the length of the episodes (6 to 10 steps on average), makes it hard for the cross-entropy method to understand what was done right in the episode and which step was a mistake. Value iteration works with individual values of the state (or action) and incorporates the probabilistic outcome of actions naturally by estimating probability and calculating the expected value. So, it's much simpler for value iteration and requires much less data from the environment (which is called **sample efficiency** in RL).

The second reason is the fact that value iteration doesn't need full episodes to start learning. In an extreme case, we can start updating our values just from a single example.

However, for FrozenLake, due to the reward structure (we get 1 only after successfully reaching the target state), we still need to have at least one successful episode to start learning from a useful value table, which may be challenging to achieve in more complex environments. For example, you can try switching the existing code to a larger version of FrozenLake, which has the name `FrozenLake8x8-v1`. The larger version of FrozenLake can take from 150 to 1,000 iterations to solve, and, according to TensorBoard charts, most of the time it waits for the first successful episode, then it very quickly reaches convergence.

The following are two charts: the first one shows reward dynamics during training on `FrozenLake-4x4` and the second is for the 8×8 version.

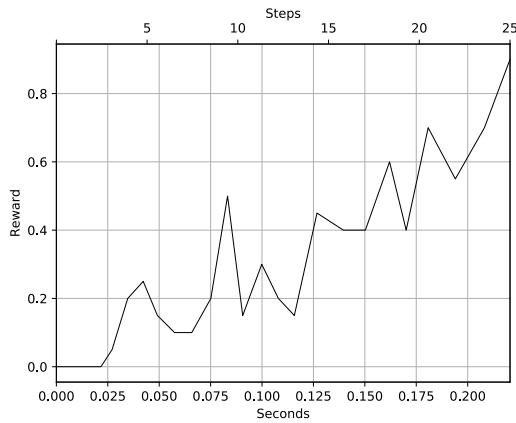


Figure 5.9: The reward dynamics for `FrozenLake-4x4`

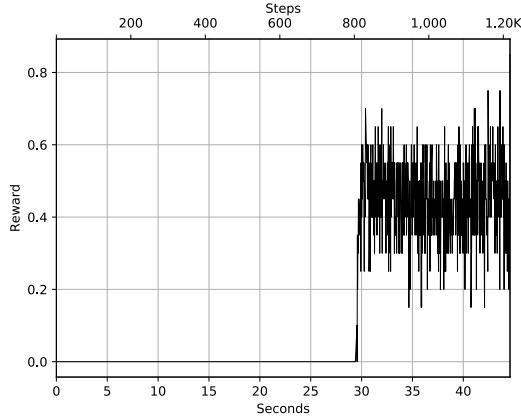


Figure 5.10: The reward dynamics on `FrozenLake-8x8`

Now it's time to compare the code that learns the values of the states, as we just discussed, with the code that learns the values of the actions.

Q-iteration for FrozenLake

The whole example is in the `Chapter05/02_frozenlake_q_iteration.py` file, and the differences are really minor:

- The most obvious change is to our value table. In the previous example, we kept the value of the state, so the key in the dictionary was just a state. Now we need to store values of the Q-function, which has two parameters, `state` and `action`, so the key in the value table is now a composite of (`State, Action`) values.
- The second difference is in our `calc_action_value()` function. We just don't need it anymore, as our action values are stored in the value table.
- Finally, the most important change in the code is in the agent's `value_iteration()` method. Before, it was just a wrapper around the `calc_action_value()` call, which did the job of Bellman approximation. Now, as this function has gone and been replaced by a value table, we need to do this approximation in the `value_iteration()` method.

Let's look at the code. As it's almost the same, I will jump directly to the most interesting `value_iteration()` function:

```
def value_iteration(self):
    for state in range(self.env.observation_space.n):
        for action in range(self.env.action_space.n):
            action_value = 0.0
            target_counts = self.transits[(state, action)]
            total = sum(target_counts.values())
            for tgt_state, count in target_counts.items():
                rw_key = (state, action, tgt_state)
                reward = self.rewards[rw_key]
                best_action = self.select_action(tgt_state)
                val = reward + GAMMA * self.values[(tgt_state, best_action)]
                action_value += (count / total) * val
            self.values[(state, action)] = action_value
```

The code is very similar to `calc_action_value()` in the previous example and, in fact, it does almost the same thing. For the given state and action, it needs to calculate the value of this action using statistics about target states that we have reached with the action. To calculate this value, we use the Bellman equation and our counters, which allow us to approximate the probability of the target state. However, in Bellman's equation, we have the value of the state; now, we need to calculate it differently.

Before, we had it stored in the value table (as we approximated the value of the states), so we just took it from this table.

We can't do this anymore, so we have to call the `select_action` method, which will choose for us the action with the largest Q-value, and then we take this Q-value as the value of the target state. Of course, we can implement another function that can calculate this value of the state, but `select_action` does almost everything we need, so we will reuse it here.

There is another piece of this example that I'd like to emphasize here. Let's look at our `select_action` method:

```
def select_action(self, state: State) -> Action:
    best_action, best_value = None, None
    for action in range(self.env.action_space.n):
        action_value = self.values[(state, action)]
        if best_value is None or best_value < action_value:
            best_value = action_value
            best_action = action
    return best_action
```

As I said, we don't have the `calc_action_value` method anymore; so, to select an action, we just iterate over the actions and look up their values in our values table. It could look like a minor improvement, but if you think about the data that we used in `calc_action_value`, it may become obvious why the learning of the Q-function is much more popular in RL than the learning of the V-function.

Our `calc_action_value` function uses both information about the reward and probabilities. It's not a huge problem for the value iteration method, which relies on this information during training. However, in the next chapter, you will learn about the value iteration method extension, which doesn't require probability approximation, but just takes it from the environment samples. For such methods, this dependency on probability adds an extra burden for the agent. In the case of Q-learning, what the agent needs to make the decision is just Q-values.

I don't want to say that V-functions are completely useless, because they are an essential part of the actor-critic method, which we will talk about in *Part 3* of this book. However, in the area of value learning, Q-functions are the definite favorite. With regard to convergence speed, both our versions are almost identical (but the Q-learning version requires four times more memory for the value table).

The following is the output of the Q-learning version and it has no major differences from the value iteration version:

```
Chapter05$ ./02_frozenlake_q_iteration.py
8: Best reward updated 0.0 -> 0.35
11: Best reward updated 0.35 -> 0.45
14: Best reward updated 0.45 -> 0.55
```

```
15: Best reward updated 0.55 -> 0.65  
17: Best reward updated 0.65 -> 0.75  
18: Best reward updated 0.75 -> 0.9  
Solved in 18 iterations!
```

Summary

My congratulations; you have made another step toward understanding modern, state-of-the-art RL methods! In this chapter, you learned about some very important concepts that are widely used in deep RL: the value of the state, the value of the action, and the Bellman equation in various forms.

We also covered the value iteration method, which is a very important building block in the area of Q-learning. Finally, you got to know how value iteration can improve our FrozenLake solution.

In the next chapter, you will learn about deep Q-networks, which started the deep RL revolution in 2013 by beating humans on lots of Atari 2600 games.

6

Deep Q-Networks

In *Chapter 5*, you became familiar with the Bellman equation and the practical method of its application called *value iteration*. This approach allowed us to significantly improve our speed and convergence in the FrozenLake environment, which is promising, but can we go further? In this chapter, we will apply the same approach to problems of much greater complexity: arcade games from the Atari 2600 platform, which are the de facto benchmark of the **reinforcement learning (RL)** research community.

To deal with this new and more challenging goal, in this chapter, we will:

- Talk about problems with the value iteration method and consider its variation, called *Q-learning*.
- Apply Q-learning to so-called grid world environments, which is called **tabular Q-learning**.
- Discuss Q-learning in conjunction with **neural networks (NNs)**. This combination has the name **deep Q-network (DQN)**.

At the end of the chapter, we will reimplement a DQN algorithm from the famous paper *Playing Atari with deep reinforcement learning* [Mni13], which was published in 2013 and started a new era in RL development. Although it is too early to discuss the practical applicability of these basic methods, this will become clearer to you as you progress with the book.

Real-life value iteration

The improvements that we got in the FrozenLake environment by switching from the cross-entropy method to the value iteration method are quite encouraging, so it's tempting to apply the value iteration method to more challenging problems. However, it is important to look at the assumptions and limitations that our value iteration method has. But let's start with a quick recap of the method. On every step, the value iteration method does a loop on all states, and for every state, it performs an update of its value with a Bellman approximation. The variation of the same method for Q-values (values for actions) is almost the same, but we approximate and store values for every state and action. So what's wrong with this process?

The first obvious problem is the count of environment states and our ability to iterate over them. In value iteration, we assume that we know all states in our environment in advance, can iterate over them, and can store their value approximations. It's easy to do for the simple grid world environment of FrozenLake, but what about other tasks?

To understand this, let's first look at how scalable the value iteration approach is, or, in other words, how many states we can easily iterate over in every loop. Even a moderate-sized computer can keep several billion float values in memory (8.5 billion in 32 GB of RAM), so the memory required for value tables doesn't look like a huge constraint. Iteration over billions of states and actions will be more **central processing unit (CPU)**-demanding but is not an insurmountable problem.

Nowadays, we have multicore systems that are mostly idle, so by using parallelism, we can iterate over billions of values in a reasonable amount of time. The real problem is the number of samples required to get good approximations for state transition dynamics. Imagine that you have some environment with, say, a billion states (which corresponds approximately to a FrozenLake of size 31600×31600). To calculate even a rough approximation for every state of this environment, we would need hundreds of billions of transitions evenly distributed over our states, which is not practical.

To give you an example of an environment with an even larger number of potential states, let's consider the Atari 2600 game console again. This was very popular in the 1980s, and many arcade-style games were available for it. The Atari console is archaic by today's gaming standards, but its games provide an excellent set of RL problems that humans can master fairly quickly, yet are still challenging for computers. Not surprisingly, this platform (using an emulator, of course) is a very popular benchmark within RL research, as I mentioned.

Let's calculate the state space for the Atari platform. The resolution of the screen is 210×160 pixels, and every pixel has one of 128 colors. So every frame of the screen has $210 \cdot 160 = 33600$ pixels and the total number of different screens possible is 128^{33600} , which is slightly more than 10^{70802} .

If we decide to just enumerate all possible states of the Atari once, it will take billions of billions of years even for the fastest supercomputer. Also, 99.(9)% of this job will be a waste of time, as most of the combinations will never be shown during even long gameplay, so we will never have samples of those states. However, the value iteration method wants to iterate over them just in case.

The second main problem with the value iteration approach is that it limits us to discrete action spaces. Indeed, both $Q(s, a)$ and $V(s)$ approximations assume that our actions are a mutually exclusive discrete set, which is not true for continuous control problems where actions can represent continuous variables, such as the angle of a steering wheel, the force on an actuator, or the temperature of a heater. This issue is much more challenging than the first, and we will talk about it in the last part of the book, in chapters dedicated to continuous action space problems. For now, let's assume that we have a discrete count of actions and that this count is not very large (i.e., orders of 10s). How should we handle the state space size issue?

Tabular Q-learning

The key question to focus on when trying to handle the state space issue is, do we really need to iterate over every state in the state space? We have an environment that can be used as a source of real-life samples of states. If some state in the state space is not shown to us by the environment, why should we care about its value? We can only use states obtained from the environment to update the values of states, which can save us a lot of work.

This modification of the value iteration method is known as Q-learning, as mentioned earlier, and for cases with explicit state-to-value mappings, it entails the following steps:

1. Start with an empty table, mapping states to values of actions.
2. By interacting with the environment, obtain the tuple s, a, r, s' (state, action, reward, and the new state). In this step, you need to decide which action to take, and there is no single proper way to make this decision. We discussed this problem as exploration versus exploitation in *Chapter 1* and will talk a lot about it in this chapter.
3. Update the $Q(s, a)$ value using the Bellman approximation:

$$Q(s, a) \leftarrow r + \gamma \max_{a' \in A} Q(s', a')$$

4. Repeat from step 2.

As in value iteration, the end condition could be some threshold of the update, or we could perform test episodes to estimate the expected reward from the policy.

Another thing to note here is how to update the Q -values. As we take samples from the environment, it's generally a bad idea to just assign new values on top of existing values, as training can become unstable.

What is usually done in practice is updating the $Q(s, a)$ with approximations using a "blending" technique, which is just averaging between old and new values of Q using learning rate α with a value from 0 to 1:

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha(r + \gamma \max_{a' \in A} Q(s', a'))$$

This allows values of Q to converge smoothly, even if our environment is noisy. The final version of the algorithm is as follows:

1. Start with an empty table for $Q(s, a)$.
2. Obtain (s, a, r, s') from the environment.
3. Make a Bellman update:

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha(r + \gamma \max_{a' \in A} Q(s', a'))$$

4. Check convergence conditions. If not met, repeat from step 2.

As mentioned earlier, this method is called tabular Q-learning, as we keep a table of states with their Q -values.

Let's try it on our FrozenLake environment. The whole example code is in `Chapter06/01_frozenlake_q_learning.py`:

First, we import packages and define constants and used types:

```
import typing as tt
import gymnasium as gym
from collections import defaultdict
from torch.utils.tensorboard.writer import SummaryWriter

ENV_NAME = "FrozenLake-v1"
GAMMA = 0.9
ALPHA = 0.2
TEST_EPISODES = 20

State = int
Action = int
ValuesKey = tt.Tuple[State, Action]

class Agent:
    def __init__(self):
        self.env = gym.make(ENV_NAME)
        self.state, _ = self.env.reset()
        self.values: tt.Dict[ValuesKey] = defaultdict(float)
```

The new thing here is the value of α , which will be used as the learning rate in the value update. The initialization of our Agent class is simpler now, as we don't need to track the history of rewards and transition counters, just our value table. This will make our memory footprint smaller, which is not a big issue for FrozenLake but can be critical for larger environments.

The method `sample_env` is used to obtain the next transition from the environment:

```
def sample_env(self) -> tt.Tuple[State, Action, float, State]:
    action = self.env.action_space.sample()
    old_state = self.state
    new_state, reward, is_done, is_tr, _ = self.env.step(action)
    if is_done or is_tr:
        self.state, _ = self.env.reset()
    else:
        self.state = new_state
    return old_state, action, float(reward), new_state
```

We sample a random action from the action space and return the tuple of the old state, the action taken, the reward obtained, and the new state. The tuple will be used in the training loop later.

The next method receives the state of the environment:

```
def best_value_and_action(self, state: State) -> tt.Tuple[float, Action]:
    best_value, best_action = None, None
    for action in range(self.env.action_space.n):
        action_value = self.values[(state, action)]
        if best_value is None or best_value < action_value:
            best_value = action_value
            best_action = action
    return best_value, best_action
```

This method finds the best action to take from the given state of the environment by taking the action with the largest value that we have in the table. If we don't have the value associated with the state and action pair, then we take it as zero. This method will be used two times: first, in the test method that plays one episode using our current values table (to evaluate our policy's quality), and second, in the method that performs the value update to get the value of the next state.

Next, we update our values table using one step from the environment:

```
def value_update(self, state: State, action: Action, reward: float, next_state: State):
    best_val, _ = self.best_value_and_action(next_state)
    new_val = reward + GAMMA * best_val
    old_val = self.values[(state, action)]
    key = (state, action)
    self.values[key] = old_val * (1-ALPHA) + new_val * ALPHA
```

Here, we first calculate the Bellman approximation for our state, s , and action, a , by summing the immediate reward with the discounted value of the next state. Then, we obtain the previous value of the state and action pair and blend these values together using the learning rate. The result is the new approximation for the value of state s and action a , which is stored in our table.

The last method in our Agent class plays one full episode using the provided test environment:

```
def play_episode(self, env: gym.Env) -> float:
    total_reward = 0.0
    state, _ = env.reset()
    while True:
        _, action = self.best_value_and_action(state)
        new_state, reward, is_done, is_tr, _ = env.step(action)
        total_reward += reward
        if is_done or is_tr:
            break
        state = new_state
    return total_reward
```

The action on every step is taken using our current value table of Q-values. This method is used to evaluate our current policy to check the progress of learning. Note that this method doesn't alter our value table; it only uses it to find the best action to take.

The rest of the example is the training loop, which is very similar to examples from *Chapter 5*: we create a test environment, agent, and summary writer, and then, in the loop, we do one step in the environment and perform a value update using the obtained data. Next, we test our current policy by playing several test episodes. If a good reward is obtained, then we stop training:

```
if __name__ == "__main__":
    test_env = gym.make(ENV_NAME)
    agent = Agent()
```

```
writer = SummaryWriter(comment="-q-learning")

iter_no = 0
best_reward = 0.0
while True:
    iter_no += 1
    state, action, reward, next_state = agent.sample_env()
    agent.value_update(state, action, reward, next_state)

    test_reward = 0.0
    for _ in range(TEST_EPISODES):
        test_reward += agent.play_episode(test_env)
    test_reward /= TEST_EPISODES
    writer.add_scalar("reward", test_reward, iter_no)
    if test_reward > best_reward:
        print("%d: Best test reward updated %.3f -> %.3f" % (iter_no, best_reward,
                                                               test_reward))
        best_reward = test_reward
    if test_reward > 0.80:
        print("Solved in %d iterations!" % iter_no)
        break
writer.close()
```

The result of the example is shown here:

```
Chapter06$ ./01_frozenlake_q_learning.py
1149: Best test reward updated 0.000 -> 0.500
1150: Best test reward updated 0.500 -> 0.550
1164: Best test reward updated 0.550 -> 0.600
1242: Best test reward updated 0.600 -> 0.650
2685: Best test reward updated 0.650 -> 0.700
2988: Best test reward updated 0.700 -> 0.750
3025: Best test reward updated 0.750 -> 0.850
Solved in 3025 iterations!
```

You may have noticed that this version used more iterations (but your experiment might have a different count of steps) to solve the problem compared to the value iteration method from the previous chapter. The reason for that is that we are no longer using the experience obtained during testing. In the example Chapter05/02_frozenlake_q_iteration.py, periodical tests caused an update of Q-table statistics. Here, we don't touch Q-values during the test, which causes more iterations before the environment gets solved.

Overall, the total number of samples required from the environment is almost the same.

The reward chart in TensorBoard also shows good training dynamics, which is very similar to the value iteration method (the reward plot for value iteration is shown in *Figure 5.9*):

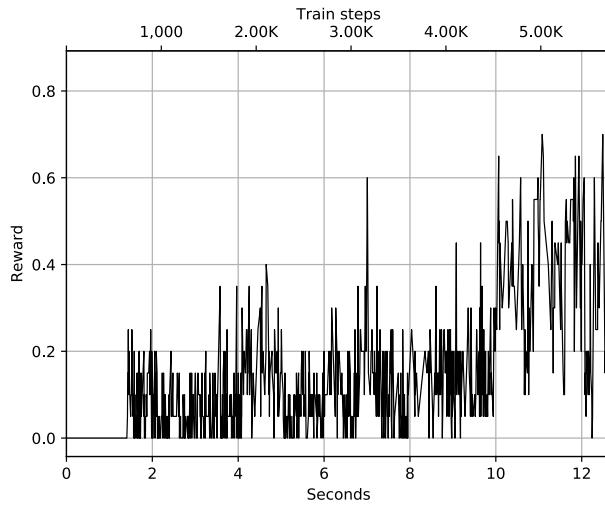


Figure 6.1: Reward dynamics of FrozenLake

In the next section, we will extend the Q-learning method with NNs' preprocessing environment states. This will greatly extend the flexibility and applicability of the method we discussed.

Deep Q-learning

The Q-learning method that we have just covered solves the issue of iteration over the full set of states, but it can still struggle with situations when the count of the observable set of states is very large. For example, Atari games can have a large variety of different screens, so if we decide to use raw pixels as individual states, we will quickly realize that we have too many states to track and approximate values for.

In some environments, the count of different observable states could be almost infinite. For example, in CartPole, the environment gives us a state that is four floating point numbers. The number of value combinations is finite (they're represented as bits), but this number is extremely large. With just bit values, it is around $2^{4 \cdot 32} \approx 3.4 \cdot 10^{38}$. In reality, it is less, as state values of the environment are bounded, so not all bit combinations of 4 `float32` values are possible, but the resulting state space is still too large. We could create some bins to discretize those values, but this often creates more problems than it solves; we would need to decide what ranges of parameters are important to distinguish as different states and what ranges could be clustered together. As we're trying to implement RL methods in a general way (without looking inside the environment's internals), this is not a very promising direction.

In the case of Atari, one single pixel change doesn't make much difference, so we might want to treat similar images as one state. However, we still need to distinguish some of the states.

The following image shows two different situations in a game of Pong. We're playing against the **artificial intelligence (AI)** opponent by controlling a paddle (our paddle is on the right, whereas our opponent's is on the left). The objective of the game is to get the bouncing ball past our opponent's paddle, while preventing the ball from getting past our paddle. We can consider the two situations to be completely different. In the situation shown on the right, the ball is close to the opponent, so we can relax and watch. However, the situation on the left is more demanding; assuming that the ball is moving from left to right, the ball is moving toward our side, so we need to move our paddle quickly to avoid losing a point. The situations in *Figure 6.2* are just two from the 10^{70802} possible situations, but we want our agent to act on them differently.

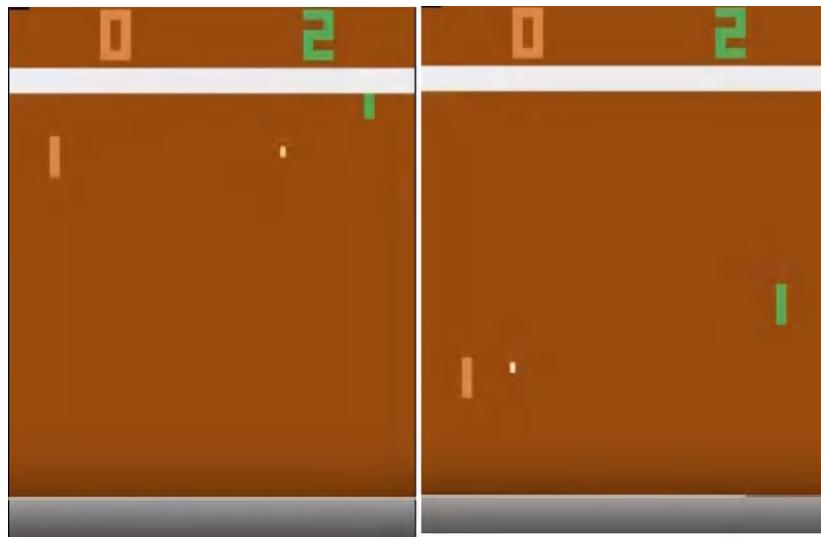


Figure 6.2: The ambiguity of observations in Pong. In the left image, the ball is moving to the right, toward our paddle, and on the right, its direction is opposite

As a solution to this problem, we can use a nonlinear representation that maps both the state and action onto a value. In machine learning, this is called a “regression problem.” The concrete way to represent and train such a representation can vary, but, as you may have already guessed from this section’s title, using a deep NN is one of the most popular options, especially when dealing with observations represented as screen images. With this in mind, let’s make modifications to the Q-learning algorithm:

1. Initialize $Q(s, a)$ with some initial approximation.
2. By interacting with the environment, obtain the tuple (s, a, r, s') .

3. Calculate the loss:

$$\begin{aligned}\mathcal{L} &= (Q(s, a) - r)^2 && \text{if episode has ended,} \\ \mathcal{L} &= (Q(s, a) - (r + \gamma \max_{a' \in A} Q_{s', a'}))^2 && \text{otherwise,}\end{aligned}$$

4. Update $Q(s, a)$ using the **stochastic gradient descent (SGD)** algorithm, by minimizing the loss with respect to the model parameters.
5. Repeat from step 2 until converged.

This algorithm looks simple, but, unfortunately, it won't work very well. Let's discuss some of the aspects that could go wrong and the potential ways we could approach these scenarios.

Interaction with the environment

First of all, we need to interact with the environment somehow to receive data to train on. In simple environments, such as FrozenLake, we can act randomly, but is this the best strategy to use? Imagine the game of Pong. What's the probability of winning a single point by randomly moving the paddle? It's not zero, but it's extremely small, which just means that we will need to wait for a very long time for such a rare situation. As an alternative, we can use our Q-function approximation as a source of behavior (as we did before in the value iteration method, when we remembered our experience during testing).

If our representation of Q is good, then the experience that we get from the environment will show the agent relevant data to train on. However, we're in trouble when our approximation is not perfect (at the beginning of the training, for example). In such a case, our agent can be stuck with bad actions for some states without ever trying to behave differently. This is the *exploration versus exploitation* dilemma mentioned briefly in *Chapter 1*, which we will discuss in detail now. On the one hand, our agent needs to explore the environment to build a complete picture of transitions and action outcomes. On the other hand, we should use interaction with the environment efficiently; we shouldn't waste time by randomly trying actions that we have already tried and learned outcomes for.

As you can see, random behavior is better at the beginning of the training when our Q approximation is bad, as it gives us more uniformly distributed information about the environment states. As our training progresses, random behavior becomes inefficient, and we want to fall back to our Q approximation to decide how to act.

A method that performs such a mix of two extreme behaviors is known as an **epsilon-greedy method**, which just means switching between random and Q policy using the probability hyperparameter ϵ . By varying ϵ , we can select the ratio of random actions. The usual practice is to start with $\epsilon = 1.0$ (100% random actions) and slowly decrease it to some small value, such as 5% or 2% random actions.

Using an epsilon-greedy method helps us to both explore the environment in the beginning and stick to good policy at the end of the training. There are other solutions to the exploration versus exploitation problem, and we will discuss some of them in the third part of the book. This problem is one of the fundamental open questions in RL and an active area of research that is not even close to being resolved completely.

SGD optimization

The core of our Q-learning procedure is borrowed from supervised learning. Indeed, we are trying to approximate a complex, nonlinear function, $Q(s, a)$, with an NN. To do this, we must calculate targets for this function using the Bellman equation and then pretend that we have a supervised learning problem at hand. That's okay, but one of the fundamental requirements for SGD optimization is that the training data is **independent and identically distributed** (frequently abbreviated as **iid**), which means that our training data is randomly sampled from the underlying dataset we're trying to learn on.

In our case, data that we are going to use for the SGD update doesn't fulfill these criteria:

1. Our samples are not independent. Even if we accumulate a large batch of data samples, they will all be very close to each other, as they will belong to the same episode.
2. Distribution of our training data won't be identical to samples provided by the optimal policy that we want to learn. Data that we have will be a result of some other policy (our current policy, a random one, or both in the case of epsilon-greedy), but we don't want to learn how to play randomly: we want an optimal policy with the best reward.

To deal with this nuisance, we usually need to use a large buffer of our past experience and sample training data from it, instead of using our latest experience. This technique is called a **replay buffer**. The simplest implementation is a buffer of a fixed size, with new data added to the end of the buffer so that it pushes the oldest experience out of it.

The replay buffer allows us to train on more-or-less independent data, but the data will still be fresh enough to train on samples generated by our recent policy. In *Chapter 8*, we will check another kind of replay buffer, **prioritized**, which provides a more sophisticated sampling approach.

Correlation between steps

Another practical issue with the default training procedure is also related to the lack of iid data, but in a slightly different manner. The Bellman equation provides us with the value of $Q(s, a)$ via $Q(s', a')$ (this process is called *bootstrapping*, when we use the formula recursively). However, both the states s and s' have only one step between them. This makes them very similar, and it's very hard for NNs to distinguish between them. When we perform an update of our NNs' parameters to make $Q(s, a)$ closer to the desired result, we can indirectly alter the value produced for $Q(s', a')$ and other states nearby.

This can make our training very unstable, like chasing our own tail; when we update Q for state s , then on subsequent states, we will discover that $Q(s', a')$ becomes worse but attempts to update it can spoil our $Q(s, a)$ approximation even more, and so on.

To make training more stable, there is a trick, called **target network**, by which we keep a copy of our network and use it for the $Q(s', a')$ value in the Bellman equation. This network is synchronized with our main network only periodically, for example, once in N steps (where N is usually quite a large hyperparameter, such as 1k or 10k training iterations).

The Markov property

Our RL methods use **Markov decision process (MDP)** formalism as their basis, which assumes that the environment obeys the Markov property: observations from the environment are all that we need to act optimally. In other words, our observations allow us to distinguish states from one another.

As you saw from the preceding Pong screenshot in *Figure 6.2*, one single image from the Atari game is not enough to capture all the important information (using only one image, we have no idea about the speed and direction of objects, like the ball and our opponent's paddle). This obviously violates the Markov property and moves our single-frame Pong environment into the area of **partially observable MDPs (POMDPs)**. A POMDP is basically an MDP without the Markov property, and it is very important in practice. For example, for most card games in which you don't see your opponents' cards, game observations are POMDPs because the current observation (i.e., your cards and the cards on the table) could correspond to different cards in your opponents' hands.

We won't discuss POMDPs in detail in this book, but we will use a small technique to push our environment back into the MDP domain. The solution is maintaining several observations from the past and using them as a state. In the case of Atari games, we usually stack k subsequent frames together and use them as the observation at every state. This allows our agent to deduct the dynamics of the current state, for instance, to get the speed of the ball and its direction. The usual "classical" number of k for Atari is four. Of course, it's just a hack, as there can be longer dependencies in the environment, but for most of the games, it works well.

The final form of DQN training

There are many more tips and tricks that researchers have discovered to make DQN training more stable and efficient, and we will cover the best of them in *Chapter 8*. However, epsilon-greedy, the replay buffer, and the target network form a basis that has allowed the company DeepMind to successfully train a DQN on a set of 49 Atari games, demonstrating the efficiency of this approach when applied to complicated environments.

The original paper *Playing Atari with deep reinforcement learning* [Mni13] (without a target network) was published at the end of 2013 and used seven games for testing.

Later, at the beginning of 2015, a revised version of the article with the title *Human-level control through deep reinforcement learning* [Mni+15], already with 49 different games, was published in *Nature*.

The algorithm for DQN from the preceding papers has the following steps:

1. Initialize parameters for $Q(s, a)$ and $\hat{Q}(s, a)$ with random weights, $\epsilon \leftarrow 1.0$, and empty the replay buffer.
2. With probability ϵ , select a random action a ; otherwise, $a = \arg \max_a Q(s, a)$.
3. Execute action a in an emulator and observe the reward, r , and the next state, s' .
4. Store the transition (s, a, r, s') in the replay buffer.
5. Sample a random mini-batch of transitions from the replay buffer.
6. For every transition in the buffer, calculate the target:

$$y = r \quad \text{if the episode has ended,}$$

$$y = r + \gamma \max_{a' \in A} \hat{Q}(s', a') \quad \text{otherwise,}$$

7. Calculate the loss: $\mathcal{L} = (Q(s, a) - y)^2$.
8. Update $Q(s, a)$ using the SGD algorithm by minimizing the loss in respect to the model parameters.
9. Every N steps, copy weights from Q to \hat{Q} .
10. Repeat from step 2 until converged.

Let's implement this algorithm now and try to beat some of the Atari games!

DQN on Pong

Before we jump into the code, some introduction is needed. Our examples are becoming increasingly challenging and complex, which is not surprising, as the complexity of the problems that we are trying to tackle is also growing. The examples are as simple and concise as possible, but some of the code may be difficult to understand at first.

Another thing to note is performance. Our previous examples for FrozenLake, or CartPole, were not demanding from a resource perspective, as observations were small, NN parameters were tiny, and shaving off extra milliseconds in the training loop wasn't important. However, from now on, that's not the case. One single observation from the Atari environment is 100k values, which have to be preprocessed, rescaled, and stored in the replay buffer. One extra copy of this data array can cost you training speed, which will not be seconds and minutes anymore but, instead, hours on even the fastest **graphics processing unit (GPU)** available.

The NN training loop could also be a bottleneck. Of course, RL models are not as huge monsters as state-of-the-art **large language models (LLMs)**, but even the DQN model from 2015 has more than 1.5M parameters, which has to be adjusted millions of times.

So, to cut a long story short, performance matters, especially when you are experimenting with hyperparameters and need to wait not for a single model to train but dozens of them.

PyTorch is quite expressive, so more-or-less efficient processing code could look much less cryptic than optimized TensorFlow graphs, but there is still a significant opportunity to do things slowly and make mistakes. For example, a naïve version of DQN loss computation, which loops over every batch sample, is about two times slower than a parallel version. However, a single extra copy of the data batch could make the speed of the same code 13 times slower, which is quite significant.

This example has been split into three modules due to its length, logical structure, and reusability. The modules are as follows:

- `Chapter06/lib/wrappers.py`: These are Atari environment wrappers, mostly taken from the **Stable Baselines3 (SB3)** project: <https://github.com/DLR-RM/stable-baselines3>.
- `Chapter06/lib/dqn_model.py`: This is the DQN NN layer, with the same architecture as the DeepMind DQN from the *Nature* paper.
- `Chapter06/02_dqn_pong.py`: This is the main module, with the training loop, loss function calculation, and experience replay buffer.

Wrappers

Tackling Atari games with RL is quite demanding from a resource perspective. To make things faster, several transformations are applied to the Atari platform interaction, which are described in DeepMind’s paper. Some of these transformations influence only performance, but some address Atari platform features that make learning long and unstable. Transformations are implemented as Gym wrappers of various kinds. The full list is quite lengthy and there are several implementations of the same wrappers in various sources. My personal favorite is the SB3 repository, which is an evolution of OpenAI Baselines code.

SB3 includes lots of RL methods implemented using PyTorch and is supposed to be a unifying benchmark to compare various methods. At the moment, we’re not interested in those methods’ implementation (we’re going to reimplement most of them ourselves), but some wrappers are very useful. The repository is available at <https://github.com/DLR-RM/stable-baselines3> and wrappers are documented at https://stable-baselines3.readthedocs.io/en/master/common/atari_wrappers.html.

The list of the most popular Atari transformations used by RL researchers includes:

- *Converting individual lives in the game into separate episodes:* In general, an episode contains all the steps from the beginning of the game until the “Game over” screen, which can last for thousands of game steps (observations and actions). Usually, in arcade games, the player is given several lives, which provide several attempts in the game. This transformation splits a full episode into individual small episodes for every life that a player has. Internally, this is implemented as checking an emulator’s information about remaining lives. Not all games support this feature (although Pong does), but for the supported environments, it usually helps to speed up convergence, as our episodes become shorter. This logic is implemented in the `EpisodicLifeEnv` wrapper in SB3 code.
- *At the beginning of the game, performing a random amount (up to 30) of empty actions (also called “no-op”):* This skips intro screens in some Atari games, which are not relevant for the gameplay. It is implemented in the `NoopResetEnv` wrapper.
- *Making an action decision every K steps, where K is usually 3 or 4:* On intermediate frames, the chosen action is simply repeated. This allows training to speed up significantly, as processing every frame with an NN is quite a demanding operation, but the difference between consequent frames is usually minor. This is implemented in the `MaxAndSkipEnv` wrapper, which also includes the next transformation in the list (the maximum between two frames).
- *Taking the maximum of every pixel in the last two frames and using it as an observation:* Some Atari games have a flickering effect, which is due to the platform’s limitation. (Atari has a limited number of sprites that can be shown on a single frame.) For the human eye, such quick changes are not visible, but they can confuse NNs.
- *Pressing FIRE at the beginning of the game:* Some games (including Pong and Breakout) require a user to press the **FIRE** button to start the game. Without this, the environment becomes a POMDP, as from observation, an agent cannot tell whether **FIRE** was already pressed. This is implemented in the `FireResetEnv` wrapper class.
- *Scaling every frame down from 210×160 , with three color frames, to a single-color 84×84 image:* Different approaches are possible. For example, the DeepMind paper describes this transformation as taking the Y-color channel from the YCbCr color space and then rescaling the full image to an 84×84 resolution. Some other researchers do grayscale transformation, cropping non-relevant parts of an image and then scaling down. In the SB3 repository, the latter approach is used. This is implemented in the `WarpFrame` wrapper class.
- *Stacking several (usually four) subsequent frames together to give the network information about the dynamics of the game’s objects:* This approach was already discussed as a quick solution to the lack of game dynamics in a single game frame. There is no wrapper in the SB3 project, I implemented my version in `wrappers.BufferWrapper`.

- *Clipping the reward to -1, 0, and 1 values:* The obtained score can vary wildly among the games. For example, in Pong, you get a score of 1 for every ball you pass behind the opponent's paddle. However, in some games, like KungFuMaster, you get a reward of 100 for every enemy killed. This spread in reward values makes our loss have completely different scales between the games, which makes it harder to find common hyperparameters for a set of games. To fix this, the reward just gets clipped to the range $-1 \dots 1$. This is implemented in the `ClipRewardEnv` wrapper.
- *Rearrange observation dimensions to meet the PyTorch convolution layer:* As we're going to use convolution, our tensors have to be rearranged the way PyTorch expects them. The Atari environment returns the observation in a (height, width, color) format, but the PyTorch convolution layer wants the channel layer to come first. This is implemented in `wrappers.ImageToPyTorch`.

Most of those wrappers are implemented in the `stable-baseline3` library, which provides the `AtariWrapper` class that applies wrappers in the required order, according to the constructor's parameters. It also detects the underlying environment properties and enables `FireResetEnv` if needed. Not all of the wrappers are required for the Pong game, but you should be aware of existing wrappers, just in case you decide to experiment with other games. Sometimes, when the DQN does not converge, the problem is not in the code but in the wrongly wrapped environment. I once spent several days debugging convergence issues, which were caused by missing the **FIRE** button press at the beginning of a game!

Let's take a look at the implementation of individual wrappers. We will start with classes provided by `stable-baseline3`:

```
class FireResetEnv(gym.Wrapper[np.ndarray, int, np.ndarray, int]):
    def __init__(self, env: gym.Env) -> None:
        super().__init__(env)
        assert env.unwrapped.get_action_meanings()[1] == "FIRE"
        assert len(env.unwrapped.get_action_meanings()) >= 3

    def reset(self, **kwargs) -> AtariResetReturn:
        self.env.reset(**kwargs)
        obs, _, terminated, truncated, _ = self.env.step(1)
        if terminated or truncated:
            self.env.reset(**kwargs)
        obs, _, terminated, truncated, _ = self.env.step(2)
        if terminated or truncated:
            self.env.reset(**kwargs)
        return obs, {}
```

The preceding wrapper presses the **FIRE** button in environments that require that for the game to start. In addition to pressing **FIRE**, this wrapper checks for several corner cases that are present in some games.

This wrapper combines the repetition of actions during K frames and pixels from two consecutive frames:

```
class MaxAndSkipEnv(gym.Wrapper[np.ndarray, int, np.ndarray, int]):
    def __init__(self, env: gym.Env, skip: int = 4) -> None:
        super().__init__(env)
        self._obs_buffer = np.zeros((2, *env.observation_space.shape),
                                    dtype=env.observation_space.dtype)
        self._skip = skip

    def step(self, action: int) -> AtariStepReturn:
        total_reward = 0.0
        terminated = truncated = False
        for i in range(self._skip):
            obs, reward, terminated, truncated, info = self.env.step(action)
            done = terminated or truncated
            if i == self._skip - 2:
                self._obs_buffer[0] = obs
            if i == self._skip - 1:
                self._obs_buffer[1] = obs
            total_reward += float(reward)
            if done:
                break
        # Note that the observation on the done=True frame
        # doesn't matter
        max_frame = self._obs_buffer.max(axis=0)

    return max_frame, total_reward, terminated, truncated, info
```

The goal of the following wrapper is to convert input observations from the emulator, which has a resolution of 210×160 pixels with RGB color channels, to a grayscale 84×84 image. It does this using CV2 library's function cvtColor, which does a colorimetric grayscale conversion (which is closer to human color perception than a simple averaging of color channels), and then the image is resized:

```
class WarpFrame(gym.ObservationWrapper[np.ndarray, int, np.ndarray]):
    def __init__(self, env: gym.Env, width: int = 84, height: int = 84) -> None:
        super().__init__(env)
        self.width = width
        self.height = height
        self.observation_space = spaces.Box(
            low=0, high=255, shape=(self.height, self.width, 1),
            dtype=env.observation_space.dtype,
        )

    def observation(self, frame: np.ndarray) -> np.ndarray:
        frame = cv2.cvtColor(frame, cv2.COLOR_RGB2GRAY)
```

```

frame = cv2.resize(frame, (self.width, self.height),
interpolation=cv2.INTER_AREA)
return frame[:, :, None]

```

So far, we've used wrappers from `stable-baseline3` (I skipped `EpisodicLifeEnv` wrapper, as it is a bit complicated and not very relevant); you can find the code of other available wrappers in the repo `stable_baselines3/common/atari_wrappers.py`. Now, let's check out two wrappers from `lib/wrappers.py`:

```

class BufferWrapper(gym.ObservationWrapper):
    def __init__(self, env, n_steps):
        super(BufferWrapper, self).__init__(env)
        obs = env.observation_space
        assert isinstance(obs, spaces.Box)
        new_obs = gym.spaces.Box(
            obs.low.repeat(n_steps, axis=0), obs.high.repeat(n_steps, axis=0),
            dtype=obs.dtype)
        self.observation_space = new_obs
        self.buffer = collections.deque(maxlen=n_steps)

    def reset(self, *, seed: tt.Optional[int] = None, options: tt.Optional[dict[str,
        tt.Any]] = None):
        for _ in range(self.buffer maxlen-1):
            self.buffer.append(self.env.observation_space.low)
        obs, extra = self.env.reset()
        return self.observation(obs), extra

    def observation(self, observation: np.ndarray) -> np.ndarray:
        self.buffer.append(observation)
        return np.concatenate(self.buffer)

```

The `BufferWrapper` class creates a stack (implemented with the `deque` class) of subsequent frames along the first dimension and returns them as an observation. The purpose is to give the network an idea about the dynamics of the objects, such as the speed and direction of the ball in Pong or how enemies are moving. This is very important information, which it is not possible to obtain from a single image.

One very important but not very obvious detail about this wrapper is that the `observation` method returns the `copy` of our buffered observations. This is very important, as we're going to keep our observations in the replay buffer, so the copy is needed to avoid buffer modification on the future environment's steps. In principle, we can avoid making a copy (and reduce our memory footprint four times) by keeping the episodes' observations and indices in them, but it will require much more sophisticated data structure management.

What is important currently is that this wrapper has to be the last in the chain of the wrappers applied to the environment.

The last wrapper is `ImageToPyTorch`, and it changes the shape of the observation from **height, width, channel (HWC)** to the **channel, height, width (CHW)** format required by PyTorch:

```
class ImageToPyTorch(gym.ObservationWrapper):
    def __init__(self, env):
        super(ImageToPyTorch, self).__init__(env)
        obs = self.observation_space
        assert isinstance(obs, gym.spaces.Box)
        assert len(obs.shape) == 3
        new_shape = (obs.shape[-1], obs.shape[0], obs.shape[1])
        self.observation_space = gym.spaces.Box(
            low=obs.low.min(), high=obs.high.max(),
            shape=new_shape, dtype=obs.dtype)

    def observation(self, observation):
        return np.moveaxis(observation, 2, 0)
```

The input shape of the tensor has a color channel as the last dimension, but PyTorch's convolution layers assume the color channel to be the first dimension.

At the end of the file is a simple function that creates an environment with a name and applies all the required wrappers to it:

```
def make_env(env_name: str, **kwargs):
    env = gym.make(env_name, **kwargs)
    env = atari_wrappers.AtariWrapper(env, clip_reward=False, noop_max=0)
    env = ImageToPyTorch(env)
    env = BufferWrapper(env, n_steps=4)
    return env
```

As you can see, we're using the `AtariWrapper` class from `stable-baseline3` and disabling some unnecessary wrappers.

That's it for wrappers; let's look at our model.

The DQN model

The model published in *Nature* has three convolution layers followed by two fully connected layers. All layers are separated by **rectified linear unit (ReLU)** nonlinearities. The output of the model is Q-values for every action available in the environment, without nonlinearity applied (as Q-values can have any value).

The approach of having all Q-values calculated with one pass through the network helps us to increase speed significantly in comparison to treating $Q(s, a)$ literally, feeding observations and actions to the network to obtain the value of the action.

The code of the model is in Chapter06/lib/dqn_model.py:

```

import torch
import torch.nn as nn

class DQN(nn.Module):
    def __init__(self, input_shape, n_actions):
        super(DQN, self).__init__()

        self.conv = nn.Sequential(
            nn.Conv2d(input_shape[0], 32, kernel_size=8, stride=4),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, stride=1),
            nn.ReLU(),
            nn.Flatten(),
        )
        size = self.conv(torch.zeros(1, *input_shape)).size()[-1]
        self.fc = nn.Sequential(
            nn.Linear(size, 512),
            nn.ReLU(),
            nn.Linear(512, n_actions)
        )
    
```

To be able to write our network in a generic way, it was implemented in two parts: convolution and linear. The convolution part processes the input image, which is a $4 \times 84 \times 84$ tensor. The output from the last convolution filter is flattened into a one-dimensional vector and fed into two `Linear` layers.

Another small problem is that we don't know the exact number of values in the output from the convolution layer produced with the input of the given shape. However, we need to pass this number to the first fully connected layer constructor. One possible solution would be to hard-code this number, which is a function of the input shape and the last convolution layer configuration (for 84×84 input, the output from the convolution layer will have 3,136 values); however, it's not the best way, as our code will become less robust to input shape change. The better solution is to obtain the required dimension in runtime, by applying the convolution part to a fake input tensor. The dimension of the result would be equal to the number of parameters returned by this application. It would be fast, as this call would be done once on model creation, and it would also allow us to have generic code.

The final piece of the model is the `forward()` function, which accepts the 4D input tensor. The first dimension is the batch size and the second is the color channel, which is our stack of subsequent frames; the third and fourth are image dimensions:

```
def forward(self, x: torch.ByteTensor):
    # scale on GPU
    xx = x / 255.0
    return self.fc(self.conv(xx))
```

Here, before applying our network, we perform the scaling and type conversion of the input data. This requires a bit of explanation.

Every pixel in the Atari image is represented as an unsigned byte with a value from 0 to 255. This is convenient in two aspects: memory efficiency and GPU bandwidth. From the memory standpoint, we should keep environment observations as small as possible because our replay buffer will keep thousands of observations, and we want to keep it small. On the other hand, during the training, we need to transfer those observations into GPU memory to calculate the gradients and update the network parameters. The bandwidth between the main memory and GPU is a limited resource, so it also makes sense to keep observations as small as possible.

That's why we keep observations as a numpy array with `dtype=uint8`, and the input tensor to the network is `ByteTensor`. But the `Conv2D` layer expects the float tensor as an input, so by dividing the input tensor by 255.0, we scale to the 0 ... 1 range and do type conversion. This is fast, as the input byte tensor is already inside the GPU memory. After that, we apply both parts of our network to the resulting scaled tensor.

Training

The third module contains the experience replay buffer, the agent, the loss function calculation, and the training loop itself. Before going into the code, something needs to be said about the training hyperparameters.

DeepMind's *Nature* paper contained a table with all the details about the hyperparameters used to train its model on *all* 49 Atari games used for evaluation. DeepMind kept all those parameters the same for all games (but trained individual models for every game), and it was the team's intention to show that the method is robust enough to solve lots of games with varying complexity, action space, reward structure, and other details using one single model architecture and hyperparameters. However, our goal here is much more modest: we want to solve just the Pong game.

Pong is quite simple and straightforward in comparison to other games in the Atari test set, so the hyperparameters in the paper are overkill for our task. For example, to get the best result on all 49 games, DeepMind used a million-observations replay buffer, which requires approximately 20 GB of RAM to store it and lots of samples from the environment to populate it.

The epsilon decay schedule that was used is also not the best for a single Pong game. In the training, DeepMind linearly decayed epsilon from 1.0 to 0.1 during the first million frames obtained from the environment. However, my own experiments have shown that for Pong, it's enough to decay epsilon over the first 150k frames and then keep it stable. The replay buffer also can be much smaller: 10k transitions will be enough.

In the following example, I've used my parameters. These differ from the parameters in the paper but will allow us to solve Pong about 10 times faster. On a GeForce GTX 1080 Ti, the following version converges to a mean score of 19.0 in about 50 minutes, but with DeepMind's hyperparameters, it will require at least a day.

This speedup, of course, involves fine-tuning for one particular environment and can break convergence on other games. You are free to play with the options and other games from the Atari set.

First, we import the required modules:

```
import gymnasium as gym
from lib import dqn_model
from lib import wrappers

from dataclasses import dataclass
import argparse
import time
import numpy as np
import collections
import typing as tt

import torch
import torch.nn as nn
import torch.optim as optim

from torch.utils.tensorboard.writer import SummaryWriter
```

We then define the hyperparameters:

```
DEFAULT_ENV_NAME = "PongNoFrameskip-v4"
MEAN_REWARD_BOUND = 19
```

These two values set the default environment to train on and the reward boundary for the last 100 episodes to stop training. If you want, you can redefine the environment name using the command-line `-env` argument:

```
GAMMA = 0.99
BATCH_SIZE = 32
REPLAY_SIZE = 10000
```

```
LEARNING_RATE = 1e-4
SYNC_TARGET_FRAMES = 1000
REPLAY_START_SIZE = 10000
```

The preceding parameters define the following:

- Our γ value used for the Bellman approximation (GAMMA)
- The batch size sampled from the replay buffer (BATCH_SIZE)
- The maximum capacity of the buffer (REPLAY_SIZE)
- The count of frames we wait for before starting training to populate the replay buffer (REPLAY_START_SIZE)
- The learning rate used in the Adam optimizer, which is used in this example (LEARNING_RATE)
- How frequently we sync model weights from the training model to the target model, which is used to get the value of the next state in the Bellman approximation (SYNC_TARGET_FRAMES)

```
EPSILON_DECAY_LAST_FRAME = 150000
EPSILON_START = 1.0
EPSILON_FINAL = 0.01
```

The last batch of hyperparameters is related to the epsilon decay schedule. To achieve proper exploration, we start with $\epsilon = 1.0$ at the early stages of training, which causes all actions to be selected randomly. Then, during the first 150,000 frames, ϵ is linearly decayed to 0.01, which corresponds to the random action taken in 1% of steps. A similar scheme was used in the original DeepMind paper, but the duration of decay was almost 10 times longer (so $\epsilon = 0.01$ was reached after a million frames).

Here, we define our type aliases and the dataclass `Experience`, used to keep entries in the experience replay buffer. It contains the current state, the action taken, the reward obtained, the termination or truncation flag, and the new state:

```
State = np.ndarray
Action = int
BatchTensors = tt.Tuple[
    torch.ByteTensor,           # current state
    torch.LongTensor,           # actions
    torch.Tensor,                # rewards
    torch.BoolTensor,            # done || trunc
    torch.ByteTensor             # next state
]

@dataclass
class Experience:
```

```
state: State
action: Action
reward: float
done_trunc: bool
new_state: State
```

The next chunk of code defines our experience replay buffer, the purpose of which is to keep the transitions obtained from the environment:

```
class ExperienceBuffer:
    def __init__(self, capacity: int):
        self.buffer = collections.deque(maxlen=capacity)

    def __len__(self):
        return len(self.buffer)

    def append(self, experience: Experience):
        self.buffer.append(experience)

    def sample(self, batch_size: int) -> tt.List[Experience]:
        indices = np.random.choice(len(self), batch_size, replace=False)
        return [self.buffer[idx] for idx in indices]
```

Each time we do a step in the environment, we push the transition into the buffer, keeping only a fixed number of steps (in our case, 10k transitions). For training, we randomly sample the batch of transitions from the replay buffer, which allows us to break the correlation between subsequent steps in the environment.

Most of the experience replay buffer code is quite straightforward: it basically exploits the capability of the deque class to maintain the given number of entries in the buffer. In the `sample()` method, we create a list of random indices and return a list of `Experience` items to be repackaged and converted into tensors.

The next class we need to have is an Agent, which interacts with the environment and saves the result of the interaction in the experience replay buffer that you have just seen:

```
class Agent:
    def __init__(self, env: gym.Env, exp_buffer: ExperienceBuffer):
        self.env = env
        self.exp_buffer = exp_buffer
        self.state: tt.Optional[np.ndarray] = None
        self._reset()

    def _reset(self):
```

```
    self.state, _ = env.reset()
    self.total_reward = 0.0
```

During the agent's initialization, we need to store references to the environment and experience replay buffer, tracking the current observation and the total reward accumulated so far.

The main method of the agent is to perform a step in the environment and store its result in the buffer. To do this, we need to select the action first:

```
@torch.no_grad()
def play_step(self, net: dqn_model.DQN, device: torch.device,
              epsilon: float = 0.0) -> tt.Optional[float]:
    done_reward = None

    if np.random.random() < epsilon:
        action = env.action_space.sample()
    else:
        state_v = torch.as_tensor(self.state).to(device)
        state_v.unsqueeze_(0)
        q_vals_v = net(state_v)
        _, act_v = torch.max(q_vals_v, dim=1)
        action = int(act_v.item())
```

With the probability `epsilon` (passed as an argument), we take the random action; otherwise, we use the model to obtain the Q-values for all possible actions and choose the best. In this method, we use the PyTorch `no_grad()` decorator to disable gradient tracking during the whole method, as we don't need them anyway.

As the action has been chosen, we pass it to the environment to get the next observation and reward, store the data in the experience buffer, and then handle the end-of-episode situation:

```
new_state, reward, is_done, is_tr, _ = self.env.step(action)
self.total_reward += reward

exp = Experience(
    state=self.state, action=action, reward=float(reward),
    done_trunc=is_done or is_tr, new_state=new_state
)
self.exp_buffer.append(exp)
self.state = new_state
if is_done or is_tr:
    done_reward = self.total_reward
    self._reset()
return done_reward
```

The result of the function is the total accumulated reward if we have reached the end of the episode with this step, or `None` otherwise.

The function `batch_to_tensors` takes the batch of `Experience` objects and returns a tuple with states, actions, rewards, done flags, and new states repacked as PyTorch tensors of the corresponding types:

```
def batch_to_tensors(batch: tt.List[Experience], device: torch.device) -> BatchTensors:
    states, actions, rewards, dones, new_state = [], [], [], [], []
    for e in batch:
        states.append(e.state)
        actions.append(e.action)
        rewards.append(e.reward)
        dones.append(e.done_trunc)
        new_state.append(e.new_state)
    states_t = torch.as_tensor(np.asarray(states))
    actions_t = torch.LongTensor(actions)
    rewards_t = torch.FloatTensor(rewards)
    dones_t = torch.BoolTensor(dones)
    new_states_t = torch.as_tensor(np.asarray(new_state))
    return states_t.to(device), actions_t.to(device), rewards_t.to(device), \
           dones_t.to(device), new_states_t.to(device)
```

When we work with states, we try to avoid memory copy (by using `np.asarray()` function), which is important, as Atari observations are large (4 frames with 84×84 bytes each), and we have a batch of 32 such objects. Without this optimization, performance drops about 20 times.

Now, it is time for the last function in the training module, which calculates the loss for the sampled batch. This function is written in a form to maximally exploit GPU parallelism by processing all batch samples with vector operations, which makes it harder to understand when compared with a naïve loop over the batch. Yet this optimization pays off: the parallel version is more than two times faster than an explicit loop.

As a reminder, here is the loss expression we need to calculate:

$$\mathcal{L} = (Q(s, a) - (r + \gamma \max_{a' \in A} \hat{Q}(s', a')))^2$$

We use the preceding equation for steps that aren't at the end of the episode and the following for the final steps:

$$\mathcal{L} = (Q(s, a) - r)^2$$

```
def calc_loss(batch: tt.List[Experience], net: dqn_model.DQN, tgt_net: dqn_model.DQN,
             device: torch.device) -> torch.Tensor:
    states_t, actions_t, rewards_t, dones_t, new_states_t = batch_to_tensors(batch,
        device)
```

In the arguments, we pass our batch, the network that we are training, and the target network, which is periodically synced with the trained one.

The first model (passed as the parameter `net`) is used to calculate gradients; the second model in the `tgt_net` argument is used to calculate values for the next states, and this calculation shouldn't affect gradients. To achieve this, we use the `detach()` function of the PyTorch tensor to prevent gradients from flowing into the target network's graph. This function was described in *Chapter 3*.

At the beginning of the function, we call the function `batch_to_tensors` to repack the batch into individual tensor variables.

The next line is a bit tricky:

```
state_action_values = net(states_t).gather(
    1, actions_t.unsqueeze(-1)
).squeeze(-1)
```

Let's discuss it in detail. Here, we pass observations to the first model and extract the specific Q-values for the taken actions, using the `gather()` tensor operation. The first argument to the `gather()` call is a dimension index that we want to perform gathering on (in our case, it is equal to 1, which corresponds to actions).

The second argument is a tensor of indices of elements to be chosen. Extra `unsqueeze()` and `squeeze()` calls are required to compute the index argument for the `gather()` function and to get rid of the extra dimensions that we created, respectively. (The index should have the same number of dimensions as the data we are processing.)

In Figure 6.3, you can see an illustration of what `gather()` does on the example case, with a batch of six entries and four actions.

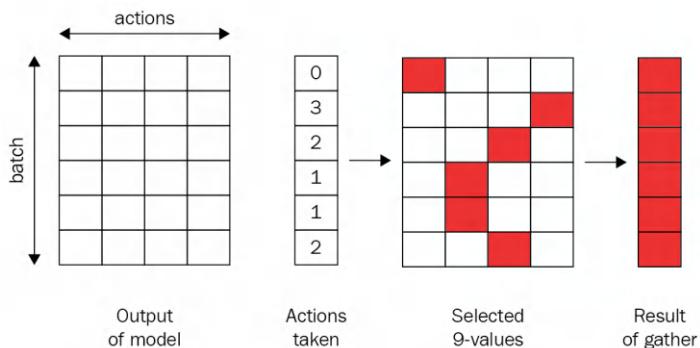


Figure 6.3: Transformation of tensors during a DQN loss calculation

Keep in mind that the result of `gather()` applied to tensors is a differentiable operation that will keep all gradients with respect to the final loss value.

Next, we disable the gradients' calculations (which ends up in a small speedup), apply the target network to our next state observations, and calculate the maximum Q-value along the same action dimension, 1:

```
with torch.no_grad():
    next_state_values = tgt_net(new_states_t).max(1)[0]
```

The function `max()` returns both maximum values and indices of those values (so it calculates both `max` and `argmax`), which is very convenient. However, in this case, we are interested only in values, so we take the first entry of the result (`max` values).

The following is the next line:

```
next_state_values[dones_t] = 0.0
```

Here, we make one simple but very important transformation: if the transition in the batch is from the last step in the episode, then our value of the action doesn't have a discounted reward for the next state, as there is no next state from which to gather the reward. This may look minor, but it is very important in practice; without this, training will not converge (I personally have wasted several hours debugging this case).

In the next line, we detach the value from its computation graph to prevent gradients from flowing into the NN used to calculate the Q approximation for the next states:

```
next_state_values = next_state_values.detach()
```

This is important, as without this, our backpropagation of the loss will start to affect both predictions for the current state and the next state. However, we don't want to touch predictions for the next state, as they are used in the Bellman equation to calculate the reference Q-values. To block gradients from flowing into this branch of the graph, we use the `detach()` method of the tensor, which returns the tensor without connection to its calculation history.

Finally, we calculate the Bellman approximation value and the mean squared error loss:

```
expected_state_action_values = next_state_values * GAMMA + rewards_t
return nn.MSELoss()(state_action_values, expected_state_action_values)
```

To get the full picture of the loss function calculation code, let's look at this function in full:

```
def calc_loss(batch: tt.List[Experience], net: dqn_model.DQN, tgt_net: dqn_model.DQN,
             device: torch.device) -> torch.Tensor:
    states_t, actions_t, rewards_t, dones_t, new_states_t = batch_to_tensors(batch,
                                                                           device)

    state_action_values = net(states_t).gather(
        1, actions_t.unsqueeze(-1)
    ).squeeze(-1)
    with torch.no_grad():
        next_state_values = tgt_net(new_states_t).max(1)[0]
        next_state_values[dones_t] = 0.0
        next_state_values = next_state_values.detach()

    expected_state_action_values = next_state_values * GAMMA + rewards_t
    return nn.MSELoss()(state_action_values, expected_state_action_values)
```

This ends our loss function calculation.

The rest of the code is our training loop. To begin with, we create a parser of command-line arguments:

```
if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("--dev", default="cpu", help="Device name, default=cpu")
    parser.add_argument("--env", default=DEFAULT_ENV_NAME,
                       help="Name of the environment, default=" + DEFAULT_ENV_NAME)
    args = parser.parse_args()
```

```
device = torch.device(args.dev)
```

Our script allows us to specify a device for computation and train on environments that are different from the default.

Here, we create our environment:

```
env = wrappers.make_env(args.env)
net = dqn_model.DQN(env.observation_space.shape, env.action_space.n).to(device)
tgt_net = dqn_model.DQN(env.observation_space.shape, env.action_space.n).to(device)
```

Our environment has all the required wrappers applied, the NN that we are going to train, and our target network with the same architecture. At first, they will be initialized with different random weights, but it doesn't matter much, as we will sync them every 1k frames, which roughly corresponds to one episode of Pong.

Then, we create our experience replay buffer of the required size and pass it to the agent:

```
writer = SummaryWriter(comment=("-" + args.env))
print(net)
buffer = ExperienceBuffer(REPLAY_SIZE)
agent = Agent(env, buffer)
epsilon = EPSILON_START
```

Epsilon is initially initialized to 1.0 but will be decreased every iteration.

Here are the last things we do before the training loop:

```
optimizer = optim.Adam(net.parameters(), lr=LEARNING_RATE)
total_rewards = []
frame_idx = 0
ts_frame = 0
ts = time.time()
best_m_reward = None
```

We create an optimizer, a buffer for full episode rewards, a counter of frames and several variables to track our speed, and the best mean reward reached. Every time our mean reward beats the record, we will save the model in the file.

At the beginning of the training loop, we count the number of iterations completed and decrease epsilon according to our schedule:

```

while True:
    frame_idx += 1
    epsilon = max(EPSILON_FINAL, EPSILON_START - frame_idx /
EPSILON_DECAY_LAST_FRAME)

```

Epsilon will drop linearly during the given number of frames (`EPSILON_DECAY_LAST_FRAME=150k`) and then be kept on the same level as `EPSILON_FINAL=0.01`.

In this block of code, we ask our agent to make a single step in the environment (using our current network and value for epsilon):

```

reward = agent.play_step(net, device, epsilon)
if reward is not None:
    total_rewards.append(reward)
    speed = (frame_idx - ts_frame) / (time.time() - ts)
    ts_frame = frame_idx
    ts = time.time()
    m_reward = np.mean(total_rewards[-100:])

    print(f"{frame_idx}: done {len(total_rewards)} games, reward {m_reward:.3f}, "
          f"eps {epsilon:.2f}, speed {speed:.2f} f/s")
    writer.add_scalar("epsilon", epsilon, frame_idx)
    writer.add_scalar("speed", speed, frame_idx)
    writer.add_scalar("reward_100", m_reward, frame_idx)
    writer.add_scalar("reward", reward, frame_idx)

```

This function returns a float value only if this step is the final step in the episode. In that case, we report our progress. Specifically, we calculate and show, both in the console and in TensorBoard, these values:

- Speed as a count of frames processed per second
- Count of episodes played
- Mean reward for the last 100 episodes
- Current value for epsilon

Every time our mean reward for the last 100 episodes reaches a maximum, we report this and save the model parameters:

```

if best_m_reward is None or best_m_reward < m_reward:
    torch.save(net.state_dict(), args.env + "-best_%0f.dat" % m_reward)
    if best_m_reward is not None:
        print(f"Best reward updated {best_m_reward:.3f} -> {m_reward:.3f}")
    best_m_reward = m_reward

```

```

if m_reward > MEAN_REWARD_BOUND:
    print("Solved in %d frames!" % frame_idx)
    break

```

If our mean reward exceeds the specified boundary, then we stop training. For Pong, the boundary is 19.0, which means winning more than 19 from 21 total games.

Here, we check whether our buffer is large enough for training:

```

if len(buffer) < REPLAY_START_SIZE:
    continue
if frame_idx % SYNC_TARGET_FRAMES == 0:
    tgt_net.load_state_dict(net.state_dict())

```

First, we should wait for enough data to be accumulated, which in our case is 10k transitions. The next condition syncs parameters from our main network to the target network every SYNC_TARGET_FRAMES, which is 1k by default.

The last piece of the training loop is very simple but requires the most time to execute:

```

optimizer.zero_grad()
batch = buffer.sample(BATCH_SIZE)
loss_t = calc_loss(batch, net, tgt_net, device)
loss_t.backward()
optimizer.step()

```

Here, we zero gradients, sample data batches from the experience replay buffer, calculate loss, and perform the optimization step to minimize the loss.

Running and performance

This example is demanding on resources. On Pong, it requires about 400k frames to reach a mean reward of 17 (which means winning more than 80% of games). A similar number of frames will be required to get from 17 to 19, as our learning progress will saturate, and it will be hard for the model to “polish the policy” and further improve the score. So, on average, a million game frames are needed to train it fully. On the GTX 1080Ti, I have a speed of about 250 frames per second, which is about an hour of training. On a CPU (i5-7600k), the speed is much slower, about 40 frames per second, which will take about seven hours. Remember that this is for Pong, which is relatively easy to solve. Other games might require hundreds of millions of frames and a 100 times larger experience replay buffer.

In *Chapter 8*, we will look at various approaches found by researchers since 2015 that can help to increase both training speed and data efficiency. *Chapter 9* will be devoted to engineering tricks to speed up RL methods' performance. Nevertheless, for Atari, you will need resources and patience. The following figure shows a chart with reward dynamics during the training:

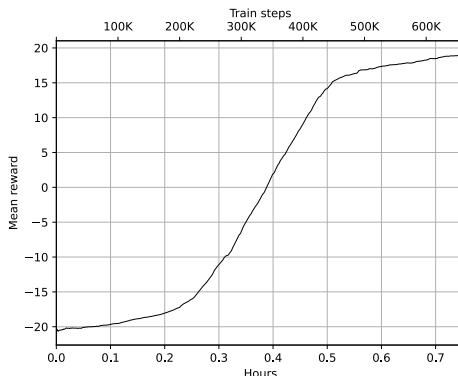


Figure 6.4: Dynamics of average reward calculated over the last 100 episodes

Now, let's look at the console output from our training process (only the beginning of the output is shown):

```
Chapter06$ ./02_dqn_pong.py --dev cuda
A.L.E: Arcade Learning Environment (version 0.8.1+53f58b7)
[Powered by Stella]
DQN(
    (conv): Sequential(
        (0): Conv2d(4, 32, kernel_size=(8, 8), stride=(4, 4))
        (1): ReLU()
        (2): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2))
        (3): ReLU()
        (4): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1))
        (5): ReLU()
        (6): Flatten(start_dim=1, end_dim=-1)
    )
    (fc): Sequential(
        (0): Linear(in_features=3136, out_features=512, bias=True)
        (1): ReLU()
        (2): Linear(in_features=512, out_features=6, bias=True)
    )
)
940: done 1 games, reward -21.000, eps 0.99, speed 1214.95 f/s
1946: done 2 games, reward -20.000, eps 0.99, speed 1420.09 f/s
Best reward updated -21.000 -> -20.000
```

```

2833: done 3 games, reward -20.000, eps 0.98, speed 1416.26 f/s
3701: done 4 games, reward -20.000, eps 0.98, speed 1421.84 f/s
4647: done 5 games, reward -20.200, eps 0.97, speed 1421.63 f/s
5409: done 6 games, reward -20.333, eps 0.96, speed 1395.67 f/s
6171: done 7 games, reward -20.429, eps 0.96, speed 1411.90 f/s
7063: done 8 games, reward -20.375, eps 0.95, speed 1404.49 f/s
7882: done 9 games, reward -20.444, eps 0.95, speed 1388.26 f/s
8783: done 10 games, reward -20.400, eps 0.94, speed 1283.64 f/s
9545: done 11 games, reward -20.455, eps 0.94, speed 1376.47 f/s
10307: done 12 games, reward -20.500, eps 0.93, speed 431.94 f/s
11362: done 13 games, reward -20.385, eps 0.92, speed 276.14 f/s
12420: done 14 games, reward -20.214, eps 0.92, speed 276.44 f/s

```

During the first 10k steps, our speed is very high, as we don't do any training, which is the most expensive operation in our code. After 10k, we start sampling the training batches and the performance drops to more representative numbers. During the training, the performance also decreases slightly, just because of the epsilon decrease. When ϵ is high, the actions are chosen randomly. As ϵ approaches zero, we need to perform inference to get Q-values for action selection, which also costs time.

Several dozens of games later, our DQN should start to figure out how to win 1 or 2 games out of 21, and an average reward begins to grow (this normally happens around $\epsilon = 0.5$):

```

66024: done 68 games, reward -20.162, eps 0.56, speed 260.89 f/s
67338: done 69 games, reward -20.130, eps 0.55, speed 257.63 f/s
68440: done 70 games, reward -20.100, eps 0.54, speed 260.17 f/s
69467: done 71 games, reward -20.113, eps 0.54, speed 260.02 f/s
70792: done 72 games, reward -20.125, eps 0.53, speed 258.88 f/s
72031: done 73 games, reward -20.123, eps 0.52, speed 259.54 f/s
73314: done 74 games, reward -20.095, eps 0.51, speed 258.16 f/s
74815: done 75 games, reward -20.053, eps 0.50, speed 257.56 f/s
76339: done 76 games, reward -20.026, eps 0.49, speed 256.79 f/s
77576: done 77 games, reward -20.013, eps 0.48, speed 257.86 f/s
78978: done 78 games, reward -19.974, eps 0.47, speed 255.90 f/s
80093: done 79 games, reward -19.962, eps 0.47, speed 256.84 f/s
81565: done 80 games, reward -19.938, eps 0.46, speed 256.34 f/s
83365: done 81 games, reward -19.901, eps 0.44, speed 254.22 f/s
84841: done 82 games, reward -19.878, eps 0.43, speed 254.80 f/s

```

Finally, after many more games, our DQN can finally dominate and beat the (not very sophisticated) built-in Pong AI opponent:

```
737860: done 371 games, reward 18.540, eps 0.01, speed 225.22 f/s
739935: done 372 games, reward 18.650, eps 0.01, speed 232.70 f/s
Best reward updated 18.610 -> 18.650
741910: done 373 games, reward 18.650, eps 0.01, speed 231.66 f/s
743964: done 374 games, reward 18.760, eps 0.01, speed 231.59 f/s
Best reward updated 18.650 -> 18.760
745939: done 375 games, reward 18.770, eps 0.01, speed 223.45 f/s
Best reward updated 18.760 -> 18.770
747950: done 376 games, reward 18.810, eps 0.01, speed 229.84 f/s
Best reward updated 18.770 -> 18.810
749925: done 377 games, reward 18.810, eps 0.01, speed 228.05 f/s
752008: done 378 games, reward 18.910, eps 0.01, speed 225.41 f/s
Best reward updated 18.810 -> 18.910
753983: done 379 games, reward 18.920, eps 0.01, speed 229.75 f/s
Best reward updated 18.910 -> 18.920
755958: done 380 games, reward 19.030, eps 0.01, speed 228.71 f/s
Best reward updated 18.920 -> 19.030
Solved in 755958 frames!
```

Due to randomness in the training process, your actual dynamics might differ from what is displayed here. In some rare cases (1 run from 10 according to my experiments), the training does not converge at all, which looks like a constant stream of rewards –21 for a long time. This is not an uncommon situation in deep learning (due to the randomness of the training) and might occur even more often in RL (due to the added randomness of environment communication). If your training doesn't show any positive dynamics for the first 100k–200k iterations, you should restart it.

Your model in action

The training process is just one part of the picture. Our final goal is not only to train the model; we also want our model to play the game with a good outcome. During the training, every time we update the maximum of the mean reward for the last 100 games, we save the model into the file `PongNoFrameskip-v4-best_<score>.dat`. In the `Chapter06/03_dqn_play.py` file, we have a program that can load this model file and play one episode, displaying the model's dynamics.

The code is very simple, but it can be like magic seeing how several matrices, with just a million parameters, can play Pong with superhuman accuracy by observing only the pixels.

First, we import the familiar PyTorch and Gym modules:

```
import gymnasium as gym
import argparse
import numpy as np
```

```

import typing as tt

import torch

from lib import wrappers
from lib import dqn_model

import collections

DEFAULT_ENV_NAME = "PongNoFrameskip-v4"

```

The script accepts the filename of the saved model and allows the specification of the Gym environment (of course, the model and environment have to match):

```

if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("-m", "--model", required=True, help="Model file to load")
    parser.add_argument("-e", "--env", default=DEFAULT_ENV_NAME,
                        help="Environment name to use, default=" + DEFAULT_ENV_NAME)
    parser.add_argument("-r", "--record", required=True, help="Directory for video")
    args = parser.parse_args()

```

Additionally, you have to pass option `-r` with the name of a nonexistent directory, which will be used to save a video of your game.

The following code is also not very complicated:

```

env = wrappers.make_env(args.env, render_mode="rgb_array")
env = gym.wrappers.RecordVideo(env, video_folder=args.record)
net = dqn_model.DQN(env.observation_space.shape, env.action_space.n)
state = torch.load(args.model, map_location=lambda stg, _: stg)
net.load_state_dict(state)

state, _ = env.reset()
total_reward = 0.0
c: tt.Dict[int, int] = collections.Counter()

```

We create the environment, wrap it in the `RecordVideo` wrapper, create the model, and then we load weights from the file passed in the arguments. The argument `map_location`, passed to the `torch.load()` function, is needed to map the loaded tensor location from the GPU to the CPU. By default, torch tries to load tensors on the same device where they were saved, but if you copy the model from the machine you used for training (with a GPU) to a laptop without a GPU, the locations need to be remapped.

Our example doesn't use the GPU at all, as inference is fast enough without acceleration.

This is almost an exact copy of the Agent class' method `play_step()` from the training code, without the epsilon-greedy action selection:

```
while True:
    state_v = torch.tensor([state])
    q_vals = net(state_v).data.numpy()[0]
    action = int(np.argmax(q_vals))
    c[action] += 1
```

We just pass our observation to the agent and select the action with the maximum value.

The rest of the code is also simple:

```
state, reward, is_done, is_trunc, _ = env.step(action)
total_reward += reward
if is_done or is_trunc:
    break
print("Total reward: %.2f" % total_reward)
print("Action counts:", c)
env.close()
```

We pass the action to the environment, count the total reward, and stop our loop when the episode ends. After the episode, we show the total reward and the number of times that the agent executed the action.

In this YouTube playlist, you can find recordings of the gameplay at different stages of the training: <https://www.youtube.com/playlist?list=PLMVwuZENsfJklt4vCltrWq0KV9aEZ3ylu>.

Things to try

If you are curious and want to experiment with this chapter's material on your own, then here is a shortlist of directions to explore. Be warned though: they can take lots of time and may cause you some moments of frustration during your experiments. However, these experiments are a very efficient way to really master the material from a practical point of view:

- Try to take some other games from the Atari set, such as Breakout, Atlantis, or River Raid (my childhood favorite). This could require the tuning of hyperparameters.
- As an alternative to FrozenLake, there is another tabular environment, Taxi, which emulates a taxi driver who needs to pick up passengers and take them to a destination.

- Play with Pong hyperparameters. Is it possible to train faster? OpenAI claims that it can solve Pong in 30 minutes using the asynchronous advantage actor-critic method (which is a subject of *Part 3* of this book). Maybe it's possible with a DQN.
- Can you make the DQN training code faster? The OpenAI Baselines project has shown 350 FPS using TensorFlow on GTX 1080 Ti. So, it looks like it's possible to optimize the PyTorch code. We will discuss this topic in *Chapter 9*, but meanwhile, you can do your own experiments.
- In the video recording, you might notice that models with a mean score around zero play quite well. In fact, I had the impression that those models play better than models with mean scores of 10–19. This might be the case due to overfitting to the particular game situations. Could you try to fix this? Maybe it would be possible to use a generative adversarial network-style approach to make one model play with another?
- Can you get the Ultimate Pong Dominator model with a mean score of 21? It shouldn't be very hard – the learning rate decay is the obvious method to try.

Summary

In this chapter, we covered a lot of new and complex material. You became familiar with the limitations of value iteration in complex environments with large observation spaces, and we discussed how to overcome them with Q-learning. We checked the Q-learning algorithm on the FrozenLake environment and discussed the approximation of Q-values with NNs, as well as the extra complications that arise from this approximation.

We covered several tricks with DQNs to improve their training stability and convergence, such as an experience replay buffer, target networks, and frame stacking. Finally, we combined those extensions into one single implementation of DQN that solves the Pong environment from the Atari games suite.

In the next chapter, we will take a quick look at higher-level RL libraries, and after that, we will take a look at a set of tricks that researchers have found since 2015 to improve DQN convergence and quality, which (combined) can produce state-of-the-art results on most of the 54 (newly added) Atari games. This set was published in 2017, and we will analyze and reimplement all of the tricks.

7

Higher-Level RL Libraries

In *Chapter 6*, we implemented the **deep Q-network (DQN)** model published by DeepMind in 2015 [Mni+15]. This paper had a significant effect on the RL field by demonstrating that, despite common belief, it's possible to use nonlinear approximators in RL. This proof of concept stimulated great interest in the deep Q-learning field and in deep RL in general.

In this chapter, we will take another step toward a practical RL by discussing higher-level RL libraries, which will allow you to build your code from higher-level blocks and focus on the details of the method that you are implementing, avoiding reimplementing the same logic multiple times. Most of the chapter will describe the **PyTorch AgentNet (PTAN)** library, which will be used in the rest of the book to prevent code repetition, so will be covered in detail.

We will cover:

- The motivation for using high-level libraries, rather than reimplementing everything from scratch
- The PTAN library, including coverage of the most important parts, which will be illustrated with code examples
- DQN on CartPole, implemented using the PTAN library
- Other RL libraries that you might consider

Why RL libraries?

Our implementation of basic DQN in *Chapter 6* wasn't very, long and complicated—about 200 lines of training code plus 50 lines in environment wrappers. When you are becoming familiar with RL methods, it is very useful to implement everything yourself to understand how things actually work. However, the more involved you become in the field, the more often you will realize that you are writing the same code over and over again.

This repetition comes from the generality of RL methods. As we discussed in *Chapter 1*, RL is quite flexible, and many real-life problems fall into the environment-agent interaction scheme. RL methods don't make many assumptions about the specifics of observations and actions, so code implemented for the CartPole environment will be applicable to Atari games (maybe with some minor tweaks).

Writing the same code over and over again is not very efficient, as bugs might be introduced every time, which will cost you time for debugging and understanding. In addition, carefully designed code that has been used in several projects usually has a higher quality in terms of performance, unit tests, readability, and documentation.

The practical applications of RL are quite young by computer science standards, so in comparison to other more mature domains, you might not have that rich a choice of approaches. For example, in web development, even if you limit yourself to just Python, you have hundreds of very good libraries of all sorts: Django for heavyweight, fully functional websites; Flask for light **Web Server Gateway Interface (WSGI)** apps; and much more, large and small.

RL is not as mature as web frameworks, but still, you can choose from several projects that are trying to simplify RL practitioners' lives. In addition, you can always write your own set of tools, as I did several years ago. The tool I created is a library called PTAN, and, as mentioned, it will be used in the rest of the book to illustrate examples.

The PTAN library

This library is available on GitHub: <https://github.com/Shmuma/ptan>. All the subsequent examples were implemented using version 0.8 of PTAN, which can be installed in your virtual environment by running the following:

```
$ pip install ptan==0.8
```

The original goal of PTAN was to simplify my RL experiments, and it tries to keep the balance between two extremes:

- Import the library and then write just a couple of lines with tons of parameters to train one of the provided methods, like DQN (a very vivid example is the OpenAI Baselines and Stable Baselines3 projects). This first approach is very inflexible. It works well when you are using the library the way it is supposed to be used. But if you want to do something fancy, you will quickly find yourself hacking the library and fighting with the constraints it imposes, rather than solving the problem you want to solve.
- Implement all the method's logic from scratch. This second extreme gives too much freedom and requires implementing replay buffers and trajectory handling over and over again, which is error-prone, boring, and inefficient.

PTAN tries to balance those extremes, providing high-quality building blocks to simplify your RL code, but at the same time being flexible and not limiting your creativity.

At a high level, PTAN provides the following entities:

- **Agent:** A class that knows how to convert a batch of observations to a batch of actions to be executed. It can contain an optional state, in case you need to track some information between consequent actions in one episode. (We will use this approach in *Chapter 15*, in the **deep deterministic policy gradient (DDPG)** method, which includes the Ornstein–Uhlenbeck random process for exploration.) The library provides several agents for the most common RL cases, but you always can write your own subclass of `BaseAgent` if none of the predefined classes are meeting your needs.
- **ActionSelector:** A small piece of logic that knows how to choose the action from some output of the network. It works in tandem with the `Agent` class.
- **ExperienceSource** and subclasses: The `Agent` instance and a Gym environment object can provide information about the trajectory of the agent during the episodes. In its simplest form, it is one single (a, r, s') transition at a time, but its functionality goes beyond this.
- **ExperienceSourceBuffer** and subclasses: Replay buffers with various characteristics. They include a simple replay buffer and two versions of prioritized replay buffers.
- Various utility classes: An example is `TargetNet` and wrappers for time-series preprocessing (used for tracking training progress in TensorBoard).
- PyTorch Ignite helpers: These can be used to integrate PTAN into the Ignite framework.
- Wrappers for Gym environments: For example, wrappers for Atari games (very similar to the wrappers we described in *Chapter 6*).

And that's basically it. In the following sections, we will take a look at these entities in detail.

Action selectors

In PTAN terminology, an *action selector* is an object that helps with going from network output to concrete action values. The most common cases include:

- **Greedy (or argmax):** Commonly used by Q-value methods when the network predicts Q-values for a set of actions and the desired action is the action with the largest $Q(s, a)$.
- **Policy-based:** The network outputs the probability distribution (in the form of logits or normalized distribution), and an action needs to be sampled from this distribution. You have already seen this in *Chapter 4*, when we discussed the cross-entropy method.

An action selector is used by the Agent and rarely needs to be customized (but you have the option). The concrete classes provided by the library are:

- `ArgmaxActionSelector`: Applies `argmax` on the second axis of a passed tensor. It assumes a matrix with batch dimension along the first axis.
- `ProbabilityActionSelector`: Samples from the probability distribution of a discrete set of actions.
- `EpsilonGreedyActionSelector`: Has the `epsilon` parameter, which specifies the probability of a random action being taken. It also holds another `ActionSelector` instance, which is used when we're not sampling random actions.

All the classes assume that NumPy arrays will be passed to them. The complete example from this section can be found in `Chapter07/01_actions.py`. Here, I'm going to show you how these classes are supposed to be used:

```
>>> import numpy as np
>>> import ptan
>>> q_vals = np.array([[1, 2, 3], [1, -1, 0]])
>>> q_vals
array([[ 1,  2,  3],
       [ 1, -1,  0]])
>>> selector = ptan.actions.ArgmaxActionSelector()
>>> selector(q_vals)
array([2, 0])
```

As you can see, the selector returns indices of actions with the largest values.

The next action selector is `EpsilonGreedyActionSelector`, which “wraps” another action selector and, depending on the `epsilon` parameter, either uses the wrapped action selector or takes the random action. This action selector is used during training to introduce randomness to the agent's actions. If `epsilon` is `0.0`, no random actions are taken:

```
>>> selector = ptan.actions.EpsilonGreedyActionSelector(epsilon=0.0,
   selector=ptan.actions.ArgmaxActionSelector())
>>> selector(q_vals)
array([2, 0])
```

If we change `epsilon` to 1, actions will be random:

```
>>> selector = ptan.actions.EpsilonGreedyActionSelector(epsilon=1.0)
>>> selector(q_vals)
array([0, 1])
```

You can also change the value of `epsilon` by assigning the action selector's attribute:

```
>>> selector.epsilon
1.0
>>> selector.epsilon = 0.0
>>> selector(q_vals)
array([2, 0])
```

Working with `ProbabilityActionSelector` is the same, but the input needs to be a normalized probability distribution:

```
>>> selector = ptan.actions.ProbabilityActionSelector()
>>> for _ in range(10):
...     acts = selector(np.array([
...         [0.1, 0.8, 0.1],
...         [0.0, 0.0, 1.0],
...         [0.5, 0.5, 0.0]
...     ]))
...     print(acts)
...
[0 2 1]
[1 2 1]
[1 2 1]
[0 2 1]
[2 2 0]
[0 2 0]
[1 2 1]
[1 2 0]
[1 2 1]
[1 2 0]
```

In the preceding example, we sample from three distributions (as we have three rows in the passed matrix):

- The first is defined by the vector $[0.1, 0.8, 0.1]$; as a result, the action with index 1 is chosen with probability 80%
- The vector $[0.0, 0.0, 1.0]$ always gives us an action with index 2
- The distribution $[0.5, 0.5, 0.0]$ produces actions 0 and 1 with 50% chance

The agent

The agent entity provides an unified way of bridging observations from the environment and the actions that we want to execute. So far, you have seen only a simple, stateless DQN agent that uses a **neural network (NN)** to obtain action values from the current observation and behaves greedily on those values. We have used epsilon-greedy behavior to explore the environment, but this doesn't change the picture much.

In the RL field, this could be more complicated. For example, instead of predicting the values of the actions, our agent could predict a probability distribution over the actions. Such agents are called *policy agents*, and we will talk about those methods in *Part 3* of the book.

In some situations, it might be necessary for the agent to keep state between observations. For example, very often, one observation (or even the k last observations) is not enough to make a decision about the action, and we want to keep some memory in the agent to capture the necessary information. There is a whole subdomain of RL that tries to address this complication with **partially observable Markov decision process (POMDP)** formalism, which we briefly mentioned in *Chapter 6* but is not covered extensively in the book.

The third variant of the agent is very common in *continuous control problems*, which will be discussed in *Part 4* of the book. For now, it suffices to say that in such cases, actions are not discrete anymore but continuous values, and the agent needs to predict them from the observations.

To capture all those variants and make the code flexible, the agent in PTAN is implemented as an extensible hierarchy of classes with the `ptan.agent.BaseAgent` abstract class at the top. From a high level, the agent needs to accept the batch of observations (in the form of a NumPy array or a list of NumPy arrays) and return the batch of actions that it wants to take. The batch is used to make the processing more efficient, as processing several observations in one pass in a **graphics processing unit (GPU)** is frequently much faster than processing them individually.

The abstract base class doesn't define the types of input and output, which makes it very flexible and easy to extend. For example, in the continuous domain, our actions will no longer be indices of discrete actions, but float values.

In any case, the agent can be seen as something that knows how to convert observations into actions, and it's up to the agent how to do this. In general, there are no assumptions made on observation and action types, but the concrete implementation of agents is more limiting. PTAN provides two of the most common ways to convert observations into actions: `DQNAgent` and `PolicyAgent`. We will explore these in subsequent sections.

However, in real problems, a custom agent is often needed. These are some of the reasons:

- The architecture of the NN is fancy—its action space is a mixture of continuous and discrete and it has multimodal observations (text and pixels, for example), or something like that.
- You want to use non-standard exploration strategies, for example, the Ornstein–Uhlenbeck process (a very popular exploration strategy in the continuous control domain).
- You have a POMDP environment, and the agent's decision is not fully defined by observations, but by some internal agent state (which is also the case for Ornstein–Uhlenbeck exploration).

All those cases are easily supported by subclassing the `BaseAgent` class, and in the rest of the book, several examples of such redefinition will be given.

Let's now check the standard agents provided by the library: `DQNAgent` and `PolicyAgent`. The complete example is in `Chapter07/02_agents.py`.

DQNAgent

This class is applicable in Q-learning when the action space is not very large, which covers Atari games and lots of classical problems. This representation is not universal and, later in the book, you will see ways of dealing with that. `DQNAgent` takes a batch of observations as input (as a NumPy array), applies the network to them to get Q-values, and then uses the provided `ActionSelector` to convert Q-values to indices of actions.

Let's consider a small example. For simplicity, our network always produces the same output for the input batch.

First, we define the NN class, which is supposed to convert observations to actions. In our example, it doesn't use NNs at all and always produces the same output:

```
class DQNNet(nn.Module):
    def __init__(self, actions: int):
        super(DQNNet, self).__init__()
        self.actions = actions

    def forward(self, x):
        # we always produce diagonal tensor of shape
        # (batch_size, actions)
        return torch.eye(x.size()[0], self.actions)
```

Once we have defined the model class, we can use it as a DQN model:

```
>>> net = DQNNNet(actions=3)
>>> net(torch.zeros(2, 10))
tensor([[1., 0., 0.],
       [0., 1., 0.]])
```

We start with the simple argmax policy (which returns the action with the largest value), so the agent will always return actions corresponding to ones in the network output:

```
>>> selector = ptan.actions.ArgmaxActionSelector()
>>> agent = ptan.agent.DQNAgent(model=net, action_selector=selector)
>>> agent(torch.zeros(2, 5))
(array([0, 1]), [None, None])
```

In the input, a batch of two observations, each having five values, was given, and in the output, the agent returned a tuple of two objects:

- An array with actions to be executed for our batch. In our case, this is action `0` for the first batch sample and action `1` for the second.
- A list with the agent's internal state. This is used for stateful agents and is a list of `None` in our case. As our agent is stateless, you can ignore it.

Now, let's make the agent with an epsilon-greedy exploration strategy. To do this, we just need to pass a different action selector:

```
>>> selector = ptan.actions.EpsilonGreedyActionSelector(epsilon=1.0)
>>> agent = ptan.agent.DQNAgent(model=net, action_selector=selector)
>>> agent(torch.zeros(10, 5))[0]
array([2, 0, 0, 0, 1, 2, 1, 2, 2, 1])
```

As `epsilon` is `1.0`, all the actions will be random, regardless of the network's output. But we can change the `epsilon` value on the fly, which is very handy during the training when we anneal `epsilon` over time:

```
>>> selector.epsilon = 0.5
>>> agent(torch.zeros(10, 5))[0]
array([0, 1, 2, 2, 0, 0, 1, 2, 0, 2])
>>> selector.epsilon = 0.1
>>> agent(torch.zeros(10, 5))[0]
array([0, 1, 2, 0, 0, 0, 0, 0, 0, 0])
```

PolicyAgent

PolicyAgent expects the network to produce a policy distribution over a discrete set of actions. The policy distribution could be either logits (unnormalized) or a normalized distribution. In practice, you should always use logits to improve the numeric stability of the training process.

Let's reimplement our previous example, but now, the network will produce a probability.

We begin by defining the following class:

```
class PolicyNet(nn.Module):
    def __init__(self, actions: int):
        super(PolicyNet, self).__init__()
        self.actions = actions

    def forward(self, x):
        # Now we produce the tensor with first two actions
        # having the same logit scores
        shape = (x.size()[0], self.actions)
        res = torch.zeros(shape, dtype=torch.float32)
        res[:, 0] = 1
        res[:, 1] = 1
        return res
```

The class above could be used to get the action logits for a batch of observations:

```
>>> net = PolicyNet(actions=5)
>>> net(torch.zeros(6, 10))
tensor([[1., 1., 0., 0., 0.],
        [1., 1., 0., 0., 0.],
        [1., 1., 0., 0., 0.],
        [1., 1., 0., 0., 0.],
        [1., 1., 0., 0., 0.],
        [1., 1., 0., 0., 0.]])
```

Now, we can use PolicyAgent in combination with ProbabilityActionSelector. As the latter expects normalized probabilities, we need to ask PolicyAgent to apply softmax to the network's output:

```
>>> selector = ptan.actions.ProbabilityActionSelector()
>>> agent = ptan.agent.PolicyAgent(model=net, action_selector=selector,
apply_softmax=True)
>>> agent(torch.zeros(6, 5))[0]
array([2, 1, 2, 0, 2, 3])
```

Please note that the softmax operation produces non-zero probabilities for zero logits, so our agent can still select actions with zero logit values:

```
>>> torch.nn.functional.softmax(torch.tensor([1., 1., 0., 0., 0.]))
tensor([0.3222, 0.3222, 0.1185, 0.1185, 0.1185])
```

Experience source

The agent abstraction described in the previous section allows us to implement environment communications in a generic way. These communications happen in the form of trajectories, produced by applying the agent's actions to the Gym environment.

At a high level, experience source classes take the agent instance and environment and provide you with step-by-step data from the trajectories. The functionality of those classes includes:

- Support for multiple environments being communicated at the same time. This allows efficient GPU utilization as a batch of observations is being processed by the agent at once.
- A trajectory can be preprocessed and presented in a convenient form for further training. For example, there is an implementation of subtrajectory rollouts with accumulation of the reward. That preprocessing is convenient for DQN and n-step DQN, when we are not interested in individual intermediate steps in subtrajectories, so they can be dropped. This saves memory and reduces the amount of code we need to write.
- Support for vectorized environments from Gymnasium (classes `AsyncVectorEnv` and `SyncVectorEnv`). We will cover this in *Chapter 17*.

So, the experience source classes act as a “magic black box” to hide the environment interaction and trajectory handling complexities from the library user. But the overall PTAN philosophy is to be flexible and extensible, so if you want, you can subclass one of the existing classes or implement your own version as needed.

Three classes are provided by the system:

- `ExperienceSource`: Using the agent and the set of environments, it produces n-step subtrajectories with all intermediate steps.
- `ExperienceSourceFirstLast`: This is the same as `ExperienceSource`, but instead of the full subtrajectory (with all steps), it keeps only the first and last steps, with proper reward accumulation in between. This can save a lot of memory in the case of n-step DQN or **advantage actor-critic (A2C)** rollouts.
- `ExperienceSourceRollouts`: This follows the **asynchronous advantage actor-critic (A3C)** rollouts scheme described in Mnih’s paper about Atari games (we will discuss this topic in *Chapter 12*).

All the classes are written to be efficient both in terms of **central processing unit** (CPU) and memory, which is not very important for toy problems, but will become relevant in the next chapter when we get to Atari games with much larger amounts of data to be stored and processed.

Toy environment

For demonstration, we will implement a very simple Gym environment with a small predictable observation state to show how `ExperienceSource` classes work. This environment has integer observation, which increases from 0 to 4, integer action, and a reward equal to the action given. All episodes produced by the environment always have 10 steps:

```
class ToyEnv(gym.Env):
    def __init__(self):
        super(ToyEnv, self).__init__()
        self.observation_space = gym.spaces.Discrete(n=5)
        self.action_space = gym.spaces.Discrete(n=3)
        self.step_index = 0

    def reset(self):
        self.step_index = 0
        return self.step_index, {}

    def step(self, action: int):
        is_done = self.step_index == 10
        if is_done:
            return self.step_index % self.observation_space.n, 0.0, is_done, False, {}
        self.step_index += 1
        return self.step_index % self.observation_space.n, float(action), \
               self.step_index == 10, False, {}
```

In addition to this environment, we will use an agent that always generates fixed actions regardless of observations:

```
class DullAgent(ptan.agent.BaseAgent):
    def __init__(self, action: int):
        self.action = action

    def __call__(self, observations: tt.List[int], state: tt.Optional[list] = None) -> \
                tt.Tuple[tt.List[int], tt.Optional[list]]:
        return [self.action for _ in observations], state
```

Both classes are defined in the `Chapter07/lib.py` module. Now that we have defined the agent, let's talk about the data it produces.

The ExperienceSource class

The first class we will discuss is `ptan.experience.ExperienceSource`, which generates chunks of agent trajectories of the given length. The implementation automatically handles the end-of-episode situation (when the `step()` method in the environment returns `is_done=True`) and resets the environment. The constructor accepts several arguments:

- The Gym environment to be used. Alternatively, it could be the list of environments.
- The agent instance.
- `steps_count=2`: The length of subtrajectories to be generated.

The class instance provides the standard Python iterator interface, so you can just iterate over it to get subtrajectories:

```
>>> from lib import *
>>> env = ToyEnv()
>>> agent = DullAgent(action=1)
>>> exp_source = ptan.experience.ExperienceSource(env=env, agent=agent, steps_count=2)
>>> for idx, exp in zip(range(3), exp_source):
...     print(exp)
...
(Experience(state=0, action=1, reward=1.0, done_trunc=False), Experience(state=1,
action=1, reward=1.0, done_trunc=False))
(Experience(state=1, action=1, reward=1.0, done_trunc=False), Experience(state=2,
action=1, reward=1.0, done_trunc=False))
(Experience(state=2, action=1, reward=1.0, done_trunc=False), Experience(state=3,
action=1, reward=1.0, done_trunc=False))
```

On every iteration, `ExperienceSource` returns a piece of the agent's trajectory in environment communication.

It might look simple, but there are several things happening under the hood of our example:

1. `reset()` was called in the environment to get the initial state.
2. The agent was asked to select the action to execute from the state returned.
3. The `step()` method was executed to get the reward and the next state.
4. This next state was passed to the agent for the next action.
5. Information about the transition from one state to the next state was returned.
6. If the environment returns the end-of-episode flag, we emit the rest of the trajectory and reset the environment to start over.
7. The process continues (from step 3) during the iteration over the experience source.

If the agent changes the way it generates actions (we can get this by updating the network weights, decreasing epsilon, or by some other means), it will immediately affect the experience trajectories that we get.

The `ExperienceSource` instance returns tuples with lengths equal to or less than the `step_count` argument passed on construction. In our case, we asked for two-step subtrajectories, so tuples will be of length 2 or 1 (at the end of episodes). Every object in a tuple is an instance of the `ptan.experience.Experience` class, which is a dataclass with the following fields:

- `state`: The state we observed before taking the action
- `action`: The action we completed
- `reward`: The immediate reward we got from env
- `done_trunc`: Whether the episode was done or truncated

If the episode reaches the end, the subtrajectory will be shorter and the underlying environment will be reset automatically, so we don't need to bother with this and can just keep iterating:

```
>>> for idx, exp in zip(range(15), exp_source):
...     print(exp)
...
(Experience(state=0, action=1, reward=1.0, done_trunc=False), Experience(state=1,
action=1, reward=1.0, done_trunc=False))
.....
(Experience(state=3, action=1, reward=1.0, done_trunc=False), Experience(state=4,
action=1, reward=1.0, done_trunc=True))
(Experience(state=4, action=1, reward=1.0, done_trunc=True),)
(Experience(state=0, action=1, reward=1.0, done_trunc=False), Experience(state=1,
action=1, reward=1.0, done_trunc=False))
(Experience(state=1, action=1, reward=1.0, done_trunc=False), Experience(state=2,
action=1, reward=1.0, done_trunc=False))
```

We can ask `ExperienceSource` for subtrajectories of any length:

```
>>> exp_source = ptan.experience.ExperienceSource(env=env, agent=agent, steps_count=4)
>>> next(iter(exp_source))
(Experience(state=0, action=1, reward=1.0, done_trunc=False), Experience(state=1,
action=1, reward=1.0, done_trunc=False), Experience(state=2, action=1, reward=1.0,
done_trunc=False), Experience(state=3, action=1, reward=1.0, done_trunc=False))
```

We can pass it several instances of `gym.Env`. In that case, they will be used in a round-robin fashion:

```
>>> exp_source = ptan.experience.ExperienceSource(env=[ToyEnv(), ToyEnv()], agent=agent,
steps_count=4)
>>> for idx, exp in zip(range(5), exp_source):
...     print(exp)
...
(Experience(state=0, action=1, reward=1.0, done_trunc=False), Experience(state=1,
action=1, reward=1.0, done_trunc=False), Experience(state=2, action=1, reward=1.0,
done_trunc=False), Experience(state=3, action=1, reward=1.0, done_trunc=False))
(Experience(state=0, action=1, reward=1.0, done_trunc=False), Experience(state=1,
action=1, reward=1.0, done_trunc=False), Experience(state=2, action=1, reward=1.0,
done_trunc=False), Experience(state=3, action=1, reward=1.0, done_trunc=False))
(Experience(state=1, action=1, reward=1.0, done_trunc=False), Experience(state=2,
action=1, reward=1.0, done_trunc=False), Experience(state=3, action=1, reward=1.0,
done_trunc=False), Experience(state=4, action=1, reward=1.0, done_trunc=False))
(Experience(state=1, action=1, reward=1.0, done_trunc=False), Experience(state=2,
action=1, reward=1.0, done_trunc=False), Experience(state=3, action=1, reward=1.0,
done_trunc=False), Experience(state=4, action=1, reward=1.0, done_trunc=False))
(Experience(state=2, action=1, reward=1.0, done_trunc=False), Experience(state=3,
action=1, reward=1.0, done_trunc=False), Experience(state=4, action=1, reward=1.0,
done_trunc=False), Experience(state=0, action=1, reward=1.0, done_trunc=False))
```



Please note that when you're passing several environments to the `ExperienceSource`, they have to be independent instances and not a single environment instance, otherwise your observations will become a mess.

The `ExperienceSourceFirstLast` Class

The `ExperienceSource` class provides us with full subtrajectories of the given length as a list of (s, a, r) objects. The next state, s' , is returned in the next tuple, which is not always convenient. For example, in DQN training, we want to have tuples (s, a, r, s') at once to do one-step Bellman approximation during the training. In addition, some extension of DQN, like n-step DQN, might want to collapse longer sequences of observations into $(first-state, action, total-reward-for-n-steps, state-after-step-n)$.

To support this in a generic way, a simple subclass of `ExperienceSource` is implemented: `ExperienceSourceFirstLast`. It accepts almost the same arguments in the constructor, but returns different data:

```
>>> exp_source = ptan.experience.ExperienceSourceFirstLast(env, agent, gamma=1.0,
steps_count=1)
>>> for idx, exp in zip(range(11), exp_source):
```

```

...     print(exp)
...
ExperienceFirstLast(state=0, action=1, reward=1.0, last_state=1)
ExperienceFirstLast(state=1, action=1, reward=1.0, last_state=2)
ExperienceFirstLast(state=2, action=1, reward=1.0, last_state=3)
ExperienceFirstLast(state=3, action=1, reward=1.0, last_state=4)
ExperienceFirstLast(state=4, action=1, reward=1.0, last_state=0)
ExperienceFirstLast(state=0, action=1, reward=1.0, last_state=1)
ExperienceFirstLast(state=1, action=1, reward=1.0, last_state=2)
ExperienceFirstLast(state=2, action=1, reward=1.0, last_state=3)
ExperienceFirstLast(state=3, action=1, reward=1.0, last_state=4)
ExperienceFirstLast(state=4, action=1, reward=1.0, last_state=None)
ExperienceFirstLast(state=0, action=1, reward=1.0, last_state=1)

```

Now, instead of the tuple, it returns a single object on every iteration, which is again a dataclass with the following fields:

- `state`: The state we used to decide on the action to take.
- `action`: The action we took at this step.
- `reward`: The partial accumulated reward for `steps_count` (in our case, `steps_count=1`, so it is equal to the immediate reward).
- `last_state`: The state we got after executing the action. If our episode ends, we have `None` here.

This data is much more convenient for DQN training, as we can apply Bellman approximation directly to it.

Let's check the result with a larger number of steps:

```

>>> exp_source = ptan.experience.ExperienceSourceFirstLast(env, agent, gamma=1.0,
steps_count=2)
>>> for idx, exp in zip(range(11), exp_source):
...     print(exp)
...
ExperienceFirstLast(state=0, action=1, reward=2.0, last_state=2)
ExperienceFirstLast(state=1, action=1, reward=2.0, last_state=3)
ExperienceFirstLast(state=2, action=1, reward=2.0, last_state=4)
ExperienceFirstLast(state=3, action=1, reward=2.0, last_state=0)
ExperienceFirstLast(state=4, action=1, reward=2.0, last_state=1)
ExperienceFirstLast(state=0, action=1, reward=2.0, last_state=2)
ExperienceFirstLast(state=1, action=1, reward=2.0, last_state=3)
ExperienceFirstLast(state=2, action=1, reward=2.0, last_state=4)
ExperienceFirstLast(state=3, action=1, reward=2.0, last_state=None)
ExperienceFirstLast(state=4, action=1, reward=1.0, last_state=None)
ExperienceFirstLast(state=0, action=1, reward=2.0, last_state=2)

```

So, now we are collapsing two steps on every iteration and calculating the immediate reward (that's why `reward=2.0` for most of the samples). More interesting samples are at the end of the episode:

```
ExperienceFirstLast(state=3, action=1, reward=2.0, last_state=None)
ExperienceFirstLast(state=4, action=1, reward=1.0, last_state=None)
```

As the episode ends, we have `last_state=None` in those samples, but additionally, we calculate the reward for the tail of the episode. Those tiny details are very easy to implement wrongly if you are doing all the trajectory handling yourself.

Experience replay buffers

In DQN, we rarely deal with immediate experience samples, as they are heavily correlated, which leads to instability in the training. Normally, we have large replay buffers, which are populated with experience pieces. Then the buffer is sampled (randomly or with priority weights) to get the training batch. The replay buffer normally has a maximum capacity, so old samples are pushed out when the replay buffer reaches the limit.

There are several implementation tricks here, which become extremely important when you need to deal with large problems:

- How to efficiently sample from a large buffer
- How to push old samples from the buffer
- In the case of a prioritized buffer, how priorities need to be maintained and handled in the most efficient way

All this becomes a quite non-trivial task if you want to deal with Atari games, keeping 10-100M samples, where every sample is an image from the game. A small mistake can lead to a 10-100x memory increase and major slowdowns in the training process.

PTAN provides several variants of replay buffers, which integrate simply with the `ExperienceSource` and `Agent` machinery. Normally, what you need to do is ask the buffer to pull a new sample from the source and sample the training batch. The provided classes are:

- `ExperienceReplayBuffer`: A simple replay buffer of a predefined size with uniform sampling.
- `PrioReplayBufferNaive`: A simple, but not very efficient, prioritized replay buffer implementation. The complexity of sampling is $O(n)$, which might become an issue with large buffers. This version has the advantage over the optimized class, having much easier code. For medium-sized buffers the performance is still acceptable, so we will use it in some examples.
- `PrioritizedReplayBuffer`: Uses segment trees for sampling, which makes the code cryptic, but with $O(\log(n))$ sampling complexity.

The following shows how the replay buffer could be used:

```
>>> exp_source = ptan.experience.ExperienceSourceFirstLast(env, agent, gamma=1.0,
... steps_count=1)
>>> buffer = ptan.experience.ExperienceReplayBuffer(exp_source, buffer_size=100)
>>> len(buffer)
0
>>> buffer.populate(1)
>>> len(buffer)
1
```

All replay buffers provide the following interface:

- A Python iterator interface to walk over all the samples in the buffer
- The `populate(N)` method to get N samples from the experience source and put them into the buffer
- The method `sample(N)` to get the batch of N experience objects

So, the normal training loop for DQN looks like an infinite repetition of the following steps:

1. Call `buffer.populate(1)` to get a fresh sample from the environment.
2. Call `batch = buffer.sample(BATCH_SIZE)` to get the batch from the buffer.
3. Calculate the loss on the sampled batch.
4. Backpropagate.
5. Repeat until convergence (hopefully).

All the rest happens automatically—resetting the environment, handling subtrajectories, buffer size maintenance, and so on:

```
>>> for step in range(6):
...     buffer.populate(1)
...     if len(buffer) < 5:
...         continue
...     batch = buffer.sample(4)
...     print(f"Train time, {len(batch)} batch samples")
...     for s in batch:
...         print(s)
...
Train time, 4 batch samples
ExperienceFirstLast(state=1, action=1, reward=1.0, last_state=2)
ExperienceFirstLast(state=2, action=1, reward=1.0, last_state=3)
ExperienceFirstLast(state=2, action=1, reward=1.0, last_state=3)
ExperienceFirstLast(state=0, action=1, reward=1.0, last_state=1)
Train time, 4 batch samples
```

```
ExperienceFirstLast(state=0, action=1, reward=1.0, last_state=1)
ExperienceFirstLast(state=4, action=1, reward=1.0, last_state=0)
ExperienceFirstLast(state=3, action=1, reward=1.0, last_state=4)
ExperienceFirstLast(state=3, action=1, reward=1.0, last_state=4)
```

The TargetNet class

We mentioned the bootstrapping problem in the previous chapter, when the network used for the next state evaluation becomes influenced by our training process. This was solved by disentangling the currently trained network from the network used for next-state Q-values prediction.

TargetNet is a small but useful class that allows us to synchronize two NNs of the same architecture. This class supports two modes of such synchronization:

- `sync()`: Weights from the source network are copied into the target network.
- `alpha_sync()`: The source network's weights are blended into the target network with some alpha weight (between 0 and 1).

The first mode is the standard way to perform a target network sync in discrete action space problems, like Atari and CartPole, as we did in *Chapter 6*. The latter mode is used in continuous control problems, which will be described in *Part 4* of the book. In such problems, the transition between two networks' parameters should be smooth, so alpha blending is used, given by the formula $w_i = w_i\alpha + s_i(1 - \alpha)$, where w_i is the target network's i -th parameter and s_i is the source network's weight. The following is a small example of how TargetNet should be used in code. Let's assume we have the following network:

```
class DQNNNet(nn.Module):
    def __init__(self):
        super(DQNNNet, self).__init__()
        self.ff = nn.Linear(5, 3)
    def forward(self, x):
        return self.ff(x)
```

The target network could be created as follows:

```
>>> net = DQNNNet()
>>> net
DQNNNet(
  (ff): Linear(in_features=5, out_features=3, bias=True)
)
>>> tgt_net = ptan.agent.TargetNet(net)
```

The target network contains two fields: `model`, which is the reference to the original network, and `target_model`, which is a deep copy of it. If we examine both networks' weights, they will be the same:

```
>>> net.ff.weight
Parameter containing:
tensor([[ 0.2039,  0.1487,  0.4420, -0.0210, -0.2726],
       [-0.2020, -0.0787,  0.2852, -0.1565,  0.4012],
       [-0.0569, -0.4184, -0.3658,  0.4212,  0.3647]], requires_grad=True)
>>> tgt_net.target_model.ff.weight
Parameter containing:
tensor([[ 0.2039,  0.1487,  0.4420, -0.0210, -0.2726],
       [-0.2020, -0.0787,  0.2852, -0.1565,  0.4012],
       [-0.0569, -0.4184, -0.3658,  0.4212,  0.3647]], requires_grad=True)
```

They are independent of each other, however, just having the same architecture:

```
>>> net.ff.weight.data += 1.0
>>> net.ff.weight
Parameter containing:
tensor([[1.2039, 1.1487, 1.4420, 0.9790, 0.7274],
       [0.7980, 0.9213, 1.2852, 0.8435, 1.4012],
       [0.9431, 0.5816, 0.6342, 1.4212, 1.3647]], requires_grad=True)
>>> tgt_net.target_model.ff.weight
Parameter containing:
tensor([[ 0.2039,  0.1487,  0.4420, -0.0210, -0.2726],
       [-0.2020, -0.0787,  0.2852, -0.1565,  0.4012],
       [-0.0569, -0.4184, -0.3658,  0.4212,  0.3647]], requires_grad=True)
```

To synchronize them again, the `sync()` method can be used:

```
>>> tgt_net.sync()
>>> tgt_net.target_model.ff.weight
Parameter containing:
tensor([[1.2039, 1.1487, 1.4420, 0.9790, 0.7274],
       [0.7980, 0.9213, 1.2852, 0.8435, 1.4012],
       [0.9431, 0.5816, 0.6342, 1.4212, 1.3647]], requires_grad=True)
```

For the blended sync, you can use the `alpha_sync()` method:

```
>>> net.ff.weight.data += 1.0
>>> net.ff.weight
Parameter containing:
```

```

tensor([[2.2039, 2.1487, 2.4420, 1.9790, 1.7274],
       [1.7980, 1.9213, 2.2852, 1.8435, 2.4012],
       [1.9431, 1.5816, 1.6342, 2.4212, 2.3647]], requires_grad=True)
>>> tgt_net.target_model.ff.weight
Parameter containing:
tensor([[1.2039, 1.1487, 1.4420, 0.9790, 0.7274],
       [0.7980, 0.9213, 1.2852, 0.8435, 1.4012],
       [0.9431, 0.5816, 0.6342, 1.4212, 1.3647]], requires_grad=True)
>>> tgt_net.alpha_sync(0.1)
>>> tgt_net.target_model.ff.weight
Parameter containing:
tensor([[2.1039, 2.0487, 2.3420, 1.8790, 1.6274],
       [1.6980, 1.8213, 2.1852, 1.7435, 2.3012],
       [1.8431, 1.4816, 1.5342, 2.3212, 2.2647]], requires_grad=True)

```

Ignite helpers

PyTorch Ignite was briefly discussed in *Chapter 3*, and it will be used in the rest of the book to reduce the amount of training loop code. PTAN provides several small helpers to simplify integration with Ignite, which reside in the `ptan.ignite` package:

- `EndOfEpisodeHandler`: Attached to `ignite.Engine`, it emits an `EPISODE_COMPLETED` event and tracks the reward and number of steps in the event in the engine's metrics. It also can emit an event when the average reward for the last episodes reaches the predefined boundary, which is supposed to be used to stop the training when some goal reward has been reached.
- `EpisodeFPSHandler`: Tracks the number of interactions between the agent and environment that are performed and calculates performance metrics as frames per second. It also tracks the number of seconds passed since the start of the training.
- `PeriodicEvents`: Emits corresponding events every 10, 100, or 1,000 training iterations. It is useful for reducing the amount of data being written into TensorBoard.

A detailed illustration of how these classes can be used will be given in the next chapter, when we will use them to reimplement the DQN training from *Chapter 6*, and then check several DQN extensions and tweaks to improve basic DQN convergence.

The PTAN CartPole solver

Let's now take the PTAN classes (without Ignite so far) and try to combine everything to solve our first environment: CartPole. The complete code is in `Chapter07/06_cartpole.py`. I will show only the important parts of the code related to the material that we have just covered.

First, we create the NN (the simple two-layer feed-forward NN that we used for CartPole before) and target the NN epsilon-greedy action selector and DQNAgent. Then, the experience source and replay buffer are created:

```
net = Net(obs_size, HIDDEN_SIZE, n_actions)
tgt_net = ptan.agent.TargetNet(net)
selector = ptan.actions.ArgmaxActionSelector()
selector = ptan.actions.EpsilonGreedyActionSelector(epsilon=1, selector=selector)
agent = ptan.agent.DQNAgent(net, selector)
exp_source = ptan.experience.ExperienceSourceFirstLast(env, agent, gamma=GAMMA)
buffer = ptan.experience.ExperienceReplayBuffer(exp_source, buffer_size=REPLAY_SIZE)
```

With these few lines, we have finished with our data pipeline.

Now, we just need to call `populate()` on the buffer and sample training batches from it:

```
while True:
    step += 1
    buffer.populate(1)

    for reward, steps in exp_source.pop_rewards_steps():
        episode += 1
        print(f"{step}: episode {episode} done, reward={reward:.2f}, "
              f"epsilon={selector.epsilon:.2f}")
        solved = reward > 150
    if solved:
        print("Whee!")
        break
    if len(buffer) < 2*BATCH_SIZE:
        continue
    batch = buffer.sample(BATCH_SIZE)
```

At the beginning of every training loop iteration, we ask the buffer to fetch one sample from the experience source and then check for the finished episode. The `pop_rewards_steps()` method in the `ExperienceSource` class returns the list of tuples with information about episodes completed since the last call to the method.

Later in the training loop, we convert a batch of `ExperienceFirstLast` objects into tensors suitable for DQN training:

```
batch = buffer.sample(BATCH_SIZE)
states_v, actions_v, tgt_q_v = unpack_batch(batch, tgt_net.target_model, GAMMA)
optimizer.zero_grad()
q_v = net(states_v)
q_v = q_v.gather(1, actions_v.unsqueeze(-1)).squeeze(-1)
```

```

loss_v = F.mse_loss(q_v, tgt_q_v)
loss_v.backward()
optimizer.step()
selector.epsilon *= EPS_DECAY

if step % TGT_NET_SYNC == 0:
    tgt_net.sync()

```

We calculate the loss and do a backpropagation step. Finally, we decay epsilon in our action selector (with the hyperparameters used, epsilon decays to zero at training step 500) and ask the target network to sync every 10 training iterations.

The `unpack_batch` method is the last piece of our implementation:

```

@torch.no_grad()
def unpack_batch(batch: tt.List[ExperienceFirstLast], net: Net, gamma: float):
    states = []
    actions = []
    rewards = []
    done_masks = []
    last_states = []
    for exp in batch:
        states.append(exp.state)
        actions.append(exp.action)
        rewards.append(exp.reward)
        done_masks.append(exp.last_state is None)
        if exp.last_state is None:
            last_states.append(exp.state)
        else:
            last_states.append(exp.last_state)

    states_v = torch.as_tensor(np.stack(states))
    actions_v = torch.tensor(actions)
    rewards_v = torch.tensor(rewards)
    last_states_v = torch.as_tensor(np.stack(last_states))
    last_state_q_v = net(last_states_v)
    best_last_q_v = torch.max(last_state_q_v, dim=1)[0]
    best_last_q_v[done_masks] = 0.0
    return states_v, actions_v, best_last_q_v * gamma + rewards_v

```

It takes a sampled batch of `ExperienceFirstLast` objects and converts them into three tensors: states, actions, and target Q-values.

The code should converge in 2,000-3,000 training iterations:

```
Chapter07$ python 06_cartpole.py
26: episode 1 done, reward=25.00, epsilon=1.00
52: episode 2 done, reward=26.00, epsilon=0.82
67: episode 3 done, reward=15.00, epsilon=0.70
80: episode 4 done, reward=13.00, epsilon=0.62
112: episode 5 done, reward=32.00, epsilon=0.45
123: episode 6 done, reward=11.00, epsilon=0.40
139: episode 7 done, reward=16.00, epsilon=0.34
148: episode 8 done, reward=9.00, epsilon=0.31
156: episode 9 done, reward=8.00, epsilon=0.29
...
2481: episode 113 done, reward=58.00, epsilon=0.00
2544: episode 114 done, reward=63.00, epsilon=0.00
2594: episode 115 done, reward=50.00, epsilon=0.00
2786: episode 116 done, reward=192.00, epsilon=0.00
Whee!
```

Other RL libraries

As we discussed earlier, there are several RL-specific libraries available. A few years ago, TensorFlow was more popular than PyTorch, but nowadays, PyTorch is dominating the field, and there is a recent trend of JAX being used as it provides better performance. The following is my recommended list of libraries you might want to take into consideration for your projects:

- **stable-baselines3**: We mentioned this library when we discussed Atari wrappers. This is a fork of the OpenAI Stable Baselines repository, and the main idea is to have an optimized and reproducible set of RL algorithms that you can use to check your methods (<https://github.com/DLR-RM/stable-baselines3>).
- **TorchRL**: RL extensions for PyTorch. This library is relatively young—the first release was at the end of 2022—but provides rich set of helper classes for RL. Its design philosophy is very close to PTAN—a Python-first set of flexible classes that you can combine and extend to build your system—so I highly recommend that you learn this library. In the rest of the book, we’ll use this library’s classes. Most likely, examples in the next edition of this book (unless we reach “AI Singularity” and books become obsolete, like clay tablets) will not be based on PTAN but on TorchRL, which is better maintained. Documentation: <https://pytorch.org/rl/>, source code: <https://github.com/pytorch/rl>.

- **Spinning Up:** Another repo from OpenAI, but with a different goal in mind: providing valuable and clean education materials about state-of-the-art methods. This repo hasn't been updated for several years (the last commit was in 2020), but still provides very valuable materials about the methods. Documentation: <https://spinningup.openai.com/>. Code: <https://github.com/openai/spinningup>.
- **Keras-RL:** Started by Matthias Plappert in 2016, this includes basic deep RL methods. As suggested by the name, this library was implemented using Keras, which is a high-level wrapper around TensorFlow (<https://github.com/keras-rl/keras-rl>). Unfortunately, the last commit was in 2019, so the project has been abandoned.
- **Dopamine:** A library from Google published in 2018. It is TensorFlow-specific, which is not surprising for a library from Google (<https://github.com/google/dopamine>).
- **Ray:** A library for distributed execution of machine learning code. It includes RL utilities as part of the library (<https://github.com/ray-project/ray>).
- **TF-Agents:** Another library from Google published in 2018 (<https://github.com/tensorflow/agents>).
- **ReAgent:** A library from Facebook Research. It uses PyTorch internally and uses a declarative style of configuration (when you are creating a JSON file to describe your problem), which limits extensibility. But, of course, as it is open source, you can always extend the functionality (<https://github.com/facebookresearch/ReAgent>). Recently, ReAgent was archived and replaced by the **Pearl** library from the same team: <https://github.com/facebookresearch/Pearl/>.

Summary

In this chapter, we talked about higher-level RL libraries, their motivation, and their requirements. Then, we took a deep look at the PTAN library, which will be used in the rest of the book to simplify example code. This focus on the details of the methods rather than implementation will be extremely useful for you in later chapters of this book, as you progress further with RL.

In the next chapter, we will return to DQN methods by exploring extensions that researchers and practitioners have discovered since the classic DQN introduction to improve the stability and performance of the method.

8

DQN Extensions

Since DeepMind published its paper on the **deep Q-network (DQN)** model in 2015, many improvements have been proposed, along with tweaks to the basic architecture, which, significantly, have improved the convergence, stability, and sample efficiency of DeepMind's basic DQN. In this chapter, we will take a deeper look at some of those ideas.

In October 2017, Hessel et al. from DeepMind published a paper called *Rainbow: Combining improvements in deep reinforcement learning* [Hes+18], which presented the six most important improvements to DQN; some were invented in 2015, but others are relatively recent. In this paper, state-of-the-art results on the Atari games suite were reached, just by combining those six methods.



Since 2017, more papers have been published and state-of-the-art results have been pushed further, but all the methods presented in the paper are still relevant and widely used in practice. For example, in 2023, Marc Bellemare published the book *Distributional reinforcement learning* [BDR23] about one of the paper's methods. In addition, the improvements described are relatively simple to implement and understand, so I have not made any major modifications to this chapter in this edition.

The DQN extensions that we will become familiar with are the following:

- **N-step DQN:** How to improve convergence speed and stability with a simple unrolling of the Bellman equation, and why it's not an ultimate solution

- **Double DQN:** How to deal with DQN overestimation of the values of the actions
- **Noisy networks:** How to make exploration more efficient by adding noise to the network weights
- **Prioritized replay buffer:** Why uniform sampling of our experience is not the best way to train
- **Dueling DQN:** How to improve convergence speed by making our network's architecture more closely represent the problem that we are solving
- **Categorical DQN:** How to go beyond the single expected value of the action and work with full distributions

This chapter will go through all these methods. We will analyze the ideas behind them, alongside how they can be implemented and compared to the classic DQN performance. Finally, we will analyze how the combined system with all the methods performs.

Basic DQN

To get started, we will implement the same DQN method as in *Chapter 6*, but leveraging the high-level primitives described in *Chapter 7*. This will make our code much more compact, which is good, as non-relevant details won't distract us from the method's logic. At the same time, the purpose of this book is not to teach you how to use the existing libraries but rather how to develop intuition about RL methods and, if necessary, implement everything from scratch. From my perspective, this is a much more valuable skill, as libraries come and go, but true understanding of the domain will allow you to quickly make sense of other people's code and apply it consciously.

In the basic DQN implementation, we have three modules in the `Chapter08` folder of the GitHub repository for this book:

- `Chapter08/lib/dqn_model.py`: The DQN **neural network (NN)**, which is the same as in *Chapter 6*, so I won't repeat it
- `Chapter08/lib/common.py`: Common functions and declarations shared by the code in this chapter
- `Chapter08/01_dqn_basic.py`: 77 lines of code leveraging the PTAN and Ignite libraries, implementing the basic DQN method

Common library

Let's start with the contents of `lib/common.py`. First of all, we have hyperparameters for our Pong environment from the previous chapter. The hyperparameters are stored in the `dataclass` object, which is a standard way to store a bunch of data fields with their type annotations. This makes it easy to add another configuration set for different, more complicated Atari games and allows us to experiment with hyperparameters:

```
@dataclasses.dataclass
class Hyperparams:
    env_name: str
    stop_reward: float
    run_name: str
    replay_size: int
    replay_initial: int
    target_net_sync: int
    epsilon_frames: int

    learning_rate: float = 0.0001
    batch_size: int = 32
    gamma: float = 0.99
    epsilon_start: float = 1.0
    epsilon_final: float = 0.1

    tuner_mode: bool = False
    episodes_to_solve: int = 500

GAME_PARAMS = {
    'pong': Hyperparams(
        env_name="PongNoFrameskip-v4",
        stop_reward=18.0,
        run_name="pong",
        replay_size=100_000,
        replay_initial=10_000,
        target_net_sync=1000,
        epsilon_frames=100_000,
        epsilon_final=0.02,
    ),
}
```

The next function from `lib/common.py` has the name `unpack_batch`, and it takes the batch, of transitions and converts it into the set of NumPy arrays suitable for training. Every transition from `ExperienceSourceFirstLast` has a type of `ExperienceFirstLast`, which is a dataclass with the following fields:

- `state`: Observation from the environment.
- `action`: Integer action taken by the agent.
- `reward`: If we have created `ExperienceSourceFirstLast` with the attribute `steps_count=1`, it's just the immediate reward. For larger step counts, it contains the discounted sum of rewards for this number of steps.
- `last_state`: If the transition corresponds to the final step in the environment, then this field is `None`; otherwise, it contains the last observation in the experience chain.

The code of `unpack_batch` is as follows:

```
def unpack_batch(batch: tt.List[ExperienceFirstLast]):
    states, actions, rewards, dones, last_states = [],[],[],[],[]
    for exp in batch:
        states.append(exp.state)
        actions.append(exp.action)
        rewards.append(exp.reward)
        dones.append(exp.last_state is None)
        if exp.last_state is None:
            lstate = exp.state # the result will be masked anyway
        else:
            lstate = exp.last_state
        last_states.append(lstate)
    return np.asarray(states), np.array(actions), np.array(rewards, dtype=np.float32), \
        np.array(dones, dtype=bool), np.asarray(last_states)
```

Note how we handle the final transitions in the batch. To avoid the special handling of such cases, for terminal transitions, we store the initial state in the `last_states` array. To make our calculations of the Bellman update correct, we have to mask such batch entries during the loss calculation using the `dones` array. Another solution would be to calculate the value of the last states only for non-terminal transitions, but it would make our loss function logic a bit more complicated.

Calculation of the DQN loss function is provided by the `calc_loss_dqn` function, and the code is almost the same as in *Chapter 6*. One small addition is `torch.no_grad()`, which stops the PyTorch calculation graph from being recorded for the target net:

```
def calc_loss_dqn(
    batch: tt.List[ExperienceFirstLast], net: nn.Module, tgt_net: nn.Module,
    gamma: float, device: torch.device) -> torch.Tensor:
    states, actions, rewards, dones, next_states = unpack_batch(batch)

    states_v = torch.as_tensor(states).to(device)
    next_states_v = torch.as_tensor(next_states).to(device)
    actions_v = torch.tensor(actions).to(device)
    rewards_v = torch.tensor(rewards).to(device)
    done_mask = torch.BoolTensor(dones).to(device)

    actions_v = actions_v.unsqueeze(-1)
    state_action_vals = net(states_v).gather(1, actions_v)
    state_action_vals = state_action_vals.squeeze(-1)
    with torch.no_grad():
        next_state_vals = tgt_net(next_states_v).max(1)[0]
        next_state_vals[done_mask] = 0.0
```

```

bellman_vals = next_state_vals.detach() * gamma + rewards_v
return nn.MSELoss()(state_action_vals, bellman_vals)

```

Besides those core DQN functions, `common.py` provides several utilities related to our training loop, data generation, and TensorBoard tracking. The first such utility is a small class that implements epsilon decay during the training. Epsilon defines the probability of taking the random action by the agent. It should be decayed from 1.0 in the beginning (fully random agent) to some small number, like 0.02 or 0.01. The code is trivial but is needed in almost any DQN, so it is provided by the following little class:

```

class EpsilonTracker:
    def __init__(self, selector: EpsilonGreedyActionSelector, params: Hyperparams):
        self.selector = selector
        self.params = params
        self.frame(0)

    def frame(self, frame_idx: int):
        eps = self.params.epsilon_start - frame_idx / self.params.epsilon_frames
        self.selector.epsilon = max(self.params.epsilon_final, eps)

```

Another small function is `batch_generator`, which takes `ExperienceReplayBuffer` (the PTAN class described in *Chapter 7*) and infinitely generates training batches sampled from the buffer. In the beginning, the function ensures that the buffer contains the required amount of samples:

```

def batch_generator(buffer: ExperienceReplayBuffer, initial: int, batch_size: int) -> \
    tt.Generator[tt.List[ExperienceFirstLast], None, None]:
    buffer.populate(initial)
    while True:
        buffer.populate(1)
        yield buffer.sample(batch_size)

```

Finally, a lengthy, but nevertheless very useful, function called `setup_ignite` attaches the needed Ignite handlers, showing the training progress and writing metrics to TensorBoard. Let's look at this function piece by piece:

```

def setup_ignite(
    engine: Engine, params: Hyperparams, exp_source: ExperienceSourceFirstLast,
    run_name: str, extra_metrics: tt.Iterable[str] = (),
    tuner_reward_episode: int = 100, tuner_reward_min: float = -19,
):

```

```

handler = ptan_ignite.EndOfEpisodeHandler(
    exp_source, bound_avg_reward=params.stop_reward)
handler.attach(engine)
ptan_ignite.EpisodeFPSHandler().attach(engine)

```

Initially, `setup_ignite` attaches two Ignite handlers provided by PTAN:

- `EndOfEpisodeHandler`, which emits the Ignite event every time a game episode ends. It can also fire an event when the averaged reward for episodes crosses some boundary. We use this to detect when the game is finally solved.
- `EpisodeFPSHandler`, a small class that tracks the time the episode has taken and the amount of interactions that we have had with the environment. From this, we calculate **frames per second (FPS)**, which is an important performance metric to track.

Then, we install two event handlers:

```

@engine.on(ptan_ignite.EpisodeEvents.EPISODE_COMPLETED)
def episode_completed(trainer: Engine):
    passed = trainer.state.metrics.get('time_passed', 0)
    print("Episode %d: reward=%.0f, steps=%s, speed=%.1f f/s, elapsed=%s" %
        (trainer.state.episode, trainer.state.episode_reward,
         trainer.state.episode_steps, trainer.state.metrics.get('avg_fps', 0),
         timedelta(seconds=int(passed))))

@engine.on(ptan_ignite.EpisodeEvents.BOUND_REWARD_REACHED)
def game_solved(trainer: Engine):
    passed = trainer.state.metrics['time_passed']
    print("Game solved in %s, after %d episodes and %d iterations!" %
        (timedelta(seconds=int(passed)), trainer.state.episode,
         trainer.state.iteration))
    trainer.should_terminate = True
    trainer.state.solved = True

```

One of the event handlers is called at the end of an episode. It will show information about the completed episode on the console. Another function will be called when the average reward grows above the boundary defined in the hyperparameters (18.0 in the case of Pong). This function shows a message about the solved game and stops the training.

The rest of the function is related to the TensorBoard data that we want to track. First, we create a `TensorboardLogger`:

```
now = datetime.now().isoformat(timespec='minutes').replace(':', '')
logdir = f"runs/{now}-{params.run_name}-{run_name}"
tb = tb_logger.TensorboardLogger(log_dir=logdir)
run_avg = RunningAverage(output_transform=lambda v: v['loss'])
run_avg.attach(engine, "avg_loss")
```

This is a special class provided by Ignite to write into TensorBoard. Our processing function will return the loss value, so we attach the `RunningAverage` transformation (also provided by Ignite) to get a smoothed version of the loss over time.

Next, we attach the metrics we want to track to the Ignite events:

```
metrics = ['reward', 'steps', 'avg_reward']
handler = tb_logger.OutputHandler(tag="episodes", metric_names=metrics)
event = ptan_ignite.EpisodeEvents.EPISODE_COMPLETED
tb.attach(engine, log_handler=handler, event_name=event)
```

`TensorboardLogger` can track two groups of values from Ignite: outputs (values returned by the transformation function) and metrics (calculated during the training and kept in the engine state). `EndOfEpisodeHandler` and `EpisodeFPSHandler` provide metrics, which are updated at the end of every game episode. So, we attach `OutputHandler`, which will write into TensorBoard information about the episode every time it is completed.

Next, we track another group of values, metrics from the training process: loss, FPS, and, possibly, some custom metrics relevant to the specific extension's logic:

```
ptan_ignite.PeriodicEvents().attach(engine)
metrics = ['avg_loss', 'avg_fps']
metrics.extend(extra_metrics)
handler = tb_logger.OutputHandler(tag="train", metric_names=metrics,
                                  output_transform=lambda a: a)
event = ptan_ignite.PeriodEvents.ITERS_100_COMPLETED
tb.attach(engine, log_handler=handler, event_name=event)
```

Those values are updated every training iteration, but we are going to do millions of iterations, so we will store values in TensorBoard every 100 training iterations; otherwise, the data files will be huge. All this functionality might look too complicated, but it provides us with the unified set of metrics gathered from the training process. In fact, Ignite is not very tricky, given the flexibility it provides. That's it for `common.py`.

Implementation

Now, let's take a look at `01_dqn_basic.py`, which creates the needed classes and starts the training. I'm going to omit non-relevant code and focus only on important pieces (the full version is available in the GitHub repo). First, we create the environment:

```
env = gym.make(params.env_name)
env = ptan.common.wrappers.wrap_dqn(env)

net = dqn_model.DQN(env.observation_space.shape, env.action_space.n).to(device)
tgt_net = ptan.agent.TargetNet(net)
```

Here, we apply a set of standard wrappers. We discussed them in *Chapter 6* and will also touch upon them in the next chapter, when we optimize the performance of the Pong solver. Then, we create the DQN model and the target network.

Next, we create the agent, passing it an epsilon-greedy action selector:

```
selector = ptan.actions.EpsilonGreedyActionSelector(epsilon=params.epsilon_start)
epsilon_tracker = common.EpsilonTracker(selector, params)
agent = ptan.agent.DQNAgent(net, selector, device=device)
```

During the training, epsilon will be decreased by the `EpsilonTracker` class that we have already discussed. This will decrease the amount of randomly selected actions and give more control to our NN.

The next two very important objects are `ExperienceSourceFirstLast` and `ExperienceReplayBuffer`:

```
exp_source = ptan.experience.ExperienceSourceFirstLast(
    env, agent, gamma=params.gamma, env_seed=common.SEED)
buffer = ptan.experience.ExperienceReplayBuffer(
    exp_source, buffer_size=params.replay_size)
```

`ExperienceSourceFirstLast` takes the agent and environment and provides transitions over game episodes. Those transitions will be kept in the experience replay buffer.

Then we create an optimizer and define the processing function:

```
optimizer = optim.Adam(net.parameters(), lr=params.learning_rate)

def process_batch(engine, batch):
    optimizer.zero_grad()
```

```

        loss_v = common.calc_loss_dqn(batch, net, tgt_net.target_model,
                                      gamma=params.gamma, device=device)
        loss_v.backward()
        optimizer.step()
        epsilon_tracker.frame(engine.state.iteration)
        if engine.state.iteration % params.target_net_sync == 0:
            tgt_net.sync()
    return {
        "loss": loss_v.item(),
        "epsilon": selector.epsilon,
    }
}

```

The processing function will be called for every batch of transitions to train the model. To do this, we call the `common.calc_loss_dqn` function and then backpropagate on the result. This function also asks `EpsilonTracker` to decrease the epsilon and does periodical target network synchronization.

And, finally, we create the Ignite Engine object:

```

engine = Engine(process_batch)
common.setup_ignite(engine, params, exp_source, NAME)
engine.run(common.batch_generator(buffer, params.replay_initial, params.batch_size))

```

We configure it using a function from `common.py`, and run our training process.

Hyperparameter tuning

To make our comparison of DQN extensions fair, we also need to tune hyperparameters. This is essential because even for the same game (Pong), using the fixed set of training parameters might give less optimal results when we change the details of the method.

In principle, every explicit or implicit constant in our code could be tuned, such as:

- Network configuration: Amount and size of layers, activation function, dropout, etc.
- Optimization parameters: Method (vanilla SGD, Adam, AdaGrad, etc.), learning rate, and other optimizer parameters
- Exploration parameters: Decay rate of ϵ , final ϵ value
- Discount factor γ in Bellman equation

But every new parameter we tune has a multiplicative effect on the amount of trial training we need to perform, so having too many hyperparameters might require hundreds or even thousands of trainings. Large companies like Google and Meta have access to a much larger amount of GPUs than individual researchers like us, so we need to keep the balance there.

In my case, I'm going to demonstrate how hyperparameter tuning is done in general, but we'll do the search only on a few values:

- Learning rate
- Discount factor γ
- Parameters specific to the DQN extension we're considering

There are several libraries that might be helpful with hyperparameter tuning. Here, I'm using Ray Tune (<https://docs.ray.io/en/latest/tune/index.html>), which is a part of the Ray project — a distributed computing framework for ML and DL. At a high level, you need to define:

- The hyperparameter space you want to explore (boundaries for values to sample from or an explicit list of values to try)
- The function that performs the training with specific values of hyperparameters and returns the metric you want to optimize with the tuning

This might look very similar to ML problems, and in fact it is — this is also an optimization problem. But there are substantial differences: the function we're optimizing is not differentiable (so you cannot perform the gradient descent to push your hyperparameters towards the desired direction of the metric) and the optimization space might be discrete (you cannot train the network with the number of layers equal to 2.435, for example, since we cannot take the derivative of a non-smooth function).

In later chapters, we'll touch on this problem slightly in the context of black-box optimization methods (*Chapter 17*) and RL in discrete optimizations (*Chapter 21*), but for now, we'll use the simplest approach — a random search of hyperparameters. In this case, the `ray.tune` library randomly samples concrete parameters several times and calls the function to obtain the metric. The smallest (or highest) metric corresponds to the best hyperparameter combination found in this run.

In this chapter, our metric (optimization objective) will be the *number of games the agent needs to play* before solving the game (reaching a mean score of greater than 18 for Pong).

To illustrate the effect of tuning, for every DQN extension, we check the training dynamics using a fixed set of parameters (the same as in *Chapter 6*), and the dynamics using the best hyperparameters found after 20-30 rounds of tuning. If you wish, you can do your own experiments, optimizing more hyperparameters. Most likely, this will allow you to find a better configuration for the training.

The core of the process is implemented in the `common.tune_params` function. Let's take a look at its code.

We start with the type declaration and hyperparameter space:

```
TrainFunc = tt.Callable[
    [Hyperparams, torch.device, dict],
    tt.Optional[int]
]

BASE_SPACE = {
    "learning_rate": tune.loguniform(1e-5, 1e-4),
    "gamma": tune.choice([0.9, 0.92, 0.95, 0.98, 0.99, 0.995]),
}
```

Here, we first define the type for the training function, which takes the `Hyperparams` dataclass `torch.device` to use and a dictionary with extra parameters (as some DQN extensions we're going to present might require extra parameters besides those declared in `Hyperparams`).

The result of the function is either the `int` value, which will be the amount of games we played before reaching the score of 18, or `None` if we decided to stop the training early. This is required, as some hyperparameter combinations might fail to converge or converge too slowly, so to save time we stop the training without waiting for too long.

Then we define the hyperparameter search space – which is a `dict` with string keys (parameter name) and the `tune` declaration of possible values to explore. It could be a probability distribution (uniform, loguniform, normal, etc.) or an explicit list of values to try. You can also use `tune.grid_search` declaration with a list of values. In that case, all the values will be tried.

In our case, we sample the learning rate from the loguniform distribution and gamma from the list of 6 values ranging from 0.9 to 0.995.

Next, we have the `tune_params` function:

```
def tune_params(
    base_params: Hyperparams, train_func: TrainFunc, device: torch.device,
    samples: int = 10, extra_space: tt.Optional[tt.Dict[str, tt.Any]] = None,
):
    search_space = dict(BASE_SPACE)
    if extra_space is not None:
        search_space.update(extra_space)
    config = tune.TuneConfig(num_samples=samples)

    def objective(config: dict, device: torch.device) -> dict:
        keys = dataclasses.asdict(base_params).keys()
        upd = {"tuner_mode": True}
```

```

for k, v in config.items():
    if k in keys:
        upd[k] = v
params = dataclasses.replace(base_params, **upd)
res = train_func(params, device, config)
return {"episodes": res if res is not None else 10**6}

```

This function is given the following arguments:

- Basic set of hyperparameters that will be used for training
- Training function
- Torch device to use
- Amount of samples to perform during the round
- Additional dictionary with search space

Inside this function, we have an objective function, which creates the `Hyperparameters` object from the sampled dict, calls the training function, and returns the dictionary (which is a requirement of the `ray.tune` library).

The rest of the `tune_params` function is simple:

```

obj = tune.with_parameters(objective, device=device)
if device.type == "cuda":
    obj = tune.with_resources(obj, {"gpu": 1})
tuner = tune.Tuner(obj, param_space=search_space, tune_config=config)
results = tuner.fit()
best = results.get_best_result(metric="episodes", mode="min")
print(best.config)
print(best.metrics)

```

Here, we wrap the objective function to pass the torch device and take into account GPU resources. This is needed to allow Ray to properly parallelize the tuning process. If you have multiple GPUs installed on the machine, it will run several trainings in parallel. Then, we just create the `Tuner` object and ask it to perform the hyperparameter search.

The final piece relevant to hyperparameter tuning is in the `setup_ignite` function. It checks for situations when the training process is not converging, so we stop the training to avoid infinite waiting.

To do this, we install the Ignite event handler if we're in the hyperparameter tuning mode:

```
if params.tuner_mode:
    @engine.on(ptan_ignite.EpisodeEvents.EPISODE_COMPLETED)
    def episode_completed(trainer: Engine):
        avg_reward = trainer.state.metrics.get('avg_reward')
        max_episodes = params.episodes_to_solve * 1.1
        if trainer.state.episode > tuner_reward_episode and \
            avg_reward < tuner_reward_min:
            trainer.should_terminate = True
            trainer.state.solved = False
        elif trainer.state.episode > max_episodes:
            trainer.should_terminate = True
            trainer.state.solved = False
        if trainer.should_terminate:
            print(f"Episode {trainer.state.episode}, "
                  f"avg_reward {avg_reward:.2f}, terminating")
```

Here, we check for two conditions:

- If the mean reward is lower than `tuner_reward_min` (which is an argument to the `setup_ignite` function and equal to -19 by default) after 100 games (provided in the `tuner_reward_episode` argument). This means that it's quite unlikely that we'll converge at all.
- We played more than `max_episodes` amount of games and still haven't solved the game. In the default config, we set this limit to 500 games.

In both cases, we stop the training and set the `solved` attribute to `False`, which will return a high constant metric value in our tuning process.

That's it for the hyperparameter tuning code. Before we run it and check the results, let's first start a single training using the parameters we used in *Chapter 6*.

Results with common parameters

If we run the training with the argument `-params common`, we'll train the Pong game using hyperparameters from the `common.py` module. As an option, you can use the `-params best` command line to train on the best values for this particular DQN extension.

Okay, let's start the training using the following command:

```
Chapter08$ ./01_dqn_basic.py --dev cuda --params common
A.L.E: Arcade Learning Environment (version 0.8.1+53f58b7)
```

```
[Powered by Stella]
Episode 1: reward=-21, steps=848, speed=0.0 f/s, elapsed=0:00:11
Episode 2: reward=-21, steps=850, speed=0.0 f/s, elapsed=0:00:11
Episode 3: reward=-19, steps=1039, speed=0.0 f/s, elapsed=0:00:11
Episode 4: reward=-21, steps=884, speed=0.0 f/s, elapsed=0:00:11
Episode 5: reward=-19, steps=1146, speed=0.0 f/s, elapsed=0:00:11
Episode 6: reward=-20, steps=997, speed=0.0 f/s, elapsed=0:00:11
Episode 7: reward=-21, steps=972, speed=0.0 f/s, elapsed=0:00:11
Episode 8: reward=-21, steps=882, speed=0.0 f/s, elapsed=0:00:11
Episode 9: reward=-21, steps=898, speed=0.0 f/s, elapsed=0:00:11
Episode 10: reward=-20, steps=947, speed=0.0 f/s, elapsed=0:00:11
Episode 11: reward=-21, steps=762, speed=227.7 f/s, elapsed=0:00:12
Episode 12: reward=-20, steps=991, speed=227.8 f/s, elapsed=0:00:17
Episode 13: reward=-21, steps=762, speed=227.9 f/s, elapsed=0:00:20
Episode 14: reward=-20, steps=948, speed=227.9 f/s, elapsed=0:00:24
Episode 15: reward=-20, steps=992, speed=228.0 f/s, elapsed=0:00:28
....
```

Every line in the output is written at the end of the game episode, showing the episode reward, a count of steps, the speed, and the total training time. For the basic DQN version and common hyperparameters, it usually takes about 700K frames and about 400 games to reach the mean reward of 18, so be patient. During the training, we can check the dynamics of the training process in TensorBoard, which shows charts for epsilon, raw reward values, average reward, and speed. The following charts show the reward and the number of steps for episodes (the bottom x axis shows the wall clock time, and the top axis is the episode number):

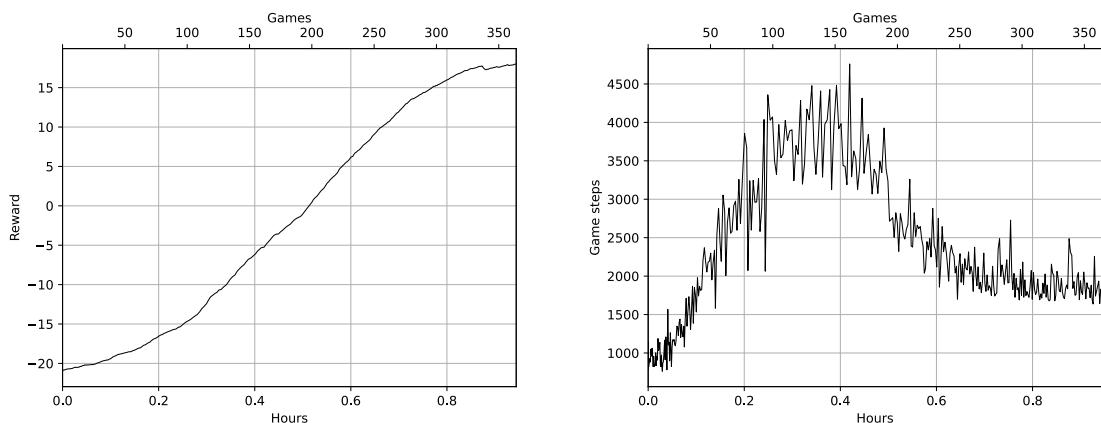


Figure 8.1: Plots with reward (left) and count of steps per episode (right)

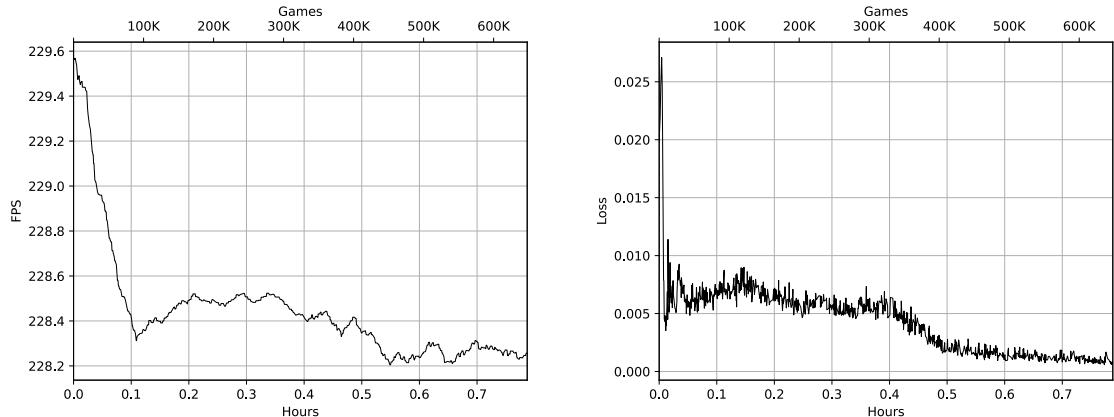


Figure 8.2: Plots with training speed (left) and average training loss (right)

It is also worth noting how the count of steps per episode changes during the training. Initially, it increases, as our network starts winning more and more games, but after a certain level, the count of steps decreases 2x and stays almost constant. This is driven by our γ parameter, which discounts the agent's reward over time, so it tries not just to accumulate as much of a reward as possible, but also to do it *efficiently*.

Tuned baseline DQN

After running the baseline DQN with the command-line argument `-tune 30` (which took about a day on one GPU), I was able to find the following parameters, which solves Pong in 340 episodes (instead of 360):

```
learning_rate=9.932831968547505e-05,
gamma=0.98,
```

As you can see, the learning rate is almost the same as before (10^{-4}), but gamma is lower (0.98 versus 0.99). This might be an indication that Pong has relatively short subtrajectories with action-reward causality, so decreasing the γ has a stabilizing effect on the training.

In the following figure, you can see a comparison of the reward and steps per episode for both tuned and untuned versions (and the difference is quite minor):

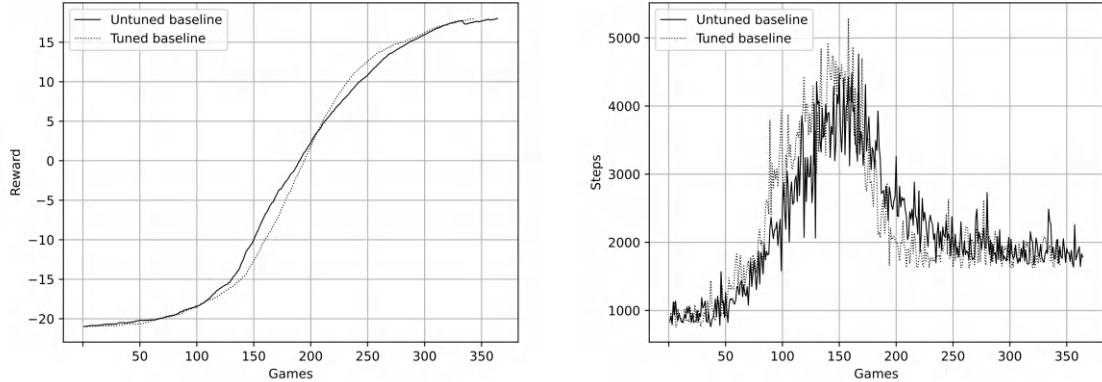


Figure 8.3: Plots with reward (left) and count of steps per episode (right) for tuned and untuned hyperparameters

Now we have our baseline DQN version and are ready to explore method modifications proposed by Hessel et al.

N-step DQN

The first improvement that we will implement and evaluate is quite an old one. It was first introduced by Sutton in the paper *Learning to Predict by the Methods of Temporal Differences* [Sut88]. To get the idea, let's look at the Bellman update used in Q-learning once again:

$$Q(s_t, a_t) = r_t + \gamma \max_a Q(s_{t+1}, a_{t+1})$$

This equation is recursive, which means that we can express $Q(s_{t+1}, a_{t+1})$ in terms of itself, which gives us this result:

$$Q(s_t, a_t) = r_t + \gamma \max_a [r_{a,t+1} + \gamma \max_{a'} Q(s_{t+2}, a')]$$

Value $r_{a,t+1}$ means local reward at time $t + 1$, after issuing action a . However, if we assume that action a at step $t + 1$ was chosen optimally, or close to optimally, we can omit the \max_a operation and obtain this:

$$Q(s_t, a_t) = r_t + \gamma r_{t+1} + \gamma^2 \max_{a'} Q(s_{t+2}, a')$$

This value can be unrolled again and again any number of times. As you may guess, this unrolling can be easily applied to our DQN update by replacing one-step transition sampling with longer transition sequences of n-steps. To understand why this unrolling will help us to speed up training, let's consider the example illustrated in *Figure 8.4*. Here, we have a simple environment of four states (s_1, s_2, s_3, s_4) and the only action available at every state, except s_4 , which is a terminal state:

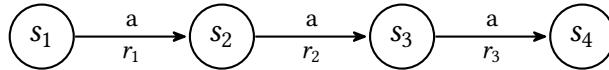


Figure 8.4: A transition diagram for a simple environment

So, what happens in a one-step case? We have three total updates possible (we don't use max, as there is only one action available):

1. $Q(s_1, a) \leftarrow r_1 + \gamma Q(s_2, a)$
2. $Q(s_2, a) \leftarrow r_2 + \gamma Q(s_3, a)$
3. $Q(s_3, a) \leftarrow r_3$

Let's imagine that, at the beginning of the training, we complete the preceding updates in this order. The first two updates will be useless, as our current $Q(s_2, a)$ and $Q(s_3, a)$ are incorrect and contain initial random values. The only useful update will be update 3, which will correctly assign reward r_3 to the state s_3 prior to the terminal state.

Now let's perform the updates over and over again. On the second iteration, the correct value will be assigned to $Q(s_2, a)$, but the update of $Q(s_1, a)$ will still be noisy. Only on the third iteration will we get the valid values for every Q . So, even in a one-step case, it takes three steps to propagate the correct values to all the states.

Now let's consider a two-step case. This situation again has three updates:

1. $Q(s_1, a) \leftarrow r_1 + \gamma r_2 + \gamma^2 Q(s_3, a)$
2. $Q(s_2, a) \leftarrow r_2 + \gamma r_3$
3. $Q(s_3, a) \leftarrow r_3$

In this case, on the first loop over the updates, the correct values will be assigned to both $Q(s_2, a)$ and $Q(s_3, a)$. On the second iteration, the value of $Q(s_1, a)$ will also be properly updated. So, multiple steps improve the propagation speed of values, which improves convergence. You may be thinking, "If it's so helpful, let's unroll the Bellman equation, say, 100 steps ahead. Will it speed up our convergence 100 times?" Unfortunately, the answer is no. Despite our expectations, our DQN will fail to converge at all.

To understand why, let's again return to our unrolling process, especially where we dropped the \max_a . Was it correct? Strictly speaking, no.

We omitted the *max* operation at the intermediate step, assuming that our action selection during experience gathering (or our policy) was optimal. What if it wasn't, for example, at the beginning of the training, when our agent acted randomly? In that case, our calculated value for $Q(s_t, a_t)$ may be smaller than the optimal value of the state (as some steps have been taken randomly, but not following the most promising paths by maximizing the Q-value). The more steps on which we unroll the Bellman equation, the more incorrect our update could be.

Our large experience replay buffer will make the situation even worse, as it will increase the chance of getting transitions obtained from the old bad policy (dictated by old bad approximations of Q). This will lead to a wrong update of the current Q approximation, so it can easily break our training progress. This problem is a fundamental characteristic of RL methods, as was briefly mentioned in *Chapter 4*, when we talked about RL methods' taxonomy.

There are two large classes of methods:

- **Off-policy methods:** The first class of off-policy methods doesn't depend on the "freshness of data." For example, a simple DQN is off-policy, which means that we can use very old data sampled from the environment several million steps ago, and this data will still be useful for learning. That's because we are just updating the value of the action, $Q(s_t, a_t)$, with the immediate reward, plus the discounted current approximation of the best action's value. Even if action a_t was sampled randomly, it doesn't matter because for this particular action a_t , in the state s_t , our update will be correct. That's why in off-policy methods, we can use a very large experience buffer to make our data closer to being **independent and identically distributed (iid)**.
- **On-policy methods:** On the other hand, on-policy methods heavily depend on the training data to be sampled according to the current policy that we are updating. That happens because on-policy methods try to improve the current policy indirectly (as in the previous n-step DQN) or directly (all of *Part 3* of the book is devoted to such methods).

So, which class of methods is better? Well, it depends. Off-policy methods allow you to train on the previous large history of data or even on human demonstrations, but they usually are slower to converge. On-policy methods are typically faster, but require much more fresh data from the environment, which can be costly. Just imagine a self-driving car trained with the on-policy method. It will cost you a lot of crashed cars before the system learns that walls and trees are things that it should avoid!

You may have a question: why are we talking about an n-step DQN if this "n-stepness" turns it into an on-policy method, which will make our large experience buffer useless? In practice, this is usually not black and white. You may still use an n-step DQN if it will help to speed up DQNs, but you need to be modest with the selection of n .

Small values of two or three usually work well, because our trajectories in the experience buffer are not that different from one-step transitions. In such cases, convergence speed usually improves proportionally, but large values of n can break the training process. So, the number of steps should be tuned, but convergence speeding up usually makes it worth doing.

Implementation

As the `ExperienceSourceFirstLast` class already supports the multi-step Bellman unroll, our n -step version of a DQN is extremely simple. There are only two modifications that we need to make to the basic DQN to turn it into an n -step version:

- Pass the count of steps that we want to unroll on `ExperienceSourceFirstLast` creation in the `steps_count` parameter.
- Pass the correct gamma to the `calc_loss_dqn` function. This modification is really easy to overlook, which could be harmful to convergence. As our Bellman is now n -steps, the discount coefficient for the last state in the experience chain will no longer be just γ , but γ^n .

You can find the whole example in `Chapter08/02_dqn_n_steps.py`, with only the modified lines shown here:

```
exp_source = ptan.experience.ExperienceSourceFirstLast(  
    env, agent, gamma=params.gamma, env_seed=common.SEED,  
    steps_count=n_steps  
)
```

The `n_steps` value is a count of steps passed in command-line arguments; the default is to use four steps.

Another modification is in `gamma` passed to the `calc_loss_dqn` function:

```
loss_v = common.calc_loss_dqn(  
    batch, net, tgt_net.target_model,  
    gamma=params.gamma**n_steps, device=device)
```

Results

The training module `Chapter08/02_dqn_n_steps.py` can be started as before, with the additional command-line option `-n`, which gives a count of steps to unroll the Bellman equation. These are charts for our baseline and n -step DQN (using a common set of parameters), with n being equal to 2 and 3.

As you can see, the Bellman unroll has given a significant convergence speedup:

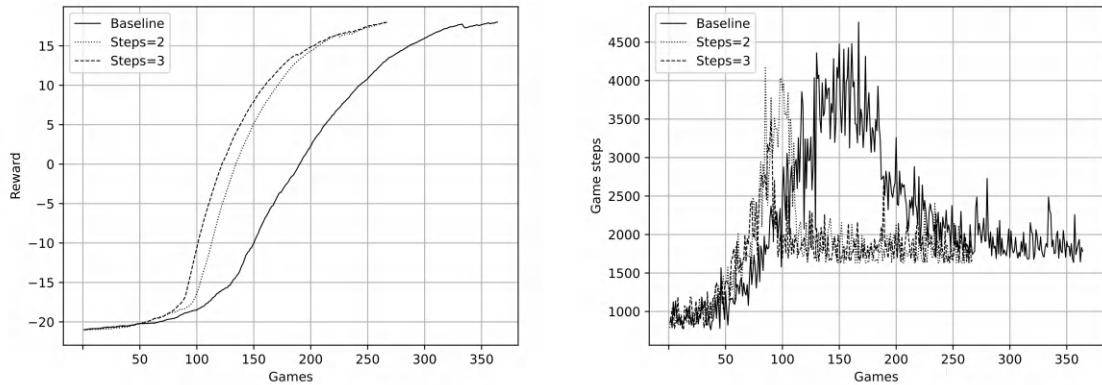


Figure 8.5: The reward and number of steps for basic (one-step) DQN and n-step versions

As you can see in the diagram, the three-step DQN converges significantly faster than the simple DQN, which is a nice improvement. So, what about a larger n ? *Figure 8.6* shows the reward dynamics for $n = 3 \dots 6$:

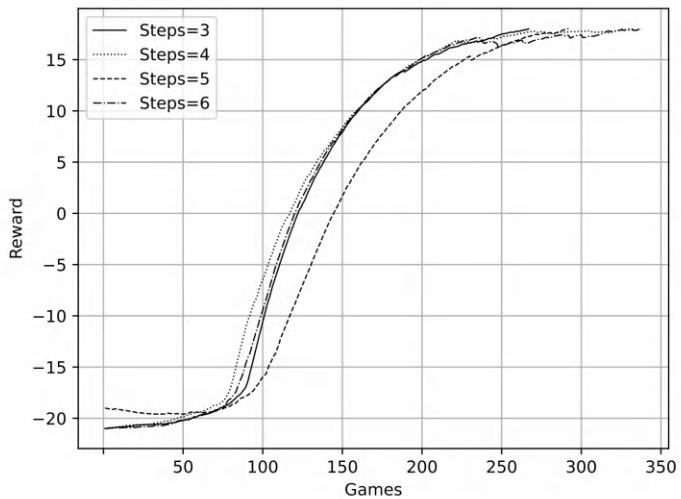


Figure 8.6: Reward dynamics for cases with $n = 3 \dots 6$ with common hyperparameters

As you can see, going from three steps to four has given some improvement, but it is much less than before. The variant with $n = 5$ is worse and very close to $n = 2$. The same is true for $n = 6$. So, in our case, $n = 3$ looks optimal.

Hyperparameter tuning

In this extension, hyperparameter tuning was done individually for every n from 2 to 7. The following table shows the best parameters and number of games they require to solve the game:

n	Learning rate	γ	Games
2	$3.97 \cdot 10^{-5}$	0.98	293
3	$7.82 \cdot 10^{-5}$	0.98	260
4	$6.07 \cdot 10^{-5}$	0.98	290
5	$7.52 \cdot 10^{-5}$	0.99	268
6	$6.78 \cdot 10^{-5}$	0.995	261
7	$8.59 \cdot 10^{-5}$	0.98	284

Table 8.1: The best hyperparameters (learning rate and gamma) for every n

This table also confirms the conclusions of the untuned version comparison – unrolling the Bellman equation for two and three steps improves the convergence, but a further increase of n produces worse results. $n = 6$ gives us a comparable result to $n = 3$, but the outcomes for $n = 4$ and $n = 5$ are worse, so we should stop at $n = 3$.

Figure 8.7 compares the training dynamics of tuned versions of the baseline and N-step DQN with $n = 2$ and $n = 3$.

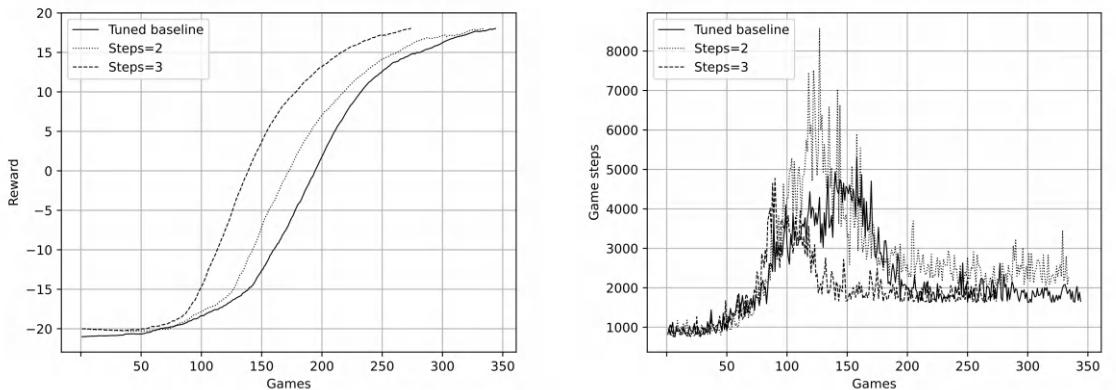


Figure 8.7: The reward and number of steps after hyperparameter tuning

Double DQN

The next fruitful idea on how to improve a basic DQN came from DeepMind researchers in the paper titled *Deep reinforcement learning with double Q-learning* [VGS16]. In the paper, the authors demonstrated that the basic DQN tends to overestimate values for Q , which may be harmful to training performance and sometimes can lead to suboptimal policies.

The root cause of this is the max operation in the Bellman equation, but the strict proof is a bit complicated (you can find the full explanation in the paper). As a solution to this problem, the authors proposed modifying the Bellman update a bit.

In the basic DQN, our target value for Q looked like this:

$$Q(s_t, a_t) = r_t + \gamma \max_a Q'(s_{t+1}, a)$$

$Q'(s_{t+1}, a)$ was Q-values calculated using our target network, the weights of which are copied from the trained network every n steps. The authors of the paper proposed choosing actions for the next state using the trained network, but taking values of Q from the target network. So, the new expression for target Q-values will look like this:

$$Q(s_t, a_t) = r_t + \gamma \max_a Q'(s_{t+1}, \arg \max_a Q(s_{t+1}, a))$$

The authors proved that this simple tweak fixes overestimation completely, and they called this new architecture **double DQN**.

Implementation

The core implementation is very simple. What we need to do is slightly modify our loss function. But let's go a step further and compare action values produced by basic DQN and double DQN. According to the paper author's our baseline DQN should have consistently higher values predicted for the same states than the double DQN version. To do this, we store a random held-out set of states and periodically calculate the mean value of the best action for every state in the evaluation set.

The complete example is in `Chapter08/03_dqn_double.py`. Let's first take a look at the loss function:

```
def calc_loss_double_dqn(
    batch: tt.List[ptan.experience.ExperienceFirstLast],
    net: nn.Module, tgt_net: nn.Module, gamma: float, device: torch.device):
    states, actions, rewards, dones, next_states = common.unpack_batch(batch)

    states_v = torch.as_tensor(states).to(device)
    actions_v = torch.tensor(actions).to(device)
    rewards_v = torch.tensor(rewards).to(device)
    done_mask = torch.BoolTensor(dones).to(device)
```

We will use this function instead of `common.calc_loss_dqn` and they both share lots of code. The main difference is in the next Q-values estimation:

```

actions_v = actions_v.unsqueeze(-1)
state_action_vals = net(states_v).gather(1, actions_v)
state_action_vals = state_action_vals.squeeze(-1)
with torch.no_grad():
    next_states_v = torch.as_tensor(next_states).to(device)
    next_state_acts = net(next_states_v).max(1)[1]
    next_state_acts = next_state_acts.unsqueeze(-1)
    next_state_vals = tgt_net(next_states_v).gather(1, next_state_acts).squeeze(-1)
    next_state_vals[done_mask] = 0.0
    exp_sa_vals = next_state_vals.detach() * gamma + rewards_v
return nn.MSELoss()(state_action_vals, exp_sa_vals)

```

The preceding code snippet calculates the loss in a slightly different way. In the double DQN version, we calculate the best action to take in the next state using our main trained network, but values corresponding to this action come from the target network.



This part could be implemented in a faster way, by combining `next_states_v` with `states_v` and calling our main network only once, but it will make the code less clear.

The rest of the function is the same: we mask completed episodes and compute the **mean squared error (MSE)** loss between Q-values predicted by the network and approximated Q-values.

The last function that we consider calculates the values of our held-out state:

```

@torch.no_grad()
def calc_values_of_states(states: np.ndarray, net: nn.Module, device: torch.device):
    mean_vals = []
    for batch in np.array_split(states, 64):
        states_v = torch.tensor(batch).to(device)
        action_values_v = net(states_v)
        best_action_values_v = action_values_v.max(1)[0]
        mean_vals.append(best_action_values_v.mean().item())
    return np.mean(mean_vals)

```

There is nothing complicated here: we just split our held-out states array into equal chunks and pass every chunk to the network to obtain action values. From those values, we choose the action with the largest value (for every state) and calculate the mean of such values. As our array with states is fixed for the whole training process, and this array is large enough (in the code, we store 1,000 states), we can compare the dynamics of this mean value in both DQN variants.

The rest of the `03_dqn_double.py` file is almost the same; the two differences are usage of our tweaked loss function and keeping a randomly sampled 1,000 states for periodical evaluation. This happens in the `process_batch` function:

```
if engine.state.iteration % EVAL_EVERY_FRAME == 0:
    eval_states = getattr(engine.state, "eval_states", None)
    if eval_states is None:
        eval_states = buffer.sample(STATES_TO_EVALUATE)
        eval_states = [
            np.asarray(transition.state)
            for transition in eval_states
        ]
        eval_states = np.asarray(eval_states)
        engine.state.eval_states = eval_states
    engine.state.metrics["values"] = \
        common.calc_values_of_states(eval_states, net, device)
```

Results

My experiments show that with common hyperparameters, double DQN has a negative effect on reward dynamics. Sometimes, double DQN leads to better initial dynamics and the trained agent learns how to win more games faster, but reaching the end reward boundary takes longer. You can perform your own experiment on other games or try parameters from the original paper.

The following are reward charts from the experiment where double DQN was a bit better than the baseline version:

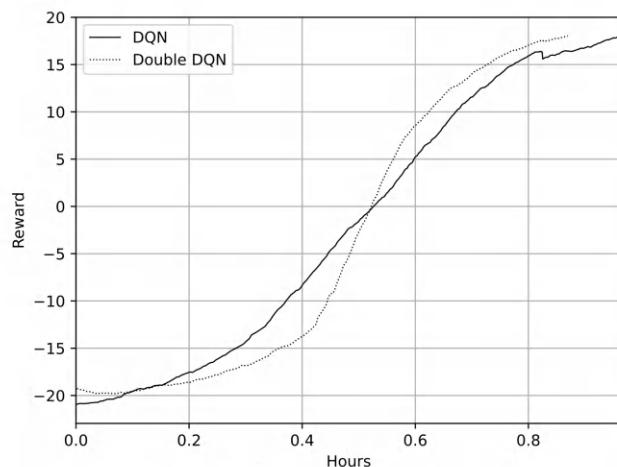


Figure 8.8: Reward dynamics for double and baseline DQN

Besides the standard metrics, the example also outputs the mean value for the held-out set of states, which are shown in *Figure 8.9*.

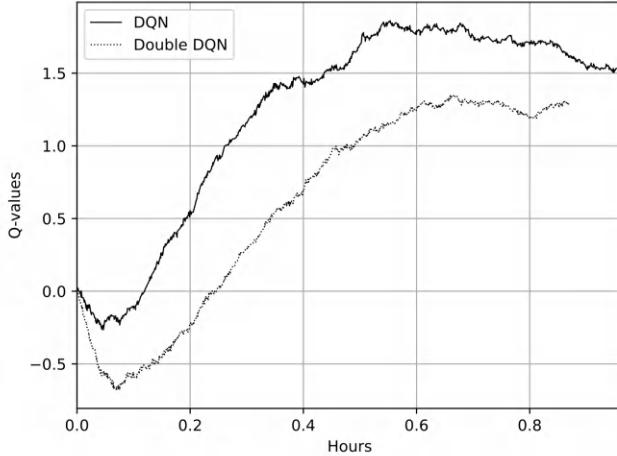


Figure 8.9: Values predicted by the network for held-out states

As you can see, the basic DQN does an overestimation of values, so values decrease after a certain level. In contrast, the double DQN grows more consistently. In my experiments, the double DQN has only a small effect on the training time, but this doesn't necessarily mean that the double DQN is useless, as Pong is a simple environment. In more complicated games, the double DQN could give better results.

Hyperparameter tuning

The tuning of hyperparameters also wasn't very successful for the double DQN. After 30 trials, the best values for the learning rate and gamma were able to solve Pong in 412 games, which is worse than the baseline DQN.

Noisy networks

The next improvement that we are going to look at addresses another RL problem: exploration of the environment. The paper that we will draw from is called *Noisy networks for exploration* [For+17] and it has a very simple idea for learning exploration characteristics during training instead of having a separate schedule related to exploration.

A classical DQN achieves exploration by choosing random actions with a specially defined hyperparameter ϵ , which is slowly decreased over time from 1.0 (fully random actions) to some small ratio of 0.1 or 0.02.

This process works well for simple environments with short episodes, without much non-stationarity during the game; but even in such simple cases, it requires tuning to make the training processes efficient.

In the *Noisy Networks* paper, the authors proposed a quite simple solution that, nevertheless, works well. They add noise to the weights of fully connected layers of the network and adjust the parameters of this noise during training using backpropagation.



This method shouldn't be confused with "the network decides where to explore more," which is a much more complex approach that also has widespread support (for example, see articles about intrinsic motivation and count-based exploration methods [Ost+17], [Mar+17]). We will discuss advanced exploration techniques in *Chapter 21*.

The authors proposed two ways of adding the noise, both of which work according to their experiments, but they have different computational overheads:

- **Independent Gaussian noise:** For every weight in a fully connected layer, we have a random value that we draw from the normal distribution. Parameters of the noise, μ and σ , are stored inside the layer and get trained using backpropagation in the same way that we train weights of the standard linear layer. The output of such a "noisy layer" is calculated in the same way as in a linear layer.
- **Factorized Gaussian noise:** To minimize the number of random values to be sampled, the authors proposed keeping only two random vectors: one with the size of the input and another with the size of the output of the layer. Then, a random matrix for the layer is created by calculating the outer product of the vectors.

Implementation

In PyTorch, both methods can be easily implemented in a very straightforward way. What we need to do is create our own custom `nn.Linear` layer with weights calculated as $w_{i,j} = \mu_{i,j} + \sigma_{i,j} \cdot \epsilon_{i,j}$, where μ and σ are trainable parameters and $\epsilon \sim \mathcal{N}(0, 1)$ is random noise sampled from the normal distribution after every optimization step.

Previous editions of the book used my implementation of both methods, but now we'll simply use the implementation from the popular TorchRL library I mentioned in *Chapter 7*. Let's take a look at relevant parts of the implementation (the full code can be found in `torchrl/modules/models/exploration.py` in the TorchRL repository).

The following is the constructor of the `NoisyLinear` class, which creates all the parameters we need to optimize:

```
class NoisyLinear(nn.Linear):
    def __init__(self, in_features: int, out_features: int, bias: bool = True,
                 device: Optional[DEVICE_TYPING] = None, dtype: Optional[torch.dtype] = None,
                 std_init: float = 0.1,
                 ):
        nn.Module.__init__(self)
        self.in_features = int(in_features)
        self.out_features = int(out_features)
        self.std_init = std_init

        self.weight_mu = nn.Parameter(
            torch.empty(out_features, in_features, device=device,
                        dtype=dtype, requires_grad=True)
        )
        self.weight_sigma = nn.Parameter(
            torch.empty(out_features, in_features, device=device,
                        dtype=dtype, requires_grad=True)
        )
        self.register_buffer(
            "weight_epsilon",
            torch.empty(out_features, in_features, device=device, dtype=dtype),
        )
        if bias:
            self.bias_mu = nn.Parameter(
                torch.empty(out_features, device=device, dtype=dtype, requires_grad=True)
            )
            self.bias_sigma = nn.Parameter(
                torch.empty(out_features, device=device, dtype=dtype, requires_grad=True)
            )
            self.register_buffer(
                "bias_epsilon", torch.empty(out_features, device=device, dtype=dtype),
            )
        else:
            self.bias_mu = None
        self.reset_parameters()
        self.reset_noise()
```

In the constructor, we create matrices for μ and σ . This implementation inherits from `torch.nn.Linear`, but calls the `nn.Module.__init__()` method, so normal `Linear` weights and bias buffers are not created.

To make new matrices trainable, we need to wrap their tensors in an `nn.Parameter`. The `register_buffer` method creates a tensor in the network that won't be updated during backpropagation, but will be handled by the `nn.Module` machinery (for example, it will be copied to the GPU with the `cuda()` call).

An extra parameter and buffer are created for the bias of the layer. At the end, we call the `reset_parameters()` and `reset_noise()` methods, which perform the initialization of the created trainable parameters and the buffer with the epsilon value.

In the following three methods, we initialize the trainable parameters μ and σ according to the paper:

```

def reset_parameters(self) -> None:
    mu_range = 1 / math.sqrt(self.in_features)
    self.weight_mu.data.uniform_(-mu_range, mu_range)
    self.weight_sigma.data.fill_(self.std_init / math.sqrt(self.in_features))
    if self.bias_mu is not None:
        self.bias_mu.data.uniform_(-mu_range, mu_range)
        self.bias_sigma.data.fill_(self.std_init / math.sqrt(self.out_features))

def reset_noise(self) -> None:
    epsilon_in = self._scale_noise(self.in_features)
    epsilon_out = self._scale_noise(self.out_features)
    self.weight_epsilon.copy_(epsilon_out.outer(epsilon_in))
    if self.bias_mu is not None:
        self.bias_epsilon.copy_(epsilon_out)

def _scale_noise(
    self, size: Union[int, torch.Size, Sequence]) -> torch.Tensor:
    if isinstance(size, int):
        size = (size,)
    x = torch.randn(*size, device=self.weight_mu.device)
    return x.sign().mul_(x.abs().sqrt_())

```

The matrix for μ is initialized with uniform random values. The initial value for σ is constant depending on the count of neurons in the layer.

For the noise initialization, factorized Gaussian noise is used – we sample two random vectors and calculate the outer product to get the matrix for ϵ . The outer product is a linear algebra operation when two vectors of the same size are producing the square matrix filled with product of all combination of each vector's element.

The rest is simple: we redefine the `weight` and `bias` properties, which are expected in `nn.Linear` layer, so `NoisyLinear` could be used everywhere `nn.Linear` is used:

```
@property
def weight(self) -> torch.Tensor:
    if self.training:
        return self.weight_mu + self.weight_sigma * self.weight_epsilon
    else:
        return self.weight_mu

@property
def bias(self) -> Optional[torch.Tensor]:
    if self.bias_mu is not None:
        if self.training:
            return self.bias_mu + self.bias_sigma * self.bias_epsilon
        else:
            return self.bias_mu
    else:
        return None
```

This implementation is simple, but has one very subtle nuance — the ϵ values are not updated after every optimization step (and it is not mentioned in the documentation). This issue is already reported in the TorchRL repo, but for the current stable release, we have to call the `reset_noise()` method explicitly. Hopefully, it will be fixed and the `NoisyLinear` layer will update the noise automatically.

From the implementation point of view, that's it. What we now need to do to turn the classic DQN into a noisy network variant is just replace `nn.Linear` (which are the two last layers in our DQN network) with the `NoisyLinear` layer. Of course, you have to remove all the code related to the epsilon-greedy strategy.

To check the internal noise level during training, we can monitor the **signal-to-noise ratio (SNR)** of our noisy layers, which is $RMS(\mu)/RMS(\sigma)$, where RMS is the root mean square of the corresponding weights. In our case, the SNR shows how many times the stationary component of the noisy layer is larger than the injected noise.

Results

After the training, the TensorBoard charts show much better training dynamics. The model was able to reach the mean score of 18 after 250 games, which is an improvement in comparison to 350 for the baseline DQN. But because of extra operations required for noisy networks, their training is a bit slower (194 FPS versus 240 FPS for the baseline), so, time-wise, the difference is less impressive.

But still, the results look good:

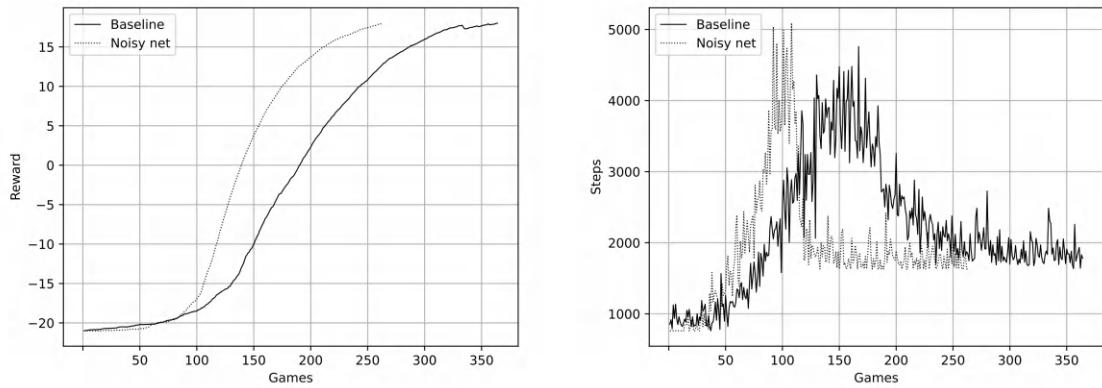


Figure 8.10: Noisy networks compared to the baseline DQN

After checking the SNR chart (Figure 8.11), you may notice that both layers' noise levels have decreased very quickly.

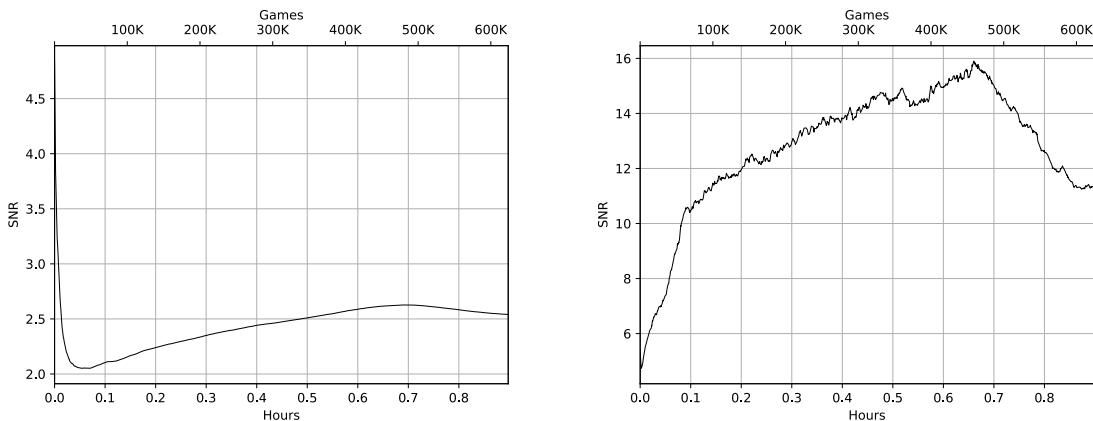


Figure 8.11: SNR change in layer 1 (left) and layer 2 (right)

The first layer has gone from $\frac{1}{2}$ to almost $\frac{1}{2.6}$ ratio of noise. The second layer is even more interesting, as its noise level decreased from $\frac{1}{4}$ in the beginning to $\frac{1}{16}$, but after 450K frames (roughly the same time as when raw rewards climbed close to the 20 score), the level of noise in the last layer started to increase again, pushing the agent to explore the environment more.

This makes a lot of sense, as after reaching high score levels, the agent basically knows how to play at a good level, but still needs to “polish” its actions to improve the results even more.

Hyperparameter tuning

After the tuning, the best set of parameters was able to solve the game after 273 rounds, which is an improvement over the baseline:

```
learning_rate=7.142520950425814e-05,
gamma=0.99,
```

The following are charts comparing the reward dynamics and steps for tuned baseline DQN and tuned noisy networks:

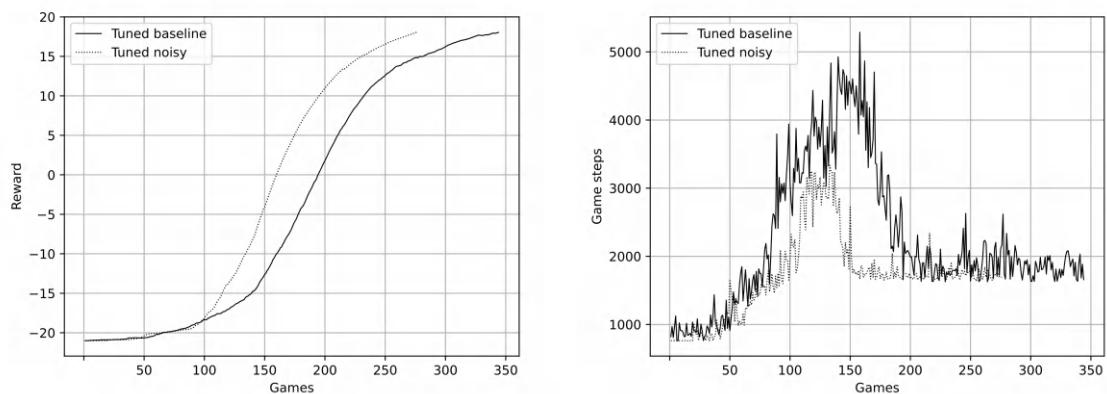


Figure 8.12: Comparison of tuned baseline DQN and tuned noisy network

On both charts, we see improvements introduced by noisy networks: it takes fewer games to reach a score of 21 and during the training, games have a smaller amount of steps.

Prioritized replay buffer

The next very useful idea on how to improve DQN training was proposed in 2015 in the paper *Prioritized experience replay* [Sch+15]. This method tries to improve the efficiency of samples in the replay buffer by prioritizing those samples according to the training loss.

The basic DQN used the replay buffer to break the correlation between immediate transitions in our episodes.

As we discussed in *Chapter 6*, the examples we experience during the episode will be highly correlated, as most of the time, the environment is “smooth” and doesn’t change much according to our actions. However, the **stochastic gradient descent (SGD)** method assumes that the data we use for training has an iid property. To solve this problem, the classic DQN method uses a large buffer of transitions, randomly and uniformly sampled to get the next training batch.

The authors of the paper questioned this uniform random sample policy and proved that by assigning priorities to buffer samples, according to training loss and sampling the buffer proportional to those priorities, we can significantly improve convergence and the policy quality of the DQN. This method’s basic idea could be explained as “train more on data that surprises you.” The tricky point here is to keep the balance of training on an “unusual” sample and training on the rest of the buffer. If we focus only on a small subset of the buffer, we can lose our i.i.d. property and simply overfit on this subset.

From the mathematical point of view, the priority of every sample in the buffer is calculated as $P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$, where p_i is the priority of the i -th sample in the buffer and α is the number that shows how much emphasis we give to the priority. If $\alpha = 0$, our sampling will become uniform as in the classic DQN method. Larger values for α put more stress on samples with higher priority. So, it’s another hyperparameter to tune, and the starting value of α proposed by the paper is 0.6.

There were several options proposed in the paper for how to define the priority, and the most popular is to make it proportional to the loss for this particular example in the Bellman update. New samples added to the buffer need to be assigned a maximum value of priority to be sure that they will be sampled soon.

By adjusting the priorities for the samples, we are introducing bias into our data distribution (we sample some transitions much more frequently than others), which we need to compensate for if SGD is to work. To get this result, the authors of the study used sample weights, which needed to be multiplied by the individual sample loss. The value of the weight for each sample is defined as $w_i = (N \cdot P(i))^{-\beta}$, where β is another hyperparameter that should be between 0 and 1.

With $\beta = 1$, the bias introduced by the sampling is fully compensated for, but the authors showed that it’s good for convergence to start with β between 0 and 1 and slowly increase it to 1 during the training.

Implementation

To implement this method, we have to introduce certain changes in our code:

- First of all, we need a new replay buffer that will track priorities, sample a batch according to them, calculate weights, and let us update priorities after the loss has become known.

- The second change will be the loss function itself. Now we not only need to incorporate weights for every sample, but we need to pass loss values back to the replay buffer to adjust the priorities of the sampled transitions.

In the main module, `Chapter08/05_dqn_prio_replay.py`, we have all those changes implemented. For the sake of simplicity, the new priority replay buffer class uses a very similar storage scheme to our previous replay buffer. Unfortunately, new requirements for prioritization make it impossible to implement sampling in $\mathcal{O}(1)$ time (in other words, sampling time will grow with an increase in buffer size). If we are using simple lists, every time that we sample a new batch, we need to process all the priorities, which makes our sampling have $\mathcal{O}(N)$ time complexity in proportion to the buffer size. It's not a big deal if our buffer is small, such as 100k samples, but may become an issue for real-life large buffers of millions of transitions. There are other storage schemes that support efficient sampling in $\mathcal{O}(\log N)$ time, for example, using the segment tree data structure. There are different versions of such optimized buffers available in various libraries – for example, in TorchRL.

The PTAN library also provides an efficient prioritized replay buffer in the class `ptan.experience.PrioritizedReplayBuffer`. You can update the example to use the more efficient version and check the effect on training performance.

But, for now, let's take a look at the naïve version, whose source code you will find in `lib/dqn_extra.py`.

In the beginning, we define parameters for the β increase rate:

```
BETA_START = 0.4
BETA_FRAMES = 100_000
```

Our beta will be changed from 0.4 to 1.0 during the first 100k frames.

Next comes the prioritized replay buffer class:

```
class PrioReplayBuffer(ExperienceReplayBuffer):
    def __init__(self, exp_source: ExperienceSource, buf_size: int,
                 prob_alpha: float = 0.6):
        super().__init__(exp_source, buf_size)
        self.experience_source_iter = iter(exp_source)
        self.capacity = buf_size
        self.pos = 0
        self.buffer = []
        self.prob_alpha = prob_alpha
        self.priorities = np.zeros((buf_size, ), dtype=np.float32)
        self.beta = BETA_START
```

The class for the priority replay buffer inherits from the simple replay buffer in PTAN, which stores samples in a circular buffer (it allows us to keep a fixed amount of entries without reallocating the list). Our subclass uses a NumPy array to keep priorities.

The `update_beta()` method needs to be called periodically to increase beta according to a schedule. The `populate()` method needs to pull the given number of transitions from the `ExperienceSource` object and store them in the buffer:

```
def update_beta(self, idx: int) -> float:
    v = BETA_START + idx * (1.0 - BETA_START) / BETA_FRAMES
    self.beta = min(1.0, v)
    return self.beta

def populate(self, count: int):
    max_prio = self.priorities.max(initial=1.0)
    for _ in range(count):
        sample = next(self.experience_source_iter)
        if len(self.buffer) < self.capacity:
            self.buffer.append(sample)
        else:
            self.buffer[self.pos] = sample
        self.priorities[self.pos] = max_prio
        self.pos = (self.pos + 1) % self.capacity
```

As our storage for the transitions is implemented as a circular buffer, we have two different situations with this buffer:

- When our buffer hasn't reached the maximum capacity, we just need to append a new transition to the buffer.
- If the buffer is already full, we need to overwrite the oldest transition, which is tracked by the `pos` class field, and adjust this position modulo buffer's size.

In the `sample` method, we need to convert priorities to probabilities using our α hyperparameter:

```
def sample(self, batch_size: int) -> tt.Tuple[
    tt.List[ExperienceFirstLast], np.ndarray, np.ndarray
]:
    if len(self.buffer) == self.capacity:
        prios = self.priorities
    else:
        prios = self.priorities[:self.pos]
    probs = prios ** self.prob_alpha
    probs /= probs.sum()
```

Then, using those probabilities, we sample our buffer to obtain a batch of samples:

```
indices = np.random.choice(len(self.buffer), batch_size, p=probs)
samples = [self.buffer[idx] for idx in indices]
```

As the last step, we calculate weights for samples in the batch:

```
total = len(self.buffer)
weights = (total * probs[indices]) ** (-self.beta)
weights /= weights.max()
return samples, indices, np.array(weights, dtype=np.float32)
```

This returns three objects: the batch, indices, and weights. Indices for batch samples are required to update priorities for sampled items.

The last function of the priority replay buffer allows us to update new priorities for the processed batch:

```
def update_priorities(self, batch_indices: np.ndarray, batch_priorities: np.ndarray):
    for idx, prio in zip(batch_indices, batch_priorities):
        self.priorities[idx] = prio
```

It's the responsibility of the caller to use this function with the calculated losses for the batch.

The next custom function that we have in our example is the loss calculation. As the `MSELoss` class in PyTorch doesn't support weights (which is understandable, as MSE is loss used in regression problems, but weighting of the samples is commonly utilized in classification losses), we need to calculate the MSE and explicitly multiply the result on the weights:

```
def calc_loss(batch: tt.List[ExperienceFirstLast], batch_weights: np.ndarray,
              net: nn.Module, tgt_net: nn.Module, gamma: float,
              device: torch.device) -> tt.Tuple[torch.Tensor, np.ndarray]:
    states, actions, rewards, dones, next_states = common.unpack_batch(batch)

    states_v = torch.as_tensor(states).to(device)
    actions_v = torch.tensor(actions).to(device)
    rewards_v = torch.tensor(rewards).to(device)
    done_mask = torch.BoolTensor(dones).to(device)
    batch_weights_v = torch.tensor(batch_weights).to(device)

    actions_v = actions_v.unsqueeze(-1)
    state_action_vals = net(states_v).gather(1, actions_v)
```

```

state_action_vals = state_action_vals.squeeze(-1)
with torch.no_grad():
    next_states_v = torch.as_tensor(next_states).to(device)
    next_s_vals = tgt_net(next_states_v).max(1)[0]
    next_s_vals[done_mask] = 0.0
    exp_sa_vals = next_s_vals.detach() * gamma + rewards_v
    l = (state_action_vals - exp_sa_vals) ** 2
    losses_v = batch_weights_v * l
return losses_v.mean(), (losses_v + 1e-5).data.cpu().numpy()

```

In the last part of the loss calculation, we implement the same MSE loss but write our expression explicitly, rather than using the library. This allows us to take into account the weights of samples and keep individual loss values for every sample. Those values will be passed to the priority replay buffer to update priorities. A small value is added to every loss to handle the situation of zero loss value, which will lead to zero priority for an entry in the replay buffer.

In the main section of our program, we have only two updates: the creation of the replay buffer and our processing function. Buffer creation is straightforward, so we will only take a look at a new processing function:

```

def process_batch(engine, batch_data):
    batch, batch_indices, batch_weights = batch_data
    optimizer.zero_grad()
    loss_v, sample_prios = calc_loss(
        batch, batch_weights, net, tgt_net.target_model,
        gamma=params.gamma, device=device)
    loss_v.backward()
    optimizer.step()
    buffer.update_priorities(batch_indices, sample_prios)
    epsilon_tracker.frame(engine.state.iteration)
    if engine.state.iteration % params.target_net_sync == 0:
        tgt_net.sync()
    return {
        "loss": loss_v.item(),
        "epsilon": selector.epsilon,
        "beta": buffer.update_beta(engine.state.iteration),
    }

```

There are several changes here:

- Our batch now contains three entities: the batch of data, indices of sampled items, and samples' weights.
- We call our new loss function, which accepts weights and returns the additional items' priorities. They are passed to the `buffer.update_priorities()` function to reprioritize items that we have sampled.
- We call the `update_beta()` method of the buffer to change the `beta` parameter according to the schedule.

Results

This example can be trained as usual. According to my experiments, the prioritized replay buffer took almost the same absolute time to solve the environment: almost an hour. But it took fewer training iterations and fewer episodes. So, wall clock time is the same mostly due to the less efficient replay buffer, which, of course, could be resolved by proper $\mathcal{O}(\log N)$ implementation of the buffer.

Here is the comparison of reward dynamics of the baseline and prioritized replay buffer (right). The x axis is the game episodes:

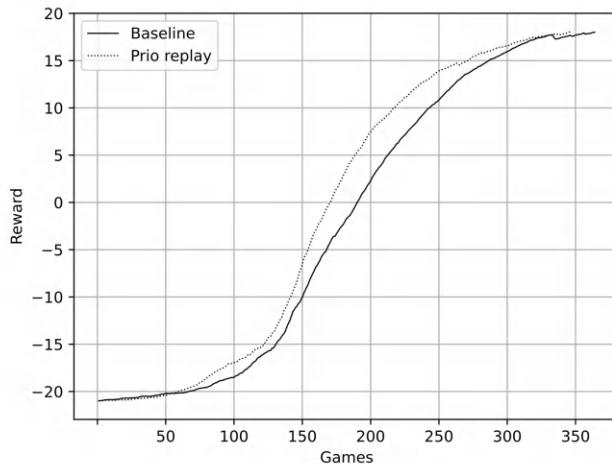


Figure 8.13: Reward dynamics for prioritized replay buffer in comparison to basic DQN

Another difference to note on the TensorBoard charts is a much lower loss for the prioritized replay buffer. The following chart shows the comparison:

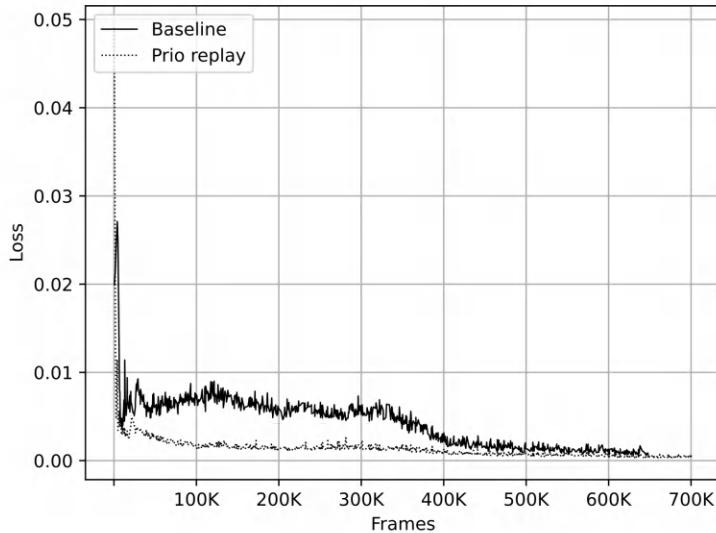


Figure 8.14: The comparison of loss during the training

Lower loss value is also expected and is a good sign that our implementation works. The idea of prioritization is to train more on samples with high loss value, so training becomes more efficient. But there is a danger here: loss value during the training is not the primary objective to optimize; we can have very low loss, but due to a lack of exploration, the final policy learned could be far from being optimal.

Hyperparameter tuning

Hyperparameter tuning for the prioritized replay buffer was done with an additional parameter for α , which was sampled from a fixed list of values ranging from 0.3 to 0.9 (with steps of 0.1). The best combination was able to solve Pong after 330 episodes and had $\alpha = 0.6$ (the same as in the paper):

```
learning_rate=8.839010139505506e-05,
gamma=0.99,
```

The following are charts comparing the tuned baseline DQN with the tuned prioritized replay buffer:

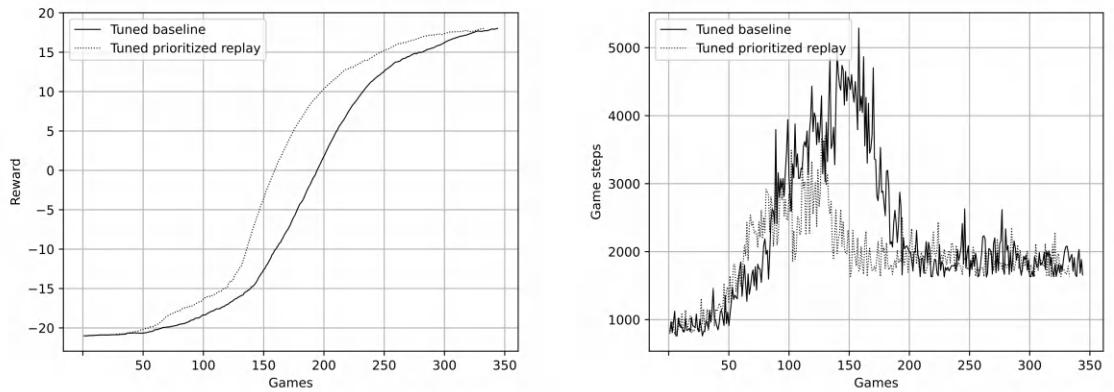


Figure 8.15: Comparison of tuned baseline DQN and tuned prioritized replay buffer

Here, we see the prioritized replay buffer had faster gameplay improvement, but it took almost the same amount of games to reach score 21. On the right chart (with the amount of game steps), the prioritized replay buffer was also a bit better.

Dueling DQN

This improvement to DQN was proposed in 2015, in the paper called *Dueling network architectures for deep reinforcement learning* [Wan+16]. The core observation of this paper is that the Q-values, $Q(s, a)$, that our network is trying to approximate can be divided into quantities: the value of the state, $V(s)$, and the advantage of actions in this state, $A(s, a)$.

You have seen the quantity $V(s)$ before, as it was the core of the value iteration method from *Chapter 5*. It is just equal to the discounted expected reward achievable from this state. The advantage $A(s, a)$ is supposed to bridge the gap from $V(s)$ to $Q(s, a)$, as, by definition, $Q(s, a) = V(s) + A(s, a)$. In other words, the advantage $A(s, a)$ is just the delta, saying how much extra reward some particular action from the state brings us. The advantage could be positive or negative and, in general, could have any magnitude. For example, at some tipping point, the choice of one action over another can cost us a lot of the total reward.

The *Dueling* paper's contribution was an explicit separation of the value and the advantage in the network's architecture, which brought better training stability, faster convergence, and better results on the Atari benchmark. The architecture difference from the classic DQN network is shown in the following illustration. The classic DQN network (top) takes features from the convolution layer and, using fully connected layers, transforms them into a vector of Q-values, one for each action.

On the other hand, dueling DQN (bottom) takes convolution features and processes them using two independent paths: one path is responsible for $V(s)$ prediction, which is just a single number, and another path predicts individual advantage values, having the same dimension as Q-values in the classic case. After that, we add $V(s)$ to every value of $A(s, a)$ to obtain $Q(s, a)$, which is used and trained as normal. *Figure 8.16* (from the paper) compares the basic DQN and dueling DQN:

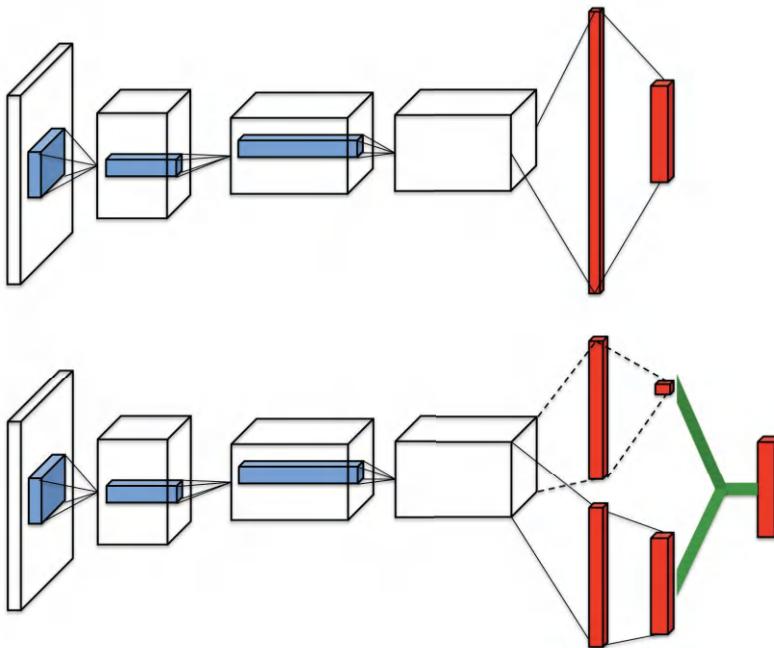


Figure 8.16: A basic DQN (top) and dueling architecture (bottom)

These changes in the architecture are not enough to make sure that the network will learn $V(s)$ and $A(s, a)$ as we want it to. Nothing prevents the network, for example, from predicting some state, $V(s) = 0$, and $A(s) = [1, 2, 3, 4]$, which is completely wrong, as the predicted $V(s)$ is not the expected value of the state. We have yet another constraint to set: we want the mean value of the advantage of any state to be zero. In that case, the correct prediction for the preceding example will be $V(s) = 2.5$ and $A(s) = [-1.5, -0.5, 0.5, 1.5]$.

This constraint could be enforced in various ways, for example, via the loss function; but in the *Dueling* paper, the authors proposed the very elegant solution of subtracting the mean value of the advantage from the Q expression in the network, which effectively pulls the mean for the advantage to zero:

$$Q(s, a) = V(s) + A(s, a) - \frac{1}{N} \sum_k A(s, k)$$

This keeps the changes that need to be made in the classic DQN very simple: to convert it to the double DQN, you need to change only the network architecture, without affecting other pieces of the implementation.

Implementation

The complete example is available in `Chapter08/06_dqn_dueling.py`. All the changes sit in the network architecture, so here, I'll only show the network class (which is in the `lib/dqn_extra.py` module).

The convolution part is exactly the same as before:

```
class DuelingDQN(nn.Module):
    def __init__(self, input_shape: tt.Tuple[int, ...], n_actions: int):
        super(DuelingDQN, self).__init__()

        self.conv = nn.Sequential(
            nn.Conv2d(input_shape[0], 32, kernel_size=8, stride=4),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, stride=1),
            nn.ReLU(),
            nn.Flatten()
    )
```

Instead of defining a single path of fully connected layers, we create two different transformations: one for advantages and one for value prediction:

```
size = self.conv(torch.zeros(1, *input_shape)).size()[-1]
self.fc_adv = nn.Sequential(
    nn.Linear(size, 256),
    nn.ReLU(),
    nn.Linear(256, n_actions)
)
self.fc_val = nn.Sequential(
    nn.Linear(size, 256),
    nn.ReLU(),
    nn.Linear(256, 1)
)
```

Also, to keep the number of parameters in the model comparable to the original network, the inner dimension in both paths is decreased from 512 to 256.

The changes in the `forward()` function are also very simple, thanks to PyTorch's expressiveness:

```
def forward(self, x: torch.ByteTensor):
    adv, val = self.adv_val(x)
    return val + (adv - adv.mean(dim=1, keepdim=True))

def adv_val(self, x: torch.ByteTensor):
    xx = x / 255.0
    conv_out = self.conv(xx)
    return self.fc_adv(conv_out), self.fc_val(conv_out)
```

Here, we calculate the value and advantage for our batch of samples and add them together, subtracting the mean of the advantage to obtain the final Q-values. A subtle, but important, difference lies in calculating the mean along the second dimension of the tensor, which produces a vector of the mean advantage for every sample in our batch.

Results

After training a dueling DQN, we can compare it to the classic DQN convergence on our Pong benchmark. Dueling architecture has faster convergence in comparison to the basic DQN version:

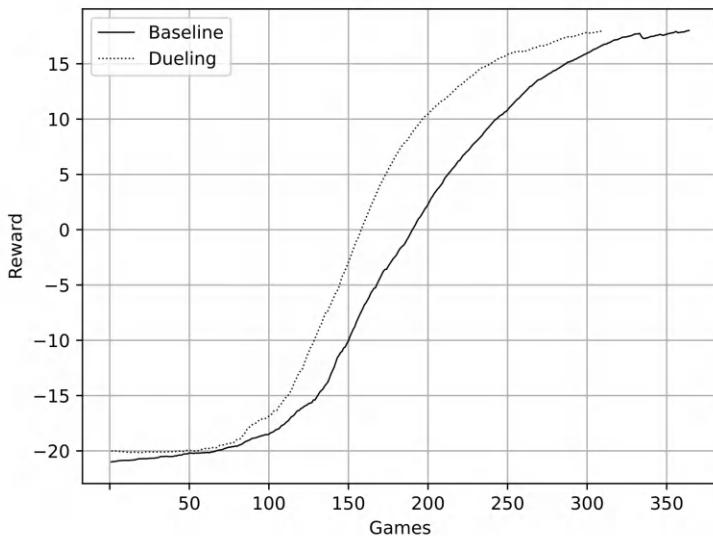


Figure 8.17: The reward dynamic of dueling DQN compared to the baseline version

Our example also outputs the advantage and value for a fixed set of states, shown in the following charts.

They meet our expectations: the advantage is not very different from zero, but the value improves over time (and resembles the value from the *Double DQN* section):

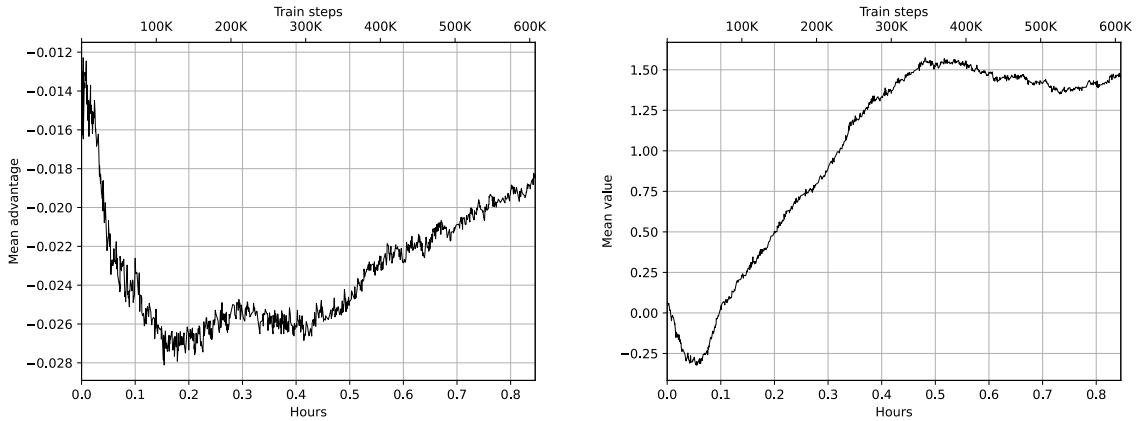


Figure 8.18: Mean advantage (left) and value (right) on a fixed set of states

Hyperparameter tuning

The tuning of the hyperparameters was not very fruitful. After 30 tuning iterations, there were no combinations of learning rate and gamma that were able to converge faster than the common set of parameters.

Categorical DQN

The last, and the most complicated, method in our *DQN improvements toolbox* is from the paper published by DeepMind in June 2017, called *A distributional perspective on reinforcement learning* [BDM17]. Although this paper is a few years old now, it remains highly relevant, and active research is still ongoing in this area. The book *Distributional reinforcement learning* was published in 2023, where the same authors describe the method in greater detail [BDR23].

In the paper, the authors questioned the fundamental pieces of Q-learning — Q-values — and tried to replace them with a more generic Q-value probability distribution. Let's try to understand the idea. Both the Q-learning and value iteration methods work with the values of the actions or states represented as simple numbers and showing how much total reward we can achieve from a state, or an action and a state. However, is it practical to squeeze all future possible rewards into one number? In complicated environments, the future could be stochastic, giving us different values with different probabilities.

For example, imagine the commuter scenario when you regularly drive from home to work. Most of the time, the traffic isn't that heavy, and it takes you around 30 minutes to reach your destination.

It's not exactly 30 minutes, but on average it's 30. From time to time, something happens, like road repairs or an accident, and due to traffic jams, it takes you three times longer to get to work. The probability of your commute time can be represented as a distribution of the "commute time" random variable, and it is shown in the following chart:

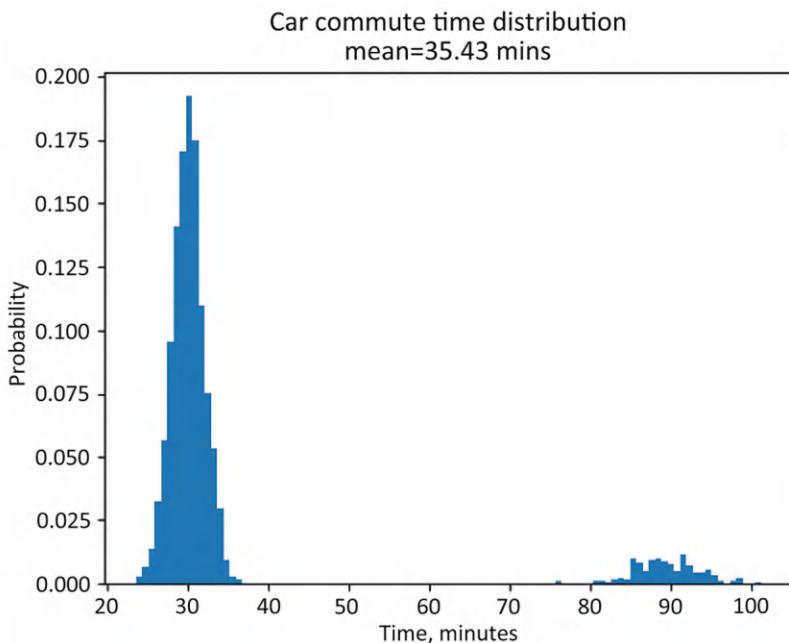


Figure 8.19: The probability distribution of commute time

Now, imagine that you have an alternative way to get to work: the train. It takes a bit longer, as you need to get from home to the train station and from the station to the office, but they are much more reliable than traveling by car (in some countries, like Germany, it might not be the case, but let's consider Swiss trains for our example). Say, for instance, that the train commute time is 40 minutes on average, with a small chance of train disruption, which adds 20 minutes of extra time to the journey. The distribution of the train commute is shown in the following graph:

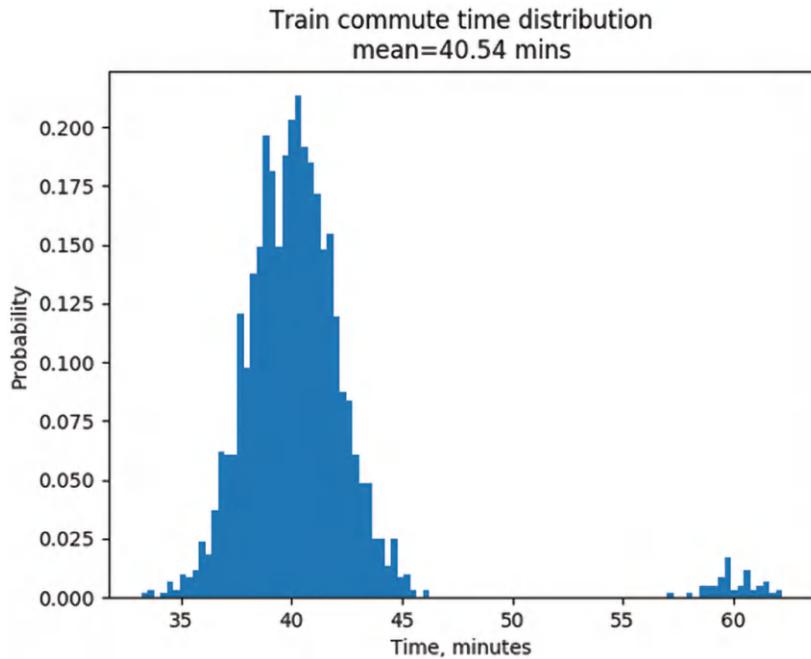


Figure 8.20: The probability distribution of train commute time

Imagine that now we want to make the decision on how to commute. If we know only the mean time for both car and train, a car looks more attractive, as on average it takes 35.43 minutes to travel, which is better than 40.54 minutes for the train.

However, if we look at full distributions, we may decide to go by train, as even in the worst-case scenario, it will be one hour of commuting versus one hour and 30 minutes. Switching to statistical language, the car distribution has much higher **variance**, so in situations when you really have to be at the office in 60 minutes max, the train is better.

The situation becomes even more complicated in the **Markov decision process (MDP)** scenario, when the sequence of decisions needs to be made and every decision might influence the future situation. In the commute example, it might be the time of an important meeting that you need to arrange given the way that you are going to commute. In that case, working with mean reward values might mean losing lots of information about the underlying environment dynamics.

Exactly the same idea was proposed by the authors of *Distributional Perspective on Reinforcement Learning* [9]. Why do we limit ourselves by trying to predict an average value for an action, when the underlying value may have a complicated underlying distribution? Maybe it will help us to work with distributions directly.

The results presented in the paper show that, in fact, this idea could be helpful, but at the cost of introducing a more complicated method. I'm not going to put a strict mathematical definition here, but the overall idea is to predict the distribution of value for every action, similar to the distributions for our car/train example. As the next step, the authors showed that the Bellman equation can be generalized for a distribution case, and it will have the form $Z(x, a) \stackrel{D}{=} R(x, a) + \gamma Z(x', a')$, which is very similar to the familiar Bellman equation, but now $Z(x, a)$ and $R(x, a)$ are the probability distributions and are not single numbers. The notation $A \stackrel{D}{=} B$ indicates equality of distributions A and B .

The resulting distribution can be used to train our network to give better predictions of value distribution for every action of the given state, exactly in the same way as with Q-learning. The only difference will be in the loss function, which now has to be replaced with something suitable for distribution comparison. There are several alternatives available, for example, **Kullback-Leibler (KL)** divergence (or cross-entropy loss), which is used in classification problems, or the Wasserstein metric. In the paper, the authors gave theoretical justification for the Wasserstein metric, but when they tried to apply it in practice, they faced limitations. So, in the end, the paper used KL divergence.

Implementation

As mentioned, the method is quite complex, so it took me a while to implement it and make sure it was working. The complete code is in `Chapter08/07_dqn_distrib.py`, which uses the `distr_projection` function in `lib/dqn_extra.py` to perform distribution projection. Before we check it, I need to say a few words about the implementation logic.

The central part of the method is the probability distribution, which we are approximating. There are lots of ways to represent the distribution, but the authors of the paper chose a quite generic parametric distribution, which is basically a fixed number of values placed regularly on a values range. The range of values should cover the range of possible accumulated discounted reward. In the paper, the authors did experiments with various numbers of atoms, but the best results were obtained with the range split on `N_ATOMS=51` intervals in the range of values from `Vmin=-10` to `Vmax=10`.

For every atom (we have 51 of them), our network predicts the probability that the future discounted value will fall into this atom's range. The central part of the method is the code, which performs the contraction of distribution of the next state's best action using gamma, adds local reward to the distribution, and projects the results back into our original atoms. This logic is implemented in the `dqn_extra.distr_projection` function. In the beginning, we allocate the array that will keep the result of the projection:

```
def distr_projection(next_distr: np.ndarray, rewards: np.ndarray,
                     dones: np.ndarray, gamma: float):
    batch_size = len(rewards)
    proj_distr = np.zeros((batch_size, N_ATOMS), dtype=np.float32)
    delta_z = (Vmax - Vmin) / (N_ATOMS - 1)
```

This function expects the batch of distributions with a shape `(batch_size, N_ATOMS)`, the array of rewards, flags for completed episodes, and our hyperparameters: `Vmin`, `Vmax`, `N_ATOMS`, and `gamma`. The `delta_z` variable is the width of every atom in our value range.

In the following code, we iterate over every atom in the original distribution that we have and calculate the place that this atom will be projected to by the Bellman operator, taking into account our value bounds:

```
for atom in range(N_ATOMS):
    v = rewards + (Vmin + atom * delta_z) * gamma
    tz_j = np.minimum(Vmax, np.maximum(Vmin, v))
```

For example, the very first atom, with index 0, corresponds with the value `Vmin=-10`, but for the sample with reward +1 will be projected into the value $-10 \cdot 0.99 + 1 = -8.9$. In other words, it will be shifted to the right (assume `gamma=0.99`). If the value falls beyond our value range given by `Vmin` and `Vmax`, we clip it to the bounds.

In the next line, we calculate the atom numbers that our samples have projected:

```
b_j = (tz_j - Vmin) / delta_z
```

Of course, samples can be projected between atoms. In such situations, we spread the value in the original distribution at the source atom between the two atoms that it falls between. This spreading should be carefully handled, as our target atom can land exactly at some atom's position. In that case, we just need to add the source distribution value to the target atom.

The following code handles the situation when the projected atom lands exactly on the target atom. Otherwise, `b_j` won't be the integer value and variables `l` and `u` (which correspond to the indices of atoms below and above the projected point):

```
l = np.floor(b_j).astype(np.int64)
u = np.ceil(b_j).astype(np.int64)
eq_mask = u == l
```

```
proj_distr[eq_mask, l[eq_mask]] += next_distr[eq_mask, atom]
```

When the projected point lands between atoms, we need to spread the probability of the source atom between the atoms below and above. This is carried out by two lines in the following code:

```
ne_mask = u != 1
proj_distr[ne_mask, l[ne_mask]] += next_distr[ne_mask, atom] * (u - b_j)[ne_mask]
proj_distr[ne_mask, u[ne_mask]] += next_distr[ne_mask, atom] * (b_j - l)[ne_mask]
```

Of course, we need to properly handle the final transitions of episodes. In that case, our projection shouldn't take into account the next distribution and should just have a 1 probability corresponding to the reward obtained.

However, we again need to take into account our atoms and properly distribute this probability if the reward value falls between atoms. This case is handled by the following code branch, which zeroes the resulting distribution for samples with the done flag set and then calculates the resulting projection:

```
if dones.any():
    proj_distr[dones] = 0.0
    tz_j = np.minimum(Vmax, np.maximum(Vmin, rewards[dones]))
    b_j = (tz_j - Vmin) / delta_z
    l = np.floor(b_j).astype(np.int64)
    u = np.ceil(b_j).astype(np.int64)
    eq_mask = u == 1
    eq_dones = dones.copy()
    eq_dones[dones] = eq_mask
    if eq_dones.any():
        proj_distr[eq_dones, l[eq_mask]] = 1.0
    ne_mask = u != 1
    ne_dones = dones.copy()
    ne_dones[dones] = ne_mask
    if ne_dones.any():
        proj_distr[ne_dones, l[ne_mask]] = (u - b_j)[ne_mask]
        proj_distr[ne_dones, u[ne_mask]] = (b_j - l)[ne_mask]
return proj_distr
```

To give you an illustration of what this function does, let's look at artificially made distributions processed by this function (*Figure 8.21*). I used them to debug the function and make sure that it worked as intended. The code for these checks is in `Chapter08/adhoc/distr_test.py`.

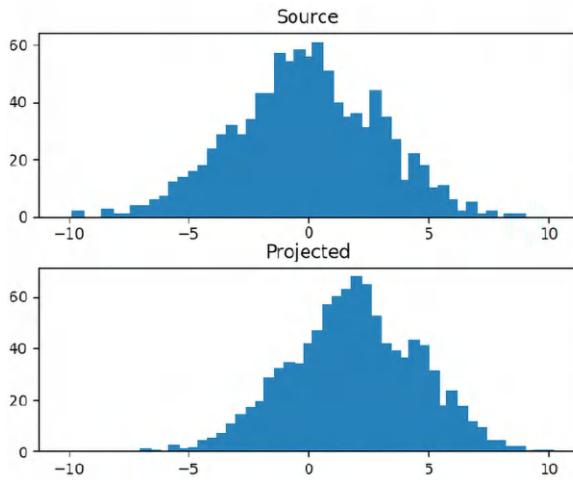


Figure 8.21: The sample of the probability distribution transformation applied to a normal distribution

The top chart of *Figure 8.21* (named *Source*) is a normal distribution with $\mu = 0$ and $\sigma = 3$. The second chart (named *Projected*) is obtained from distribution projection with $\gamma = 0.9$ and is shifted to the right with `reward=2`.

In the situation where we pass `done=True` with the same data, the result will be different and is shown in *Figure 8.22*. In such cases, the source distribution will be ignored completely, and the result will have only the reward projected.

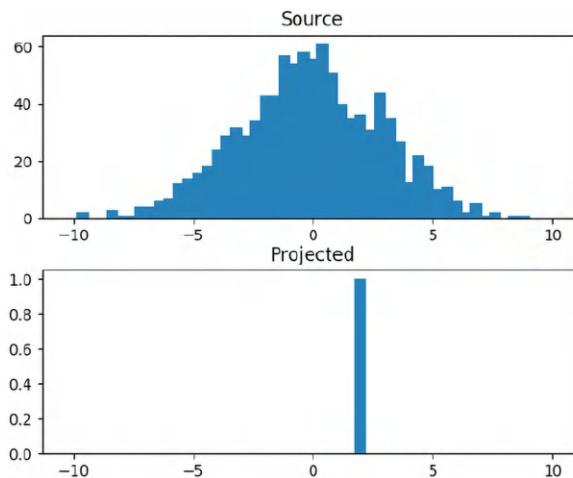


Figure 8.22: The projection of distribution for the final step in the episode

The implementation of this method is in `Chapter08/07_dqn_distrib.py`, which has an optional command-line parameter, `-img-path`. If this option is given, it has to be a directory where plots with a probability distribution from a fixed set of states will be stored during the training. This is useful to monitor how the model converges from uniform probability in the beginning of the training to a more spiked weight of probability masses. Sample images from my experiments are shown in *Figure 8.24* and *Figure 8.25*.

I'm going to show only essential pieces of the implementation here. The core of the method, the `distr_projection` function, was already covered, and it is the most complicated piece. What is still missing is the network architecture and modified loss function, which we will describe here.

Let's start with the network, which is in `lib/dqn_extra.py`, in the `DistributionalDQN` class:

```
Vmax = 10
Vmin = -10
N_ATOMS = 51
DELTA_Z = (Vmax - Vmin) / (N_ATOMS - 1)

class DistributionalDQN(nn.Module):
    def __init__(self, input_shape: tt.Tuple[int, ...], n_actions: int):
        super(DistributionalDQN, self).__init__()

        self.conv = nn.Sequential(
            nn.Conv2d(input_shape[0], 32, kernel_size=8, stride=4),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, stride=1),
            nn.ReLU(),
            nn.Flatten()
        )
        size = self.conv(torch.zeros(1, *input_shape)).size()[-1]
        self.fc = nn.Sequential(
            nn.Linear(size, 512),
            nn.ReLU(),
            nn.Linear(512, n_actions * N_ATOMS)
        )

        supers = torch.arange(Vmin, Vmax + DELTA_Z, DELTA_Z)
        self.register_buffer("supports", supers)
        self.softmax = nn.Softmax(dim=1)
```

The main difference is the output of the fully connected layer. Now it outputs the vector of `n_actions * N_ATOMS` values, which is $6 \times 51 = 306$ for Pong. For every action, it needs to predict the probability distribution on 51 atoms. Every atom (called *support*) has a value, which corresponds to a particular reward.

Those atoms' rewards are evenly distributed from -10 to 10, which gives a grid with step 0.4. Those supports are stored in the network's buffer.

The `forward()` method returns the predicted probability distribution as a 3D tensor (`batch`, `actions`, and `supports`):

```
def forward(self, x: torch.ByteTensor) -> torch.Tensor:
    batch_size = x.size()[0]
    xx = x / 255
    fc_out = self.fc(self.conv(xx))
    return fc_out.view(batch_size, -1, N_ATOMS)

def both(self, x: torch.ByteTensor) -> tt.Tuple[torch.Tensor, torch.Tensor]:
    cat_out = self(x)
    probs = self.apply_softmax(cat_out)
    weights = probs * self.supports
    res = weights.sum(dim=2)
    return cat_out, res
```

Besides `forward()`, we define the `both()` method, which calculates the probability distribution for atoms and Q-values in one call.

The network also defines several helper functions to simplify the calculation of Q-values and apply softmax on the probability distribution:

```
def qvals(self, x: torch.ByteTensor) -> torch.Tensor:
    return self.both(x)[1]

def apply_softmax(self, t: torch.Tensor) -> torch.Tensor:
    return self.softmax(t.view(-1, N_ATOMS)).view(t.size())
```

The final change is the new loss function that has to apply distribution projection instead of the Bellman equation, and calculate KL divergence between predicted and projected distributions:

```
def calc_loss(batch: tt.List[ExperienceFirstLast], net: dqn_extra.DistributionalDQN,
             tgt_net: dqn_extra.DistributionalDQN, gamma: float,
             device: torch.device) -> torch.Tensor:
    states, actions, rewards, dones, next_states = common.unpack_batch(batch)
    batch_size = len(batch)

    states_v = torch.as_tensor(states).to(device)
    actions_v = torch.tensor(actions).to(device)
```

```

next_states_v = torch.as_tensor(next_states).to(device)

# next state distribution
next_distr_v, next_qvals_v = tgt_net.both(next_states_v)
next_acts = next_qvals_v.max(1)[1].data.cpu().numpy()
next_distr = tgt_net.apply_softmax(next_distr_v)
next_distr = next_distr.data.cpu().numpy()

next_best_distr = next_distr[range(batch_size), next_acts]
proj_distr = dqn_extra.distr_projection(next_best_distr, rewards, dones, gamma)

distr_v = net(states_v)
sa_vals = distr_v[range(batch_size), actions_v.data]
state_log_sm_v = F.log_softmax(sa_vals, dim=1)
proj_distr_v = torch.tensor(proj_distr).to(device)

loss_v = -state_log_sm_v * proj_distr_v
return loss_v.sum(dim=1).mean()

```

The preceding code is not very complicated; it just prepares to call `distr_projection` and KL divergence, which is defined as:

$$D_{KL}(P\|Q) = - \sum_i p_i \log q_i$$

To calculate the logarithm of probability, we use the PyTorch `log_softmax` function, which combines both `log` and `softmax` in a numerically stable way.

Results

From my experiments, the distributional version of DQN converged a bit slower and less stably than the original DQN, which is not surprising, as the network output is now 51 times larger and the loss function has changed. Without hyperparameter tuning (which will be described in the next subsection), the distributional version requires 20% more episodes to solve the game.

Another factor that might be important here is that Pong is just too simple a game to draw conclusions. In the *A Distributional Perspective* paper, the authors reported state-of-the-art scores (at the time of publishing in 2017) for more than half of the games from the Atari benchmark (Pong was not among them).

The following are charts comparing reward dynamics and loss for the distributional DQN. As you can see, the reward dynamics for the distributional method is worse than the baseline DQN:

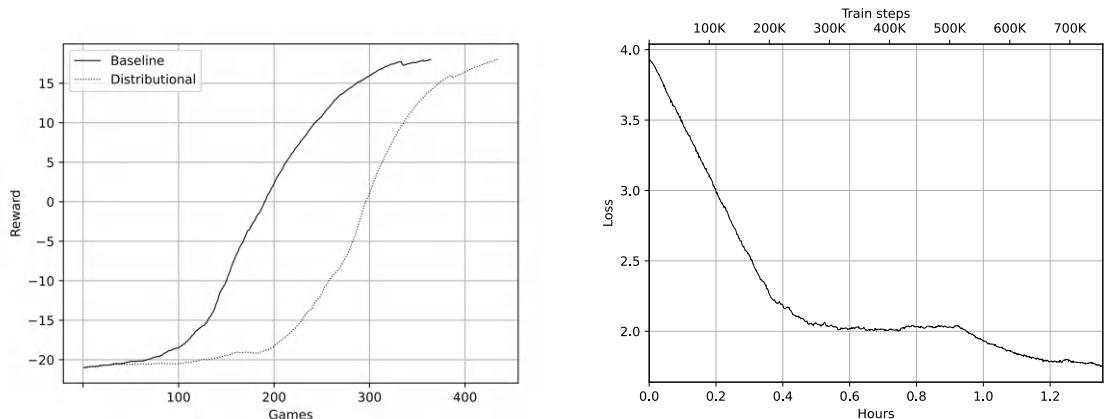


Figure 8.23: Reward dynamics (left) and loss decrease (right)

It might be interesting to look into the dynamics of the probability distribution during the training. If you start the training with the `-img-path` parameter (providing the directory name), the training process will save plots with the probability distribution for a fixed set of states. For example, the following figure shows the probability distribution for all six actions for one state at the beginning of the training (after 30k frames):

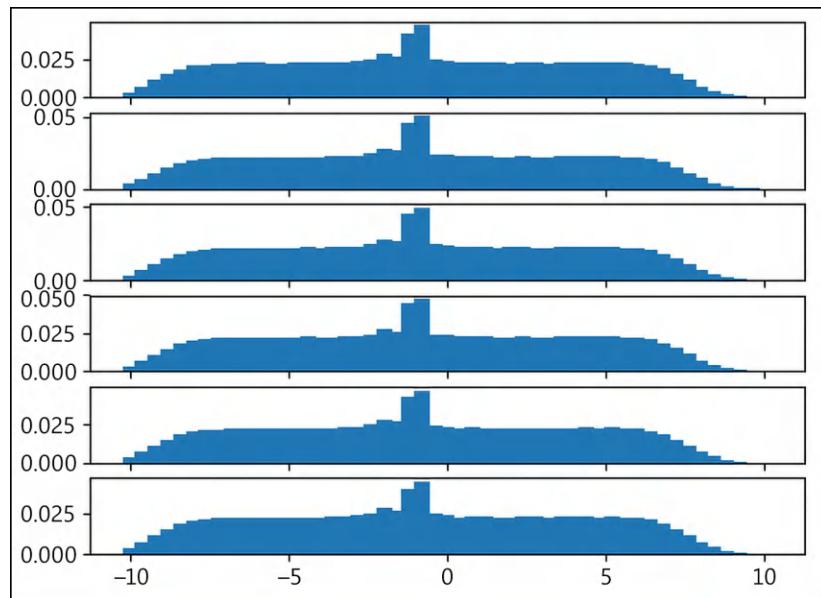


Figure 8.24: Probability distribution at the beginning of training

All the distributions are very wide (as the network hasn't converged yet), and the peak in the middle corresponds to the negative reward that the network expects to get from its actions. The same state after 500k frames of training is shown in the following figure:

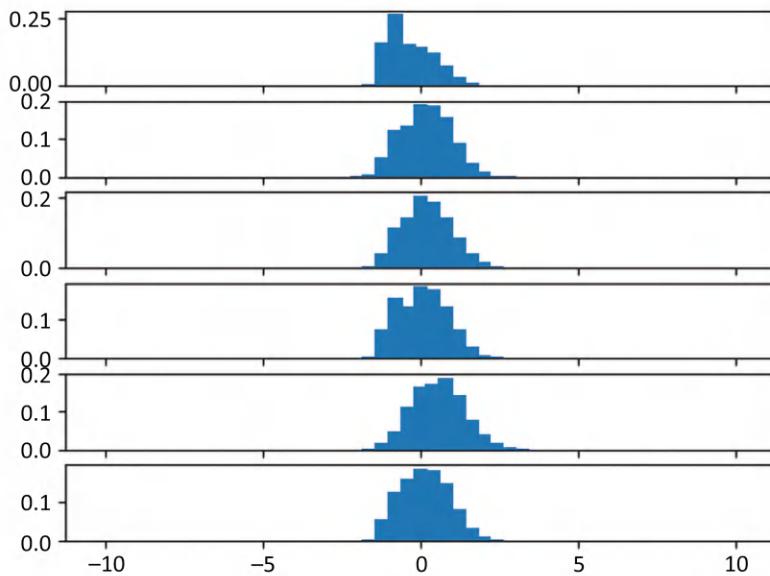


Figure 8.25: Probability distribution produced by the trained network

Now we can see that different actions have different distributions. The first action (which corresponds to the NOOP, the *do nothing action*) has its distribution shifted to the left, so doing nothing in this state usually leads to losing. The fifth action, which is RIGHTFIRE, has the mean value shifted to the right, so this action leads to a better score.

Hyperparameter tuning

The tuning of hyperparameters was not very fruitful. After 30 tuning iterations, there were no combinations of learning rate and gamma that were able to converge faster than the common set of parameters.

Combining everything

You have now seen all the DQN improvements mentioned in the paper *Rainbow: Combining Improvements in Deep Reinforcement Learning*, but it was done in an incremental way, which (I hope) was helpful to understand the idea and implementation of every improvement. The main point of the paper was to combine those improvements and check the results.

In the final example, I've decided to exclude categorical DQN and double DQN from the final system, as they haven't shown too much improvement on our guinea pig environment. If you want, you can add them and try using a different game. The complete example is available in `Chapter08/08_dqn_rainbow.py`.

First of all, we need to define our network architecture and the methods that have contributed to it:

- **Dueling DQN:** Our network will have two separate paths for the value of the state distribution and advantage distribution. On the output, both paths will be summed together, providing the final value probability distributions for actions. To force the advantage distribution to have a zero mean, we will subtract the distribution with the mean advantage in every atom.
- **Noisy networks:** Our linear layers in the value and advantage paths will be noisy variants of `nn.Linear`.

In addition to network architecture changes, we will use the prioritized replay buffer to keep environment transitions and sample them proportionally to the MSE loss.

Finally, we will unroll the Bellman equation to n-steps.

I'm not going to repeat all the code, as individual methods have already been given in the preceding sections, and it should be obvious what the final result of combining the methods will look like. If you have any trouble, you can find the code on GitHub.

Results

The following are charts comparing the smoothed reward and count of steps with the baseline DQN. In both, we can see significant improvement in terms of the amount of games played:

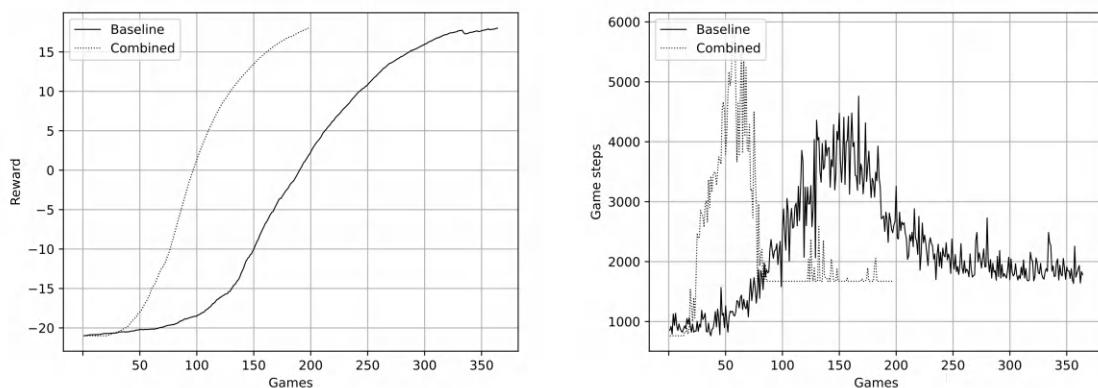


Figure 8.26: Comparison of baseline DQN with combined system

In addition to the averaged reward, it is worth checking the raw reward chart, which is even more dramatic than the smoothed reward. It shows that our system was able to jump from the negative outcome to the positive very quickly – after just 100 games, it won almost every game. So, it took us another 100 games to make the smoothed reward reach +18:

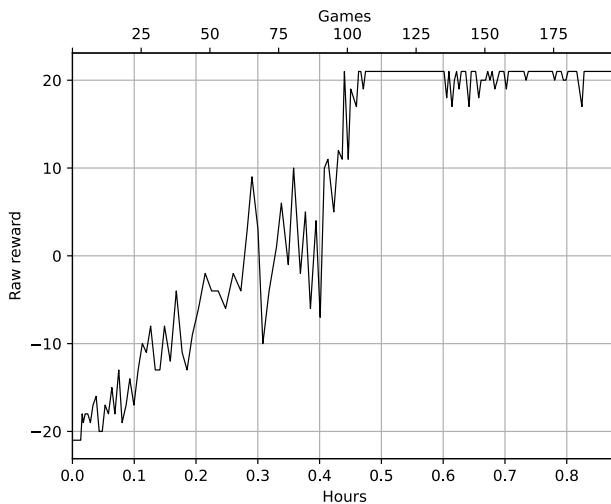


Figure 8.27: Raw reward for combined system

As a downside, the combined system is slower than the baseline, as we have a more complicated NN architecture and prioritized replay buffer. The FPS chart shows that the combined system starts at 170 FPS and degrades to 130 FPS due to the $\mathcal{O}(n)$ buffer complexity:

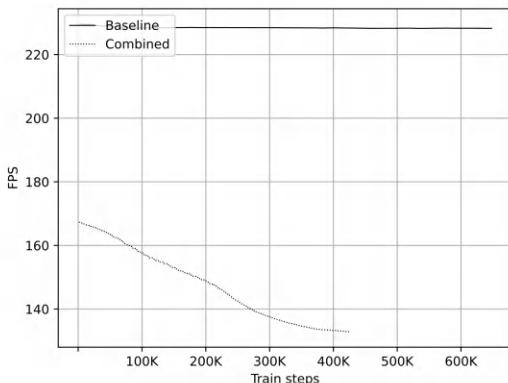


Figure 8.28: Performance comparison (in frames per second)

Hyperparameter tuning

Tuning was done as before and was able to further improve the combined system training in terms of games played before solving the game. The following are charts comparing the tuned baseline DQN with the tuned combined system:

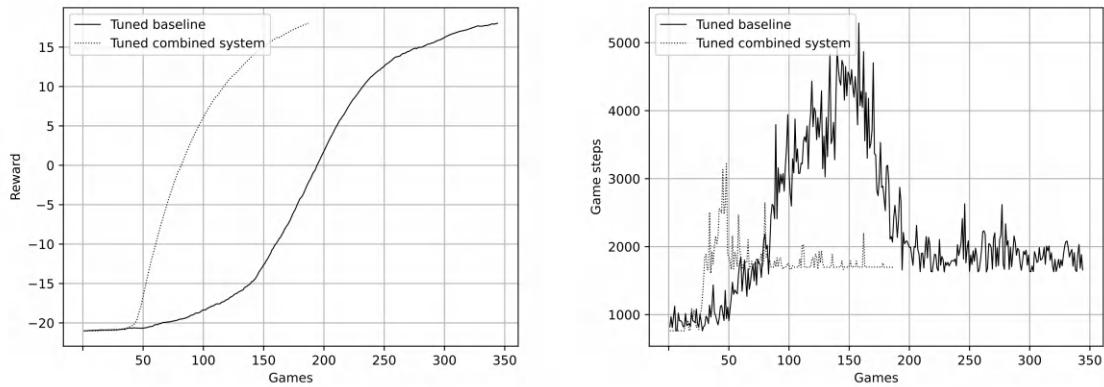


Figure 8.29: Comparison of tuned baseline DQN with tuned combined system

Another chart showing the effect of the tuning is the comparison of raw game rewards before and after the tuning. The tuned system starts to get the maximum score even earlier — just after 40 games, which is quite impressive:

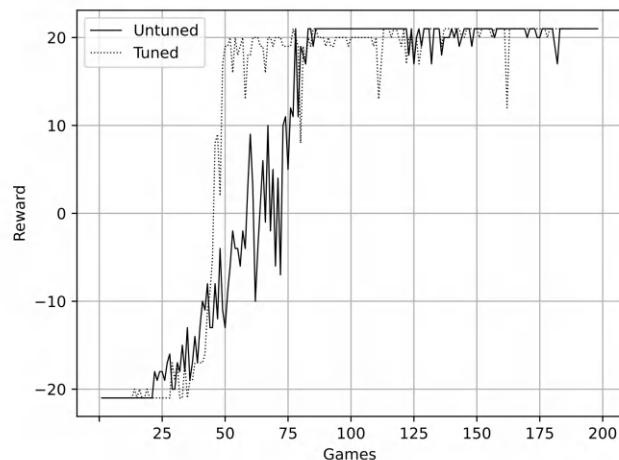


Figure 8.30: Raw reward for untuned and tuned combined DQN

Summary

In this chapter, we have walked through and implemented a lot of DQN improvements that have been discovered by researchers since the first DQN paper was published in 2015. This list is far from complete. First of all, for the list of methods, I used the paper *Rainbow: Combining improvements in deep reinforcement learning* [Hes+18], which was published by DeepMind, so the list of methods is definitely biased to DeepMind papers. Secondly, RL is so active nowadays that new papers come out almost every day, which makes it very hard to keep up, even if we limit ourselves to one kind of RL model, such as a DQN. The goal of this chapter was to give you a practical view of different ideas that the field has developed.

In the next chapter, we will continue discussing practical DQN applications from an engineering perspective by talking about ways to improve DQN performance without touching the underlying method.

Join our community on Discord

Read this book alongside other users, Deep Learning experts, and the author himself.

Ask questions, provide solutions to other readers, chat with the author via Ask Me Anything sessions, and much more.

Scan the QR code or visit the link to join the community.

<https://packt.link/r1>



9

Ways to Speed Up RL

In *Chapter 8*, you saw several practical tricks to make the **deep Q-network (DQN)** method more stable and converge faster. They involved basic DQN method modifications (like injecting noise into the network or unrolling the Bellman equation) to get a better policy, with less time spent on training. But in this chapter, we will explore another way to do this: tweaking the implementation details of the method to improve the speed of the training. This is a pure engineering approach, but it's also important since it is useful in practice.

In this chapter, we will:

- Take the Pong environment from the previous chapter and try to get it solved as fast as possible
- In a step-by-step manner, get Pong solved almost 2 times faster using exactly the same commodity hardware

Why speed matters

First, let's talk a bit about why speed is important and why we optimize it at all. It might not be obvious, but enormous hardware performance improvements have happened in the last decade or two. Almost 20 years ago, I was involved with a project that focused on building a supercomputer for **computational fluid dynamics (CFD)** simulations performed by an aircraft engine design company. The system consisted of 64 servers, occupied three 42-inch racks, and required dedicated cooling and power subsystems. The hardware alone (without cooling) cost around \$1M.

In 2005, this supercomputer was ranked fourth among ex-USSR supercomputers and was the fastest system installed in the industry. Its theoretical performance was 922 GFLOPS (almost a trillion floating-point operations per second), but in comparison to the GTX 1080 Ti released 12 years later, all the capabilities of this pile of iron look tiny.

One single GTX 1080 Ti is able to perform 11,340 GFLOPS, which is 12.3 times more than what supercomputers from 2005 could do. And the price was only \$700 per GPU when it was released! If we count computation power per \$1, we get a price drop of more than 17,500 times for every GFLOP. This number is even more dramatic with the latest (at the time of writing) H100 GPU, which provides 134 teraflops (with FP32 operations).

It has been said many times that **artificial intelligence (AI)** progress (and **machine learning (ML)** in general) is being driven by data availability and computing power increases, and I believe that this is absolutely true. Imagine some computations that require a month to complete on one machine (a very common situation in CFD and other physics simulations). If we are able to increase speed by five times, this month of patient waiting will turn into six days. Speeding up by 100 times will mean that this heavy one-month computation will end up taking eight hours, so you could have three of them done in just one day! It's very cool to be able to get 20,000 times more power for the same money nowadays. By the way, speeding up by 20k times will mean that our one-month problem will be done in two to three minutes!

This has happened not only in the “big iron” (also known as *high-performance computing*) world; basically, it is everywhere. Modern microcontrollers have the performance characteristics of the desktops that we worked with 15 years ago (for example, you can build a pocket computer for \$50, with a 32-bit microcontroller running at 120 MHz, that is able to run the Atari 2600 emulator: <https://hackaday.io/project/80627-badge-for-hackaday-conference-2018-in-belgrade>). I’m not even talking about modern smartphones, which normally have four to eight cores, a **graphics processing unit (GPU)**, and several GB of RAM.

Of course, there are a lot of complications there. It’s not just taking the same code that you used a decade ago and now, magically, finding that it works several thousand times faster. It might be the opposite: you might not be able to run it at all, due to a change in libraries, operating system interfaces, and other factors. (Have you ever tried to read old CD-RW disks written just a decade ago?) Nowadays, to get the full capabilities of modern hardware, you need to parallelize your code, which automatically means tons of details about distributed systems, data locality, communications, and the internal characteristics of the hardware and libraries. High-level libraries try to hide all those complications from you, but you can’t ignore all of them if you want to use these libraries efficiently. However, it is definitely worth it – one month of patient waiting could be turned into three minutes, remember.

On the other hand, it might not be fully obvious why we need to speed things up in the first place. One month is not that long, after all; just lock the computer in a server room and go on vacation! But think about the process involved in preparing and making this computation work. You might already have noticed that even simple ML problems can be almost impossible to implement properly on the first attempt.

They require many trial runs before you find good hyperparameters and fix all the bugs and code ready for a clean launch. There is exactly the same process in physics simulations, RL research, big data processing, and programming in general. So, if we are able to make something run faster, it's not only beneficial for the single run but also enables us to iterate quickly and do more experiments with code, which might significantly speed up the whole process and improve the quality of the final result.

I remember one situation from my career when we deployed a Hadoop cluster in our department, where we were developing a web search engine (similar to Google, but for Russian websites). Before the deployment, it took several weeks to conduct even simple experiments with data. Several terabytes of data were lying on different servers; you needed to run your code several times on every machine, gather and combine intermediate results, deal with occasional hardware failures, and do a lot of manual tasks not related to the problem that you were supposed to solve. After integrating the Hadoop platform into the data processing, the time needed for experiments dropped to several hours, which was completely game-changing. Since then, developers have been able to conduct many experiments much more easily and faster without bothering with unnecessary details. The number of experiments (and willingness to run them) has increased significantly, which has also increased the quality of the final product.

Another reason in favor of optimization is the size of problems that we can deal with. Making some method run faster might mean two different things: we can get the results sooner, or we can increase the size (or some other measure of the problem's complexity). A complexity increase might have different meanings in different cases, like getting more accurate results, making fewer simplifications of the real world, or taking into account more data, but, almost always, this is a good thing.

Returning to the main topic of the book, let's outline how RL methods might benefit from speed-ups. First of all, even state-of-the-art RL methods are not very sample efficient, which means that training needs to communicate with the environment many times (in the case of Atari, millions of times) before learning a good policy, and that might mean weeks of training. If we can speed up this process a bit, we can get the results faster, do more experiments, and find better hyperparameters. Besides this, if we have faster code, we can even increase the complexity of the problems that they are applied to.

In modern RL, Atari games are considered solved; even so-called "hard-exploration games," like *Montezuma's Revenge*, can be trained to superhuman accuracy.

Therefore, new frontiers in research require more complex problems, with richer observation and action spaces, which inevitably require more training time and more hardware. Such research has already been started (and has increased the complexity of problems a bit too much, from my point of view) by DeepMind and OpenAI, which have switched from Atari to much more challenging problems like protein folding (AlphaFold system) and **Large Language Models (LLMs)**. Those problems require thousands of GPUs working in parallel.

I want to end this introduction with a small warning: all performance optimizations make sense only when the core method is working properly (which is not always obvious in cases of RL and ML in general). As an instructor of an online course about performance optimizations said, “It’s much better to have a slow and correct program than a fast but incorrect one.”

Baseline

In this chapter, we will take the Atari Pong environment that you are already familiar with and try to speed up its convergence. As a baseline, we will take the same simple DQN that we used in *Chapter 8*, and the hyperparameters will also be the same. To compare the effect of our changes, we will use two characteristics:

- The **number of frames** that we consume from the environment every second (FPS). This indicates how fast we can communicate with the environment during the training. It is very common in RL papers to indicate the number of frames that the agent observed during the training; normal numbers are 25M–50M frames. So, if our FPS=200, it will take $\frac{50 \cdot 10^6}{200 \cdot 60 \cdot 60 \cdot 24} \approx 2.89$ days. In such calculations, you need to take into account that RL papers commonly report raw environment frames. But if frame skip is used (and it almost always is), the count of frames needs to be divided by this factor, which is commonly equal to 4. In our measurements, we calculate FPS in terms of agent communications with the environment, so the “raw environment FPS” will be four times larger.
- The **wall clock time** before the game is solved. We stop training when the smoothed reward for the last 100 episodes reaches 18 (the maximum score in Pong is 21.) This boundary could be increased, but normally 18 is a good indication that the agent has almost mastered the game and polishing the policy to perfection is just a matter of the training time. We check the wall clock time because FPS alone is not the best indicator of training speed-up.

Due to our manipulations performed with the code, we can get a very high FPS, but convergence might suffer. This value alone also can’t be used as a reliable characteristic of our improvements, as the training process is stochastic. Even by specifying random seeds (we need to set seeds explicitly for PyTorch, Gym, and NumPy), parallelization (which will be used in subsequent steps) adds randomness to the process, which is almost impossible to avoid. So, the best we can do is run the benchmark several times and average the results. But one single run’s outcome can’t be used to make any decisions.

Because of the randomness mentioned above, all the charts in this chapter were obtained from averaging 5 runs of the same experiment. All the benchmarks use the same machine with an Intel i5-7600K CPU, a GTX 1080 Ti GPU with CUDA 12.3, and NVIDIA drivers version 545.29.06.

Our first benchmark will be our baseline version, which is in `Chapter09/01_baseline.py`. I will not provide the source code here, as it has already been given in the previous chapter and is the same here. During the training, the code writes into TensorBoard several metrics:

- `reward`: The raw undiscounted reward from the episode; the x axis is the episode number.
- `avg_reward`: The same as `reward` but smoothed by running the average with $\alpha = 0.98$.
- `steps`: The number of steps that the episode lasted. Normally, in the beginning, the agent loses very quickly, so every episode is around 1,000 steps. Then, it learns how to act better, so the number of steps increases to 3,000–4,000 with the reward increase; but, in the end, when the agent masters the game, the number of steps drops back to 2,000 steps, as the policy is polished to win as quickly as possible (due to the discount factor γ). In fact, this drop in episode length might be an indication of overfitting to the environment, which is a huge problem in RL. However, dealing with this issue is beyond the scope of our experiments.
- `loss`: The loss during the training, sampled every 100 iterations. It should be around $2 \cdot 10^{-3} \dots 1 \cdot 10^{-2}$, with occasional increases when the agent discovers new behavior, leading to a different reward from that learned by the Q-value.
- `avg_loss`: A smoothed version of the loss.
- `epsilon`: The current value of ϵ – probability of taking the random action.
- `avg_fps`: The speed of agent communication with the environment (observations per second), smoothed with a running average.

In *Figure 9.1* and *Figure 9.2*, the charts are averaged from 5 baseline runs. As before, each chart is drawn with two x axes: the bottom one is the wall clock time in hours, and the top is the step number (episode in *Figure 9.1* and training iteration in *Figure 9.2*):

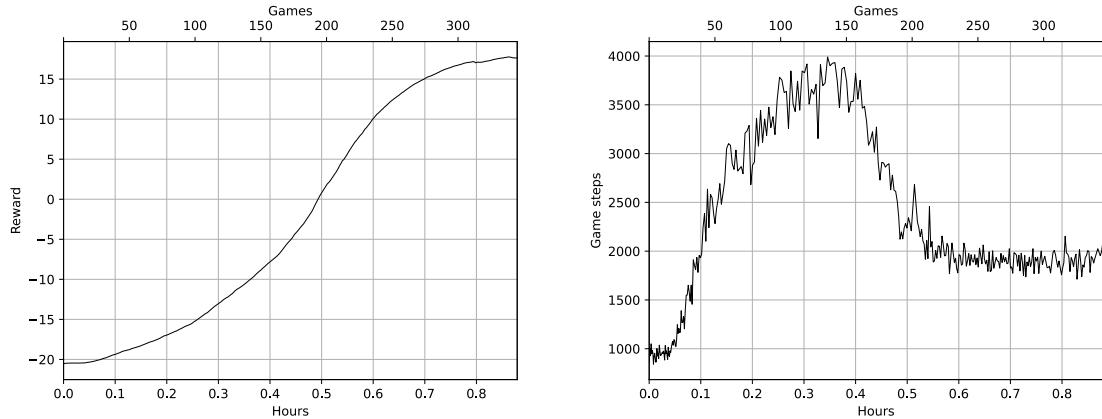


Figure 9.1: Reward and episode length in baseline version

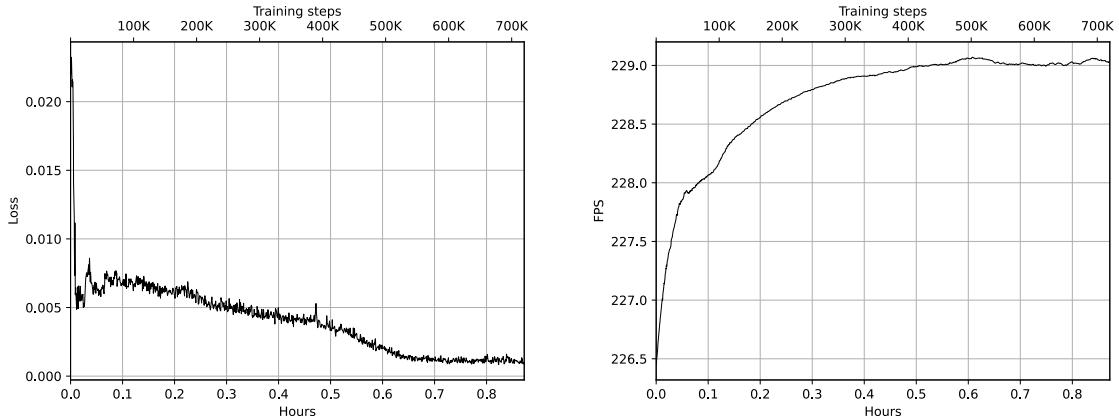


Figure 9.2: Loss and FPS during the training of baseline version

The computation graph in PyTorch

Our first examples won't be around speeding up the baseline, but will show one common, and not always obvious, situation that can cost you performance. In *Chapter 3*, we discussed the way PyTorch calculates gradients: it builds the graph of all operations that you perform on tensors, and when you call the `backward()` method of the final loss, all gradients in the model parameters are automatically calculated.

This works well, but RL code is normally much more complex than traditional supervised learning training, so the RL model that we are currently training is also being applied to get the actions that the agent needs to perform in the environment. The target network discussed in *Chapter 6* makes it even more tricky. So, in DQN, a **neural network (NN)** is normally used in three different situations:

- When we want to calculate Q-values predicted by the network to get the loss in respect to reference Q-values approximated by the Bellman equation
- When we apply the target network to get Q-values for the next state to calculate a Bellman approximation
- When the agent wants to make a decision about the action to perform

In our training, we need gradients calculated only for the first situation. In *Chapter 6*, we avoided gradients by explicitly calling `detach()` on the tensor returned by the target network. This `detach` is very important, as it prevents gradients from flowing into our model “from the unexpected direction” and, without this, the DQN might not converge at all. In the third situation, gradients were stopped by converting the network result into a NumPy array.

Our code in *Chapter 6*, worked, but we missed one subtle detail: the computation graph that is created for all three situations. This is not a major problem, but creating the graph still uses some resources (in terms of both speed and memory), which are wasted because PyTorch creates this computation graph even if we don’t call `backward()` on some graph. To prevent this, one very nice option exists: the decorator `torch.no_grad()`.

Decorators in Python is a very wide topic. They give the developer a lot of power (when properly used), but are well beyond the scope of this book. Here, I’ll just give an example where we define two functions:

```
>>> import torch
>>> @torch.no_grad
... def fun_a(t):
...     return t*2
...
>>> def fun_b(t):
...     return t*2
...
```

Both these functions are doing the same thing, doubling its argument, but the first function is declared with `torch.no_grad()` and the second is just a normal function. This decorator temporarily disables gradient computation for all tensors passed to the function.

As you can see, although the tensor, `t`, requires grad, the result from `fun_a` (the decorated function) doesn't have gradients:

```
>>> t = torch.ones(3, requires_grad=True)
>>> t
tensor([1., 1., 1.], requires_grad=True)
>>> a = fun_a(t)
>>> b = fun_b(t)
>>> b
tensor([2., 2., 2.], grad_fn=<MulBackward0>)
>>> a
tensor([2., 2., 2.])
```

But this effect is bounded inside the decorated function:

```
>>> a*t
tensor([2., 2., 2.], grad_fn=<MulBackward0>)
```

The function `torch.no_grad()` also could be used as a *context manager* (another powerful Python concept that I recommend you learn about) to stop gradients in some chunk of code:

```
>>> with torch.no_grad():
...     c = t*2
...
>>> c
tensor([2., 2., 2.])
```

This functionality provides you with a very convenient way to indicate parts of your code that should be excluded from the gradient machinery completely. This has already been done in `ptan.agent.DQNAgent` (and other agents provided by PTAN) and in the `common.calc_loss_dqn` function. But if you are writing a custom agent or implementing your own code, it might be very easy to forget about this.

To benchmark the effect of unnecessary graph calculation, I've provided the modified baseline code in `Chapter09/00_slow_grads.py`, which is exactly the same, but the agent and loss calculations are copied without `torch.no_grad()`. The following charts show the effect of this:

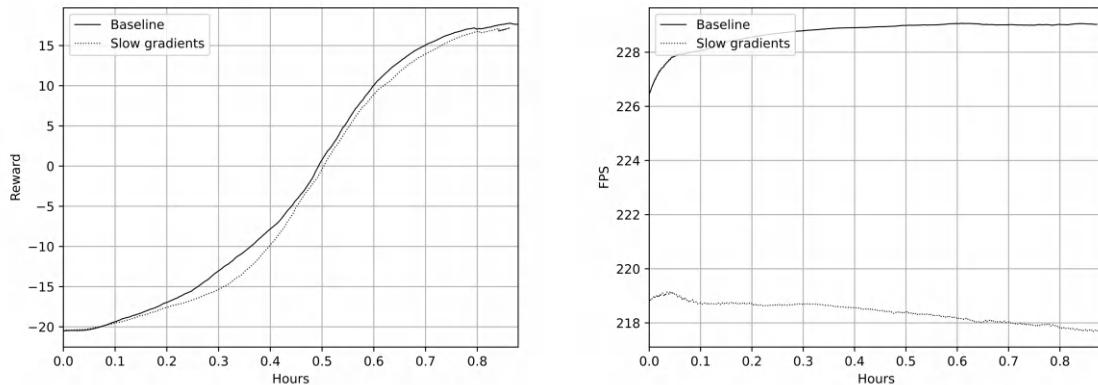


Figure 9.3: A comparison of reward and FPS between the baseline and version without `torch.no_grad()`

As you can see, the speed penalty is not that large (around 10 FPS), but that might become different in the case of a larger network with a more complicated structure. I've seen a 50% performance boost in more complex recurrent NNs obtained after adding `torch.no_grad()`.

Several environments

The first idea that we usually apply to speed up deep learning training is *larger batch size*. It's also applicable to the domain of deep RL, but you need to be careful here. In the normal supervised learning case, the simple rule "a large batch is better" is usually true: you just increase your batch as your GPU memory allows, and a larger batch normally means more samples will be processed in a unit of time thanks to enormous GPU parallelism.

The RL case is slightly different. During the training, two things happen simultaneously:

- Your network is trained to get better predictions on the current data
- Your agent explores the environment

As the agent explores the environment and learns about the outcome of its actions, the training data changes. In a shooter example, your agent can run randomly for a time while being shot by monsters and have only a miserable "death is everywhere" experience in the training buffer. But after a while, the agent will discover that it has a weapon it can use. This new experience can dramatically change the data that we are using for training.

RL convergence usually lies on a fragile balance between training and exploration. If we just increase a batch size without tweaking other options, we can easily overfit to the current data (for our shooter example, your agent can start thinking that “dying young” is the only option to minimize suffering and may never discover the gun it has).

So, in the example in `Chapter09/02_n_envs.py`, our agent uses several copies of the same environment to gather the training data. On every training iteration, we populate our replay buffer with samples from all those environments and then sample a proportionally larger batch size. This also allows us to speed up *inference time* a bit, as we can make a decision about the actions to execute for all N environments in one forward pass of the NN. In terms of implementation, the preceding logic requires just a couple of changes in the code:

- As PTAN supports several environments out of the box, what we need to do is just pass N Gym environments to the `ExperienceSource` instance
- The agent code (in our case, `DQNAgent`) is already optimized for the batched application of the NN

Several pieces of code were changed to address this. The function that generates batches now performs multiple steps (equal to the total number of environments) for every training iteration:

```
def batch_generator(buffer: ptan.experience.ExperienceReplayBuffer,
                    initial: int, batch_size: int, steps: int):
    buffer.populate(initial)
    while True:
        buffer.populate(steps)
        yield buffer.sample(batch_size)
```

The experience source accepts the array of environments instead of a single environment:

```
envs = [
    ptan.common.wrappers.wrap_dqn(gym.make(params.env_name))
    for _ in range(args.envs)
]
params.batch_size *= args.envs
exp_source = ptan.experience.ExperienceSourceFirstLast(
    envs, agent, gamma=params.gamma, env_seed=common.SEED)
```

Other changes are just minor tweaks of constants to adjust the FPS tracker and compensated speed of epsilon decay (ratio of random steps).

As the number of environments is the new hyperparameter that needs to be tuned, I ran several experiments with N from 2 ... 6. The following charts show the averaged dynamics:

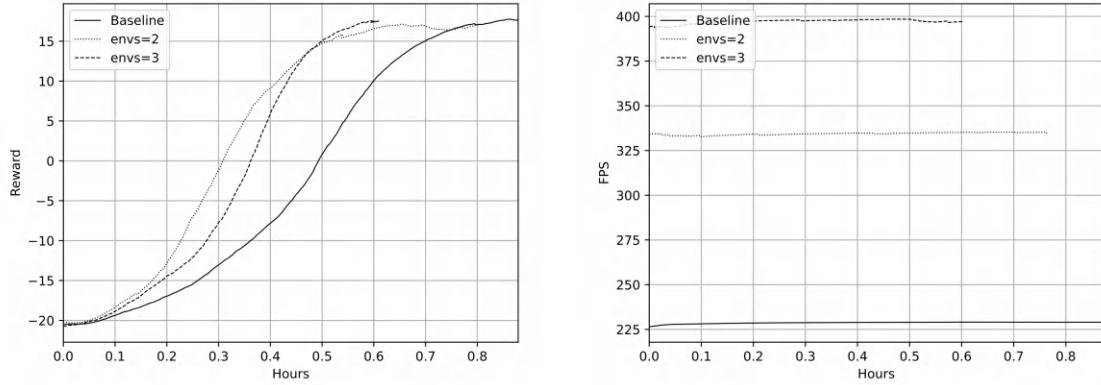


Figure 9.4: Reward and FPS in the baseline, two, and three environments

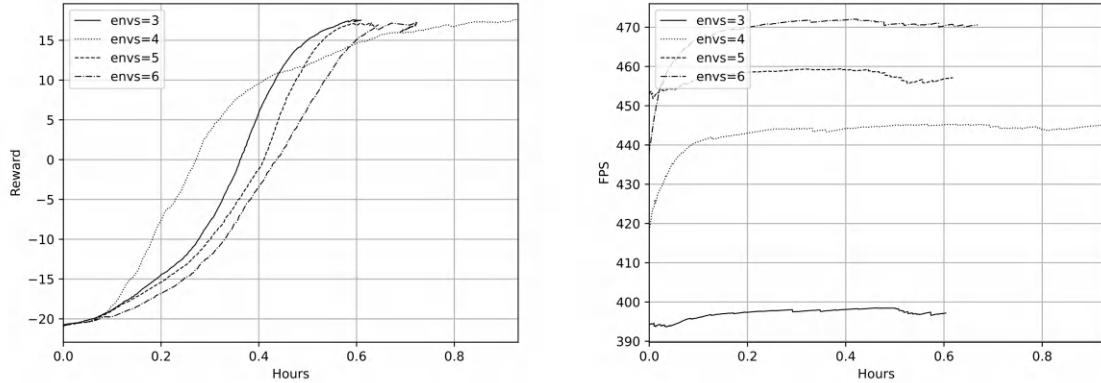


Figure 9.5: Reward and FPS for $n = 3 \dots 6$

As you can see from the charts, adding an extra environment provided a 47% gain in FPS (from 227 FPS to 335 FPS) and sped up the convergence about 10% (from 52 minutes to 48 minutes). The same effect came from adding the third environment (398 FPS, and 36 minutes), but adding more environments had a negative effect on convergence speed, despite a further increase in FPS. So, it looks like $N = 3$ is more or less the optimal value for our hyperparameter, but, of course, you are free to tweak and experiment. It also illustrates why we're monitoring not just raw speed in FPS but also how quickly the agent is able to solve the game.

Playing and training in separate processes

At a high level, our training contains a repetition of the following steps:

1. Ask the current network to choose actions and execute them in our array of environments.
2. Put observations into the replay buffer.
3. Randomly sample the training batch from the replay buffer.
4. Train on this batch.

The purpose of the first two steps is to populate the replay buffer with samples from the environment (which are observation, action, reward, and next observation). The last two steps are for training our network.

The following is an illustration of the preceding steps that will make potential parallelism slightly more obvious. On the left, the training flow is shown. The training steps use environments, the replay buffer, and our NN. The solid lines show data and code flow. Dotted lines represent usage of the NN for training and inference.

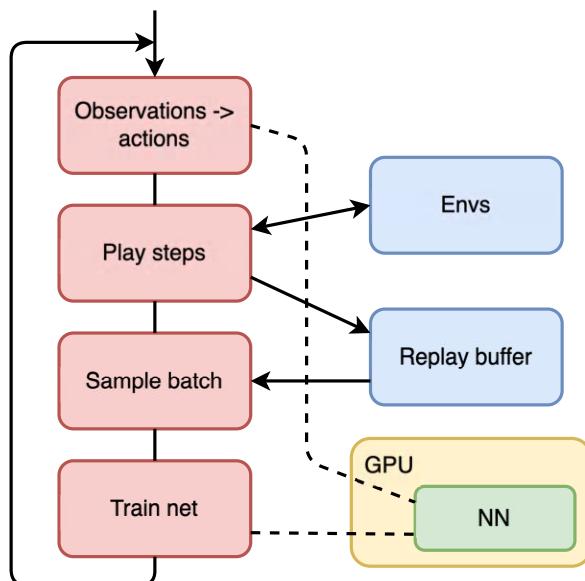


Figure 9.6: A sequential diagram of the training process

As you can see, the top two steps communicate with the bottom only via the replay buffer and NN. This makes it possible to separate those two parts in different parallel processes.

The following figure is a diagram of the scheme:

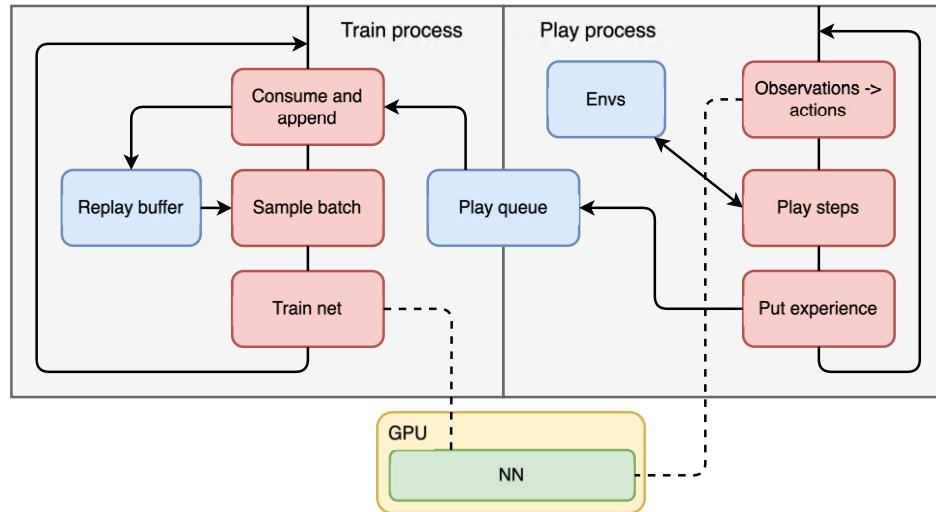


Figure 9.7: The parallel version of the training and play steps

In the case of our Pong environment, it might look like an unnecessary complication of the code, but this separation might be extremely useful in some cases. Imagine that you have a very slow and heavy environment, so every step takes seconds of computations. That's not a contrived example; for instance, past NeurIPS competitions, such as *Learning to Run*, *AI for Prosthetics Challenge*, and *Learn to Move* (<https://www.aicrowd.com/challenges/neurips-2019-learn-to-move-walk-around>), have very slow neuromuscular simulators, so you have to separate experience gathering from the training process. In such cases, you can have many concurrent environments that deliver the experience to the central training process.

To turn our serial code into parallel code, some modifications are needed. In the file `Chapter09/03_parallel.py`, you can find the full source of the example. In the following, I'll focus only on major differences.

First, we use the `torch.multiprocessing` module as a drop-in replacement for the standard Python multiprocessing module:

```

import torch.multiprocessing as mp

@dataclass
class EpisodeEnded:
    reward: float
    steps: int
    epsilon: float

```

```

def play_func(params: common.Hyperparams, net: dqn_model.DQN,
             dev_name: str, exp_queue: mp.Queue):
    env = gym.make(params.env_name)
    env = ptan.common.wrappers.wrap_dqn(env)
    device = torch.device(dev_name)

    selector = ptan.actions.EpsilonGreedyActionSelector(epsilon=params.epsilon_start)
    epsilon_tracker = common.EpsilonTracker(selector, params)
    agent = ptan.agent.DQNAgent(net, selector, device=device)
    exp_source = ptan.experience.ExperienceSourceFirstLast(
        env, agent, gamma=params.gamma, env_seed=common.SEED)

    for frame_idx, exp in enumerate(exp_source):
        epsilon_tracker.frame(frame_idx//2)
        exp_queue.put(exp)
        for reward, steps in exp_source.pop_rewards_steps():
            ee = EpisodeEnded(reward=reward, steps=steps, epsilon=selector.epsilon)
            exp_queue.put(ee)

```

The version from the standard library provides several primitives to work with code executed in separated processes, such as `mp.Queue` (distributed queue), `mp.Process` (child process), and others. PyTorch provides a wrapper around the standard multiprocessing library, which allows torch tensors to be shared between processes without copying them. This is implemented using shared memory in the case of CPU tensors, or CUDA references for tensors on a GPU. This sharing mechanism removes the major bottleneck when communication is performed within a single computer. Of course, in the case of truly distributed communications, you need to serialize data yourself.

The function `play_func` implements our “play process” and will be running in a separate child process started by the main process. Its responsibility is to get experience from the environment and push it into the shared queue. In addition, it wraps information about the end of the episode into a dataclass and pushes it into the same queue to keep the training process informed about the episode reward and the number of steps.

The function `batch_generator` is replaced by the class `BatchGenerator`:

```

class BatchGenerator:
    def __init__(self, buffer_size: int, exp_queue: mp.Queue,
                 fps_handler: ptan_ignite.EpisodeFPSHandler,
                 initial: int, batch_size: int):
        self.buffer = ptan.experience.ExperienceReplayBuffer(
            experience_source=None, buffer_size=buffer_size)
        self.exp_queue = exp_queue

```

```

        self.fps_handler = fps_handler
        self.initial = initial
        self.batch_size = batch_size
        self._rewards_steps = []
        self.epsilon = None

    def pop_rewards_steps(self) -> tt.List[tt.Tuple[float, int]]:
        res = list(self._rewards_steps)
        self._rewards_steps.clear()
        return res

    def __iter__(self):
        while True:
            while self.exp_queue.qsize() > 0:
                exp = self.exp_queue.get()
                if isinstance(exp, EpisodeEnded):
                    self._rewards_steps.append((exp.reward, exp.steps))
                    self.epsilon = exp.epsilon
                else:
                    self.buffer._add(exp)
                    self.fps_handler.step()
            if len(self.buffer) < self.initial:
                continue
            yield self.buffer.sample(self.batch_size)

```

This class provides an iterator over batches and additionally mimics the `ExperienceSource` interface with the method `pop_reward_steps()`. The logic of this class is simple: it consumes the queue (populated by the “play process”), and if the `EpisodeEnded` object was received, it remembers information about epsilon and the count of steps the game took; otherwise, the object is a piece of experience that needs to be added into the replay buffer. From the queue, we consume all objects available at the moment, and then the training batch is sampled from the buffer and yielded.

In the beginning of the training process, we need to tell `torch.multiprocessing` which start method to use:

```

if __name__ == "__main__":
    warnings.simplefilter("ignore", category=UserWarning)
    mp.set_start_method('spawn')

```

There are several of them, but `spawn` is the most flexible.

Then, the queue for communication is created, and we start our `play_func` as a separate process. As arguments, we pass the NN, hyperparameters, and queue to be used for experience:

```
exp_queue = mp.Queue(maxsize=2)
proc_args = (params, net, args.dev, exp_queue)
play_proc = mp.Process(target=play_func, args=proc_args)
play_proc.start()
```

The rest of the code is almost the same, with the exception that we use a `BatchGenerator` instance as the data source for Ignite and for `EndOfEpisodeHandler` (which requires the method `pop_rewards_steps()`). The following charts were obtained from my benchmarks:

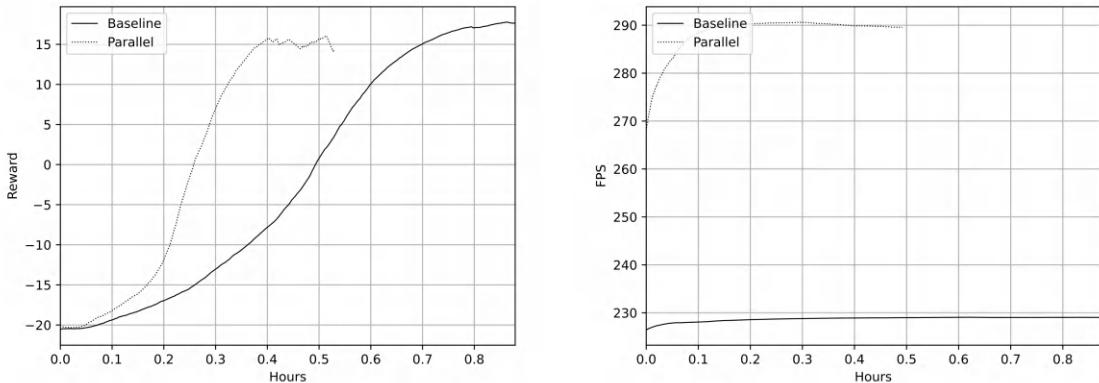


Figure 9.8: Reward and FPS in the baseline and parallel version

As you can see, in terms of FPS, we got an increase of 27%: 290 FPS in the parallel version versus 228 in the baseline. The average time to solve the environment decreased by 41%.

In terms of FPS increase, the parallel version looks worse than the best result from the previous section (with 3 game environments, we got almost 400 FPS), but the convergence speed is better.

Tweaking wrappers

The final step in our sequence of experiments will be tweaking wrappers applied to the environment. This is very easy to overlook, as wrappers are normally written once or just borrowed from other code, applied to the environment, and left to sit there. But you should be aware of their importance in terms of the speed and convergence of your method.

For example, the normal DeepMind-style stack of wrappers applied to an Atari game looks like this:

1. `NoopResetEnv`: Applies a random amount of NOOP operations to the game reset. In some Atari games, this is needed to remove weird initial observations.
2. `MaxAndSkipEnv`: Applies `max` to N observations (four by default) and returns this as an observation for the step. This solves the “flickering” problem in some Atari games, when the game draws different portions of the screen on even and odd frames (a normal practice among Atari developers to overcome the platform’s limitations and increase the complexity of the game’s sprites).
3. `EpisodicLifeEnv`: In some games, this detects a lost life and turns this situation into the end of the episode. This significantly increases convergence, as our episodes become shorter (one single life versus several given by the game logic). This is relevant only for some games supported by the Atari 2600 Learning Environment.
4. `FireResetEnv`: Executes a **FIRE** action on game reset. Some games require this to start the gameplay. Without this, our environment becomes a **partially observable Markov decision process (POMDP)**, which makes it impossible to converge.
5. `WarpFrame`: Also known as `ProcessFrame84`, this converts an image to grayscale and resizes it to 84×84 .
6. `ClipRewardEnv`: Clips the reward to a $-1 \dots 1$ range, which unifies wide variability in scoring among different Atari games. For example, Pong might have a $-21 \dots 21$ score range, but the score in the River Raid game could be $0 \dots \infty$.
7. `FrameStack`: Stacks N sequential observations into the stack (the default is four). As we already discussed in *Chapter 6*, in some games, this is required to fulfill the Markov property. For example, in Pong, from one single frame, it is impossible to get the direction the ball is moving in.

The code of those wrappers was heavily optimized by many people and several versions exist. My personal favorite is the *Stable Baselines3*, which is a fork from the OpenAI *Baselines* project. You can find it here: <https://stable-baselines3.readthedocs.io/>.

But you shouldn’t take this code as the final source of truth, as your concrete environment might have different requirements and specifics. For example, if you are interested in speeding up one specific game from the Atari suite, `NoopResetEnv` and `MaxAndSkipEnv` (more precisely, the max pooling operation from `MaxAndSkipEnv`) might not be needed. Another thing that could be tweaked is the number of frames in the `FrameStack` wrapper. The normal practice is to use four, but you need to understand that this number was used by DeepMind and other researchers to train on the full Atari 2600 game suite, which currently includes more than 50 games. For your specific case, a history of two frames might be enough to give you a performance boost, as less data will need to be processed by the NN.

Finally, the image resize could be the bottleneck of wrappers, so you might want to optimize libraries used by wrappers, for example, rebuilding them or replacing them with faster versions.

Prior to 2020, replacing the OpenCV2 library with the `pillow-simd` library gave a boost of about 50 frames per second. Nowadays, OpenCV2 uses an optimized rescaling operation, so such replacement has no effect. But still, you might experiment with different scaling methods and different libraries.

Here, we'll apply the following changes to the Pong wrappers:

- Disable `NoopResetEnv`
- Replace `MaxAndSkipEnv` with a simplified version, which just skips four frames without max pooling
- Keep only two frames in `FrameStack`

To check the combined effect of our tweaks, we'll add the above changes to the modifications done in the previous two sections: several environments and parallel execution of playing and training.

As the changes are not complex, let's just quickly discuss them without the actual code (the full code can be found in the files `Chapter09/04_wrappers_n_env.py`, `Chapter09/04_wrappers_parallel.py`, and `Chapter09/lib/atari_wrappers.py`):

- Library `atari_wrappers.py` is quite simple — it contains the copy of the `wrap_dqn` function from PTAN and the `AtariWrapper` class from Stable Baselines3.
- In `AtariWrapper`, the class `MaxAndSkipEnv` was replaced with a simplified version without max pooling between frames.
- Two modules, `04_wrappers_n_env.py` and `04_wrappers_parallel.py`, are just copies of `02_n_env.py` and `03_parallel.py` we've already seen, with tweaked environment creation.

That's it! The following are charts with reward dynamics and FPS for both versions:

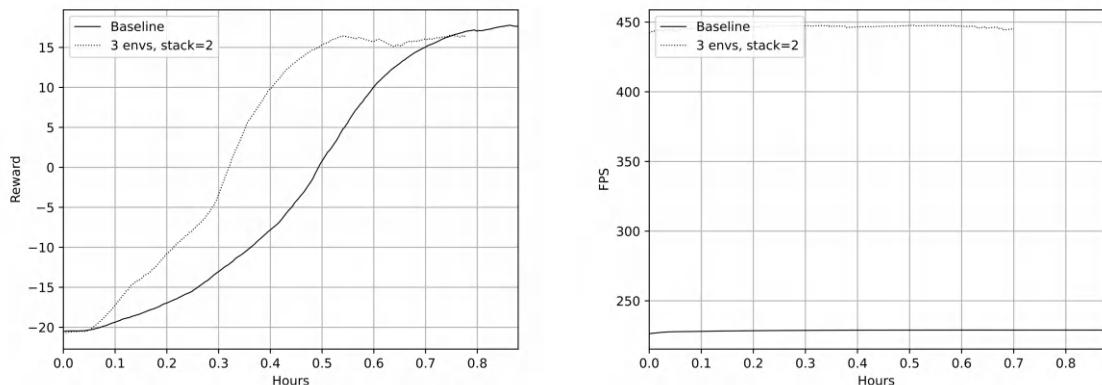


Figure 9.9: Reward and FPS in the baseline and “3 environments and 2 frames” version

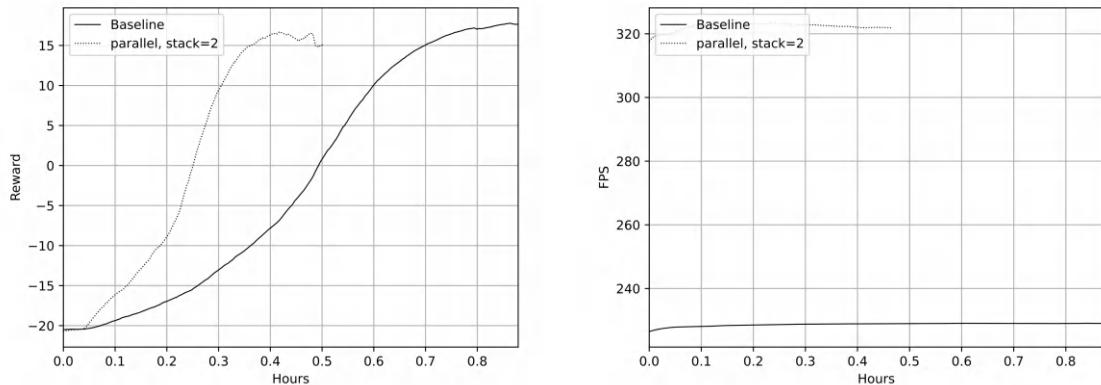


Figure 9.10: Reward and FPS in the baseline and “parallel and 2 frames” version

Out of curiosity, I also tried to reduce the number of frames kept in `FrameStack` to just one frame (you can repeat the experiment with the command-line argument `-stack 1`). Surprisingly, such a version was able to solve the game, but it took significantly longer in terms of games needed and the training became unstable (about 3 out of 8 training runs didn’t converge at all). This might be an indication that Pong with just one frame is not POMDP and the agent still can learn how to win the game having just one frame as observation. But the efficiency of training definitely suffers.

Benchmark results

I’ve summarized our experiments in the following table. The percentages show the changes versus the baseline version:

Step	FPS	FPS Δ	Time, mins	Time Δ
Baseline	229		52.2	
Without <code>torch.no_grad()</code>	219	-4.3%	51.0	-2.3%
3 environments	395	+72.5%	36.0	-31.0%
Parallel version	290	+26.6%	31.2	-40.2%
Wrappers + 3 environments	448	+95.6%	47.4	-9.2%
Wrappers + parallel	325	+41.9%	30.0	-42.5%

Table 9.1: Optimization results

Summary

In this chapter, you saw several ways to improve the performance of the RL method using a pure engineering approach, which was in contrast to the “algorithmic” or “theoretical” approach covered in *Chapter 8*. From my perspective, both approaches complement each other, and a good RL practitioner needs to both know the latest tricks that researchers have found and be aware of the implementation details.

In the next chapter, we will begin applying our DQN knowledge to stocks trading as a practical example.

10

Stocks Trading Using RL

Rather than learning new methods to solve toy **reinforcement learning (RL)** problems in this chapter, we will try to utilize our **deep Q-network (DQN)** knowledge to deal with the much more practical problem of financial trading. I can't promise that the code will make you super rich on the stock market or Forex, because my goal is much less ambitious: to demonstrate how to go beyond the Atari games and apply RL to a different practical domain.

In this chapter, we will:

- Implement our own OpenAI Gym environment to simulate the stock market
- Apply the DQN method that you learned in *Chapter 6* and *Chapter 8* to train an agent to trade stocks to maximize profit

Why trading?

There are a lot of financial instruments traded on markets every day: goods, stocks, and currencies. Even weather forecasts can be bought or sold using so-called “weather derivatives,” which is just a consequence of the complexity of the modern world and financial markets. If your income depends on future weather conditions, as it does for a business growing crops, then you might want to hedge the risks by buying weather derivatives. All these different items have a price that changes over time.

Trading is the activity of buying and selling financial instruments with different goals, like making a profit (investment), gaining protection from future price movement (hedging), or just getting what you need (like buying steel or exchanging USD for JPY to pay a contract).

Since the first financial market was established, people have been trying to predict future price movements, as this promises many benefits, like “profit from nowhere” or protecting capital from sudden market movements. This problem is known to be complex, and there are a lot of financial consultants, investment funds, banks, and individual traders trying to predict the market and find the best moments to buy and sell to maximize profit.

The question is: can we look at the problem from the RL angle? Let’s say that we have some observation of the market, and we want to make a decision: buy, sell, or wait. If we buy before the price goes up, our profit will be positive; otherwise, we will get a negative reward. What we’re trying to do is get as much profit as possible. The connections between market trading and RL are quite obvious. First, let’s define the problem statement more clearly.

Problem statement and key decisions

The finance domain is large and complex, so you can easily spend several years learning something new every day. In our example, we will just scratch the surface a bit with our RL tools, and our problem will be formulated as simply as possible, using price as an observation. We will investigate whether it will be possible for our agent to learn when the best time is to buy one single share and then close the position to maximize the profit. The purpose of this example is to show how flexible the RL model can be and what the first steps are that you usually need to take to apply RL to a real-life use case.

As you already know, to formulate RL problems, three things are needed: observation of the environment, possible actions, and a reward system. In previous chapters, all three were already given to us, and the internal machinery of the environment was hidden. Now we’re in a different situation, so we need to decide ourselves what our agent will see and what set of actions it can take. The reward system is also not given as a strict set of rules; rather, it will be guided by our feelings and knowledge of the domain, which gives us lots of flexibility.

Flexibility, in this case, is good and bad at the same time. It’s good that we have the freedom to pass some information to the agent that we feel will be important to learn efficiently. For example, you can provide to the trading agent not only prices, but also news or important statistics (which are known to influence financial markets a lot). The bad part is that this flexibility usually means that to find a good agent, you need to try a lot of variants of data representation, and it’s not always obvious which will work better.

In our case, we will implement the basic trading agent in its simplest form, as we discussed in *Chapter 1*:

- Observation: The observation will include the following information:
 - N past bars, where each has open, high, low, and close prices
 - An indication that the share was bought some time ago (only one share at a time will be possible)
 - The profit or loss that we currently have from our current position (the share bought)
- Action: At every step, after every minute's bar, the agent can take one of the following actions:
 - **Do nothing:** Skip the bar without taking an action
 - **Buy a share:** If the agent has already got the share, nothing will be bought; otherwise, we will pay the commission, which is usually some small percentage of the current price
 - **Close the position:** if we do not have a previously purchased share, nothing will happen; otherwise, we will pay the commission for the trade
- Reward: The reward that the agent receives can be expressed in various ways:
 - As the first option, we can split the reward into multiple steps during our ownership of the share. In that case, the reward on every step will be equal to the last bar's movement.
 - Alternatively, the agent can receive the reward only after the *close* action and get the full reward at once.

At first sight, both variants should have the same final result, but maybe with different convergence speeds. However, in practice, the difference could be dramatic. The environment in my implementation supports both variants, so you can experiment with the difference.

One last decision to make is how to represent the prices in our environment observation. Ideally, we would like our agent to be independent of actual price values and take into account relative movement, such as “the stock has grown 1% during the last bar” or “the stock has lost 5%.” This makes sense, as different stocks’ prices can vary, but they can have similar movement patterns. In finance, there is a branch of analytics called *technical analysis* that studies such patterns to help to make predictions from them. We would like our system to be able to discover the patterns (if they exist). To achieve this, we will convert every bar’s open, high, low, and close prices to three numbers showing high, low, and close prices represented as a percentage of the open price.

This representation has its own drawbacks, as we’re potentially losing the information about key price levels. For example, it’s known that markets have a tendency to bounce from round price numbers (like \$70,000 per bitcoin) and levels that were turning points in the past. However, as already stated, we’re just playing with the data here and checking the concept. Representation in the form of relative price movement will help the system to find repeating patterns in the price level (if they exist, of course), regardless of the absolute price position. Potentially, the neural network (NN) could learn this on its own (it’s just the mean price that needs to be subtracted from the absolute price values), but relative representation simplifies the NN’s task.

Data

In our example, we will use the Russian stock market prices from the period of 2015-2016, which are placed in Chapter10/data/ch10-small-quotes.tgz and have to be unpacked before model training.

Inside the archive, we have CSV files with M1 bars, which means that every row in each CSV file corresponds to a single minute in time, and price movement during that minute is captured with four prices:

- **Open:** The price at the beginning of the minute
- **High:** The maximum price during the interval
- **Low:** The minimum price
- **Close:** The last price of the minute time interval

Every minute interval is called a bar and allows us to have an idea of price movement within the interval. For example, in the YNDX_160101_161231.csv file (which has Yandex company stocks for 2016), we have 130k lines in this form:

```
<DATE>,<TIME>,<OPEN>,<HIGH>,<LOW>,<CLOSE>,<VOL>
20160104,100100,1148.90000,1148.90000,1148.90000,1148.90000,0
20160104,100200,1148.90000,1148.90000,1148.90000,1148.90000,50
20160104,100300,1149.00000,1149.00000,1149.00000,1149.00000,33
20160104,100400,1149.00000,1149.00000,1149.00000,1149.00000,4
20160104,100500,1153.00000,1153.00000,1153.00000,1153.00000,0
20160104,100600,1156.90000,1157.90000,1153.00000,1153.00000,43
20160104,100700,1150.60000,1150.60000,1150.40000,1150.40000,5
20160104,100800,1150.20000,1150.20000,1150.20000,1150.20000,4
20160104,100900,1150.50000,1150.50000,1150.50000,1150.50000,2
20160104,101000,1150.00000,1150.00000,1150.00000,1150.00000,43
20160104,101100,1149.70000,1149.70000,1149.70000,1149.70000,0
20160104,101200,1150.20000,1150.20000,1149.50000,1149.70000,165
...
```

The first two columns are the date and time for the minute; the next four columns are open, high, low, and close prices; and the last value represents the number of buy and sell orders performed during the bar (also called **volume**). The exact interpretation of volume is market-dependent, but usually, it give you an idea about how active the market was.

The typical way to represent those prices is called a **candlestick chart**, where every bar is shown as a candle. Part of Yandex's quotes for one day in February 2016 is shown in the following chart:



Figure 10.1: Price data for Yandex in February 2016

The archive contains two files with M1 data for 2016 and 2015. We will use data from 2016 for model training and data from 2015 for validation (but the order is arbitrary; you can swap them or even use different time intervals and check the effect).

The trading environment

As we have a lot of code that is supposed to work with the Gym API, we will implement the trading functionality following Gym's Env class, which should be already familiar to you. Our environment is implemented in the StocksEnv class in the Chapter10/lib/environ.py module. It uses several internal classes to keep its state and encode observations.

Let's first look at the public API class:

```
import typing as tt
import gymnasium as gym
from gymnasium import spaces
from gymnasium.utils import seeding
from gymnasium.envs.registration import EnvSpec
import enum
import numpy as np
from . import data

DEFAULT_BARS_COUNT = 10
DEFAULT_COMMISSION_PERC = 0.1
```

```
class Actions(enum.Enum):
    Skip = 0
    Buy = 1
    Close = 2
```

We encode all available actions as an enumerator's fields and provide just three actions: do nothing, buy a single share, and close the existing position.



In our market model, we allow only the single share to be bought, neither supporting extending existing positions nor opening “short positions” (when you selling the share you don't have, expecting the price to decrease in the future). That was an intentional decision, as I tried to keep the example simple and to avoid overcomplications. Why don't you try experimenting with other options?

Next, we have the environment class:

```
class StocksEnv(gym.Env):
    spec = EnvSpec("StocksEnv-v0")
```

The field `spec` is required for `gym.Env` compatibility and registers our environment in the Gym internal registry.

This class provides two ways to create its instance:

```
@classmethod
def from_dir(cls, data_dir: str, **kwargs):
    prices = {
        file: data.load_relative(file)
        for file in data.price_files(data_dir)
    }
    return StocksEnv(prices, **kwargs)
```

As you can see in the preceding code, the first way is to call the class method `from_dir` with the `data` directory as the argument. In that case, it will load all quotes from the CSV files in the directory and construct the environment. To deal with price data in our form, we have several helper functions in `Chapter10/lib/data.py`. Another way is to construct the class instance directly. In that case, you should pass the `prices` dictionary, which has to map the quote name to the `Prices` dataclass declared in `data.py`. This object has five fields containing `open`, `high`, `low`, `close`, and `volume` time series as one-dimensional NumPy arrays.

The module `data.py` also provides several helping functions, like converting the prices into relative format, enumerating files in the given directory, etc.

The following is the constructor of the environment:

```
def __init__(  
    self, prices: tt.Dict[str, data.Prices],  
    bars_count: int = DEFAULT_BARS_COUNT,  
    commission: float = DEFAULT_COMMISSION_PERC,  
    reset_on_close: bool = True, state_1d: bool = False,  
    random_ofs_on_reset: bool = True,  
    reward_on_close: bool = False, volumes=False  
):
```

It accepts a lot of arguments to tweak the environment's behavior and observation representation:

- `prices`: Contains one or more stock prices for one or more instruments as a dict, where keys are the instrument's name and the value is a container object `data.Prices`, which holds price data arrays.
- `bars_count`: The count of bars that we pass in the observation. By default, this is 10 bars.
- `commission`: The percentage of the stock price that we have to pay to the broker on buying and selling the stock. By default, it's 0.1%.
- `reset_on_close`: If this parameter is set to `True`, which it is by default, every time the agent asks us to close the existing position (in other words, sell a share), we stop the episode. Otherwise, the episode will continue until the end of our time series, which is one year of data.
- `conv_1d`: This Boolean argument switches between different representations of price data in the observation passed to the agent. If it is set to `True`, observations have a 2D shape, with different price components for subsequent bars organized in rows. For example, high prices (max price for the bar) are placed on the first row, low prices on the second, and close prices on the third. This representation is suitable for doing 1D convolution on time series, where every row in the data has the same meaning as different color planes (red, green, or blue) in Atari 2D images. If we set this option to `False`, we have one single array of data with every bar's components placed together. This organization is convenient for a fully connected network architecture. Both representations are illustrated in *Figure 10.2*.
- `random_ofs_on_reset`: If the parameter is `True` (by default), on every reset of the environment, the random offset in the time series will be chosen. Otherwise, we will start from the beginning of the data.
- `reward_on_close`: This Boolean parameter switches between the two reward schemes discussed previously. If it is set to `True`, the agent will receive a reward only on the "close" action issue. Otherwise, we will give a small reward every bar, corresponding to price movement during that bar.

- `volumes`: This argument switches on volumes in observations and is disabled by default.

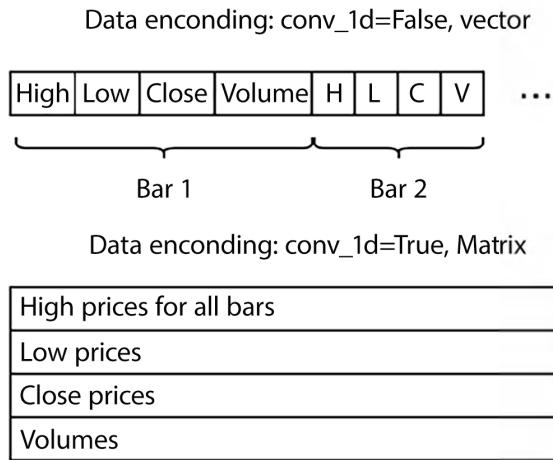


Figure 10.2: Different data representations for the NN

Now we will continue looking at the environment constructor:

```

self._prices = prices
if state_1d:
    self._state = State1D(bars_count, commission, reset_on_close,
                          reward_on_close=reward_on_close, volumes=volumes)
else:
    self._state = State(bars_count, commission, reset_on_close,
                        reward_on_close=reward_on_close, volumes=volumes)
self.action_space = spaces.Discrete(n=len(Actions))
self.observation_space = spaces.Box(
    low=-np.inf, high=np.inf, shape=self._state.shape, dtype=np.float32)
self.random_ofs_on_reset = random_ofs_on_reset

```

Most of the functionality of the `StocksEnv` class is implemented in two internal classes: `State` and `State1D`. They are responsible for observation preparation and our bought share state and reward. They implement a different representation of our data in the observations, and we will take a look at their code later. In the constructor, we create the state object, action space, and observation space fields that are required by Gym.

This method defines the `reset()` functionality for our environment:

```

def reset(self, *, seed: int | None = None, options: dict[str, tt.Any] | None =
          None):
    # make selection of the instrument and it's offset. Then reset the state

```

```
super().reset(seed=seed, options=options)
self._instrument = self.np_random.choice(list(self._prices.keys()))
prices = self._prices[self._instrument]
bars = self._state.bars_count
if self.random_ofs_on_reset:
    offset = self.np_random.choice(prices.high.shape[0]-bars*10) + bars
else:
    offset = bars
self._state.reset(prices, offset)
return self._state.encode(), {}
```

According to the `gym.Env` semantics, we randomly switch the time series that we will work on and select the starting offset in this time series. The selected price and offset are passed to our internal state instance, which then asks for an initial observation using its `encode()` function.

This method has to handle the action chosen by the agent and return the next observation, reward, and done flag:

```
def step(self, action_idx: int) -> tt.Tuple[np.ndarray, float, bool, bool, dict]:
    action = Actions(action_idx)
    reward, done = self._state.step(action)
    obs = self._state.encode()
    info = {
        "instrument": self._instrument,
        "offset": self._state._offset
    }
    return obs, reward, done, False, info
```

All real functionality is implemented in our state classes, so this method is a very simple wrapper around the call to state methods.



The API for `gym.Env` allows you to define the `render()` method handler, which is supposed to render the current state in human or machine-readable format. Generally, this method is used to peek inside the environment state and is useful for debugging or tracing the agent's behavior. For example, the market environment could render current prices as a chart to visualize what the agent sees at that moment. Our environment doesn't support rendering (as this functionality is optional), so we don't define this function at all.

Let's now look at the internal `environ.State` class, which implements the core of the environment's functionality:

```
class State:
    def __init__(self, bars_count: int, commission_perc: float, reset_on_close: bool,
                 reward_on_close: bool = True, volumes: bool = True):
        assert bars_count > 0
        assert commission_perc >= 0.0
        self.bars_count = bars_count
        self.commission_perc = commission_perc
        self.reset_on_close = reset_on_close
        self.reward_on_close = reward_on_close
        self.volumes = volumes
        self.have_position = False
        self.open_price = 0.0
        self._prices = None
        self._offset = None
```

The constructor does nothing more than just check and remember the arguments in the object's fields.

The `reset()` method is called every time that the environment is asked to reset and has to save the passed prices data and starting offset:

```
def reset(self, prices: data.Prices, offset: int):
    assert offset >= self.bars_count - 1
    self.have_position = False
    self.open_price = 0.0
    self._prices = prices
    self._offset = offset
```

In the beginning, we don't have any shares bought, so our state has `have_position=False` and `open_price=0.0`.

The `shape` property returns the tuple with dimensions of the NumPy array with encoded state:

```
@property
def shape(self) -> tt.Tuple[int, ...]:
    # [h, l, c] * bars + position_flag + rel_profit
    if self.volumes:
        return 4 * self.bars_count + 1 + 1,
    else:
        return 3 * self.bars_count + 1 + 1,
```

The `State` class is encoded into a single vector (top part in the *Figure 10.2*), which includes prices with optional volumes and two numbers indicating the presence of a bought share and position profit.

The `encode()` method packs prices at the current offset into a NumPy array, which will be the observation of the agent:

```
def encode(self) -> np.ndarray:
    res = np.ndarray(shape=self.shape, dtype=np.float32)
    shift = 0
    for bar_idx in range(-self.bars_count+1, 1):
        ofs = self._offset + bar_idx
        res[shift] = self._prices.high[ofs]
        shift += 1
        res[shift] = self._prices.low[ofs]
        shift += 1
        res[shift] = self._prices.close[ofs]
        shift += 1
    if self.volumes:
        res[shift] = self._prices.volume[ofs]
        shift += 1
    res[shift] = float(self.have_position)
    shift += 1
    if not self.have_position:
        res[shift] = 0.0
    else:
        res[shift] = self._cur_close() / self.open_price - 1.0
    return res
```

This helper method calculates the current bar's close price:

```
def _cur_close(self) -> float:
    open = self._prices.open[self._offset]
    rel_close = self._prices.close[self._offset]
    return open * (1.0 + rel_close)
```

Prices passed to the `State` class have the relative form with respect to the open price: the high, low, and close components are relative ratios to the open price. This representation was already discussed when we talked about the training data, and it will (probably) help our agent to learn price patterns that are independent of actual price value.

The `step()` method is the most complicated piece of code in the `State` class:

```
def step(self, action: Actions) -> tt.Tuple[float, bool]:
    reward = 0.0
    done = False
    close = self._cur_close()
```

It is responsible for performing one step in our environment. On exit, it has to return the reward in a percentage and an indication of the episode ending.

If the agent has decided to buy a share, we change our state and pay the commission:

```
if action == Actions.Buy and not self.have_position:
    self.have_position = True
    self.open_price = close
    reward -= self.commission_perc
```

In our state, we assume the instant order execution at the current bar's close price, which is a simplification on our side; normally, an order can be executed on a different price, which is called *price slippage*.

If we have a position and the agent asks us to close it, we pay commission again, change the done flag if we're in `reset_on_close` mode, give a final reward for the whole position, and change our state:

```
elif action == Actions.Close and self.have_position:
    reward -= self.commission_perc
    done |= self.reset_on_close
    if self.reward_on_close:
        reward += 100.0 * (close / self.open_price - 1.0)
    self.have_position = False
    self.open_price = 0.0
```

In the rest of the function, we modify the current offset and give the reward for the last bar movement:

```
self._offset += 1
prev_close = close
close = self._cur_close()
done |= self._offset >= self._prices.close.shape[0]-1

if self.have_position and not self.reward_on_close:
    reward += 100.0 * (close / prev_close - 1.0)

return reward, done
```

That's it for the State class, so let's look at State1D, which has the same behavior and just overrides the representation of the state passed to the agent:

```

class State1D(State):
    @property
    def shape(self) -> tt.Tuple[int, ...]:
        if self.volumes:
            return 6, self.bars_count
        else:
            return 5, self.bars_count

```

The shape of this representation is different, as our prices are encoded as a 2D matrix suitable for a 1D convolution operator.

This method encodes the prices in our matrix, depending on the current offset, whether we need volumes, and whether we have stock:

```

def encode(self) -> np.ndarray:
    res = np.zeros(shape=self.shape, dtype=np.float32)
    start = self._offset-(self.bars_count-1)
    stop = self._offset+1
    res[0] = self._prices.high[start:stop]
    res[1] = self._prices.low[start:stop]
    res[2] = self._prices.close[start:stop]
    if self.volumes:
        res[3] = self._prices.volume[start:stop]
        dst = 4
    else:
        dst = 3
    if self.have_position:
        res[dst] = 1.0
        res[dst+1] = self._cur_close() / self.open_price - 1.0
    return res

```

That's it for our trading environment. Compatibility with the Gym API allows us to plug it into the familiar classes that we used to handle the Atari games. Let's do that now.

Models

In this example, two architectures of DQN are used: a simple feed-forward network with three layers and a network with 1D convolution as a feature extractor, followed by two fully connected layers to output Q-values. Both of them use the dueling architecture described in *Chapter 8*. Double DQN and two-step Bellman unrolling have also been used. The rest of the process is the same as in a classical DQN (from *Chapter 6*).

Both models are in Chapter10/lib/models.py and are very simple. Let's start with the feed-forward model:

```
class SimpleFFDQN(nn.Module):
    def __init__(self, obs_len: int, actions_n: int):
        super(SimpleFFDQN, self).__init__()

        self.fc_val = nn.Sequential(
            nn.Linear(obs_len, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 1)
        )

        self.fc_adv = nn.Sequential(
            nn.Linear(obs_len, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, actions_n)
        )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        val = self.fc_val(x)
        adv = self.fc_adv(x)
        return val + (adv - adv.mean(dim=1, keepdim=True))
```

The feed forward model uses independent networks for Q-value and advantage prediction.

The convolutional model has a common feature extraction layer with the 1D convolution operations and two fully connected heads to output the value of the state and advantages for actions:

```
class DQNConv1D(nn.Module):
    def __init__(self, shape: tt.Tuple[int, ...], actions_n: int):
        super(DQNConv1D, self).__init__()

        self.conv = nn.Sequential(
            nn.Conv1d(shape[0], 128, 5),
            nn.ReLU(),
            nn.Conv1d(128, 128, 5),
            nn.ReLU(),
            nn.Flatten(),
        )
        size = self.conv(torch.zeros(1, *shape)).size()[-1]

        self.fc_val = nn.Sequential(
```

```
        nn.Linear(size, 512),
        nn.ReLU(),
        nn.Linear(512, 1)
    )

    self.fc_adv = nn.Sequential(
        nn.Linear(size, 512),
        nn.ReLU(),
        nn.Linear(512, actions_n)
)

def forward(self, x: torch.Tensor) -> torch.Tensor:
    conv_out = self.conv(x)
    val = self.fc_val(conv_out)
    adv = self.fc_adv(conv_out)
    return val + (adv - adv.mean(dim=1, keepdim=True))
```

As you can see, the model is very similar to the DQN Dueling architecture we used in Atari examples.

Training code

We have two very similar training modules in this example: one for the feed-forward model and one for 1D convolutions. For both of them, there is nothing new added to our examples from *Chapter 8*:

- They're using epsilon-greedy action selection to perform exploration. The epsilon linearly decays over the first 1M steps from 1.0 to 0.1.
- A simple experience replay buffer of size 100k is being used, which is initially populated with 10k transitions.
- For every 1,000 steps, we calculate the mean value for the fixed set of states to check the dynamics of the Q-values during the training.
- For every 100k steps, we perform validation: 100 episodes are played on the training data and on previously unseen quotes. Validation results are recorded in TensorBoard, such as the mean profit, the mean count of bars, and the share held. This step allows us to check for overfitting conditions.

The training modules are in `Chapter10/train_model.py` (feed-forward model) and `Chapter10/train_model_conv.py` (with a 1D convolutional layer). Both versions accept the same command-line options.

To start the training, you need to pass training data with the `-data` option, which could be an individual CSV file or the whole directory with files. By default, the training module uses Yandex quotes for 2016 (file `data/YNDX_160101_161231.csv`). For the validation data, there is an option, `-val`, that takes Yandex 2015 quotes by default. Another required option will be `-r`, which is used to pass the name of the run.

This name will be used in the TensorBoard run name and to create directories with saved models.

Results

Now that we've implemented them, let's compare the performance of our two models, starting with feed-forward variant.

The feed-forward model

During the training, the average reward obtained by the agent was slowly but consistently growing. After 300k episodes, the growth slowed down. The following are charts (*Figure 10.3*) showing the raw reward during the training and the same data smoothed with the simple moving average of the last 15 values:

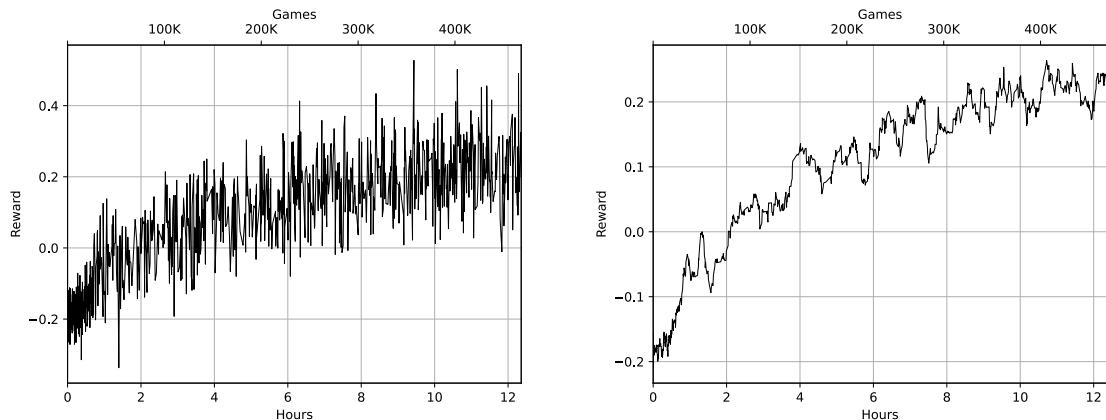


Figure 10.3: Reward during the training. Raw values (left) and smoothed (right)

Another pair of charts (Figure 10.4) shows the reward obtained from testing performed on the same training data but without random actions ($\epsilon = 0$):

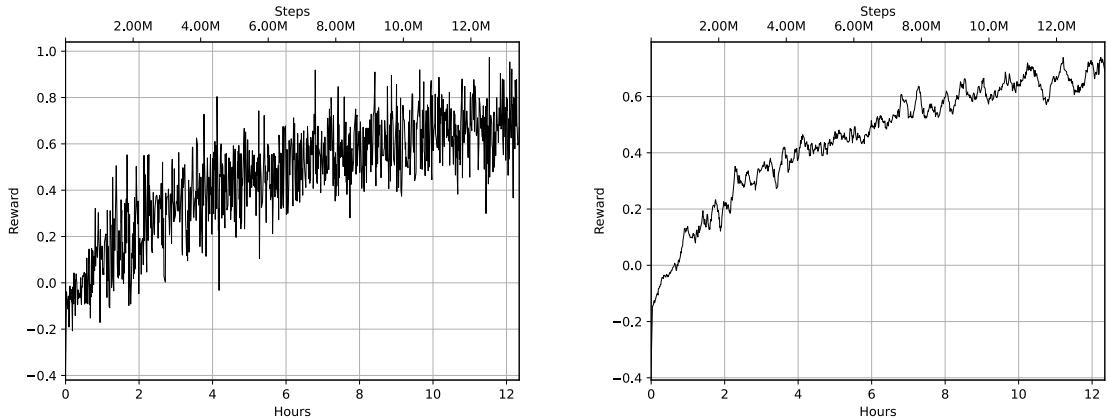


Figure 10.4: Reward from the tests. Raw values (left) and smoothed (right)

Both the training and testing reward charts show that the agent is learning how to increase the profit over time.

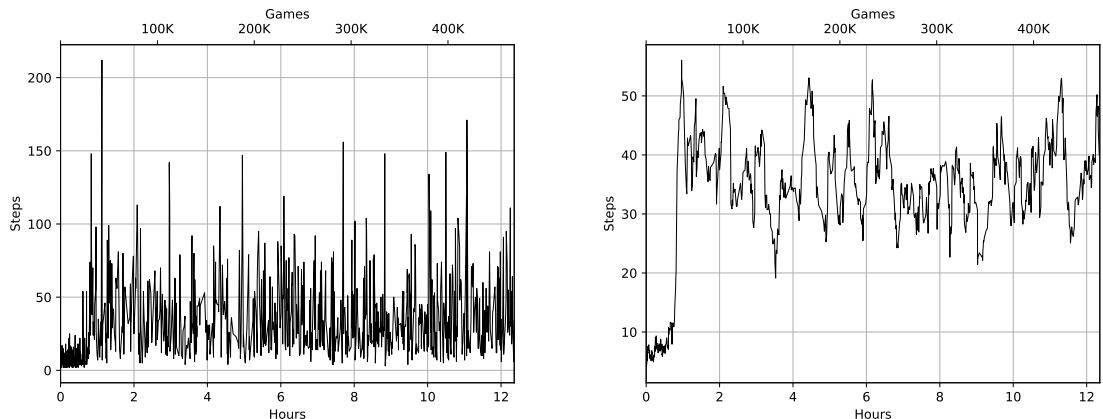


Figure 10.5: Length of the episodes. Raw values (left) and smoothed (right)

The length of each episode also increased after 100k episodes, as the agent learned that holding the share might be profitable.

In addition, we monitor the predicted value of the random set of states.

The following chart shows that the network becomes more and more optimistic about those states during the training:

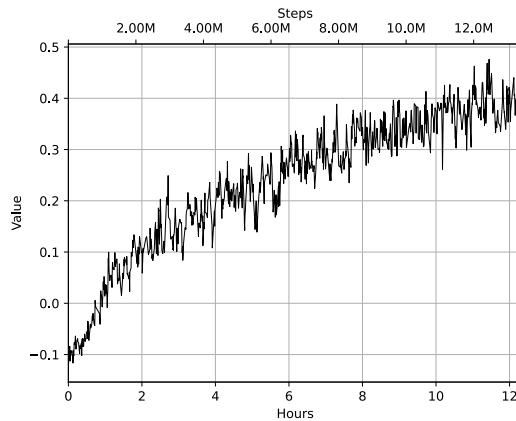


Figure 10.6: Value predicted for a random set of states

All charts look good so far, but all of them were obtained using the training data. It is great that our agent is learning how to get profits on the historical data. But will it work on data never seen before? To check that, we perform validation on 2,015 quotes, and the reward is shown in the *Figure 10.7*:

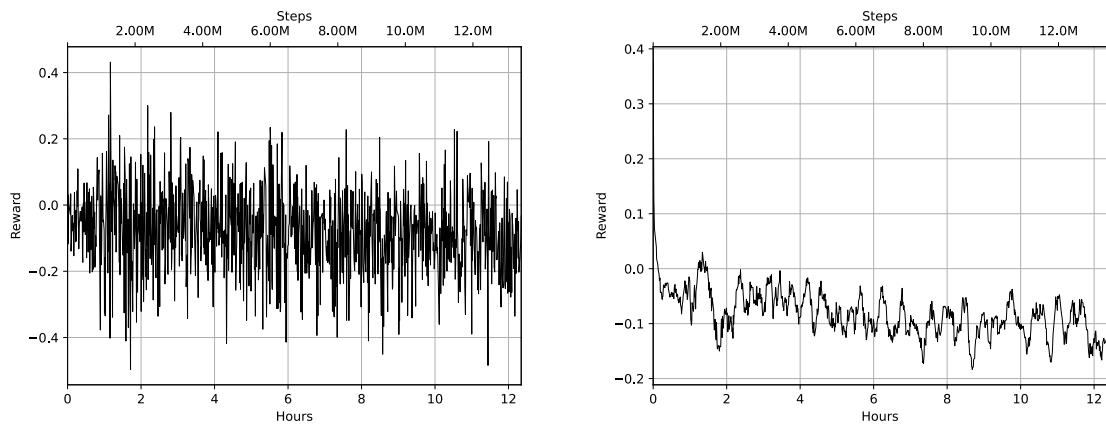


Figure 10.7: Reward on validation dataset. Raw values (left) and smoothed (right)

This chart is a bit disappointing: the reward doesn't have an uptrend. In the smoothed version of the chart, we might even see the opposite — the reward is slowly decreasing after the first hour of training (at that point, we had a significant increase in training episode length on *Figure 10.5*).

This might be an indication of overfitting of the agent, which starts after 1M training iterations. But still, for the first 4 hours of training, the reward is above -0.2% (which is a broker commission in our environment – 0.1% when we buy stock and 0.1% for selling it) and means that our agent is better than a random “buying-and-selling monkey.”

During the training, our code saves models for later experiments. It does this every time the mean Q-values on our held-out-states set update the maximum or when the reward on the validation sets beats the previous record. There is a tool that loads the model, trades on prices you’ve provided to it with the command-line option, and draws the plots with the profit change over time. The tool is called `Chapter10/run_model.py` and it can be used like this:

```
Chapter10$ ./run_model.py -d data/YNDX_160101_161231.csv -m
saves/simple-t1/mean_value-0.277.data -b 10 -n YNDX16
```

The options that the tool accepts are as follows:

- **-d:** This is the path to the quotes to use. In the shown command, we apply the model to the data that it was trained on.
- **-m:** This is the path to the model file. By default, the training code saves it in the `saves` directory.
- **-b:** This shows how many bars to pass to the model in the context. It has to match the count of bars used on training, which is 10 by default and can be changed in the training code.
- **-n:** This is the suffix to be appended to the images produced.
- **-commission:** This allows you to redefine the broker’s commission, which has a default of 0.1%.

At the end, the tool creates a chart of the total profit dynamics (in percentages). The following is the reward chart on Yandex 2016 quotes (used for training):

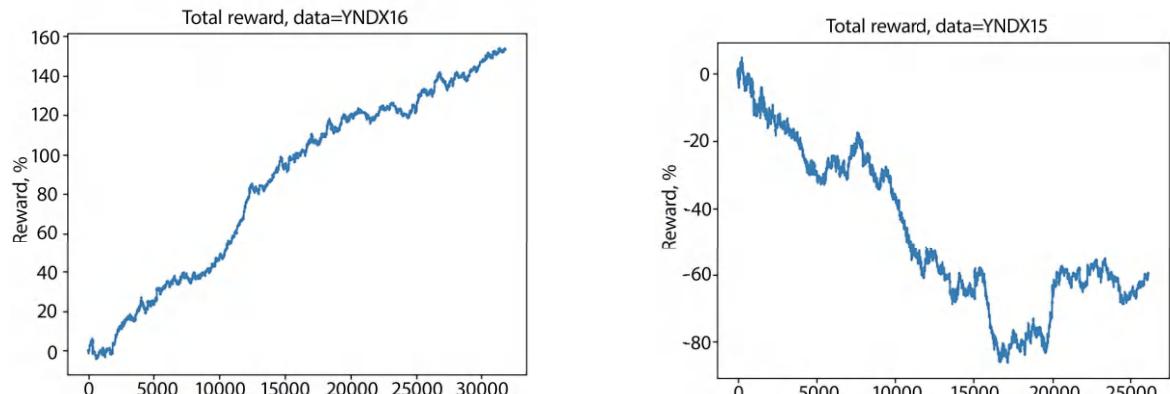


Figure 10.8: Trading profit on the training data (left) and validation (right)

The result on the training data looks amazing: 150% profit in just a year. However, the result on the validation dataset is much worse, as we've seen from the validation plots in TensorBoard.

To check that our system is profitable with zero commission, we can rerun on the same data with the `-commission 0.0` option:

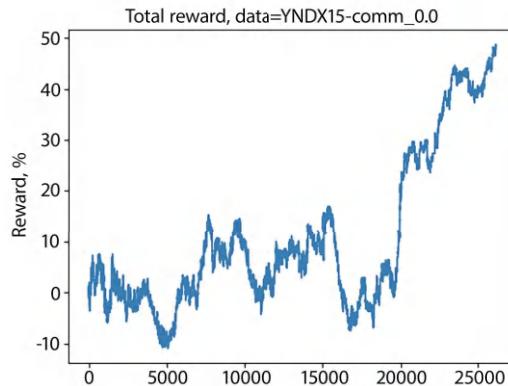


Figure 10.9: Trading profit on validation data without broker's commission

We have some bad days with drawdown, but the overall results are good: without commission, our agent can be profitable. Of course, the commission is not the only issue. Our order simulation is very primitive and doesn't take into account real-life situations, such as price spread and a slip in order execution.

If we take the model with the best reward on the validation set, the reward dynamics are a bit better. Profitability is lower, but the drawdown on unseen quotes is much lower (and commission was enabled for the following charts):

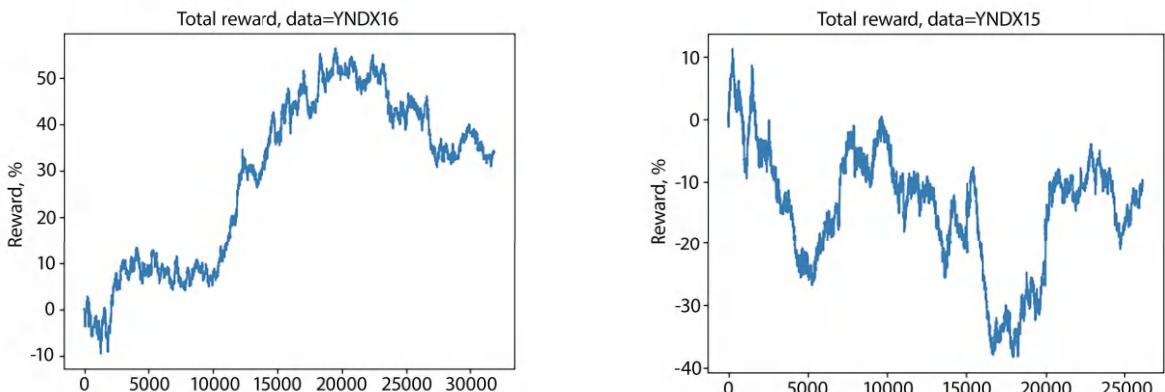


Figure 10.10: The reward from the model with the best validation reward. Training data (left) and validation (right)

But, of course, taking the best model based on *validation data* is cheating – by using validation results for model's selection, we are ruining the idea of validation. So, the charts above are just to illustrate that there are *some models* that might work alright even on unseen data.

The convolution model

The second model implemented in this example uses 1D convolution filters to extract features from the price data. This allows us to increase the number of bars in the context window that our agent sees on every step without a significant increase in the network size. By default, the convolution model example uses 50 bars of context. The training code is in `Chapter10/train_model_conv.py`, and it accepts the same set of command-line parameters as the feed-forward version.

Training dynamics are almost identical, but the reward obtained on the validation set is slightly higher and starts to overfit later:

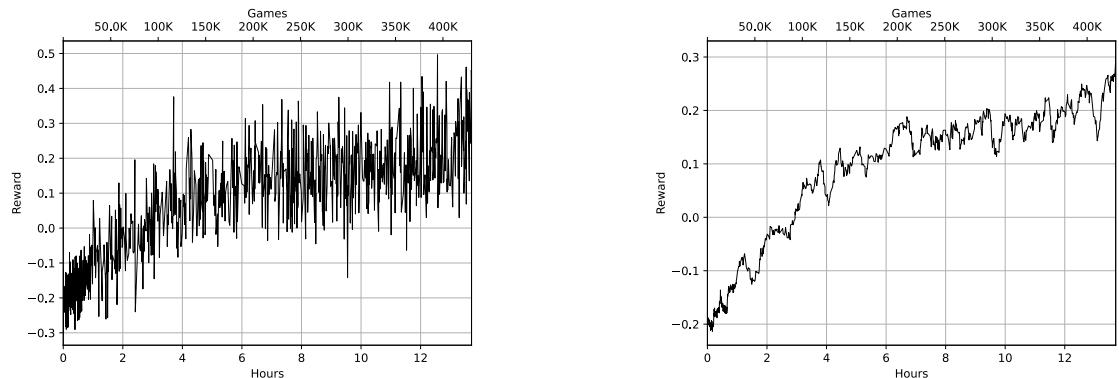


Figure 10.11: Reward during the training. Raw values (left) and smoothed (right)

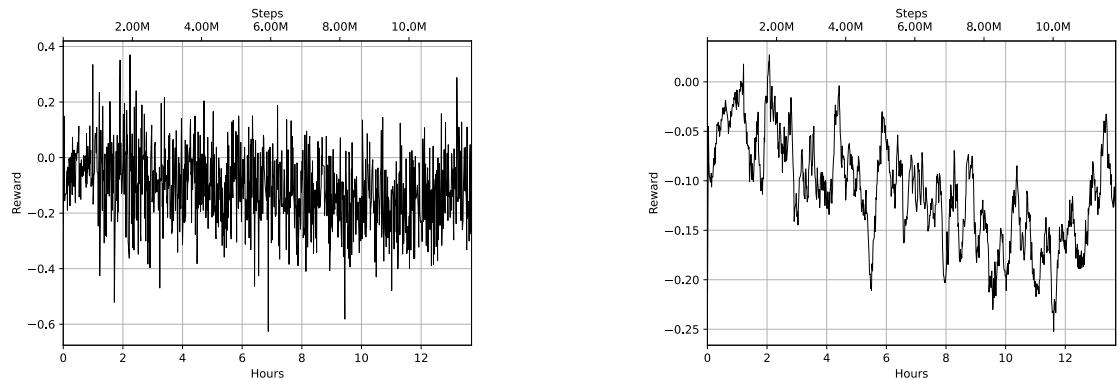


Figure 10.12: Reward on validation dataset. Raw values (left) and smoothed (right)

Things to try

As already mentioned, financial markets are large and complicated. The methods that we've tried are just the very beginning. Using RL to create a complete and profitable trading strategy is a large project, which can take several months of dedicated labor. However, there are things that we can try to get a better understanding of the topic:

- Our data representation is definitely not perfect. We don't take into account significant price levels (support and resistance), round price values, and other financial markets information. Incorporating them into the observation could be a challenging problem, which you could try exploring.
- Analyze market prices at different timeframes. Low-level data like one-minute bars are noisy (as they include lots of small price movements caused by individual trades), and it is like looking at the market using a microscope. At larger scales, such as one-hour or one-day bars, you can see large, long trends in data movement, which could be extremely important for price prediction.

In principle, our agent can look at price at various scales at the same time, taking into account not just recent low-level movements but overall trends (and recent **natural language processing (NLP)** innovations like transformers, attention mechanisms, and long context windows might be really helpful there).

- More training data is needed. One year of data for one stock is just 130k bars, which might be not enough to capture all market situations. Ideally, a real-life agent should be trained on a much larger dataset, such as the prices for hundreds of stocks for the past 10 years or more.
- Experiment with the network architecture. The convolution model has shown a bit faster convergence than the feed-forward model, but there are a lot of things to optimize: the count of layers, kernel size, residual architecture, attention mechanism, and so on.
- There are lots of similarities between NLP and financial data analysis: both work with human-created sequences of data that have variable length. You can try to represent price bars as "words" in some "financial language" (like "up price movement 1%" → token A, "up price movement 2%" → token B) and then apply NLP methods to this language. For example, train embeddings from the "sentences" to capture financial markets' structure, or use transformers or even LLMs for data prediction and classification.

Summary

In this chapter, we saw a practical example of RL and implemented a trading agent and a custom Gym environment. We tried two different architectures: a feed-forward network with price history on input and a 1D convolution network. Both architectures used the DQN method, with some of the extensions described in *Chapter 8*.

This was the last chapter in Part 2 of this book. In Part 3, we will talk about a different family of RL methods: *policy gradients*. We've touched on this approach a bit, but in the upcoming chapters, we will go much deeper into the subject, covering the REINFORCE method and the best method in the family: Asynchronous Advantage Actor-Critic, also known as A3C.

Leave a Review!

Thank you for purchasing this book from Packt Publishing—we hope you enjoy it! Your feedback is invaluable and helps us improve and grow. Once you've completed reading it, please take a moment to leave an Amazon review; it will only take a minute, but it makes a big difference for readers like you.

Scan the QR code below to receive a free ebook of your choice.

<https://packt.link/NzOWQ>



PART III

POLICY-BASED METHODS

11

Policy Gradients

In this first chapter of *Part 3* of the book, we will consider an alternative way to handle **Markov decision process (MDP)** problems, which form a full family of methods called **policy gradient** methods. In some situations, these methods work better than value-based methods, so it is really important to be familiar with them.

In this chapter, we will:

- Cover an overview of the methods, their motivations, and their strengths and weaknesses in comparison to the already familiar Q-learning
- Start with a simple policy gradient method called **REINFORCE** and try to apply it to our CartPole environment, comparing it with the **deep Q-network (DQN)** approach
- Discuss problems with the vanilla **REINFORCE** method and ways to address them with the **Policy Gradient (PG)** method, which is a step toward a much more advanced method, A3C, that we'll take a look at in the next chapter

Values and policy

Before getting to the main subject of this chapter, policy gradients, let's refresh our minds with the common characteristics of the methods covered in *Part 2* of this book. The central topic in *value iteration* and *Q-learning* is the value of the state (V_s) or value of the state and action ($Q_{s,a}$). Value is defined as the discounted total reward that we can gather from this state or by issuing this particular action from the state.

If we know this quantity, our decision on every step becomes simple and obvious: we just act greedily in terms of value, and that guarantees us a good total reward at the end of the episode. So, the values of states (in the case of the value iteration method) or state + action (in the case of Q-learning) stand between us and the best reward. To obtain these values, we have used the Bellman equation, which expresses the value in the current step via the value in the next step.

In *Chapter 1*, we defined the entity that tells us what to do in every state as the **policy**. As in Q-learning methods, when values are dictating to us how to behave, they are actually defining our policy. Formally, this can be written as $\pi(s) = \arg \max_a Q(s, a)$, which means that the result of our policy π , at every state s , is the action with the largest Q .

This policy-values connection is obvious, so I haven't placed emphasis on the policy as a separate entity, and we have spent most of our time talking about values and how to approximate them correctly. Now it's time to focus on this connection and the policy itself.

Why the policy?

There are several reasons why the policy is an interesting topic to explore. First of all, the policy is what we are looking for when we are dealing with a reinforcement learning problem. When the agent obtains the observation and needs to make a decision about what to do next, it needs the policy, not the value of the state or particular action. We do care about the total reward, but at every state, we may have little interest in the exact value of the state.

Imagine this situation: you're walking in the jungle and you suddenly realize that there is a hungry tiger hiding in the bushes. You have several alternatives, such as running, hiding, or trying to throw your backpack at it, but asking, "What's the exact value of the *run* action and is it larger than the value of the *do nothing* action?" is a bit silly. You don't care much about the value, because you need to make the decision on what to do quickly and that's it. Our Q-learning approach tried to answer the policy question indirectly by approximating the values of the states and trying to choose the best alternative, but if we are not interested in values, why do extra work?

Another reason why policies may be preferred is related to situations when an environment has lots of actions or, in the extreme case, with **continuous action space** problems. To be able to decide on the best action to take with $Q(s, a)$, we need to solve a small optimization problem, finding a , which maximizes $Q(s, a)$. In the case of an Atari game with several discrete actions, this wasn't a problem: we just approximated the values of all actions and took the action with the largest Q . If our action is not a small discrete set but has a scalar value attached to it, such as a steering wheel angle or the speed at which we want to run from the tiger, this optimization problem becomes hard because Q is usually represented by a highly nonlinear **neural network** (NN), so finding the argument that maximizes the function's values can be tricky.

In such cases, it's much more feasible to avoid values and work with the policy directly.

An extra benefit of policy learning is an environment with **stochasticity**. As you saw in *Chapter 8*, in a categorical DQN, our agent can benefit a lot from working with the distribution of Q-values, instead of expected mean values, as our network can more precisely capture the underlying probability distribution. As you will see in the next section, the policy is naturally represented as the probability of actions, which is a step in the same direction as the categorical DQN method.

Policy representation

Now that you know the benefits of the policy, let's give it a try. So, how do we represent the policy? In the case of Q-values, they were parametrized by the NN that returns values of actions as scalars. If we want our network to parametrize the actions, we have several options. The first and the simplest way could be just returning the identifier of the action (in the case of a discrete set of actions). However, this is not the best way to deal with a discrete set. A much more common solution, which is heavily used in classification tasks, is to return the **probability distribution** of our actions. In other words, for N mutually exclusive actions, we return N numbers representing the probability of taking each action in the given state (which we pass as an input to the network). This representation is shown in the following illustration:

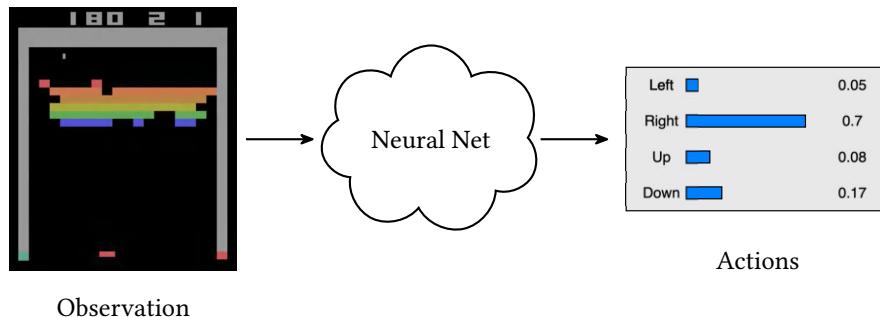


Figure 11.1: Policy approximation with an NN for a discrete set of actions

Such a representation of actions as probabilities has the additional advantage of *smooth representation*: if we change our network weights a bit, the output of the network will also change slightly. In the case of a discrete numbers output, even a small adjustment of the weights can lead to a jump to a different action. However, if our output is the probability distribution, a small change of weights will usually lead to a small change in output distribution, such as slightly increasing the probability of one action versus the others. This is a very nice property to have, as gradient optimization methods are all about tweaking the parameters of a model a bit to improve the results.

Policy gradients

We have decided on our policy representation, but what we haven't seen so far is how we are going to change our network's parameters to improve the policy. If you remember *Chapter 4*, we solved a very similar problem using the cross-entropy method: our network took observations as inputs and returned the probability distribution of the actions. In fact, the cross-entropy method is a younger brother of the methods that we will discuss in this part of the book. To start, we will get acquainted with the method called **REINFORCE**, which differs only slightly from the cross-entropy method, but first, we need to look at some mathematical notation that we will use in this and the following chapters.

We define the **policy gradient** as $\nabla J \approx \mathbb{E}[Q(s, a)\nabla \log \pi(a|s)]$. Of course, there is strong proof of this, but it's not that important. What interests us much more is the meaning of this expression.

The policy gradient defines the direction in which we need to change our network's parameters to improve the policy in terms of the accumulated total reward. The scale of the gradient is proportional to the value of the action taken, which is $Q(s, a)$ in the formula, and the gradient is equal to the gradient of the log probability of the action taken. This means that we are trying to increase the probability of actions that have given us good total rewards and decrease the probability of actions with bad final outcomes. Expectation, \mathbb{E} in the formula, just means that we average the gradient of several steps that we have taken in the environment.

From a practical point of view, policy gradient methods could be implemented by performing optimization of this loss function: $\mathcal{L} = -Q(s, a) \log \pi(a|s)$. The minus sign is important, as the loss function is **minimized** during **stochastic gradient descent (SGD)**, but we want to **maximize** our policy gradient. You will see code examples of policy gradient methods later in this and the following chapters.

The REINFORCE method

The formula of policy gradient that you have just seen is used by most policy-based methods, but the details can vary. One very important point is how exactly gradient scales, $Q(s, a)$, are calculated. In the cross-entropy method from *Chapter 4*, we played several episodes, calculated the total reward for each of them, and trained on transitions from episodes with a better-than-average reward. This training procedure is a policy gradient method with $Q(s, a) = 1$ for state and action pairs from good episodes (with a large total reward) and $Q(s, a) = 0$ for state and action pairs from worse episodes.

The cross-entropy method worked even with those simple assumptions, but the obvious improvement will be to use $Q(s, a)$ for training instead of just 0 and 1. Why should it help? The answer is a more fine-grained separation of episodes. For example, transitions from the episode with a total reward of 10 should contribute to the gradient more than transitions from the episode with the reward of 1.

Another reason to use $Q(s, a)$ instead of just 0 or 1 constants is to increase the probabilities of good actions at the beginning of the episode and decrease the probability of actions closer to the end of the episode. In the cross-entropy method, we take “elite” episodes and train on their actions regardless of the actions’ offset in the episode. By using $Q(s, a)$ (which includes discount factor γ), we put more emphasis on good actions in the beginning of the episode than on the actions at the end of the episode. That’s exactly the idea of the method called **REINFORCE**. Its steps are as follows:

1. Initialize the network with random weights.
2. Play N full episodes, saving their (s, a, r, s') transitions.
3. For every step, t , of every episode, k , calculate the discounted total reward for the subsequent steps:

$$Q_{k,t} = \sum_{i=0}^{\infty} \gamma^i r_i$$

4. Calculate the loss function for all transitions:

$$\mathcal{L} = - \sum_{k,t} Q_{k,t} \log(\pi(s_{k,t}, a_{k,t}))$$

5. Perform an SGD update of weights, minimizing the loss.
6. Repeat from step 2 until convergence is achieved.

This algorithm is different from Q-learning in several important ways:

- **No explicit exploration is needed:** In Q-learning, we used an epsilon-greedy strategy to explore the environment and prevent our agent from getting stuck with a non-optimal policy. Now, with the probabilities returned by the network, the exploration is performed automatically. At the beginning, the network is initialized with random weights, and it returns a uniform probability distribution. This distribution corresponds to random agent behavior.
- **No replay buffer is used:** Policy gradient methods belong to the on-policy methods class, which means that we can’t train on data obtained from the old policy. This is both good and bad. The good part is that such methods usually converge faster. The bad side is that they usually require much more interaction with the environment than off-policy methods such as DQN.
- **No target network is needed:** Here, we use Q-values, but they are obtained from our experience in the environment. In DQN, we used the target network to break the correlation in Q-value approximation, but we are not approximating anymore. In the next chapter, you will see that the target network trick can still be useful in policy gradient methods.

The CartPole example

To see the method in action, let's check the implementation of the REINFORCE method on the familiar CartPole environment. The full code of the example is in `Chapter11/02_cartpole_reinforce.py`.

In the beginning, we define hyperparameters (imports are omitted):

```
GAMMA = 0.99
LEARNING_RATE = 0.01
EPISODES_TO_TRAIN = 4
```

The `EPISODES_TO_TRAIN` value specifies how many complete episodes we will use for training.

The following network should also be familiar to you:

```
class PGN(nn.Module):
    def __init__(self, input_size: int, n_actions: int):
        super(PGN, self).__init__()

        self.net = nn.Sequential(
            nn.Linear(input_size, 128),
            nn.ReLU(),
            nn.Linear(128, n_actions)
        )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.net(x)
```

Note that despite the fact our network returns probabilities, we are not applying `softmax` nonlinearity to the output. The reason behind this is that we will use the PyTorch `log_softmax` function to calculate the logarithm of the softmax output at once. This method of calculation is much more numerically stable; however, we need to remember that output from the network is not probability, but raw scores (usually called logits).

This next function is a bit tricky:

```
def calc_qvals(rewards: tt.List[float]) -> tt.List[float]:
    res = []
    sum_r = 0.0
    for r in reversed(rewards):
        sum_r *= GAMMA
        sum_r += r
        res.append(sum_r)
    return list(reversed(res))
```

It accepts a list of rewards for the whole episode and needs to calculate the discounted total reward for every step. To do this efficiently, we calculate the reward from the end of the local reward list. Indeed, the last step of the episode will have a total reward equal to its local reward. The step before the last will have the total reward of $r_{t-1} + \gamma \cdot r_t$ (if t is an index of the last step).

Our `sum_r` variable contains the total reward for the previous steps, so to get the total reward for the current step, we need to multiply `sum_r` by γ and add the local reward from that step.

The preparation steps before the training loop should also be familiar to you:

```
if __name__ == "__main__":
    env = gym.make("CartPole-v1")
    writer = SummaryWriter(comment="-cartpole-reinforce")

    net = PGN(env.observation_space.shape[0], env.action_space.n)
    print(net)

    agent = ptan.agent.PolicyAgent(
        net, preprocessor=ptan.agent.float32_preprocessor, apply_softmax=True)
    exp_source = ExperienceSourceFirstLast(env, agent, gamma=GAMMA)

    optimizer = optim.Adam(net.parameters(), lr=LEARNING_RATE)
```

The only new element is the agent class from the PTAN library. Here, we are using `ptan.agent.PolicyAgent`, which needs to make a decision about actions for every observation. As our network now returns the policy as the probabilities of the actions, in order to select the action to take, we need to obtain the probabilities from the network and then perform random sampling from this probability distribution.

When we worked with DQN, the output of the network was Q-values, so if one action had a value of 0.4 and another action had a value of 0.5, the second action was preferred 100% of the time. In the case of the probability distribution, if the first action has a probability of 0.4 and the second 0.5, our agent should take the first action with a 40% chance and the second with a 50% chance. Of course, our network can decide to take the second action 100% of the time, and in this case, it returns a probability of 0 for the first action and a probability of 1 for the second action.

This difference is important to understand, but the change in the implementation is not large. Our `PolicyAgent` internally calls the NumPy `random.choice()` function with probabilities from the network. The `apply_softmax` argument instructs it to convert the network output to probabilities by calling `softmax` first. The third argument, `preprocessor`, is a way to get around the fact that the CartPole environment in Gymnasium returns the observation as a `float64` instead of the `float32` required by PyTorch.

Before we can start the training loop, we need several variables:

```
total_rewards = []
done_episodes = 0

batch_episodes = 0
batch_states, batch_actions, batch_qvals = [], [], []
cur_rewards = []
```

The first two variables, `total_rewards` and `done_episodes`, are used for reporting and contain the total rewards for the episodes and the count of completed episodes. The next few variables are used to gather the training data. The `cur_rewards` list contains local rewards for the episode being currently played. As this episode reaches the end, we calculate the discounted total rewards from local rewards using the `calc_qvals()` function and append them to the `batch_qvals` list. The `batch_states` and `batch_actions` lists contain states and actions that we saw in the last training.

The following code snippet is the beginning of the training loop:

```
for step_idx, exp in enumerate(exp_source):
    batch_states.append(exp.state)
    batch_actions.append(int(exp.action))
    cur_rewards.append(exp.reward)

    if exp.last_state is None:
        batch_qvals.extend(calc_qvals(cur_rewards))
        cur_rewards.clear()
        batch_episodes += 1
```

Every experience that we get from the experience source contains the state, action, local reward, and next state. If the end of the episode has been reached, the next state will be `None`. For non-terminal experience entries, we just save the state, action, and local reward in our lists. At the end of the episode, we convert the local rewards into Q-values and increment the episodes counter.

This part of the training loop is performed at the end of the episode and is responsible for reporting the current progress and writing metrics to TensorBoard:

```
new_rewards = exp_source.pop_total_rewards()
if new_rewards:
    done_episodes += 1
    reward = new_rewards[0]
    total_rewards.append(reward)
```

```

mean_rewards = float(np.mean(total_rewards[-100:]))
print(f"{step_idx}: reward: {reward:6.2f}, mean_100: {mean_rewards:6.2f}, "
      f"episodes: {done_episodes}")
writer.add_scalar("reward", reward, step_idx)
writer.add_scalar("reward_100", mean_rewards, step_idx)
writer.add_scalar("episodes", done_episodes, step_idx)
if mean_rewards > 450:
    print(f"Solved in {step_idx} steps and {done_episodes} episodes!")
    break

```

When enough episodes have passed since the last training step, we can optimize the gathered examples. As a first step, we convert states, actions, and Q-values into the appropriate PyTorch form:

```

if batch_episodes < EPISODES_TO_TRAIN:
    continue

optimizer.zero_grad()
states_t = torch.as_tensor(np.asarray(batch_states))
batch_actions_t = torch.as_tensor(np.asarray(batch_actions))
batch_qvals_t = torch.as_tensor(np.asarray(batch_qvals))

```

Then, we calculate the loss from the steps:

```

logits_t = net(states_t)
log_prob_t = F.log_softmax(logits_t, dim=1)
batch_idx = range(len(batch_states))
act_probs_t = log_prob_t[batch_idx, batch_actions_t]
log_prob_actions_v = batch_qvals_t * act_probs_t
loss_t = -log_prob_actions_v.mean()

```

Here, we ask our network to calculate states into logits and calculate the logarithm and softmax of them. On the third line, we select log probabilities from the actions taken and scale them with Q-values. On the last line, we average those scaled values and do negation to obtain the loss to minimize. To reiterate, this minus sign is very important because our policy gradient needs to be maximized to improve the policy. As the optimizer in PyTorch minimizes the loss function, we need to negate the policy gradient.

The rest of the code is clear:

```

loss_t.backward()
optimizer.step()

```

```
batch_episodes = 0
batch_states.clear()
batch_actions.clear()
batch_qvals.clear()

writer.close()
```

Here, we perform backpropagation to gather gradients in our variables and ask the optimizer to perform an SGD update. At the end of the training loop, we reset the episodes counter and clear our lists for fresh data to gather.

Results

For reference, I've implemented DQN in the CartPole environment with almost the same hyperparameters as our **REINFORCE** example. You'll find it in `Chapter11/01_cartpole_dqn.py`. Neither example requires any command-line arguments, and they should converge in less than a minute:

```
Chapter11$ ./02_cartpole_reinforce.py
PGN(
    (net): Sequential(
        (0): Linear(in_features=4, out_features=128, bias=True)
        (1): ReLU()
        (2): Linear(in_features=128, out_features=2, bias=True)
    )
)
31: reward: 31.00, mean_100: 31.00, episodes: 1
42: reward: 11.00, mean_100: 21.00, episodes: 2
54: reward: 12.00, mean_100: 18.00, episodes: 3
94: reward: 40.00, mean_100: 23.50, episodes: 4
159: reward: 65.00, mean_100: 31.80, episodes: 5
...
65857: reward: 500.00, mean_100: 440.60, episodes: 380
66357: reward: 500.00, mean_100: 442.42, episodes: 381
66857: reward: 500.00, mean_100: 445.59, episodes: 382
67357: reward: 500.00, mean_100: 448.24, episodes: 383
67857: reward: 500.00, mean_100: 451.31, episodes: 384
Solved in 67857 steps and 384 episodes!
```

The convergence dynamics for both DQN and **REINFORCE** are shown in the following charts. Your training dynamics may vary due to the randomness of the training.

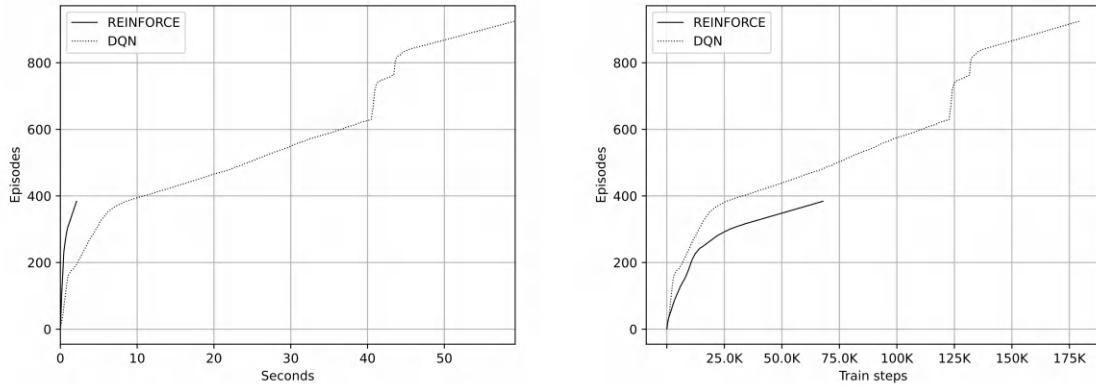


Figure 11.2: Count of episodes played over time (left) and training steps (right)

These two charts compare the count of episodes played over time and over the training steps.

The next chart compares the smoothed reward for episodes played:

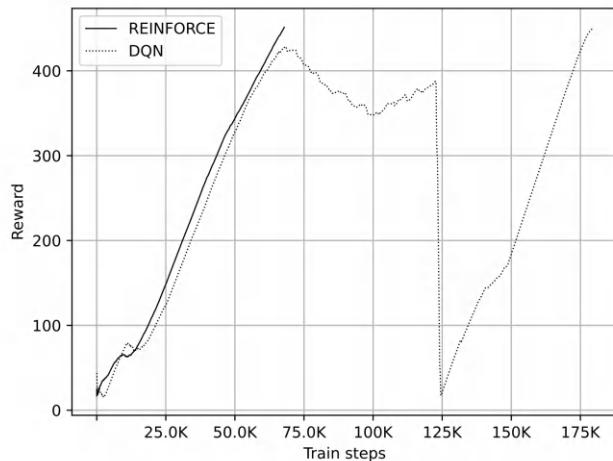


Figure 11.3: Reward dynamics for two methods

As you can see, the methods converged almost identically (with **REINFORCE** being slightly faster), but then DQN had problems when the mean reward climbed above 400 and had to start almost from scratch.

If you remember from *Chapter 4*, the cross-entropy method required about 40 batches of 16 episodes each to solve the CartPole environment, which is 640 episodes in total. The **REINFORCE** method was able to do the same in fewer than 400 episodes, which is a nice improvement.

Policy-based versus value-based methods

Let's now step back from the code that we have just seen and check the differences between these families of methods:

- Policy methods directly optimize what we care about: our behavior. Value methods, such as DQN, do the same indirectly, learning the value first and providing us with the policy based on this value.
- Policy methods are on-policy and require fresh samples from the environment. Value methods can benefit from old data, obtained from the old policy, human demonstration, and other sources.
- Policy methods are usually less sample-efficient, which means they require more interaction with the environment. Value methods can benefit from large replay buffers. However, sample efficiency doesn't mean that value methods are more computationally efficient, and very often, it's the opposite.
- In the preceding example, during the training, we needed to access our NN only once, to get the probabilities of actions. In DQN, we need to process two batches of states: one for the current state and another for the next state in the Bellman update.

As you can see, there is no strong preference for one family or another. In some situations, policy methods will be the more natural choice, like in continuous control problems or cases when access to the environment is cheap and fast. However, there are many situations when value methods will shine, for example, the recent state-of-the-art results on Atari games achieved by DQN variants. Ideally, you should be familiar with both families and understand the strong and weak sides of both camps.

In the next section, we will talk about the **REINFORCE** method's limitations, ways to improve it, and how to apply a policy gradient method to our favorite Pong game.

REINFORCE issues

In the previous section, we discussed the **REINFORCE** method, which is a natural extension of the cross-entropy method. Unfortunately, both **REINFORCE** and the cross-entropy method still suffer from several problems, which make both of them limited to simple environments.

Full episodes are required

First of all, we still need to wait for the full episode to complete before we can start training. Even worse, both **REINFORCE** and the cross-entropy method behave better with more episodes used for training (just because more episodes means more training data, which means more accurate policy gradients).

This situation is fine for short episodes in the CartPole, when in the beginning, we can barely handle the bar for more than 10 steps; but in Pong, it is completely different: every episode can last for hundreds or even thousands of frames. It's equally bad from the training perspective, as our training batch becomes very large, and from the sample efficiency perspective, as we need to communicate with the environment a lot just to perform a single training step.

The purpose of the complete episode requirement is to get as accurate a Q-estimation as possible. When we talked about DQN, you saw that, in practice, it's fine to replace the exact value for a discounted reward with our estimation using the one-step Bellman equation: $Q(s, a) = r_a + \gamma V(s')$. To estimate $V(s)$, we used our own Q-estimation, but in the case of the policy gradient, we don't have $V(s)$ or $Q(s, a)$ anymore.

To overcome this, two approaches exist:

- We can ask our network to estimate $V(s)$ and use this estimation to obtain Q . This approach will be discussed in the next chapter and is called the *actor-critic method*, which is the most popular method from the policy gradient family.
- Alternatively, we can do the Bellman equation, unrolling N steps ahead, which will effectively exploit the fact that the value contribution decreases when gamma is less than 1. Indeed, with $\gamma = 0.9$, the value coefficient at the 10th step will be $0.9^{10} \approx 0.35$. At step 50, this coefficient will be $0.9^{50} \approx 0.00515$, which is a really small contribution to the total reward. With $\gamma = 0.99$, the required count of steps will become larger, but we can still do this.

High gradient variance

In the policy gradient formula, $\nabla J \approx \mathbb{E}[Q(s, a)\nabla \log \pi(a|s)]$, we have a gradient proportional to the discounted reward from the given state. However, the range of this reward is heavily environment-dependent. For example, in the CartPole environment, we get a reward of 1 for every timestamp that we are holding the pole vertically. If we can do this for five steps, we get a total (undiscounted) reward of 5. If our agent is smart and can hold the pole for, say, 100 steps, the total reward will be 100. The difference in value between those two scenarios is 20 times, which means that the scale between the gradients of unsuccessful samples will be 20 times lower than for more successful ones. Such a large difference can seriously affect our training dynamics, as one lucky episode will dominate in the final gradient.

In mathematical terms, the policy gradient has high variance, and we need to do something about this in complex environments; otherwise, the training process can become unstable. The usual approach to handling this is subtracting a value called the baseline from the Q . The possible choices for the baseline are as follows:

- Some constant value, which is normally the mean of the discounted rewards
- The moving average of the discounted rewards

- The value of the state, $V(s)$

To illustrate the baseline effect on the training, in `Chapter11/03_cartpole_reinforce_baseline.py` I implemented the second way of calculating the baseline (the average of rewards). The only difference with the version you've already seen is in the `calc_qvals()` function. I'm not going to discuss the results here; you can experiment yourself.

Exploration problems

Even with the policy represented as the probability distribution, there is a high chance that the agent will converge to some locally optimal policy and stop exploring the environment. In DQN, we solved this using epsilon-greedy action selection: with the probability ϵ , the agent took a random action instead of the action dictated by the current policy. We can use the same approach, of course, but policy gradient methods allow us to follow a better path, called the *entropy bonus*.

In information theory, entropy is a measure of uncertainty in a system. Being applied to the agent's policy, entropy shows how uncertain the agent is about which action to take. In math notation, the entropy of the policy is defined as $H(\pi) = -\sum \pi(a|s) \log \pi(a|s)$. The value of entropy is always greater than zero and has a single maximum when the policy is uniform; in other words, all actions have the same probability. Entropy becomes minimal when our policy has 1 for one action and 0 for all others, which means that the agent is absolutely sure what to do. To prevent our agent from being stuck in the local minimum, we subtract the entropy from the loss function, punishing the agent for being too certain about the action to take.

High correlation of samples

As we discussed in *Chapter 6*, training samples in a single episode are usually heavily correlated, which is bad for SGD training. In the case of DQN, we solved this issue by having a large replay buffer with a size from 100,000 to several million observations. This solution is not applicable to the policy gradient family anymore because those methods belong to the on-policy class. The implication is simple: using old samples generated by the old policy, we will get policy gradients for that old policy, not for our current one.

The obvious, but unfortunately wrong, solution would be to reduce the replay buffer size. It might work in some simple cases but, in general, we need fresh training data generated by our current policy. To solve this, parallel environments are normally used. The idea is simple: instead of communicating with one environment, we use several and use their transitions as training data.

Policy gradient methods on CartPole

Nowadays, almost nobody uses the vanilla policy gradient method, as the much more stable actor-critic method exists. However, I still want to show the policy gradient implementation, as it establishes very important concepts and metrics to check the policy gradient method's performance.

Implementation

We will start with a much simpler environment of CartPole, and in the next section, we will check its performance in our favorite Pong environment. The complete code for the following example is available in `Chapter11/04_cartpole_pg.py`.

Besides the already familiar hyperparameters, we have two new ones:

```
GAMMA = 0.99
LEARNING_RATE = 0.001
ENTROPY_BETA = 0.01
BATCH_SIZE = 8

REWARD_STEPS = 10
```

The `ENTROPY_BETA` value is the scale of the entropy bonus and the `REWARD_STEPS` value specifies how many steps ahead the Bellman equation is unrolled to estimate the discounted total reward of every transition.

The following is the network architecture:

```
class PGN(nn.Module):
    def __init__(self, input_size: int, n_actions: int):
        super(PGN, self).__init__()

        self.net = nn.Sequential(
            nn.Linear(input_size, 128),
            nn.ReLU(),
            nn.Linear(128, n_actions)
        )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.net(x)
```

This is exactly the same as in the previous examples for CartPole: a two-layer network with 128 neurons in the hidden layer. The preparation code is also the same as before, except the experience source is asked to unroll the Bellman equation for 10 steps.

The following is the part that differs from `04_cartpole_pg.py`:

```
exp_source = ptan.experience.ExperienceSourceFirstLast(
    env, agent, gamma=GAMMA, steps_count=REWARD_STEPS)
```

In the training loop, we maintain the sum of the discounted reward for every transition and use it to calculate the baseline for the policy scale:

```
for step_idx, exp in enumerate(exp_source):
    reward_sum += exp.reward
    baseline = reward_sum / (step_idx + 1)
    writer.add_scalar("baseline", baseline, step_idx)
    batch_states.append(exp.state)
    batch_actions.append(int(exp.action))
    batch_scales.append(exp.reward - baseline)
```

In the loss calculation, we use the same code as before to calculate the policy loss (which is the negated policy gradient):

```
optimizer.zero_grad()
logits_t = net(states_t)
log_prob_t = F.log_softmax(logits_t, dim=1)
act_probs_t = log_prob_t[range(BATCH_SIZE), batch_actions_t]
log_prob_actions_t = batch_scale_t * act_probs_t
loss_policy_t = -log_prob_actions_t.mean()
```

Then we add the entropy bonus to the loss by calculating the entropy of the batch and subtracting it from the loss. As entropy has a maximum for uniform probability distribution and we want to push the training toward this maximum, we need to subtract from the loss.

```
prob_t = F.softmax(logits_t, dim=1)
entropy_t = -(prob_t * log_prob_t).sum(dim=1).mean()
entropy_loss_t = -ENTROPY_BETA * entropy_t
loss_t = loss_policy_t + entropy_loss_t

loss_t.backward()
optimizer.step()
```

Then, we calculate the **Kullback-Leibler (KL)** divergence between the new policy and the old policy. KL divergence in information theory measures how one probability distribution diverges from another expected probability distribution, as we saw in *Chapter 4*. In our example, it is being used to compare the policy returned by the model before and after the optimization step:

```
new_logits_t = net(states_t)
new_prob_t = F.softmax(new_logits_t, dim=1)
kl_div_t = -((new_prob_t / prob_t).log() * prob_t).\
    sum(dim=1).mean()
writer.add_scalar("kl", kl_div_t.item(), step_idx)
```

High spikes in KL are usually a bad sign since it means that our policy was pushed too far from the previous policy, which is a bad idea most of the time (as our NN is a very nonlinear function in a high-dimensional space, such large changes in the model weight could have a very strong influence on the policy).

Finally, we calculate the statistics about the gradients on this training step. It's usually good practice to show the graph of the maximum and L2 norm (which is the length of the vector) of gradients to get an idea about the training dynamics.

```
grad_max = 0.0
grad_means = 0.0
grad_count = 0
for p in net.parameters():
    grad_max = max(grad_max, p.grad.abs().max().item())
    grad_means += (p.grad ** 2).mean().sqrt().item()
    grad_count += 1
```

At the end of the training loop, we dump all the values that we want to monitor in TensorBoard:

```
writer.add_scalar("baseline", baseline, step_idx)
writer.add_scalar("entropy", entropy, step_idx)
writer.add_scalar("loss_entropy", l_entropy, step_idx)
writer.add_scalar("loss_policy", l_policy, step_idx)
writer.add_scalar("loss_total", l_total, step_idx)
writer.add_scalar("grad_l2", grad_means / grad_count, step_idx)
writer.add_scalar("grad_max", grad_max, step_idx)
writer.add_scalar("batch_scales", bs_smoothed, step_idx)

batch_states.clear()
batch_actions.clear()
batch_scales.clear()
```

Results

In this example, we will plot a lot of charts in TensorBoard. Let's start with the familiar one: reward. As you can see in the following chart, the dynamics and performance are not very different from the REINFORCE method:

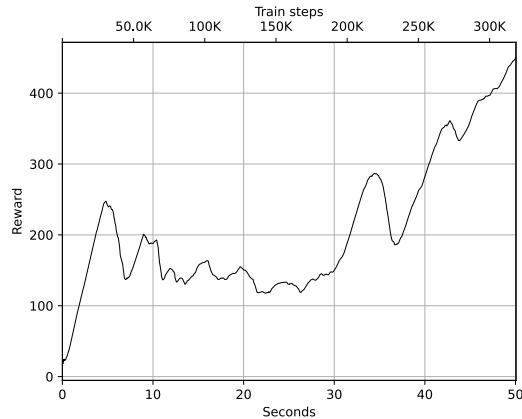


Figure 11.4: The reward dynamics of the policy gradient method

The next two charts are related to our baseline and scales of policy gradients. We expect the baseline to converge to $1 + 0.99 + 0.99^2 + \dots + 0.99^9$, which is approximately 9.56. Scales of policy gradients should oscillate around zero. That's exactly what we can see in the following graph:

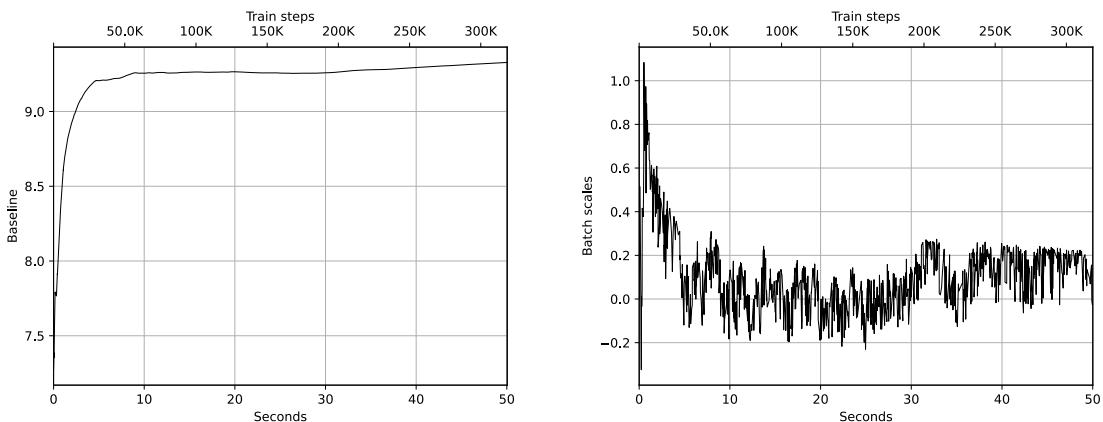


Figure 11.5: Baseline value (left) and batch scales (right)

The entropy is decreasing over time from 0.69 to 0.52 (*Figure 11.6*). The starting value corresponds to the maximum entropy with two actions, which is approximately 0.69:

$$H(\pi) = - \sum_a \pi(a|s) \log \pi(a|s) = \\ - \left(\frac{1}{2} \log \left(\frac{1}{2} \right) + \frac{1}{2} \log \left(\frac{1}{2} \right) \right) \approx 0.69$$

The fact that the entropy is decreasing during the training, as indicated by the following chart, shows that our policy is moving from uniform distribution to more deterministic actions:

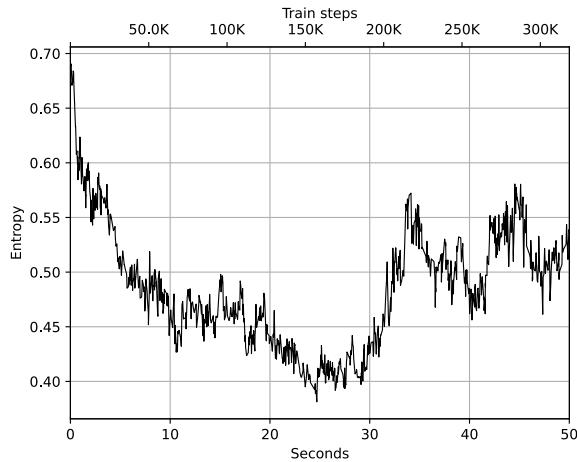


Figure 11.6: Entropy during the training

The next group of plots (*Figure 11.7* and *Figure 11.8*) is related to loss, which includes policy loss, entropy loss, and their sum. The entropy loss is scaled and is a mirrored version of the preceding entropy chart. The policy loss shows the mean scale and direction of the policy gradient computed on the batch.

Here, we should check the relative size of both of them to prevent entropy loss from dominating too much.

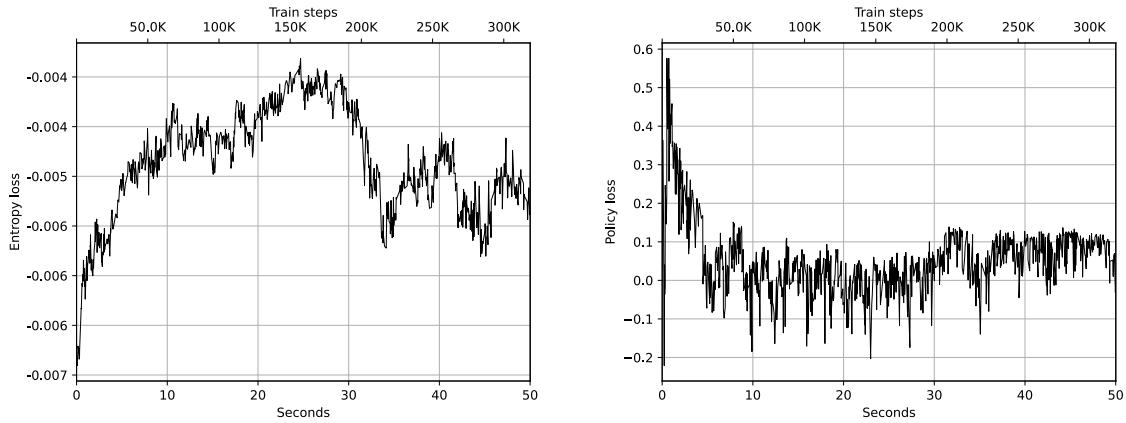


Figure 11.7: Entropy loss (left) and policy loss (right)

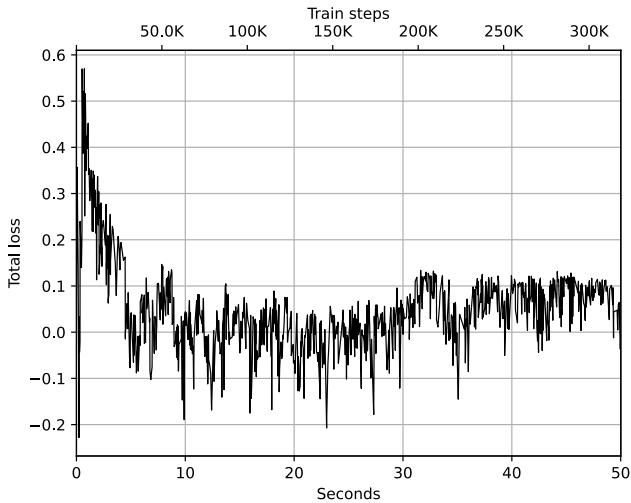


Figure 11.8: Total loss

The final set of charts (*Figure 11.9* and *Figure 11.10*) shows the gradient's L2 values, the maximum of L2, and KL. Our gradients look healthy during the whole training: they are not too large and not too small, and there are no huge spikes. The KL charts also look normal as there are some spikes, but they are not very large and don't exceed 10^{-3} :

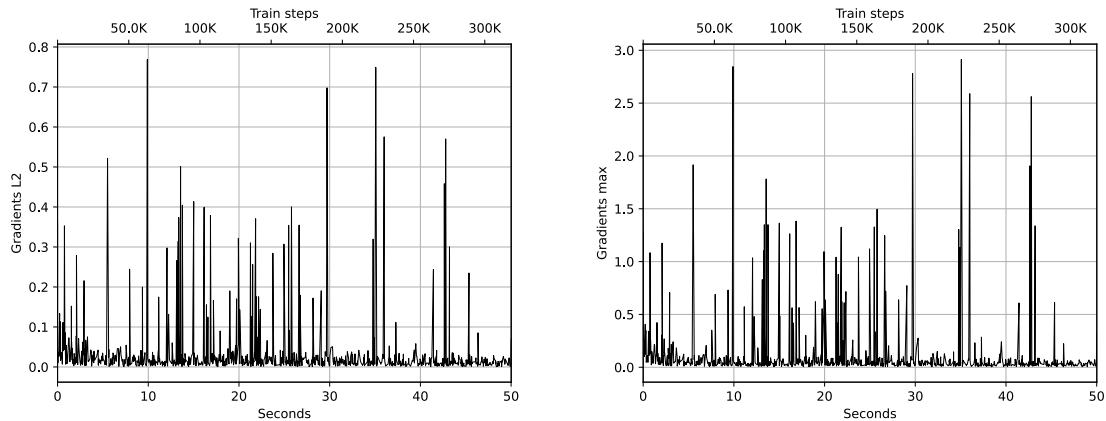


Figure 11.9: Gradients L2 (left) and maximum value (right)

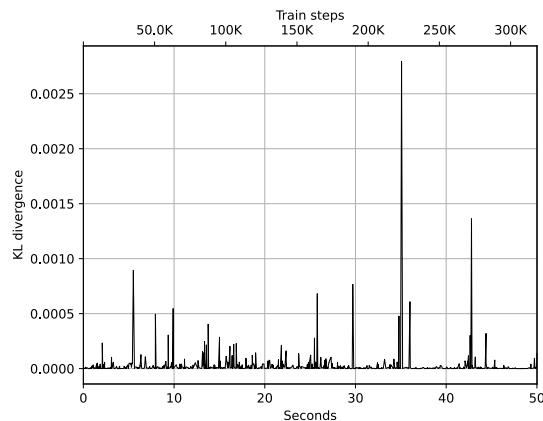


Figure 11.10: KL divergence

Policy gradient methods on Pong

As we've seen in the previous section, the vanilla policy gradient method works well on a simple CartPole environment, but it works surprisingly badly in more complicated environments.

For the relatively simple Atari game Pong, our DQN was able to completely solve it in 1 million frames and showed positive reward dynamics in just 100,000 frames, whereas the policy gradient method failed to converge. Due to the instability of policy gradient training, it became very hard to find good hyperparameters and was still very sensitive to initialization.

This doesn't mean that the policy gradient method is bad, because, as you will see in the next chapter, just one tweak of the network architecture to get a better baseline in the gradients will turn the policy gradient method into one of the best methods (the asynchronous advantage actor-critic method). Of course, there is a good chance that my hyperparameters are completely wrong or the code has some hidden bugs, or there could be other unforeseen problems. Regardless, unsuccessful results still have value, at least as a demonstration of bad convergence dynamics.

Implementation

You can find the complete code for the example in `Chapter11/05_pong_pg.py`.

The three main differences from the previous example's code are as follows:

- The baseline is estimated with a moving average for 1 million past transitions, instead of all examples.

To make moving average calculations faster, a deque-backed buffer is created:

```
class MeanBuffer:
    def __init__(self, capacity: int):
        self.capacity = capacity
        self.deque = collections.deque(maxlen=capacity)
        self.sum = 0.0

    def add(self, val: float):
        if len(self.deque) == self.capacity:
            self.sum -= self.deque[0]
        self.deque.append(val)
        self.sum += val

    def mean(self) -> float:
        if not self.deque:
            return 0.0
        return self.sum / len(self.deque)
```

- Several concurrent environments are used. The second difference in this example is working with multiple environments, and this functionality is supported by the PTAN library. The only action we have to take is to pass the array of Env objects to the ExperienceSource class. All the rest is done automatically. In the case of several environments, the experience source asks them for transitions in round-robin fashion, providing us with less-correlated training samples.
- Gradients are clipped to improve training stability. The last difference from the CartPole example is gradient clipping, which is performed using the PyTorch `clip_grad_norm` function from the `torch.nn.utils` package.

The hyperparameters for the best variant are the following:

```
GAMMA = 0.99
LEARNING_RATE = 0.0001
ENTROPY_BETA = 0.01
BATCH_SIZE = 128

REWARD_STEPS = 10
BASELINE_STEPS = 1000000
GRAD_L2_CLIP = 0.1

ENV_COUNT = 32
```

Results

Despite all my efforts to make the example converge, it wasn't very successful. Even after hyperparameter tuning (≈ 400 samples of hyperparameters), the best result has the average reward around -19.7 after 1 million training steps.

You can try it yourself, the code is in `Chapter11/05_pong_pg.py` and `Chapter11/05_pong_pg_tune.py`. But I can only conclude that Pong turned out to be too complex for the vanilla PG method.

Summary

In this chapter, you saw an alternative way of solving RL problems: policy gradient methods, which are different in many ways from the familiar DQN method. We explored a basic method called REINFORCE, which is a generalization of our first method in RL-domain cross-entropy. This policy gradient method is simple, but when applied to the Pong environment, it didn't produce good results.

In the next chapter, we will consider ways to improve the stability of policy gradient methods by combining the families of value-based and policy-based methods.

12

Actor-Critic Method: A2C and A3C

In *Chapter 11*, we started to investigate a policy-based alternative to the familiar value-based methods family. In particular, we focused on the method called **REINFORCE** and its modification, which uses discounted reward to obtain the gradient of the policy (which gives us the direction in which to improve the policy). Both methods worked well for a small CartPole problem, but for a more complicated Pong environment, we got no convergence.

Here, we will discuss another extension to the vanilla policy gradient method, which magically improves the stability and convergence speed of that method. Despite the modification being only minor, the new method has its own name, **actor-critic**, and it's one of the most powerful methods in deep **reinforcement learning (RL)**.

In this chapter, we will:

- Explore how the baseline impacts statistics and the convergence of gradients
- Cover an extension of the baseline idea
- Implement the **advantage actor-critic (A2C)** method and check it on the Pong environment
- Add asynchronous execution to the A2C method using two different ways: data parallelism and gradient parallelism

Variance reduction

In the previous chapter, I briefly mentioned that one of the ways to improve the stability of policy gradient methods is to reduce the variance of the gradient. Now let's try to understand why this is important and what it means to reduce the variance. In statistics, variance is the expected square deviation of a random variable from the expected value of that variable:

$$\text{Var}[x] = \mathbb{E}[(x - \mathbb{E}[x])^2]$$

Variance shows us how far values are dispersed from the mean. When variance is high, the random variable can take values that deviate widely from the mean. In the following plot, there is a normal (Gaussian) distribution with the same value for the mean, $\mu = 10$, but with different values for the variance.

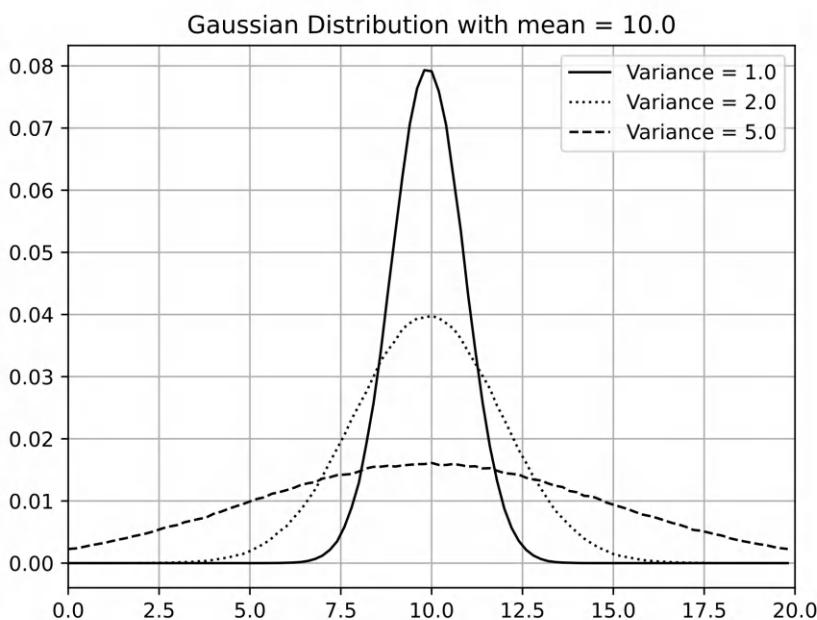


Figure 12.1: The effect of variance on Gaussian distribution

Now let's return to policy gradients. It was stated in the previous chapter that the idea is to increase the probability of good actions and decrease the chance of bad ones. In math notation, our policy gradient was written as $\nabla J \approx \mathbb{E}[Q(s, a)\nabla \log \pi(a|s)]$. The scaling factor $Q(s, a)$ specifies how much we want to increase or decrease the probability of the action taken in the particular state.

In the **REINFORCE** method, we used the discounted total reward as the scaling of the gradient. In an attempt to increase **REINFORCE** stability, we subtracted the mean reward from the gradient scale.

To understand why this helped, let's consider the very simple scenario of an optimization step on which we have three actions with different total discounted rewards: Q_1 , Q_2 , and Q_3 . Now let's check what will happen with policy gradients with regard to the relative values of those Q_s .

As the first example, let both Q_1 and Q_2 be equal to some small positive number and Q_3 be a large negative number. So, actions at the first and second steps led to some small reward, but the third step was not very successful. The resulting **combined** gradient for all three steps will try to push our policy far from the action at step three and slightly toward the actions taken at steps one and two, which is a totally reasonable thing to do.

Now let's imagine that our reward is always positive and only the value is different. This corresponds to adding some constant to each of the rewards from the previous example: Q_1 , Q_2 , and Q_3 . In this case, Q_1 and Q_2 will become large positive numbers and Q_3 will have a small positive value. However, our policy update will become different! We will try hard to push our policy toward actions at the first and second steps, and slightly push it toward an action at step three. So, strictly speaking, we are no longer trying to avoid the action taken for step three, despite the fact that the relative rewards are the same.

This dependency of our policy update on the constant added to the reward can slow down our training significantly, as we may require many more samples to *average out* the effect of such a shift in the policy gradient. Even worse, as our total discounted reward changes over time, with the agent learning how to act better and better, our policy gradient variance can also change. For example, in the Atari Pong environment, the average reward in the beginning is $-21 \dots -20$, so all the actions look almost equally bad.

To overcome this in the previous chapter, we subtracted the mean total reward from the Q-value and called this mean the **baseline**. This trick normalized our policy gradient: in the case of the average reward being -21 , getting a reward of -20 looks like a win for the agent and it pushes its policy toward the taken actions.

CartPole variance

To check this theoretical conclusion in practice, let's plot our policy gradient variance during the training for both the baseline version and the version without the baseline. The complete example is in `Chapter12/01_cartpole_pg.py`, and most of the code is the same as in *Chapter 11*. The differences in this version are the following:

- It now accepts the command-line option `-baseline`, which enables the mean subtraction from the reward. By default, no baseline is used.

- On every training loop, we gather the gradients from the policy loss and use this data to calculate the variance.

To gather only the gradients from the policy loss and exclude the gradients from the entropy bonus added for exploration, we need to calculate the gradients in two stages. Luckily, PyTorch allows this to be done easily. In the following code, only the relevant part of the training loop is included to illustrate the idea:

```
optimizer.zero_grad()
logits_v = net(states_v)
log_prob_v = F.log_softmax(logits_v, dim=1)
log_p_a_v = log_prob_v[range(BATCH_SIZE), batch_actions_t]
log_prob_actions_v = batch_scale_v * log_p_a_v
loss_policy_v = -log_prob_actions_v.mean()
```

We calculate the policy loss as before, by calculating the log from the probabilities of taken actions and multiplying it by policy scales (which are the total discounted reward if we are not using the baseline or the total reward minus the baseline).

In the next step, we ask PyTorch to backpropagate the policy loss, calculating the gradients and keeping them in our model's buffers:

```
loss_policy_v.backward(retain_graph=True)
```

As we have previously performed `optimizer.zero_grad()`, those buffers will contain only the gradients from the policy loss. One tricky thing here is the `retain_graph=True` option when we call `backward()`. It instructs PyTorch to keep the graph structure of the variables. Normally, this is destroyed by the `backward()` call, but in our case, this is not what we want. In general, retaining the graph could be useful when we need to backpropagate the loss multiple times before the call to the optimizer, although this is not a very common situation.

Then, we iterate all parameters from our model (every parameter of our model is a tensor with gradients) and extract their `grad` field in a flattened NumPy array:

```
grads = np.concatenate([p.grad.data.numpy().flatten()
                       for p in net.parameters()
                       if p.grad is not None])
```

This gives us one long array with all gradients from our model's variables. However, our parameter update should take into account not only the policy gradient but also the gradient provided by our entropy bonus.

To achieve this, we calculate the entropy loss and call `backward()` again. To be able to do this the second time, we need to pass `retain_graph=True`.

On the second `backward()` call, PyTorch will backpropagate our entropy loss and add the gradients to the internal gradients' buffers. So, what we now need to do is just ask our optimizer to perform the optimization step using those combined gradients:

```
prob_v = F.softmax(logits_v, dim=1)
entropy_v = -(prob_v * log_prob_v).sum(dim=1).mean()
entropy_loss_v = -ENTROPY_BETA * entropy_v
entropy_loss_v.backward()
optimizer.step()
```

Later, the only thing we need to do is write statistics that we are interested in into TensorBoard:

```
g_l2 = np.sqrt(np.mean(np.square(grads)))
g_max = np.max(np.abs(grads))
writer.add_scalar("grad_l2", g_l2, step_idx)
writer.add_scalar("grad_max", g_max, step_idx)
writer.add_scalar("grad_var", np.var(grads), step_idx)
```

By running this example twice, once with the `-baseline` command-line option and once without it, we get a plot of variance of our policy gradient. The following charts show the smoothed reward (average for last 100 episodes) and variance (smoothed with window 20):

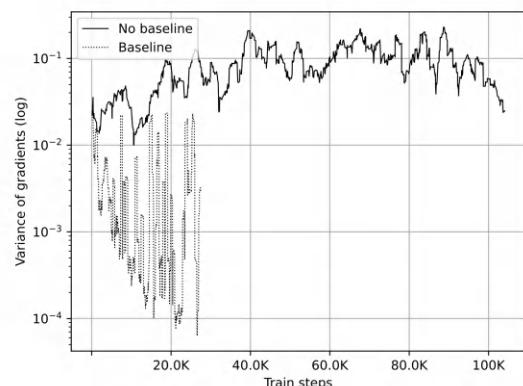
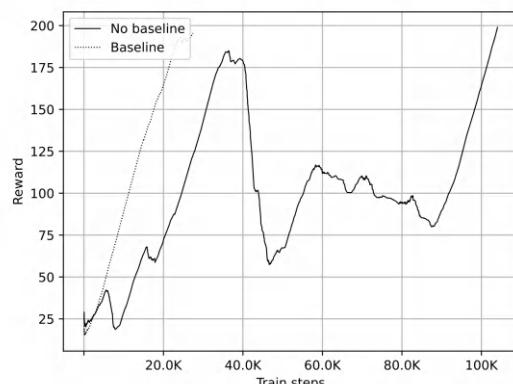


Figure 12.2: Smoothed reward (left) and variance (right)

These next two charts show the gradients' magnitude (L2) and maximum value. All values are smoothed with window 20:

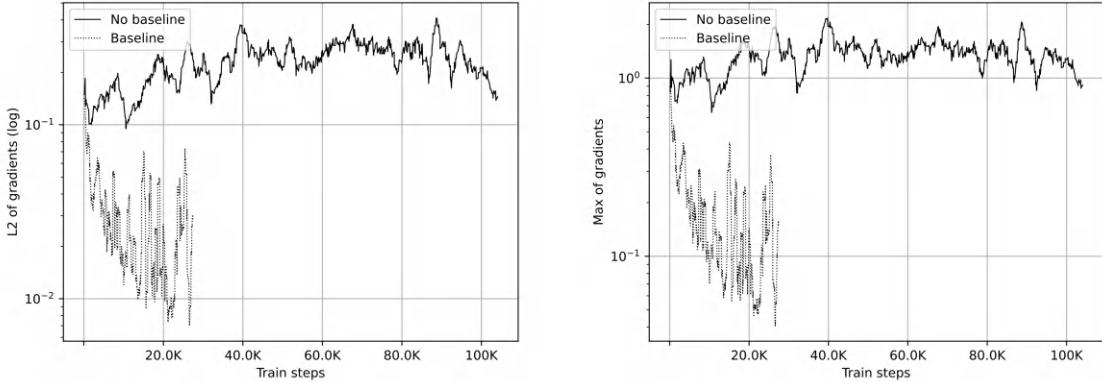


Figure 12.3: Gradients' L2 norm (left) and maximum value (right)

As you can see, variance for the version with the baseline is two to three orders of magnitude lower than the version without one, which helps the system to converge faster.

Advantage actor-critic (A2C)

The next step in reducing the variance is making our baseline state-dependent (which is a good idea, as different states could have very different baselines). Indeed, to decide on the suitability of a particular action in some state, we use the discounted total reward of the action. However, the total reward itself could be represented as a *value* of the state plus the *advantage* of the action: $Q(s, a) = V(s) + A(s, a)$. You saw this in *Chapter 8*, when we discussed DQN modifications, particularly dueling DQN.

So, why can't we use $V(s)$ as a baseline? In that case, the scale of our gradient will be just advantage, $A(s, a)$, showing how this taken action is better in respect to the average state's value. In fact, we can do this, and it is a very good idea for improving the policy gradient method. The only problem here is that we don't know the value, $V(s)$, of the state that we need to subtract from the discounted total reward, $Q(s, a)$. To solve this, let's use *another neural network*, which will approximate $V(s)$ for every observation. To train it, we can exploit the same training procedure we used in DQN methods: we will carry out the Bellman step and then minimize the mean square error to improve $V(s)$ approximation.

When we know the value for any state (or at least have some approximation of it), we can use it to calculate the policy gradient and update our policy network to increase probabilities for actions with good advantage values and decrease the chance of actions with bad advantage values.

The policy network (which returns a probability distribution of actions) is called the *actor*, as it tells us what to do. Another network is called *critic*, as it allows us to understand how good our actions were by returning $V(s)$. This improvement is known under a separate name, the **advantage actor-critic method**, which is often abbreviated to A2C. *Figure 12.4* is an illustration of its architecture:

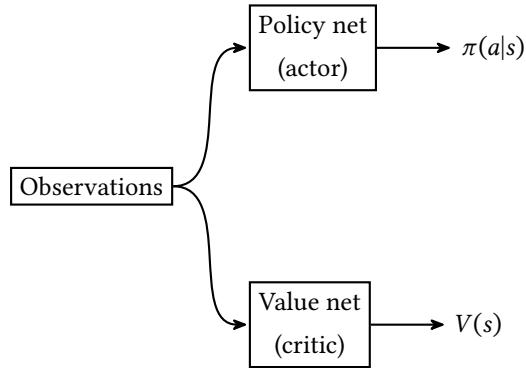


Figure 12.4: The A2C architecture

In practice, the policy and value networks partially overlap, mostly due to efficiency and convergence considerations. In this case, the policy and value are implemented as different heads of the network, taking the output from the common body and transforming it into the probability distribution and a single number representing the value of the state.

This helps both networks to share low-level features (such as convolution filters in the Atari agent), but combine them in a different way. The following figure shows this architecture:

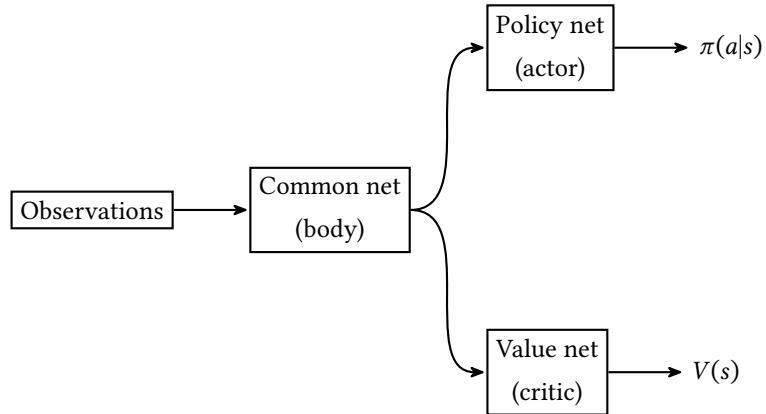


Figure 12.5: The A2C architecture with a shared network body

From a training point of view, we complete these steps:

1. Initialize network parameters, θ , with random values.
2. Play N steps in the environment, using the current policy, π_θ , and saving the state, s_t , action, a_t , and reward, r_t .
3. Set $R \leftarrow 0$ if the end of the episode is reached or $V_\theta(s_t)$.
4. For $i = t - 1 \dots t_{start}$ (note that steps are processed backward):
 - $R \leftarrow r_i + \gamma R$
 - Accumulate the policy gradients:

$$\partial\theta_\pi \leftarrow \partial\theta_\pi + \nabla_\theta \log \pi_\theta(a_i|s_i)(R - V_\theta(s_i))$$

- Accumulate the value gradients:

$$\partial\theta_v \leftarrow \partial\theta_v + \frac{\partial(R - V_\theta(s_i))^2}{\partial\theta_v}$$

5. Update the network parameters using the accumulated gradients, moving in the direction of the policy gradients, $\partial\theta_\pi$, and in the opposite direction of the value gradients, $\partial\theta_v$.
6. Repeat from step 2 until convergence is reached.

This algorithm is just an outline and similar to those that are usually printed in research papers. In practice, several extensions to improve the stability of the method may be used:

- An entropy bonus is usually added to improve exploration. It's typically written as an entropy value added to the loss function:

$$\mathcal{L}_H = \beta \sum_i \pi_\theta(s_i) \log \pi_\theta(s_i)$$

This function has a minimum when the probability distribution is uniform, so by adding it to the loss function, we push our agent away from being too certain about its actions. The value of β is a hyperparameter scaling the entropy bonus and prioritizing the exploration during the training. Normally, it is constant or linearly decreased during the training.

- Gradient accumulation is usually implemented as a loss function combining all three components: policy loss, value loss, and entropy loss. You should be careful with signs of these losses, as policy gradients show you the direction of policy improvement, but both the value and entropy losses should be minimized.

- To improve stability, it's worth using several environments, providing you with observations concurrently (when you have multiple environments, your training batch will be created from their observations). We will look at several ways of doing this later in this chapter when we discuss the A3C method.

The version of the preceding method that uses several environments running in parallel is called advantage asynchronous actor-critic, which is also known as A3C. The A3C method will be discussed later, but for now, let's implement A2C.

A2C on Pong

In the previous chapter, you saw a (not very successful) attempt to solve our favorite Pong environment with policy gradient methods. Let's try it again with the actor-critic method at hand. The full source code is available in `Chapter12/02_pong_a2c.py`.

We start, as usual, by defining hyperparameters (imports are omitted):

```
GAMMA = 0.99
LEARNING_RATE = 0.001
ENTROPY_BETA = 0.01
BATCH_SIZE = 128
NUM_ENVS = 50

REWARD_STEPS = 4
CLIP_GRAD = 0.1
```

These values are not tuned, which is left as an exercise for the reader. We have one new value here: `CLIP_GRAD`. This hyperparameter specifies the threshold for gradient clipping, which basically prevents our gradients from becoming too large at the optimization stage and pushing our policy too far. Clipping is implemented using the PyTorch functionality, but the idea is very simple: if the L2 norm of the gradient is larger than this hyperparameter, then the gradient vector is clipped to this value.

The `REWARD_STEPS` hyperparameter determines how many steps ahead we will take to approximate the total discounted reward for every action.

In the policy gradient methods, we used about 10 steps, but in A2C, we will use our value approximation to get a state value for further steps, so it will be fine to decrease the number of steps.

The following is our network architecture:

```
class AtariA2C(nn.Module):
    def __init__(self, input_shape: tt.Tuple[int, ...], n_actions: int):
        super(AtariA2C, self).__init__()

        self.conv = nn.Sequential(
            nn.Conv2d(input_shape[0], 32, kernel_size=8, stride=4),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, stride=1),
            nn.ReLU(),
            nn.Flatten(),
        )

        size = self.conv(torch.zeros(1, *input_shape)).size()[-1]
        self.policy = nn.Sequential(
            nn.Linear(size, 512),
            nn.ReLU(),
            nn.Linear(512, n_actions)
        )
        self.value = nn.Sequential(
            nn.Linear(size, 512),
            nn.ReLU(),
            nn.Linear(512, 1)
        )
```

It has a shared convolution body and two heads: the first returns the policy with the probability distribution over our actions and the second head returns one single number, which will approximate the state's value. It might look similar to our dueling DQN architecture from *Chapter 8*, but our training procedure is different.

The forward pass through the network returns a tuple of two tensors – policy and value:

```
def forward(self, x: torch.ByteTensor) -> tt.Tuple[torch.Tensor, torch.Tensor]:
    xx = x / 255
    conv_out = self.conv(xx)
    return self.policy(conv_out), self.value(conv_out)
```

Now we have to discuss a large and important function, which takes the batch of environment transitions and returns three tensors: the batch of states, batch of actions taken, and batch of Q-values calculated using the formula $Q(s, a) = \sum_{i=0}^{N-1} \gamma^i r_i + \gamma^N V(s_N)$.

This Q-value will be used in two places: to calculate **mean squared error (MSE)** loss to improve the value approximation in the same way as DQN, and to calculate the advantage of the action.

```
def unpack_batch(batch: tt.List[ExperienceFirstLast], net: AtariA2C,
                device: torch.device, gamma: float, reward_steps: int):
    states = []
    actions = []
    rewards = []
    not_done_idx = []
    last_states = []
    for idx, exp in enumerate(batch):
        states.append(np.asarray(exp.state))
        actions.append(int(exp.action))
        rewards.append(exp.reward)
        if exp.last_state is not None:
            not_done_idx.append(idx)
            last_states.append(np.asarray(exp.last_state))
```

In the beginning, we just walk through our batch of transitions and copy their fields into the lists. Note that the reward value already contains the discounted reward for REWARD_STEPS, as we use the `ptan.ExperienceSourceFirstLast` class. We also need to handle episode-ending situations and remember indices of batch entries for non-terminal episodes.

In the following code, we convert the gathered state and actions into a PyTorch tensor and copy them into the **graphics processing unit (GPU)** if needed:

```
states_t = torch.FloatTensor(np.asarray(states)).to(device)
actions_t = torch.LongTensor(actions).to(device)
```

Here, the extra call to `np.asarray()` might look redundant, but without it, the performance of tensor creation degrades 5-10x. This is known as issue #13918 in PyTorch, and at the time of writing, it hasn't been solved, so one solution is to pass a single NumPy array instead of a list of arrays.

The rest of the function calculates Q-values, taking into account the terminal episodes:

```
rewards_np = np.array(rewards, dtype=np.float32)
if not_done_idx:
    last_states_t = torch.FloatTensor(
        np.asarray(last_states)).to(device)
    last_vals_t = net(last_states_t)[1]
    last_vals_np = last_vals_v.data.cpu().numpy()[:, 0]
    last_vals_np *= gamma ** reward_steps
```

```
rewards_np[not_done_idx] += last_vals_np
```

The preceding code prepares the variable with the last state in our transition chain and queries our network for $V(s)$ approximation. Then, this value is multiplied by the discount factor and added to the immediate rewards.

At the end of the function, we pack our Q-values into the tensor and return it:

```
ref_vals_t = torch.FloatTensor(rewards_np).to(device)
return states_t, actions_t, ref_vals_t
```

In the following code, you can notice a new way to create environments, the class `gym.vector.SyncVectorEnv`, which is being passed a list of lambda functions creating the underlying environments:

```
if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("--dev", default="cpu", help="Device to use, default=cpu")
    parser.add_argument("--use-async", default=False, action='store_true',
                        help="Use async vector env (A3C mode)")
    parser.add_argument("-n", "--name", required=True, help="Name of the run")
    args = parser.parse_args()
    device = torch.device(args.dev)

    env_factories = [
        lambda: ptan.common.wrappers.wrap_dqn(gym.make("PongNoFrameskip-v4"))
        for _ in range(NUM_ENVS)
    ]
    if args.use_async:
        env = gym.vector.AsyncVectorEnv(env_factories)
    else:
        env = gym.vector.SyncVectorEnv(env_factories)
    writer = SummaryWriter(comment="-pong-a2c_" + args.name)
```

The class `gym.vector.SyncVectorEnv` is provided by Gymnasium and allows wrapping several environments into one single “vectorized” environment. Underlying environments have to have identical action and observation spaces, which allows the vectorized environment to accept a vector of actions and return batches of observations and rewards. You can find more details in the Gymnasium documentation: <https://gymnasium.farama.org/api/vector/>.

Synchronized vectorized environments (the `SyncVectorEnv` class) are almost identical to the optimization we used in *Chapter 9*, in the section *Several environments*, where we passed multiple gym environments into the experience source to increase the performance of the DQN training.

But in the case of vectorized environments, a different experience source class has to be used: `VectorExperienceSourceFirstLast`, which takes into account vectorization and optimizes the agent application to the observation. From the outside, the interface of this experience source is exactly as before.

The command-line argument `-use-async` (which switches our wrapper class from `SyncVectorEnv` to `AsyncVectorEnv`) is not relevant at the moment – we will use it later, when discussing the A3C method.

Then, we create the network, agent, and experience source:

```
net = common.AtariA2C(env.single_observation_space.shape,
                      env.single_action_space.n).to(device)
print(net)

agent = ptan.agent.PolicyAgent(lambda x: net(x)[0], apply_softmax=True,
                                 device=device)
exp_source = VectorExperienceSourceFirstLast(
    env, agent, gamma=GAMMA, steps_count=REWARD_STEPS)

optimizer = optim.Adam(net.parameters(), lr=LEARNING_RATE, eps=1e-3)
```

One very important detail here is passing the `eps` parameter to the optimizer. If you’re familiar with the `Adam` algorithm, you may know that `epsilon` is a small number added to the denominator to prevent zero-division situations. Normally, this value is set to some small number, such as 10^{-8} or 10^{-10} , but in our case, these values turned out to be too small. I have no mathematically strict explanation for this, but with the default value of `epsilon`, the method does not converge at all. Very likely, the division to a small value of 10^{-8} makes the gradients too large, which turns out to be fatal for training stability.

Another detail is to use `VectorExperienceSourceFirstLast` instead of `ExperienceSourceFirstLast`. This is required because of the vectorized environment wrapping several normal Atari environments. The vectorized environment also exposes the attributes `single_observation_space` and `single_action_space`, which are the observation and action spaces of an individual environment.

In the training loop, we use two wrappers:

```
batch = []

with common.RewardTracker(writer, stop_reward=18) as tracker:
    with TBMeanTracker(writer, batch_size=10) as tb_tracker:
        for step_idx, exp in enumerate(exp_source):
            batch.append(exp)
```

```

new_rewards = exp_source.pop_total_rewards()
if new_rewards:
    if tracker.reward(new_rewards[0], step_idx):
        break

if len(batch) < BATCH_SIZE:
    continue

```

The first wrapper in this code is already familiar to you: `common.RewardTracker`, which computes the mean reward for the last 100 episodes and tells us when this mean reward exceeds the desired threshold. Another wrapper, `TBMeanTracker`, is from the PTAN library and is responsible for writing into TensorBoard the mean of the measured parameters for the last 10 steps. This is helpful, as training can take millions of steps and we don't want to write millions of points into TensorBoard, but rather write smoothed values every 10 steps.

The next code chunk is responsible for our calculation of losses, which is the core of the A2C method. First, we unpack our batch using the function we described earlier and ask our network to return the policy and values for this batch:

```

states_t, actions_t, vals_ref_t = common.unpack_batch(
    batch, net, device=device, gamma=GAMMA, reward_steps=REWARD_STEPS)
batch.clear()

optimizer.zero_grad()
logits_t, value_t = net(states_t)

```

The policy is returned in an unnormalized form, so to convert it into the probability distribution, we need to apply softmax to it. As the policy loss requires the logarithm of the probability distribution, we will use the function `log_softmax`, which is more numerically stable than calling `softmax` and then `log`.

In the value loss part, we calculate the MSE between the value returned by our network and the approximation we performed using the Bellman equation unrolled four steps forward:

```
loss_value_t = F.mse_loss(value_t.squeeze(-1), vals_ref_t)
```

Next, we calculate the policy loss to obtain the policy gradient:

```
log_prob_t = F.log_softmax(logits_t, dim=1)
adv_t = vals_ref_t - value_t.detach()
log_act_t = log_prob_t[range(BATCH_SIZE), actions_t]
log_prob_actions_t = adv_t * log_act_t
loss_policy_t = -log_prob_actions_t.mean()
```

The first two steps obtain a log of our policy and calculate the advantage of actions, which is $A(s, a) = Q(s, a) - V(s)$. The call to `value_t.detach()` is important, as we don't want to propagate the policy gradient into our value approximation head. Then, we take the log of probability for the actions taken and scale them with the advantage. Our policy gradient loss value will be equal to the negated mean of this scaled log of policy, as the policy gradient directs us toward policy improvement, but loss value is supposed to be minimized.

The last piece of our loss function is entropy loss:

```
prob_t = F.softmax(logits_t, dim=1)
entropy_loss_t = ENTROPY_BETA * (prob_t * log_prob_t).sum(dim=1).mean()
```

Entropy loss is equal to the scaled entropy of our policy, taken with the opposite sign (entropy is calculated as $H(\pi) = -\sum \pi \log \pi$).

In the following code, we calculate and extract gradients of our policy, which will be used to track the maximum gradient, its variance, and the L2 norm:

```
loss_policy_t.backward(retain_graph=True)
grads = np.concatenate([
    p.grad.data.cpu().numpy().flatten()
    for p in net.parameters() if p.grad is not None
])
```

As the final step of our training, we backpropagate the entropy loss and the value loss, clip gradients, and ask our optimizer to update the network:

```
loss_v = entropy_loss_t + loss_value_t
loss_v.backward()
nn.utils.clip_grad_norm_(net.parameters(), CLIP_GRAD)
optimizer.step()
loss_v += loss_policy_t
```

At the end of the training loop, we track all of the values that we are going to monitor in TensorBoard:

```
tb_tracker.track("advantage", adv_t, step_idx)
tb_tracker.track("values", value_t, step_idx)
tb_tracker.track("batch_rewards", vals_ref_t, step_idx)
tb_tracker.track("loss_entropy", entropy_loss_t, step_idx)
tb_tracker.track("loss_policy", loss_policy_t, step_idx)
tb_tracker.track("loss_value", loss_value_t, step_idx)
tb_tracker.track("loss_total", loss_v, step_idx)
tb_tracker.track("grad_l2", np.sqrt(np.mean(np.square(grads))), step_idx)
tb_tracker.track("grad_max", np.max(np.abs(grads)), step_idx)
tb_tracker.track("grad_var", np.var(grads), step_idx)
```

There are plenty of values that we need to monitor and we will discuss them in the next section.

Results

To start the training, run `02_pong_a2c.py` with the `-dev` (for GPU) and `-n` options (which provides a name for the run for TensorBoard):

```
Chapter12$ ./02_pong_a2c.py --dev cuda -n tt
A.L.E: Arcade Learning Environment (version 0.8.1+53f58b7)
[Powered by Stella]
AtariA2C(
    (conv): Sequential(
        (0): Conv2d(4, 32, kernel_size=(8, 8), stride=(4, 4))
        (1): ReLU()
        (2): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2))
        (3): ReLU()
        (4): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1))
        (5): ReLU()
        (6): Flatten(start_dim=1, end_dim=-1)
    )
    (policy): Sequential(
        (0): Linear(in_features=3136, out_features=512, bias=True)
        (1): ReLU()
        (2): Linear(in_features=512, out_features=6, bias=True)
    )
    (value): Sequential(
        (0): Linear(in_features=3136, out_features=512, bias=True)
        (1): ReLU()
        (2): Linear(in_features=512, out_features=1, bias=True)
    )
)
37850: done 1 games, mean reward -21.000, speed 1090.79 f/s
```

```

39250: done 2 games, mean reward -21.000, speed 1111.24 f/s
39550: done 3 games, mean reward -21.000, speed 1118.06 f/s
40000: done 4 games, mean reward -21.000, speed 1083.18 f/s
40300: done 5 games, mean reward -21.000, speed 1141.46 f/s
40750: done 6 games, mean reward -21.000, speed 1077.44 f/s
40850: done 7 games, mean reward -21.000, speed 940.09 f/s
...

```

As a word of warning, the training process is lengthy. With the original hyperparameters, it requires about 10 million frames to solve, which is about three hours on a GPU.

Later in the chapter, we'll check the asynchronous version of the A2C method, which executes the environment in a separate process (which increases both training stability and performance). But first, let's focus on our plots in TensorBoard.

The reward dynamics look much better than in the example from the previous chapter:

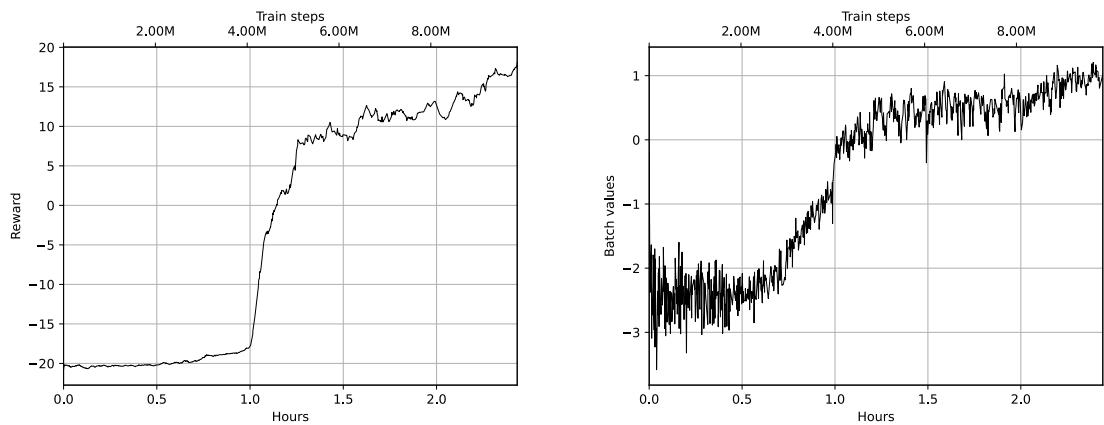


Figure 12.6: Smoothed reward (left) and mean batch values (right)

The left plot is the mean training episodes reward averaged over the 100 last episodes. The right plot, “batch value,” shows Q-values approximated using the Bellman equation and an overall positive dynamic in Q approximation. This shows that our training process is improving more or less consistently over time.

The next four charts are related to our loss and include the individual loss components and the total loss:

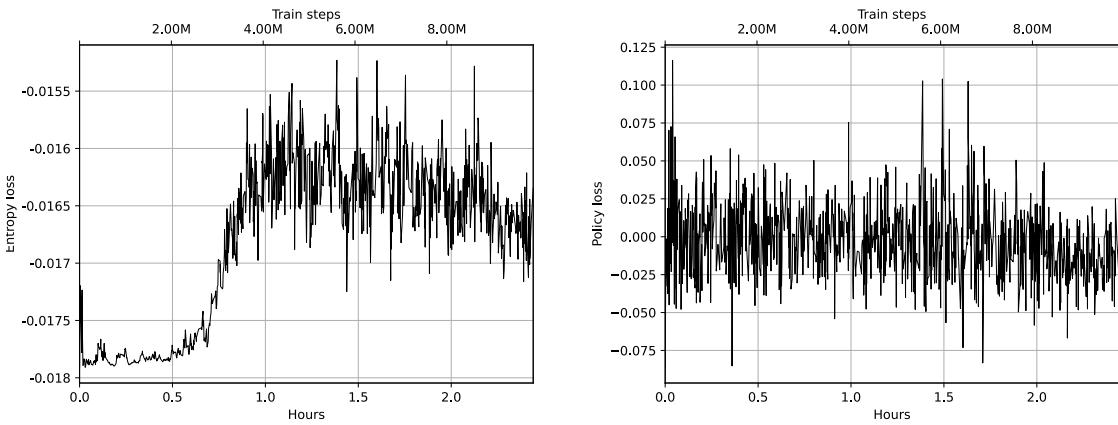


Figure 12.7: Entropy loss (left) and policy loss (right)

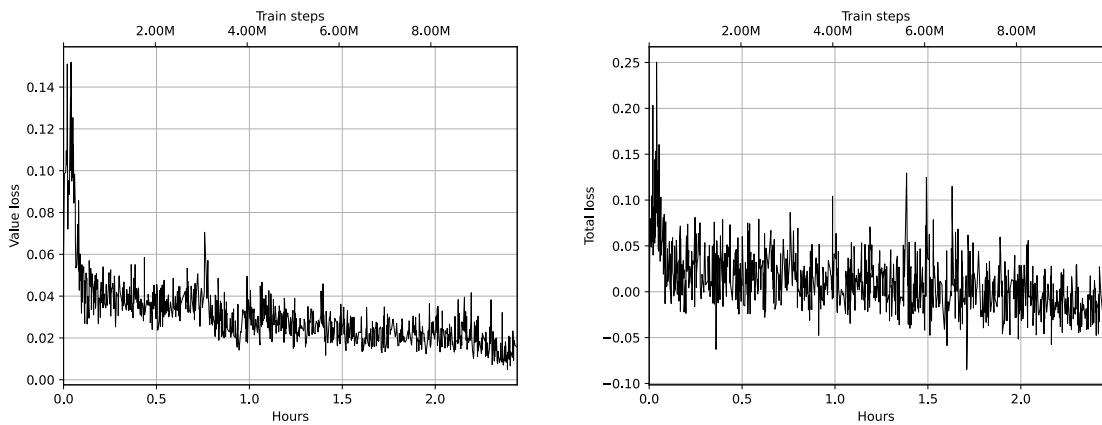


Figure 12.8: Value loss (left) and total loss (right)

Here, we must note the following:

- First, our value loss (Figure 12.8, on the left) is decreasing consistently, which shows that our $V(s)$ approximation is improving during the training.
- The second observation is that our entropy loss (Figure 12.7, on the left) is growing in the middle of the training, but it doesn't dominate in the total loss. This basically means that our agent becomes more confident in its actions as the policy becomes less uniform.

- The last thing to note here is that policy loss (Figure 12.7, on the right) is decreasing most of the time and is correlated to the total loss, which is good, as we are interested in the gradients for our policy first of all.

The last set of plots displays the advantage value and policy gradient metrics:

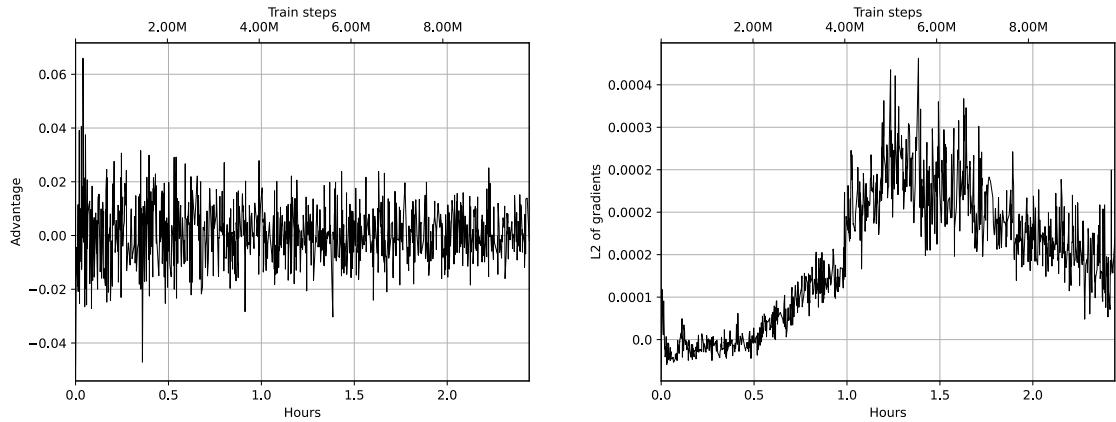


Figure 12.9: Advantage (left) and L2 of gradients (right)

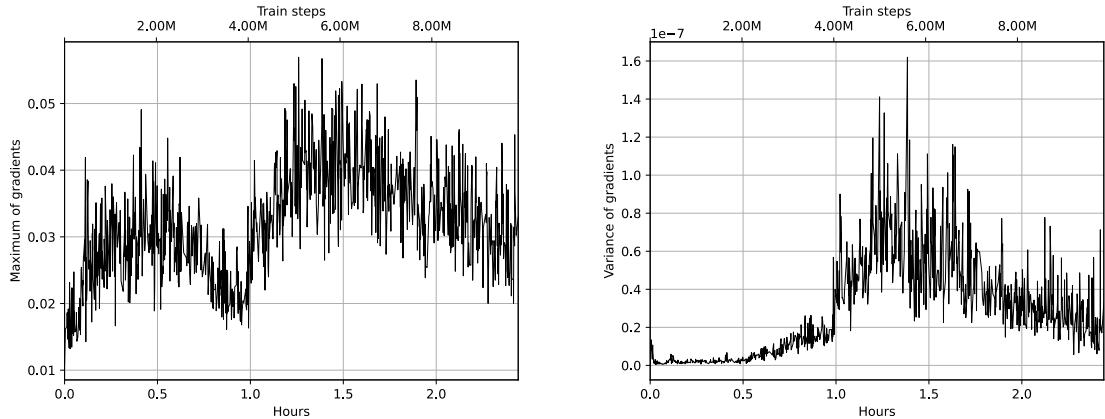


Figure 12.10: Max of gradients (left) and gradients variance (right)

The advantage is a scale of our policy gradients, and it equals $Q(s, a) - V(s)$. We expect it to oscillate around 0 (because, on average, the effect of the single action on the state's value shouldn't be large), and the chart meets our expectations. The gradient charts demonstrate that our gradients are not too small and not too large. Variance is very small at the beginning of the training (for 2 million frames), but starts to grow later, which means that our policy is changing.

Asynchronous Advantage Actor-Critic (A3C)

In this section, we will extend the A2C method. This extension adds true asynchronous environment interaction, and is called **asynchronous advantage actor-critic (A3C)**. This method is one of the most widely used by RL practitioners.

We will take a look at two approaches for adding asynchronous behavior to the basic A2C method: data-level and gradient-level parallelism. They have different resource requirements and characteristics, which makes them applicable to different situations.

Correlation and sample efficiency

One of the approaches to improving the stability of the policy gradient family of methods is using multiple environments in parallel. The reason behind this is the fundamental problem we discussed in *Chapter 6*, when we talked about the correlation between samples, which breaks the **independent and identically distributed (iid)** assumption, which is critical for **stochastic gradient descent (SGD)** optimization. The negative consequence of such correlation is very high variance in gradients, which means that our training batch contains very similar examples, all of them pushing our network in the same direction. However, this may be totally the wrong direction in the global sense, as all those examples may be from one single lucky or unlucky episode.

With our **deep Q-network (DQN)**, we solved the issue by storing a large number of previous states in the replay buffer and sampling our training batch from this buffer. If the buffer is large enough, the random sample from it will be a much better representation of the states' distribution at large. Unfortunately, this solution won't work for policy gradient methods. This is because most of them are on-policy, which means that we have to train on samples generated by our current policy, so remembering old transitions will not be possible anymore. You can try to do this, but the resulting policy gradient will be for the old policy used to generate the samples and not for your current policy that you want to update.

Researchers have focused on this issue for many years. Several ways to address it have been proposed, but the problem is still far from being solved. The most commonly used solution is gathering transitions using several parallel environments, all of them exploiting the current policy. This breaks the correlation within one single episode, as we now train on several episodes obtained from different environments. At the same time, we are still using our current policy. The one very large disadvantage of this is **sample inefficiency**, as we basically throw away all the experience that we have obtained after one single training round.

It's very simple to compare DQN with policy gradient approaches. For example, for DQN, if we use 1 million samples of a replay buffer and a training batch size of 32 samples for every new frame, every single transition will be used approximately 32 times before it is pushed from the experience replay.

For the priority replay buffer, which was discussed in *Chapter 8*, this number could be much higher, as the sample probability is not uniform. In the case of policy gradient methods, each experience obtained from the environment can be used only once, as our method requires fresh data, so the data efficiency of policy gradient methods could be an order of magnitude lower than the value-based, off-policy methods.

On the other hand, our A2C agent converged on Pong in 8 million frames, which is just eight times more than 1 million frames for basic DQN in *Chapter 6* and *Chapter 8*. So, this shows us that policy gradient methods are not completely useless; they're just different and have their own specificities that you need to take into account on method selection. If your environment is “cheap” in terms of the agent interaction (the environment is fast, has a low memory footprint, allows parallelization, and so on), policy gradient methods could be a better choice. On the other hand, if the environment is “expensive” and obtaining a large amount of experience could slow down the training process, the value-based methods could be a smarter way to go.

Adding an extra “A” to A2C

From a practical point of view, communicating with several parallel environments is simple. We already did this in *Chapter 9* and earlier in the current chapter, but it wasn't explicitly stated. In the A2C agent, we passed an array of Gym environments into the `ExperienceSource` class, which switched it into round-robin data gathering mode. This means that every time we ask for a transition from the experience source, the class uses the next environment from our array (of course, keeping the state for every environment). This simple approach is equivalent to parallel communication with environments, but with one single difference: communication is not parallel in the strict sense but performed in a serial way. However, samples from our experience source are shuffled. This idea is shown in the following figure:

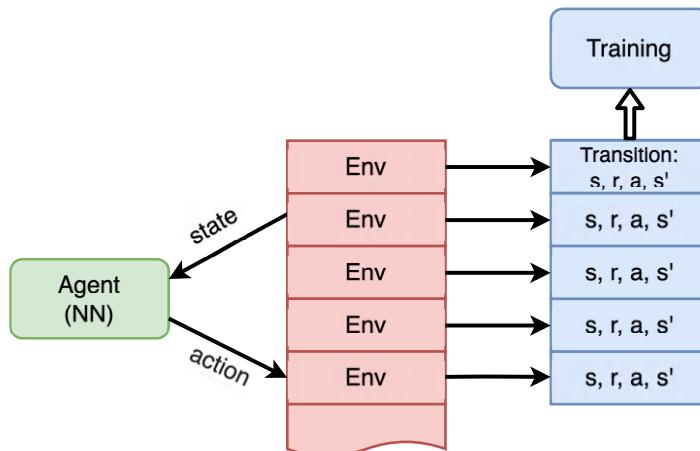


Figure 12.11: An agent training from multiple environments in parallel

This method works fine and helped us to get convergence in the A2C method, but it is still not perfect in terms of computing resource utilization, as all the processing is done sequentially. Even a modest workstation nowadays has several CPU cores, which can be used for computation, such as training and environment interaction. On the other hand, parallel programming is harder than the traditional paradigm, when you have a clear stream of execution. Luckily, Python is a very expressive and flexible language with lots of third-party libraries, which allows you to do parallel programming without much trouble. We have already seen the example of the `torch.multiprocessing` library in *Chapter 9*, where we parallelized agents' execution during the DQN training. But there are other higher-level libraries, like `ray`, which allow us to parallelize execution of the code, hiding the low-level communication details.

With regard to actor-critic parallelization, two approaches exist:

1. **Data parallelism:** We can have several processes, each of them communicating with one or more environments and providing us with transitions (s, r, a, s') . All those samples are gathered together in one single training process, which calculates losses and performs an SGD update. Then, the updated **neural network (NN)** parameters need to be broadcast to all other processes to use in future environment communications. This model is illustrated in *Figure 12.12*.
2. **Gradients parallelism:** As the goal of the training process is the calculation of gradients to update our NN, we can have several processes calculating gradients on their own training samples. Then, these gradients can be summed together to perform the SGD update in one process. Of course, updated NN weights also have to be propagated back to all workers to keep data on-policy. This is illustrated in *Figure 12.13*.

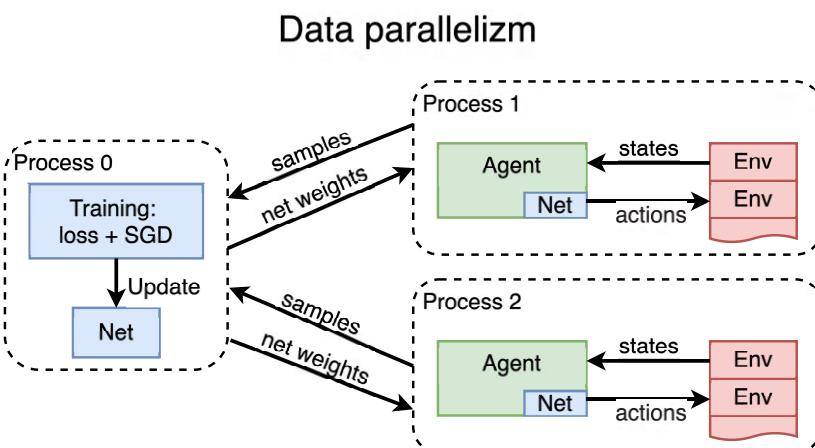


Figure 12.12: The first approach to actor-critic parallelism, based on distributed training samples being gathered

Gradients parallelism

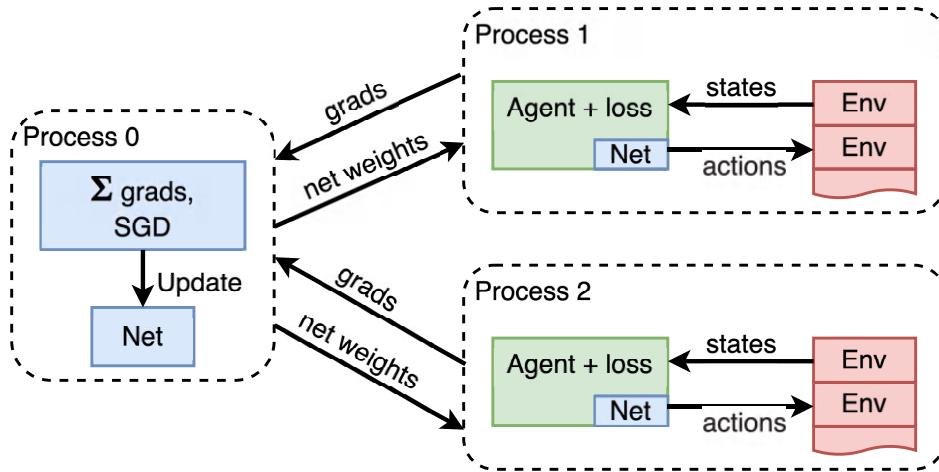


Figure 12.13: The second approach to parallelism, gathering gradients for the model

The difference between the two methods might not look very significant from the diagrams, but you need to be aware of the computation cost. The heaviest operation in A2C optimization is the training process, which consists of loss calculation from data samples (forward pass) and the calculation of gradients with respect to this loss. The SGD optimization step is quite lightweight – basically, just adding the scaled gradients to the NN's weights. By moving the computation of loss and gradients in the second approach (gradient parallelism) from the central process, we eliminated the major potential bottleneck and made the whole process significantly more scalable.

In practice, the choice of the method mainly depends on your resources and your goals. If you have one single optimization problem and lots of distributed computation resources, such as a couple of dozen GPUs spread over several machines in the networks, then gradients parallelism will be the best approach to speed up your training.

However, in the case of one single GPU, both methods will provide a similar performance, but the first approach is generally simpler to implement, as you don't need to deal with low-level gradient values. In this chapter, we will compare both methods on our favorite Pong game to see the difference between the approaches and look at PyTorch's multiprocessing capabilities.

A3C with data parallelism

The first version of A3C parallelization that we will check (which was outlined in *Figure 12.12*) has both one main process that carries out training and several child processes communicating with environments and gathering experience to train on.

In fact, we already implemented this version in *Chapter 9* when we ran several agents in subprocesses when we trained the DQN model (then we got a speed-up of 27% in terms of FPS). In this section, I'm not going to reimplement the same approach with the A3C method, but rather want to illustrate the “power of libraries.”

We already briefly mentioned the class `gym.vector.SyncVectorEnv` from Gymnasium (it exists only in the Farma fork, not in the original OpenAI Gym) and the PTAN experience source, which supports “vectorized” environments: `VectorExperienceSourceFirstLast`. The class `SyncVectorEnv` handles wrapped environments sequentially, but there is a drop-in replacement class, `AsyncVectorEnv`, which uses `mp.multiprocessing` for subenvironments. So, to get the data-parallel version of the A2C method, we just need to replace `SyncVectorEnv` with `AsyncVectorEnv` and we’re done.

The code in `Chapter12/02_pong_a2c.py` already supports this replacement, which is done by passing the `-use-async` command-line option.

Results

The asynchronous version with 50 environments shows a performance of 2000 FPS, which is a 2x improvement over the sequential version. The following charts compare the performance and reward dynamics of these two versions:

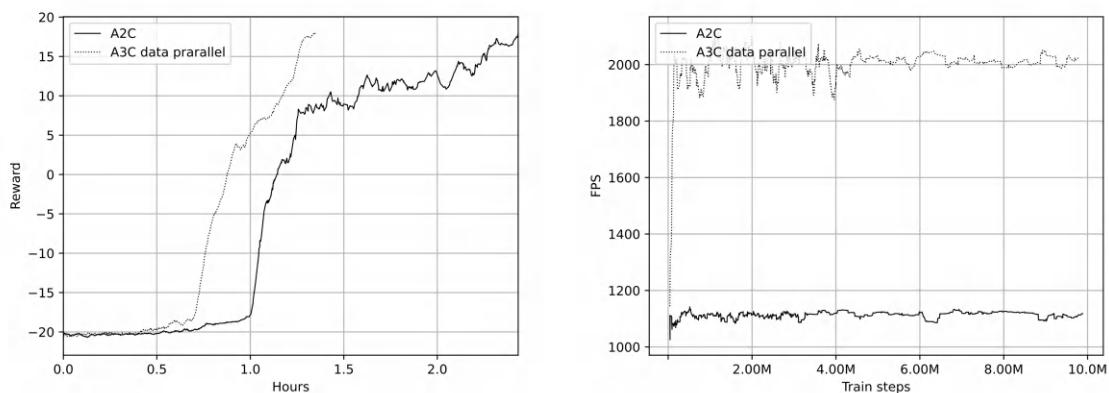


Figure 12.14: Comparison of A2C and A3C in terms of reward (left) and speed (right)

A3C with gradient parallelism

The next approach that we will consider to parallelize A2C implementation will have several child processes, but instead of feeding training data to the central training loop, they will calculate the gradients using their local training data, and send those gradients to the central master process. This process is responsible for combining those gradients (which is basically just summing them) and performing an SGD update on the shared network.

The difference might look minor, but this approach is much more scalable, especially if you have several powerful nodes with multiple GPUs connected to the network. In this case, the central process in the data-parallel model quickly becomes a bottleneck, as the loss calculation and backpropagation are computationally demanding. Gradient parallelization allows for the spreading of the load on several GPUs, performing only a relatively simple operation of gradient combination in a central place.

Implementation

The complete example is in the `Chapter12/03_a3c_grad.py` file, and it uses the same `Chapter12/lib/common.py` module that we've already seen.

As usual, we first define the hyperparameters:

```
GAMMA = 0.99
LEARNING_RATE = 0.001
ENTROPY_BETA = 0.01
REWARD_STEPS = 4
CLIP_GRAD = 0.1

PROCESSES_COUNT = 4
NUM_ENVS = 8
GRAD_BATCH = 64
TRAIN_BATCH = 2

ENV_NAME = "PongNoFrameskip-v4"
NAME = 'pong'
REWARD_BOUND = 18
```

These are mostly the same as in the previous example, except `BATCH_SIZE` is replaced by two parameters: `GRAD_BATCH` and `TRAIN_BATCH`. The value of `GRAD_BATCH` defines the size of the batch used by every child process to compute the loss and get the value of the gradients. The second parameter, `TRAIN_BATCH`, specifies how many gradient batches from the child processes will be combined on every SGD iteration. Every entry produced by the child process has the same shape as our network parameters, and we sum up `TRAIN_BATCH` values of them together.

So, for every optimization step, we use the `TRAIN_BATCH * GRAD_BATCH` training samples. As the loss calculation and backpropagation are quite heavy operations, we use a large `GRAD_BATCH` to make them more efficient.

Due to this large batch, we should keep `TRAIN_BATCH` relatively low to keep our network update on policy.

Now we have two functions – `make_env()`, which is used to create a wrapped Pong environment, and `grads_func()`, which is much more complicated and implements most of the training logic we normally do in the training loop. As a compensation, the training loop in the main process becomes almost trivial:

```
def make_env() -> gym.Env:
    return ptan.common.wrappers.wrap_dqn(gym.make("PongNoFrameskip-v4"))

def grads_func(proc_name: str, net: common.AtariA2C, device: torch.device,
              train_queue: mp.Queue):
    env_factories = [make_env for _ in range(NUM_ENVS)]
    env = gym.vector.SyncVectorEnv(env_factories)

    agent = ptan.agent.PolicyAgent(lambda x: net(x)[0], device=device,
                                    apply_softmax=True)
    exp_source = VectorExperienceSourceFirstLast(
        env, agent, gamma=GAMMA, steps_count=REWARD_STEPS)

    batch = []
    frame_idx = 0
    writer = SummaryWriter(comment=proc_name)
```

On the creation of the child process, we pass several arguments to the `grads_func()` function:

- The name of the process, which is used to create the TensorBoard writer. In this example, every child process writes its own TensorBoard dataset.
- The shared NN.
- A `torch.device` instance, specifying the computation device.
- The queue used to deliver the calculated gradients to the central process.

Our child process function looks very similar to the main training loop in the data-parallel version, which is not surprising, as the responsibilities of our child process increased. However, instead of asking the optimizer to update the network, we gather gradients and send them to the queue.

The rest of the code is almost the same:

```
with common.RewardTracker(writer, REWARD_BOUND) as tracker:
    with TBMeanTracker(writer, 100) as tb_tracker:
        for exp in exp_source:
            frame_idx += 1
            new_rewards = exp_source.pop_total_rewards()
            if new_rewards and tracker.reward(new_rewards[0], frame_idx):
                break

            batch.append(exp)
            if len(batch) < GRAD_BATCH:
                continue
```

Up to this point, we've gathered the batch with transitions and handled the end-of-episode rewards.

In the next part of the function, we calculate the combined loss from the training data and perform backpropagation of the loss:

```
data = common.unpack_batch(batch, net, device=device, gamma=GAMMA,
                         reward_steps=REWARD_STEPS)
states_v, actions_t, vals_ref_v = data
batch.clear()

net.zero_grad()
logits_v, value_v = net(states_v)
loss_value_v = F.mse_loss(value_v.squeeze(-1), vals_ref_v)

log_prob_v = F.log_softmax(logits_v, dim=1)
adv_v = vals_ref_v - value_v.detach()
log_p_a = log_prob_v[range(GRAD_BATCH), actions_t]
log_prob_actions_v = adv_v * log_p_a
loss_policy_v = -log_prob_actions_v.mean()

prob_v = F.softmax(logits_v, dim=1)
ent = (prob_v * log_prob_v).sum(dim=1).mean()
entropy_loss_v = ENTROPY_BETA * ent

loss_v = entropy_loss_v + loss_value_v + loss_policy_v
loss_v.backward()
```

In the following code, we send our intermediate values that we're going to monitor during the training to TensorBoard:

```
tb_tracker.track("advantage", adv_v, frame_idx)
tb_tracker.track("values", value_v, frame_idx)
tb_tracker.track("batch_rewards", vals_ref_v, frame_idx)
tb_tracker.track("loss_entropy", entropy_loss_v, frame_idx)
tb_tracker.track("loss_policy", loss_policy_v, frame_idx)
tb_tracker.track("loss_value", loss_value_v, frame_idx)
tb_tracker.track("loss_total", loss_v, frame_idx)
```

At the end of the loop, we need to clip the gradients and extract them from the network's parameters into a separate buffer (to prevent them from being corrupted by the next iteration of the loop). Here, we effectively store gradients in the `tensor.grad` field for every network parameter. This could be done without bothering with synchronization with other workers, as our network's parameters are shared, but the gradients are locally allocated by every process:

```
nn_utils.clip_grad_norm_(
    net.parameters(), CLIP_GRAD)
grads = [
    param.grad.data.cpu().numpy() if param.grad is not None else None
    for param in net.parameters()
]
train_queue.put(grads)

train_queue.put(None)
```

The last line in `grads_func` puts `None` into the queue, signaling that this child process has reached the *game solved* state and training should be stopped.

The main process starts with the creation of the network and sharing of its weights:

```
if __name__ == "__main__":
    mp.set_start_method('spawn')
    os.environ['OMP_NUM_THREADS'] = "1"
    parser = argparse.ArgumentParser()
    parser.add_argument("--dev", default="cpu", help="Device to use, default=cpu")
    parser.add_argument("-n", "--name", required=True, help="Name of the run")
    args = parser.parse_args()
    device = torch.device(args.dev)

    env = make_env()
```

```
net = common.AtariA2C(env.observation_space.shape, env.action_space.n).to(device)
net.share_memory()
```

Here, as in the previous section, we need to set a start method for `torch.multiprocessing` and limit the number of threads started by OpenMP. This is done by setting the environment variable `OMP_NUM_THREADS`, which instructs the OpenMP library about the number of threads it can start. OpenMP (<https://www.openmp.org/>) is heavily used by the Gym and OpenCV libraries to provide a speed-up on multicore systems, which is a good thing most of the time. By default, the process that uses OpenMP starts a thread for every core in the system. But in our case, the effect from OpenMP is the opposite: as we're implementing our own parallelism, by launching several processes, extra threads overload the cores with frequent context switches, which negatively impacts performance. To avoid this, we explicitly limit the amount of threads to one thread. If you want, you can experiment yourself with this parameter. On my system, I experienced a 3-4x performance drop without this environment variable assignment.

Then, we create the communication queue and spawn the required count of child processes:

```
optimizer = optim.Adam(net.parameters(), lr=LEARNING_RATE, eps=1e-3)

train_queue = mp.Queue(maxsize=PROCESSES_COUNT)
data_proc_list = []
for proc_idx in range(PROCESSES_COUNT):
    proc_name = f"-a3c-grad_pong_{args.name}#{proc_idx}"
    p_args = (proc_name, net, device, train_queue)
    data_proc = mp.Process(target=grads_func, args=p_args)
    data_proc.start()
    data_proc_list.append(data_proc)
```

Now we can get to the training loop:

```
batch = []
step_idx = 0
grad_buffer = None

try:
    while True:
        train_entry = train_queue.get()
        if train_entry is None:
            break
```

The major difference from the data-parallel version of A3C lies in the training loop, which is much simpler here, as child processes have done all the heavy calculations for us. In the beginning of the loop, we handle the situation when one of the processes has reached the required mean reward (when this happens, we have `None` in the queue). In this case, we just exit the loop to stop the training.

We sum gradients together for all the parameters in our network:

```
step_idx += 1

if grad_buffer is None:
    grad_buffer = train_entry
else:
    for tgt_grad, grad in zip(grad_buffer, train_entry):
        tgt_grad += grad
```

When we have accumulated enough gradient pieces, we convert the sum of the gradients into the PyTorch `FloatTensor` and assign them to the `grad` field of the network parameters. To average the gradients from different children, we call the optimizer's `step()` function for every `TRAIN_BATCH` gradient obtained. For intermediate steps, we just sum the corresponding gradients together:

```
if step_idx % TRAIN_BATCH == 0:
    for param, grad in zip(net.parameters(), grad_buffer):
        param.grad = torch.FloatTensor(grad).to(device)

    nn.utils.clip_grad_norm_(net.parameters(), CLIP_GRAD)
    optimizer.step()
    grad_buffer = None
```

After that, all we need to do is call the optimizer's `step()` method to update the network parameters using the accumulated gradients.

On the exit from the training loop, we stop all child processes to make sure that we terminated them, even if `Ctrl + C` was pressed to stop the optimization:

```
finally:
    for p in data_proc_list:
        p.terminate()
        p.join()
```

This step is needed to prevent zombie processes from occupying GPU resources.

Results

This example can be started the same way as the previous example, and after a while, it should start displaying the speed and mean reward. However, you need to be aware that displayed information is local for every child process, which means that speed, the count of games completed, and the number of frames need to be multiplied by the number of processes. My benchmarks have shown speed to be around 500-600 FPS for every child, which gives 2,000-2,400 FPS in total.

Convergence dynamics are also very similar to the previous version. The total number of observations is about 8...10 million, which requires about 1.5 hours to complete. The reward chart on the left shows individual processes, but the speed chart on the right shows the sum of all processes. As you can see, gradient parallelism gives slightly higher performance than data parallelism:

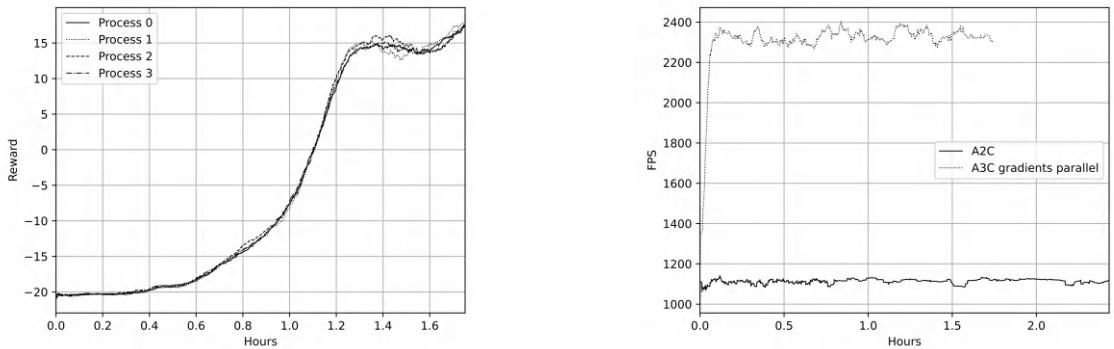


Figure 12.15: Comparison of A2C and A3C in terms of reward (left) and speed (right)

Summary

In this chapter, you learned about one of the most widely used methods in deep RL: A2C, which wisely combines the policy gradient update with the value of the state approximation. We analyzed the effect of the baseline on the statistics and convergence of gradients. Then, we checked the extension of the baseline idea: A2C, where a separate network head provides us with the baseline for the current state. In addition, we discussed why it is important for policy gradient methods to gather training data from multiple environments, due to their on-policy nature. We also implemented two different approaches to A3C, in order to parallelize and stabilize the training process. Parallelization will come up once again in this book, when we discuss black-box methods (*Chapter 17*).

In the next two chapters, we will take a look at practical problems that can be solved using policy gradient methods, which will wrap up the policy gradient methods part of the book.

Join our community on Discord

Read this book alongside other users, Deep Learning experts, and the author himself.

Ask questions, provide solutions to other readers, chat with the author via Ask Me Anything sessions, and much more.

Scan the QR code or visit the link to join the community.

<https://packt.link/r1>



13

The TextWorld Environment

In this chapter, we will now use RL to solve text-based interactive fiction games, using the environment published by Microsoft Research called TextWorld. This will provide a good illustration of how RL can be applied to complicated environments with a rich observation space. In addition, we'll touch on deep NLP methods a bit and play with LLMs.

In this chapter, we will:

- Cover a brief historical overview of interactive fiction
- Study the TextWorld environment
- Implement the simple baseline **deep Q-network (DQN)** method, and then try to improve it by adding several tweaks to the observation
- Use pretrained transformers from the Hugging Face Hub to implement sentence embedding for our agent
- Use OpenAI ChatGPT to check the power of modern **Large Language Models (LLMs)** on interactive fiction games

Interactive fiction

As you have already seen, computer games are not only entertaining for humans but also provide challenging problems for RL researchers due to the complicated observations and action spaces, long sequences of decisions to be made during the gameplay, and natural reward systems.

Arcade games like those on the Atari 2600 are just one of many genres that the gaming industry has. Let's take a step back and take a quick look at the historical perspective. The Atari 2600 platform peaked in popularity during the late 70s and early 80s. Then followed the era of Z80 and clones, which evolved into the period of the PC-compatible platforms and consoles we have now. Over time, computer games continually become more complex, colorful, and detailed in terms of graphics, which inevitably increased hardware requirements. This trend makes it harder for RL researchers and practitioners to apply RL methods to the more recent games; for example, almost everybody can train an RL agent to solve an Atari game, but for StarCraft II, DeepMind had to burn electricity for weeks, leveraging clusters of **graphics processing unit (GPU)** machines. Of course, this activity is needed for future research, as it allows us to check ideas and optimize methods, but the complexity of StarCraft II and Dota, for example, makes them prohibitively expensive for most people.

There are several ways of solving this problem:

- The first one is to take games that are “in the middle” of the complexities of Atari and StarCraft. Luckily, there are literally thousands of games from the Z80, NES, Sega, and C64 platforms.
- Another way is to take a challenging game but make a simplification to the environment. There are several Doom environments (available in Gym), for example, that use the game engine as a platform, but the goal is much simpler than in the original game, like navigating the corridor, gathering weapons, or shooting enemies. Those microgames are also available on StarCraft II.
- The third, and completely different, approach is to take some game that may not be very complex in terms of observation but requires long-term planning, complex exploration of the state space, and has challenging interactions between objects. An example of this family is the famous Montezuma’s Revenge from the Atari suite, which is still challenging even for modern RL methods.

The last approach is quite appealing, due to the accessibility of resources, combined with still having a complexity that reaches the edge of RL methods' limits. Another example of this is text-based games, which are also known as interactive fiction. This genre is almost dead now, being made obsolete by modern games and hardware progress, but at the time of Atari and Z80, interactive fiction was provided concurrently with traditional games. Instead of using rich graphics to show the game state (which was tricky for hardware from the 70s), these games relied on the players' minds and imagination.

The gaming process was communicated via text when the description of the current game state was given to the player. An example is, *You are standing at the end of a road before a small brick building. Around you is a forest. A small stream flows out of the building and down a gully.*

As you can see in *Figure 13.1*, this is the very beginning of the Adventure game from 1976, which was the first game of this kind. Actions in the game were given in the form of free-text commands, which normally had a simple structure and a limited set of words, for example, “verb + noun.”

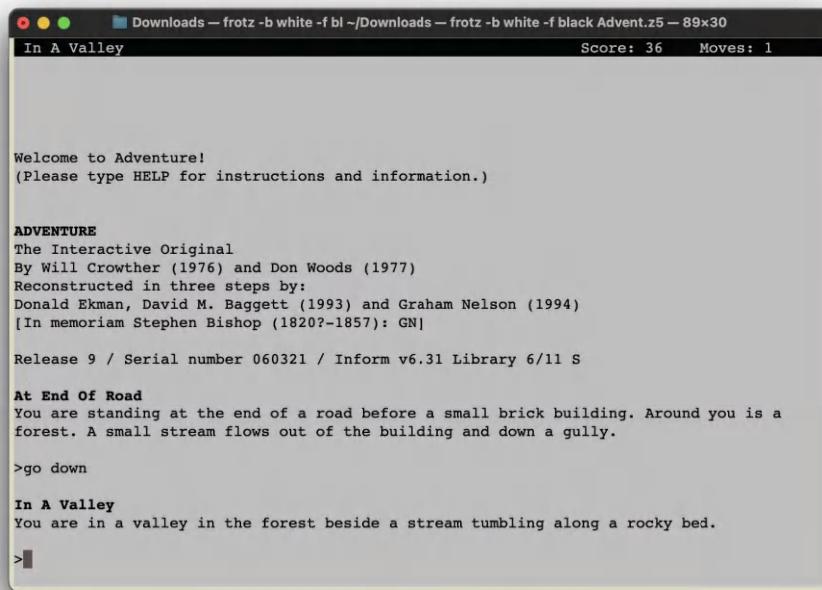


Figure 13.1: An example of the interactive fiction game process

Despite the simplistic descriptions, in the 80s and early 90s, hundreds of large and small games were developed by individual developers and commercial studios. Those games sometimes required many hours of gameplay, contained thousands of locations, and had a lot of objects to interact with.

For example, *Figure 13.2* shows part of the Zork I game map published by Infocom in 1980.

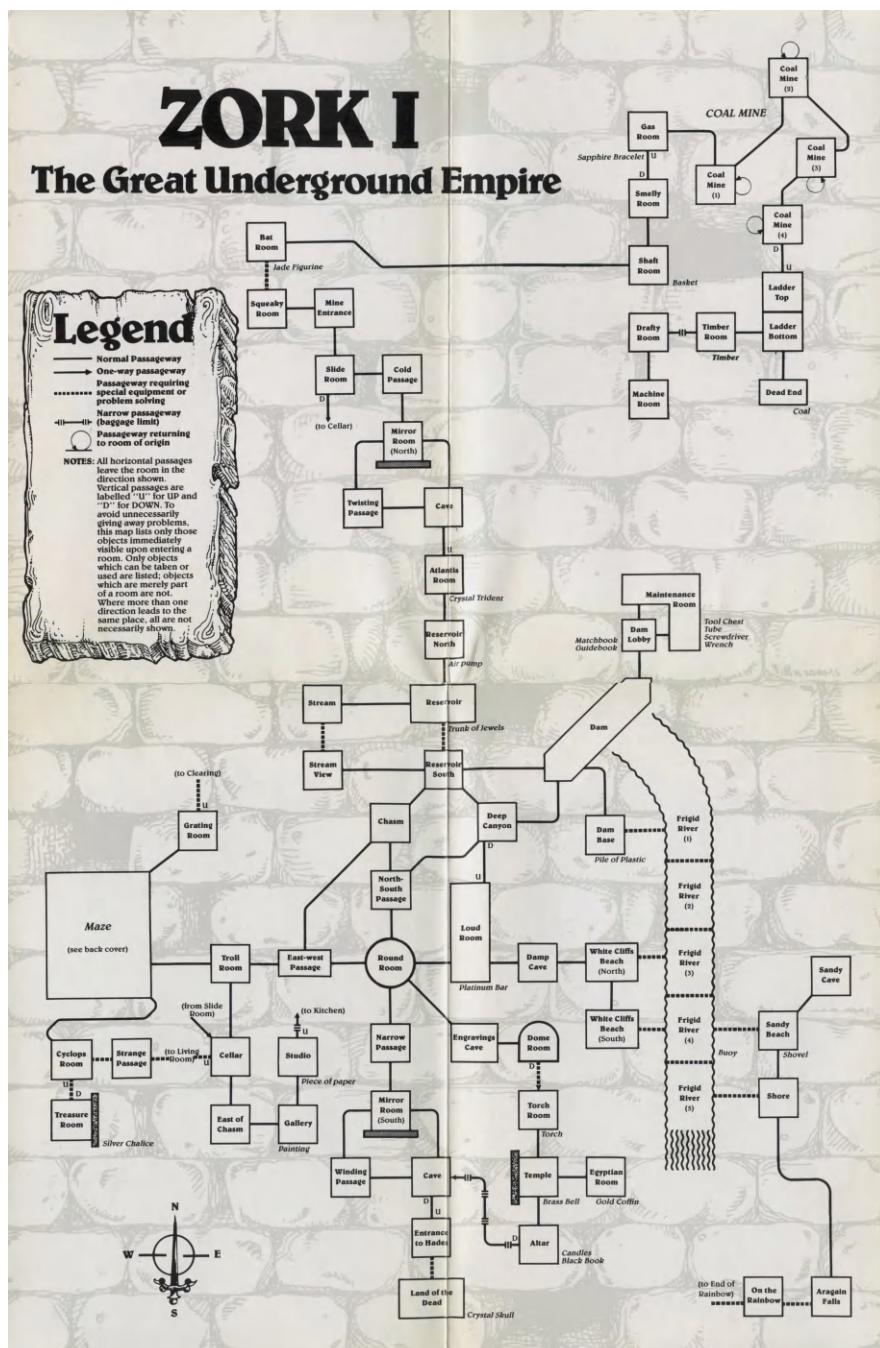


Figure 13.2: The underground portion of the Zork I map (for better visualization, refer to <https://packt.link/gbp/9781835882702>)

As you can imagine, the challenges of such games could be increased almost infinitely, as complex interactions between objects, exploration of game states, communication with other characters, and other real-life scenarios could be included. There are many such games available on the Interactive Fiction Archive website: <http://ifarchive.org>.

In June 2018, Microsoft Research released an open source project that aimed to provide researchers and RL enthusiasts with a simple way to experiment with text-based games using familiar tools. Their project, called TextWorld, is available on GitHub (<https://github.com/microsoft/TextWorld>) and provides the following functionality:

- A Gym environment for text-based games. It supports games in two formats: Z-machine bytecode (versions 1–8 are supported) and Glulx games.
- A game generator that allows you to produce randomly generated quests with pre-defined complexity like the number of objects, the description, and the quest length.
- The capability to tune (for generated games) the complexity of the environment by peeking at the game state. For example, intermediate rewards could be enabled, which will give a positive reward to the agent every time it makes a step in the right direction. Several such factors will be described in the next section.



As we progress with this chapter, we will experiment with several games to explore the environment's capabilities and implement several versions of training code to solve the generated games. You need to generate them by using the provided script: `Chapter13/game/make_games.sh`. It generates 21 games of length 5, using different seed values to ensure variability between the games. The complexity of the games will not be very high, but you can use them as a basis for your own experiments and idea validation.

The environment

At the time of writing, the TextWorld environment supports only Linux and macOS platforms (for Windows, you can use a Docker container) and internally relies on the Inform 7 system (<https://inform7.com>). There are two web pages for the project: one is the Microsoft Research web page (<https://www.microsoft.com/en-us/research/project/textworld/>), which contains general information about the environment, and the another is on GitHub (<https://github.com/microsoft/TextWorld>) and describes installation and usage. Let's start with installation.

Installation

Installation can be done with simple `pip install textworld==1.6.1`. All the examples in this chapter were tested with the latest 1.6.1 release of the package.

Once installed, the package can be imported in Python code, and it also provides two command-line utilities for game generation and gameplay: `tw-make` and `tw-play`. They are not needed if you have ambitious plans to solve full-featured interactive fiction games from <http://ifarchive.org>, but in our case, we will start with artificially generated quests for simplicity.

Game generation

The `tw-make` utility allows you to generate games with the following characteristics:

- **Game scenario:** For example, you can choose a classic quest with the aim of using objects and following some sequence of actions, or a “coin collection” scenario, when the player needs to navigate the scenes and find coins
- **Game theme:** You can set up the interior of the game, but at the moment, only the “house” and “basic” themes exist
- **Object properties:** You can include adjectives with objects; for instance, it might be the “green key” that opens the box, not just the “key”
- **The number of parallel quests that the game can have:** By default, there is only one sequence of actions to be found, but you can change this and allow the game to have subgoals and alternative paths
- **The length of the quest:** You can define how many steps the player needs to take before reaching the end or solution of the game
- **Random seeds:** You can use these to generate reproducible games

The resulting game generated could be in Glulx or Z-machine format, which are standard portable virtual machine instructions that are widely used for normal games and supported by several interactive fiction interpreters, so you can play the generated games in the same way as normal interactive fiction games.

Let’s generate some games and check what they bring us:

```
$ tw-make tw-coin_collector --output t1 --seed 10 --level 5 --format ulx
Global seed: 10
Game generated: t1.ulx
```

The command generates three files: `t1.ulx`, `t1.ni`, and `t1.json`. The first one contains bytecode to be loaded into the interpreter, and the others are extended data that could be used by the environment to provide extra information during the gameplay.

To play the game in interactive mode, you can use any interactive fiction interpreter supporting the Glulx format, or use the provided utility `tw-play`, which might not be the most convenient way to play interactive fiction games but will enable you to check the result:

```
$ tw-play t1.ulx
Using TWInform7.

...
Hey, thanks for coming over to the TextWorld today, there
is something I need you to do for me. First thing I need you
to do is to try to venture east. Then, venture south. After
that, try to go to the south. Once you succeed at that, try
to go west. If you can finish that, pick-up the coin from
the floor of the chamber. Once that's all handled, you can stop!

-= Spare Room =
You are in a spare room. An usual one.

You don't like doors? Why not try going east, that entranceway
is unblocked.

> _
```

Observation and action spaces

Generating and playing a game might be fun, but the core value of TextWorld is in its ability to provide an RL interface for generated or existing games. Let's check what we can do with the game we just generated in the previous section:

```
>>> from textworld import gym
>>> from textworld.gym import register_game
>>> env_id = register_game("t1.ulx")
>>> env_id
'tw-v0'
>>> env = gym.make(env_id)
>>> env
<textworld.gym.envs.textworld.TextworldGymEnv object at 0x102f77350>
>>> r = env.reset()
>>> print(r[1])
{}
>>> print(r[0][1205:])
$$
```

```
Hey, thanks for coming over to the TextWorld today, there is something I need you to do
for me. First thing I need you to do is to try to venture east. Then, venture south.
After that, try to go to the south. Once you succeed at that, try to go west. If you can
finish that, pick-up the coin from the floor of the chamber. Once that's all handled, you
can stop!
```

```
-- Spare Room --
```

```
You are in a spare room. An usual one.
```

```
You don't like doors? Why not try going east, that entranceway is unblocked.
```

Here, we registered the generated game and created the environment. You might notice that we are not using the Gymnasium `make()` function, but instead, we use a function from the `textworld` module, which has the same name. This is not a mistake. In fact, the latest TextWorld release (at the time of writing) removed dependency on Gym API packages and provides their own environment class that looks very similar to the `Env` class (but not exactly the same).

I believe this removal is temporary and part of the transition from OpenAI Gym to Farama Gymnasium. But at the moment, there are several aspects we have to take into account when using TextWorld:

- You have to create games using the `textworld.gym.make()` function, not `gym.make()`.
- Created environments don't have observation and action space specifications. By default, both observation and actions are strings.
- The function `step()` in the environment doesn't return the `is_truncated` flag, just observation, reward, flag `is_done`, and a dictionary with extra information. Because of that, you cannot apply Gymnasium wrappers to this environment — small “adapter” wrapper has to be created.

In previous versions of TextWorld, they provided tokenization functions, but they were removed, so we'll need to deal with text preprocessing ourselves.

Let's now take a look at additional information the game engine provides us.

Extra game information

Before we start planning our first training code, we need to discuss one additional functionality of TextWorld that we will use. As you might guess, even a simple problem might be too challenging for us:

- Observations are text sequences of up to 200 tokens from the vocabulary of size 1,250. Actions could be up to eight tokens long. Generated games have five actions to be executed in the correct order. So, our chance of randomly finding the proper sequence of $8 \times 5 = 40$ tokens is something around $\frac{1}{1250^{40}} \approx \frac{1}{10^{123}}$.

This is not very promising, even with the fastest GPUs. Of course, we have start- and end-sequence tokens, which we can take into account to increase our chances; still, the probability of finding the correct sequence of actions with random exploration is tiny.

- Another challenge is the **partially observable Markov decision process (POMDP)** nature of the environment, which comes from the fact that our inventory in the game is usually not shown. It is a normal practice in interactive fiction games to display the objects your character possesses only after some explicit command, like `inventory`. But our agent has no idea about the previous state. So, from its point of view, the situation after the command `take apple` is exactly the same as before (with the difference that the apple is no longer mentioned in the scene description). We can deal with that by stacking states, as we did in Atari games, but we need to do it explicitly, and the amount of information the agent needs to process will increase significantly.

With all this being said, we should make some simplifications in the environment. Luckily, TextWorld provides us with convenient means for such workarounds. During the game registration, we can pass extra flags to enrich the observation space with extra pieces of more structured information. Here is the list of internals that we can peek into:

- A separate description of the current room, as it will be given by the `look` command
- The current inventory
- The name of the current location
- The facts of the current world state
- The last action and the last command performed
- The list of admissible commands in the current state
- The sequence of actions to execute to win the game

In addition, besides extra structured observations provided on every step, we can ask TextWorld to give us intermediate rewards every time we move in the right direction in the quest. As you might guess, this is extremely helpful for speeding up the convergence.

The most useful features in the additional information we can add are admissible commands, which enormously decrease our action space from 1250^{40} to just a dozen, and intermediate rewards, which guide the training in the right direction. To enable this extra information, we need to pass an optional argument to the `register_game()` method:

```
>>> from textworld import gym, EnvInfos
>>> from textworld.gym import register_game
>>> env_id = register_game("t1.ulx", request_infos=EnvInfos(inventory=True,
intermediate_reward=True, admissible_commands=True, description=True))
```

```
>>> env = gym.make(env_id)
>>> r = env.reset()
>>> r[1]
{'description': "-- Spare Room --\nYou are in a spare room. An usual one.\n\n\n\nYou don't like doors? Why not try going east, that entranceway is unblocked.", 'admissible_commands': ['go east', 'inventory', 'look'], 'inventory': 'You are carrying nothing.', 'intermediate_reward': 0}
```

As you can see, the environment now provides us with extra information in the dictionary that was empty before. In this state, only three commands make sense (`go east`, `inventory`, and `look`). Let's try the first one:

```
>>> r = env.step('go east')
>>> r[1:]
(0, False, {'description': "-- Attic --\nYou make a grand eccentric entrance into an attic.\n\n\n\nYou need an unblocked exit? You should try going south. You don't like doors? Why not try going west, that entranceway is unblocked.", 'admissible_commands': ['go south', 'go west', 'inventory', 'look'], 'inventory': 'You are carrying nothing.', 'intermediate_reward': 1})
```

The command was accepted, and we were given an intermediate reward of 1. Okay, that's great. Now we have everything needed to implement our first baseline DQN agent to solve TextWorld problems! But before that, we need to dive a bit into the **natural language processing (NLP)** world.

The deep NLP basics

In this short section, I'm going to walk you through deep NLP building blocks and standard approaches. This domain is evolving at enormous speed, especially now, as ChatGPT and LLMs have set a new standards in chatbots and text processing.

The material in this section just scratches the surface and covers the most common and standard building blocks. Some of them, like RNNs and LSTMs, might even look outdated — I still believe this is fine, as being aware of historical perspective is important. For simple tasks, you might consider using simple tools depending on what is most suitable for the task at hand, even if they are not hyped anymore.

Recurrent Neural Networks (RNNs)

NLP has its own specifics that make it different from computer vision or other domains. One such feature is processing variable-length objects. At various levels, NLP deals with objects that could have different lengths; for example, a word in a language could contain several characters. Sentences are formed from variable-length word sequences. Paragraphs or documents consist of varying numbers of sentences. Such variability is not NLP-specific and can arise in different domains, like in signal processing or video processing. Even standard computer vision problems could be seen as a sequence of some objects, like an image captioning problem when a **neural network (NN)** can focus on various amounts of regions of the same image to better describe the image.

RNNs provide one of the standard building blocks to deal with this variability. An RNN is a network with fixed input and output that is applied to a sequence of objects and can pass information along this sequence. This information is called the hidden state, and it is normally just a vector of numbers of some size.

In the following diagram, we have an RNN with one input, which is a fixed-sized vector of numbers; the output is another vector. What makes it different from a standard feed-forward or convolutional NN is two extra gates: one input and one output. The extra input feeds the hidden state from the previous item into the RNN unit, and the extra output provides a transformed hidden state to the next sequence:

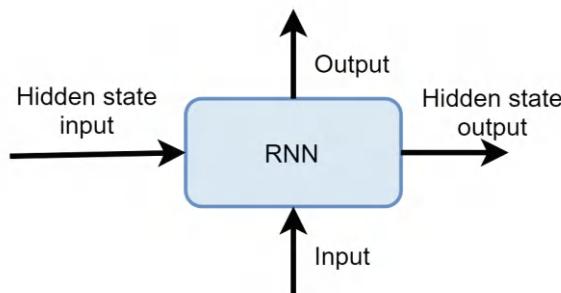


Figure 13.3: The structure of an RNN building block

As an RNN has two inputs, it can be applied to input sequences of any length, just by passing the hidden state produced by the previous entry to the next one. In *Figure 13.4*, an RNN is applied to the sentence *this is a cat*, producing the output for every word in the sequence.

During the application, we have the same RNN applied to every input item, but by having the hidden state, it can now pass information along the sequence:

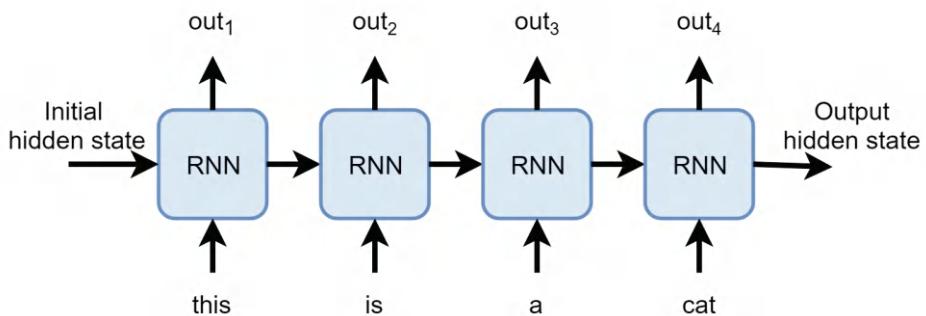


Figure 13.4: How an RNN is applied to a sentence

This is similar to convolutional NNs, when we have the same set of filters applied to various locations of the image, but the difference is that a convolutional NN can't pass the hidden state.

Despite the simplicity of this model, it adds an extra degree of freedom to the standard feed-forward NN model. The feed-forward NNs are determined by their input and always produce the same output for some fixed input (during the inference, of course, and not during the training). An RNN's output depends not only on the input but also on the hidden state, which could be changed by the NN itself. So, the NN could pass some information from the beginning of the sequence to the end and produce a different output for the same input in different contexts. This context dependency is very important in NLP, as in natural language, a single word could have a completely different meaning in different contexts, and the meaning of a whole sentence could be changed by a single word.

Of course, such flexibility comes with its own cost. RNNs usually require more time to train and can produce some weird behavior, like loss oscillations or sudden amnesia during the training. However, the research community has already done a lot of work and is still working hard to make RNNs more practical and stable, so RNNs and their modern alternatives like transformers can be seen as a standard building block of the systems that need to process variable-length input.

In our example, we'll use the evolution of RNNs, called the **Long Short-Term Memory (LSTM)** model, which was first proposed in 1995 by Sepp Hochreiter and Jürgen Schmidhuber in the paper *LSTM can solve hard long time lag problems*, and then published in 1996 at a **Neural Information Processing Systems (NIPS)** conference [HS96]. This model is very similar to the RNN we just discussed, but has more complicated internal structure to address some RNN problems.

Word embedding

Another standard building block of modern DL-driven NLP is **word embeddings**, which is also called **word2vec** by one of the most popular training methods for simple tasks. The idea comes from the problem of representing our language sequences in NNs. Normally, NNs work with fixed-sized vectors of numbers, but in NLP, we normally have words or characters as input to the model.



While older methods like word2vec are commonly used for more simple tasks and remain very relevant in the field, other methods such as BERT and transformers are widely used for more complex tasks. We'll briefly discuss transformers later in this chapter.

One possible solution might be *one-hot encoding* our dictionary, which is when every word has its own position in the input vector and we set this number to 1 when we encounter this word in the input sequence. This is a standard approach for NNs when you have to deal with some relatively small discrete set of items and want to represent them in an NN-friendly way. Unfortunately, one-hot encoding doesn't work very well for several reasons:

- Our input set is usually not small. If we want to encode only the most commonly used English dictionary, it will contain at least several thousand words. The *Oxford English Dictionary* has 170,000 commonly used words and 50,000 obsolete and rare words. This is only established vocabulary and doesn't count slang, new words, scientific terms, abbreviations, typos, jokes, Twitter/X memes, and so on. And this is only for the English language!
- The second problem related to the one-hot representation of words is the uneven frequency of vocabulary. There are relatively small sets of very frequent words, like *a* and *cat*, but a very large set of much more rarely used words, like *covfefe* or *bibliopole*, and those rare words can occur only once or twice in a very large text corpus. So, our one-hot representation is very inefficient in terms of space.
- Another issue with simple one-hot representation is not capturing a word's relations. For example, some words are synonyms and have the same meaning, but they will be represented by different vectors. Some words are used very frequently together, like *United Nations* or *fair trade*, and this fact is also not captured in one-hot representation.

To overcome all this, we can use word embeddings, which map every word in some vocabulary into a dense, fixed-length vector of numbers. These numbers are not random but trained on a large corpus of text to capture the context of words. A detailed description of word embeddings is beyond the scope of this book, but this is a really powerful and widely used NLP technique to represent words, characters, and other objects in some sequence. For now, you can think about them as just mapping words into number vectors, and this mapping is convenient for the NN to be able to distinguish words from each other.

To obtain this mapping, two methods exist. First, you can download pretrained vectors for the language that you need. There are several sources of embeddings available; just search on Google for “GloVe pretrained vectors” or “word2vec pretrained” (GloVe and word2vec are different methods used to train such vectors, which produce similar results). An alternate way to obtain embeddings is to train them on your own dataset. To do this, you can either use special tools, such as fastText (<https://fasttext.cc/>, an open source utility from Facebook), or just initialize embeddings randomly and allow your model to adjust them during normal training.

In addition, LLMs (and, in general, any sequence-to-sequence architectures) can produce very high-quality embeddings of texts. The OpenAI ChatGPT API has a special request that converts any piece of text into an embedding vector.

The Encoder-Decoder architecture

Another model that is widely used in NLP is called **Encoder-Decoder**, or seq2seq. It originally comes from machine translation, when your system needs to accept a sequence of words in the source language and produce another sequence in the target language. The idea behind seq2seq is to use an RNN to process an input sequence and *encode* this sequence into some fixed-length representation. This RNN is called an **encoder**. Then you feed the encoded vector into another RNN, called a **decoder**, which has to produce the resulting sequence in the target language. An example of this idea is shown next, where we are translating an English sentence into Russian:

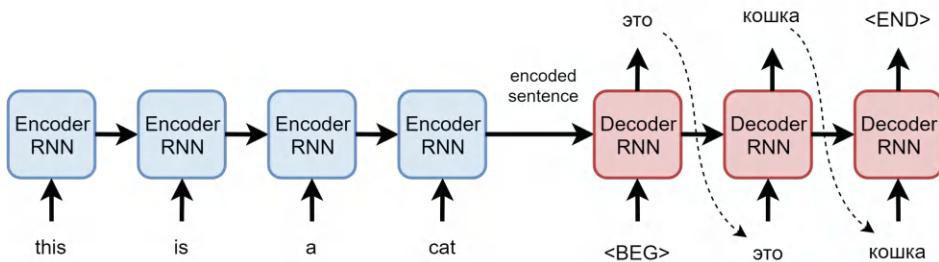


Figure 13.5: The Encoder-Decoder architecture in machine translation

This model (with a lot of modern tweaks and extensions) is still a major workhorse of machine translation, but is general enough to be applicable to a much wider set of domains, for example, audio processing, image annotation, and video captioning. In our TextWorld example, we'll use it to generate embeddings of variable-sized observations from the environment.

RNNs continue to be very effective in certain contexts, but in recent years, NLP has seen significant advancements with the introduction of the more complex Transformer models.

Let's take a look at Transformer architecture next.

Transformers

Transformers is an architecture proposed in 2017 in the paper *Attention is all you need* by Vaswani et al. from Google [Vas17]. At a high-level, it uses the same encoder-decoder architecture we just discussed, but adds several improvements to the underlying building blocks, which turned out to be very important for addressing existing RNN problems:

- **Positional encoding:** This injects information about the input and output sequences' positions into embeddings
- **Attention mechanism:** This concept was proposed in 2015 and could be seen as a trainable way for systems to focus on specific parts of input sequences. In transformers, attention was heavily used (which you can guess from the paper's title)

Nowadays, transformers are at the core of almost every NLP and DL system, including LLMs. I'm not going to go deep into this architecture, as there are lots of resources available about this topic, but if you're curious, you can check the following article: <https://jalammar.github.io/illustrated-transformer/>.

Now we have everything needed to implement our first baseline DQN agent to solve TextWorld problems.

Baseline DQN

Getting back to our TextWorld environment, the following are the major challenges:

- Text sequences might be problematic on their own, as we discussed earlier in this chapter. The variability of sequence lengths might cause vanishing and exploding gradients in RNNs, slow training, and convergence issues. In addition to that, our TextWorld environment provides us with several such sequences that we need to handle separately. Our scene description string, for example, might have a completely different meaning to the agent than the inventory string, which describes our possessions.
- Another obstacle is the action space. As you have seen in the previous section, TextWorld might provide us with a list of commands that we can execute in every state. It significantly reduces the action space we need to choose from, but there are other complications. One of them is that the list of admissible commands changes from state to state (as different locations might allow different commands to be executed). Another issue is that every entry in the admissible commands list is a sequence of words.

Potentially, we might get rid of both of those variabilities by building a dictionary of all possible commands and using it as a discrete, fixed-size action space.

In simple games, this might work, as the number of locations and objects is not that large. You can try this as an exercise, but we will follow a different path.

Thus far, you have seen only discrete action spaces having a small number of predefined actions, and this influenced the architecture of the DQN: the output from the network predicted Q-values for all actions in one pass, which was convenient both during the training and model application (as we need all Q-values for all actions to find argmax anyway). But this choice of DQN architecture is not something dictated by the method, so if needed, we can tweak it. And our issue with a variable number of actions might be solved this way. To get a better understanding of how, let's check the architecture of our TextWorld baseline DQN, as shown in the following figure:

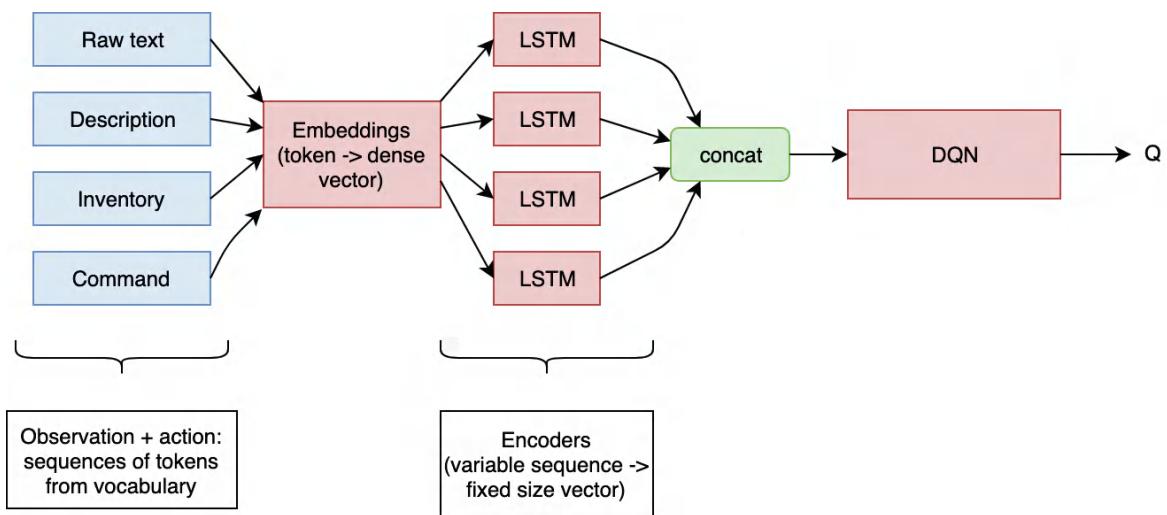


Figure 13.6: The architecture of the TextWorld baseline DQN

The major part of the diagram is occupied by preprocessing blocks. On input to the network (blocks on the left), we get variable sequences of individual parts of observations (“Raw text”, “Description”, and “Inventory”) and the sequence of one action command to be evaluated. This command will be taken from the admissible commands list, and the goal of our network will be to predict a single Q-value for the current game state and this particular command. This approach is different from the DQNs we have used before, but as we don't know in advance which commands will be evaluated in every state, we will evaluate every command individually.

Those four input sequences (which are lists of token IDs in our vocabulary) will be passed through an embeddings layer and then fed into separate LSTM RNNs.

The goal of LSTM networks (which are called “Encoders” in the figure, since LSTMs are concrete implementations of encoders) is to convert variable-length sequences into fixed-size vectors.

Every input piece is processed by its own LSTM with separated weights, which will allow the network to capture different data from different input sequences. Later in this chapter, we'll replace LSTMs with pretrained transformers from the Hugging Face Hub to check the effect of using a much smarter and larger model on the same problem.

The output from the encoders is concatenated into one single vector and passed to the main DQN network. As our variable-length sequences have been transformed into fixed-size vectors, the DQN network is simple: just several feed-forward layers producing one single Q-value. This is less efficient computationally, but for the baseline, it is fine.

The complete source code is in the `Chapter13` directory and it includes the following modules:

- `train_basic.py`: A baseline training program
- `lib/common.py`: Common utilities to set up the Ignite engine and hyperparameters
- `lib/preproc.py`: The preprocessing pipeline, including embeddings and encoder classes
- `lib/model.py`: The DQN model and DQN agent with helper functions

We won't be presenting the full source code in the chapter. Instead, we will be explaining only the most important or tricky parts in the subsequent sections.

Observation preprocessing

Let's start with the leftmost part of our pipeline (*Figure 13.6*). On the input, we're going to get several lists of tokens, both for the individual state observation and for our command that we're going to evaluate. But as you have already seen, the TextWorld environment produces the string and a dict with the extended information, so we need to tokenize the strings and get rid of non-relevant information. That's the responsibility of the `TextWorldPreproc` class, which is defined in the `lib/preproc.py` module:

```
class TextWorldPreproc(gym.Wrapper):
    log = logging.getLogger("TextWorldPreproc")

    OBS_FIELD = "obs"

    def __init__(self, env: gym.Env, vocab_rev: tt.Optional[tt.Dict[str, int]],
                 encode_raw_text: bool = False,
                 encode_extra_fields: tt.Iterable[str] = ('description', 'inventory'),
                 copy_extra_fields: tt.Iterable[str] = (),
                 use_admissible_commands: bool = True, keep_admissible_commands: bool = False,
                 use_intermediate_reward: bool = True, tokens_limit: tt.Optional[int] = None,
                 reward_wrong_last_command: tt.Optional[float] = None
                 ):
        super().__init__(env)
        self.vocab_rev = vocab_rev
        self.encode_raw_text = encode_raw_text
        self.encode_extra_fields = encode_extra_fields
        self.copy_extra_fields = copy_extra_fields
        self.use_admissible_commands = use_admissible_commands
        self.keep_admissible_commands = keep_admissible_commands
        self.use_intermediate_reward = use_intermediate_reward
        self.tokens_limit = tokens_limit
        self.reward_wrong_last_command = reward_wrong_last_command
        self.log.info(f"Using {vocab_rev} vocabulary with {len(vocab_rev)} words")
```

```

super(TextWorldPreproc, self).__init__(env)
self._vocab_rev = vocab_rev
self._encode_raw_text = encode_raw_text
self._encode_extra_field = tuple(encode_extra_fields)
self._copy_extra_fields = tuple(copy_extra_fields)
self._use_admissible_commands = use_admissible_commands
self._keep_admissible_commands = keep_admissible_commands
self._use_intermediate_reward = use_intermediate_reward
self._num_fields = len(self._encode_extra_field) + int(self._encode_raw_text)
self._last_admissible_commands = None
self._last_extra_info = None
self._tokens_limit = tokens_limit
self._reward_wrong_last_command = reward_wrong_last_command
self._cmd_hist = []

```

The class implements the `gym.Wrapper` interface, so it will transform the TextWorld environment observations and actions in the way we need. The constructor accepts several flags, which simplifies future experiments. For example, you can disable the usage of admissible commands or intermediate rewards, set the limit of tokens, or change the set of observation fields to be processed.

Next, the `num_fields` property returns the count of observation sequences, which is used to get the idea of the encoded observation's shape:

```

@property
def num_fields(self):
    return self._num_fields

def _maybe_tokenize(self, s: str) -> str | tt.List[int]:
    if self._vocab_rev is None:
        return s
    tokens = common.tokenize(s, self._vocab_rev)
    if self._tokens_limit is not None:
        tokens = tokens[:self._tokens_limit]
    return tokens

```

The `_maybe_tokenize()` method performs tokenization of input string. If no vocabulary is given, the string is returned unchanged. We will use this functionality in the transformer version, as Hugging Face libraries are performing their own tokenization.

The `_encode()` method is the heart of the observation transformation:

```

def _encode(self, obs: str, extra_info: dict) -> dict:
    obs_result = []
    if self._encode_raw_text:
        obs_result.append(self._maybe_tokenize(obs))
    for field in self._encode_extra_field:
        extra = extra_info[field]
        obs_result.append(self._maybe_tokenize(extra))
    result = {self.OBS_FIELD: obs_result}
    if self._use_admissible_commands:
        result[KEY_ADM_COMMANDS] = [
            self._maybe_tokenize(cmd) for cmd in extra_info[KEY_ADM_COMMANDS]
        ]
        self._last_admissible_commands = extra_info[KEY_ADM_COMMANDS]
    if self._keep_admissible_commands:
        result[KEY_ADM_COMMANDS] = extra_info[KEY_ADM_COMMANDS]
        if 'policy_commands' in extra_info:
            result['policy_commands'] = extra_info['policy_commands']
    self._last_extra_info = extra_info
    for field in self._copy_extra_fields:
        if field in extra_info:
            result[field] = extra_info[field]
    return result

```

The preceding method takes the observation string and the extended information dictionary and returns a single dictionary with the following keys:

- `obs`: The list of lists with the token IDs of input sequences.
- `admissible_commands`: A list with commands available from the current state. Every command is tokenized and converted into the list of token IDs.

In addition, the method remembers the extra information dictionary and raw admissible commands list. This is not needed for training, but will be useful during the model application, to be able to get back the command text from the index of the command.

With the `_encode()` method defined, implementation of the `reset()` and `step()` methods is simple – we're encoding observations and handling intermediate rewards (if they are enabled):

```

def reset(self, seed: tt.Optional[int] = None):
    res, extra = self.env.reset()
    self._cmd_hist = []
    return self._encode(res, extra), extra

def step(self, action):
    if self._use_admissible_commands:

```

```

        action = self._last_admissible_commands[action]
        self._cmd_hist.append(action)
    obs, r, is_done, extra = self.env.step(action)
    if self._use_intermediate_reward:
        r += extra.get('intermediate_reward', 0)
    if self._reward_wrong_last_command is not None:
        if action not in self._last_extra_info[KEY_ADM_COMMANDS]:
            r += self._reward_wrong_last_command
    return self._encode(obs, extra), r, is_done, False, extra

```

It's worth noting that the `step()` method is expecting 4 items to be returned from the wrapped environment, but returns 5 elements. This hides the TextWorld environment incompatibility with the modern Gym interface we've already discussed.

Finally, there are two properties that give access to the remembered state:

```

@property
def last_admissible_commands(self):
    if self._last_admissible_commands:
        return tuple(self._last_admissible_commands)
    return None

@property
def last_extra_info(self):
    return self._last_extra_info

```

To illustrate how the preceding class is supposed to be applied and what it does with the observation, let's check the following small interactive session. Here, we register the game, asking for inventory, intermediate reward, admissible commands, and scene description:

```

>>> from textworld import gym, EnvInfos
>>> from lib import preproc, common
>>> env_id = gym.register_game("games/simple1.ulx", request_infos=EnvInfos(inventory=True,
intermediate_reward=True, admissible_commands=True, description=True))
>>> env = gym.make(env_id)
>>> env.reset()[1]

```

```
'intermediate_reward': 0, 'inventory': 'You are carrying: a type D latchkey, a teacup and a sponge.', 'description': "=- Spare Room =-\nThis might come as a shock to you, but you've just walked into a spare room. You can barely contain your excitement.\n\nYou can make out a closed usual looking crate close by. You can make out a rack. However, the rack, like an empty rack, has nothing on it.\n\nThere is an exit to the east. Don't worry, it is unblocked. You don't like doors? Why not try going south, that entranceway is unguarded.", 'admissible_commands': ['drop sponge', 'drop teacup', 'drop type D latchkey', 'examine crate', 'examine rack', 'examine sponge', 'examine teacup', 'examine type D latchkey', 'go east', 'go south', 'inventory', 'look', 'open crate', 'put sponge on rack', 'put teacup on rack', 'put type D latchkey on rack']}
```

So, that's our raw observation obtained from the TextWorld environment. Now let's extract the game vocabulary and apply our preprocessor:

```
>>> vocab, action_space, obs_space = common.get_games_spaces(["games/simple1.ulx"])
>>> vocab
{0: 'a', 1: 'about', 2: 'accomplished', 3: 'an', 4: 'and', 5: 'appears', 6: 'are', 7: 'arrive', 8: 'as', 9: 'barely', 10: 'be', 11: 'because', 12: 'begin', 13: 'being', 14: 'believe'
...
>>> len(vocab)
192
>>> vocab_rev = common.build_rev_vocab(vocab)
>>> vocab_rev
{'a': 0, 'about': 1, 'accomplished': 2, 'an': 3, 'and': 4, 'appears': 5, 'are': 6, 'arrive': 7
...
>>> pr_env = preproc.TextWorldPreproc(env, vocab_rev)
>>> r = pr_env.reset()
>>> r[0]
{'obs': [[142, 132, 166, 106, 26, 8, 0, 136, 167, 188, 17, 188, 86, 180, 82, 0, 142, 132, 188, 20, 9, 27, 191, 57, 188, 20, 103, 121, 0, 24, 178, 101, 35, 23, 18, 188, 20, 103, 121, 0, 129, 77, 161, 129, 94, 3, 50, 129, 73, 111, 115, 85, 163, 84, 3, 58, 167, 161, 44, 152, 186, 85, 84, 172, 188, 152, 94, 41, 184, 110, 169, 72, 141, 159, 53, 84, 173], [188, 6, 0, 170, 36, 92, 0, 157, 4, 0, 143]], 'admissible_commands': [[42, 143], [42, 157], [42, 170, 36, 92], [55, 35], [55, 129], [55, 143], [55, 157], [55, 170, 36, 92], [71, 44], [71, 141], [83], [100], [117, 35], [127, 143, 115, 129], [127, 157, 115, 129], [127, 170, 36, 92, 115, 129]]}
>>> r[1]
```

```
{'intermediate_reward': 0, 'inventory': 'You are carrying: a type D latchkey, a teacup and a sponge.', 'description': "=- Spare Room =-\nThis might come as a shock to you, but you've just walked into a spare room. You can barely contain your excitement.\n\nYou can make out a closed usual looking crate close by. You can make out a rack. However, the rack, like an empty rack, has nothing on it.\n\nThere is an exit to the east. Don't worry, it is unblocked. You don't like doors? Why not try going south, that entranceway is unguarded.", 'admissible_commands': ['drop sponge', 'drop teacup', 'drop type D latchkey', 'examine crate', 'examine rack', 'examine sponge', 'examine teacup', 'examine type D latchkey', 'go east', 'go south', 'inventory', 'look', 'open crate', 'put sponge on rack', 'put teacup on rack', 'put type D latchkey on rack']}
```

Let's try to execute an action. The 0th action corresponds to the first entry in the admissible commands list, which is “drop sponge” in our case:

```
>>> r[1]['inventory']
'You are carrying: a type D latchkey, a teacup and a sponge.'
>>> obs, reward, is_done, _, info = pr_env.step(0)
>>> info['inventory']
'You are carrying: a type D latchkey and a teacup.'
>>> reward
0
```

As you can see, we no longer have the sponge, but it wasn't the right action to take, thus an intermediate reward was not given.

Okay, this representation still can't be fed directly into NNs, but it is much closer to what we want.

Embeddings and encoders

The next step in the preprocessing pipeline is implemented in two classes:

- **Encoder**: A wrapper around the LSTM unit that transforms one single sequence (after embeddings have been applied) into a fixed-size vector
- **Preprocessor**: This class is responsible for the application of embeddings and the transformation of individual sequences with corresponding encoder classes

The Encoder class is simpler, so let's start with it:

```
class Encoder(nn.Module):
    def __init__(self, emb_size: int, out_size: int):
        super(Encoder, self).__init__()
        self.net = nn.LSTM(input_size=emb_size, hidden_size=out_size, batch_first=True)
```

```
def forward(self, x):
    self.net.flatten_parameters()
    _, hid_cell = self.net(x)
    return hid_cell[0].squeeze(0)
```

The logic is that: we apply the LSTM layer and return its hidden state after processing the sequence.

The Preprocessor class is a bit more complicated, as it combines several Encoder instances and is also responsible for embeddings:

```
class Preprocessor(nn.Module):
    def __init__(self, dict_size: int, emb_size: int, num_sequences: int,
                 enc_output_size: int, extra_flags: tt.Sequence[str] = ()):
        super(Preprocessor, self).__init__()
        self._extra_flags = extra_flags
        self._enc_output_size = enc_output_size
        self.emb = nn.Embedding(num_embeddings=dict_size, embedding_dim=emb_size)
        self.encoders = []
        for idx in range(num_sequences):
            enc = Encoder(emb_size, enc_output_size)
            self.encoders.append(enc)
            self.add_module(f"enc_{idx}", enc)
        self.enc_commands = Encoder(emb_size, enc_output_size)
```

In the constructor, we create an embeddings layer, which will map every token in our dictionary into a fixed-size dense vector. Then we create num_sequences instances of Encoder for every input sequence and one additional instance to encode command tokens.

The internal method `_apply_encoder()` takes the batch of sequences (every sequence is a list of token IDs) and transforms it with an encoder:

```
def _apply_encoder(self, batch: tt.List[tt.List[int]], encoder: Encoder):
    dev = self.emb.weight.device
    batch_t = [self.emb(torch.tensor(sample).to(dev)) for sample in batch]
    batch_seq = rnn_utils.pack_sequence(batch_t, enforce_sorted=False)
    return encoder(batch_seq)
```

In earlier versions of PyTorch, we needed to sort a batch of variable-length sequences before RNN application. Since PyTorch 1.0, this is no longer needed, as this sorting and transformation is handled by the `PackedSequence` class internally. To enable this functionality, we need to pass the `enforce_sorted=False` parameter.

The `encode_observations()` method takes a batch of observations (from `TextWorldPreproc`) and encodes them into a tensor:

```
def encode_observations(self, observations: tt.List[dict]) -> torch.Tensor:
    sequences = [obs[TextWorldPreproc.OBS_FIELD] for obs in observations]
    res_t = self.encode_sequences(sequences)
    if not self._extra_flags:
        return res_t
    extra = [[obs[field] for field in self._extra_flags] for obs in observations]
    extra_t = torch.Tensor(extra).to(res_t.device)
    res_t = torch.cat([res_t, extra_t], dim=1)
    return res_t
```

Besides variable sequences, we can pass extra “flags” fields directly into the encoded tensor. This functionality will be used in later experiments and extensions to the basic method.

Finally, two methods, `encode_sequences()` and `encode_commands()`, are used to apply different encoders to the batch of variable-length sequences:

```
def encode_sequences(self, batches):
    data = []
    for enc, enc_batch in zip(self.encoders, zip(*batches)):
        data.append(self._apply_encoder(enc_batch, enc))
    res_t = torch.cat(data, dim=1)
    return res_t

def encode_commands(self, batch):
    return self._apply_encoder(batch, self.enc_commands)
```

The DQN model and the agent

With all those preparations made, let’s look at the brains of our agent: the DQN model. It should accept vectors of `num_sequences × encoder_size` and produce a single scalar value. But there is one difference from the other DQN models covered, which is in the way we apply the model:

```
class DQNNModel(nn.Module):
    def __init__(self, obs_size: int, cmd_size: int, hid_size: int = 256):
        super(DQNNModel, self).__init__()

        self.net = nn.Sequential(
            nn.Linear(obs_size + cmd_size, hid_size),
            nn.ReLU(),
```

```
        nn.Linear(hid_size, 1)
    )

    def forward(self, obs, cmd):
        x = torch.cat((obs, cmd), dim=1)
        return self.net(x)

    @torch.no_grad()
    def q_values(self, obs_t, commands_t):
        result = []
        for cmd_t in commands_t:
            qval = self(obs_t, cmd_t.unsqueeze(0))[0].cpu().item()
            result.append(qval)
        return result
```

In the preceding code, the `forward()` method accepts two batches – observations and commands – producing the batch of Q-values for every pair. Another method, `q_values()`, takes one observation produced by the `Preprocessor` class and the tensor of encoded commands, then applies the model and returns a list of Q-values for every command.

In the `model.py` module, we have the `DQNAgent` class, which takes the preprocessor and implements the PTAN Agent interface to hide the details of observation preprocessing on decision-making.

Training code

With all the preparations and preprocessing in place, the rest of the code is almost the same as we already implemented in previous chapters, so I won't repeat the training code; I will just describe the training logic.

To train the model, the `Chapter13/train_basic.py` utility has to be used. It allows several command-line arguments to change the training behavior:

- `-g` or `-game`: This is the prefix of the game files in the `games` directory. The provided script generates several games named `simpleNN.ulx`, where `NN` is the game seed.
- `-s` or `-suffices`: This is the count of games to be used during the training. If you specify 1 (which is the default), the training will be performed only on the file `simple1.ulx`. If option `-s 10` is given, 10 games with indices 1 ... 10 will be registered and used for training. This option is used to increase the variability in the training games, as our goal is not just to learn how to play concrete games but also (hopefully) to learn how to behave in other similar games.
- `-v` or `-validation`: This is the suffix of the game to be used for validation. It equals to `-val` by default and defines the game file that will be used to check the generalization of our trained agent.

- **-params**: This means the hyperparameters to be used. Two sets are defined in `lib/common.py`: small and medium. The first one has a small number of embeddings and encoder vectors, which is great for solving a couple of games quickly; however, this set struggles with converging when many games are used for training.
- **-dev**: This option specifies the device name for computations.
- **-r or -run**: This is the name of the run and is used in the name of the save directory and TensorBoard.

During the training, validation is performed every 100 training iterations and the validation game is run on the current network. The reward and the number of steps are recorded in TensorBoard and help us to understand the generalization capabilities of our agent. Generalization in RL is known to be a large issue, as with a limited set of trajectories, the training process has a tendency to overfit to some states, which doesn't guarantee good behavior on unseen games. In comparison to Atari games, where the gameplay normally doesn't change much, the variability of interactive fiction games might be high, due to different quests, objects, and the way they communicate. So, it's an interesting experiment to check how our agent is able to generalize between games.

Training results

By default, the script `games/make_games.sh` generates 20 games with names from `simple1.ulx` to `simple20.ulx`, plus a game for validation: `simple-val.ulx`.

To begin, let's train the agent on one game, using the small hyperparameters set:

```
$ ./train_basic.py -s 1 --dev cuda -r t1
Registered env tw-simple-v0 for game files ['games/simple1.ulx']
Game tw-simple-v1, with file games/simple-val.ulx will be used for validation
Episode 1: reward=0 (avg 0.00), steps=50 (avg 50.00), speed=0.0 f/s, elapsed=0:00:04
Episode 2: reward=1 (avg 0.02), steps=50 (avg 50.00), speed=0.0 f/s, elapsed=0:00:04
1: best avg training reward: 0.020, saved
Episode 3: reward=-2 (avg -0.02), steps=50 (avg 50.00), speed=0.0 f/s, elapsed=0:00:04
Episode 4: reward=6 (avg 0.10), steps=30 (avg 49.60), speed=0.0 f/s, elapsed=0:00:04
...
```

Option `-s` specifies the number of game indices that will be used for training. In this case, only one will be used. The training stops when the average number of steps in the game drops below 15, which means the agent has found the proper sequence of steps and can reach the end of the game in an efficient way.

In the case of one game, it takes just 3 minutes and about 120 episodes to solve the game. The following figure shows the reward and number of steps dynamics during the training:

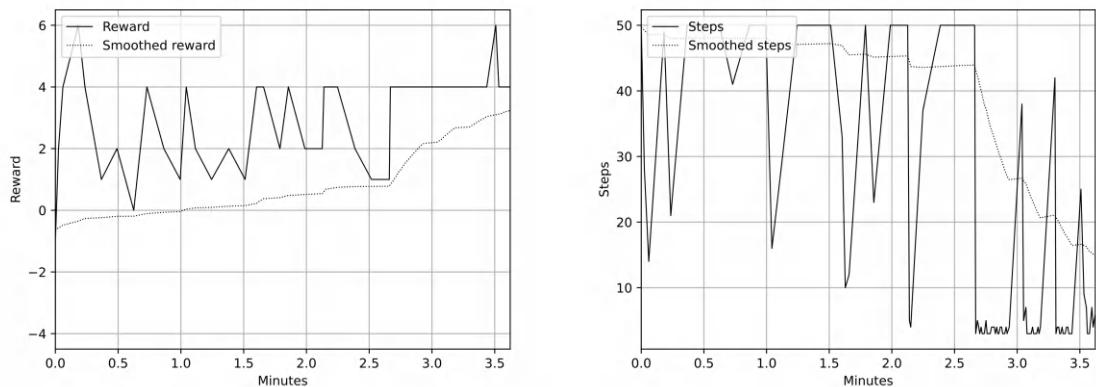


Figure 13.7: Training reward (left) and count of steps in episodes (right) for training on one game

But if we check the validation reward (which is a reward obtained on the game `simple-val.ulx`), we see zero improvement over time. In my case, validation reward was zero and count of steps on validation episodes were 50 (which is a default time limit). It just means that the learned agent wasn't able to generalize.

If we try to increase the number of games used for the training, the convergence will require more time, as the network needs to discover more sequences of actions in different states. The following are the same charts for reward and steps for 20 games (with option `-s 20` passed):

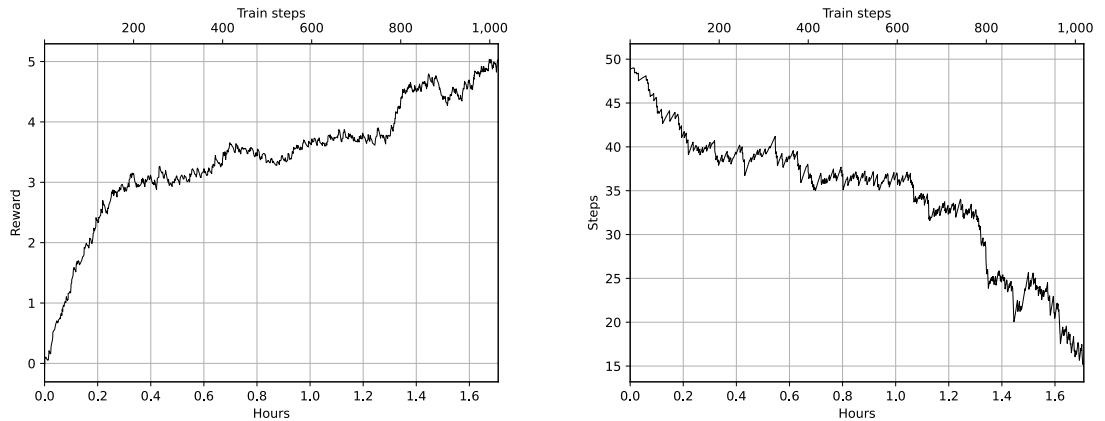


Figure 13.8: Training reward (left) and count of steps in episodes (right) for training on 20 games

As you can see, it takes almost two hours to converge, but still, our small hyperparameter set is able to improve the performance on 20 games played during the training.

Validation metrics, as shown in the following figure, are now slightly more interesting – at the end of the training, the agent was able to obtain the score of 2 (with maximum 6) and somewhere in the middle of the training, it got 4. But count of steps on validation game are still 50, which means that the agent just walks around semi-randomly executing some actions. Not very impressive.

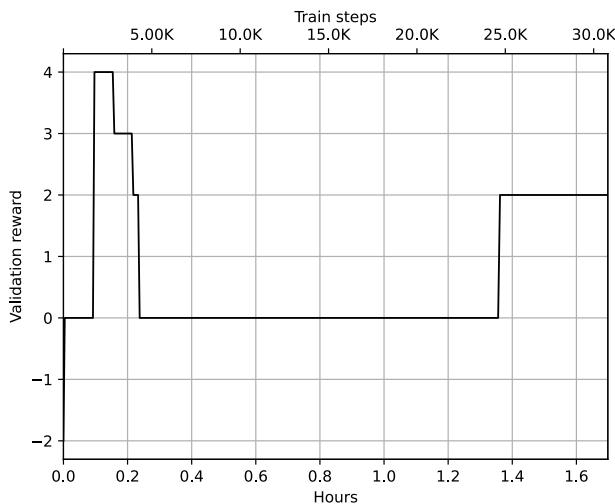


Figure 13.9: Validation reward during the training on 20 games

I haven't tried different hyperparameters on this agent (you can do this with `-s medium`).

Tweaking observations

Our first series of attempts will be in feeding more information to the agent. Here, I will just briefly introduce the changes made and effect they had on a training result. You can find the full example in `Chapter13/train_prepoc.py`.

Tracking visited rooms

First, you will notice that our agent has no idea whether the current room was already visited or not. In situations when the agent already knows the optimal way to the goal, it might be not needed (as generated games always have different rooms). But if the policy is not perfect, it might be useful to have a clear indication that we're visiting the same room over and over again.

To feed this knowledge into the observation, I implemented a simple room tracking in the `preproc.LocationWrapper` class, which tracks visited rooms over the episode. Then this flag is concatenated to the agent's observation as a single 1 if the room was visited before or 0 if it is a new location.

To train our agent with this extension, you can run `train_prepoc.py` with the additional command-line option `-seen-rooms`.

The following are charts comparing our baseline version with this extra observation on 20 games. As you can see, reward on training games are almost the same, but validation reward was improved – we were able to get non-zero validation reward almost during the whole training. But count of steps on validation game are still 50.

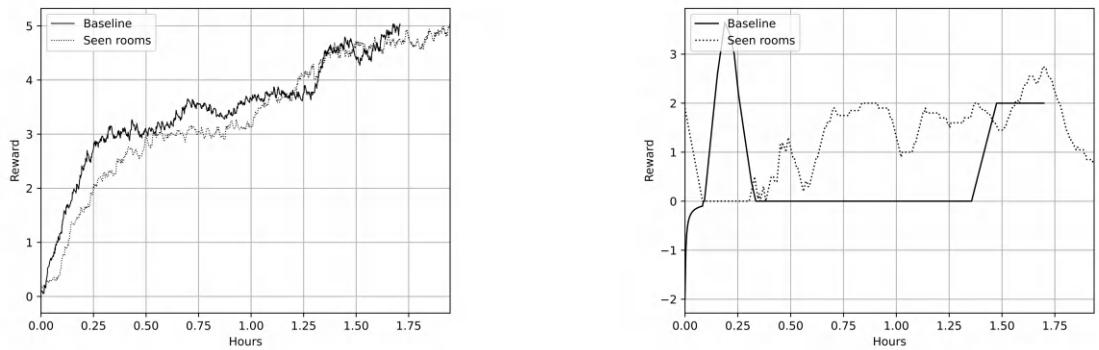


Figure 13.10: Training reward (left) and validation reward (right) on 20 games

But after trying this extension on 200 games (you need to change the script to generate them), I've got an interesting result: after 14 hours of training and 8,000 episodes, the agent was not just getting the maximum score on validation game but was able to do this efficiently (with count of steps less than 10). This is shown in *Figure 13.11* and *Figure 13.12*.

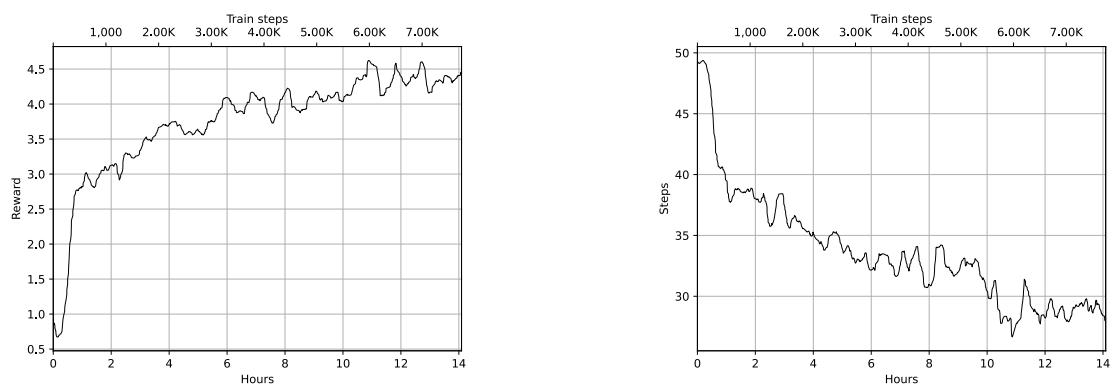


Figure 13.11: Training reward (left) and episode steps (right) on 200 games

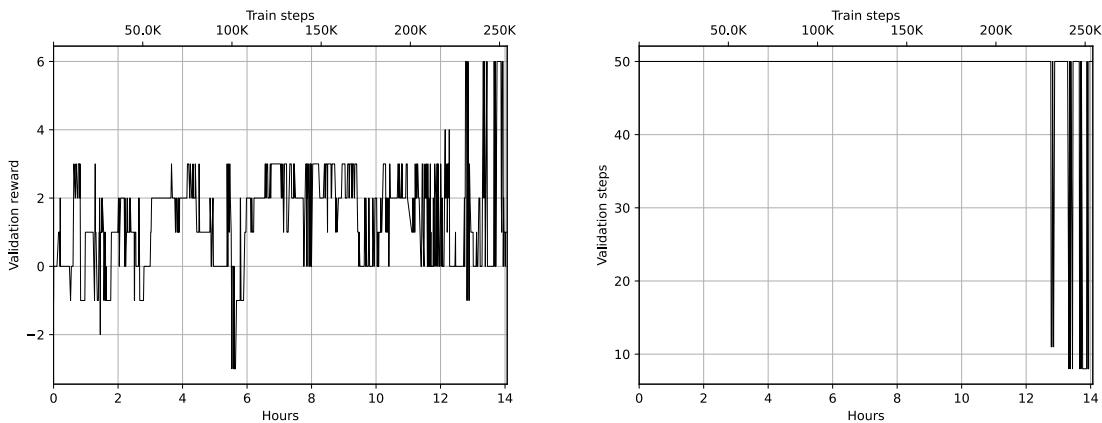


Figure 13.12: Validation reward (left) and episode steps (right)

Relative actions

The second attempt to improve the agent's learning was about the action space. In principle, our agent's task is to navigate the rooms and perform specific actions on objects around (like opening the locker and taking something out of it). So, navigation is a very important aspect in learning process.

At the moment, we move around by executing "absolute coordinate" commands, like "go north" or "go east", which are room-specific, as different rooms might have different exits available. In addition, after executing some action, the inverse action (to get back to the original room) depends on the first action. For example, if we are in the room with an exit to the north, after using this exit, we need to execute "go south" to get back. But our agent has no memory of the history of actions, so after going north, we have no idea how to get back.

In the previous section, we added information about whether the room was visited or not. Now we'll transform absolute actions into relative actions. To get that, our wrapper `preproc.RelativeDirectionsWrapper` tracks our "heading direction" and replaces the "go north" or "go east" commands with "go left", "go right", "go forward", or "go back" depending on the heading direction. In this example, when we're in the room with an exit to the north and we're heading north, we need to execute the command "go forward" to use the exit. After that, we can run the command "go back" to step back in the originating room. Hopefully, this transformation will allow our model to navigate the TextWorld games with more ease.

To enable this extension, you need to run `train_prepoc.py` with the `-relative-actions` command-line option. This extension also requires "seen rooms" to be enabled, so here, we're testing the effect of both modifications combined.

On 20 games, training dynamics and validation results are very similar to the baseline version (*Figure 13.13*):

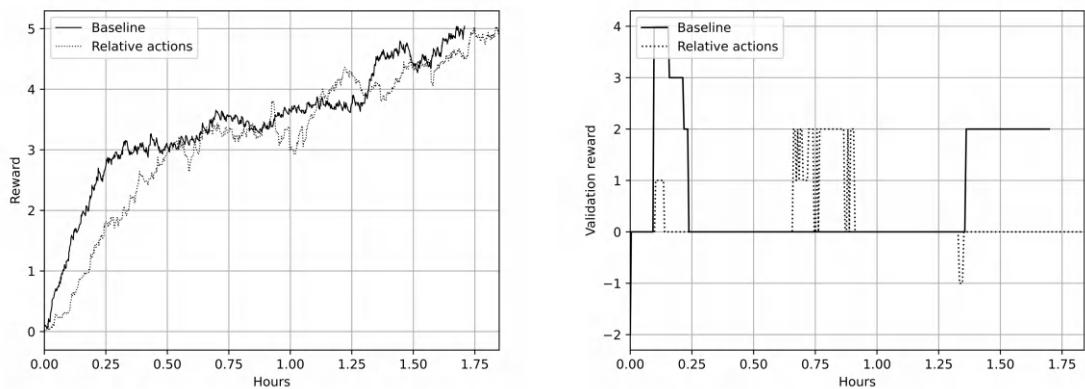


Figure 13.13: Training reward (left) and validation reward (right) on 20 games

But on 200 games, the agent was able to get the maximum score on validation game after just 2.5 hours (instead of 13 in the “Seen rooms” extension). The number of steps on validation was also decreased below 10. But, unfortunately, after further training, validation metrics reverted to lower validation scores, so the agent overfitted to the games and unlearned the skills it had:

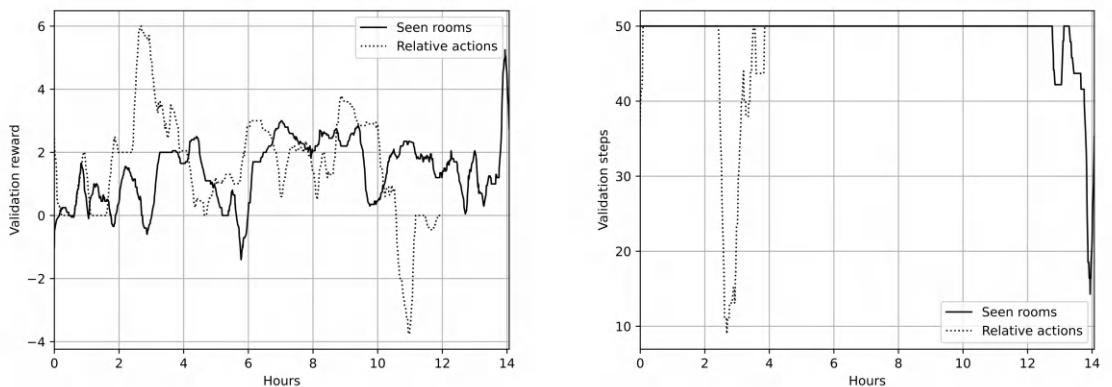


Figure 13.14: Validation reward (left) and episode steps (right) on 200 games

Objective in observation

Another idea is to feed the game objective into the agent observations. The objective is presented as text at the beginning of the game, for example, *First thing I need you to do is to try to venture east. Then, venture south. After that, try to go to the south. Once you succeed at that, try to go west. If you can finish that, pick up the coin from the floor of the chamber. Once that's all handled, you can stop!*

This information might be useful for the agent to plan its actions, so let's add it to the encoded vectors. We don't need to implement another wrapper, as our existing ones are flexible enough already. Just a couple of extra arguments need to be passed to them. To enable the objective, you need to run `train_prepoc.py` with the `-objective` command-line argument.

Results on 20 games are almost identical to the baseline and shown in *Figure 13.15*:

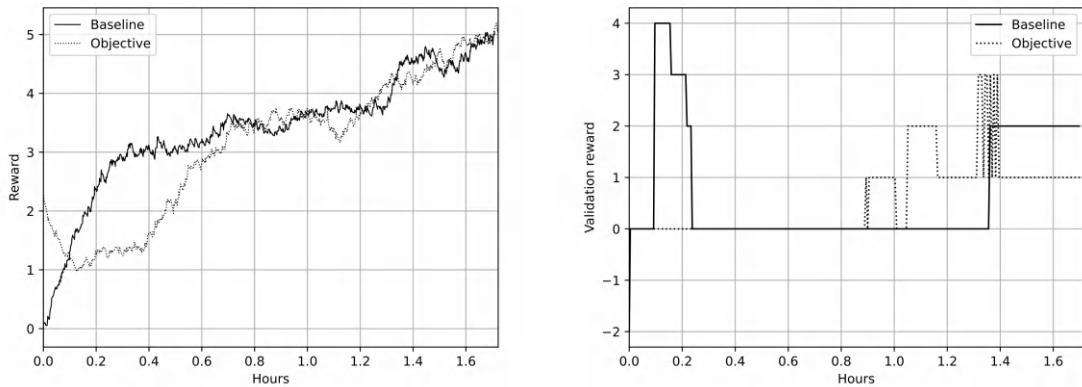


Figure 13.15: Training reward (left) and validation reward (right) on 20 games

Training on 200 games was less successful than for previous modifications: during the validation, score was around 2-4 but never reached 6. Charts for reward and validation reward are shown in *Figure 13.16*:

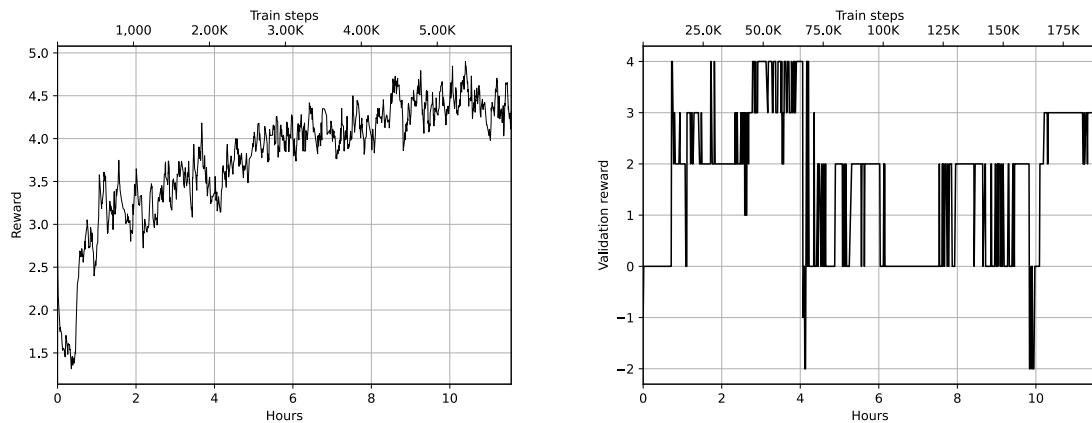


Figure 13.16: Training reward (left) and validation reward (right) on 200 games

Transformers

The next approach we'll try is pretrained language models, which is a de facto standard in modern NLP. Thanks to public model repositories, like the Hugging Face Hub, we don't need to train them from scratch, which might be very costly. We can just plug the pretrained model into our architecture and fine-tune a small portion of our network to our dataset.

There is a wide variety of models – different sizes, datasets they were pretrained on, training techniques, etc. But all of them use a simple API, so plugging them into our code is simple and straightforward.

First, we need to install the libraries. For our task, we'll use the package `sentence-transformers==2.6.1`, which you need to install manually. Once this is done, you can use it to compute embeddings of any sentences given as strings:

```
>>> from sentence_transformers import SentenceTransformer
>>> tr = SentenceTransformer("sentence-transformers/all-MiniLM-L6-v2")
>>> tr.get_sentence_embedding_dimension()
384
>>> r = tr.encode("You're standing in an ordinary boring room")
>>> type(r)
<class 'numpy.ndarray'>
>>> r.shape
(384,)
>>> r2 = tr.encode(["sentence 1", "sentence 2"], convert_to_tensor=True)
>>> type(r2)
<class 'torch.Tensor'>
```

```
>>> r2.shape
torch.Size([2, 384])
```

Here we used the `all-MiniLM-L6-v2` model, which is relatively small — 22M parameters trained on 1.2B tokens. You can find more information on the Hugging Face website: <https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2>.

In our case, we'll use the high-level interface, where we feed strings with sentences, and the library and model are doing all the conversion for us. But there is lots of flexibility if needed.

The `preproc.TransformerPreprocessor` class implements the same interface as our old `Preprocessor` class (which used LSTM for embeddings) and I'm not going to show the code as it is very straightforward.

To train our agent with transformers, you need to run the `Chapter13/train_tr.py` module. During the training, transformers turned out to be slower (2 FPS vs 6 FPS on my machine), which is not surprising, as the model is much more complicated than LSTM models. But training dynamics is better on 20 and 200 games. In *Figure 13.17*, you can see training reward and count of episode steps for transformers and baseline. The baseline version required 1,000 episodes to reach 15 steps, where transformers required just 400. Validation on 20 games had worse reward than baseline version (max score was 2):

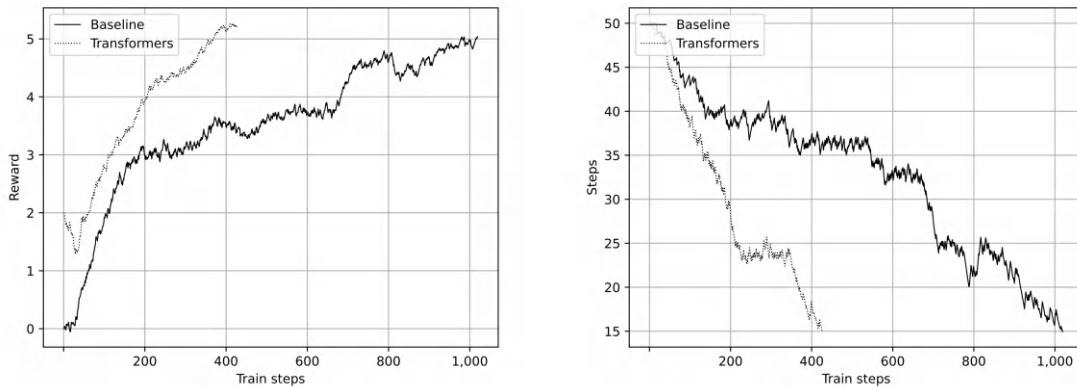


Figure 13.17: Training reward (left) and training episodes length (right) on 20 games

The same situation was on 200 games – the agent learns more efficiently (in terms of games), but validation is not great. This could be explained by the much larger capacity of transformers – the embeddings they produce are almost 20 times larger than our baseline model (384 vs 20), so it is easier for our agent to just memorize the correct sequence of steps instead of trying to find high-level generic observations to actions mapping.

ChatGPT

To finalize the discussion of TextWorld, let's try a different approach – using LLMs. Right after public release at the end of 2022, OpenAI ChatGPT became very popular and literally transformed the chatbot and text-based assistant landscape. Just in a year since its release, hundreds of new use cases appeared and thousands of applications using LLMs under the hood were developed. Let's try to apply this technology to our problem of solving TextWorld games.

Setup

First, you will need an account on <https://openai.com>. We'll start our experiment with an interactive web-based chat, which could be tried for free and without registration (at the moment of writing), but our next example will use the ChatGPT API, for which you will need to generate an API key at <https://platform.openai.com>. Once the key is created, you need to set it to the environment variable `OPENAI_API_KEY` in the shell you're using.

We'll also use the `langchain` library to communicate with ChatGPT from Python, so please install it with the following commands:

```
$ pip install langchain==0.1.15 langchain-openai==0.1.2
```



Note that these packages are quite dynamic and new versions might break compatibility.

Interactive mode

In our first example, we'll use the web-based ChatGPT interface, asking it to generate game commands from room descriptions and game objectives. The code is in `Chapter13/chatgpt_interactive.py` and it does the following:

1. Starts the TextWorld environment for the game ID given in the command line
2. Creates the prompt for ChatGPT with instructions, game objective, and room description

3. Writes this prompt to the console
4. Reads the command to be executed from the console
5. Executes the command in the environment
6. Repeats from step 2 until the limit of steps has been reached or until we've solved the game

So, your task is to copy the generated prompt and paste it into the <https://chat.openai.com> web interface. ChatGPT will generate the command that has to be entered into the console.

The full code is very simple and short. It has just a single `play_game` function, which executes the game loop using the created environment:

```
env_id = register_game(
    gamefile=f"games/{args.game}{index}.ulx",
    request_infos=EnvInfos(description=True, objective=True),
)
env = gym.make(env_id)
```

During environment creation, we ask just for two extra information pieces: room description and game objective. In principle, both are present in free-text observations, so we could parse them from this text. But for convenience, we ask TextWorld to provide this explicitly.

In the beginning of the `play_game` function, we reset the environment and generate the initial prompt:

```
def play_game(env, max_steps: int = 20) -> bool:
    commands = []

    obs, info = env.reset()

    print(textwrap.dedent("""\
        You're playing the interactive fiction game.
        Here is the game objective: %s

        Here is the room description: %s

        What command do you want to execute next? Reply with
        just a command in lowercase and nothing else.
        """") % (info['objective'], info['description']))

    print("== Send this to chat.openai.com and type the reply...")
```

I haven't spent much time designing it, as basically, everything worked from the first attempt and I'm sure it could be improved. The last sentence, "Reply with just a command in lowercase and nothing else," prevents the chatbot from being too verbose and saves us from parsing the output.

Then we execute the loop until the game is solved or the limit of steps has been reached:

```
while len(commands) < max_steps:
    cmd = input("">>>> ")
    commands.append(cmd)
    obs, r, is_done, info = env.step(cmd)
    if is_done:
        print(f"You won in {len(commands)} steps! "
              f"Don't forget to congratulate ChatGPT!")
        return True

    print(textwrap.dedent("""\
Last command result: %s
Room description: %s

What's the next command?
""") % (obs, info['description']))
    print("==> Send this to chat.openai.com and type the reply...")

print(f"Wasn't able to solve after {max_steps} steps, commands: {commands}")
return False
```

The continuation prompt is much simpler – we just give the obtained observation (which is an outcome of the command) and new room description. We don't need to pass the objective again, as the web interface keeps the context of conversation, so the chatbot is aware of our prior instructions.

Let's take a look at one game test (with seed 1). I stripped the room descriptions to decrease the verbosity; otherwise, it would take several pages of the book. But you should copy the generated text fully:

```
$ ./chatgpt_interactive.py 1
Starting game 1

You're playing the interactive fiction game.
Here is the game objective: Get ready to pick stuff up and put it in places, because
you've just entered TextWorld! Here is your task for today. First stop, go to the south.
And then, head east. With that over with, make an effort to go north. With that
accomplished, ensure that the type D locker is open. Then, take the keyboard from the
type D locker. That's it!

Here is the room description: -= Spare Room =-
What command do you want to execute next? Reply with
just a command in lowercase and nothing else.

==> Send this to chat.openai.com and type the reply...
>>> go south
```

```
Last command result: -- Dish-Pit --
Room description: -- Dish-Pit --
What's the next command?

==== Send this to chat.openai.com and type the reply...
>>> go east
Last command result: -- Cookhouse --
Room description: -- Cookhouse --
What's the next command?

==== Send this to chat.openai.com and type the reply...
>>> go north
Last command result: -- Studio --
Room description: -- Studio --
What's the next command?

==== Send this to chat.openai.com and type the reply...
>>> open type d locker
Last command result: You open the type D locker, revealing a keyboard.
Room description: -- Studio --
What's the next command?

==== Send this to chat.openai.com and type the reply...
>>> take keyboard
You won in 5 steps! Don't forget to congratulate ChatGPT!
```

As you can see, the LLM was able to solve the task perfectly. What is even more spectacular is that overall task is harder – we ask it to generate commands and not to make a decision from a list of “admissible commands” as earlier in the chapter.

ChatGPT API

Since copy-pasting is tedious and boring, let’s automate our agent using the ChatGPT API. We’ll use the langchain library (<https://python.langchain.com/>), which provides enough flexibility and control to leverage the LLM functionality.

The full code example is in Chapter13/chatgpt_auto.py. Here, I will cover the core function, play_game():

```
from langchain_openai import ChatOpenAI
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder

def play_game(env, max_steps: int = 20) -> bool:
    prompt_init = ChatPromptTemplate.from_messages([
        {"role": "system", "content": "You are a game player."},
```

```
("system", "You're playing the interactive fiction game. "
            "Reply with just a command in lowercase and nothing else"),
    ("system", "Game objective: {objective}"),
    ("user", "Room description: {description}"),
    ("user", "What command you want to execute next?"),
)
llm = ChatOpenAI()
output_parser = StrOutputParser()
```

Our initial prompt is the same as before — we’re instructing the chatbot about the kind of the game we’re playing and asking it to reply only with commands to be fed into the game.

Then we reset the environment and generate the first message, passing the information from TextWorld:

```
commands = []

obs, info = env.reset()
init_msg = prompt_init.invoke({
    "objective": info['objective'],
    "description": info['description'],
})

context = init_msg.to_messages()
ai_msg = llm.invoke(init_msg)
context.append(ai_msg)
cmd = output_parser.invoke(ai_msg)
```

The variable `context` is very important and it contains the list of all messages (both from human and the chatbot) in our conversation so far. We’ll pass those messages to the chatbot to preserve the process of the game. This is needed because the game objective is being shown only once and not repeated again. Without the history, the agent doesn’t have enough information to perform the required sequence of steps. On the other hand, having lots of text passed to the chatbot might lead to high costs (as the ChatGPT API is billed for tokens being processed). Our game is not long (5-7 steps is enough to finish the task), so it is not a major concern, but for more complex games, history might be optimized.

Then the game loop follows, which is very similar to what we had in interactive version, but without console communication:

```
prompt_next = ChatPromptTemplate.from_messages([
    MessagesPlaceholder(variable_name="chat_history"),
    ("user", "Last command result: {result}"),
```

```

        ("user", "Room description: {description}"),
        ("user", "What command you want to execute next?"),
    ])

for _ in range(max_steps):
    commands.append(cmd)
    print(">>>", cmd)
    obs, r, is_done, info = env.step(cmd)
    if is_done:
        print(f"I won in {len(commands)} steps!")
        return True

user_msgs = prompt_next.invoke({
    "chat_history": context,
    "result": obs.strip(),
    "description": info['description'],
})
context = user_msgs.to_messages()
ai_msg = llm.invoke(user_msgs)
context.append(ai_msg)
cmd = output_parser.invoke(ai_msg)

```

In the continuation prompt, we pass the history of conversation, result of last command, description of the current room, and ask for the next command.

We also limit the amount of steps to prevent the agent from getting stuck in loops (it happens sometimes). If the game isn't solved after 20 steps, we exit the loop:

```

print(f"Wasn't able to solve after {max_steps} steps, commands: {commands}")
return False

```

I did experiment with the preceding code on 20 TextWorld games (with seeds 1 ... 20) and it was able to solve 9 games out of 20. Most of the failed situations were because the agent went into the loop — issuing the wrong command not properly interpreted by TextWorld (like “take the key” instead of “take the key from the box”), or getting stuck in navigation.

In two games, ChatGPT failed because of generating the command “exit”, which makes TextWorld stop immediately. Most likely, detecting this command or prohibiting its generation in the prompt might increase the number of solved games.

But still, even 9 games solved by the agent without any prior training is quite an impressive result. In terms of ChatGPT costs, running the experiment took 450K tokens to be processed, which cost me \$0.20. Not a big price for having fun!

Summary

In this chapter, you have seen how DQN can be applied to interactive fiction games, which is an interesting and challenging domain at the intersection of RL and NLP. You learned how to handle complex textual data with NLP tools and experimented with fun and challenging interactive fiction environments, with lots of opportunities for future practical experimentation. In addition, we used the transformer model from the Hugging Face library and experimented with ChatGPT.

In the next chapter, we will continue our exploration of “RL in the wild” and check the applicability of RL methods in web automation.

14

Web Navigation

We will now take a look at some other practical applications of **reinforcement learning (RL)**: web navigation and browser automation. This is a really useful example of how RL methods could be applied to a practical problem, including the complications you might face and how they could be addressed.

In this chapter, we will:

- Discuss **web navigation** in general and the practical application of browser automation
- Explore how web navigation can be solved with an RL approach
- Take a deep look at one very interesting, but commonly overlooked and a bit abandoned, RL benchmark that was implemented by OpenAI, called **Mini World of Bits (MiniWoB)**.

The evolution of web navigation

When the web was invented, it started as several text-only web pages interconnected by hyperlinks. If you're curious, here is the home of the first web page, <http://info.cern.ch>, with text and links. The only thing you can do is read the text and click on links to navigate between pages.

Several years later, in 1995, the **Internet Engineering Task Force (IETF)** published the HTML 2.0 specification, which had a lot of extensions to the original version invented by Tim Berners-Lee. Among these extensions were forms and form elements that allowed web page authors to add activity to their websites. Users could enter and change text, toggle checkboxes, select drop-down lists, and push buttons.

The set of controls was similar to a minimalistic set of **graphical user interface (GUI)** application controls. The difference was that this happened inside the browser's window, and both the data and **user interface (UI)** controls that users interacted with were defined by the server's page, but not by the local installed application.

Fast forward 29 years, and now we have JavaScript, HTML5 canvas, and Microsoft Office applications working inside our browsers. The boundary between the desktop and the web is so thin and blurry that you may not even know whether the app you're using is an HTML page or a native app. However, it is still the browser that understands HTML and communicates with the outside world using HTTP.

At its core, web navigation is defined as the process of a user interacting with a website or websites. The user can click on links, type text, or carry out any other actions to reach their goal, such as sending an email, finding out the exact dates of the French Revolution, or checking recent Facebook notifications. All this will be done using web navigation, so that leaves a question: can our program learn how to do the same?

Browser automation and RL

For a long time, automating website interaction focused on the very practical tasks of **website testing** and **web scraping**. Website testing is especially critical when you have a complicated website that you (or other people) have developed and you want to ensure that it does what it is supposed to do. For example, if you have a login page that has been redesigned and is ready to be deployed on a live website, then you will want to be sure that this new design does sane things in case a wrong password is entered, the user clicks on *I forgot my password*, and so on. A complex website could potentially include hundreds or thousands of use cases that should be tested on every release, so all such functions should be automated.

Web scraping solves the problem of extracting data from websites at scale. For example, if you want to build a system that aggregates all prices for all the pizza places in your town, you will potentially need to deal with hundreds of different websites, which could be problematic to build and maintain. Web scraping tools try to solve the problem of interacting with websites, providing various functionality from simple HTTP requests and subsequent HTML parsing to full emulation of the user moving the mouse, clicking buttons, user's reaction delays, and so on.

The standard approach to browser automation normally allows you to control the real browser, such as Chrome or Firefox, with your program, which can observe the web page data, like the **Document Object Model (DOM)** tree and an object's location on the screen, and issue the actions, like moving the mouse, pressing some keys, pushing the **Back** button, or just executing some JavaScript code. The connection to the RL problem setup is obvious: our agent interacts with the web page and browser by issuing actions and observing the state. The reward is not that clear and should be task-specific, like successfully filling a form in or reaching the page with the desired information.

Practical applications of a system that could learn browser tasks are related to the previous use cases, and include the following:

- In web testing for very large websites, it's extremely tedious to define the testing process using low-level browser actions like "move the mouse five pixels to the left, then press the left button." What you want to do is give the system some demonstrations and let it generalize and repeat the shown actions in all similar situations, or at least make it robust enough for UI redesign, button text change, and so on.
- There are many cases when you don't know the problem in advance, for example, when you want the system to explore the weak points of the website, like security vulnerabilities. In that case, the RL agent could try a lot of weird actions very quickly, much faster than humans could. Of course, the action space for security testing is enormous, so random clicking won't be as effective as experienced human testers. In that case, the RL-based system could, potentially, combine the prior knowledge and experience of humans but still keep the ability to explore and learn from this exploration.
- Another potential domain that could benefit from RL browser automation is scraping and web data extraction in general. For example, you might want to extract some data from hundreds of thousands of different websites, like hotel websites, car rental agents, or other businesses around the world. Very often, before you get to the desired data, a form with parameters needs to be filled out, which becomes a very nontrivial task given the different websites' design, layout, and natural language flexibility. With such a task, an RL agent can save tons of time and effort by extracting the data reliably and at scale.

Challenges in browser automation

Potential practical applications of browser automation with RL are attractive but have one very serious drawback: they're too large to be used for research and the comparison of methods. In fact, the implementation of a full-sized web scraping system could take months of effort from a team, and most of the issues would not be directly related to RL, like data gathering, browser engine communication, input and output representation, and lots of other questions that real production system development consists of.

By solving all these issues, we can easily miss the forest by looking at the trees. That's why researchers love benchmark datasets, like MNIST, ImageNet, and the Atari suite. However, not every problem makes a good benchmark. On the one hand, it should be simple enough to allow quick experimentation and comparison between methods. On the other hand, the benchmark has to be challenging and leave room for improvement. For example, Atari benchmarks consist of a wide variety of games, from very simple ones that can be solved in half an hour (like Pong), to quite complex games that were properly solved only recently (like Montezuma's Revenge, which requires the complex planning of actions).

To the best of my knowledge, there is only one such benchmark for the browser automation domain, which makes it even worse that this benchmark was undeservedly forgotten by the RL community. As an attempt to fix this issue, we will take a look at the benchmark in this chapter. Let's talk about its history first.

The MiniWoB benchmark

In December 2016, OpenAI published a dataset called MiniWoB that contains 80 browser-based tasks. These tasks are observed at the pixel level (strictly speaking, besides pixels, a text description of tasks is given to the agent) and are supposed to be communicated with the mouse and keyboard actions using the **Virtual Network Computing (VNC)** client. VNC is a standard remote desktop protocol by which a VNC server allows clients to connect to and work with a server's GUI applications using the mouse and keyboard via the network.

The 80 tasks vary a lot in terms of complexity and the actions required from the agent. Some tasks are very simple, even for RL, like “click on the dialog’s close button,” or “push the single button,” but some require multiple steps, for example, “open collapsed groups and click on the link with some text,” or “select a specific date using the date picker tool” (and this date is randomly generated every episode). Some of the tasks are simple for humans but require character recognition, for example, “mark checkboxes with this text” (and the text is generated randomly). Some screenshots of MiniWoB problems are shown in the following figure:

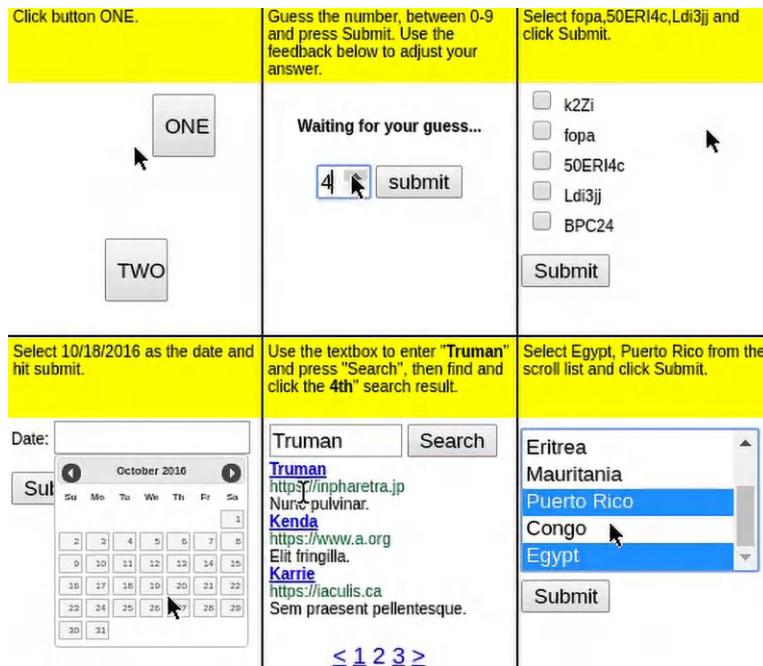


Figure 14.1: MiniWoB environments

Unfortunately, despite the brilliant idea and the challenging nature of MiniWoB, it was almost abandoned by OpenAI right after the initial release. Several years later, a group of Stanford researchers released an updated version called **MiniWoB++**, which had more games and a reworked architecture.

MiniWoB++

Instead of using VNC protocol and a real web browser, MiniWoB++ uses the Selenium (<https://www.selenium.dev>) library for web browser automation, which has significantly increased the performance and stability of the environment.

Currently, MiniWoB++ is being maintained by the Farama Foundation (<https://miniwob.farama.org/>), which is really great news for the RL community. In this chapter, we'll use their latest version, but before jumping into the RL part of the agent, we need to understand how MiniWoB++ works.

Installation

The original MiniWoB used VNC and OpenAI Universe, which created lots of complications during installation and usage. The previous edition of this book provided a custom Docker image with detailed installation instructions. Now, it is much simpler: you don't need to deal with Docker and VNC anymore. The Selenium library (which is a de facto standard in browser automation) hides all the complications of communicating with the browser, which is started in the background in headless mode. Selenium supports various browsers, but the MiniWoB++ developers recommend using Chrome or Chromium, as other browsers might render environments differently.

Besides the MiniWoB++ package (which can be installed with `pip install miniwob==1.0`), you will need `chromedriver` to be set up on your machine. ChromeDriver is a small binary that communicates with the browser and runs it in the "testing mode." The version of ChromeDriver has to match the installed version of Chrome (to check, go to **Chrome → About Google Chrome**), so please download the `chromedriver` archive for your platform and Chrome version from this website: <https://googlechromelabs.github.io/chrome-for-testing/>.

Be careful: besides the ChromeDriver archive, they also provide archives for the full version of Chrome, most likely you don't need it. For example, `chromedriver` for Chrome v123 on Mac M2 hardware will have this URL: <https://storage.googleapis.com/chrome-for-testing-public/123.0.6312.122/mac-arm64/chromedriver-mac-arm64.zip>. In the archive, a single `chromedriver` binary is present, which should be put somewhere in the `PATH` of your shell (on Mac and Linux machines, you can use the `which chromedriver` console command, which has to write the full path to the binary. If nothing is shown, you need to modify the `PATH`).

To test your installation, you can use a simple program, `Chapter14/adhoc/01_wob_create.py`. If everything is working, a browser window with a task will appear for 2 seconds.

Actions and observations

In contrast with Atari games and the other Gym environments that we have worked with so far, MiniWoB exposes a much more generic action space. Atari games used six or seven discrete actions corresponding to the controller's buttons and joystick directions. CartPole's action space is even smaller, with just two actions. However, the browser gives our agent much more flexibility in terms of what it can do. First, the full keyboard, with control keys and the up/down state of every key, is exposed. So, your agent can decide to press 10 buttons simultaneously and it will be totally fine from a MiniWoB point of view. The second part of the action space is the mouse: you can move the mouse to any coordinates and control the state of its buttons. This significantly increases the dimensionality of the action space that the agent needs to learn how to handle. In addition, the mouse allows double-clicking and mouse-wheel up/down scrolling events.

In terms of observation space, MiniWoB is also much richer than the environments we've dealt with so far. The full observation is represented as a dict with the following data:

- Text with a description of the task, like **Click button ONE or You are playing as X in TicTacToe, win the game**
- Screen's pixel as RGB values
- List of all DOM elements from the underlying web page with attributes (dimensions, colors, font, etc.)

Besides that, you can access the underlying browser to get even more information (to get some information that is not directly provided, like CSS attributes or raw HTML data).

As you can see, this set of tasks has lots of flexibility for experimentation: you can focus on the visual side of the task, working at the pixel level; you can use DOM information (the environment allows you to click on specific elements); or use NLP components – to understand the task description and plan the actions.

Simple example

To gain some practical experience with MiniWoB, let's take a look at the program you used to validate your installation, which you will find at `Chapter14/adhoc/01_wob_create.py`.

First, we need to register the MiniWoB environment in Gymnasium, which is done with the `register_envs()` function:

```
import time
import gymnasium as gym
import miniwob
from miniwob.action import ActionTypes

RENDER_ENV = True

if __name__ == "__main__":
    gym.register_envs(miniwob)
```

In fact, this `register_envs()` function does nothing, as all the environments are registered when the module is imported. But modern IDEs are smart enough to start complaining about unused modules, so this method creates the impression for the IDE that the module is being used in the code.

Then we create an environment using the standard `gym.make()` method:

```
env = gym.make('miniwob/click-test-2-v1', render_mode='human' if RENDER_ENV else
None)
print(env)
try:
    obs, info = env.reset()
    print("Obs keys:", list(obs.keys()))
    print("Info dict:", info)
    assert obs["utterance"] == "Click button ONE."
    assert obs["fields"] == (("target", "ONE"),)
    print("Screenshot shape:", obs['screenshot'].shape)
```

In our example, we're using the `click-test-2` problem, which asks you to click on one of two buttons randomly placed on the webpage. The Farama website contains a very convenient list of environments that you can play with yourself. The `click-test-2` problem is available here: <https://miniwob.farama.org/environments/click-test-2/>.

On environment creation, we passed the `render_mode` argument. If it equals 'human', then the browser window will be shown in the background. In *Figure 14.2*, you can see the window:

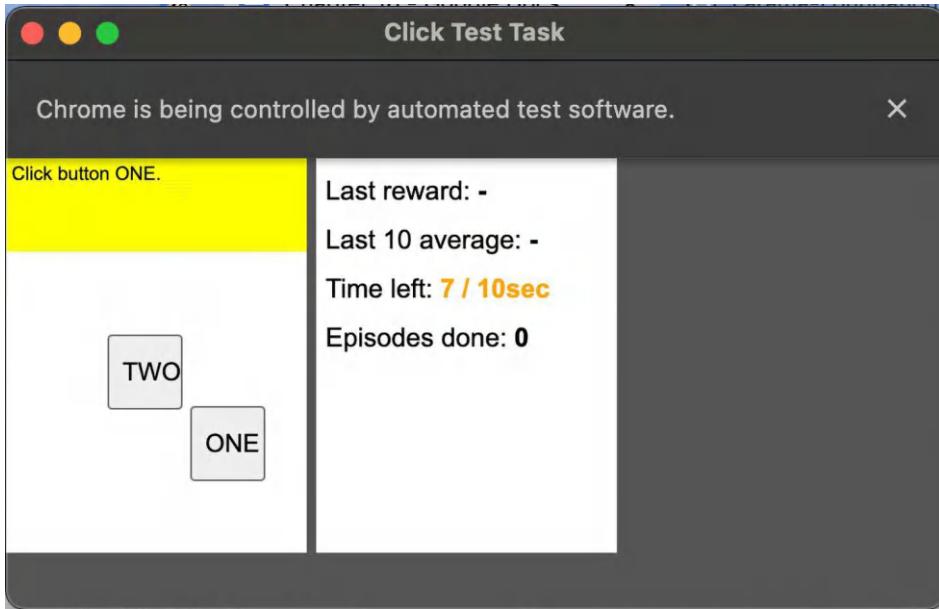


Figure 14.2: click-test-2 environment

If we run the program, it will show us the environment object and information about the observation (which is quite a large dict, so, I output just a list of its keys). The following is the part that is shown by the code:

```
$ python adhoc/01_wob_create.py
<OrderEnforcing<PassiveEnvChecker<ClickTest2Env<miniwob/click-test-2-v1>>>
Obs keys: ['utterance', 'dom_elements', 'screenshot', 'fields']
Info dict: {'done': False, 'env_reward': 0, 'raw_reward': 0, 'reason': None, 'root_dom':
[1] body @ (0, 0) classes=[] children=1}
Screenshot shape: (210, 160, 3)
```

As you can see, we have `utterance` (which is a task to be performed), DOM elements, `screenshot` with exactly the same dimensions as the Atari platform (I don't think this is just a coincidence!), and a list of `fields`, which are task-specific important elements in the DOM tree.

Now, let's go back to our code. The following snippet finds the element in the `dom_elements` list that we have to click on to perform the task:

```

if RENDER_ENV:
    time.sleep(2)

target_elems = [e for e in obs['dom_elements'] if e['text'] == "ONE"]
assert target_elems
print("Target elem:", target_elems[0])

```

The code is iterating over the `dom_elements` observation's field, filtering elements that have the text `ONE`. The element that is found has quite a rich set of attributes:

```

Target elem: {'ref': 4, 'parent': 3, 'left': array([80.], dtype=float32), 'top': array([134.], dtype=float32), 'width': array([40.], dtype=float32), 'height': array([40.], dtype=float32), 'tag': 'button', 'text': 'ONE', 'value': '', 'id': 'subbtn', 'classes': '', 'bg_color': array([0.9372549, 0.9372549, 0.9372549, 1.        ], dtype=float32), 'fg_color': array([0., 0., 0., 1.], dtype=float32), 'flags': array([0, 0, 0, 1], dtype=int8)}

```

Now, let's look at the final piece of the code, where we take the reference of the element (which is an integer identifier) and create the `CLICK_ELEMENT` action:

```

action = env.unwrapped.create_action(
    ActionTypes.CLICK_ELEMENT, ref=target_elems[0]["ref"])
obs, reward, terminated, truncated, info = env.step(action)
print(reward, terminated, info)
finally:
    env.close()

```

As we have already mentioned, MiniWoB provides a rich set of actions to be executed. This particular one emulates a mouse click on a specific DOM element.

As a result of this action, we should get a reward, which in fact does happen:

```

0.7936 True {'done': True, 'env_reward': 0.7936, 'raw_reward': 1, 'reason': None,
'elapsed': 2.066638231277466}

```

If you disable rendering with `RENDER_ENV = False`, everything that happens in the console and the browser won't be shown. This mode will also lead to a higher reward, as the reward decreases with time. Full headless mode on my machine obtains a reward of 0.9918 in 0.09 seconds.

The simple clicking approach

To get started with web navigation, let's implement a simple A3C agent that decides where it should click given the image observation. This approach can solve only a small subset of the full MiniWoB suite, and we will discuss the restrictions of this approach later. For now, it will allow us to get a better understanding of the problem.

As with the previous chapter, I won't discuss the complete source code here. Instead, we will focus on the most important functions and I will provide a brief overview of the rest. The complete source code is available in the GitHub repository.

Grid actions

When we talked about MiniWoB architecture and organization, we mentioned that the richness and flexibility of the action space creates a lot of challenges for the RL agent. The active area inside the browser is just 210×160 pixels, but even with such a small area, our agent could be asked to move the mouse, perform clicks, drag objects, and so on. Just the mouse alone could be problematic to master, as, in the extreme case, there could be an almost infinite number of different actions that the agent could perform, like pressing the mouse button at some point and dragging the mouse to a different location. In our example, we will simplify our problem a lot by just considering clicks at some fixed grid points inside the active webpage area. The sketch of our action space is shown in the following figure:

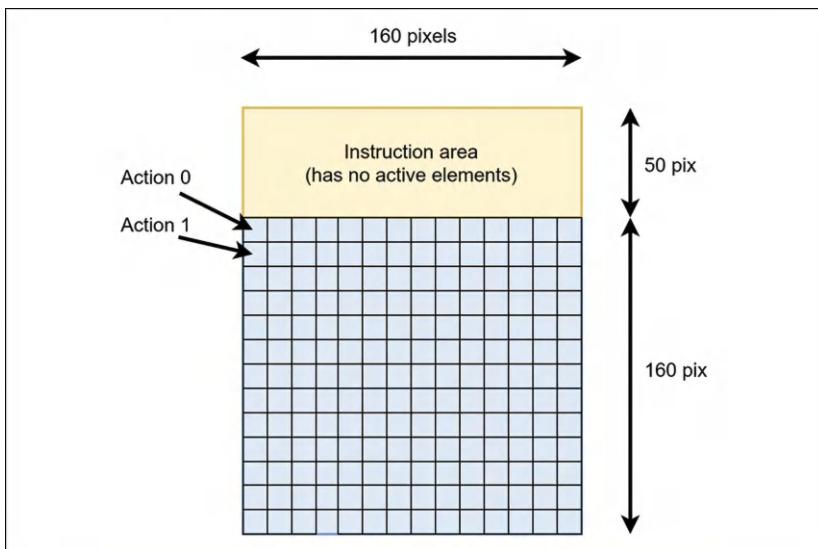


Figure 14.3: A grid action space

In the original version of MiniWob, the wrapper for such actions was already present in OpenAI Universe. But since it is not available for MiniWoB++, I implemented it myself in the `lib/wob.py` module. Let's quickly check the code, starting with the constructor:

```
WIDTH = 160
HEIGHT = 210
X_OFST = 0
Y_OFST = 50
BIN_SIZE = 10
WOB_SHAPE = (3, HEIGHT, WIDTH)

class MiniWoBClickWrapper(gym.ObservationWrapper):
    FULL_OBS_KEY = "full_obs"

    def __init__(self, env: gym.Env, keep_text: bool = False,
                 keep_obs: bool = False, bin_size: int = BIN_SIZE):
        super(MiniWoBClickWrapper, self).__init__(env)
        self.bin_size = bin_size
        self.keep_text = keep_text
        self.keep_obs = keep_obs
        img_space = spaces.Box(low=0, high=255, shape=WOB_SHAPE, dtype=np.uint8)
        if keep_text:
            self.observation_space = spaces.Tuple(
                (img_space, spaces.Text(max_length=1024)))
        else:
            self.observation_space = img_space
        self.x_bins = WIDTH // bin_size
        count = self.x_bins * ((HEIGHT - Y_OFST) // bin_size)
        self.action_space = spaces.Discrete(count)
```

In the constructor, we create the observation space (which is a tensor of $3 \times 210 \times 160$) and the action space, which will be 256 discrete actions for a bin size of 10. As an option, we can ask the wrapper to preserve the text of the task to be performed. This functionality will be used in subsequent examples in the chapter.

Then we provide a class method to create the environment with a specific configuration:

```
@classmethod
def create(cls, env_name: str, bin_size: int = BIN_SIZE, keep_text: bool = False,
           keep_obs: bool = False, **kwargs) -> "MiniWoBClickWrapper":
    gym.register_envs(miniwob)
    x_bins = WIDTH // bin_size
    y_bins = (HEIGHT - Y_OFST) // bin_size
    act_cfg = ActionSpaceConfig(
        action_types=(ActionTypes.CLICK_COORDS, ), coord_bins=(x_bins, y_bins))
```

```

env = gym.make(env_name, action_space_config=act_cfg, **kwargs)
return MiniWoBClickWrapper(
    env, keep_text=keep_text, keep_obs=keep_obs, bin_size=bin_size)

```

Besides just creating the environment and wrapping it, we're asking for a custom ActionSpaceConfig, which will take into account our grid's dimensions. With this customization, we will need to pass the (x, y) coordinates of the grid cell to perform the click action. Then, we define a helper method, which converts the full observation dict into the format we need. The `reset()` method is just calling this method:

```

def _observation(self, observation: dict) -> np.ndarray | tt.Tuple[np.ndarray, str]:
    text = observation['utterance']
    scr = observation['screenshot']
    scr = np.transpose(scr, (2, 0, 1))
    if self.keep_text:
        return scr, text
    return scr

def reset(self, *, seed: int | None = None, options: dict[str, tt.Any] | None = None)
\      -> tuple[gym.core.WrapperObsType, dict[str, tt.Any]]:
    obs, info = self.env.reset(seed=seed, options=options)
    if self.keep_obs:
        info[self.FULL_OBS_KEY] = obs
    return self._observation(obs), info

```

Now, the final piece of the wrapper, the `step()` method:

```

def step(self, action: int) -> tt.Tuple[
    gym.core.WrapperObsType, gym.core.SupportsFloat, bool, bool, dict[str, tt.Any]
]:
    b_x, b_y = action_to_bins(action, self.bin_size)
    new_act = {
        "action_type": 0,
        "coords": np.array((b_x, b_y), dtype=np.int8),
    }
    obs, reward, is_done, is_tr, info = self.env.step(new_act)
    if self.keep_obs:
        info[self.FULL_OBS_KEY] = obs
    return self._observation(obs), reward, is_done, is_tr, info

def action_to_bins(action: int, bin_size: int = BIN_SIZE) -> tt.Tuple[int, int]:
    row_bins = WIDTH // bin_size

```

```
b_y = action // row_bins  
b_x = action % row_bins  
return b_x, b_y
```

To perform the action, we need to convert the index of the grid cell index (in the 0 ... 255 range) into the (x, y) coordinates of the cell. Then, as an action for the underlying MiniWoB environment, we pass a dict with `action_type=0` (which is an index in `ActionSpaceConfig` we used in the environment creation) and a NumPy array with those cell coordinates.

To illustrate the wrapper, there is a small program in the GitHub repository in the `adhoc/03_clicker.py` file, which uses a brute force approach on the `click-dialog-v1` task. The goal is to close the randomly placed dialog using the corner button with the cross. In this example (we're not showing the code here), we sequentially click through all the 256 grid cells to illustrate the wrapper.

The RL part of our implementation

With the transformation of observation and actions, the RL part is quite straightforward. We will use the A3C method to train the agent, which should decide from the 160×210 observation which grid cell to click on. Besides the policy, which is a probability distribution over 256 grid cells, our agent estimates the value of the state, which will be used as a baseline in policy gradient estimation.

There are several modules in this example:

- `lib/common.py`: Methods shared among examples in this chapter, including the already familiar `RewardTracker` and `unpack_batch` functions
- `lib/model.py`: Includes a definition of the model, which we'll take a look at in the next section
- `lib/wob.py`: Includes MiniWoB-specific code, like environment wrappers and other utility functions
- `wob_click_train.py`: The script used to train the clicker model
- `wob_click_play.py`: The script that loads the model weights and uses them against the single environment, recording observations and counting statistics about the reward

There is nothing new in the code in these modules, so it is not shown here. You can find it in the GitHub repository.

The model and training code

The model is very straightforward and uses the same patterns that you have seen in other A3C examples. I haven't spent much time optimizing and fine-tuning the architecture and hyperparameters, so it's likely that the final result could be improved significantly (you can try doing this yourself based on what you've learned in this book so far). The following is the model definition with two convolution layers, a single-layered policy, and value heads:

```
class Model(nn.Module):
    def __init__(self, input_shape: tt.Tuple[int, ...], n_actions: int):
        super(Model, self).__init__()

        self.conv = nn.Sequential(
            nn.Conv2d(input_shape[0], 64, 5, stride=5),
            nn.ReLU(),
            nn.Conv2d(64, 64, 3, stride=2),
            nn.ReLU(),
            nn.Flatten(),
        )
        size = self.conv(torch.zeros(1, *input_shape)).size()[-1]
        self.policy = nn.Linear(size, n_actions)
        self.value = nn.Linear(size, 1)

    def forward(self, x: torch.ByteTensor) -> tt.Tuple[torch.Tensor, torch.Tensor]:
        xx = x / 255.0
        conv_out = self.conv(xx)
        return self.policy(conv_out), self.value(conv_out)
```

You'll find the training script in `wob_click_train.py`, and it is exactly the same as in *Chapter 12*. We're using `AsyncVectorEnv` with 8 parallel environments, which starts 8 Chrome instances in the background. If your machine's memory allows, you can increase this count and check the effect on the training.

Training results

By default, the training uses the `click-dialog-v1` problem, and it took about 8 minutes of training to reach an average reward of 0.9. *Figure 14.4* shows the plots with average reward and number of steps:

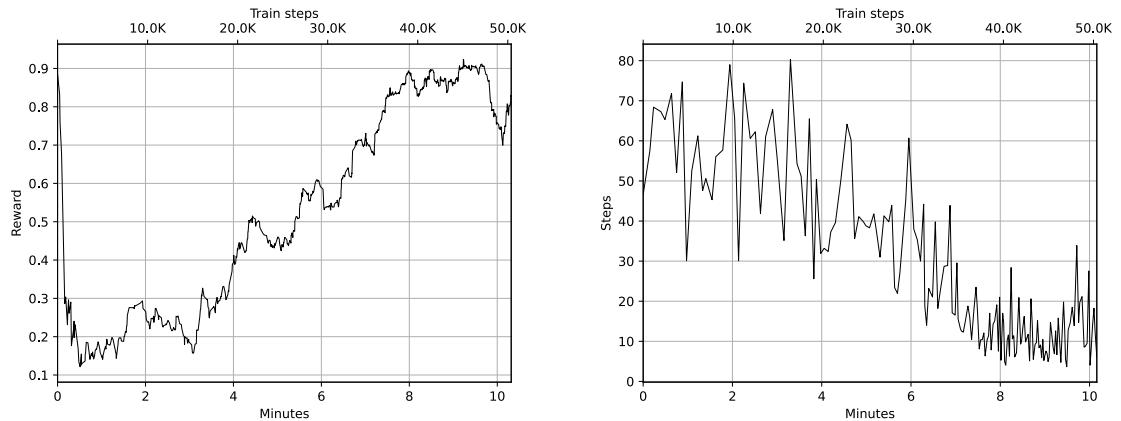


Figure 14.4: Training reward (left) and count of steps in episodes (right)

The `episode_steps` chart on the right shows the mean count of actions that the agent should carry out before the end of the episode. Ideally, for this problem, the count should be 1, as the only action that the agent needs to take is to click on the dialog’s close button. However, in fact, the agent sees seven to nine frames before the episode ends. This happens for two reasons: the cross on the dialog close button may appear after some delay, and the browser inside the container adds a time gap before the agent clicks and the reward is obtained.

To check the learned policy, you can use the `wob_click_play.py` tool, which loads the model and uses it in one environment. It can play several episodes to test the average model performance:

```
$ ./wob_click_play.py -m saves/best_0.923_45400.dat --verbose
0 0.0 False {'done': False, 'env_reward': 0, 'raw_reward': 0, 'reason': None, 'elapsed': 0.1620042324066162, 'root_dom': [1] body @ (0, 0) classes=[] children=2}
1 0.9788 True {'done': True, 'env_reward': 0.9788, 'raw_reward': 1, 'reason': None, 'elapsed': 0.19491100311279297}
Round 0 done
Done 1 rounds, mean steps 2.00, mean reward 0.979
```

If this begins with the `-render` command-line option, the browser window will be shown during the agent’s actions.

Simple clicking limitations

Unfortunately, the demonstrated approach can only be used to solve relatively simple problems, like `click-dialog`. If you try to use it for more complicated tasks, convergence is unlikely. There are several reasons for this.

First, our agent is stateless, which means that it makes the decisions about actions only from observations, without taking into account its previous actions. You may remember that in *Chapter 1*, we discussed the Markov property of the **Markov decision process (MDP)** and that this Markov property allowed us to discard all previous history, keeping only the current observation. Even in relatively simple problems from MiniWoB, this Markov property could be violated. For example, there is a problem called `click-button-sequence` (the screenshot is shown in *Figure 14.5*, and documentation for this environment is available at <https://miniwob.farama.org/environments/click-button-sequence/>), which requires our agent to first click on button ONE and then on button TWO. Even if our agent is lucky enough to randomly click on the buttons in the required order, it won't be able to distinguish from the single image which button needs to be clicked on next.

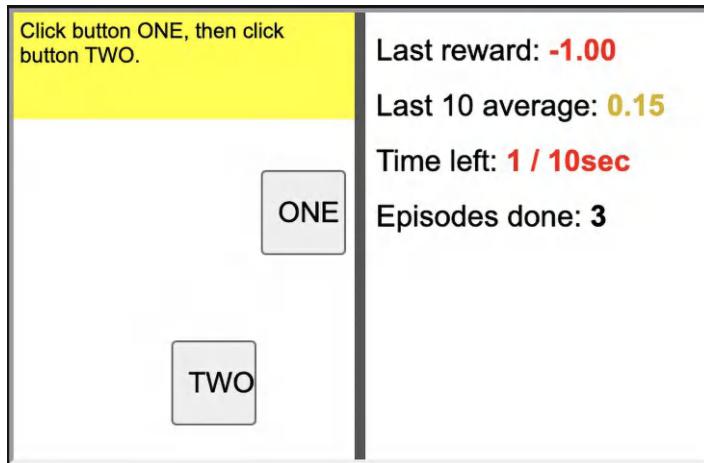


Figure 14.5: An example of an environment that the stateless agent could struggle to solve

Despite the simplicity of this problem, we cannot use our RL methods to solve it, because MDP formalism is not applicable anymore. Such problems are called **partially observable MDPs**, or POMDPs (we briefly discussed these in *Chapter 6*), and the usual approach for them is to allow the agent to keep some kind of state. The challenge here is to find the balance between keeping only minimal relevant information and overwhelming the agent with non-relevant information by adding everything into the observation.

Another issue that we can face with our example is that the data required to solve the problem might not be available in the image or could be in an inconvenient form. For example, two problems, `click-tab` (<https://miniwob.farama.org/environments/click-tab/>) and `click-checkboxes` (<https://miniwob.farama.org/environments/click-checkboxes/>), are shown in *Figure 14.6*:

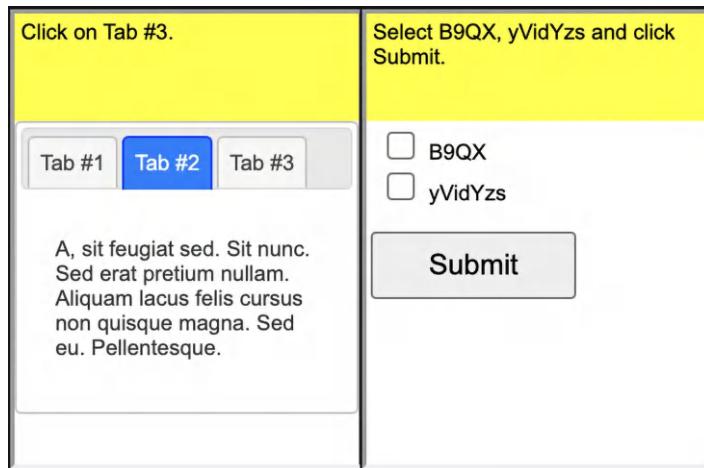


Figure 14.6: An example of environments where the text description is important

In the first one, you need to click on one of three tabs, but every time, the tab that needs to be clicked is randomly chosen. Which tab needs to be clicked is shown in a description (provided with an in-text field of observation and shown at the top of the environment's page), but our agent sees only pixels, which makes it complicated to connect the tiny number at the top with the outcome of the random click result. The situation is even worse with the `click-checkboxes` problem, when several checkboxes with randomly generated text need to be clicked. One of the possible ways to prevent overfitting to the problem is to use some kind of **optical character recognition (OCR)** network to convert the image in the observation into text form. Another approach (which will be shown in the next section) is to mix the text description into the agent's observations.

Yet another issue could be related to the dimensionality of the action space that the agent needs to explore. Even for single-click problems, the number of actions could be very large, so it can take a long time for the agent to discover how to behave. One of the possible solutions here is incorporating demonstrations into the training. For example, in *Figure 14.7*, there is a problem called `count-sides` (<https://miniwob.farama.org/environments/count-sides/>).

The goal there is to click on the button that corresponds to the number of sides of the shape shown:

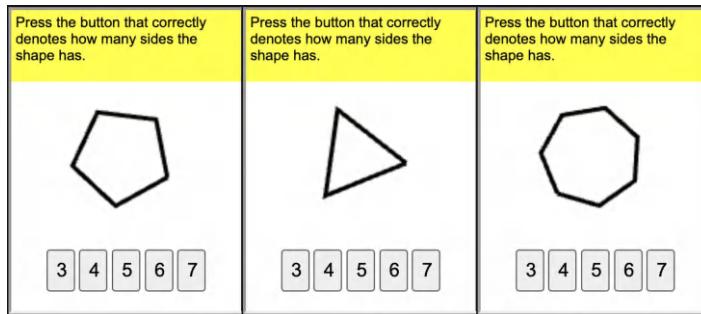


Figure 14.7: Examples of the count-sides environment

This issue was addressed by adding human demonstrations into the training. In my experiments, training from scratch gave zero progress after a day of training. However, after adding a couple of dozen examples of correct clicks, it successfully solved the problem in 15 minutes of training. Of course, we could spend time fine-tuning the hyperparameters further, but still, the effect of the demonstrations is quite impressive. Later in this chapter, we will take a look at how we can record and inject human demonstrations to improve convergence.

Adding text description

As a first step to improve our clicker agent, we'll add the text description of the problem into the model. I have already mentioned that some problems contain vital information that is provided in a text description, like the index of tabs that need to be clicked or the list of entries that the agent needs to check. The same information is shown at the top of the image observation, but pixels are not always the best representation of simple text.

To take this text into account, we need to extend our model's input from an image only to an image and text data. We worked with text in the previous chapter, so a **recurrent neural network (RNN)** is quite an obvious choice (maybe not the best for such a toy problem, but it is flexible and scalable).

Implementation

In this section, we will just focus on the most important points of the implementation. You will find the whole code in the `Chapter16/wob_click_mm_train.py` module. In comparison to our clicker model, a text extension doesn't add too much.

First, we should ask `MiniWoBClickWrapper` to keep the text obtained from the observation. The complete source code of this class was shown earlier in this chapter, in the *Grid actions* section. To keep the text, we should pass `keep_text=True` to the wrapper constructor, which makes this class return a tuple with a NumPy array and text string, instead of just a NumPy array with the image.

Then, we need to prepare our model to be able to process such tuples instead of a batch of NumPy arrays. This needs to be done in two places: in our agent (when we use the model to choose the action) and in the training code. To adapt the observation in a model-friendly way, we can use a special functionality of the PTAN library, called `preprocessor`. The core idea is very simple: `preprocessor` is a callable function that needs to convert the list of observations to a form that is ready to be passed to the model. By default, `preprocessor` converts the list of NumPy arrays into a PyTorch tensor and, optionally, copies it into GPU memory. However, sometimes, more sophisticated transformations are required, like in our case, when we need to pack the images into the tensor, but text strings require special handling. In that case, you can redefine the default `preprocessor` and pass it into the `ptan.Agent` class.

In theory, the `preprocessor` functionality could be moved into the model itself, thanks to PyTorch's flexibility, but the default `preprocessor` simplifies our lives in cases when observations are just NumPy arrays. The following is the `preprocessor` class source code taken from the `lib/model.py` module:

```
MM_EMBEDDINGS_DIM = 50
MM_HIDDEN_SIZE = 128
MM_MAX_DICT_SIZE = 100

TOKEN_UNK = "#unk"

class MultimodalPreprocessor:
    log = logging.getLogger("MultimodalPreprocessor")

    def __init__(self, max_dict_size: int = MM_MAX_DICT_SIZE,
                 device: torch.device = torch.device('cpu')):
        self.max_dict_size = max_dict_size
        self.token_to_id = {TOKEN_UNK: 0}
        self.next_id = 1
        self.tokenizer = TweetTokenizer(preserve_case=True)
        self.device = device

    def __len__(self):
        return len(self.token_to_id)
```

In the constructor in the preceding code, we create a mapping from the token to the identifier (which will be dynamically extended) and create the tokenizer from the `nltk` package.

Next, we have the `__call__()` method, which transforms the batch:

```
def __call__(self, batch: tt.Tuple[tt.Any, ...] | tt.List[tt.Tuple[tt.Any, ...]]):
    tokens_batch = []
```

```

if isinstance(batch, tuple):
    batch_iter = zip(*batch)
else:
    batch_iter = batch
for img_obs, txt_obs in batch_iter:
    tokens = self.tokenizer.tokenize(txt_obs)
    idx_obs = self.tokens_to_idx(tokens)
    tokens_batch.append((img_obs, idx_obs))
tokens_batch.sort(key=lambda p: len(p[1]), reverse=True)
img_batch, seq_batch = zip(*tokens_batch)
lens = list(map(len, seq_batch))

```

The goal of our preprocessor is to convert a batch of (image, text) tuples into two objects: the first has to be a tensor with the image data of shape (batch_size, 3, 210, 160), and the second has to contain the batch of tokens from text descriptions in the form of a packed sequence. The packed sequence is a PyTorch data structure suitable for efficient processing with an RNN. We discussed this in *Chapter 13*.

In fact, the batch could have two different forms: it could be a tuple with an image batch and a text batch, or it could be a list of tuples with individual (image, text tokens) samples. This happens because of the difference in `VectorEnv` handling of `gym.Tuple` observation space. But those details are not very relevant here; we just handle the difference by checking the type of the `batch` variable and performing the necessary processing.

As the first step of our transformation, we tokenize text strings and convert every token into the list of integer IDs. Then, we sort our batch by decreasing the token length, which is a requirement of the underlying cuDNN library for efficient RNN processing.

Then, we convert the images into a tensor and the sequences into a padded sequence, which is a matrix of batch size \times the length of the longest sequence. We saw this in the previous chapter:

```

img_v = torch.FloatTensor(np.asarray(img_batch)).to(self.device)
seq_arr = np.zeros(
    shape=(len(seq_batch), max(len(seq_batch[0]), 1)), dtype=np.int64)
for idx, seq in enumerate(seq_batch):
    seq_arr[idx, :len(seq)] = seq
    if len(seq) == 0:
        lens[idx] = 1
seq_v = torch.LongTensor(seq_arr).to(self.device)
seq_p = rnn_utils.pack_padded_sequence(seq_v, lens, batch_first=True)
return img_v, seq_p

```

The following `tokens_to_idx()` function converts the list of tokens into a list of IDs:

```

def tokens_to_idx(self, tokens):
    res = []
    for token in tokens:
        idx = self.token_to_id.get(token)
        if idx is None:
            if self.next_id == self.max_dict_size:
                self.log.warning("Maximum size of dict reached, token "
                                 "'%s' converted to #UNK token", token)
            idx = 0
        else:
            idx = self.next_id
            self.next_id += 1
            self.token_to_id[token] = idx
    res.append(idx)
return res

```

The tricky thing is that we don't know in advance the size of the dictionary from the text descriptions. One approach would be to work on the character level and feed individual characters into the RNN, but it would result in sequences that are too long to process. The alternative solution is to hard-code a reasonable dictionary size, say 100 tokens, and dynamically assign token IDs to tokens that we have never seen before. In this implementation, the latter approach is used, but it might not be applicable to MiniWoB problems that contain randomly generated strings in the text description. As potential solutions for this issue, we can either use character-level tokenization or use a pre-defined dictionary.

Now, let's take a look at our model class:

```

class ModelMultimodal(nn.Module):
    def __init__(self, input_shape: tt.Tuple[int, ...], n_actions: int,
                 max_dict_size: int = MM_MAX_DICT_SIZE):
        super(ModelMultimodal, self).__init__()

        self.conv = nn.Sequential(
            nn.Conv2d(input_shape[0], 64, 5, stride=5),
            nn.ReLU(),
            nn.Conv2d(64, 64, 3, stride=2),
            nn.ReLU(),
            nn.Flatten(),
        )
        size = self.conv(torch.zeros(1, *input_shape)).size()[-1]

        self.emb = nn.Embedding(max_dict_size, MM_EMBEDDINGS_DIM)
        self.rnn = nn.LSTM(MM_EMBEDDINGS_DIM, MM_HIDDEN_SIZE, batch_first=True)
        self.policy = nn.Linear(size + MM_HIDDEN_SIZE*2, n_actions)

```

```
self.value = nn.Linear(size + MM_HIDDEN_SIZE*2, 1)
```

The difference is in a new embedding layer, which converts integer token IDs into dense token vectors and a **long short-term memory (LSTM)** RNN. The outputs from the convolution and RNN layers are concatenated and fed into the policy and value heads, so the dimensionality of their input is the image and text features combined.

This function performs the concatenation of the image and RNN features into a single tensor:

```
def _concat_features(self, img_out: torch.Tensor,
                     rnn_hidden: torch.Tensor | tt.Tuple[torch.Tensor, ...]):
    batch_size = img_out.size()[0]
    if isinstance(rnn_hidden, tuple):
        flat_h = list(map(lambda t: t.view(batch_size, -1), rnn_hidden))
        rnn_h = torch.cat(flat_h, dim=1)
    else:
        rnn_h = rnn_hidden.view(batch_size, -1)
    return torch.cat((img_out, rnn_h), dim=1)
```

Finally, in the `forward()` function, we expect two objects prepared by the preprocessor: a tensor with input images and packed sequences of the batch:

```
def forward(self, x: tt.Tuple[torch.Tensor, rnn_utils.PackedSequence]):
    x_img, x_text = x

    emb_out = self.emb(x_text.data)
    emb_out_seq = rnn_utils.PackedSequence(emb_out, x_text.batch_sizes)
    rnn_out, rnn_h = self.rnn(emb_out_seq)

    xx = x_img / 255.0
    conv_out = self.conv(xx)
    feats = self._concat_features(conv_out, rnn_h)
    return self.policy(feats), self.value(feats)
```

Images are processed with convolutions and text data is fed through the RNN; then, the results are concatenated, and the policy and value results are calculated.

That's most of the new code. The training Python script, `wob_click_mm_train.py`, is mostly a copy of `wob_click_train.py`, with just the small modifications in the wrapper creation, a different model, and preprocessor.

Results

I ran several experiments in the click-button environment (<https://miniwob.farama.org/environment/s/click-button/>) that have the goal of making a selection between several random buttons. In *Figure 14.8*, several situations in this environment are shown:

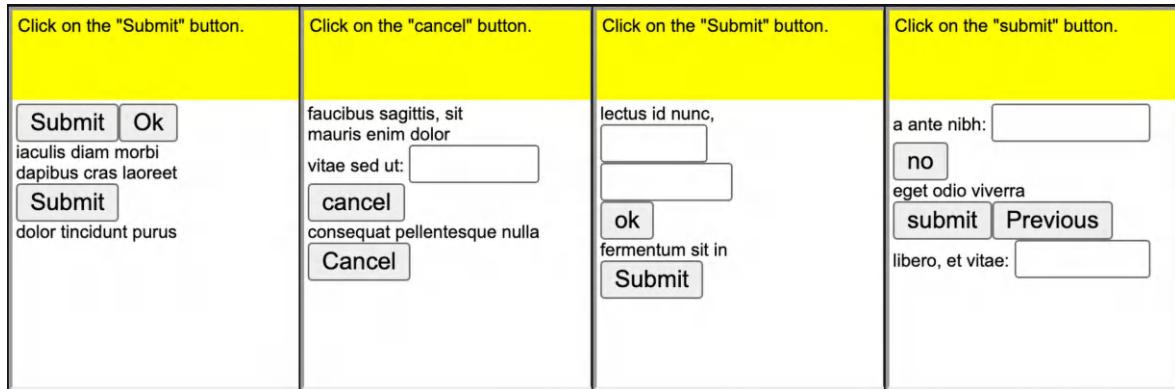


Figure 14.8: Tasks in the click-button environment

As shown in *Figure 14.9*, after 3 hours of training, the model was able to learn how to click (the average count of steps in episodes was reduced to 5-7) and get to an average reward of 0.2. But subsequent training had no visible effect. It might be an indication that the hyperparameters have to be tuned, or of the ambiguity of the environment. In this case, I noticed that this environment sometimes shows several buttons with the same title, but only one of them gives a positive reward. An example of this is shown in the first section of *Figure 14.8*, where two identical **Submit** buttons are present.

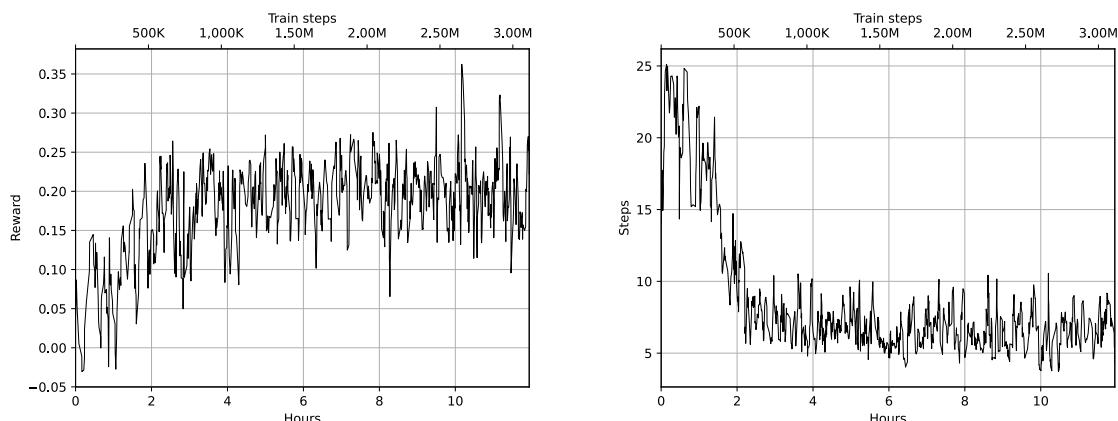


Figure 14.9: Training reward (left) and count of steps in episodes (right) on click-button

Another environment in which the text description is important is `click-tab`, which demands the agent to click on a specific tab, chosen randomly. Screenshots are shown in *Figure 14.10*.

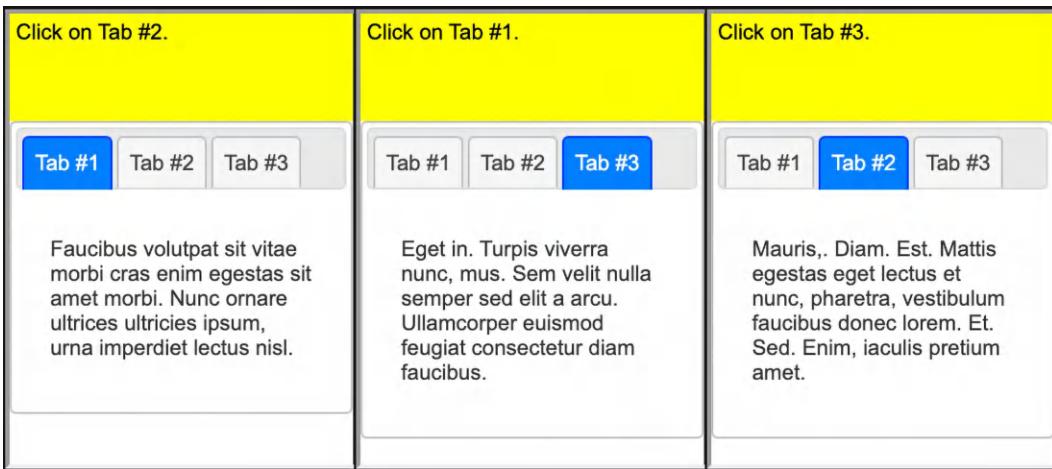


Figure 14.10: Tasks in the click-tab environment

In this environment, the training was not successful, which is a bit strange, as the task looks easier than click-button (the position of the place to click is fixed). Most likely, hyperparameter tuning is required. This is another interesting challenge that you can try to address through experimentation, using the knowledge you've gained so far.

Human demonstrations

In order to improve the training process, let's try to incorporate human demonstrations. The idea behind demonstrations is simple: to help our agent to discover the best way to solve the task, we show it some examples of actions that we think are required for the problem. Those examples might not be the best solution or not 100% accurate, but they should be good enough to show the agent promising directions to explore.

In fact, this is a very natural thing to do, as all human learning is based on some prior examples given by a teacher in class, parents, or other people. Those examples could be in a written form (for example, recipe books) or given as demonstrations that you need to repeat several times to get right (for example, dance classes). Such forms of training are much more effective than random searches. Just imagine how complicated and lengthy it would be to learn how to clean your teeth by trial and error alone. Of course, there is a danger from learning how to follow demonstrations, which could be wrong or not the most efficient way to solve the problem; but overall, it's much more effective than a random search.

All our previous examples followed this workflow:

1. They used zero prior knowledge and started with random weight initializations, which caused random actions to be performed at the beginning of the training.
2. After some iterations, the agent discovered that some actions in some states give more promising results (via the Q-value or policy with the higher advantage) and started to prefer those actions over the others.
3. Finally, this process led to a more or less optimal policy, which gave the agent a high reward at the end.

This worked well when our action space dimensionality was low and the environment's behavior wasn't very complex, but just doubling the action count caused at least twice the observations needed. In the case of our clicker agent, we have 256 different actions corresponding to 10×10 grids in the active area, which is 128 times more actions than we had in the CartPole environment. It is not surprising that the training process is lengthy and may fail to converge at all.

This issue of dimensionality can be addressed in various ways, like smarter exploration methods, training with better sampling efficiency (one-shot training), incorporating prior knowledge (transfer learning), and other means. There is a lot of research activity focused on making RL better and faster, and we can be sure that many breakthroughs are ahead. In this section, we will try the more traditional approach of incorporating the demonstration recorded by humans into the training process.

You might remember our discussion about on-policy and off-policy methods (which were discussed in *Chapter 4* and *Chapter 8*). This is very relevant to our human demonstrations because, strictly speaking, we can't use off-policy data (human observation-action pairs) with an on-policy method (A3C in our case). That is due to the nature of on-policy methods: they estimate the policy gradients using the samples gathered from the current policy. If we just push human-recorded samples into the training process, the estimated gradient will be relevant for a human policy, but not our current policy given by the **neural network (NN)**. To solve this issue, we need to cheat a bit and look at our problem from the supervised learning angle. To be concrete, we will use the log-likelihood objective to push our NN toward taking actions based on demonstrations.

With this, we're not replacing RL with supervised learning. Rather, we're reusing supervised learning techniques to help our RL methods. Fundamentally, this isn't the first time we've done something similar; for instance, the training of the value function in Q-learning is purely supervised learning.

Before we can go into the implementation details, we need to address a very important question: how do we obtain the demonstrations in the most convenient form?

Recording the demonstrations

Before MiniWoB++ and the transition to Selenium, recording a demonstration was technically challenging. In particular, the VNC protocol has to be captured and decoded to be able to extract screenshots of the browser and the actions executed by the user.

In the previous edition of the book, I provided my own version of the VNC protocol parser to record the demonstrations.

Luckily, those challenges are mostly gone now. There is no VNC anymore and the browser has been started in the local process (before, it was inside the Docker container), so we can communicate with it almost directly.

Farama MiniWoB++ is shipped with a Python script that can capture the demonstrations in a JSON file. This script can be started with the `python -m miniwob.scripts.record` command and is documented at <https://miniwob.farama.org/content/demonstrations/>.

Unfortunately, it has a limitation: in observations, it captures only the DOM structure of the webpage and has no pixel-level information. As examples in this chapter make heavy use of pixels, demonstrations recorded by this script are useless. To overcome this, I implemented my own version of a tool to record demonstrations that include pixels from the browser. It is called `Chapter14/record_demo.py` and can be started as follows:

```
$ ./record_demo.py -o demos/test -g tic-tac-toe-v1 -d 1
Bottle v0.12.25 server starting up (using WSGIRefServer())...
Listening on http://localhost:8032/
Hit Ctrl-C to quit.

WARNING:root:Cannot call {'action_type': 0} on instance 0, which is already done
127.0.0.1 - - [26/Apr/2024 12:19:49] "POST /record HTTP/1.1" 200 17
Saved in demos/test/tic-tac-toe_0426101949.json
New episode starts in 1 seconds...
```

This command starts the environment with `render_mode='human'`, which shows the browser window and allows you to communicate with the page. In the background, it records the observations (with screenshots) and, when the episode is done, it joins screenshots to your actions and stores everything in a JSON file in the directory given by the `-o` command-line option. Using the `-g` command-line option allows you to change the environment, and the `-d` parameter sets the delay in seconds between the episodes. If the `-d` option is not given, you need to press *Enter* in the console to start a new episode. The following screenshot shows the process of recording a demonstration:

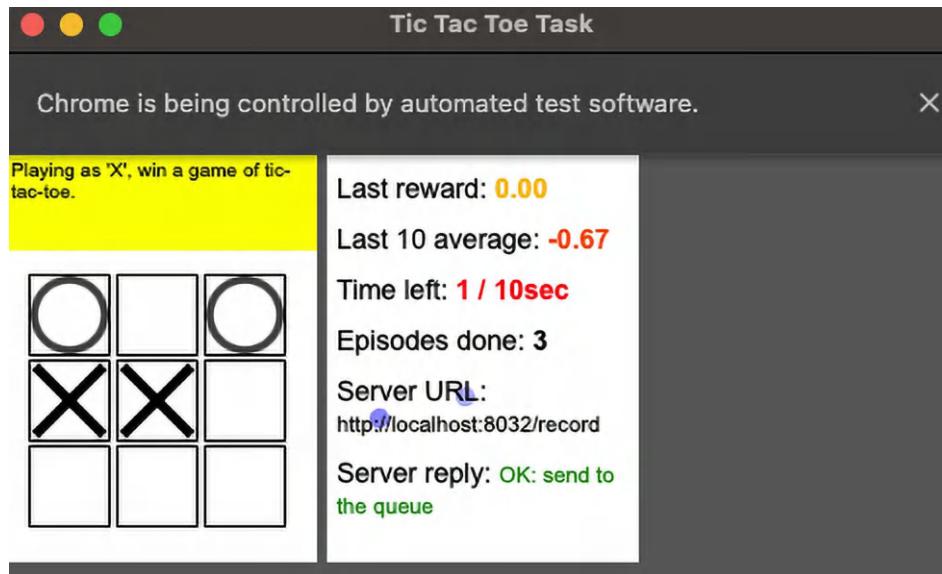


Figure 14.11: Recording a human demonstration for tic-tac-toe

In the `Chapter14/demos` directory, I stored the demonstrations used for experiments, but you, of course, can record your own demonstrations using the provided script.

Training with demonstrations

Now that we know how to record the demonstration data, we have only one question unanswered: how does our training process need to be modified to incorporate human demonstrations? The simplest solution, which nevertheless works surprisingly well, is to use the log-likelihood objective that we used in training the cross-entropy method in *Chapter 4*. To do this, we need to look at our A3C model as a classification problem producing the classification of input observations in its policy head. In its simplest form, the value head will be left untouched, but, in fact, it won't be hard to train it: we know the rewards obtained during the demonstrations, so what is needed is to calculate the discounted reward from every observation until the end of the episode.

To check how it was implemented, let's look at the relevant code pieces in `Chapter16/wob_click_train.py`. First, we can pass the directory with the demonstration data by passing the `demo <DIR>` option in the command line. This will enable the branch shown in the following code block, where we load the demonstration samples from the specified directory. The `demos.load_demo_dir()` function automatically loads demonstrations from JSON files in the given directory and converts them into `ExperienceFirstLast` instances:

```
demo_samples = None
if args.demo:
    demo_samples = demos.load_demo_dir(args.demo, gamma=GAMMA, steps=REWARD_STEPS)
    print(f"Loaded {len(demo_samples)} demo samples")
```

The second piece of code relevant to demonstration training is inside the training loop and is executed before any normal batch. The training from demonstrations is performed with some probability (by default, it is 0.5) and specified by the DEMO_PROB hyperparameter:

```
if demo_samples and step_idx < DEMO_FRAMES:
    if random.random() < DEMO_PROB:
        random.shuffle(demo_samples)
        demo_batch = demo_samples[:BATCH_SIZE]
        model.train_demo(net, optimizer, demo_batch, writer,
                         step_idx, device=device)
```

The logic is simple: with probability DEMO_PROB, we sample BATCH_SIZE samples from our demonstration data and perform a round of training of our network on the data in the batch.

The actual training, which is very simple and straightforward, is performed by the `model.train_demo()` function:

```
def train_demo(net: Model, optimizer: torch.optim.Optimizer,
               batch: tt.List[ptan.experience.ExperienceFirstLast], writer, step_idx: int,
               preprocessor=ptan.agent.default_states_preprocessor,
               device: torch.device = torch.device("cpu")):
    batch_obs, batch_act = [], []
    for e in batch:
        batch_obs.append(e.state)
        batch_act.append(e.action)
    batch_v = preprocessor(batch_obs)
    if torch.is_tensor(batch_v):
        batch_v = batch_v.to(device)
    optimizer.zero_grad()
    ref_actions_v = torch.LongTensor(batch_act).to(device)
    policy_v = net(batch_v)[0]
    loss_v = F.cross_entropy(policy_v, ref_actions_v)
    loss_v.backward()
    optimizer.step()
    writer.add_scalar("demo_loss", loss_v.item(), step_idx)
```

We split our batch on the observation and the actions list, preprocess the observations to convert them into a PyTorch tensor, and place them on the GPU. We then ask our A3C network to return the policy and calculate the cross-entropy loss between the result and the desired actions. From an optimization point of view, we're pushing our network toward the actions taken in the demonstrations.

Results

To check the effect of demonstrations, I performed two sets of training on the count-sides problem with the same hyperparameters: one was done without demonstrations, and another used 25 demonstration episodes, which are available in the `demos/count-sides` directory.

The difference was dramatic. Training performed from scratch reached the best mean reward of -0.4 after 12 hours of training and 4 million frames without any significant improvement in the training dynamics. On the other hand, training with demonstrations was able to get to the average reward of 0.5 just after 30,000 training frames, which took 8 minutes. *Figure 14.12* shows the reward and the count of steps.

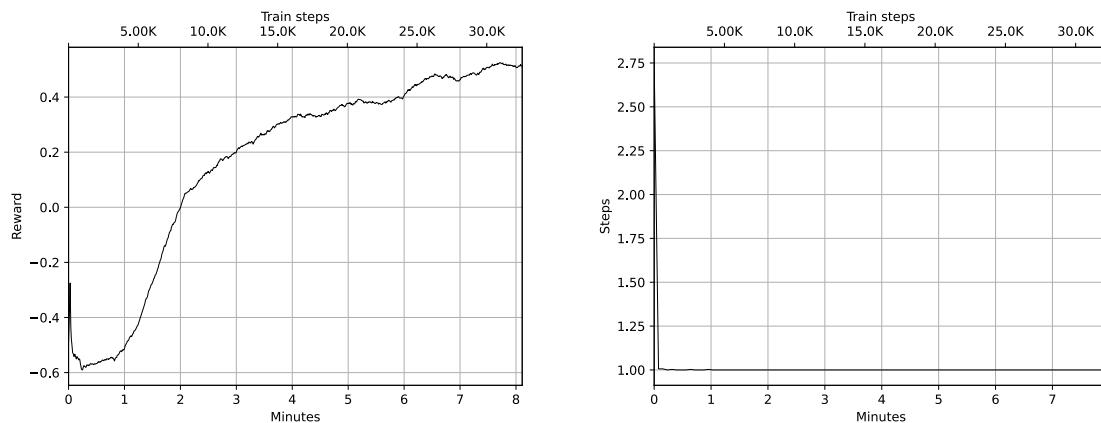


Figure 14.12: Training reward (left) and count of steps in episodes (right) on count-sides with demonstrations

A more challenging problem I experimented with is the Tic Tac Toe game, available as the `tic-tac-toe` environment.

Figure 14.13 shows the process of one of the demo games I recorded (available in the `demos/tic-tac-toe` directory). The dot shows where the click was performed:

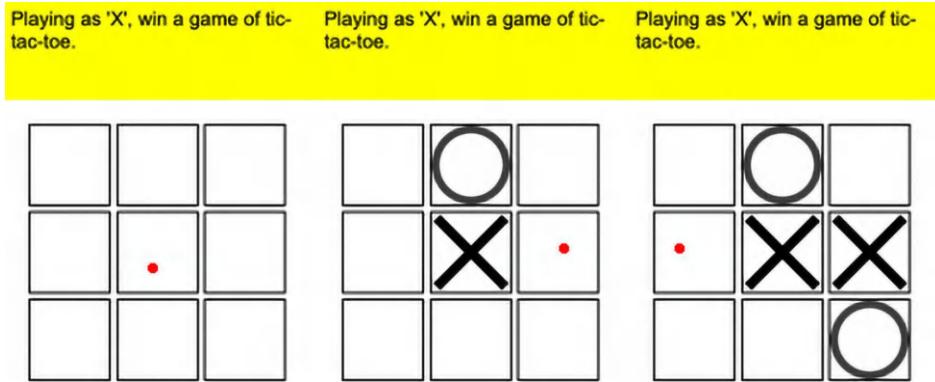


Figure 14.13: Demonstration TicTacToe game

After two hours of training, the best average reward reached was 0.05, which means that the agent can win some games, but some are lost or end up in a draw. In *Figure 14.14*, plots with reward dynamics and the count of episode steps are shown.

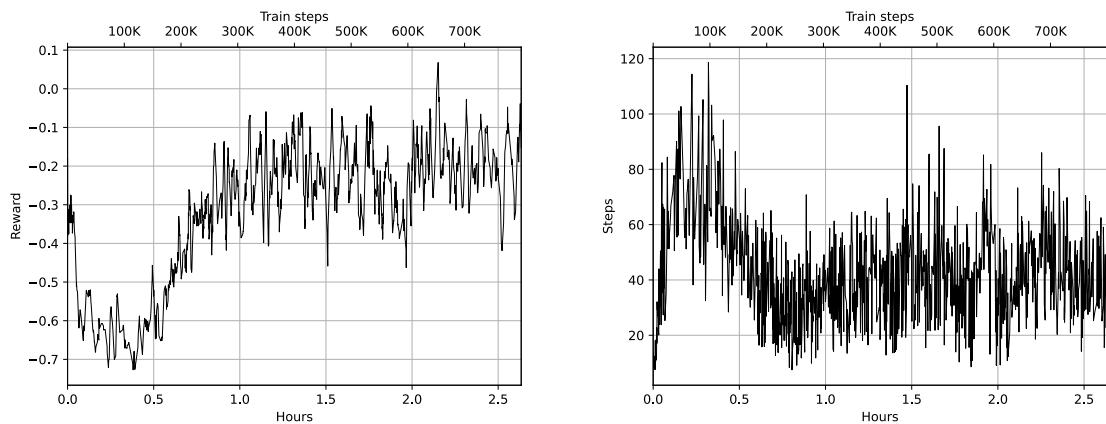


Figure 14.14: Training reward (left) and count of steps in episodes (right) on tic-tac-toe with demonstrations

Things to try

In this chapter, we only started playing with MiniWoB++ by looking at some of the easiest environments from the full set of over 100 problems, so there is plenty of uncharted territory ahead. If you want to practice, there are several items you can experiment with:

- Testing the robustness of demonstrations to noisy clicks.
- The action space for the clicking approach could be improved by predicting the x and y coordinates of the place to click.
- DOM data could be used instead of (or in addition to) screen pixels. Then, the prediction will be the element of the tree to be clicked.
- Try other problems. There is a wide variety of them, requiring keyboard events to be generated, the sequence of actions planned, etc.
- Very recently, the LaVague project was published (<https://github.com/lavague-ai/LaVague>), which uses LLMs for web automation. Their approach is to ask an LLM to generate Selenium Python code to perform specific tasks. It will be very interesting to check it against MiniWoB++ problems.

Summary

In this chapter, you saw the practical application of RL methods for browser automation and used the MiniWoB++ benchmark. I believe that browser automation (and communicating with software humans are using in general) is an important milestone in future AI development.

This chapter concludes *Part 3* of the book. The next part will be devoted to more complicated and recent methods related to continuous action spaces, non-gradient methods, and other more advanced methods of RL.

In the next chapter, we will take a look at continuous control problems, which are an important subfield of RL, both theoretically and practically.

Leave a Review!

Thank you for purchasing this book from Packt Publishing—we hope you enjoy it! Your feedback is invaluable and helps us improve and grow. Once you've completed reading it, please take a moment to leave an Amazon review; it will only take a minute, but it makes a big difference for readers like you.

Scan the QR code below to receive a free ebook of your choice.

<https://packt.link/NzOWQ>



PART IV

ADVANCED RL

15

Continuous Action Space

This chapter kicks off the advanced **reinforcement learning (RL)** part of the book by taking a look at a problem that has only been briefly mentioned so far: working with environments when our action space is not discrete. Continuous action space problems are an important subfield of RL, both theoretically and practically, because they have essential applications in robotics, control problems, and other fields in which we communicate with physical objects. In this chapter, you will become familiar with the challenges that arise in such cases and learn how to solve them.

This material might be applicable even in problems and environments we've already seen. For example, in the previous chapter, when we implemented a mouse clicking in the browser environment, the x and y coordinates for the click position could be seen as two continuous variables to be predicted as actions. This might look a bit artificial, but such representation has a lot of sense from the environment perspective: it is much more compact and naturally captures possible click dispersion. At the end, clicking at coordinate (x, y) isn't much different from clicking at the $(x + 1, y + 1)$ position for most of the tasks.

In this chapter, we will:

- Cover the continuous action space, why it is important, how it differs from the already familiar discrete action space, and the way it is implemented in the Gym API
- Discuss the domain of continuous control using RL methods
- Check three different algorithms on the problem of a four-legged robot

Why a continuous space?

All the examples that we have seen so far in the book had a discrete action space, so you might have the wrong impression that discrete actions dominate the field. This is a very biased view, of course, and just reflects the selection of domains that we picked our test problems from. Besides Atari games and simple, classic RL problems, there are many tasks that require more than just making a selection from a small and discrete set of things to do.

To give you an example, just imagine a simple robot with only one controllable joint that can be rotated in some range of degrees. Usually, to control a physical joint, you have to specify either the desired position or the force applied. In both cases, you need to make a decision about a continuous value. This value is fundamentally different from a discrete action space, as the set of values on which you can make a decision is potentially infinite. For instance, you could ask the joint to move to a 13.5° angle or 13.512° angle, and the results could be different. Of course, there are always some physical limitations of the system, as you can't specify the action with infinite precision, but the size of the potential values could be very large.

In fact, when you need to communicate with a physical world, a **continuous action space** is much more likely than having a discrete set of actions. As an example, different kinds of robots control systems (such as a heating/cooling controller). The methods of RL could be applied to this domain, but there are some details that you need to take into consideration before using the **advantage actor-critic (A2C)** or **deep Q-network (DQN)** methods.

In this chapter, we will explore how to deal with this family of problems. This will act as a good starting point for learning about this very interesting and important domain of RL.

The action space

The fundamental and obvious difference with a continuous action space is its continuity. In contrast to a discrete action space, when the action is defined as a discrete, mutually exclusive set of options to choose from (for example `{left, right}`, which contains only two elements), the continuous action has a value from some range (for instance, $[0 \dots 1]$, which includes infinite elements, like 0.5 , $\frac{\sqrt{3}}{2}$, and $\frac{\pi^3}{e^5}$). On every time step, the agent needs to select the concrete value for the action and pass it to the environment.

In Gym, a continuous action space is represented as the `gym.spaces.Box` class, which was described, when we talked about the observation space. You may remember that `Box` includes a set of values with a shape and bounds. For example, every observation from the Atari emulator was represented as `Box(low=0, high=255, shape=(210, 160, 3))`, which means 100,800 values organized as a 3D tensor, with values from the $0 \dots 255$ range.

For the action space, it's unlikely that you'll work with such large numbers of actions. For example, the four-legged robot that we will use as a testing environment has eight continuous actions, which correspond to eight motors, two in every leg. For this environment, the action space will be defined as `Box(low=-1, high=1, shape= (8,))`, which means eight values from the range $-1 \dots 1$ have to be selected at every timestamp to control the robot.

In this case, the action passed to the `env.step()` at every step won't be an integer anymore; it will be a NumPy vector of some shape with individual action values. Of course, there could be more complicated cases when the action space is a combination of discrete and continuous actions, which may be represented with the `gym.spaces.Tuple` class.

Environments

Most of the environments that include continuous action spaces are related to the physical world, so physics simulations are normally used. There are lots of software packages that can simulate physical processes, from very simple open source tools to complex commercial packages that can simulate multiphysics processes (such as fluid, burning, and strength simulations).

In the case of robotics, one of the most popular packages is MuJoCo, which stands for Multi-Joint dynamics with Contact (<https://www.mujoco.org>). This is a physics engine in which you can define the components of the system and their interaction and properties. Then the simulator is responsible for solving the system by taking into account your intervention and finding the parameters (usually the location, velocities, and accelerations) of the components. This makes it ideal as a playground for RL environments, as you can define fairly complicated systems (such as multipede robots, robotic arms, or humanoids) and then feed the observation into the RL agent, getting actions back.

For a long time, MuJoCo was a commercial package and required an expensive license to be purchased. Trial licenses and education licenses existed, but they limited the audience for this software. But in 2022, DeepMind acquired MuJoCo and made the source code publicly available for everybody, which was a really great and generous move. Farama Gymnasium includes several MuJoCo environments (<https://gymnasium.farama.org/environments/mujoco/>) out of the box; to get them working, you need to install the `gymnasium[mujoco]` package.

Besides MuJoCo, there are other physics simulators you can use for RL. One of the most popular is PyBullet (<https://pybullet.org/>), which was open source from the very beginning. In this chapter, we'll use PyBullet in our experiments, and later in the book, we'll take a look at MuJoCo as well. To install PyBullet, you need to execute `pip install pybullet==3.2.6` in your Python environment.

As PyBullet wasn't updated to the Gymnasium API, we also need to install OpenAI Gym for compatibility:

```
pip install gym==0.25.1
```



We use version 0.25.1, as later versions of OpenAI Gym are not compatible with the latest version of PyBullet.

The following code (which is available in `Chapter15/01_check_env.py`) allows you to check that PyBullet works. It looks at the action space and renders an image of the environment that we will use as a guinea pig in this chapter:

```
import gymnasium as gym

ENV_ID = "MinitaurBulletEnv-v0"
ENTRY = "pybullet_envs.bullet.minitaur_gym_env:MinitaurBulletEnv"
RENDER = True

if __name__ == "__main__":
    gym.register(ENV_ID, entry_point=ENTRY, max_episode_steps=1000,
                reward_threshold=15.0, disable_env_checker=True)
    env = gym.make(ENV_ID, render=RENDER)

    print("Observation space:", env.observation_space)
    print("Action space:", env.action_space)
    print(env)
    print(env.reset())
    input("Press any key to exit\n")
    env.close()
```

After you start the utility, it should open the **graphical user interface (GUI)** window with our four-legged robot, shown in the following figure, that we will train to move:

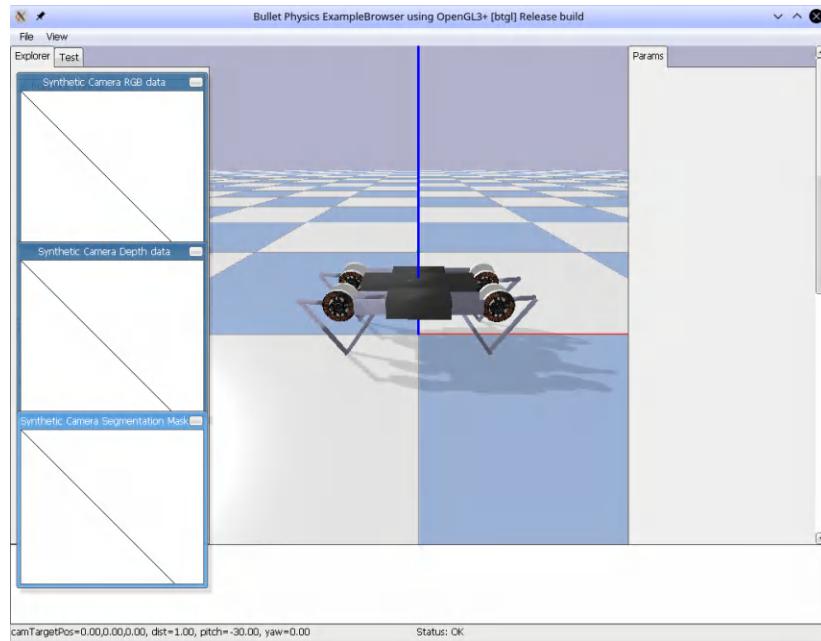


Figure 15.1: The Minitaur environment in the PyBullet GUI (for better visualization, refer to <https://packt.link/gbp/9781835882702>)

This environment provides you with 28 numbers as the observation and they correspond to different physical parameters of the robot: velocity, position, and acceleration. (You can check the source code of `MinitaurBulletEnv-v0` for details.) The action space is eight numbers that define the parameters of the motors. There are two in every leg (one in every knee). The reward of this environment is the distance traveled by the robot minus the energy spent.

The A2C method

The first method that we will apply to our walking robot problem is A2C, which we experimented with in Part 3 of the book. This choice of method is quite obvious, as A2C is very easy to adapt to the continuous action domain. As a quick refresher, A2C’s idea is to estimate the gradient of our policy as $\nabla J = \nabla_\theta \log \pi_\theta(a|s)(R - V_\theta(s))$. The policy $\pi_\theta(s)$ is supposed to provide the probability distribution of actions given the observed state. The quantity $V_\theta(s)$ is called a critic, equal to the value of the state, and is trained using the **mean squared error (MSE)** loss between the critic’s return and the value estimated by the Bellman equation. To improve exploration, the entropy bonus $L_H = \pi_\theta(s) \log \pi_\theta(s)$ is usually added to the loss.

Obviously, the value head of the actor-critic will be unchanged for continuous actions. The only thing that is affected is the representation of the policy.

In the discrete cases that we have seen, we had only one action with several mutually exclusive discrete values. For such a case, the obvious representation of the policy was the probability distribution over all actions.

In a continuous case, we usually have several actions, each of which can take a value from some range. With that in mind, the simplest policy representation will be just those values returned for every action. These values should not be confused with the value of the state, $V(s)$, which indicates how many rewards we can get from the state. To illustrate the difference, let's imagine a simple car steering case in which we can only turn the wheel. The action at every moment will be the wheel angle (action value), but the value of every state will be the potential discounted reward from the state (for example, the distance the car can travel), which is a totally different thing.

Returning to our action representation options, if you remember what we covered in the *Policy representation* section in *Chapter 11*, the representation of an action as a concrete value has different disadvantages, mostly related to the exploration of the environment. A much better choice will be something stochastic, for example, the network returning parameters of the Gaussian distribution. For N actions, those parameters will be two vectors of size N . The first will be the mean values, μ , and the second vector will contain variances, σ^2 . In that case, our policy will be represented as a random N -dimensional vector of uncorrelated, normally distributed random variables, and our network can make a selection about the mean and the variance of every variable.

By definition, the probability density function of the Gaussian distribution is given by

$$f(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

We could directly use this formula to get the probabilities, but to improve numerical stability, it is worth doing some math and simplifying the expression for $\log \pi_\theta(a|s)$.

The final result will be this:

$$\log \pi_\theta(a|s) = -\frac{(x-\mu)^2}{2\sigma^2} - \log \sqrt{2\pi\sigma^2}$$

The entropy of the Gaussian distribution could be obtained using the differential entropy definition and will be $\ln \sqrt{2\pi e\sigma^2}$. Now we have everything we need to implement the A2C method, so let's do this.

Implementation

The complete source code is in `02_train_a2c.py`, `lib/model.py`, and `lib/common.py`. You will be familiar with most of the code, so the following includes only the parts that differ. Let's start with the model class defined in `lib/model.py`:

```
HID_SIZE = 128

class ModelA2C(nn.Module):
    def __init__(self, obs_size: int, act_size: int):
        super(ModelA2C, self).__init__()

        self.base = nn.Sequential(
            nn.Linear(obs_size, HID_SIZE),
            nn.ReLU(),
        )
        self.mu = nn.Sequential(
            nn.Linear(HID_SIZE, act_size),
            nn.Tanh(),
        )
        self.var = nn.Sequential(
            nn.Linear(HID_SIZE, act_size),
            nn.Softplus(),
        )
    self.value = nn.Linear(HID_SIZE, 1)
```

As you can see, our network has three heads, instead of the normal two for a discrete variant of A2C. The first two heads return the mean value and the variance of the actions, while the last is the critic head returning the value of the state. The mean value returned has an activation function of a hyperbolic tangent, which is the squashed output to the range of $-1 \dots 1$. The variance is transformed with the softplus activation function, which is $\log(1 + e^x)$ and has the shape of a smoothed **rectified linear unit (ReLU)** function. This activation helps to make our variance positive. The value head, as usual, has no activation function applied.

The forward pass is obvious; we apply the common layer first, and then we calculate individual heads:

```
def forward(self, x: torch.Tensor):
    base_out = self.base(x)
    return self.mu(base_out), self.var(base_out), self.value(base_out)
```

The next step is to implement the PTAN Agent class, which is used to convert the observation into actions:

```
class AgentA2C(ptan.agent.BaseAgent):
    def __init__(self, net: ModelA2C, device: torch.device):
        self.net = net
        self.device = device

    def __call__(self, states: ptan.agent.States, agent_states: ptan.agent.AgentStates):
        states_v = ptan.agent.float32_preprocessor(states)
```

```

states_v = states_v.to(self.device)

mu_v, var_v, _ = self.net(states_v)
mu = mu_v.data.cpu().numpy()
sigma = torch.sqrt(var_v).data.cpu().numpy()
actions = np.random.normal(mu, sigma)
actions = np.clip(actions, -1, 1)
return actions, agent_states

```

In the discrete case, we used the `ptan.agent.DQNAgent` and `ptan.agent.PolicyAgent` classes, but for our problem, we need to write our own, which is not complicated: you just need to write a class, derived from `ptan.agent.BaseAgent`, and override the `__call__` method, which needs to convert observations into actions.

In this class, we get the mean and the variance from the network and sample the normal distribution using NumPy functions. To prevent the actions from going outside of the environment's $-1 \dots 1$ bounds, we use `np.clip()`, which replaces all values less than -1 with -1, and values more than 1 with 1. The `agent_states` argument is not used, but it needs to be returned with the chosen actions, as our `BaseAgent` supports keeping the state of the agent. We don't need this functionality right now, but it will be handy in the next section on deep deterministic policy gradients, when we will need to implement a random exploration using the **Ornstein-Uhlenbeck (OU)** process.

With the model and the agent at hand, we can now go to the training process, defined in `02_train_a2c.py`. It consists of the training loop and two functions. The first is used to perform periodical tests of our model on the separate testing environment. During the testing, we don't need to do any exploration; we will just use the mean value returned by the model directly, without any random sampling. The testing function is as follows:

```

def test_net(net: model.ModelA2C, env: gym.Env, count: int = 10,
            device: torch.device = torch.device("cpu")):
    rewards = 0.0
    steps = 0
    for _ in range(count):
        obs, _ = env.reset()
        while True:
            obs_v = ptan.agent.float32_preprocessor([obs])
            obs_v = obs_v.to(device)
            mu_v = net(obs_v)[0]
            action = mu_v.squeeze(dim=0).data.cpu().numpy()
            action = np.clip(action, -1, 1)
            obs, reward, done, is_tr, _ = env.step(action)
            rewards += reward
            steps += 1

```

```

    if done or is_tr:
        break
    return rewards / count, steps / count

```

The second function defined in the training module implements the calculation of the logarithm of the taken actions' probabilities given the policy. The function is a straightforward implementation of the formula we saw earlier:

$$\log \pi_\theta(a|s) = -\frac{(x-\mu)^2}{2\sigma^2} - \log \sqrt{2\pi\sigma^2}:$$

```

def calc_logprob(mu_v: torch.Tensor, var_v: torch.Tensor, actions_v: torch.Tensor):
    p1 = - ((mu_v - actions_v) ** 2) / (2*var_v.clamp(min=1e-3))
    p2 = - torch.log(torch.sqrt(2 * math.pi * var_v))
    return p1 + p2

```

The only tiny difference is in using the `torch.clamp()` function to prevent the division on zero when the returned variance is too small.

The training loop, as usual, creates the network and the agent, and then instantiates the two-step experience source and optimizer. The hyperparameters used are given as follows. They weren't tweaked much, so there is plenty of room for optimization:

```

GAMMA = 0.99
REWARD_STEPS = 2
BATCH_SIZE = 32
LEARNING_RATE = 5e-5
ENTROPY_BETA = 1e-4

TEST_ITERS = 1000

```

The code used to perform the optimization step on the collected batch is very similar to the A2C training that we implemented in *Chapter 12*. The difference is only in using our `calc_logprob()` function and a different expression for the entropy bonus, which is shown next:

```

states_v, actions_v, vals_ref_v = common.unpack_batch_a2c(
    batch, net, device=device, last_val_gamma=GAMMA ** REWARD_STEPS)
batch.clear()

optimizer.zero_grad()
mu_v, var_v, value_v = net(states_v)

```

```
loss_value_v = F.mse_loss(value_v.squeeze(-1), vals_ref_v)
adv_v = vals_ref_v.unsqueeze(dim=-1) - value_v.detach()
log_prob_v = adv_v * calc_logprob(mu_v, var_v, actions_v)
loss_policy_v = -log_prob_v.mean()
ent_v = -(torch.log(2*math.pi*var_v) + 1)/2
entropy_loss_v = ENTROPY_BETA * ent_v.mean()

loss_v = loss_policy_v + entropy_loss_v + loss_value_v
loss_v.backward()
optimizer.step()
```

Every TEST_ITERS frames, the model is tested, and in the case of the best reward obtained, the model weights are saved.

Results

In comparison to other methods that we will look at in this chapter, A2C shows the worst results, both in terms of the best reward and convergence speed. That's likely because of the single environment used to gather experience, which is a weak point of the **policy gradient (PG)** methods. So, you may want to check the effect of several parallel environments on A2C.

To start the training, we pass the `-n` argument with the run name, which will be used in TensorBoard and a new directory to save the models. The `-dev` option could be used to enable the GPU usage, but due to the small dimensionality of the input and the tiny network size, it gives only a marginal increase in speed.

After 9M frames, which took 16 hours of optimization, the training process reached the best score of 0.35 during the testing, which is not very impressive. If we leave it running for a week or two, we might be able to achieve a better score. The reward and episode steps during the training and testing are shown in the following graphs:

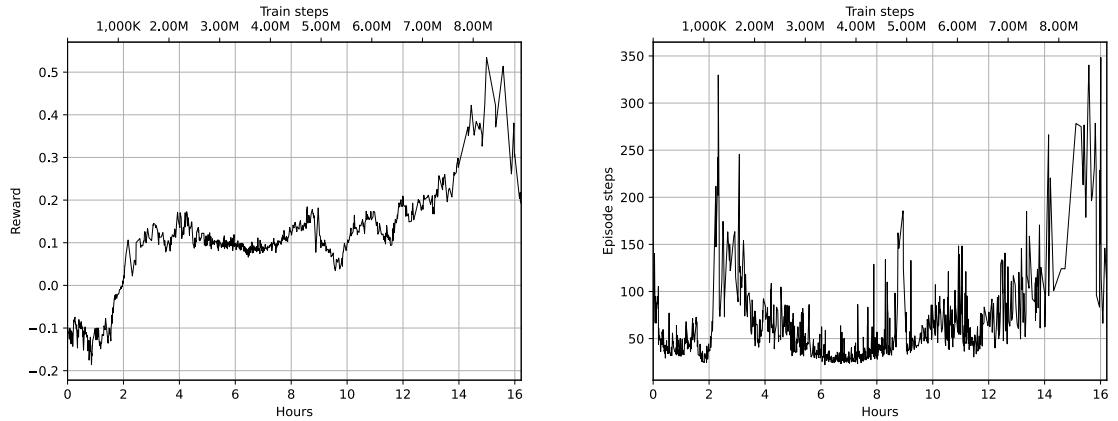


Figure 15.2: The reward (left) and steps (right) for training episodes

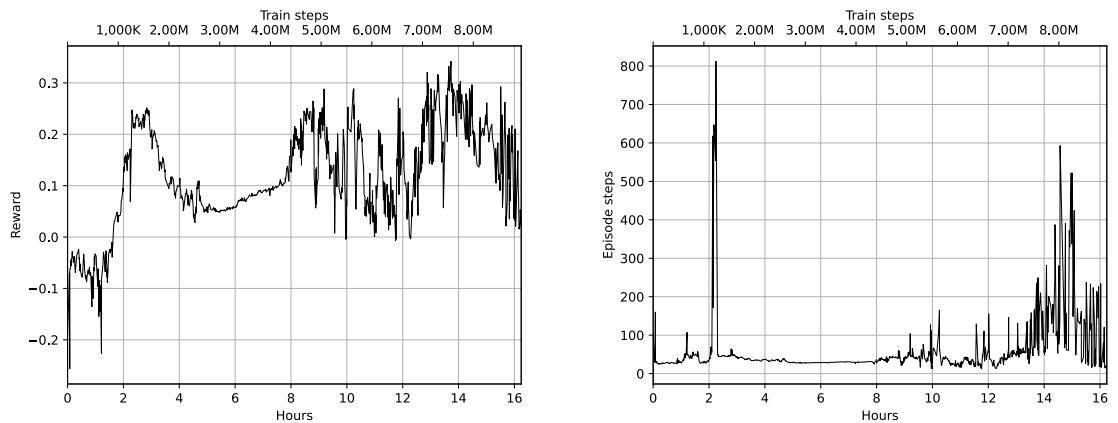


Figure 15.3: The reward (left) and steps (right) for testing episodes

The episode steps charts (right plots on both figures) shows the average count of steps performed in the episode before the end. The time limit of the environment is 1,000 steps, so everything lower than 1,000 indicates that the episode was stopped due to environment checks. For most of the PyBullet environments, special checks for self-damage are implemented internally, which stop the simulation.

Using models and recording videos

As you have seen before, the physical simulator can render the state of the environment, which makes it possible to see how our trained model behaves. To do that for our A2C models, there is a utility, `03_play_a2c.py`. Its logic is the same as in the `test_net()` function, so its code is not shown here.

To start it, you need to pass the `-m` option with the model file and optional parameter `-r` with a directory name, which will be used to save the video using the `RecordVideo` wrapper we discussed in *Chapter 2*.

At the end of the simulation, the utility shows the number of steps and accumulated reward. For example, the best A2C model from my training was able to get the reward 0.312 and the video is just 2 seconds long (you can find it here: <https://youtu.be/s9BReDUtpQs>). *Figure 15.4* shows the last frame of the video and it looks like our model had problems keeping the balance.

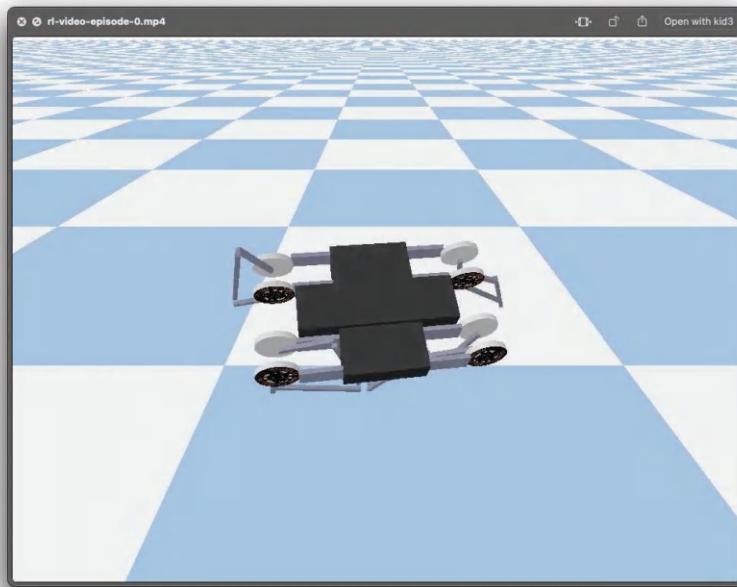


Figure 15.4: Last frame of A2C model simulation

Deep deterministic policy gradients

The next method that we will take a look at is called deep **deterministic policy gradients** (DDPG), which is an actor-critic method but has a very nice property of being off-policy. The following is a simplified interpretation of the strict proofs. If you are interested in understanding the core of this method deeply, you can always refer to the article by Silver et al. called *Deterministic policy gradient algorithms* [Sil+14], published in 2014, and the paper by Lillicrap et al. called *Continuous control with deep reinforcement learning* [Lil15], published in 2015.

The simplest way to illustrate the method is through comparison with the already familiar A2C method. In this method, the actor estimates the stochastic policy, which returns the probability distribution over discrete actions or, as we have just covered in the previous section, the parameters of normal distribution. In both cases, our policy is stochastic, so, in other words, our action taken is sampled from this distribution.

Deterministic policy gradients also belong to the A2C family, but the policy is *deterministic*, which means that it directly provides us with the action to take from the state. This makes it possible to apply the chain rule to the Q-value, and by maximizing the Q , the policy will be improved as well. To understand this, let's look at how the actor and critic are connected in a continuous action domain.

Let's start with the actor, as it is the simpler of the two. What we want from it is the action to take for every given state. In a continuous action domain, every action is a number, so the actor network will take the state as an input and return N values, one for every action. This mapping will be deterministic, as the same network always returns the same output if the input is the same. (We're not going to use dropout or anything adding stochasticity to the inference; we're just going to use an ordinary feed-forward network.)

Now let's look at the critic. The role of the critic is to estimate the Q-value, which is a discounted reward of the action taken in some state. However, our action is a vector of numbers, so our critic network now accepts two inputs: the state and the action. The output from the critic will be the single number, which corresponds to the Q-value. This architecture is different from the DQN, when our action space was discrete and, for efficiency, we returned values for all actions in one pass. This mapping is also deterministic.

So, we have two functions:

- The actor, let's call it $\mu(s)$, which converts the state into the action
- The critic, which, through the state and the action, gives us the Q-value: $Q(s, a)$

We can substitute the actor function into the critic and get the expression with only one input parameter of our state: $Q(s, \mu(s))$. In the end, neural networks are just functions.

Now, the output of the critic gives us the approximation of the entity that we're interested in maximizing in the first place: the discounted total reward. This value depends not only on the input state but also on the parameters of the θ_μ actor and the θ_Q critic networks. At every step of our optimization, we want to change the actor's weights to improve the total reward that we get. In mathematical terms, we want the gradient of our policy.

In his deterministic policy gradient theorem, Silver et al. proved that the stochastic policy gradient is equivalent to the deterministic policy gradient. In other words, to improve the policy, we just need to calculate the gradient of the $Q(s, \mu(s))$ function. By applying the chain rule, we get the gradient: $\nabla_a Q(s, a) \nabla_{\theta_\mu} \mu(s)$.

Note that, despite both the A2C and DDPG methods belonging to the A2C family, the way that the critic is used is different. In A2C, we use the critic as a baseline for a reward from the experienced trajectories, so the critic is an optional piece (without it, we will get the **REINFORCE** method) and is used to improve the stability. This happens as the policy in A2C is stochastic, which builds a barrier in our backpropagation capabilities (we have no way of differentiating the random sampling step).

In DDPG, the critic is used in a different way. As our policy is deterministic, we can now calculate the gradients from Q , which is obtained from the critic network, which uses actions produced by the actor (check *Figure 15.5*), so the whole system is differentiable and could be optimized end to end with **stochastic gradient descent (SGD)**. To update the critic network, we can use the Bellman equation to find the approximation of $Q(s, a)$ and minimize the MSE objective.

All this may look a bit cryptic, but behind it stands a quite simple idea: the critic is updated as we did in A2C, and the actor is updated in a way to maximize the critic's output. The beauty of this method is that it is off-policy, which means that we can now have a huge replay buffer and other tricks that we used in DQN training. Nice, right?

Exploration

The price we have to pay for all this goodness is that our policy is now deterministic, so we have to explore the environment somehow. We can do this by adding noise to the actions returned by the actor before we pass them to the environment. There are several options here. The simplest method is just to add the random noise to the actions: $\mu(s) + \epsilon \mathcal{N}$. We will use this in the next method that we will consider in this chapter.

A more advanced (and sometimes giving better results) approach to the exploration is to use the previously mentioned Ornstein-Uhlenbeck process, which is very popular in the financial world and other domains dealing with stochastic processes. This process models the velocity of a massive Brownian particle under the influence of friction and is defined by this stochastic differential equation:

$$\partial x_t = \theta(\mu - x_t)\partial t + \sigma \partial W,$$

where θ , μ , and σ are parameters of the process and W_t is the Wiener process. In a discrete-time case, the OU process could be written as

$$x_{t+1} = x_t + \theta(\mu - x_t) + \sigma \mathcal{N}.$$

This equation expresses the next value generated by the process via the previous value of the noise, adding normal noise, \mathcal{N} . In our exploration, we will add the value of the OU process to the action returned by the actor.

Implementation

This example consists of three source files:

- `lib/model.py` contains the model and the PTAN agent
- `lib/common.py` has a function used to unpack the batch
- `04_train_ddpg.py` has the startup code and the training loop

Here, I will show only the significant pieces of the code. The model consists of two separate networks for the actor and critic, and it follows the architecture from the paper by Lillicrap et al. [Lil15] mentioned earlier. The actor is extremely simple and is a feed-forward network with two hidden layers. The input is an observation vector, whereas the output is a vector with N values, one for each action. The output actions are transformed with hyperbolic tangent nonlinearity to squeeze the values to the $-1 \dots 1$ range.

The critic is a bit unusual, as it includes two separate paths for the observation and the actions, and those paths are concatenated together to be transformed into the critic output of one number. *Figure 15.5* shows the structure of both networks:

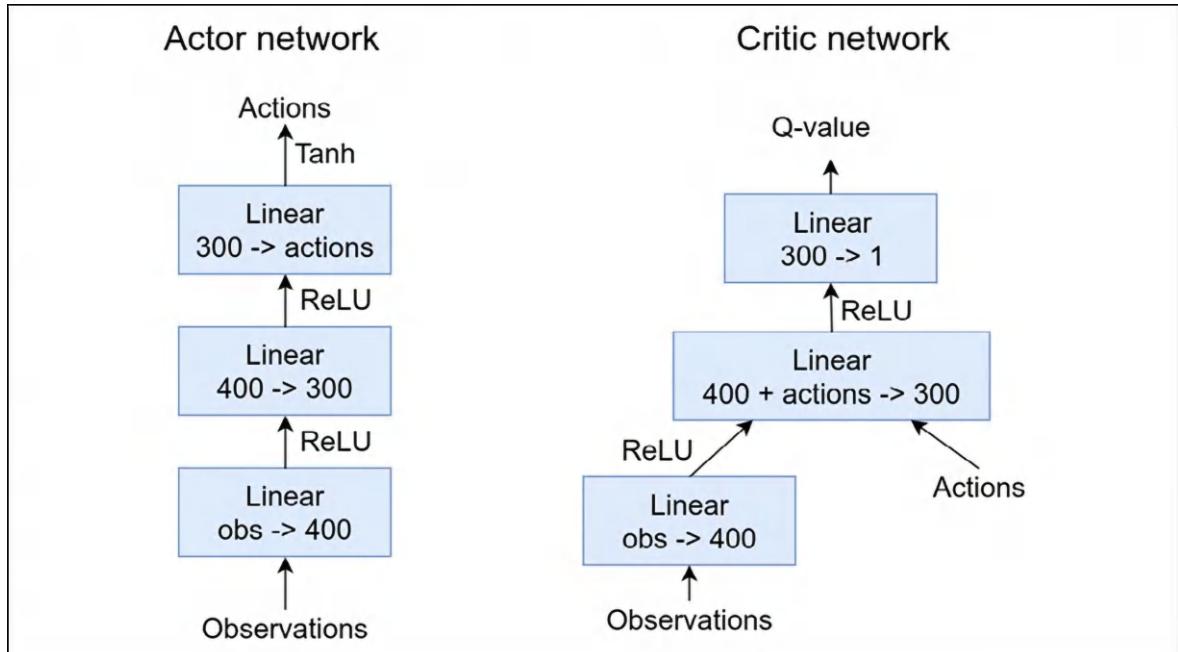


Figure 15.5: The DDPG actor and critic networks

The code for the actor includes a three-layer network that produces the action value:

```
class DDPGActor(nn.Module):
    def __init__(self, obs_size: int, act_size: int):
        super(DDPGActor, self).__init__()

        self.net = nn.Sequential(
            nn.Linear(obs_size, 400),
            nn.ReLU(),
            nn.Linear(400, 300),
            nn.ReLU(),
            nn.Linear(300, act_size),
            nn.Tanh()
        )

    def forward(self, x: torch.Tensor):
        return self.net(x)
```

Similarly, the following is the code used for the critic:

```
class DDPGCritic(nn.Module):
    def __init__(self, obs_size: int, act_size: int):
        super(DDPGCritic, self).__init__()

        self.obs_net = nn.Sequential(
            nn.Linear(obs_size, 400),
            nn.ReLU(),
        )

        self.out_net = nn.Sequential(
            nn.Linear(400 + act_size, 300),
            nn.ReLU(),
            nn.Linear(300, 1)
        )

    def forward(self, x: torch.Tensor, a: torch.Tensor):
        obs = self.obs_net(x)
        return self.out_net(torch.cat([obs, a], dim=1))
```

The `forward()` function of the critic first transforms the observations with its small network and then concatenates the output and given actions to transform them into one single value of Q. To use the actor network with the PTAN experience source, we need to define the agent class that has to transform the observations into actions.

This class is the most convenient place to put our OU exploration process, but to do this properly, we should use the functionality of the PTAN agents that we haven't used so far: *optional statefulness*.

The idea is simple: our agent transforms the observations into actions. But what if it needs to remember something between the observations? All our examples have been stateless so far, but sometimes this is not enough. The issue with OU is that we have to track the OU values between the observations.

Another very useful use case for stateful agents is a **partially observable Markov decision process (POMDP)**, which was briefly mentioned in *Chapter 6* and *Chapter 14*. The POMDP is a Markov decision process when the state observed by the agent doesn't comply with the Markov property and doesn't include the full information to distinguish one state from another. In that case, our agent needs to track the state along the trajectory to be able to take the action.

So, the code for the agent that implements the OU for exploration is as follows:

```
class AgentDDPG(ptan.agent.BaseAgent):
    def __init__(self, net: DDPGActor, device: torch.device = torch.device('cpu'),
                 ou_enabled: bool = True, ou_mu: float = 0.0, ou_teta: float = 0.15,
                 ou_sigma: float = 0.2, ou_epsilon: float = 1.0):
        self.net = net
        self.device = device
        self.ou_enabled = ou_enabled
        self.ou_mu = ou_mu
        self.ou_teta = ou_teta
        self.ou_sigma = ou_sigma
        self.ou_epsilon = ou_epsilon

    def initial_state(self):
        return None
```

The constructor accepts a lot of parameters, most of which are the default hyperparameters of the OU process taken from the paper *Continuous Control with Deep Reinforcement Learning*.

The `initial_state()` method is derived from the `BaseAgent` class and has to return the initial state of the agent when a new episode is started. As our initial state has to have the same dimension as the actions (we want to have individual exploration trajectories for every action of the environment), we postpone the initialization by returning `None` as the initial state.

In the `__call__` method, we'll take this into account:

```
def __call__(self, states: ptan.agent.States, agent_states: ptan.agent.AgentStates):
    states_v = ptan.agent.float32_preprocessor(states)
    states_v = states_v.to(self.device)
    mu_v = self.net(states_v)
    actions = mu_v.data.cpu().numpy()
```

This method is the core of the agent and the purpose of it is to convert the observed state and internal agent state into the action. As the first step, we convert the observations into the appropriate form and ask the actor network to convert them into deterministic actions. The rest of the method is for adding the exploration noise by applying the OU process.

In this loop, we iterate over the batch of observations and the list of the agent states from the previous call, and we update the OU process value, which is a straightforward implementation of the already shown formula:

```
if self.ou_enabled and self.ou_epsilon > 0:
    new_a_states = []
    for a_state, action in zip(agent_states, actions):
        if a_state is None:
            a_state = np.zeros(shape=action.shape, dtype=np.float32)
            a_state += self.ou_teta * (self.ou_mu - a_state)
            a_state += self.ou_sigma * np.random.normal(size=action.shape)

        action += self.ou_epsilon * a_state
        new_a_states.append(a_state)
```

To finalize the loop, we add the noise from the OU process to our actions and save the noise value for the next step.

Finally, we clip the actions to enforce them to fall into the $-1 \dots 1$ range; otherwise, PyBullet will throw an exception:

```
else:
    new_a_states = agent_states
    actions = np.clip(actions, -1, 1)
return actions, new_a_states
```

The final piece of the DDPG implementation is the training loop in the `04_train_ddpg.py` file. To improve the stability, we use the replay buffer with 100,000 transitions and target networks for both the actor and the critic (we discussed both in *Chapter 6*):

```

act_net = model.DDPGActor(env.observation_space.shape[0],
                           env.action_space.shape[0]).to(device)
crt_net = model.DDPGCritic(env.observation_space.shape[0],
                           env.action_space.shape[0]).to(device)
print(act_net)
print(crt_net)
tgt_act_net = ptan.agent.TargetNet(act_net)
tgt_crt_net = ptan.agent.TargetNet(crt_net)

writer = SummaryWriter(comment="-ddpg_" + args.name)
agent = model.AgentDDPG(act_net, device=device)
exp_source = ptan.experience.ExperienceSourceFirstLast(
    env, agent, gamma=GAMMA, steps_count=1)
buffer = ptan.experience.ExperienceReplayBuffer(exp_source, buffer_size=REPLAY_SIZE)
act_opt = optim.Adam(act_net.parameters(), lr=LEARNING_RATE)
crt_opt = optim.Adam(crt_net.parameters(), lr=LEARNING_RATE)

```

We also use two different optimizers to simplify the way that we handle gradients for the actor and critic training steps. The most interesting code is inside the training loop. On every iteration, we store the experience into the replay buffer and sample the training batch:

```

batch = buffer.sample(BATCH_SIZE)
states_v, actions_v, rewards_v, dones_mask, last_states_v = \
    common.unpack_batch_ddqn(batch, device)

```

Then, two separate training steps are performed. To train the critic, we need to calculate the target Q-value using the one-step Bellman equation, with the target critic network as the approximation of the next state:

```

crt_opt.zero_grad()
q_v = crt_net(states_v, actions_v)
last_act_v = tgt_act_net.target_model(last_states_v)
q_last_v = tgt_crt_net.target_model(last_states_v, last_act_v)
q_last_v[dones_mask] = 0.0
q_ref_v = rewards_v.unsqueeze(dim=-1) + q_last_v * GAMMA

```

When we have got the reference, we can calculate the MSE loss and ask the critic's optimizer to tweak the critic's weights. The whole process is similar to the training for the DQN, so nothing is really new here:

```

critic_loss_v = F.mse_loss(q_v, q_ref_v.detach())
critic_loss_v.backward()
crt_opt.step()

```

```
tb_tracker.track("loss_critic", critic_loss_v, frame_idx)
tb_tracker.track("critic_ref", q_ref_v.mean(), frame_idx)
```

On the actor's training step, we need to update the actor's weights in a direction that will increase the critic's output. As both the actor and critic are represented as differentiable functions, what we need to do is just pass the actor's output to the critic and then minimize the negated value returned by the critic:

```
act_opt.zero_grad()
cur_actions_v = act_net(states_v)
actor_loss_v = -crt_net(states_v, cur_actions_v)
actor_loss_v = actor_loss_v.mean()
```

This negated output of the critic could be used as a loss to backpropagate it to the critic network and, finally, the actor. We don't want to touch the critic's weights, so it's important to ask only the actor's optimizer to do the optimization step. The weights of the critic will still keep the gradients from this call, but they will be discarded on the next optimization step:

```
actor_loss_v.backward()
act_opt.step()
tb_tracker.track("loss_actor", actor_loss_v, frame_idx)
```

As the last step of the training loop, we perform the update of the target networks in an unusual way:

```
tgt_act_net.alpha_sync(alpha=1 - 1e-3)
tgt_crt_net.alpha_sync(alpha=1 - 1e-3)
```

Previously, we synced the weights from the optimized network into the target periodically. In continuous action problems, such syncing works worse than so-called "soft sync." The soft sync is carried out on every step, but only a small ratio of the optimized network's weights are added to the target network. This makes a smooth and slow transition from the old weight to the new ones.

Results and video

The code can be started in the same way as the A2C example: you need to pass the run name and optional `-dev` flag. My experiments have shown $\approx 30\%$ speed increase from a GPU, so if you're in a hurry, using CUDA may be a good idea, but the increase is not that dramatic, as we have seen in the case of Atari games.

After 5M observations, which took about 20 hours, the DDPG algorithm was able to reach the mean reward of 4.5 on 10 test episodes, which is an improvement over the A2C result. The training dynamics are shown in *Figure 15.6* and *Figure 15.7*.

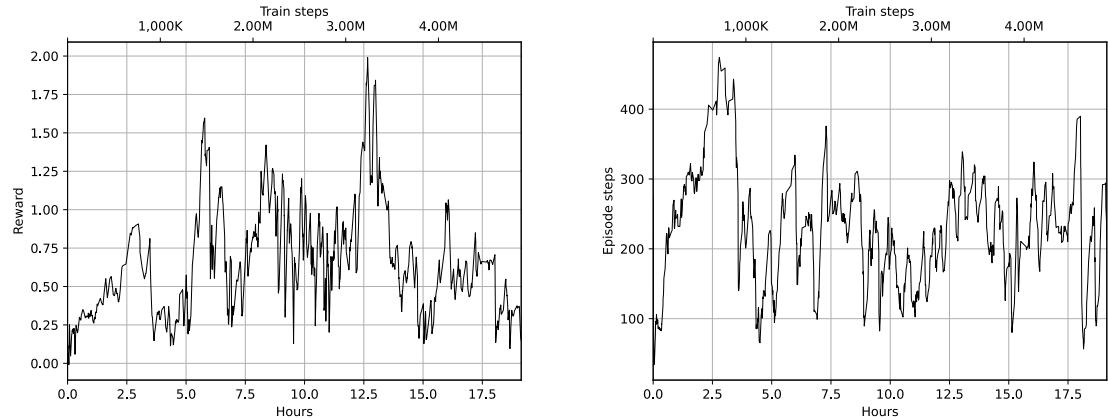


Figure 15.6: The reward (left) and steps (right) for training episodes

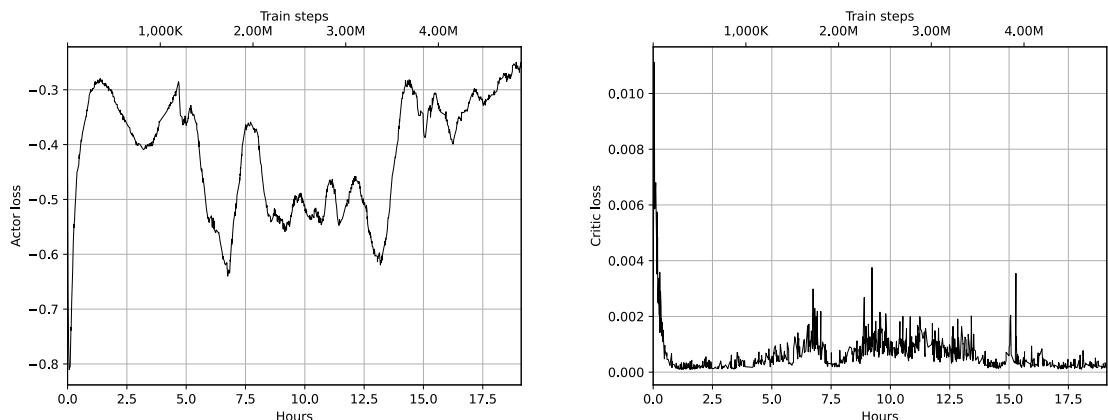


Figure 15.7: Actor loss (left) and critic loss (right) during the training

The “Episode steps” plot shows the mean length of the episodes that we used for training. The critic loss is an MSE loss and should be low, but the actor loss, as you will remember, is the negated critic’s output, so the smaller it is, the better the reward that the actor can (potentially) achieve.

In *Figure 15.8*, the shown values were obtained during the testing (which are average values obtained for 10 episodes).

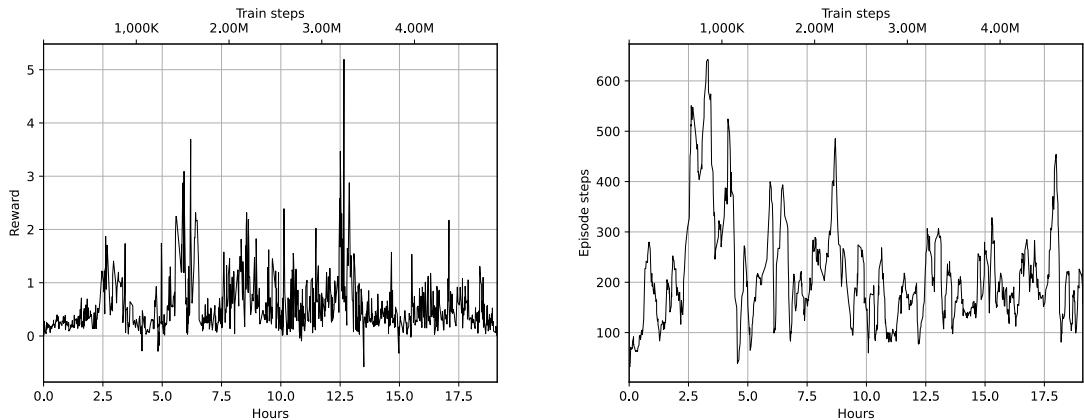


Figure 15.8: The reward (left) and steps (right) for testing episodes

To test the saved model and record the video the same way we did for the A2C model, you can use the utility `05_play_ddpg.py`. It uses the same command-line options, but is supposed to load DDPG models. In figure *Figure 15.9*, the last frame of my video is shown:

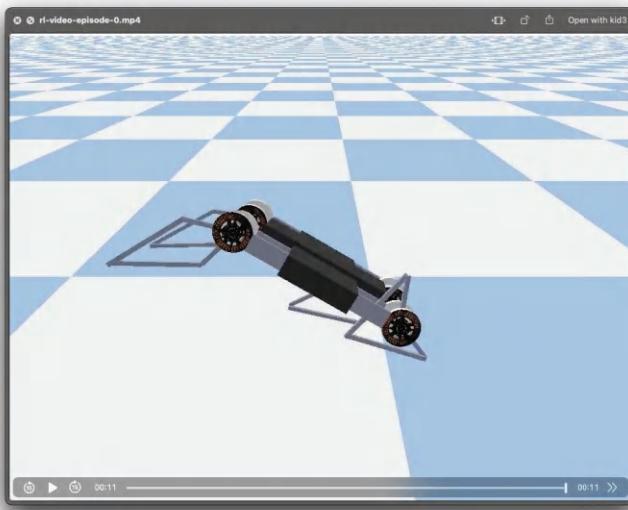


Figure 15.9: Last frame of DDPG model simulation

The score during the testing was 3.033 and the video is available at <https://youtu.be/vVnd0Nu1d9s>. Now the video is 11 seconds long and the model fails after falling forward.

Distributional policy gradients

As the last method of this chapter, we will take a look at the paper by Barth-Maron et al., called *Distributed distributional deterministic policy gradients* [Bar+18], published in 2018.

The full name of the method is **Distributed Distributional Deep Deterministic Policy Gradients** or **D4PG** for short. The authors proposed several improvements to the DDPG method to improve stability, convergence, and sample efficiency.

First, they adapted the distributional representation of the Q-value proposed in the paper by Bellemare et al. called *A distributional perspective on reinforcement learning*, published in 2017 [BDM17]. We discussed this approach in *Chapter 8*, when we talked about DQN improvements, so refer to it or to the original Bellemare paper for details. The core idea is to replace a single Q-value from the critic with a probability distribution. The Bellman equation is replaced with the Bellman operator, which transforms this distributional representation in a similar way.

The second improvement was the usage of the n-step Bellman equation, unrolled to speed up the convergence. We also discussed this in detail in *Chapter 8*.

Another difference versus the original DDPG method was the usage of the prioritized replay buffer instead of the uniformly sampled buffer. So, strictly speaking, the authors took relevant improvements from the paper by Hassel et al., called *Rainbow: Combining Improvements in Deep Reinforcement Learning*, which was published in 2017 [Hes+18], and adapted them to the DDPG method. The result was impressive: this combination showed state-of-the-art results on the set of continuous control problems. Let's try to reimplement the method and check it ourselves.

Architecture

The most notable change between D4PG and DDPG is the critic's output. Instead of returning the single Q-value for the given state and the action, it now returns `N_ATOMS` values, corresponding to the probabilities of values from the predefined range. In my code, I used `N_ATOMS=51` and the distribution range of `Vmin=-10` and `Vmax=10`, so the critic returned 51 numbers, representing the probabilities of the discounted reward falling into bins with bounds in $[-10, -9.6, -9.2, \dots, 9.6, 10]$.

Another difference between D4PG and DDPG is the exploration. DDPG used the OU process for exploration, but according to the D4PG authors, they tried both OU and adding simple random noise to the actions, and the result was the same. So, they used a simpler approach for exploration in the paper.

The last significant difference in the code is related to the training, as D4PG uses cross-entropy loss to calculate the difference between two probability distributions (returned by the critic and obtained as a result of the Bellman operator). To make both distributions aligned to the same supporting atoms, distribution projection is used in the same way as in the original paper by Bellemare et al.

Implementation

The complete source code is in `06_train_d4pg.py`, `lib/model.py`, and `lib/common.py`. As before, we start with the model class. The actor class has exactly the same architecture as in DDPG, so during the training class, `DDPGActor` is used. The critic has the same size and count of hidden layers; however, the output is not a single number, but `N_ATOMS`:

```
class D4PGCritic(nn.Module):
    def __init__(self, obs_size: int, act_size: int,
                 n_atoms: int, v_min: float, v_max: float):
        super(D4PGCritic, self).__init__()

        self.obs_net = nn.Sequential(
            nn.Linear(obs_size, 400),
            nn.ReLU(),
        )

        self.out_net = nn.Sequential(
            nn.Linear(400 + act_size, 300),
            nn.ReLU(),
            nn.Linear(300, n_atoms)
        )

        delta = (v_max - v_min) / (n_atoms - 1)
        self.register_buffer("supports", torch.arange(v_min, v_max + delta, delta))
```

We also create a helper PyTorch buffer with reward supports, which will be used to get from the probability distribution to the single mean Q-value:

```
def forward(self, x: torch.Tensor, a: torch.Tensor):
    obs = self.obs_net(x)
    return self.out_net(torch.cat([obs, a], dim=1))

def distr_to_q(self, distr: torch.Tensor):
    weights = F.softmax(distr, dim=1) * self.supports
    res = weights.sum(dim=1)
    return res.unsqueeze(dim=-1)
```

As you can see, `softmax()` application is not part of the network's `forward()` method, as we're going to use the more stable `log_softmax()` function during the training. Due to this, `softmax()` needs to be applied when we want to get actual probabilities.

The agent class is much simpler for D4PG and has no state to track:

```
class AgentD4PG(ptan.agent.BaseAgent):
    def __init__(self, net: DDPGActor, device: torch.device = torch.device("cpu"),
                 epsilon: float = 0.3):
        self.net = net
        self.device = device
        self.epsilon = epsilon

    def __call__(self, states: ptan.agent.States, agent_states: ptan.agent.AgentStates):
        states_v = ptan.agent.float32_preprocessor(states)
        states_v = states_v.to(self.device)
        mu_v = self.net(states_v)
        actions = mu_v.data.cpu().numpy()
        actions += self.epsilon * np.random.normal(size=actions.shape)
        actions = np.clip(actions, -1, 1)
        return actions, agent_states
```

For every state to be converted to actions, the agent applies the actor network and adds Gaussian noise to the actions, scaled by the epsilon value. In the training code, we have the following hyperparameters:

```
GAMMA = 0.99
BATCH_SIZE = 64
LEARNING_RATE = 1e-4
REPLAY_SIZE = 100000
REPLAY_INITIAL = 10000
REWORLD_STEPS = 5

TEST_ITERS = 1000

Vmax = 10
Vmin = -10
N_ATOMS = 51
DELTA_Z = (Vmax - Vmin) / (N_ATOMS - 1)
```

I used a smaller replay buffer of 100,000, and it worked fine. (In the D4PG paper, the authors used 1M transitions in the buffer.) The buffer is prepopulated with 10,000 samples from the environment, and then the training starts.

For every training loop, we perform the same two steps as before: we train the critic and the actor. The difference is in the way that the loss for the critic is calculated:

```

batch = buffer.sample(BATCH_SIZE)
states_v, actions_v, rewards_v, dones_mask, last_states_v = \
    common.unpack_batch_ddqn(batch, device)

crt_opt.zero_grad()
crt_distr_v = crt_net(states_v, actions_v)
last_act_v = tgt_act_net.target_model(last_states_v)
last_distr_v = F.softmax(
    tgt_crt_net.target_model(last_states_v, last_act_v), dim=1)

```

As the first step in the critic's training, we ask it to return the probability distribution for the states and actions taken. This probability distribution will be used as an input in the cross-entropy loss calculation. To get the target probability distribution, we need to calculate the distribution from the last states in the batch and then perform the Bellman projection of the distribution:

```

proj_distr = distr_projection(
    last_distr_v.detach().cpu().numpy(),
    rewards_v.detach().cpu().numpy(),
    dones_mask.detach().cpu().numpy(), gamma=GAMMA**REWARD_STEPS)
proj_distr_v = torch.tensor(proj_distr).to(device)

```

This projection function is a bit complicated and is exactly the same as the implementation explained in detail in *Chapter 8*. As a quick reminder, it calculates the transformation of the `last_states` probability distribution, which is shifted according to the immediate reward and scaled to respect the discount factor. The result is the target probability distribution that we want our network to return. As there is no general cross-entropy loss function in PyTorch, we calculate it manually by multiplying the logarithm of the input probability by the target probabilities:

```

prob_dist_v = -F.log_softmax(crt_distr_v, dim=1) * proj_distr_v
critic_loss_v = prob_dist_v.sum(dim=1).mean()
critic_loss_v.backward()
crt_opt.step()

```

The actor's training is much simpler, and the only difference from the DDPG method is the use of the `distr_to_q()` method of the model to convert from the probability distribution to the single mean Q-value using support atoms:

```

        act_opt.zero_grad()
        cur_actions_v = act_net(states_v)
        crt_distr_v = crt_net(states_v, cur_actions_v)
        actor_loss_v = -crt_net.distr_to_q(crt_distr_v)
        actor_loss_v = actor_loss_v.mean()
        actor_loss_v.backward()
        act_opt.step()
    
```

Results

The D4PG method showed the best results in both convergence speed and the reward obtained. Following 20 hours of training, after about 3.5M observations, it was able to reach the mean test reward of 17.912. Given that “gym environment threshold” is 15.0 (which is a score when the environment considered it solved), that is a great result. And this result could be improved, as the count of steps is less than 1,000 (which is a time limit for the environment). This means that our model is being terminated prematurely because of internal environment checks. In *Figure 15.10* and *Figure 15.11*, we have the train and test metrics.

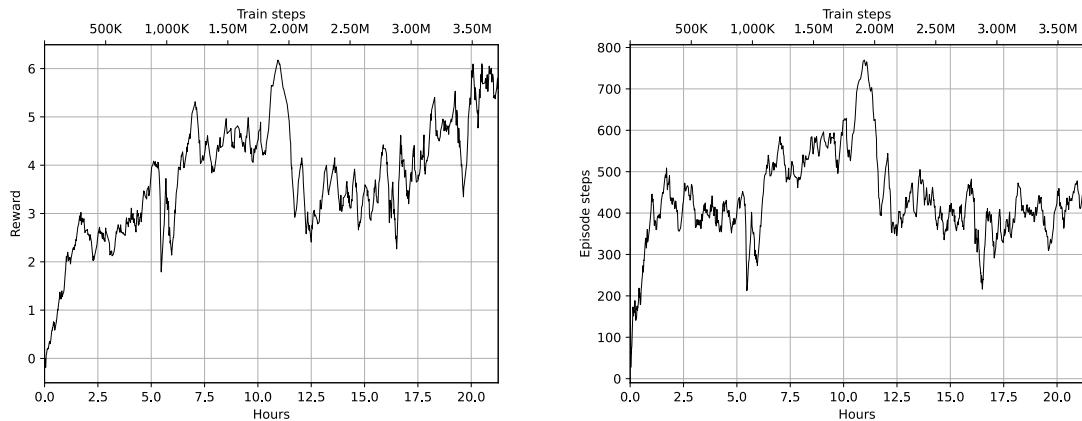


Figure 15.10: The reward (left) and steps (right) for training episodes

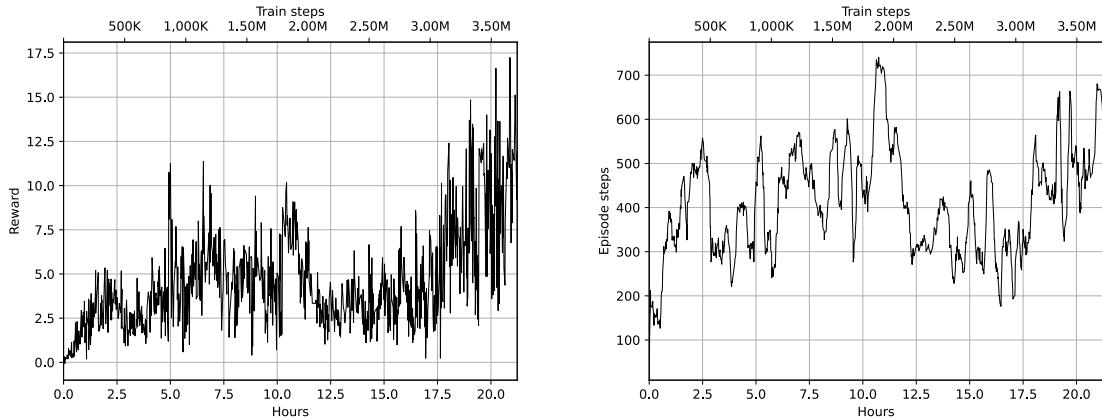


Figure 15.11: The reward (left) and steps (right) for testing episodes

To compare the implemented methods, Figure 15.12 contains test episode metrics from all three methods.

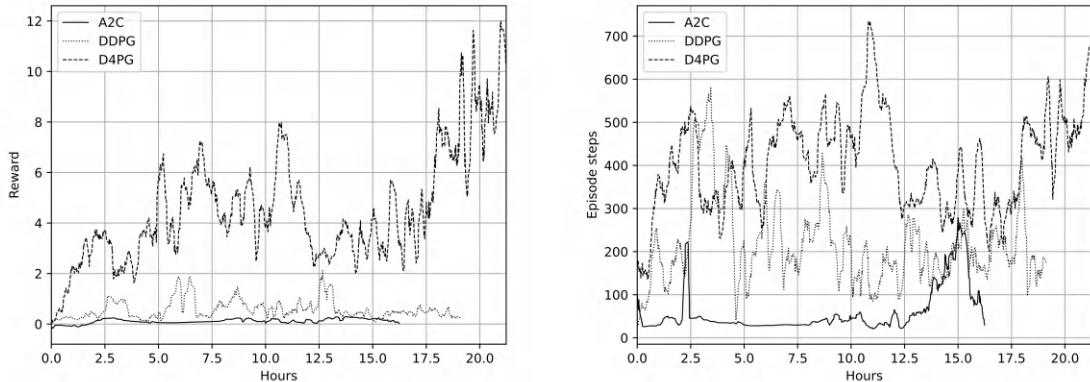


Figure 15.12: The reward (left) and steps (right) for test episodes

To check the model “in action,” you can use the same tool, `05_play_ddpg.py` (as actor has the same network structure as in DDPG). Now the video produced by the best model lasts 33 seconds and the final score was 17.827. You can watch it here: <https://youtu.be/XZdVrGPaI0M>.

Things to try

Here is a list of things you can do to improve your understanding of the topic:

- In the D4PG code, I used a simple replay buffer, which was enough to get good improvement over DDPG. You can try to switch the example to the prioritized replay buffer in the same way as we did in *Chapter 8*.
- There are lots of interesting and challenging environments around. For example, you can start with other PyBullet environments, but there is also the DeepMind Control Suite (Tassa et al., DeepMind Control Suite, arXiv abs/1801.00690 (2018)), MuJoCo-based environments in Gym, and many others.
- You can play with the very challenging *Learning to Run competition* from NIPS-2017 (which also took place in 2018 and 2019 with more challenging problems), where you are given a simulator of the human body and your agent needs to figure out how to move it around.

Summary

In this chapter, we quickly skimmed through the very interesting domain of continuous control using RL methods, and we checked three different algorithms on one problem of a four-legged robot. In our training, we used an emulator, but there are real models of this robot made by the Ghost Robotics company. (You can check out the cool video on YouTube: <https://youtu.be/bnKOeMoibLg>.) We applied three training methods to this environment: A2C, DDPG, and D4PG (which showed the best results).

In the next chapter, we will continue exploring the continuous action domain and check a different set of improvements: *trust region extension*.

16

Trust Region Methods

In this chapter, we will take a look at the approaches used to improve the stability of the stochastic policy gradient method. Some attempts have been made to make the policy improvement more stable, and in this chapter, we will focus on three methods:

- **Proximal policy optimization (PPO)**
- **Trust region policy optimization (TRPO)**
- **Advantage actor-critic (A2C) using Kronecker-factored trust region (ACKTR)**.

In addition, we will compare these methods to a relatively new off-policy method called **soft actor-critic (SAC)**, which is the evolution of the deep **deterministic policy gradients (DDPG)** method described in *Chapter 15*. To compare them to the A2C baseline, we will use several environments from the so-called “locomotion gym environments” – environments shipped with Farama Gymnasium (using MuJoCo and PyBullet). We also will do a head-to-head comparison between PyBullet and MuJoCo (which we discussed in *Chapter 15*).

The purpose of the methods that we will look at is to improve the stability of the policy update during training. There is a dilemma: on the one hand, we’d like to train as fast as we can, making large steps during the **stochastic gradient descent (SGD)** update. On the other hand, a large update of the policy is usually a bad idea. The policy is a very nonlinear thing, so a large update could ruin the policy we’ve just learned.

Things can become even worse in the **reinforcement learning (RL)** landscape because you can’t recover from making a bad update to the policy by subsequent updates.

Instead, the bad policy will provide bad experience samples that we will use in subsequent training steps, which could break our policy completely. Thus, we want to avoid making large updates by all means possible. One of the naïve solutions would be to use a small learning rate to take baby steps during SGD, but this would significantly slow down the convergence.

To break this vicious cycle, several attempts have been made by researchers to estimate the effect that our policy update is going to have in terms of future outcomes. One of the popular approaches is the **trust region optimization** extension, which constrains the steps taken during the optimization to limit its effect on the policy. The main idea is to prevent a dramatic policy update during the loss optimization by checking the **Kullback-Leibler (KL)** divergence between the old and the new policy. Of course, this is an informal explanation, but it can help you understand the idea, especially as those methods are quite math-heavy (especially TRPO).

Environments

Previous editions of this book used the Roboschool library from OpenAI (<https://openai.com/index/roboschool>) to illustrate trust region methods. But eventually, OpenAI deprecated Roboschool and stopped its support.

But environments are still available in other sources:

- **PyBullet**: The physics simulator we experimented with in the previous chapter, which includes a wide variety of environments that support Gym. PyBullet may be a bit outdated (the latest release was in 2022), but it is still workable with a bit of hacking.
- **Farama Gymnasium MuJoCo environments**: MuJoCo is a physics simulator that we discussed in *Chapter 15*. After it was made open source, MuJoCo was adopted in various products, including Gymnasium, which ships several environments: <https://gymnasium.farama.org/environments/mujoco/>.

In this chapter, we will explore two problems: HalfCheetah-v4, which models a two-legged creature, and Ant-v4, which has four legs. Their state and action spaces are very similar to the Minitaur environment that we saw in *Chapter 15*: the state includes characteristics from joints, and the actions are activations of those joints. The goal for each problem is to move as far as possible, minimizing the energy spent. The following figure shows screenshots of the two environments:

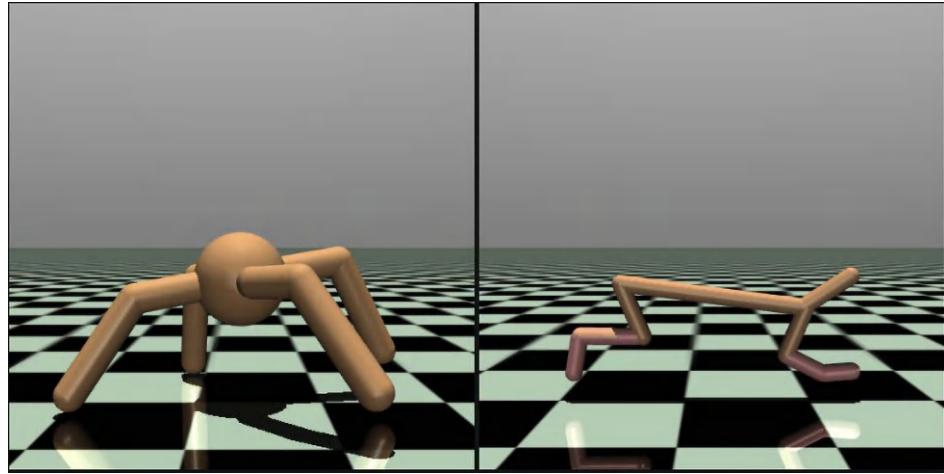


Figure 16.1: Screenshots of the cheetah and ant environments

In our experiment, we'll use PyBullet and MuJoCo to do a comparison of both simulators in terms of speed and training dynamics (however, note that the internal structure of the PyBullet and MuJoCo environments might be different, and so the comparison of training dynamics may not always be reliable). To install the Gymnasium with MuJoCo extensions, you need to run the following command in your Python environment:

```
pip install gymnasium[mujoco]==0.29.0.
```

The A2C baseline

To establish the baseline results, we will use the A2C method in a very similar way to the previous chapter. The complete source code is in the `Chapter16/01_train_a2c.py` and `Chapter16/lib/model.py` files. There are a few differences between this baseline and the version we used before:

- 16 parallel environments are used to gather experience during the training.
- They differ in model structure and the way that we perform exploration.

Implementation

To illustrate the differences between this baseline and the previously discussed version, let's look at the model and the agent classes.

The actor and critic are placed in separate networks without sharing weights. They follow the approach used in *Chapter 15*, with our critic estimating the mean and the variance for the actions. However, now, variance is not a separate head of the base network; it is just a single parameter of the model. This parameter will be adjusted during the training by SGD, but it doesn't depend on the observation.

The actor network has two hidden layers of 64 neurons, each with tanh nonlinearity (to push the output in the $-1 \dots 1$ range). The variance is modeled as a separate network parameter and is interpreted as a logarithm of the standard deviation:

```
HID_SIZE = 64

class ModelActor(nn.Module):
    def __init__(self, obs_size: int, act_size: int):
        super(ModelActor, self).__init__()

        self.mu = nn.Sequential(
            nn.Linear(obs_size, HID_SIZE),
            nn.Tanh(),
            nn.Linear(HID_SIZE, HID_SIZE),
            nn.Tanh(),
            nn.Linear(HID_SIZE, act_size),
            nn.Tanh(),
        )
        self.logstd = nn.Parameter(torch.zeros(act_size))

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.mu(x)
```

The critic network also has two hidden layers of the same size, with one single output value, which is the estimation of $V(s)$, which is a discounted value of the state:

```
class ModelCritic(nn.Module):
    def __init__(self, obs_size: int):
        super(ModelCritic, self).__init__()

        self.value = nn.Sequential(
            nn.Linear(obs_size, HID_SIZE),
            nn.ReLU(),
            nn.Linear(HID_SIZE, HID_SIZE),
            nn.ReLU(),
            nn.Linear(HID_SIZE, 1),
        )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.value(x)
```

The agent that converts the state into the action also works by simply obtaining the predicted mean from the state and applying the noise with variance, dictated by the current value of the `logstd` parameter:

```
class AgentA2C(ptan.agent.BaseAgent):
    def __init__(self, net, device: torch.device):
        self.net = net
        self.device = device

    def __call__(self, states: ptan.agent.States, agent_states: ptan.agent.AgentStates):
        states_v = ptan.agent.float32_preprocessor(states)
        states_v = states_v.to(self.device)

        mu_v = self.net(states_v)
        mu = mu_v.data.cpu().numpy()
        logstd = self.net.logstd.data.cpu().numpy()
        rnd = np.random.normal(size=logstd.shape)
        actions = mu + np.exp(logstd) * rnd
        actions = np.clip(actions, -1, 1)
        return actions, agent_states
```

Results

The training utility `01_train_a2c.py` could be started in two different modes: with PyBullet as the physics simulator (without any extra command-line options) or with MuJoCo (if the `-mujoco` parameter is given).

By default, the HalfCheetah environment is used, which simulates a flat two-legged creature that can jump around on its legs. With `-e ant`, you can switch to the Ant environment, which is a 3-dimensional 4-legged spider. You can also experiment with other environments shipped with Gymnasium and PyBullet, but this will require tweaking the `common.py` module.

Results for HalfCheetah on PyBullet are shown in *Figure 16.2*. Performance on my machine (using the GPU) was about 1,600 frames per second during the training, so 100M training steps took 20 hours in total.

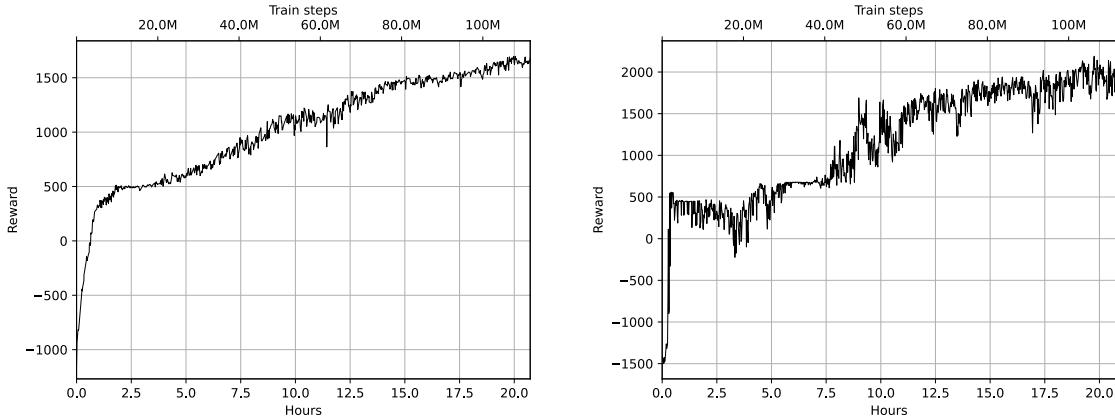


Figure 16.2: The reward during training (left) and test reward (right) for HalfCheetah on PyBullet

The dynamics suggest that the policy could be further improved with more time given to optimization, but for our purpose of method comparison, it should be enough. Of course, if you're curious and have plenty of time, you can run this for longer and find the point when the policy stops improving. According to research papers, HalfCheetah has a maximum score of around 4,000-5,000.

To use MuJoCo as a physics simulation engine, training has to be started with the `-mujoco` command-line option. MuJoCo has a performance of 5,100 frames per second, which is 3 times faster than PyBullet, which is really nice. In addition, the training has much better dynamics, so in 90M training steps (which took about 5 hours) the model got a reward of 4,500. Plots for MuJoCo are shown in *Figure 16.3*:

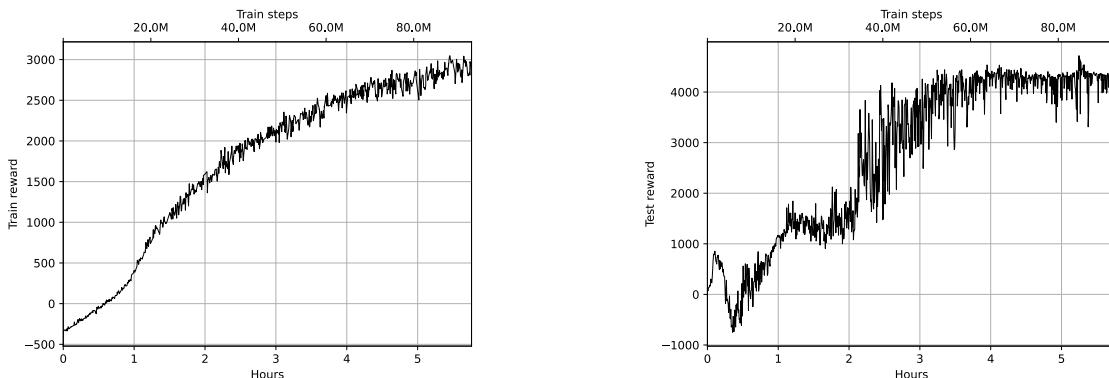


Figure 16.3: The reward during training (left) and the test reward (right) for HalfCheetah on MuJoCo

The difference could be explained by a more accurate simulation, but could also be attributed to the difference in the observation space and the underlying model differences. PyBullet's model has 26 parameters provided to the agent as observations, while MuJoCo has only 17, so those models are not identical.

To test our model in the Ant environment, the `-e ant` command-line option has to be passed to the training process. This model is more complex (due to the 3D nature of the model and more joints being used), so the simulation is slower. On PyBullet, the speed is around 1,400 frames per second. On MuJoCo, the speed is 2,500.

The MuJoCo Ant environment also has an additional check for “healthiness” – if the simulated creature is inclined more than a certain degree, the episode is terminated. This check is enabled by default and has a very negative effect on the training – in the early stage of the training, our method has no chance of figuring out how to make the ant stand on its legs. The reward in the environment is the distance traveled, but with this early termination, our training has no chance of discovering this. As a result, the training process got stuck forever in local minima without making progress. To overcome this, we need to disable this healthiness check by passing the `-no-unhealthy` command-line option (which only has to be done for MuJoCo training).



In principle, you can implement more advanced exploration methods, such as the OU process (discussed in *Chapter 15*) or other methods (covered in *Chapter 18*) to address the issue we just discussed.

The results of the training in the Ant environment are shown in *Figure 16.4* and *Figure 16.5*.

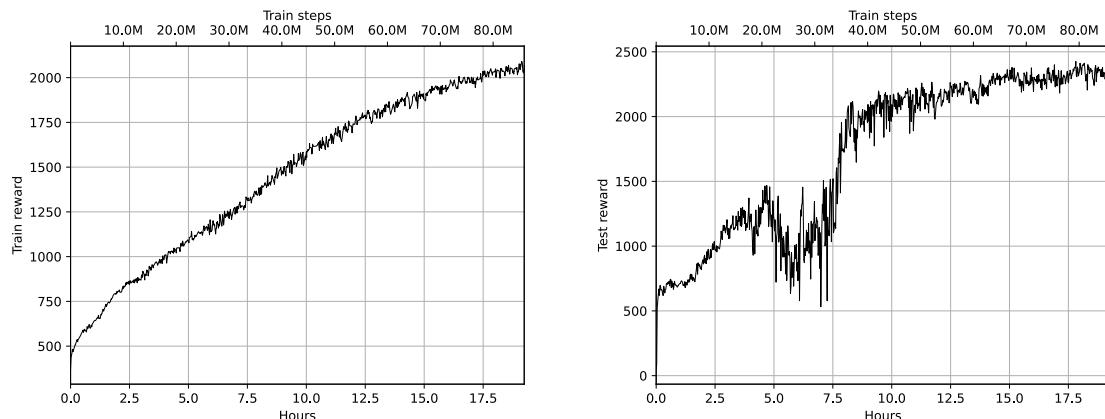


Figure 16.4: The reward during training (left) and the test reward (right) for Ant on PyBullet

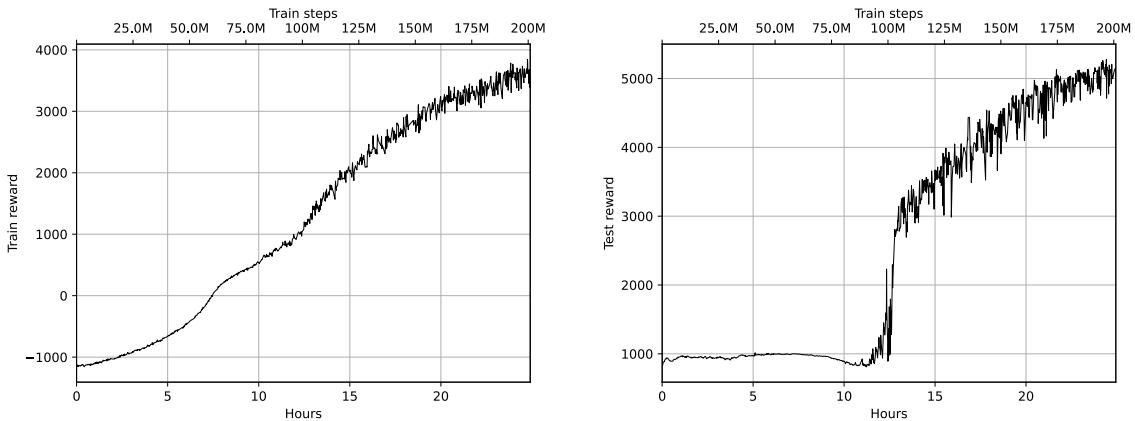


Figure 16.5: The reward during training (left) and the test reward (right) for Ant on MuJoCo

As you can see from the MuJoCo plots in *Figure 16.5*, the testing reward had almost no increase for the first 100M steps of training, but then grew to a score of 5,000 (the best model got 5,380 on testing). This result is quite impressive. According to the <https://paperswithcode.com> website, the state of the art for Ant MuJoCo environment is 4,362.9, obtained by IQ-Learn in 2021: <https://paperswithcode.com/sota/mujoco-games-on-ant>.

Video recording

As in the previous chapter, there is a utility that can benchmark the trained model and record a video of the agent in action. As all the methods in this chapter share the same actor network, the tool is universal for all the methods illustrated here: `02_play.py`.

You need to pass the model file stored in the `saves` directory during training, change the environment using the `-e ant` command line, and enable the MuJoCo engine with the `-mujoco` parameter. This is important because the same environments in PyBullet and MuJoCo have different amounts of observations, and so the physics engine has to match to the model.

You can find the individual videos for the best A2C models as follows:

- HalfCheetah on PyBullet (score 2,189): <https://youtu.be/f3ZhjnORQm0>
- HalfCheetah on MuJoCo (score 4,718): <https://youtube.com/shorts/SpaWbS0hM8I>
- Ant on PyBullet (score 2,425): https://youtu.be/SIUM_Q24zSk
- Ant on MuJoCo (score 5,380): <https://youtube.com/shorts/mapOraGKtG0>

PPO

The PPO method came from the OpenAI team, and it was proposed after TRPO, which is from 2015. However, we will start with PPO because it is much simpler than TRPO. It was first proposed in the 2017 paper named *Proximal Policy Optimization Algorithms* by Schulman et al. [Sch+17].

The core improvement over the classic A2C method is changing the formula used to estimate the policy gradients. Instead of using the gradient of the logarithm probability of the action taken, the PPO method uses a different objective: the ratio between the new and the old policy scaled by the advantages.

In math form, the A2C objective could be written like this

$$J_\theta = \mathbb{E}_t [\nabla_\theta \log \pi_\theta(a_t|s_t) A_t]$$

which means our gradient on model θ is estimated as the logarithm of the policy π multiplied by the advantage A .

The new objective proposed in PPO is the following:

$$J_\theta = \mathbb{E}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} A_t \right]$$

The reason for changing the objective is the same as with the cross-entropy method covered in *Chapter 4*: importance sampling. However, if we just start to blindly maximize this value, it may lead to a very large update to the policy weights. To limit the update, the clipped objective is used. If we write the ratio between the new and the old policy as $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$, the clipped objective could be written as

$$J_\theta^{clip} = \mathbb{E}_t [\min(r_t(\theta)A_t, clip(r_t(\theta), 1 - \epsilon, 1 + \epsilon)A_t)]$$

This objective limits the ratio between the old and the new policy to be in the interval $[1 - \epsilon, 1 + \epsilon]$, so by varying ϵ , we can limit the size of the update.

Another difference from the A2C method is the way that we estimate the advantage. In the A2C paper, the advantage obtained from the finite-horizon estimation of T steps is in the form

$$A_t = -V(s_t) + r_t + \gamma r_{t+1} + \dots + \gamma^{T-t+1} r_{T-1} + \gamma^{T-t} V(s_T)$$

In the PPO paper, the authors used a more general estimation

$$A_t = \sigma_t + (\gamma\lambda)\sigma_{t+1} + (\gamma\lambda)^2\sigma_{t+2} + \dots + (\gamma\lambda)^{T-t+1}\sigma_{T-1}$$

where $\sigma_t = r_t + \gamma V(s_{t+1}) - V(s_t)$.

The original A2C estimation is a special case of the proposed method with $\lambda = 1$. The PPO method also uses a slightly different training procedure: a long sequence of samples is obtained from the environment and then the advantage is estimated for the whole sequence before several epochs of training are performed.

Implementation

The code of the sample is placed in two source code files: `Chapter16/04_train_ppo.py` and `Chapter16/lib/model.py`.

The actor, the critic, and the agent classes are exactly the same as we had in the A2C baseline.

The differences are in the training procedure and the way that we calculate advantages, but let's start with the hyperparameters:

```
GAMMA = 0.99
GAE_LAMBDA = 0.95

TRAJECTORY_SIZE = 2049
LEARNING_RATE_ACTOR = 1e-5
LEARNING_RATE_CRITIC = 1e-4

PPO_EPS = 0.2
PPO_EPOCHES = 10
PPO_BATCH_SIZE = 64
```

The value of `GAMMA` is already familiar, but `GAE_LAMBDA` is the new constant that specifies the lambda factor in the advantage estimator. The authors chose to use a value of 0.95 in the PPO paper.

The method assumes that a large number of transitions will be obtained from the environment for every subiteration. (As mentioned previously in this section, when describing PPO, during training, it performs several epochs over the sampled training batch.) We also use two different optimizers for the actor and the critic (as they have no shared weights).

For every batch of `TRAJECTORY_SIZE` samples, we perform `PPO_EPOCHES` iterations of the PPO objective, with mini-batches of 64 samples. The value `PPO_EPS` specifies the clipping value for the ratio of the new and the old policy.

The following function takes the trajectory with steps and calculates advantages for the actor and reference values for the critic training. Our trajectory is not a single episode, but could be several episodes concatenated together:

```
def calc_adv_ref(trajectory: tt.List[ptan.experience.Experience],
                 net_crt: model.ModelCritic, states_v: torch.Tensor, gamma: float,
                 gae_lambda: float, device: torch.device):
    values_v = net_crt(states_v)
    values = values_v.squeeze().data.cpu().numpy()
```

As the first step, we ask the critic to convert states into values.

The next loop joins the values obtained and experience points:

```
last_gae = 0.0
result_adv = []
result_ref = []
for val, next_val, (exp,) in zip(
    reversed(values[:-1]), reversed(values[1:]), reversed(trajectory[:-1])):
    ...
```

For every trajectory step, we need the current value (obtained from the current state) and the value for the subsequent step (to perform the estimation using the Bellman equation). We also traverse the trajectory in reverse order in order to calculate more recent values of the advantage in one step.

```
if exp.done_trunc:
    delta = exp.reward - val
    last_gae = delta
else:
    delta = exp.reward + gamma * next_val - val
    last_gae = delta + gamma * gae_lambda * last_gae
```

In every step, our action depends on the `done_trunc` flag for this step. If this is the terminal step of the episode, we have no prior reward to take into account. (Remember, we're processing the trajectory in reverse order.) So, our value of `delta` in this step is just the immediate reward minus the value predicted for the step. If the current step is not terminal, `delta` will be equal to the immediate reward plus the discounted value from the subsequent step, minus the value for the current step.

In the classic A2C method, this delta was used as an advantage estimation, but here, the smoothed version is used, so the advantage estimation (tracked in the `last_gae` variable) is calculated as the sum of deltas with the discount factor γ^λ .

The goal of the function is to calculate advantages and reference values for the critic, so we save them in lists:

```
result_adv.append(last_gae)
result_ref.append(last_gae + val)
```

At the end of the function, we convert values to tensors and return them:

```
adv_v = torch.FloatTensor(np.asarray(list(reversed(result_adv))))
ref_v = torch.FloatTensor(np.asarray(list(reversed(result_ref))))
return adv_v.to(device), ref_v.to(device)
```

In the training loop, we gather a trajectory of the desired size using the `ExperienceSource(steps_count=1)` class from the PTAN library. This configuration provides us with individual steps from the environment in `Experience` dataclass instances, containing the state, action, reward, and termination flag. The following is the relevant part of the training loop:

```
trajectory.append(exp)
if len(trajectory) < TRAJECTORY_SIZE:
    continue

traj_states = [t[0].state for t in trajectory]
traj_actions = [t[0].action for t in trajectory]
traj_states_v = torch.FloatTensor(np.asarray(traj_states))
traj_states_v = traj_states_v.to(device)
traj_actions_v = torch.FloatTensor(np.asarray(traj_actions))
traj_actions_v = traj_actions_v.to(device)
traj_adv_v, traj_ref_v = common.calc_adv_ref(
    trajectory, net_crt, traj_states_v, GAMMA, GAE_LAMBDA, device=device)
```

When we've got a trajectory that's large enough for training (which is given by the `TRAJECTORY_SIZE` hyperparameter), we convert states and actions taken into tensors and use the already-described function to obtain advantages and reference values. Although our trajectory is quite long, the observations of our environments are small enough, so it's fine to process our batch in one step. In the case of Atari frames, such a batch could cause a GPU memory error.

In the next step, we calculate the logarithm of the probability of the actions taken. This value will be used as $\pi_{\theta_{old}}$ in the objective of PPO. Additionally, we normalize the advantage's mean and variance to improve the training stability:

```
mu_v = net_act(traj_states_v)
old_logprob_v = model.calc_logprob(mu_v, net_act.logstd, traj_actions_v)

traj_adv_v = traj_adv_v - torch.mean(traj_adv_v)
traj_adv_v /= torch.std(traj_adv_v)
```

The two subsequent lines drop the last entry from the trajectory to reflect the fact that our advantages and reference values are one step shorter than the trajectory length (as we shifted values in the loop inside the `calc_adv_ref` function):

```
trajectory = trajectory[:-1]
old_logprob_v = old_logprob_v[:-1].detach()
```

When all the preparations have been done, we perform several epochs of training on our trajectory. For every batch, we extract the portions from the corresponding arrays and do the critic and the actor training separately:

```
for epoch in range(PPO_EPOCHES):
    for batch_ofs in range(0, len(trajectory), PPO_BATCH_SIZE):
        batch_l = batch_ofs + PPO_BATCH_SIZE
        states_v = traj_states_v[batch_ofs:batch_l]
        actions_v = traj_actions_v[batch_ofs:batch_l]
        batch_adv_v = traj_adv_v[batch_ofs:batch_l]
        batch_adv_v = batch_adv_v.unsqueeze(-1)
        batch_ref_v = traj_ref_v[batch_ofs:batch_l]
        batch_old_logprob_v = old_logprob_v[batch_ofs:batch_l]
```

To train the critic, all we need to do is calculate the **mean squared error (MSE)** loss with the reference values calculated beforehand:

```
opt_crt.zero_grad()
value_v = net_crt(states_v)
loss_value_v = F.mse_loss(value_v.squeeze(-1), batch_ref_v)
loss_value_v.backward()
opt_crt.step()
```

In the actor training, we minimize the negated clipped objective:

$$\mathbb{E}_t[\min(r_t(\theta)A_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon)A_t)]$$

$$\text{where } r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$$

To achieve this, we use the following code:

```

opt_act.zero_grad()
mu_v = net_act(states_v)
logprob_pi_v = model.calc_logprob(mu_v, net_act.logstd, actions_v)
ratio_v = torch.exp(logprob_pi_v - batch_old_logprob_v)
surr_obj_v = batch_adv_v * ratio_v
c_ratio_v = torch.clamp(ratio_v, 1.0 - PPO_EPS, 1.0 + PPO_EPS)
clipped_surr_v = batch_adv_v * c_ratio_v
loss_policy_v = -torch.min(surr_obj_v, clipped_surr_v).mean()
loss_policy_v.backward()
opt_act.step()
```

Results

After being trained in both our test environments, the PPO method has shown much faster convergence than the A2C method. On HalfCheetah using PyBullet, PPO reached an average training reward of 1,800 and 2,500 during the testing after 8 hours of training and 25M training steps. A2C got lower results after 110M steps and 20 hours. *Figure 16.6* shows the comparison plots.

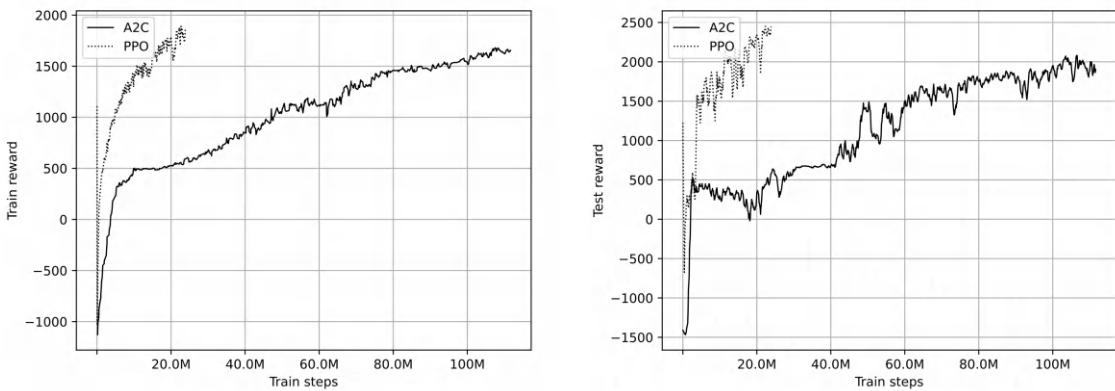


Figure 16.6: The reward during training (left) and the test reward (right) for HalfCheetah on PyBullet

But on HalfCheetah using MuJoCo, the situation is the opposite – PPO growth was much slower, and I stopped it after 50M training steps (12 hours). *Figure 16.7* shows the plots.

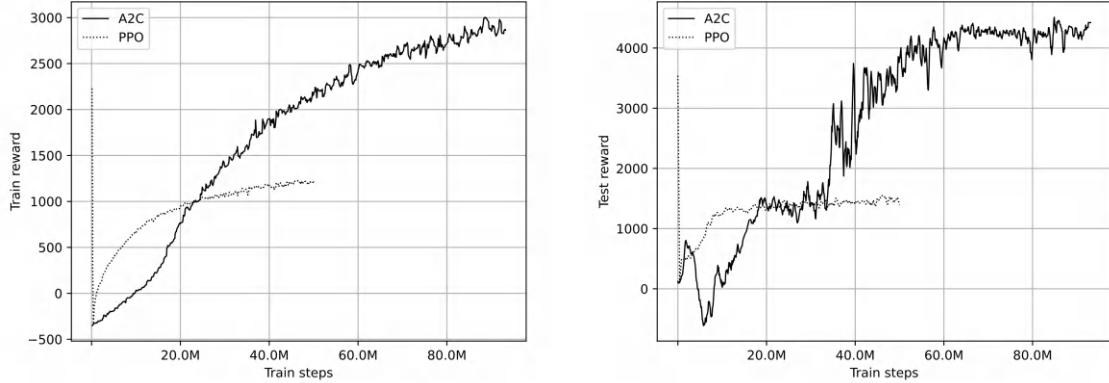


Figure 16.7: The reward during training (left) and the test reward (right) for HalfCheetah on MuJoCo

After checking the video of the model (links are provided later in this section), we might guess the reason for the low score – our agent learned how to flip the cheetah on its back and move forward in this position. During training, it wasn't able to get from this suboptimal “local maximum.” Most likely, running the training several times might yield a better policy. Another approach to solving this might be to optimize hyperparameters. Again, this is something you can try experimenting with.

In the Ant environment, PPO was better on both PyBullet and MuJoco and was able to reach the same level of reward almost twice as fast as A2C. This comparison is shown in the plots in *Figure 16.8* and *Figure 16.9*.

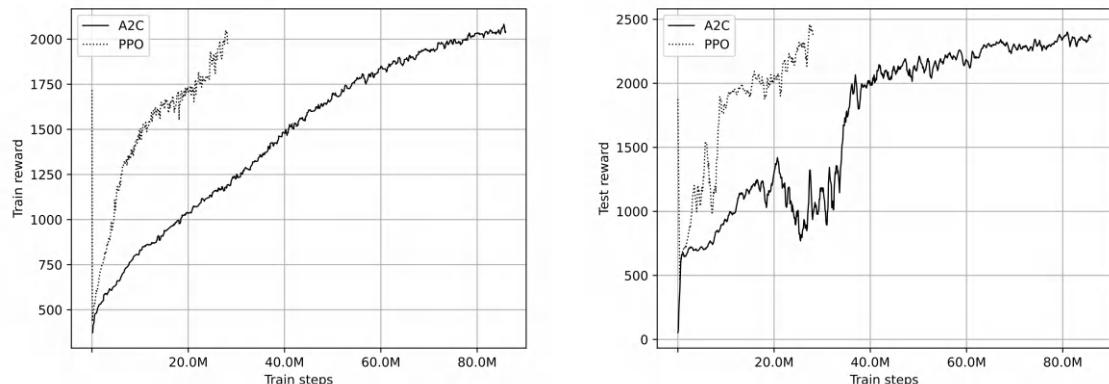


Figure 16.8: The reward during training (left) and the test reward (right) for Ant on PyBullet

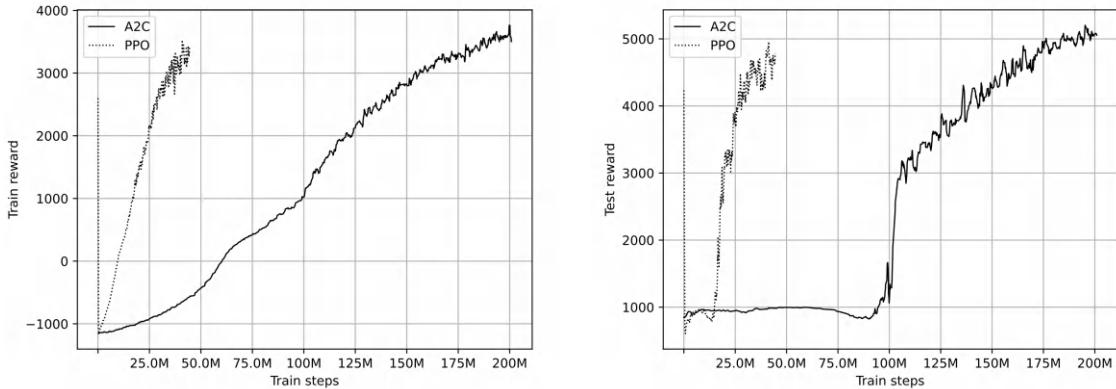


Figure 16.9: The reward during training (left) and the test reward (right) for Ant on MuJoCo

As before, you can use the `02_play.py` utility to benchmark saved models and record videos of the learned policy in action. This is the list of the best models for my training experiments:

- HalfCheetah on PyBullet (score 2,567): <https://youtu.be/Rai-smfyE>. The agent learned how to do long jumps with the back leg.
- HalfCheetah on MuJoCo (score 1,623): <https://youtube.com/shorts/VcyzNtbVzd4>. Quite a funny video: the cheetah flips on its back and moves forward this way.
- Ant on PyBullet (score 2,560): https://youtu.be/8lty_MdjnfS. The Ant policy is much better than A2C – it steadily moves forward.
- Ant on MuJoCo (score 5,108): https://youtube.com/shorts/AcXxH2f_KWs. This model is much faster; most likely, the weight of the ant in the MuJoCo model is lower than in PyBullet.

TRPO

TRPO was proposed in 2015 by Berkeley researchers in a paper by Schulman et al., called *Trust region policy optimization* [Sch15]. This paper was a step towards improving the stability and consistency of stochastic policy gradient optimization and has shown good results on various control tasks.

Unfortunately, the paper and the method are quite math-heavy, so it can be hard to understand the details. The same could be said about the implementation, which uses the conjugate gradients method to efficiently solve the constrained optimization problem.

As the first step, the TRPO method defines the discounted visitation frequencies of the state as follows:

$$\rho_\pi(s) = P(s_0 = s) + \gamma P(s_1 = s) + \gamma^2 P(s_2 = s) + \dots$$

In this equation, $P(s_i = s)$ equals the sampled probability of state s to be met at position i of the sampled trajectories.

Then, TRPO defines the optimization objective as

$$L_\pi(\tilde{\pi}) = \eta(\pi) + \sum_s \rho_\pi(s) \sum_a \tilde{\pi}(a|s) A_\pi(s, a)$$

where

$$\eta(\pi) = \mathbb{E} \left[\sum_{t=0}^{\infty} \gamma^t r(s_t) \right]$$

is the expected discounted reward of the policy and $\tilde{\pi} = \arg \max_a A_\pi(s, a)$ defines the deterministic policy. To address the issue of large policy updates, TRPO defines the additional constraint on the policy update, which is expressed as the maximum KL divergence between the old and the new policies, which could be written as

$$\bar{D}_{KL}(\pi_{\theta_{old}}, \pi_\theta) \leq \delta$$

As a reminder, KL divergence measures the similarity between probability distributions and is calculated as follows:

$$D_{KL}(P\|Q) = - \sum_i p_i \log q_i$$

We met KL divergence in *Chapter 4* and *Chapter 11*.

Implementation

Most of the TRPO implementations available on GitHub, or in other open source repositories, are very similar to each other, probably because all of them grew from the original John Schulman TRPO implementation here: https://github.com/joschu/modular_rl. My version of TRPO is also not very different and uses the core functions that implement the conjugate gradient method (used by TRPO to solve the constrained optimization problem) from this repository: <https://github.com/ikostrikov/pytorch-trpo>.

The complete example is in `03_train_trpo.py` and `lib/trpo.py`, and the training loop is very similar to the PPO example: we sample the trajectory of transitions of the predefined length and calculate the advantage estimation using the smoothed formula discussed in the PPO section (historically, this estimator was proposed first in the TRPO paper.)

Next, we do one training step of the critic using MSE loss with the calculated reference value, and one step of the TRPO update, which consists of finding the direction we should go in by using the conjugate gradients method and doing a linear search in this direction to find a step that preserves the desired KL divergence.

The following is the piece of the training loop that carries out both those steps:

```
opt_crt.zero_grad()
value_v = net_crt(traj_states_v)
loss_value_v = F.mse_loss(value_v.squeeze(-1), traj_ref_v)
loss_value_v.backward()
opt_crt.step()
```

To perform the TRPO step, we need to provide two functions: the first will calculate the loss of the current actor policy, which uses the same ratio as the PPO of the new and the old policies multiplied by the advantage estimation. The second function has to calculate KL divergence between the old and the current policy:

```
def get_loss():
    mu_v = net_act(traj_states_v)
    logprob_v = model.calc_logprob(mu_v, net_act.logstd, traj_actions_v)
    dp_v = torch.exp(logprob_v - old_logprob_v)
    action_loss_v = -traj_adv_v.unsqueeze(dim=-1)*dp_v
    return action_loss_v.mean()

def get_kl():
    mu_v = net_act(traj_states_v)
    logstd_v = net_act.logstd
    mu0_v = mu_v.detach()
    logstd0_v = logstd_v.detach()
    std_v = torch.exp(logstd_v)
    std0_v = std_v.detach()
    v = (std0_v ** 2 + (mu0_v - mu_v) ** 2) / (2.0 * std_v ** 2)
    kl = logstd_v - logstd0_v + v - 0.5
    return kl.sum(1, keepdim=True)

trpo.trpo_step(net_act, get_loss, get_kl, args.maxkl,
               TRPO_DAMPING, device=device)
```

In other words, the PPO method is TRPO that uses the simple clipping of the policy ratio to limit the policy update, instead of the complicated conjugate gradients and line search.

Results

TRPO in the HalfCheetah environment was able to reach better rewards than PPO and A2C. In *Figure 16.10*, the results from PyBullet training is shown. On MuJoCo, the results are even more impressive – the best reward was over 5,000. The plots for MuJoCo are shown in *Figure 16.11*:

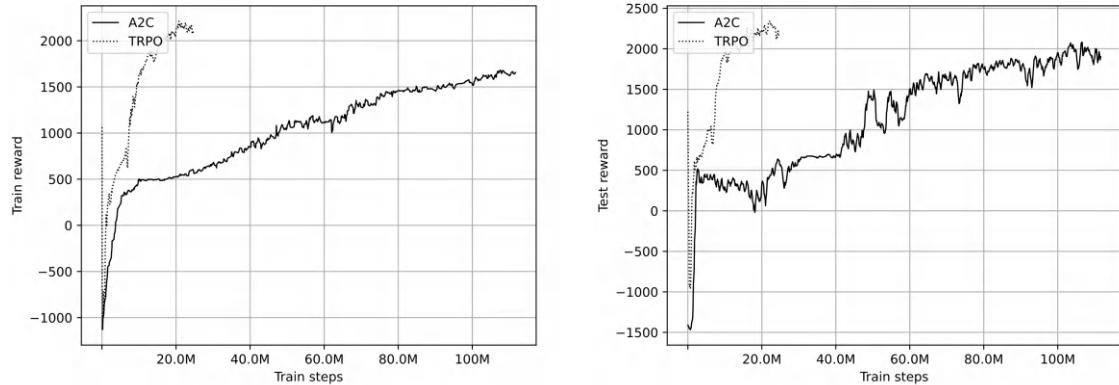


Figure 16.10: The reward during training (left) and test reward (right) for HalfCheetah on PyBullet

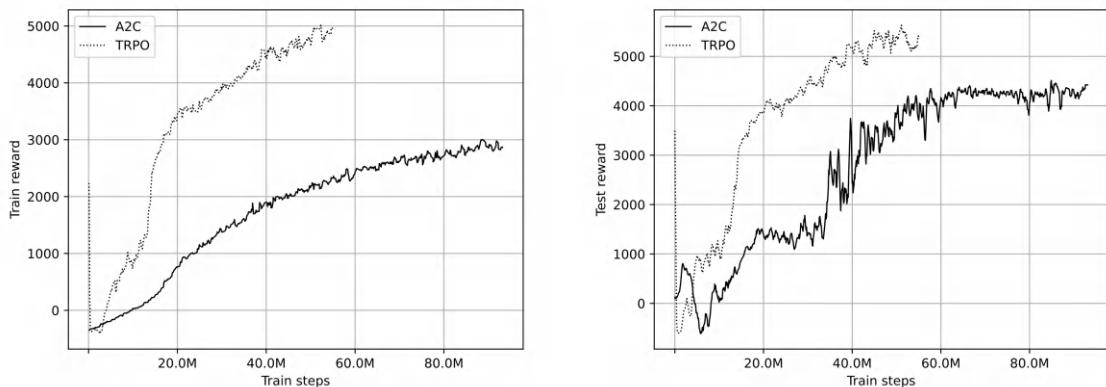


Figure 16.11: The reward during training (left) and the test reward (right) for HalfCheetah on MuJoCo

Unfortunately, the Ant environment shows much less stable convergence. The plots shown in *Figure 16.12* and *Figure 16.13* compare the train and test rewards on A2C and TRPO:

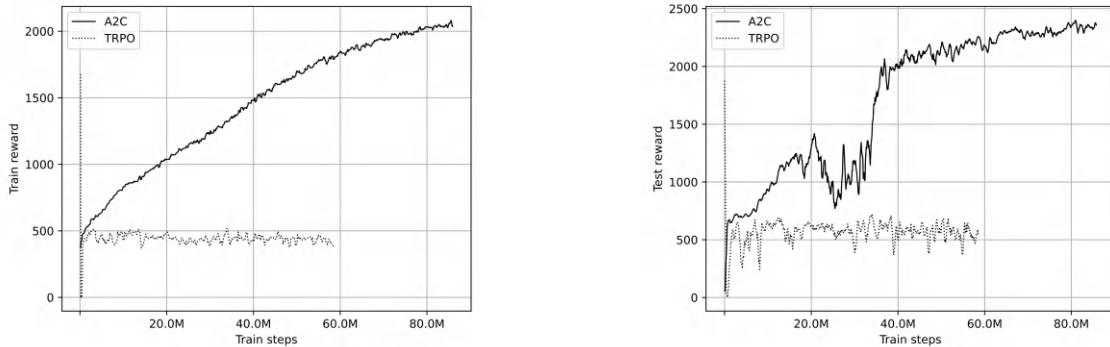


Figure 16.12: The reward during training (left) and the test reward (right) for Ant on PyBullet

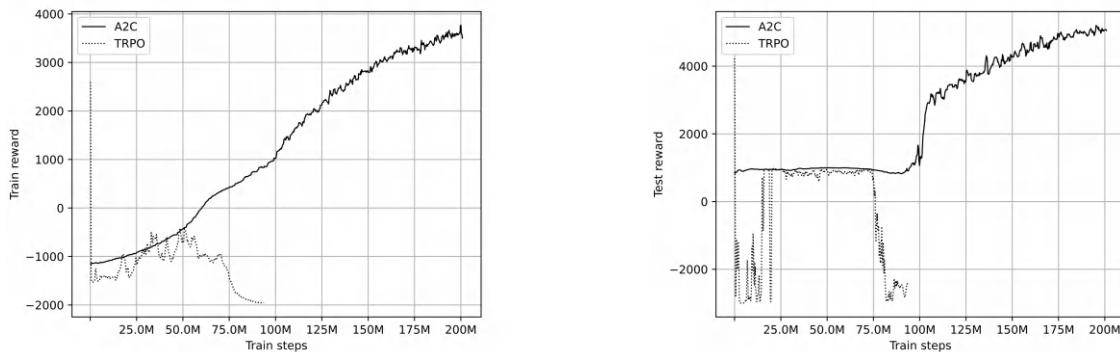


Figure 16.13: The reward during training (left) and the test reward (right) for Ant on MuJoCo

Video recordings of the best actions could be done in the same way as before. Here are some videos for the best TRPO models:

- HalfCheetah on PyBullet (score 2,419): <https://youtu.be/NIfkt2lVT74>. Front leg joints are not used.
- HalfCheetah on MuJoCo (score 5,753): <https://youtube.com/shorts/FLM2t-XWDLc?feature=share>. This is a really fast Cheetah!
- Ant on PyBullet (score 834): <https://youtu.be/Ny1WPBVluNQ>. The training got stuck in a “stand still” local minimum.
- Ant on MuJoCo (score 993): <https://youtube.com/shorts/9sybZGvXQFs>. The same as PyBullet – the agent just stands still and does not move anywhere.

ACKTR

The third method that we will compare, ACKTR, uses a different approach to address SGD stability. In the paper by Wu et al. called *Scalable trust-region method for deep reinforcement learning using Kronecker-factored approximation*, published in 2017 [Wu+17], the authors combined the second-order optimization methods and trust region approach.

The idea of the second-order methods is to improve the traditional SGD by taking the second-order derivatives of the optimized function (in other words, its curvature) to improve the convergence of the optimization process. To make things more complicated, working with the second-order derivatives usually requires you to build and invert a Hessian matrix, which can be prohibitively large, so the practical methods typically approximate it in some way. This area is currently very active in research because developing robust, scalable optimization methods is very important for the whole machine learning domain.

One of the second-order methods is called **Kronecker-factored approximate curvature (K-FAC)**, which was proposed by James Martens and Roger Grosse in their paper *Optimizing neural networks with Kronecker-factored approximate curvature*, published in 2015 [MG15]. However, a detailed description of this method is well beyond the scope of this book.

Implementation

There are not very many implementations of this method available, and none of them are part of PyTorch (unfortunately). As far as I know, there are two versions of the K-FAC optimizer that work with PyTorch; one from Ilya Kostrikov (<https://github.com/ikostrikov/pytorch-a2c-ppo-acktr>) and one from Nicholas Gao (<https://github.com/n-gao/pytorch-kfac>). I've experimented only with the first one; you can give the second one a try. There is a version of K-FAC available for TensorFlow, which comes with OpenAI Baselines, but porting and testing it on PyTorch can be difficult.

For my experiments, I've taken the K-FAC implementation from Kostrikov and adapted it to the existing code, which required replacing the optimizer and doing an extra `backward()` call to gather Fisher information. The critic was trained in the same way as in A2C. The complete example is in `05_train_acktr.py` and is not shown here, as it's basically the same as A2C. The only difference is that a different optimizer was used.

Results

Overall, the ACKTR method was very unstable in both environments and physics engines. It could be due to a lack of fine-tuning of hyperparameters or some bugs in the implementation.

The results of experiments on HalfCheetah are shown in *Figure 16.14* and *Figure 16.15*.

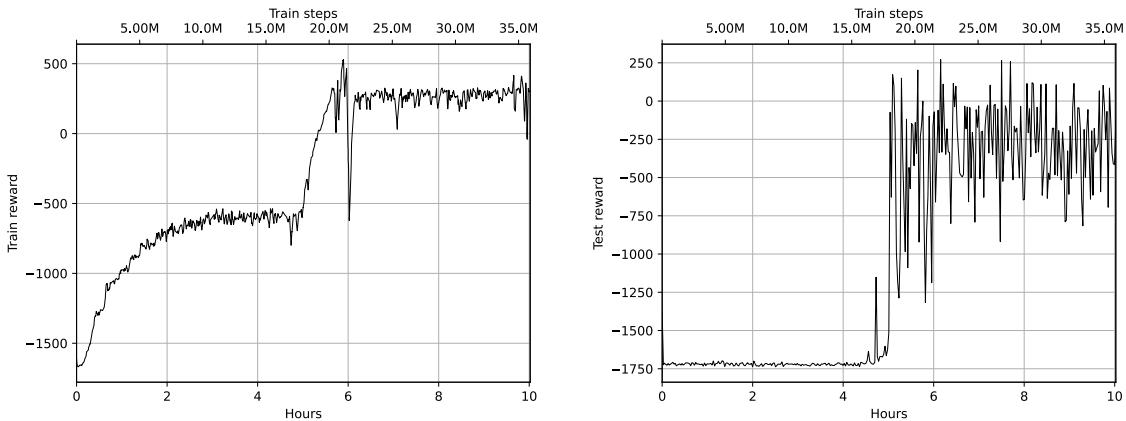


Figure 16.14: The reward during training (left) and the test reward (right) for HalfCheetah on PyBullet

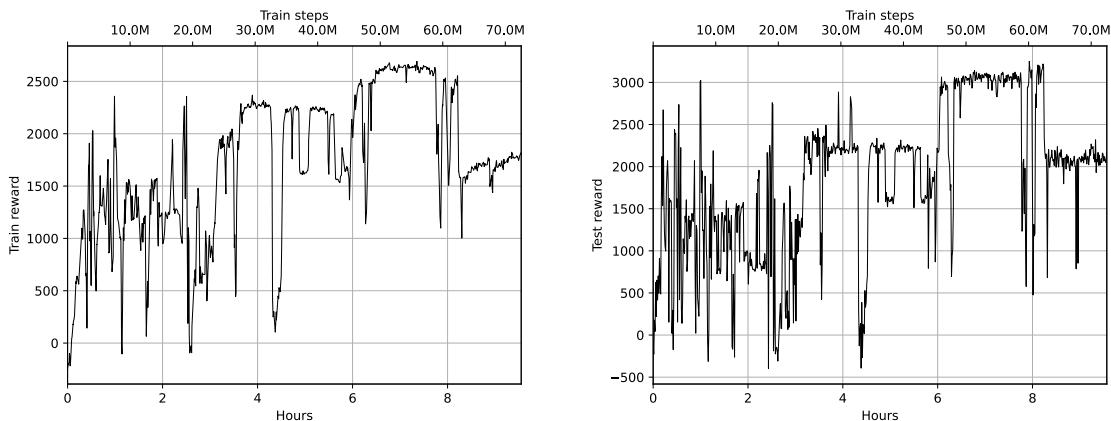


Figure 16.15: The reward during training (left) and test reward (right) for HalfCheetah on MuJoCo

In the Ant environment, the ACKTR method shows bad results on PyBullet and no reward improvements compared to training on MuJoCo. *Figure 16.16* shows plots for PyBullet.

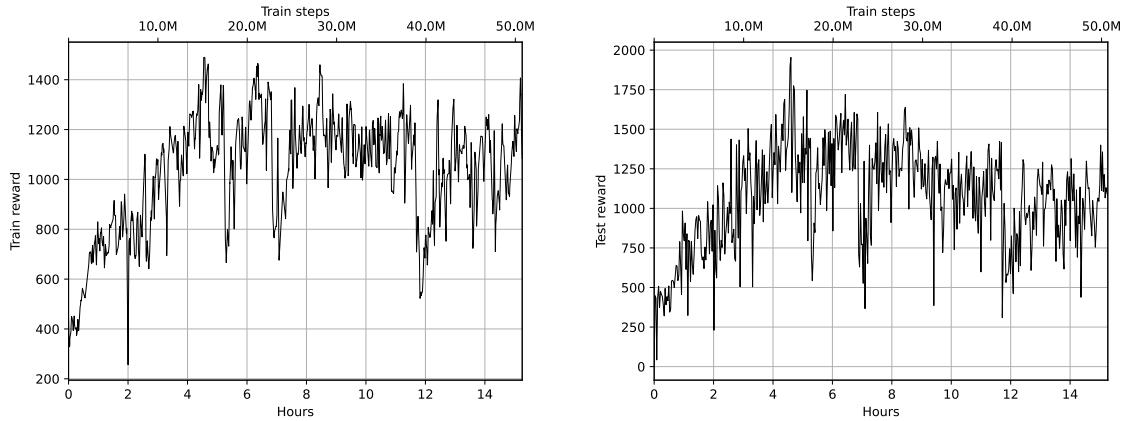


Figure 16.16: The reward during training (left) and the test reward (right) for Ant on PyBullet

SAC

In the final section, we will check our environments on a relatively new method called SAC, which was proposed by a group of Berkeley researchers and introduced in the paper *Soft actor-critic: Off-policy maximum entropy deep reinforcement learning*, by Haarnoja et al., published in 2018 [Haa+18].

At the moment, it's considered to be one of the best methods for continuous control problems and is very widely used. The core idea of the method is closer to the DDPG method than to A2C policy gradients. We will compare it directly with PPO's performance, which has been considered to be the standard in continuous control problems for a long time.

The central idea of the SAC method is **entropy regularization**, which adds a bonus reward at each timestamp that is proportional to the entropy of the policy at this timestamp. In mathematical notation, the policy we're looking for is the following:

$$\pi^* = \arg \max_a \mathbb{E}_{\tau \sim \pi} \sum_{t=0}^{\infty} \gamma^t (R(s_t, a_t, s_{t+1}) + \alpha H(\pi(\cdot | s_t)))$$

Here, $H(P) = \mathbb{E}_{x \sim P}[-\log P(x)]$ is the entropy of distribution P . In other words, we give the agent a bonus for getting into situations where the entropy is at its maximum, which is very similar to the advanced exploration methods covered in *Chapter 18*.

In addition, the SAC method incorporates the clipped double-Q trick, where, in addition to the value function, we learn two networks predicting Q-values, and choose the minimum of them for Bellman approximation. According to researchers, this helps with dealing with Q-value overestimation during training. This problem was discussed in *Chapter 8*, but was addressed differently.

So, in total, we train four networks: the policy, $\pi(s)$, value, $V(s, a)$ and two Q-networks, $Q_1(s, a)$ and $Q_2(s, a)$. For the value network, $V(s, a)$, the target network is used. So, in summary, SAC training looks like this:

- Q-networks are trained using the MSE objective by doing Bellman approximation using the target value network: $y_q(r, s') = r + \gamma V_{tgt}(s')$ (for non-terminating steps)
- The V-network is trained using the MSE objective with the following target, $y_v(s) = \min_{i=1,2} Q_i(s, \tilde{a}) - \alpha \log \pi_\theta(\tilde{a}|s)$, where \tilde{a} is sampled from policy $\pi_\theta(\cdot|s)$
- The policy network, π_θ , is trained in DDPG style by maximizing the following objective, $Q_1(s, \tilde{a}_\theta(s)) - \alpha \log \pi_\theta(\tilde{a}_\theta(s)|s)$, where \tilde{a}_θ is a sample from $\pi_\theta(\cdot|s)$

Implementation

The implementation of the SAC method is in `06_train_sac.py`. The model consists of the following networks, defined in `lib/model.py`:

- **ModelActor**: This is the same policy that we used in the previous examples in this chapter. As the policy variance is not parametrized by the state (the `logstd` field is not a network, but just a tensor), the training objective does not 100% comply with SAC. On the one hand, it might influence the convergence and performance, as the core idea of the SAC method is entropy regularization, which can't be implemented without parametrized variance. On the other hand, it decreases the number of parameters in the model. If you're curious, you can extend the example with the parametrized variance of the policy and implement a proper SAC method.
- **ModelCritic**: This is the same value network as in the previous examples.
- **ModelSACTwinQ**: These two networks take the state and action as the input and predict Q-values.

The first function implementing the method is `unpack_batch_sac()`, and it is defined in `lib/common.py`. Its goal is to take the batch of trajectory steps and calculate target values for V-networks and twin Q-networks:

```
@torch.no_grad()
def unpack_batch_sac(batch: tt.List[ptan.experience.ExperienceFirstLast],
                     val_net: model.ModelCritic, twinq_net: model.ModelSACTwinQ,
                     policy_net: model.ModelActor, gamma: float, ent_alpha: float,
                     device: torch.device):
    states_v, actions_v, ref_q_v = unpack_batch_a2c(batch, val_net, gamma, device)
```

```

mu_v = policy_net(states_v)
act_dist = distr.Normal(mu_v, torch.exp(policy_net.logstd))
acts_v = act_dist.sample()
q1_v, q2_v = twinq_net(states_v, acts_v)

ref_vals_v = torch.min(q1_v, q2_v).squeeze() - \
    ent_alpha * act_dist.log_prob(acts_v).sum(dim=1)
return states_v, actions_v, ref_vals_v, ref_q_v

```

The first step of the function uses the already defined `unpack_batch_a2c()` method, which unpacks the batch, converts states and actions into tensors, and calculates the reference for Q-networks using Bellman approximation. Once this is done, we need to calculate the reference for the V-network from the minimum of the twin Q-values minus the scaled entropy coefficient. The entropy is calculated from our current policy network. As was already mentioned, our policy has the parametrized mean value, but the variance is global and doesn't depend on the state.

In the main training loop, we use the function defined previously and do three different optimization steps: for V, for Q, and for the policy. The following is the relevant part of the training loop defined in `06_train_sac.py`:

```

batch = buffer.sample(BATCH_SIZE)
states_v, actions_v, ref_vals_v, ref_q_v = common.unpack_batch_sac(
    batch, tgt_crt_net.target_model, twinq_net, act_net, GAMMA,
    SAC_ENTROPY_ALPHA, device)

```

In the beginning, we unpack the batch to get the tensors and targets for the Q- and V-networks.

The twin Q-networks are optimized by the same target value:

```

twinq_opt.zero_grad()
q1_v, q2_v = twinq_net(states_v, actions_v)
q1_loss_v = F.mse_loss(q1_v.squeeze(), ref_q_v.detach())
q2_loss_v = F.mse_loss(q2_v.squeeze(), ref_q_v.detach())
q_loss_v = q1_loss_v + q2_loss_v
q_loss_v.backward()
twinq_opt.step()

```

The critic network is also optimized with the trivial MSE objective using the already calculated target value:

```
crt_opt.zero_grad()
val_v = crt_net(states_v)
v_loss_v = F.mse_loss(val_v.squeeze(), ref_vals_v.detach())
v_loss_v.backward()
crt_opt.step()
```

And finally, we optimize the actor network:

```
act_opt.zero_grad()
acts_v = act_net(states_v)
q_out_v, _ = twinq_net(states_v, acts_v)
act_loss = -q_out_v.mean()
act_loss.backward()
act_opt.step()
```

In comparison with the formulas given previously, the code is missing the entropy regularization term and corresponds to DDPG training. As our variance doesn't depend on the state, it can be omitted from the optimization objective.

Results

I ran SAC training in the HalfCheetah and Ant environments for 9-13 hours, with 5M observations. The results are a bit contradictory. On the one hand, the sample efficiency and reward growing dynamics of SAC were better than the PPO method. For example, SAC was able to reach a reward of 900 after just 0.5M observations on HalfCheetah. PPO required more than 1M observations to reach the same policy. In the MuJoCo environment, SAC was able to find the policy that got a reward of 7,063, which is an absolute record (demonstrating state-of-the-art performance on this environment).

On the other hand, due to the off-policy nature of SAC, the training speed was much slower, as we did more calculations than with on-policy methods. On my machine, 5M frames on HalfCheetah took 10 hours. As a reminder, A2C did 50M observations in the same time.

This demonstrates the trade-offs between on-policy and off-policy methods, as you have seen many times in this book so far: if your environment is fast and observations are cheap to obtain, an on-policy method like PPO might be the best choice. But if your observations are hard to obtain, off-policy methods will do a better job, but require more calculations to be performed.

Figure 16.17 and Figure 16.18 show the reward dynamics on HalfCheetah:

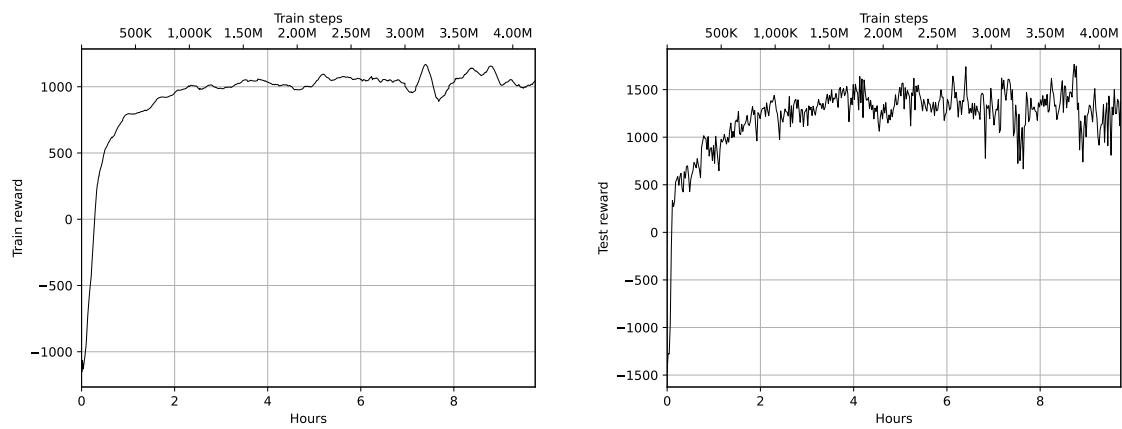


Figure 16.17: The reward during training (left) and the test reward (right) for HalfCheetah on PyBullet

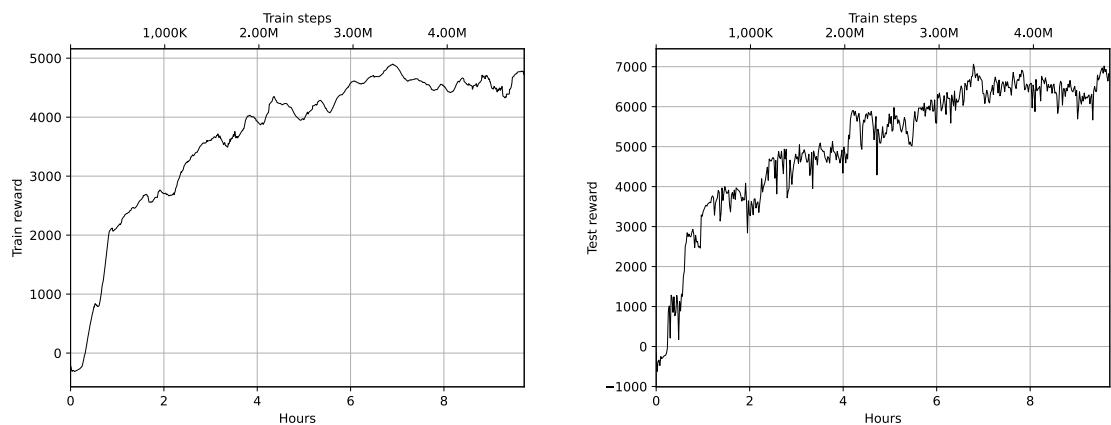


Figure 16.18: The reward during training (left) and the test reward (right) for HalfCheetah on MuJoCo

The results in the Ant environment are much worse – according to the score, the learned policy can barely stand.

The PyBullet plots are shown in *Figure 16.19*; MuJoCo plots are shown in *Figure 16.20*:

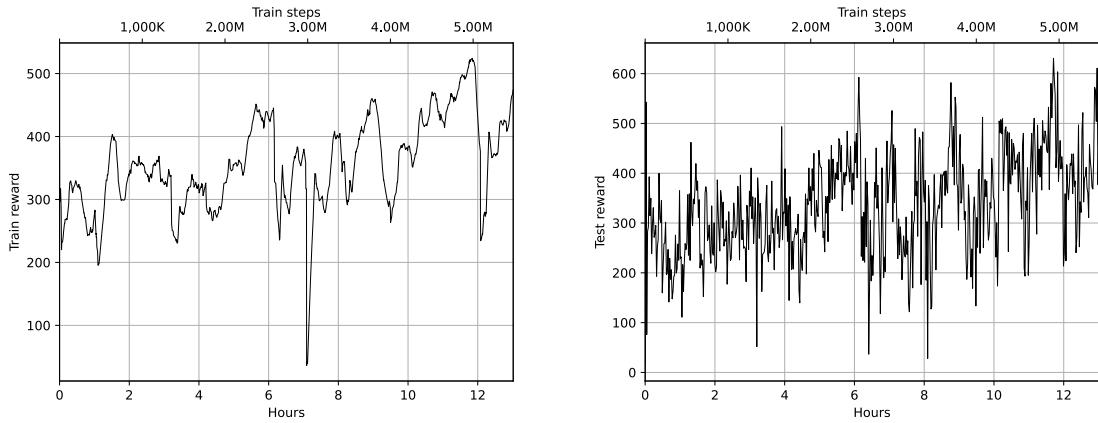


Figure 16.19: The reward during training (left) and the test reward (right) for Ant on PyBullet

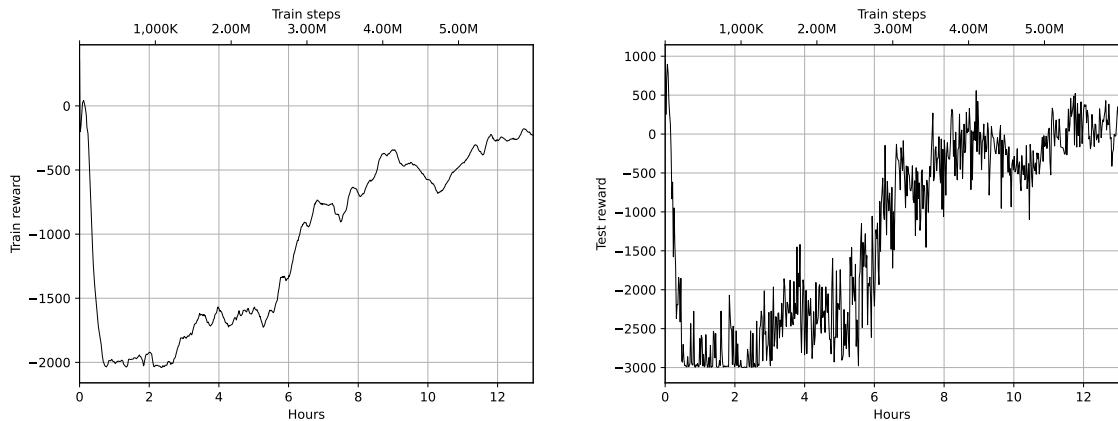


Figure 16.20: The reward during training (left) and the test reward (right) for Ant on MuJoCo

Here are the videos for the best SAC models:

- HalfCheetah on PyBullet (score 1,765): <https://youtu.be/80afu90zQ5s>. Our creature is a bit clumsy here.
- HalfCheetah on MuJoCo (score 7,063): <https://youtube.com/shorts/0Ywn3LTJxxs>. This result is really impressive – a super-fast Cheetah.
- Ant on PyBullet (score 630): <https://youtu.be/WHqXJ3VqX4k>. After a couple of steps, the ant got stuck for some reason.

Overall results

To simplify the comparison of the methods, I put all the numbers related to the best rewards obtained in the following table:

Method	HalfCheetah		Ant	
	PyBullet	MuJoCo	PyBullet	MuJoCo
A2C	2,189	4,718	2,425	5,380
PPO	2,567	1,623	2,560	5,108
TRPO	2,419	5,753	834	993
ACKTR	250	3,100	1,820	—
SAC	1,765	7,063	630	—

Table 16.1: Summary table

As you can see, there is no single winning method – some do well in some environments but get worse results in others. In principle, we can call A2C and PPO as quite consistent methods because they’re getting good results everywhere (PPO’s “backflip cheetah” on MuJoCo could be attributed to a bad starting seed, so rerunning the training might lead to a better policy).

Summary

In this chapter, we checked three different methods with the aim of improving the stability of the stochastic policy gradient and compared them to the A2C implementation on two continuous control problems. Along with the methods covered in the previous chapter (DDPG and D4PG), these methods are basic tools to work with a continuous control domain. Finally, we checked a relatively new off-policy method that is an extension of DDPG: SAC. Here, we have just scratched the surface of this topic, but it could be a good starting point to dive into it in more depth. These methods are widely used in robotics and related areas.

In the next chapter, we will switch to a different set of RL methods that have been becoming popular recently: *black-box* or *gradient-free* methods.

Join our community on Discord

Read this book alongside other users, Deep Learning experts, and the author himself.

Ask questions, provide solutions to other readers, chat with the author via Ask Me Anything sessions, and much more.

Scan the QR code or visit the link to join the community.

<https://packt.link/r1>



17

Black-Box Optimizations in RL

In this chapter, we will change our perspective on **reinforcement learning (RL)** training again and switch to the so-called **black-box optimizations**. These methods are at least a decade old, but recently, several research studies were conducted that showed their applicability to large-scale RL problems and their competitiveness with the value iteration and policy gradient methods. Despite their age, this family of methods is still more efficient in some situations. In particular, this chapter will cover two examples of black-box optimization methods:

- Evolution strategies
- Genetic algorithms

Black-box methods

To begin with, let's discuss the whole family of black-box methods and how it differs from what we've covered so far. Black-box optimization methods are the general approach to the optimization problem, when you treat the objective that you're optimizing as a black box, without any assumptions about the differentiability, the value function, the smoothness of the objective, and so on. The only requirement that those methods expose is the ability to calculate the **fitness function**, which should give us the measure of suitability of a particular instance of the optimized entity at hand.

One of the simplest examples in this family is *random search*, which is when you randomly sample the thing you're looking for (in the case of RL, it's the policy, $\pi(a|s)$), check the fitness of this candidate, and if the result is good enough (according to some reward criteria), then you're done. Otherwise, you repeat the process again and again. Despite the simplicity and even naivety of this approach, especially when compared to the sophisticated methods that you've seen so far, this is a good example to illustrate the idea of black-box methods.

Furthermore, with some modifications, as you will see shortly, this simple approach can be compared in terms of efficiency and the quality of the resulting policies to the **deep Q-network (DQN)** and policy gradient methods. In addition to that, black-box methods have several very appealing properties:

- They are at least two times faster than gradient-based methods, as we don't need to perform the backpropagation step to obtain the gradients.
- There are very few assumptions about the optimized objective and the policy that are treated as a black box. Traditional methods struggle with situations when the reward function is non-smooth or the policy contains steps with random choice. All of this is not an issue for black-box methods, as they don't expect much from the black-box internals.
- The methods can generally be parallelized very well. For example, the aforementioned random search can easily scale up to thousands of **central processing units (CPUs)** or **graphics processing units (GPUs)** working in parallel, without any dependency on each other. This is not the case for DQN or policy gradient methods, when you need to accumulate the gradients and propagate the current policy to all parallel workers, which decreases the parallelism.

The downside of the preceding is usually lower sample efficiency. In particular, the naïve random search of the policy, parameterized with the **neural network (NN)** with half a million parameters, has a very low probability of succeeding.

Evolution strategies

One subset of black-box optimization methods is called **evolution strategies (ES)**, and it was inspired by the evolution process. With ES, the most successful individuals have the highest influence on the overall direction of the search. There are many different methods that fall into this class, and in this chapter, we will consider the approach taken by the OpenAI researchers Salimans et al. in their paper, *Evolution strategies as a scalable alternative to reinforcement learning* [Sal+17], published in March 2017.

The underlying idea of ES methods is that on every iteration, we perform random perturbation of our current policy parameters and evaluate the resulting policy fitness function.

Then, we adjust the policy weights proportionally to the relative fitness function value.

The concrete method used by Salimans et al. is called **covariance matrix adaptation evolution strategy (CMA-ES)**, in which the perturbation performed is the random noise sampled from the normal distribution with the zero mean and identity variance. Then, we calculate the fitness function of the policy with weights equal to the weights of the original policy plus the scaled noise. Next, according to the obtained value, we adjust the original policy weights by adding the noise multiplied by the fitness function value, which moves our policy toward weights with a higher value of the fitness function. To improve the stability, the update of the weights is performed by averaging the batch of such steps with different random noise.

More formally, this method could be expressed as the following sequence of steps:

1. Initialize the learning rate, α , the noise standard deviation, σ , and the initial policy parameters, θ_0 .
2. For $t = 0, 1, \dots$ perform:
 - (a) The sample batch of noise with the shape of the weights from the normal distribution with zero mean and variance of one: $\epsilon_1, \dots, \epsilon_n \sim \mathcal{N}(0, I)$
 - (b) Compute returns $F_i = F(\theta_t + \sigma\epsilon_i)$ for $i = 1, \dots, n$
 - (c) Update weights:

$$\theta_{t+1} \leftarrow \theta_t + \alpha \frac{1}{n\sigma} \sum_{i=1}^n F_i \epsilon_i$$

This algorithm is the core of the method presented in the paper, but, as usual in the RL domain, the method alone is not enough to obtain good results. So, the paper includes several tweaks to improve the method, although the core is the same.

Implementing ES on CartPole

Let's implement and test the method from the paper on our fruit fly environment: CartPole. You'll find the complete example in `Chapter17/01_cartpole_es.py`.

In this example, we will use the single environment to check the fitness of the perturbed network weights. Our fitness function will be the undiscounted total reward for the episode.

We start with the imports:

```
import gymnasium as gym
import time
import numpy as np
import typing as tt

import torch
```

```
import torch.nn as nn

from torch.utils.tensorboard.writer import SummaryWriter
```

From the `import` statements, you will notice how self-contained our example is. We're not using PyTorch optimizers, as we don't perform backpropagation at all. In fact, we could avoid using PyTorch completely and work only with NumPy, as the only thing we use PyTorch for is to perform a forward pass and calculate the network's output.

Next, we define the hyperparameters:

```
MAX_BATCH_EPISODES = 100
MAX_BATCH_STEPS = 10000
NOISE_STD = 0.001
LEARNING_RATE = 0.001

TNoise = tt.List[torch.Tensor]
```

The number of hyperparameters is also small and includes the following values:

- `MAX_BATCH_EPISODES` and `MAX_BATCH_STEPS`: The limit of episodes and steps we use for training
- `NOISE_STD`: The standard deviation, σ , of the noise used for weight perturbation
- `LEARNING_RATE`: The coefficient used to adjust the weights on the training step

We also define a type alias for list of tensors containing weights' noises. It will simplify the code, as we'll deal with noise a lot.

Now let's check the network:

```
class Net(nn.Module):
    def __init__(self, obs_size: int, action_size: int):
        super(Net, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(obs_size, 32),
            nn.ReLU(),
            nn.Linear(32, action_size),
            nn.Softmax(dim=1)
        )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.net(x)
```

The model we're using is a simple one-hidden-layer NN, which gives us the action to take from the observation.

We're using PyTorch NN machinery here only for convenience, as we need only the forward pass, but it could be replaced by the multiplication of matrices and nonlinearities application.

The `evaluate()` function plays a full episode using the given policy and returns the total reward and the number of steps:

```
def evaluate(env: gym.Env, net: Net) -> tt.Tuple[float, int]:
    obs, _ = env.reset()
    reward = 0.0
    steps = 0
    while True:
        obs_v = torch.FloatTensor(np.expand_dims(obs, 0))
        act_prob = net(obs_v)
        acts = act_prob.max(dim=1)[1]
        obs, r, done, is_tr, _ = env.step(acts.data.numpy()[0])
        reward += r
        steps += 1
        if done or is_tr:
            break
    return reward, steps
```

The reward will be used as a fitness value, while the count of steps is needed to limit the amount of time we spend on forming the batch. The action selection is performed deterministically by calculating argmax from the network output. In principle, we could do the random sampling from the distribution, but we've already performed the exploration by adding noise to the network parameters, so the deterministic action selection is fine here.

In the `sample_noise()` function, we create random noise with zero mean and unit variance equal to the shape of our network parameters:

```
def sample_noise(net: Net) -> tt.Tuple[TNoise, TNoise]:
    pos = []
    neg = []
    for p in net.parameters():
        noise = np.random.normal(size=p.data.size())
        noise_t = torch.FloatTensor(noise)
        pos.append(noise_t)
        neg.append(-noise_t)
    return pos, neg
```

The function returns two sets of noise tensors: one with positive noise and another with the same random values taken with a negative sign. These two samples are later used in a batch as independent samples. This technique is known as *mirrored sampling* and is used to improve the stability of the convergence.

In fact, without the negative noise, the convergence becomes very unstable because positive noise pushes weights in a single direction.

The `eval_with_noise()` function takes the noise array created by `sample_noise()` and evaluates the network with noise added:

```
def eval_with_noise(env: gym.Env, net: nn.Module, noise: TNoise, noise_std: float,
    get_max_action: bool = True, device: torch.device = torch.device("cpu"))
old_params = net.state_dict()
for p, p_n in zip(net.parameters(), noise):
    p.data += NOISE_STD * p_n
r, s = evaluate(env, net)
net.load_state_dict(old_params)
return r, s
```

To achieve this, we add the noise to the network's parameters and call the `evaluate` function to obtain the reward and number of steps taken. After this, we need to restore the network weights to their original state, which is completed by loading the state dictionary of the network.

The last and the central function of the method is `train_step()`, which takes the batch with noise and respective rewards and calculates the update to the network parameters by applying the formula:

$$\theta_{t+1} \leftarrow \theta_t + \alpha \frac{1}{n\sigma} \sum_{i=1}^n F_i \epsilon_i$$

This can be implemented as follows:

```
def train_step(net: Net, batch_noise: tt.List[common.TNoise], batch_reward:
tt.List[float],
writer: SummaryWriter, step_idx: int):
weighted_noise = None
norm_reward = np.array(batch_reward)
norm_reward -= np.mean(norm_reward)
s = np.std(norm_reward)
if abs(s) > 1e-6:
    norm_reward /= s
```

In the beginning, we normalize rewards to have zero mean and unit variance, which improves the stability of the method.

Then, we iterate every pair (noise, reward) in our batch and multiply the noise values with the normalized reward, summing together the respective noise for every parameter in our policy:

```

for noise, reward in zip(batch_noise, norm_reward):
    if weighted_noise is None:
        weighted_noise = [reward * p_n for p_n in noise]
    else:
        for w_n, p_n in zip(weighted_noise, noise):
            w_n += reward * p_n

```

As a final step, we use the accumulated scaled noise to adjust the network parameters:

```

m_updates = []
for p, p_update in zip(net.parameters(), weighted_noise):
    update = p_update / (len(batch_reward) * NOISE_STD)
    p.data += LEARNING_RATE * update
    m_updates.append(torch.norm(update))
writer.add_scalar("update_l2", np.mean(m_updates), step_idx)

```

Technically, what we do here is a gradient ascent, although the gradient was not obtained from backpropagation but from the random sampling (also known as Monte Carlo sampling). This fact was also demonstrated by Salimans et al., where the authors showed that CMA-ES is very similar to the policy gradient methods, differing in just the way that we get the gradients' estimation.

The preparation before the training loop is simple; we create the environment and the network:

```

if __name__ == "__main__":
    writer = SummaryWriter(comment="-cartpole-es")
    env = gym.make("CartPole-v1")

    net = Net(env.observation_space.shape[0], env.action_space.n)
    print(net)

```

Every iteration of the training loop starts with batch creation, where we sample the noise and obtain rewards for both positive and negated noise:

```

step_idx = 0
while True:
    t_start = time.time()
    batch_noise = []
    batch_reward = []
    batch_steps = 0
    for _ in range(MAX_BATCH_EPISODES):
        noise, neg_noise = sample_noise(net)

```

```

batch_noise.append(noise)
batch_noise.append(neg_noise)
reward, steps = eval_with_noise(env, net, noise)
batch_reward.append(reward)
batch_steps += steps
reward, steps = eval_with_noise(env, net, neg_noise)
batch_reward.append(reward)
batch_steps += steps
if batch_steps > MAX_BATCH_STEPS:
    break

```

When we reach the limit of episodes in the batch, or the limit of the total steps, we stop gathering the data and do a training update.

To perform the update of the network, we call the `train_step()` function that we've already seen:

```

step_idx += 1
m_reward = float(np.mean(batch_reward))
if m_reward > 199:
    print("Solved in %d steps" % step_idx)
    break

train_step(net, batch_noise, batch_reward, writer, step_idx)

```

The goal of the `train_step()` function is to scale the noise according to the total reward and then adjust the policy weights in the direction of the averaged noise.

The final steps in the training loop write metrics into TensorBoard and show the training progress on the console:

```

writer.add_scalar("reward_mean", m_reward, step_idx)
writer.add_scalar("reward_std", np.std(batch_reward), step_idx)
writer.add_scalar("reward_max", np.max(batch_reward), step_idx)
writer.add_scalar("batch_episodes", len(batch_reward), step_idx)
writer.add_scalar("batch_steps", batch_steps, step_idx)
speed = batch_steps / (time.time() - t_start)
writer.add_scalar("speed", speed, step_idx)
print("%d: reward=%.2f, speed=%.2f f/s" % (
    step_idx, m_reward, speed))

```

CartPole results

Training can be started by just running the program without the arguments:

```
Chapter17$ ./01_cartpole_es.py
Net(
    (net): Sequential(
        (0): Linear(in_features=4, out_features=32, bias=True)
        (1): ReLU()
        (2): Linear(in_features=32, out_features=2, bias=True)
        (3): Softmax(dim=1)
    )
)
1: reward=10.00, speed=7458.03 f/s
2: reward=11.93, speed=8454.54 f/s
3: reward=13.71, speed=8677.55 f/s
4: reward=15.96, speed=8905.25 f/s
5: reward=18.75, speed=9098.71 f/s
6: reward=22.08, speed=9220.68 f/s
7: reward=23.57, speed=9272.45 f/s
...
...
```

From my experiments, it usually takes ES about 40–60 batches to solve CartPole. The convergence dynamics for the preceding run are shown in *Figure 17.1* and *Figure 17.2*.

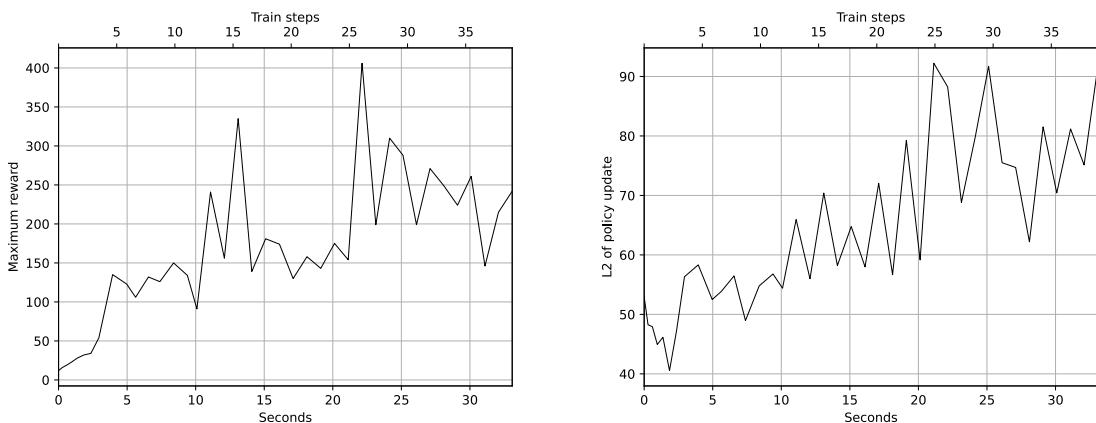


Figure 17.1: The maximum reward (left) and policy update (right) for ES on CartPole

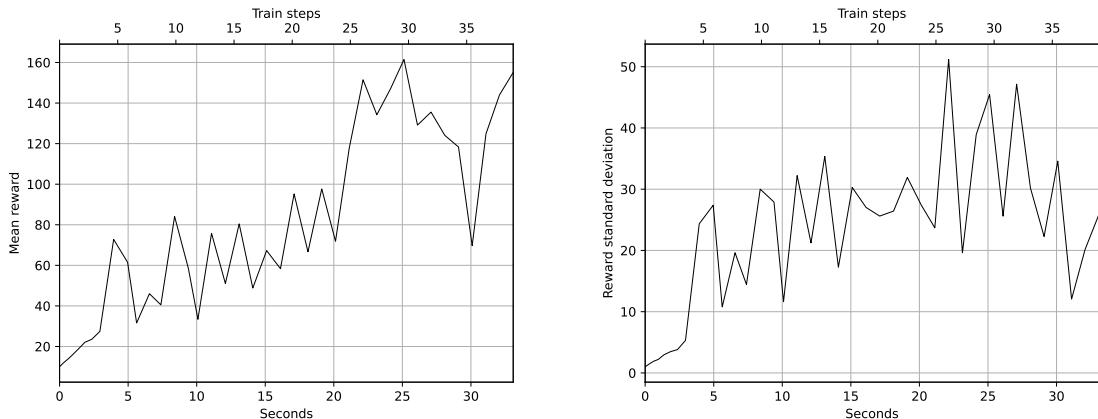


Figure 17.2: The mean (left) and standard deviation (right) of reward for ES on CartPole

The preceding graphs look quite good—being able to solve the environment in 30 seconds is on par with the cross-entropy method from *Chapter 4*.

ES on HalfCheetah

In the next example, we will go beyond the simplest ES implementation and look at how this method can be parallelized efficiently using the shared seed strategy proposed by Salimans et al. To show this approach, we will use the HalfCheetah environment using the MuJoCo physics simulator. We already experimented with it in the previous chapter, so if you haven't installed the `gymnasium[mujoco]` package, you should do so.

First, let's discuss the idea of shared seeds. The performance of the ES algorithm is mostly determined by the speed at which we can gather our training batch, which consists of sampling the noise and checking the total reward of the perturbed noise. As our training batch items are independent, we can easily parallelize this step to a large number of workers sitting on remote machines. (That's a bit similar to the example from *Chapter 12*, when we gathered gradients from A3C workers.) However, naïve implementation of this parallelization will require a large amount of data to be transferred from the worker process to the central master, which is supposed to combine the noise checked by the workers and perform the policy update. Most of this data is the noise vectors, the size of which is equal to the size of our policy parameters.

To avoid this overhead, a quite elegant solution was proposed by Salimans et al. As noise sampled on a worker is produced by a pseudo-random number generator, which allows us to set the random seed and reproduce the random sequence generated, the worker can transfer to the master only the seed that was used to generate the noise. Then, the master can generate the same noise vector again using the seed. Of course, the seed on every worker needs to be generated randomly to still have a random optimization process.

This has the effect of dramatically decreasing the amount of data that needs to be transferred from workers to the master, improving the scalability of the method. For example, Salimans et al. reported linear speed up in optimizations involving 1,440 CPUs in the cloud. In our example, we will look at local parallelization using the same approach.

Implementing ES on HalfCheetah

The code is placed in `Chapter17/02_cheetah_es.py`. As the code significantly overlaps with the CartPole version, we will focus here only on the differences.

We will begin with the worker, which is started as a separate process using the PyTorch multiprocessing wrapper. The worker's responsibilities are simple: for every iteration, it obtains the network parameters from the master process, and then it performs the fixed number of iterations, where it samples the noise and evaluates the reward. The result with the random seed is sent to the master using the queue.

The following dataclass is used by the worker to send the results of the perturbed policy evaluation to the master process:

```
@dataclass(frozen=True)
class RewardsItem:
    seed: int
    pos_reward: float
    neg_reward: float
    steps: int
```

It includes the random seed, the rewards obtained with the positive and negative noise, and the total number of steps we performed in both tests.

On every training iteration, the worker waits for the network parameters to be broadcasted from the master:

```
def worker_func(params_queue: mp.Queue, rewards_queue: mp.Queue,
                device: torch.device, noise_std: float):
    env = make_env()
    net = Net(env.observation_space.shape[0], env.action_space.shape[0]).to(device)
    net.eval()

    while True:
        params = params_queue.get()
        if params is None:
            break
        net.load_state_dict(params)
```

The value of `None` means that the master wants to stop the worker.

The rest is almost the same as the previous example, with the main difference being in the random seed generated and assigned before the noise generation. This allows the master to regenerate the same noise, only from the seed:

```
for _ in range(ITERS_PER_UPDATE):
    seed = np.random.randint(low=0, high=65535)
    np.random.seed(seed)
    noise, neg_noise = common.sample_noise(net, device=device)
    pos_reward, pos_steps = common.eval_with_noise(env, net, noise, noise_std,
        get_max_action=False, device=device)
    neg_reward, neg_steps = common.eval_with_noise(env, net, neg_noise,
        noise_std,
        get_max_action=False, device=device)
    rewards_queue.put(RewardsItem(seed=seed, pos_reward=pos_reward,
        neg_reward=neg_reward, steps=pos_steps+neg_steps))
```

Another difference lies in the function used by the master to perform the training step:

```
def train_step(optimizer: optim.Optimizer, net: Net, batch_noise: tt.List[common.TNoise],
               batch_reward: tt.List[float], writer: SummaryWriter, step_idx: int,
               noise_std: float):
    weighted_noise = None
    norm_reward = compute_centered_ranks(np.array(batch_reward))
```

In the CartPole example, we normalized the batch of rewards by subtracting the mean and dividing by the standard deviation. According to Salimans et al., better results could be obtained using ranks instead of actual rewards. As ES has no assumptions about the fitness function (which is a reward in our case), we can make any rearrangements in the reward that we want, which wasn't possible in the case of DQN, for example.

Here, **rank transformation** of the array means replacing the array with indices of the sorted array. For example, array [0.1, 10, 0.5] will have the rank array [0, 2, 1]. The `compute_centered_ranks` function takes the array with the total rewards of the batch, calculates the rank for every item in the array, and then normalizes those ranks. For example, an input array of [21.0, 5.8, 7.0] will have ranks [2, 0, 1], and the final centered ranks will be [0.5, -0.5, 0.0].

Another major difference in the training function is the use of PyTorch optimizers:

```
for noise, reward in zip(batch_noise, norm_reward):
    if weighted_noise is None:
        weighted_noise = [reward * p_n for p_n in noise]
    else:
```

```
        for w_n, p_n in zip(weighted_noise, noise):
            w_n += reward * p_n
    m_updates = []
    optimizer.zero_grad()
    for p, p_update in zip(net.parameters(), weighted_noise):
        update = p_update / (len(batch_reward) * noise_std)
        p.grad = -update
        m_updates.append(torch.norm(update))
    writer.add_scalar("update_l2", np.mean(m_updates), step_idx)
    optimizer.step()
```

To understand why they are used and how this is possible without doing backpropagation, some explanations are required.

First, Salimans et al. showed that the optimization method used by the ES algorithm is very similar to gradient ascent on the fitness function, with the difference being how the gradient is calculated. The way the **stochastic gradient descent (SGD)** method is usually applied is that the gradient is obtained from the loss function by calculating the derivative of the network parameters with respect to the loss value. This imposes the limitation on the network and loss function to be differentiable, which is not always the case; for example, the rank transformation performed by the ES method is not differentiable.

On the other hand, optimization performed by ES works differently. We randomly sample the neighborhood of our current parameters by adding the noise to them and calculating the fitness function. According to the fitness function change, we adjust the parameters, which pushes our parameters in the direction of a higher fitness function. The result of this is very similar to gradient-based methods, but the requirements imposed on our fitness function are much looser: the only requirement is our ability to calculate it.

However, if we're estimating some kind of gradient by randomly sampling the fitness function, we can use standard optimizers from PyTorch. Normally, optimizers adjust the parameters of the network using gradients accumulated in the parameters' `grad` fields.

Those gradients are accumulated after the backpropagation step, but due to PyTorch's flexibility, the optimizer doesn't care about the source of the gradients. So, the only thing we need to do is copy the estimated parameters' update in the `grad` fields and ask the optimizer to update them. Note that the update is copied with a negative sign, as optimizers normally perform gradient descent (as in a normal operation, we *minimize* the loss function), but in this case, we want to do gradient ascent. This is very similar to the actor-critic method we covered in *Chapter 12*, when the estimated policy gradient is taken with the negative sign, as it shows the direction to *improve* the policy.

The last chunk of differences in the code is taken from the training loop performed by the master process. Its responsibility is to wait for data from worker processes, perform the training update of the parameters, and broadcast the result to the workers. The communication between the master and workers is performed by two sets of queues. The first queue is per-worker and is used by the master to send the current policy parameters to use. The second queue is shared by the workers and is used to send the already mentioned RewardItem structure with the random seed and rewards:

```
params_queues = [mp.Queue(maxsize=1) for _ in range(PROCESSES_COUNT)]
rewards_queue = mp.Queue(maxsize=ITERS_PER_UPDATE)
workers = []

for params_queue in params_queues:
    p_args = (params_queue, rewards_queue, device, args.noise_std)
    proc = mp.Process(target=worker_func, args=p_args)
    proc.start()
    workers.append(proc)

print("All started!")
optimizer = optim.Adam(net.parameters(), lr=args.lr)
```

At the beginning of the master, we create all those queues, start the worker processes, and create the optimizer. Every training iteration starts with the network parameters being broadcast to the workers:

```
for step_idx in range(args.iters):
    params = net.state_dict()
    for q in params_queues:
        q.put(params)
```

Then, in the loop, the master waits for enough data to be obtained from the workers:

```
t_start = time.time()
batch_noise = []
batch_reward = []
results = 0
batch_steps = 0
while True:
    while not rewards_queue.empty():
        reward = rewards_queue.get_nowait()
        np.random.seed(reward.seed)
        noise, neg_noise = common.sample_noise(net)
        batch_noise.append(noise)
        batch_reward.append(reward.pos_reward)
    if len(batch_reward) == batch_steps:
        # Process results
        results += sum(batch_reward)
        batch_reward = []
        batch_steps = 0
    else:
        batch_steps += 1
    if time.time() - t_start > args.duration:
        break
```

```
        batch_noise.append(neg_noise)
        batch_reward.append(reward.neg_reward)
        results += 1
        batch_steps += reward.steps

    if results == PROCESSES_COUNT * ITERS_PER_UPDATE:
        break
    time.sleep(0.01)
```

Every time a new result arrives, we reproduce the noise using the random seed.

As the last step in the training loop, we call the `train_step()` function:

```
train_step(optimizer, net, batch_noise, batch_reward,
           writer, step_idx, args.noise_std)
```

You've already seen this function, which calculates the update from the noise and rewards, and calls the optimizer to adjust the weights.

HalfCheetah results

The code supports the optional `-dev` flag, but from my experiments, I got a *slowdown* if GPU was enabled: without GPU, the average speed was 20-21k observations per second, but with CUDA, it was just 9k. This might look counter-intuitive, but we can explain this with the very small network and batch size of a single observation. Potentially, we might decrease the gap (or even get some speedup) with a higher batch size, but it will complicate our code.

During the training, we show the mean reward, the speed of training (in observations per second), and two timing values (showing how long it took to gather data and perform the training step):

```
$ ./02_cheetah_es.py
Net(
(mu): Sequential(
  (0): Linear(in_features=17, out_features=64, bias=True)
  (1): Tanh()
  (2): Linear(in_features=64, out_features=6, bias=True)
  (3): Tanh()
)
)
All started!
0: reward=-505.09, speed=17621.60 f/s, data_gather=6.792, train=0.018
1: reward=-440.50, speed=20609.56 f/s, data_gather=5.815, train=0.007
```

```

2: reward=-383.76, speed=20568.74 f/s, data_gather=5.827, train=0.007
3: reward=-326.02, speed=20413.63 f/s, data_gather=5.871, train=0.007
4: reward=-259.58, speed=20181.74 f/s, data_gather=5.939, train=0.007
5: reward=-198.80, speed=20496.81 f/s, data_gather=5.848, train=0.007
6: reward=-113.22, speed=20467.71 f/s, data_gather=5.856, train=0.007

```

The dynamics of the training show very quick policy improvement in the beginning: in just 100 updates, which is 9 minutes of training, the agent was able to reach the score of 1,500-1,600. After 30 minutes, the peak reward was 2,833; but with more training, the policy was degrading.

The maximum, mean, and standard deviation of reward are shown in *Figure 17.3* and *Figure 17.4*.

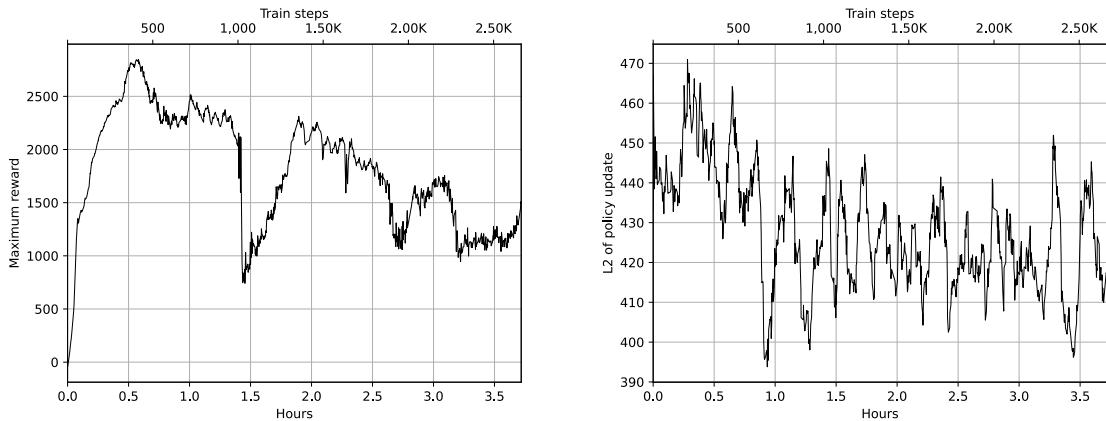


Figure 17.3: The maximum reward (left) and policy update (right) for ES on HalfCheetah

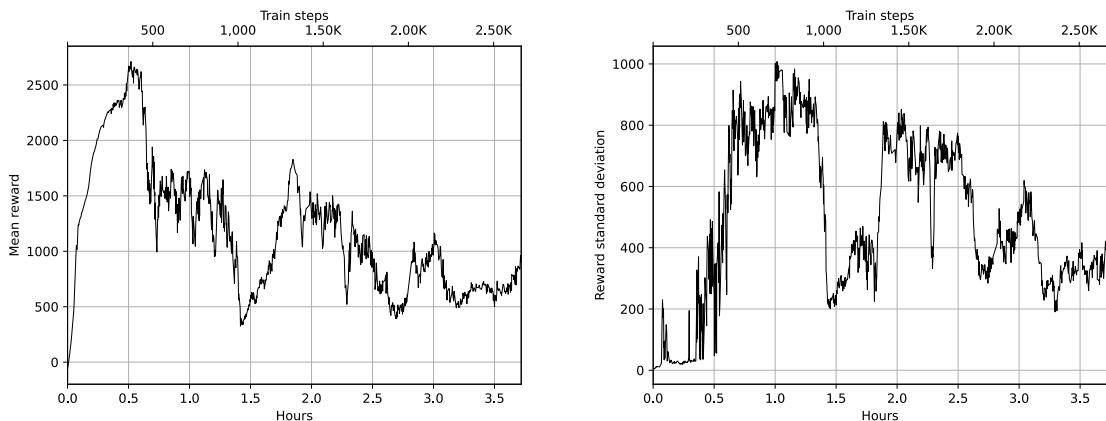


Figure 17.4: The mean (left) and standard deviation (right) of reward for ES on HalfCheetah

Genetic algorithms

Another popular class of black-box methods is **genetic algorithms (GAs)**. It is a large family of optimization methods with more than two decades of history behind it and a simple core idea of generating a population of N individuals (concrete model parameters), each of which is evaluated with the fitness function. Then, some subset of top performers is used to produce the next generation of the population (this process is called *mutation*). This process is repeated until we're satisfied with the performance of our population.

There are a lot of different methods in the GA family, for example, how to perform the mutation of the individuals for the next generation or how to rank the performers. Here, we will consider the simple GA method with some extensions, published in the paper by Such et al., called *Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning* [Suc+17].

In this paper, the authors analyzed the simple GA method, which performs Gaussian noise perturbation of the parent's weights to perform mutation. On every iteration, the top performer was copied without modification. In an algorithm form, the steps of a simple GA method can be written as follows:

1. Initialize the mutation power, σ , the population size, N , the number of selected individuals, T , and the initial population, P^0 , with N randomly initialized policies and their fitness: $F^0 = \{F(P_i^0) | i = 1 \dots N\}$
2. For generation $g = 1 \dots G$:
 - (a) Sort generation P^{g-1} in the descending order of the fitness function value F^{g-1}
 - (b) Copy elite $P_1^g = P_1^{g-1}, F_1^g = F_1^{g-1}$
 - (c) For individual $i = 2 \dots N$:
 - i. Choose the k : random parent from $1 \dots T$
 - ii. Sample $\epsilon \sim \mathcal{N}(0, I)$
 - iii. Mutate the parent: $P_i^g = P_i^{g-1} + \sigma\epsilon$
 - iv. Get its fitness: $F_i^g = F(P_i^g)$

There have been several improvements to this basic method from the paper [2], which we will discuss later. For now, let's check the implementation of the core algorithm.

GA on CartPole

The source code is in Chapter17/03_cartpole_ga.py, and it has a lot in common with our ES example. The difference is in the lack of the gradient ascent code, which was replaced by the network mutation function:

```
def mutate_parent(net: Net) -> Net:
    new_net = copy.deepcopy(net)
    for p in new_net.parameters():
        noise = np.random.normal(size=p.data.size())
        noise_t = torch.FloatTensor(noise)
        p.data += NOISE_STD * noise_t
    return new_net
```

The goal of the function is to create a mutated copy of the given policy by adding a random noise to all weights. The parent's weights are kept untouched, as a random selection of the parent is performed with replacement, so this network could be used again later.

The count of hyperparameters is even smaller than with ES and includes the standard deviation of the noise added-on mutation, the population size, and the number of top performers used to produce the subsequent generation:

```
NOISE_STD = 0.01
POPULATION_SIZE = 50
PARENTS_COUNT = 10
```

Before the training loop, we create the population of randomly initialized networks and obtain their fitness:

```
if __name__ == "__main__":
    env = gym.make("CartPole-v1")
    writer = SummaryWriter(comment="-cartpole-ga")

    gen_idx = 0
    nets = [
        Net(env.observation_space.shape[0], env.action_space.n)
        for _ in range(POPULATION_SIZE)
    ]
    population = [
        (net, common.evaluate(env, net))
        for net in nets
    ]
```

At the beginning of every generation, we sort the previous generation according to its fitness and record statistics about future parents:

```

while True:
    population.sort(key=lambda p: p[1], reverse=True)
    rewards = [p[1] for p in population[:PARENTS_COUNT]]
    reward_mean = np.mean(rewards)
    reward_max = np.max(rewards)
    reward_std = np.std(rewards)

    writer.add_scalar("reward_mean", reward_mean, gen_idx)
    writer.add_scalar("reward_std", reward_std, gen_idx)
    writer.add_scalar("reward_max", reward_max, gen_idx)
    print("%d: reward_mean=%f, reward_max=%f, reward_std=%f" % (
        gen_idx, reward_mean, reward_max, reward_std))
    if reward_mean > 199:
        print("Solved in %d steps" % gen_idx)
        break

```

In a separate loop over new individuals to be generated, we randomly sample a parent, mutate it, and evaluate its fitness score:

```

prev_population = population
population = [population[0]]
for _ in range(POPULATION_SIZE-1):
    parent_idx = np.random.randint(0, PARENTS_COUNT)
    parent = prev_population[parent_idx][0]
    net = mutate_parent(parent)
    fitness = common.evaluate(env, net)
    population.append((net, fitness))
    gen_idx += 1

```

After starting the implementation, you should see the following (concrete output and count of steps might vary due to randomness in execution):

```

Chapter17$ ./03_cartpole_ga.py
0: reward_mean=29.50, reward_max=109.00, reward_std=27.86
1: reward_mean=65.50, reward_max=111.00, reward_std=27.61
2: reward_mean=149.10, reward_max=305.00, reward_std=57.76
3: reward_mean=175.00, reward_max=305.00, reward_std=47.35
4: reward_mean=200.50, reward_max=305.00, reward_std=39.98
Solved in 4 steps

```

As you can see, the GA method is even more efficient than the ES method.

GA tweaks

Such et al. proposed two tweaks to the basic GA algorithm:

- The first, with the name **deep GA**, aimed to increase the scalability of the implementation. We will implement this later in the *GA on HalfCheetah* section.
- The second, called **novelty search**, was an attempt to replace the reward objective with a different metric of the episode. We've left this as an exercise for you to try out.

In the example used in the following *GA on HalfCheetah* section, we will implement the first improvement, whereas the second one is left as an optional exercise.

Deep GA

Being a gradient-free method, GA is potentially even more scalable than ES methods in terms of speed, with more CPUs involved in the optimization. However, the simple GA algorithm that you have seen has a similar bottleneck to ES methods: the policy parameters have to be exchanged between the workers. Such et al. (the authors) proposed a trick similar to the shared seed approach but taken to an extreme (as we're using seeds to track thousands of mutations). They called it deep GA, and at its core, the policy parameters are represented as a list of random seeds used to create this particular policy's weights.

In fact, the initial network's weights were generated randomly on the first population, so the first seed in the list defines this initialization. On every population, mutations are also fully specified by the random seed for every mutation. So, the only thing we need to reconstruct the weights is the seeds themselves. In this approach, we need to reconstruct the weights on every worker, but usually, this overhead is much less than the overhead of transferring full weights over the network.

Novelty search

Another modification to the basic GA method is **novelty search (NS)**, which was proposed by Lehman and Stanley in their paper, *Abandoning objectives: Evolution through the search for novelty alone*, which was published in 2011 [LS11].

The idea of NS is to change the objective in our optimization. We're no longer trying to increase our total reward from the environment but, rather, reward the agent for exploring the behavior that it has never checked before (that is, *novel*). According to the authors' experiments on the maze navigation problem, with many traps for the agent, NS works much better than other reward-driven approaches.

To implement NS, we define the so-called **behavior characteristic (BC)** (π), which describes the behavior of the policy and a distance between two BCs. Then, the k-nearest neighbors approach is used to check the novelty of the new policy and drive the GA according to this distance.

In the paper by Such et al., sufficient exploration by the agent was needed. The approach of NS significantly outperformed the ES, GA, and other more traditional approaches to RL problems.

GA on HalfCheetah

In our final example in this chapter, we will implement the parallelized deep GA on the HalfCheetah environment. The complete code is in `04_cheetah_ga.py`. The architecture is very close to the parallel ES version, with one master process and several workers. The goal of every worker is to evaluate the batch of networks and return the result to the master, which merges partial results into the complete population, ranks the individuals according to the obtained reward, and generates the next population to be evaluated by the workers.

Every individual is encoded by a list of random seeds used to initialize the initial network weights and all subsequent mutations. This representation allows very compact encoding of the network, even when the number of parameters in the policy is not very large. For example, in our network with one hidden layer of 64 neurons, we have 1,542 float values (the input is 17 values and the action is 6 floats, which gives $17 \times 64 + 64 + 64 \times 6 + 6 = 1542$). Every float occupies 4 bytes, which is the same size used by the random seed. So, the deep GA representation proposed by the paper will be smaller up to 1,542 generations in the optimization.

Implementation

In our example, we will perform parallelization on local CPUs so the amount of data transferred back and forth doesn't matter much; however, if you have a couple of hundred cores to utilize, the representation might become a significant issue.

The set of hyperparameters is the same as in the CartPole example, with the difference of a larger population size:

```
NOISE_STD = 0.01
POPULATION_SIZE = 2000
PARENTS_COUNT = 10
WORKERS_COUNT = 6
SEEDS_PER_WORKER = POPULATION_SIZE // WORKERS_COUNT
MAX_SEED = 2**32 - 1
```

There are two functions used to build the networks based on the seeds given. The first one performs one mutation on the already created policy network:

```
def mutate_net(net: Net, seed: int, copy_net: bool = True) -> Net:
    new_net = copy.deepcopy(net) if copy_net else net
    np.random.seed(seed)
    for p in new_net.parameters():
        noise = np.random.normal(size=p.data.size())
        noise_t = torch.FloatTensor(noise)
        p.data += NOISE_STD * noise_t
    return new_net
```

The preceding function can perform the mutation in place or by copying the target network based on arguments (copying is needed for the first generation).

The second function creates the network from scratch using the list of seeds:

```
def build_net(env: gym.Env, seeds: tt.List[int]) -> Net:
    torch.manual_seed(seeds[0])
    net = Net(env.observation_space.shape[0], env.action_space.shape[0])
    for seed in seeds[1:]:
        net = mutate_net(net, seed, copy_net=False)
    return net
```

Here, the first seed is passed to PyTorch to influence the network initialization, and subsequent seeds are used to apply network mutations.

The worker function obtains the list of seeds to evaluate and outputs individual `OutputItem` dataclass items for every result obtained:

```
@dataclass
class OutputItem:
    seeds: tt.List[int]
    reward: float
    steps: int

def worker_func(input_queue: mp.Queue, output_queue: mp.Queue):
    env = gym.make("HalfCheetah-v4")
    cache = {}

    while True:
        parents = input_queue.get()
        if parents is None:
            break
        new_cache = {}
```

```

for net_seeds in parents:
    if len(net_seeds) > 1:
        net = cache.get(net_seeds[:-1])
        if net is not None:
            net = mutate_net(net, net_seeds[-1])
        else:
            net = build_net(env, net_seeds)
    else:
        net = build_net(env, net_seeds)
    new_cache[net_seeds] = net
    reward, steps = common.evaluate(env, net, get_max_action=False)
    output_queue.put(OutputItem(seeds=net_seeds, reward=reward, steps=steps))
cache = new_cache

```

This function maintains the cache of networks to minimize the amount of time spent recreating the parameters from the list of seeds. This cache is cleared for every generation, as every new generation is created from the current generation winners, so there is only a tiny chance that old networks can be reused from the cache.

The code of the master process is also straightforward:

```

batch_steps = 0
population = []
while len(population) < SEEDS_PER_WORKER * WORKERS_COUNT:
    out_item = output_queue.get()
    population.append((out_item.seeds, out_item.reward))
    batch_steps += out_item.steps
if elite is not None:
    population.append(elite)
population.sort(key=lambda p: p[1], reverse=True)
elite = population[0]
for worker_queue in input_queues:
    seeds = []
    for _ in range(SEEDS_PER_WORKER):
        parent = np.random.randint(PARENTS_COUNT)
        next_seed = np.random.randint(MAX_SEED)
        s = list(population[parent][0]) + [next_seed]
        seeds.append(tuple(s))

```

For every generation, we send the current population's seeds to workers for evaluation and wait for the results. Then, we sort the results and generate the next population based on the top performers. On the master's side, the mutation is just a seed number generated randomly and appended to the list of seeds of the parent.

Results

In this example, we're using the MuJoCo HalfCheetah environment, which doesn't have any health checks internally, so every episode takes 2,000 steps. Because of this, every training step requires about a minute, so be patient. After 300 mutation rounds (which took about 7 hours), the best policy was able to get a reward of 6454, which is a great result. If you remember our experiments in the previous chapter, only the SAC method was able to get a higher reward of 7063 on MuJoCo HalfCheetah. Of course, HalfCheetah is not very challenging, but still – very good.

The plots are shown in *Figure 17.5* and *Figure 17.6*.

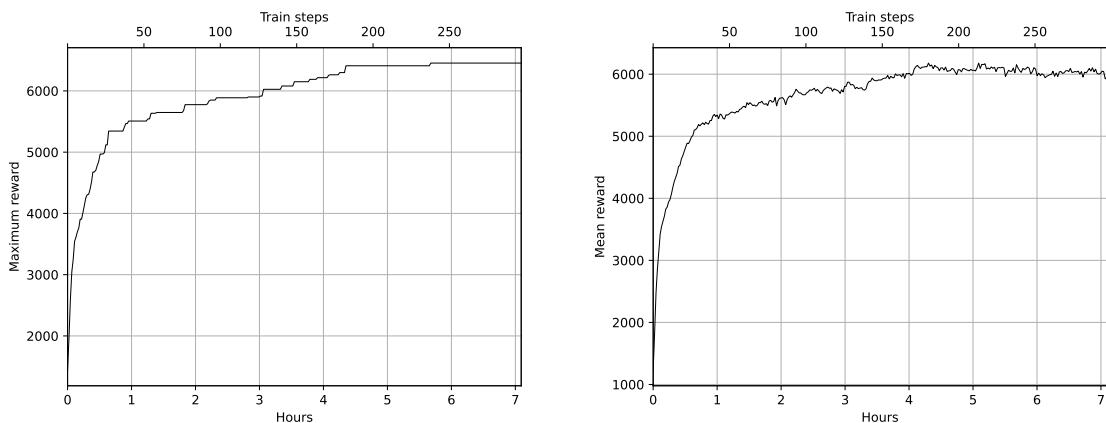


Figure 17.5: The maximum (left) and mean rewards (right) for GA on HalfCheetah

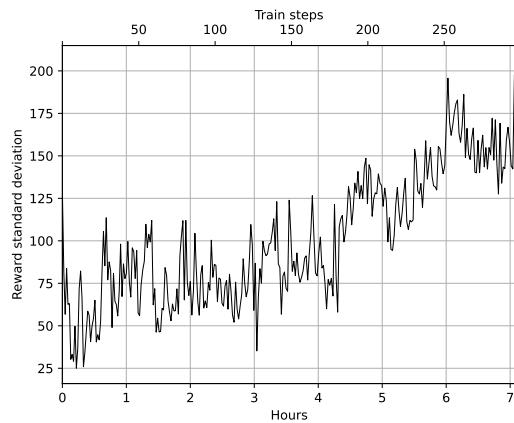


Figure 17.6: Standard deviation of reward for GA on HalfCheetah

Summary

In this chapter, you saw two examples of black-box optimization methods: evolution strategies and genetic algorithms, which can provide competition for other analytical gradient methods. Their strength lies in good parallelization on a large number of resources and the smaller number of assumptions that they have on the reward function.

In the next chapter, we will take a look at a very important aspect of RL: advanced exploration methods.

18

Advanced Exploration

In this chapter, we will talk about the topic of exploration in **reinforcement learning (RL)**. It has been mentioned several times in the book that the exploration/exploitation dilemma is a fundamental thing in RL and very important for efficient learning. However, in the previous examples, we used quite a trivial approach to exploring the environment, which was, in most cases, ϵ -greedy action selection. Now it's time to go deeper into the exploration subfield of RL, as more complicated environments might require much better exploration strategies than ϵ -greedy approach.

More specifically, we will cover the following key topics:

- Why exploration is such a fundamental topic in RL
- The effectiveness of the epsilon-greedy (ϵ -greedy) approach
- Alternatives and how they work in different environments

We will implement the methods described to solve a toy, but still challenging, problem called MountainCar. This will allow us to better understand the methods, the way they could be implemented, and their behavior. After that, we will try to tackle a harder problem from the Atari suite.

Why exploration is important

In this book, lots of environments and methods have been discussed, and in almost every chapter, exploration was mentioned. Very likely, you've already got ideas about why it's important to explore the environment effectively, so I'm just going to discuss the main reasons.

Before that, it might be useful to agree on the term "effective exploration." In theoretical RL, a strict definition of this exists, but the high-level idea is simple and intuitive. Exploration is effective when we don't waste time in states of the environment that have already been seen by and are familiar to the agent. Rather than taking the same actions again and again, the agent needs to look for a new experience. As we've already discussed, *exploration* has to be balanced by *exploitation*, which is the opposite and means using our knowledge to get the best reward in the most efficient way. Let's now quickly discuss why we might be interested in effective exploration in the first place.

First, good exploration of the environment might have a fundamental influence on our ability to learn a good policy. If the reward is sparse and the agent obtains a good reward on some rare conditions, it might experience a positive reward only once in many episodes, so the ability of the learning process to explore the environment effectively and fully might bring more samples with a good reward that the method could learn from.

In some cases, which are very frequent in practical applications of RL, a lack of good exploration might mean that the agent will never experience a positive reward at all, which makes everything else useless. If you have no good samples to learn from, you can have the most efficient RL method, but the only thing it will learn is that there is no way to get a good reward. This is the case for lots of practically interesting problems around us. For instance, we will take a closer look at the MountainCar environment later in the chapter, which has trivial dynamics, but due to a sparsity of rewards, is quite tricky to solve.

On the other hand, even if the reward is not sparse, effective exploration increases the training speed due to better convergence and training stability. This happens because our sample from the environment becomes more diverse and requires less communication with the environment. As a result, our RL method has the chance to learn a better policy in a shorter time.

What's wrong with ϵ -greedy?

Throughout the book, we have used the ϵ -greedy exploration strategy as a simple, but still acceptable, approach to exploring the environment. The underlying idea behind ϵ -greedy is to take a random action with the probability of ϵ ; otherwise, (with $1 - \epsilon$ probability) we act according to the policy (greedily). By varying the hyperparameter $0 \leq \epsilon \leq 1$, we can change the exploration ratio. This approach was used in most of the value-based methods described in the book.

Quite a similar idea was used in policy-based methods, when our network returns the probability distribution over actions to take. To prevent the network from becoming too certain about actions (by returning a probability of 1 for a specific action and 0 for others), we added the entropy loss, which is just the entropy of the probability distribution multiplied by some hyperparameter. In the early stages of the training, this entropy loss pushes our network toward taking random actions (by regularizing the probability distribution). But in later stages, when we have explored the environment enough and our reward is relatively high, the policy gradient dominates over this entropy regularization. But this hyperparameter requires tuning to work properly.

At a high level, both approaches are doing the same thing: to explore the environment, we introduce randomness into our actions. However, recent research shows that this approach is very far from being ideal:

- In the case of value iteration methods, random actions taken in some pieces of our trajectory introduce bias into our Q-value estimation. The Bellman equation assumes that the Q-value for the next state is obtained from the action with the largest Q . In other words, the rest of the trajectory is supposed to be from our optimal behavior. But with ϵ -greedy, we might take not the optimal action, but just a random action, and this piece of the trajectory will be stored in the replay buffer for a long time, until our ϵ is decayed and old samples are pushed from the buffer. Before that happens, we will learn wrong Q-values.
- With random actions injected into our trajectory, our policy changes with every step. With the frequency defined by the value of ϵ or the entropy loss coefficient, our trajectory constantly switches from a random policy to our current policy. This might lead to poor state space coverage in situations when multiple steps are needed to reach some isolated areas in the environment's state space.

To illustrate the last issue, let's consider a simple example taken from the paper by Strehl and Littman called *An analysis of model-based interval estimation for Markov decision processes*, which was published in 2008 [SL08]. The example is called "River Swim" and it models a river that the agent needs to cross. The environment contains six states and two actions: *left* and *right*. States 1 and 6 are on the river's opposite sides and states 2 to 5 are in the water.

Figure 18.1 shows the transition diagram for the first two states, 1 and 2:

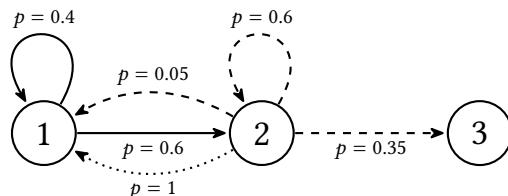


Figure 18.1: Transitions for the first two states of the River Swim environment

In the first state (the circle with the label “1”), the agent stands on the ground of the riverbank. The only action is *right* (shown in solid lines), which means entering the river and swimming against the current to state 2. But the current is strong, and our *right* action from state 1 succeeds only with a probability of 60% (the solid line from state 1 to state 2). With a probability of 40%, the current keeps us in state 1 (the solid line connecting state 1 to itself).

In the second state (the circle with the label “2”), we have two actions: *left*, which is shown by the dotted line connecting states 2 and 1 (this action always succeeds, as the current flushes us back to the riverbank), and *right* (dashed lines), which means swimming against the current to state 3. As before, swimming against the current is hard, so the probability of getting from state 2 to state 3 is just 35% (the dashed line connecting states 2 and 3). With a probability of 60%, our *left* action ends up in the same state (the curved dashed line connecting state 2 to itself). But sometimes, despite our efforts, our *left* action ends up in state 1, which happens with a 5% probability (the curved dashed line connecting states 2 and 1).

As I’ve said, there are six states in River Swim, but the transitions for states 3, 4, and 5 are identical to those for state 2. The last state, 6, is similar to state 1, so there is only one action available there: *left*, meaning to swim back. In *Figure 18.2*, you can see the full transition diagram (which is just clones of the diagram we’ve already seen, where *right* action transitions are shown as solid lines, and *left* action transitions are dotted lines):

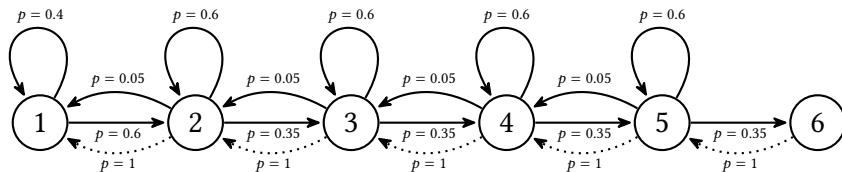


Figure 18.2: The full transition diagram for the River Swim environment

In terms of the reward, the agent gets a small reward of 1 for the transition between states 1 to 5, but it gets a very high reward of 1,000 for getting into state 6, which acts as compensation for all the efforts of swimming against the current.

Despite the simplicity of the environment, its structure creates a problem for the ϵ -greedy strategy being able to fully explore the state space. To check this, I implemented a very simple simulation of this environment, which you will find in `Chapter18/riverswim.py`. The simulated agent always acts randomly ($\epsilon = 1$) and the result of the simulation is the frequency of various state visits. The number of steps the agent can take in one episode is limited to 10, but this can be changed using the command line. We won’t go over the entire code here; you can refer to it in the GitHub repository. Here, let’s look at the results of the experiments:

```
Chapter18$ ./riverswim.py
1:    40
2:    39
3:    17
4:    3
5:    1
6:    0
```

In the preceding output, each line shows the state number and the number of times it was visited during the simulation. With default command-line options, the simulation of 100 steps (10 episodes) was performed. As you can see, the agent never reached state 6 and was only in state 5 once. By increasing the number of episodes, the situation became a bit better, but not much:

```
Chapter18$ ./riverswim.py -n 1000
1:    441
2:    452
3:    93
4:    12
5:    2
6:    0
```

With 10 times more episodes simulated, we still didn't visit state 6, so the agent had no idea about the large reward there.

Only with 10,000 episodes simulated were we able to get to state 6, but only five times, which is 0.05% of all the steps:

```
Chapter18$ ./riverswim.py -n 10000
1:    4056
2:    4506
3:    1095
4:    281
5:    57
6:    5
```

Therefore, it's not very likely that the training will be efficient, even with the best RL method. Also, we had only six states in this example. Imagine how inefficient it will be with 20 or 50 states, which is not that unlikely; for example, in Atari games, there might be hundreds of decisions to be made before something interesting happens.

If you want to, you can experiment with the `riverswim.py` tool, which allows you to change the random seed, the number of steps in the episode, the total number of steps, and even the number of states in the environment.

This simple example illustrates the issue with random actions in exploration. By acting randomly, our agent does not try to actively explore the environment; it just hopes that random actions will bring something new to its experience, which is not always the best thing to do.

Let's now discuss more efficient approaches to the exploration problem.

Alternative ways of exploration

In this section, we will provide you with an overview of a set of alternative approaches to the exploration problem. This won't be an exhaustive list of approaches that exist, but rather will provide an outline of the landscape.

We're going to explore the following three approaches to exploration:

- Randomness in the policy, when stochasticity is added to the policy that we use to get samples. The method in this family is **noisy networks**, which we have already covered in *Chapter 8*.
- **Count-based methods**, which keep track of the number of times the agent has seen the particular state. We will check two methods: the direct counting of states and the pseudo-count method.
- **Prediction-based methods**, which try to predict something from the state and from the quality of the prediction. We can make judgements about the familiarity of the agent with this state. To illustrate this approach, we will take a look at the policy distillation method, which has shown state-of-the-art results on hard-exploration Atari games like Montezuma's Revenge.

Before implementing these methods, let's try and understand them in greater detail.

Noisy networks

Let's start with an approach that is already familiar to us. We covered the method called noisy networks in *Chapter 8*, when we referred to Hessel et al. [Hes+18] and discussed **deep Q-network (DQN)** extensions. The idea is to add Gaussian noise to the network's weights and learn the noise parameters (mean and variance) using backpropagation, in the same way that we learn the model's weights. In that chapter, this simple approach gave a significant boost in Pong training.

At a high level, this might look very similar to the ϵ -greedy approach, but Fortunato et al. [For+17] claimed a difference. The difference lies in the way we apply stochasticity to the network. In ϵ -greedy, randomness is added to the actions.

In noisy networks, randomness is injected into part of the network itself (several fully connected layers close to the output), which means adding stochasticity to our current policy. In addition, parameters of the noise might be learned during the training, so the training process might increase or decrease this policy randomness if needed.

According to the paper, the noise in noisy layers needs to be sampled from time to time, which means that our training samples are not produced by our current policy, but by the ensemble of policies. With this, our exploration becomes directed, as random values added to the weights produce a different policy.

Count-based methods

This family of methods is based on the intuition to visit states that have not been explored before. In simple cases, when the state space is not very large and different states are easily distinguishable from each other, we just count the number of times we have seen the state or state + action and prefer to get to the states for which this count is low.

This could be implemented as an extra reward, not obtained from the environment but from the visit count of the state. In the literature, such a reward is called an *intrinsic reward*. In this context, the reward from the environment is called an *extrinsic reward*. One of the options to formulate such a reward is to use the **bandits exploration** approach: $r_i = c \frac{1}{\sqrt{\tilde{N}(s)}}$. Here, $\tilde{N}(s)$ is a count or pseudo-count of times we have seen the state, s , and value c defines the weight of the intrinsic reward.

If the number of states is small, like in the tabular learning case (we discussed it in *Chapter 5*), we can just count them. In more difficult cases, when there are too many states, some transformation of the states needs to be introduced, like the hashing function or some embeddings of the states (we'll discuss this later in the chapter in more detail).

For pseudo-count methods, $\tilde{N}(s)$ is factorized into the density function and the total number of states visited, given by $\tilde{N}(s) = \rho(x)n(x)$, where $\rho(x)$ is a “density function,” representing the likelihood of the state x and approximated by a neural network. There are several different methods for how to do this, but they might be tricky to implement, so we won't deal with complex cases in this chapter. If you're curious, you can refer to the paper by Georg Ostrovski et al. called *Count-based exploration with neural density models* [Ost+17].

A special case of introducing the intrinsic reward is called *curiosity-driven exploration*, when we don't take the reward from the environment into account at all. In that case, the training and exploration is driven 100% by the novelty of the agent's experience. Surprisingly, this approach might be very efficient not only in discovering new states in the environment but also in learning quite good policies.

Prediction-based methods

The third family of exploration methods is based on another idea of predicting something from the environment data. If the agent can make accurate predictions, it means the agent has been in this situation enough and it isn't worth exploring it.

But if something unusual happens and our prediction is significantly off, it might mean that we need to pay attention to the state that we're currently in. There are many different approaches to doing this, but in this chapter, we will discuss how to implement this approach, as proposed by Burda et al. in 2018 in the paper called *Exploration by random network distillation* [Bur+18]. The authors were able to reach state-of-the-art results in so-called hard-exploration games in Atari.

The approach used in the paper is quite simple: we add the intrinsic reward, which is calculated from the ability of one **neural network (NN)** (which is being trained) to predict the output from another randomly initialized (untrained) NN. The input to both NNs is the current observation, and the intrinsic reward is proportional to the **mean squared error (MSE)** of the prediction.

MountainCar experiments

In this section, we will try to implement and compare the effectiveness of different exploration approaches on a simple, but still challenging, environment, which could be classified as a “classical RL” problem that is very similar to the familiar CartPole problem. But in contrast to CartPole, the MountainCar problem is quite challenging from an exploration point of view.

The problem's illustration is shown in *Figure 18.3* and it consists of a small car starting from the bottom of the valley. The car can move left and right, and the goal is to reach the top of the mountain on the right.

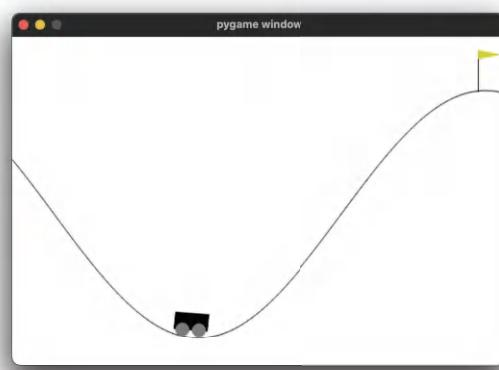


Figure 18.3: The MountainCar environment

The trick here is in the environment's dynamics and the action space. To reach the top, the actions need to be applied in a particular way to swing the car back and forth to speed it up. In other words, the agent needs to apply the actions for several time steps to make the car go faster and eventually reach the top.

Obviously, this coordination of actions is not something that is easy to achieve with just random actions, so the problem is hard from the exploration point of view and very similar to our River Swim example.

In Gym, this environment has the name `MountainCar-v0` and it has a very simple observation and action space. The observations are just two numbers: the first one gives the horizontal position of the car and the second value is the car's velocity. The action could be 0, 1, or 2, where 0 means pushing the car to the left, 1 applies no force, and 2 pushes the car to the right. The following is a very simple illustration of this in Python REPL:

```
>>> import gymnasium as gym
>>> e = gym.make("MountainCar-v0")
>>> e.reset()
(array([-0.56971574,  0.          ], dtype=float32), {})
>>> e.observation_space
Box([-1.2 -0.07], [0.6  0.07], (2,), float32)
>>> e.action_space
Discrete(3)
>>> e.step(0)
(array([-0.570371  , -0.00065523], dtype=float32), -1.0, False, False, {})
>>> e.step(0)
(array([-0.57167655, -0.00130558], dtype=float32), -1.0, False, False, {})
>>> e.step(0)
(array([-0.57362276, -0.00194625], dtype=float32), -1.0, False, False, {})
```

As you can see, in every step, we get the reward of -1, so the agent needs to learn how to get to the goal as soon as possible to get as little total negative reward as possible. By default, the number of steps is limited to 200, so if we haven't reached the goal (which happens most of the time), our total reward is -200.

DQN + ϵ -greedy

The first method that we will use is our traditional ϵ -greedy approach to exploration. It is implemented in the source file `Chapter18/mcar_dqn.py`. I won't include the source code here, as it is already familiar to you. This program implements various exploration strategies on top of the DQN method, allowing us to select between them using the `-p` command-line option. To launch the normal ϵ -greedy method, the option `-p egreedy` needs to be passed. During the training, we are decreasing ϵ from 1.0 to 0.02 for the first 10^5 training steps.

The training is quite fast; it takes just two to three minutes to do 10^5 training steps. But from the charts shown in *Figure 18.4* and *Figure 18.5*, it is obvious that during those 10^5 steps, which was 500 episodes, we didn't reach the goal state even once.

That's really bad news, as our ϵ has decayed, so we will do no more exploration in the future.

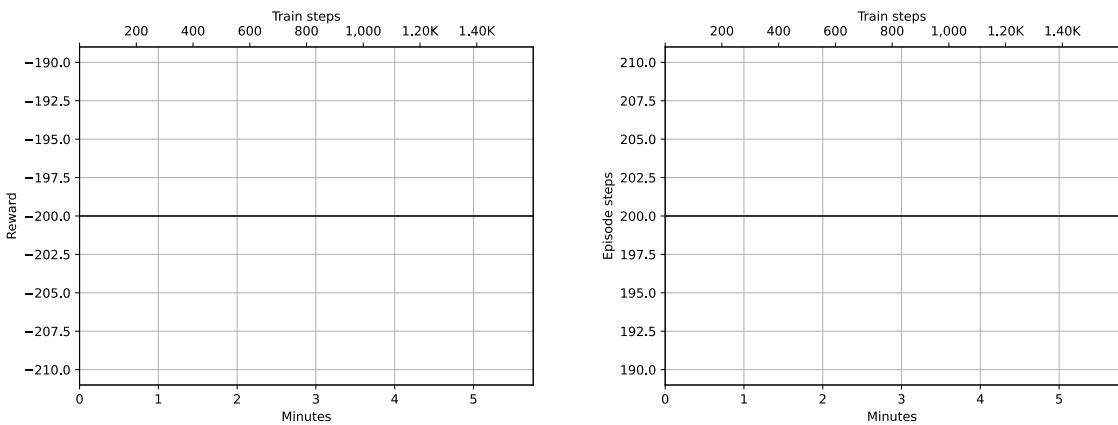


Figure 18.4: The reward (left) and steps (right) during the DQN training with the ϵ -greedy strategy

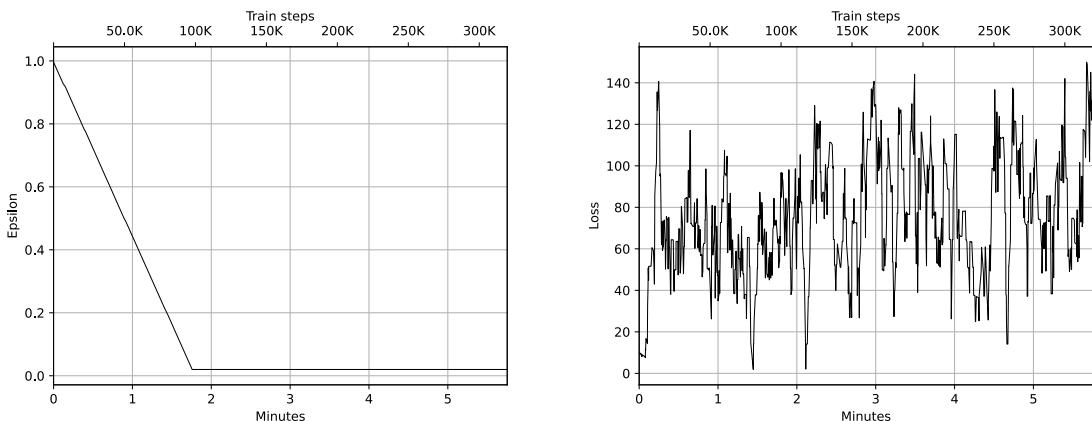


Figure 18.5: Epsilon (left) and loss (right) during the training

The 2% of random actions that we still perform are just not enough because it requires dozens of coordinated steps to reach the top of the mountain (the best policy on MountainCar has a total reward of around -80). We can now continue our training for millions of steps, but the only data we will get from the environment will be episodes, which will take 200 steps with -200 total reward. This illustrates once more how important exploration is. Regardless of the training method we have, without proper exploration, we might just fail to train.

So, what should we do? If we want to stay with ϵ -greedy, the only option for us is to explore for longer (by changing the speed of ϵ decrease). You can experiment with the hyperparameters of the `-p egreedy` mode, but I went to the extreme and implemented the `-p egreedy-long` hyperparameter set. In this regime, we keep $\epsilon = 1.0$ until we reach at least one episode with a total reward better than -200 . Once this has happened, we start training the normal way, decreasing ϵ from 1.0 to 0.02 for subsequent 10^6 frames. As we don't do training during the initial exploration phase, it normally runs 5 to 10 times faster. To start the training in this mode, we use the following command line: `./mcar_dqn.py -n t1 -p egreedy-long`.

Unfortunately, even with this improvement of ϵ -greedy, it still failed to solve the environment due to its complexity. I left this version to run for five hours, but after $500k$ episodes, it still hadn't faced even a single example of the goal, so I gave up. Of course, you could try it for a longer period.

DQN + noisy networks

To apply the noisy networks approach to our MountainCar problem, we just need to replace one of two layers in our network with the `NoisyLinear` class, so our architecture will become as follows:

```
MountainCarNoisyNetDQN(  
    (net): Sequential(  
        (0): Linear(in_features=2, out_features=128, bias=True)  
        (1): ReLU()  
        (2): NoisyLinear(in_features=128, out_features=3, bias=True)  
    )  
)
```

The only difference between the `NoisyLinear` class and the version from *Chapter 8* is that this version has an explicit method, `sample_noise()`, to update the noise tensors, so we need to call this method on every training iteration; otherwise, the noise will be constant during the training. This modification is needed for future experiments with policy-based methods, which require the noise to be constant during the relatively long period of trajectories. In any case, the modification is simple, and we just need to call this method from time to time. In the case of the DQN method, it is called on every training iteration. As in *Chapter 8*, the implementation of `NoisyLinear` is taken from the TorchRL library. The code is the same as before, so to activate the noisy networks, you need to run the training with the `-p noisynet` command line.

In *Figure 18.6*, you can see the plots for the three hours of training:

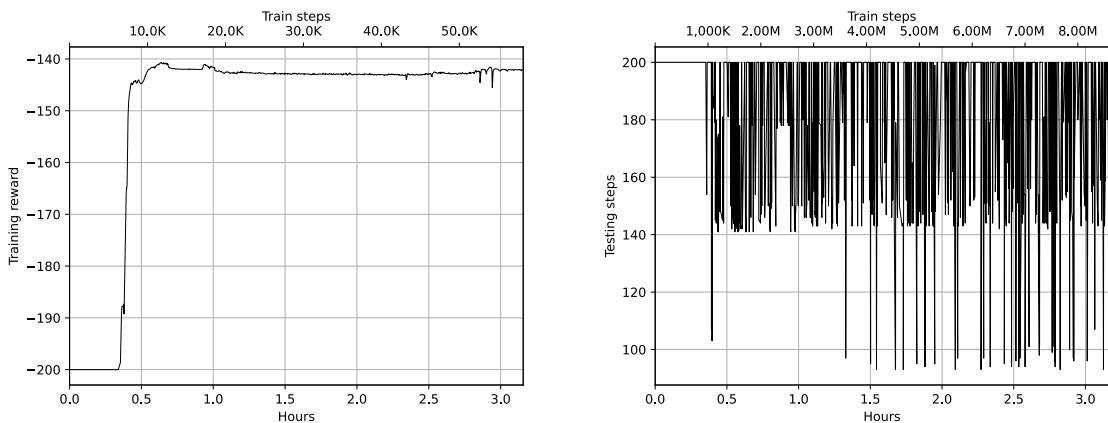


Figure 18.6: The training reward (left) and test steps (right) on DQN with noisy networks exploration

As you can see, the training process wasn't able to reach the mean test reward of -130 (as required in the code), but after just $7k$ training steps (20 minutes of training), we discovered the goal state, which is great progress in comparison to ϵ -greedy, which didn't find a single instance of the goal state after 5 hours of trial and error.

From the test steps chart (on the right of *Figure 18.6*) we can see that there are some tests with less than 100 steps, which is very close to the optimal policy. But they were not often enough to push mean test reward below the -130 level.

DQN + state counts

The last exploration technique that we will apply to the DQN method is count-based. As our state space is just two floating-point values, we will discretize the observation by rounding values to three digits after the decimal point, which should provide enough precision to distinguish different states from each other but still group similar states together. For every individual state, we will keep the count of times we have seen this state before and use that to give an extra reward to the agent. For an off-policy method, it might not be the best idea to modify rewards during the training, but we will examine the effect.

As before, I'm not going to provide the full source code; I will just emphasize the differences from the base version. First, we apply the wrapper to the environment to keep track of the counters and calculate the intrinsic reward value. You will find the code for the wrapper is in the `lib/common.py` module and it is shown here.

Let us look at the constructor first:

```
class PseudoCountRewardWrapper(gym.Wrapper):
    def __init__(self, env: gym.Env, hash_function = lambda o: o,
                 reward_scale: float = 1.0):
        super(PseudoCountRewardWrapper, self).__init__(env)
        self.hash_function = hash_function
        self.reward_scale = reward_scale
        self.counts = collections.Counter()
```

In the constructor, we take the environment we want to wrap, the optional hash function to be applied to the observations, and the scale of the intrinsic reward. We also create the container for our counters, which will map the hashed state into the number of times we have seen it.

Then, we define the helper function:

```
def _count_observation(self, obs) -> float:
    h = self.hash_function(obs)
    self.counts[h] += 1
    return np.sqrt(1/self.counts[h])
```

This function will calculate the intrinsic reward value of the state. It applies the hash to the observation, updates the counter, and calculates the reward using the formula we have already seen.

The last method of the wrapper is responsible for the environment step:

```
def step(self, action):
    obs, reward, done, is_tr, info = self.env.step(action)
    extra_reward = self._count_observation(obs)
    return obs, reward + self.reward_scale * extra_reward, done, is_tr, info
```

Here we call the helper function to get the reward and return the sum of the extrinsic and intrinsic reward components.

To apply the wrapper, we need to pass to it the hashing function:

```
def counts_hash(obs: np.ndarray):
    r = obs.tolist()
    return tuple(map(lambda v: round(v, 3), r))
```

Three digits are probably too many, so you can experiment with a different way of hashing states.

To start the training, pass `-p` counts to the training program. In *Figure 18.7*, you can see the charts with training and testing rewards. As the training environment is wrapped in our `PseudoCountReward` wrapper, values during training are higher than testing.

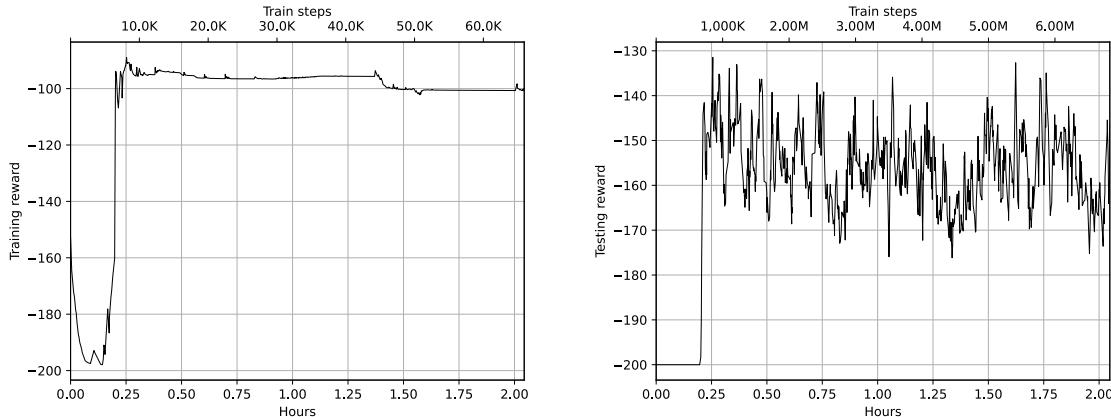


Figure 18.7: The training reward (left) and test rewards (right) on DQN with pseudo-count reward bonus

As you can see, we weren't able to get -130 average test reward using this method, but were very close to it. It took it just 10 minutes to discover the goal state, which is also quite impressive.

PPO method

Another set of experiments that we will conduct with our MountainCar problem is related to the on-policy method **Proximal Policy Optimization (PPO)**, which we covered in *Chapter 16*. There are several motivations for this choice:

- First, as you saw in the DQN method + noisy networks case, when good examples are rare, DQNs have trouble adapting to them quickly. This might be solved by increasing the replay buffer size and switching to the prioritized buffer, or we could try on-policy methods, which adjust the policy immediately according to the obtained experience.
- Another reason for choosing this method is the modification of the reward during the training. Count-based exploration and policy distillation introduce the intrinsic reward component, which might change over time. The value-based methods might be sensitive to the modification of the underlying reward as, basically, they will need to relearn values during the training. On-policy methods shouldn't have any problems with that, as an increase of the reward just puts more emphasis on a sample with higher reward in terms of the policy gradient.
- Finally, it's just interesting to check our exploration strategies on both families of RL methods.

To implement this approach, in the file `Chapter18/mcar_ppo.py`, we have a PPO implementation combined with various exploration strategies applied to MountainCar. The code is not very different from the PPO from *Chapter 16*, so I'm not going to repeat it here. To start the normal PPO without extra exploration tweaks, you should run the command `./mcar_ppo.py -n t1 -p ppo`. In this version, nothing specifically is done to perform exploration – we purely rely on random weights initialization in the beginning of the training.

As a reminder, PPO is in the policy gradient methods family, which limits Kullback-Leibler divergence between the old and new policy during the training, avoiding dramatic policy updates. Our network has two heads: the actor and the critic. The actor network returns the probability distribution over our actions (our policy) and the critic estimates the value of the state. The critic is trained using MSE loss, while the actor is driven by the PPO surrogate objective we discussed in *Chapter 16*. In addition to those two losses, we regularize the policy by applying entropy loss scaled by the hyperparameter β . There is nothing new here so far. The following is the PPO network structure:

```
MountainCarBasePPO(  
    (actor): Sequential(  
        (0): Linear(in_features=2, out_features=64, bias=True)  
        (1): ReLU()  
        (2): Linear(in_features=64, out_features=3, bias=True)  
    )  
    (critic): Sequential(  
        (0): Linear(in_features=2, out_features=64, bias=True)  
        (1): ReLU()  
        (2): Linear(in_features=64, out_features=1, bias=True)  
    )  
)
```

I stopped the training after three hours, as it showed no improvements. The goal state was found after an hour and 30k episodes.

The charts in *Figure 18.8* show the reward dynamics during the training:

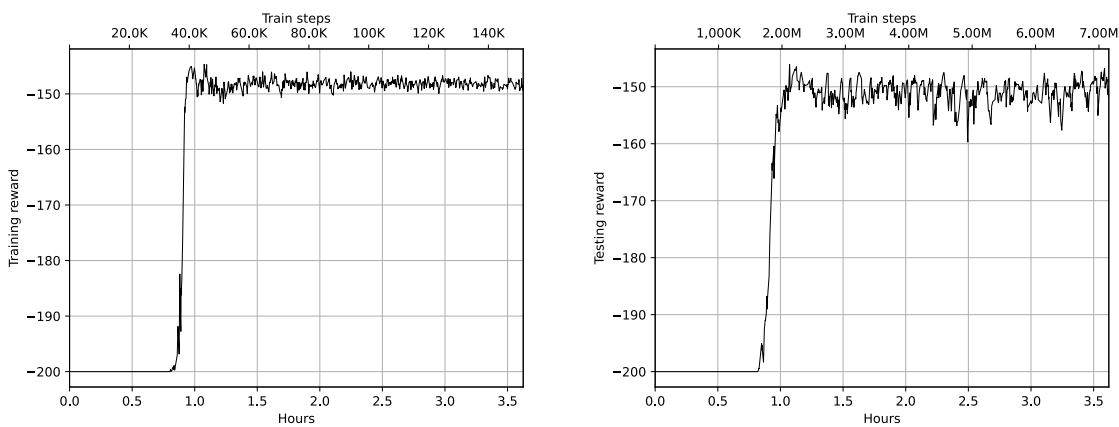


Figure 18.8: The training reward (left) and test rewards (right) on plain PPO

As the PPO result wasn't very impressive, let's try to extend it with extra exploration tricks.

PPO + Noisy Networks

As with the DQN method, we can apply the noisy networks exploration approach to our PPO method. To do that, we need to replace the output layer of the actor with the `NoisyLinear` layer. Only the actor network needs to be affected because we would like to inject the noisiness only into the policy and not into the value estimation.

There is one subtle nuance related to the application of noisy networks: where the random noise needs to be sampled. In *Chapter 8*, when you first met noisy networks, the noise was sampled on every `forward()` pass of the `NoisyLinear` layer. According to the original research paper, this is fine for off-policy methods, but for on-policy methods, it needs to be done differently. Indeed, when we train on-policy, we obtain the training samples produced by our current policy and calculate the policy gradient, which should push the policy toward improvement. The goal of noisy networks is to inject randomness, but as we have discussed, we prefer directed exploration over just a random change of policy after every step. With that in mind, our random component in the `NoisyLinear` layer needs to be updated not after every `forward()` pass, but much less frequently. In my code, I resampled the noise on every PPO batch, which was 2,048 transitions.

As before, I trained PPO+NoisyNets for 3 hours. But in this case, the goal state was found after 30 minutes and 18k episodes, which is a better result. In addition, according to the train steps count, the training process was able to drive the car in the optimal way a couple of times (with a step count less than 100). But these successes did not lead to the optimal policy at the end. The charts in *Figure 18.9* show the reward dynamics during the training:

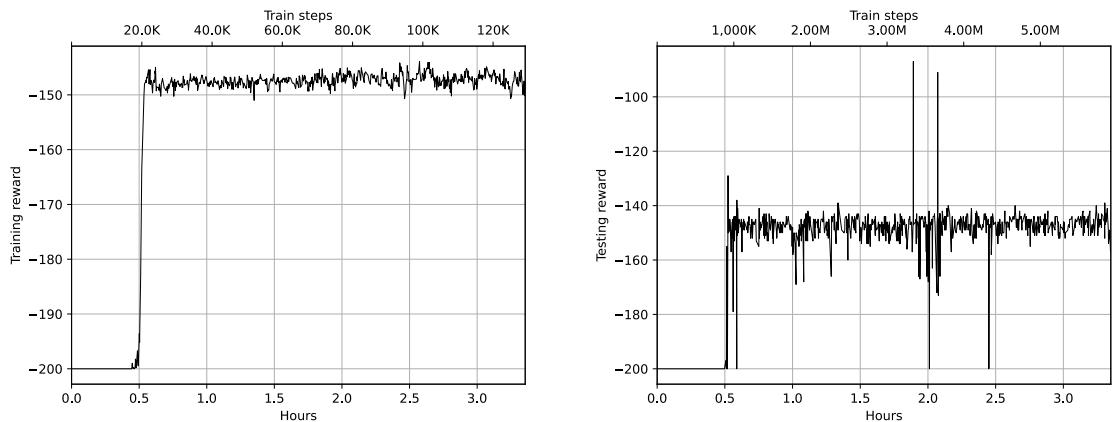


Figure 18.9: The training reward (left) and test rewards (right) on PPO with Noisy Networks

PPO + state counts

In this case, exactly the same count-based approach with three-digit hashing is implemented for the PPO method and can be triggered by passing `-p` counts to the training process.

In my experiments, the method was able to solve the environment (get an average reward higher than -130) in 1.5 hours, and it required 61k episodes. The following is the final part of the console output:

```
Episode 61454: reward=-159.17, steps=168, speed=4581.6 f/s, elapsed=1:37:18
Episode 61455: reward=-158.46, steps=164, speed=4609.0 f/s, elapsed=1:37:18
Episode 61456: reward=-158.41, steps=164, speed=4582.3 f/s, elapsed=1:37:18
Episode 61457: reward=-152.73, steps=158, speed=4556.4 f/s, elapsed=1:37:18
Episode 61458: reward=-154.08, steps=159, speed=4548.1 f/s, elapsed=1:37:18
Episode 61459: reward=-154.85, steps=162, speed=4513.0 f/s, elapsed=1:37:18
Test done: got -91.000 reward after 91 steps, avg reward -129.999
Reward boundary has crossed, stopping training. Congrats!
```

As you can see from the plots in *Figure 18.10*, the training discovered the goal state after 23k episodes. It took another 40k episodes to polish the policy to the optimal count of steps:

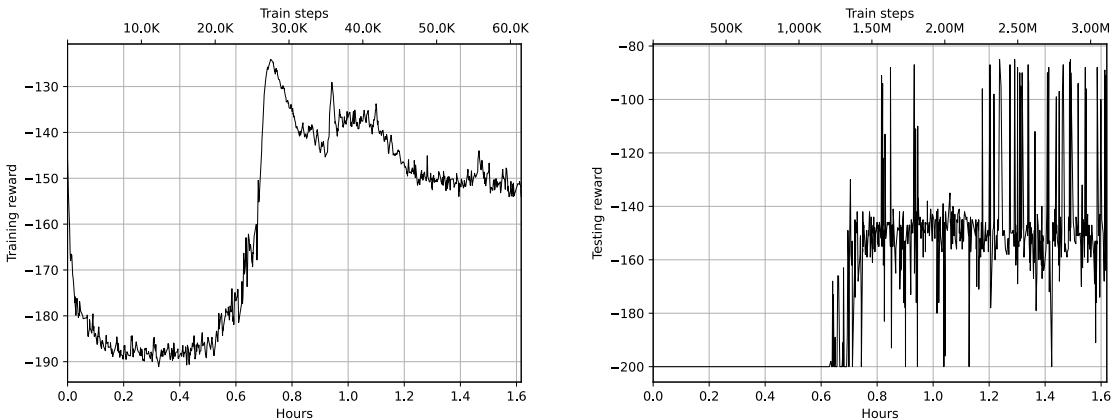


Figure 18.10: The training reward (left) and test rewards (right) on PPO with pseudo-count reward bonus

PPO + network distillation

As the final exploration method in our MountainCar experiment, I implemented the network distillation method proposed by Burda et al. [Bur+18]. In this method, two extra NNs are introduced. Both need to map the observation into one number, in the same way that our value head does. The difference is in the way they are used. The first NN is randomly initialized and kept untrained. This will be our reference NN. The second one is trained to minimize the MSE loss between the second and the first NN. In addition, the absolute difference between the outputs of the NNs is used as the intrinsic reward component.

The idea behind this is that the better the agent has explored some state, the better our second (trained) NN will predict the output of the first (untrained) one. This will lead to a smaller intrinsic reward being added to the total reward, which will decrease the policy gradient assigned to the sample.

In the paper, the authors suggested training separate value heads to predict separate intrinsic and extrinsic reward components, but for this example, I decided to keep it simple and just added both rewards in the wrapper, the same way that we did in the counter-based exploration method. This minimizes the number of modifications in the code.

In terms of those extra NN architectures, I did a small experiment and tried several architectures for both NNs. The best results were obtained with the reference NN having three layers and the trained NN having just one layer. This helps to prevent the overfitting of the trained NN, as our observation space is not very large.

Both NNs are implemented in the `MountainCarNetDistillery` class in the `lib/ppo.py` module:

```

class MountainCarNetDistillery(nn.Module):
    def __init__(self, obs_size: int, hid_size: int = 128):
        super(MountainCarNetDistillery, self).__init__()

        self.ref_net = nn.Sequential(
            nn.Linear(obs_size, hid_size),
            nn.ReLU(),
            nn.Linear(hid_size, hid_size),
            nn.ReLU(),
            nn.Linear(hid_size, 1),
        )
        self.ref_net.train(False)

        self.trn_net = nn.Sequential(
            nn.Linear(obs_size, 1),
        )

    def forward(self, x):
        return self.ref_net(x), self.trn_net(x)

    def extra_reward(self, obs):
        r1, r2 = self.forward(torch.FloatTensor([obs]))
        return (r1 - r2).abs().detach().numpy()[0][0]

    def loss(self, obs_t):
        r1_t, r2_t = self.forward(obs_t)
        return F.mse_loss(r2_t, r1_t).mean()

```

Besides the `forward()` method, which returns the output from both NNs, the class includes two helper methods for intrinsic reward calculation and for getting the loss between two NNs.

To start the training, the argument `-p distill` needs to be passed to the `mcar_ppo.py` program. In my experiment, 33k episodes were required to solve the problem, which is almost two times less than Noisy Networks. As discussed in earlier chapters, there might be some bugs and inefficiencies in my implementation, so you're welcome to modify it to make it faster and more efficient:

```

Episode 33566: reward=-93.27, steps=149, speed=2962.8 f/s, elapsed=1:23:48
Episode 33567: reward=-82.13, steps=144, speed=2968.6 f/s, elapsed=1:23:48
Episode 33568: reward=-83.77, steps=143, speed=2973.7 f/s, elapsed=1:23:48
Episode 33569: reward=-93.59, steps=160, speed=2974.0 f/s, elapsed=1:23:48
Episode 33570: reward=-83.04, steps=143, speed=2979.7 f/s, elapsed=1:23:48
Episode 33571: reward=-97.96, steps=158, speed=2984.5 f/s, elapsed=1:23:48

```

```
Episode 33572: reward=-92.60, steps=150, speed=2989.8 f/s, elapsed=1:23:48
Test done: got -87.000 reward after 87 steps, avg reward -129.549
Reward boundary has crossed, stopping training. Congrats!
```

The plots with the training and testing rewards are shown in *Figure 18.11*. In *Figure 18.12*, the total loss and distillation loss are shown.

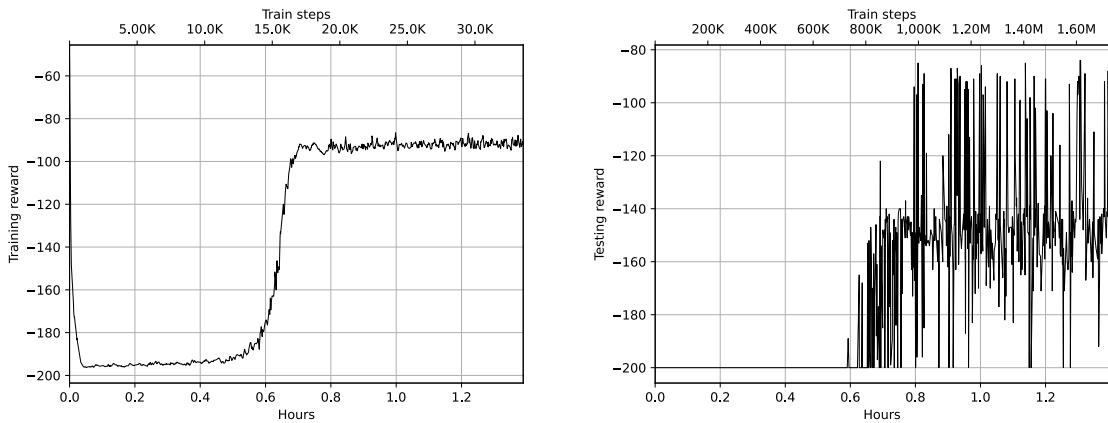


Figure 18.11: The training reward (left) and test rewards (right) on PPO with network distillation

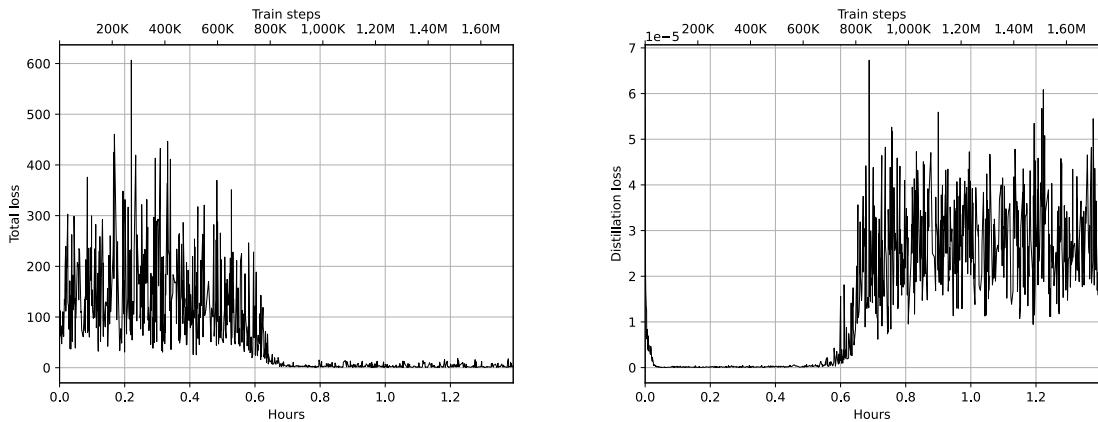


Figure 18.12: The total loss (left) and distillation loss (right)

As before, due to the intrinsic reward component, the training episodes have a higher reward on the plots.

From the distillation loss plot, it is clear that before the agent discovered the goal state, everything was boring and predictable, but once it had figured out how to end the episode earlier than 200 steps, the loss grew significantly.

Comparison of methods

To simplify the comparison of the experiments we've done on MountainCar, I put all the numbers into the following table: As you can see, both DQN and PPO with exploration extensions are able to solve the

Method	Goal state found		Solved	
	Episodes	Time	Episodes	Time
DQN + ϵ -greedy	x	x	x	x
DQN + noisy nets	8k	15 min	x	x
PPO	40k	60 min	x	x
PPO + noisy nets	20k	30 min	x	x
PPO + counts	25k	36 min	61k	90 min
PPO + distillation	16k	36 min	33k	84 min

Table 18.1: Summary of experiments

MountainCar environment. Concrete method selection is up to you and your concrete situation, but it is important to be aware of the different approaches to the exploration you might use.

Atari experiments

The MountainCar environment is a nice and fast way to experiment with exploration methods, but to conclude the chapter, I've included Atari versions of the DQN and PPO methods with the exploration tweaks we described to check a more complicated environment.

As the primary environment, I've used Seaquest, which is a game where the submarine needs to shoot fish and enemy submarines, and save aquanauts. This game is not as famous as Montezuma's Revenge, but it still might be considered as medium-hard exploration because, to continue the game, you need to control the level of oxygen. When it becomes low, the submarine needs to rise to the surface for some time. Without this, the episode will end after 560 steps and with a maximum reward of 20. But once the agent learns how to replenish the oxygen, the game might continue almost infinitely and bring to the agent a 10k-100k score. Surprisingly, traditional exploration methods struggle with discovering this; normally, training gets stuck at 560 steps, after which the oxygen runs out and the submarine dies.



The negative aspect of Atari is that every experiment requires at least half a day of training to check the effect, so my code and hyperparameters are very far from being the best, but they might be useful as a starting point for your own experiments. Of course, if you discover a way to improve the code, please share your findings on GitHub.

As before, there are two program files: `atari_dqn.py`, which implements the DQN method with ϵ -greedy and noisy networks exploration, and `atari_ppo.py`, which is the PPO method with optional noisy networks and the network distillation method. To switch between hyperparameters, the command-line option `-p` needs to be used.

In the following sections, let us look at the results that I got from a few runs of the code.

DQN + ϵ -greedy

In comparison to other methods tried on Atari, ϵ -greedy was the best, which might be surprising, as it gave the worst results in the MountainCar experiment earlier in this chapter. But this happens quite often in reality and can lead to new directions of research and even breakthroughs. After 13 hours of training, it was able to reach an average reward of 18 with a maximum reward of 25. According to the chart showing the number of steps, just a few episodes were able to discover how to get the oxygen so, maybe with more training, this method can break the 560-step boundary. In *Figure 18.13*, the plots with the average reward and number of steps are shown:

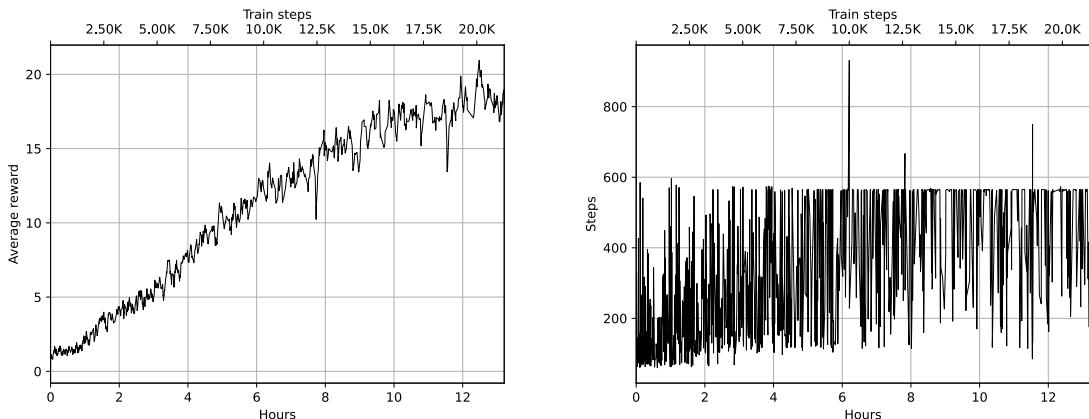


Figure 18.13: The average training reward (left) and count of steps (right) on DQN with ϵ -greedy

DQN + noisy networks

Noisy networks combined with DQN showed worse results — after 6 hours of training, it was able to reach a reward of 6. In *Figure 18.14*, the plots are shown:

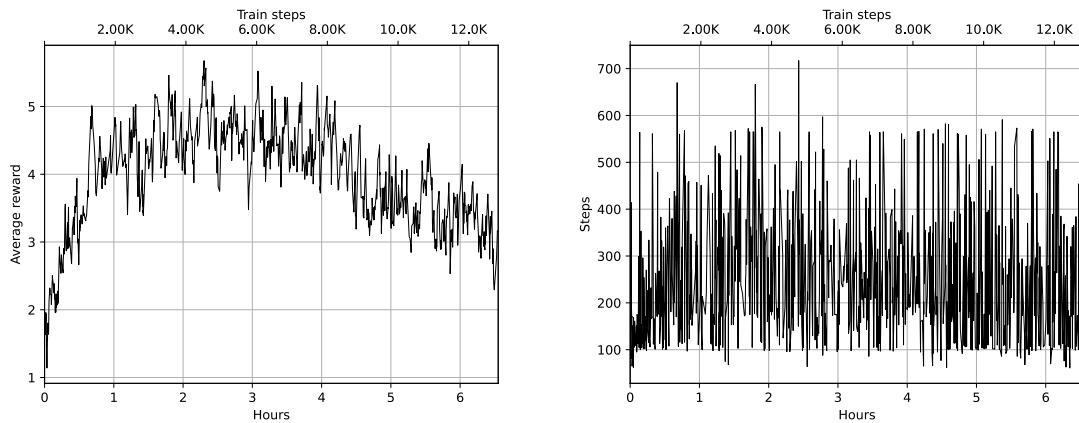


Figure 18.14: The average training reward (left) and count of steps (right) on DQN with noisy networks

PPO

PPO experiments were much worse — all the combinations (vanilla PPO, noisy networks, and network distillation) showed no reward progress and were able to reach an average reward of 4. This is a bit surprising, as experiments with the same code in the previous edition of the book were able to get better results. This might be an indication of some subtle bugs in the code or in the training environment I used. Feel free to experiment with these methods by yourself!

Summary

In this chapter, we discussed why ϵ -greedy exploration is not the best in some cases and checked alternative modern approaches for exploration. The topic of exploration is much wider and lots of interesting methods are left uncovered, but I hope you were able to get an overall impression of the new methods and the way they should be implemented and used in your own problems.

In the next chapter, we'll take a look at another approach to the exploration in complex environments: **RL with human feedback (RLHF)**.

19

Reinforcement Learning with Human Feedback

In this chapter, we'll take a look at a relatively recent method that addresses situations when the desired behavior is hard to define via the explicit reward function – **reinforcement learning with human feedback (RLHF)**. This is also related to exploration (as the method allows humans to push learning in a new direction), the problem we discussed in *Chapter 18*. Surprisingly, the method, initially developed for a very specific subproblem in the RL domain, turned out to be enormously successful in the **large language models (LLMs)**. Nowadays, RLHF is at the core of modern LLM training pipelines, and without it, the recent fascinating progress wouldn't have been possible.

As this book is not about LLMs and modern chatbots, we will focus purely on the original paper from OpenAI and Google by Christiano et al., *Deep reinforcement learning from human preferences* [Chr+17], which describes the RLHF method applied to RL problems and environments. But in the overview of the method, I will explain a bit about how this method is used in LLM training.

In this chapter, we will:

- Take a look at human feedback in RL to address problems with unclear reward objectives and exploration.

- Implement an RLHF pipeline from scratch and check it on the SeaQuest Atari game to teach it new behavior.

Reward functions in complex environments

Before we go into the details of the RLHF method, let's start by discussing the underlying motivation of the concept. As we discussed in *Chapter 1*, the *reward* is the core concept in RL. Without a reward, we're blind – all the methods we've already discussed are heavily dependent on the reward value provided by the environment:

- In value-based methods (*Part 2* of the book), we used the reward to approximate the Q -value to evaluate the actions and choose the most prominent one.
- In policy-based methods (*Part 3*), the reward was used even more directly – as a scale for the Policy Gradient. With all the math removed, we basically optimized our policy to prefer actions that bring more accumulated future reward.
- In black-box methods (*Chapter 17*), we used the reward to make a decision about agent variants: should they be kept for the future or discarded?

In almost all the RL environments we've experimented with, the reward function was predefined for us – in Atari games, we had the score; in the FrozenLake environment, it was an explicit target position; in simulated robots, it was the distance travelled, etc. The only exception was in *Chapter 10*, where we implemented the environment (stock trading system) ourselves and had to decide how the reward was to be shaped. But even in that example, it was fairly obvious what should be used as a reward.

Unfortunately, in real-life situations, it is not always that easy to formulate what should be used as a reward. Let's look at a couple of examples. If we are training the chatbot to solve a set of tasks, it's important to not only ensure the tasks are completed correctly but also consider the style in which they are done. What if we ask the system "What's the weather forecast for tomorrow?" and it replies correctly but in a rude manner? Should it be punished for this with a negative reward and to what extent? What should we do in the opposite situation – a very polite answer but the information given is wrong? If we just optimize one single criterion (like the correctness of information), we might get a system that "works" but is not usable in real life – just because it is so awkward that nobody wants to use it.

Another example of a "single optimization factor" is the transportation of goods from point A to point B. Transport companies don't just try to maximize their profits by all means. In addition, they have tons of restrictions and regulations, like driving rules, working hours, labour legislation, etc. If we optimize only one criterion in our system, we might eventually get "Drive through the neighbor's fence – this is the fastest way."

So, in real life, having a single value we want to maximize is an exception rather than the norm. Most of the time, we have several parameters that contribute to the final result and we need to find some sort of balance between them. Even in the Atari games we've already seen, the score might be calculated as the sum of different "subgoals." A very good example of this is the SeaQuest game we experimented with in the previous chapter. If you haven't played it before, you can do it in your browser to get a better understanding: https://www.retrogames.cz/play_221-Atari2600.php.

In this game, you're controlling the submarine and you are scored for the following activities:

- Shooting evil fish and enemy submarines
- Saving divers and bringing them back to the surface
- Avoiding enemy fire and ships on the surface (they appear in later levels of the game)

As the level of oxygen is limited, your submarine has to go to the surface from time to time to refill the reserves. Most of the modern RL methods have no problem discovering the reward for shooting fish and submarines – starting with trial and error, after just a couple of hours of training, the agent learns how to get the reward from firing.

But discovering scoring from saving divers is much trickier, as the reward for them is given only after collecting six divers and getting to the surface. Oxygen replenishment is also hard to discover by trial and error, as our neural network has no prior idea about oxygen, submarines, and how the sudden death of your submarine might be related to the gauge at the bottom of the screen. Our RL method with ϵ -greedy exploration could be seen as a newborn baby randomly pushing buttons and being rewarded for correct sequences of actions, which might take lots of time before the correct lengthy sequence has been executed.

As a result, most of the training episodes in SeaQuest are limited by the average score of 300 and 500 game steps. The submarine just dies from a lack of oxygen and random surface visits are too rare to discover that the game might be played for much longer. At the same time, people who haven't seen the game before can figure out how to refill the oxygen and save divers in just several minutes of gameplay.

Potentially, we could help our agent and explain somehow why oxygen is important by adding it to the reward function (as an extra reward for refilling the oxygen, for example), but it might start the vicious circle of tweaking the environment here and there – exactly those efforts we've tried to avoid by using RL methods.

And, as you might already have guessed, RLHF is exactly the approach that allows us to avoid this low-level reward function tweaking, allowing humans to give feedback to the agent's behavior.

Theoretical background

Let's take a look at the original RLHF method published in 2017 by OpenAI and Google researchers [Chr+17]. Since the publication (and especially after ChatGPT's release), this method has been an area of active research. For recent developments, you can check papers at <https://github.com/opendilab/awesome-RLHF>. In addition, we'll discuss the role of RLHF in the LLM training process.

Method overview

The authors of the paper experimented with two classes of problems: several environments from MuJoCo simulated robotics (similar to the continuous control problems we discussed in *Chapter 15* and *Chapter 16*) and several Atari games.

The core idea is to keep the original RL model, but replace the reward from the environment with a neural network called *reward predictor*, which is trained on data gathered by humans. This network (represented as $\hat{r}(o, a)$ in the paper) takes the observation and the action and returns the float value of immediate reward for the action.

The training data for this reward predictor is not provided directly by humans, but deducted from *human preferences*: people are shown two short video clips with examples of the agent's behavior and asked the question "Which one is better?". In other words, the training data for the reward predictor is two episode segments σ^1 and σ^2 (fixed-length sequences of (o_t, a_t) with observations and actions) and label μ from the human indicating which of the two is preferred. The given answer options are "first," "second," "both are good," and "cannot judge."

The network $\hat{r}(o, a)$ is trained from this data using cross-entropy loss between labels and the function $\hat{P}[\sigma^1 \succ \sigma^2]$, which is an estimation of the probability of the human preferring segment σ^1 over σ^2 :

$$\hat{P}[\sigma^1 \succ \sigma^2] = \frac{e^{\sum \hat{r}(o_t^1, a_t^1)}}{e^{\sum \hat{r}(o_t^1, a_t^1)} + e^{\sum \hat{r}(o_t^2, a_t^2)}}$$

In other words, we sum the rewards predicted for every step in the segment, exponentiate every reward, and normalize the sum. The cross-entropy loss is calculated using the standard formula for the binary classification:

$$\text{loss}(\hat{r}) = - \sum_{(\sigma^1, \sigma^2, \mu) \in D} \mu_1 \log \hat{P}[\sigma^1 \succ \sigma^2] + \mu_2 \log \hat{P}[\sigma^2 \succ \sigma^1]$$

Values for μ_1 and μ_2 are assigned based on the human's judgement. If the first segment was preferred over the second, then $\mu_1 = 1$ and $\mu_2 = 0$. If the second segment was better, then $\mu_2 = 1$ and $\mu_1 = 0$. If the human decided that both segments are good, then both μ are set to 0.5.

Such a reward model has several benefits in comparison to different approaches:

- By using a neural network for reward prediction, we can significantly reduce the required number of labels. An extreme case would be to ask humans to label every action of the policy, but this is prohibitively expensive in the case of RL, where millions of interactions take place within the environment. In the case of high-level goals, this might be an almost impossible thing to do.
- We give the network feedback not only about good behavior but also about behavior that we don't like. If you remember, in *Chapter 14*, we used the recorded human demonstrations to train the web automation agent. But human demonstrations only show positive examples ("do this") and have no way of including negative examples ("don't do that"). In addition, human demonstrations are harder to collect and might contain more errors.
- By asking for human preferences, we can handle problems where humans can *recognize* the behavior we want, but not necessarily *reproduce* it. For example, controlling the four-legged Ant robot from *Chapter 16* might be very challenging for humans. At the same time, we don't have problems detecting when the robot is behaving normally or the policy is wrong.

In the RLHF paper, the authors experimented with different approaches to the reward model training and its usage in the RL training process. In their setup, three different processes were running in parallel:

1. The RL training method (A2C) used the current $\hat{r}(o, a)$ network for reward prediction. Random trajectory segments $\sigma = (o_i, a_i)$ were stored in the labeling database.
2. Human labelers sampled pairs of segments (σ^1, σ^2) and assigned their labels μ , which were stored in the labeling database.
3. The reward model $\hat{r}(o, a)$ was periodically trained on labeled pairs from the database and sent to the RL training process.

This process is shown in *Figure 19.1*.

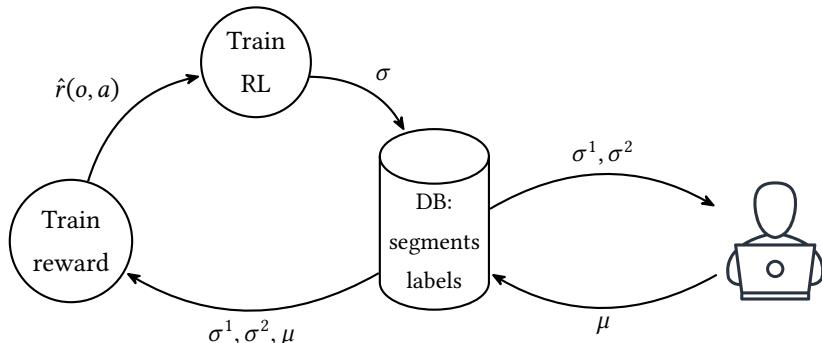


Figure 19.1: RLHF structure

As discussed earlier, the paper addressed two classes of problems: Atari games and continuous control. On both classes, the results were not especially spectacular — sometimes traditional RL was better than RLHF, sometimes not. But where RLHF really stood out was the LLM training pipeline. Let's briefly discuss why it happened before we start our RLHF experiments.

RLHF and LLMs

ChatGPT, released at the end of 2022, has quickly become a really big thing. For the general audience, it was even more influential than AlexNet in 2012, as AlexNet was “techy stuff”—it pushed the boundaries but it was much harder to explain what was so special about it. ChatGPT was different: just a month after release, it had surpassed a user base of 100M users and almost everybody was talking about it.

At the heart of ChatGPT (and any modern LLM) training pipeline is RLHF. So, very quickly, this method of fine-tuning large models has become popular and has grown in terms of research interest. As this is not a book about LLMs, I will just give a quick description of the pipeline and how RLHF is incorporated there, as, from my perspective, this is an interesting use case.

From a high level, LLM training consists of three stages:

1. **Pretraining:** Here, we perform the initial training of the language model on a huge corpus of texts. Basically, we take all the information we can possibly get and do unsupervised training of the language model. The volume (and costs) are enormous — the RedPajama dataset used for LLaMA training contains 1.2 trillion tokens (which is approximately 15 million books).

At this stage, our randomly-initialized model learns regularities and deep connections of the language. But because the data volume is huge, we cannot just curate this data — it could be fake news, hate speech posts, and other weird stuff you can easily find on the internet.

2. **Supervised fine-tuning:** In this step, we fine-tune the model on predefined curated example dialogues. The dataset used here is manually created and validated for correctness and the volume is significantly lower — around 10K–100K example dialogues.

This data is normally created by experts in the field and requires lots of effort to make and double-check it.

3. **RLHF fine-tuning** (also known as “model alignment”): This step uses the same process we've already described: pairs of generated dialogues are presented to users for labeling, the reward model is trained on those labels, and this reward model is used in the RL algorithm to fine-tune the LLM model to follow the human's preferences. The number of labeled samples is larger than on the supervised fine-tuning step (around 1M pairs), but because comparing two dialogues is a much simpler task than creating a proper dialogue from scratch, this is not a problem.

As you might guess, the first step is the most expensive and lengthy: you have to crunch terabytes of texts and

feed them through transformers. But at the same time, the *importance* of the steps is totally different. In the last step, the system not only learns what the best solution to the presented problem is but also has feedback about generating it in a socially acceptable way.

The RLHF method is very suitable for this task — with just pairs of dialogues, it can learn the reward model that represents the labelers’ implicit “preference model” for such a complicated thing as the chatbot. Doing this explicitly (via the reward function, for example) might be a very challenging problem with lots of uncertainty.

RLHF experiments

To get a better understanding of the pipeline we’ve just discussed, let’s implement it ourselves (as “doing is the best way to learn something”). In the previous chapter, we tried the Atari SeaQuest environment, which is tricky from the exploration point of view, so it is logical to take this environment and check what we can achieve with human feedback.

To limit the scope of the chapter and make the example more reproducible, I made the following modifications to the experiments described in the RLHF paper [Chr+17]:

- I focused on a single SeaQuest environment. The goal was to improve the agent’s gameplay in comparison to the A2C results we got in *Chapter 18* — an average score of 400 and episodes of 500 steps (due to the lack of oxygen).
- Instead of asynchronous labeling and reward model training, I split them into separate steps:
 1. A2C training was performed, storing trajectory segments in local files. This training might optionally load and use a reward model network, which would allow us to iterate on reward models, labeling more samples after the training.
 2. The web UI allowed me to label random pairs of trajectory segments, storing the labels in a JSON file.
 3. The reward model was trained on those segments and labels. The result of the training was stored on disk.
- I avoided all the variations with the reward model training: no L2 regularization, no ensemble, etc.
- The number of labels was significantly smaller: in every experiment, I labeling an extra 100 pairs of episode segments and retrained the models.
- Actions were explicitly added to the reward model. For the details, check the section *Reward model*.
- The reward model was used in A2C training for the fine-tuning of the best mode saved. For context, in the paper, the model was trained from scratch and improved with parallel RLHF labeling and reward model retraining.

Initial training using A2C

To get the first model (let's call it "version 0" or v0 for short), I used standard A2C code with the same Atari wrappers we've already discussed several times in this book so far.

To start the training, you need to run the `Chapter19/01_a2c.py` module, and besides basic A2C training, it contains a command-line option that enables the usage of the reward model (which we covered in earlier chapters), but we don't need it in this step.

For now, to start the training of the basic model, use the following command line:

```
Chapter19$ ./01_a2c.py --dev cuda -n v0 --save save/v0 --db-path db-v0
A.L.E: Arcade Learning Environment (version 0.8.1+53f58b7)
[Powered by Stella]
AtariA2C(
    (conv): Sequential(
        (0): Conv2d(4, 32, kernel_size=(8, 8), stride=(4, 4))
        (1): ReLU()
        (2): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2))
        (3): ReLU()
        (4): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1))
        (5): ReLU()
        (6): Flatten(start_dim=1, end_dim=-1)
    )
    (policy): Sequential(
        (0): Linear(in_features=3136, out_features=512, bias=True)
        (1): ReLU()
        (2): Linear(in_features=512, out_features=18, bias=True)
    )
    (value): Sequential(
        (0): Linear(in_features=3136, out_features=512, bias=True)
        (1): ReLU()
        (2): Linear(in_features=512, out_features=1, bias=True)
    )
)
0: Testing model...
Got best reward 40.00 and steps 213.0 in 10 episodes
1024: done 1 games, mean reward 0.000, steps 70, speed 312.22 f/s
1056: done 2 games, mean reward 0.000, steps 72, speed 1188.69 f/s
1104: done 3 games, mean reward 0.000, steps 75, speed 1216.18 f/s
```

Here is the description of the command-line options:

- `-dev`: The name of the device used for computation.
- `-n`: The name of the run, used in TensorBoard.

- **-save:** The directory name where the best models after the testing will be stored. Every 100 batches, we perform 10 test episodes of the current model on SeaQuest with disabled reward clipping (to get the original score range) and if the best reward or the count of steps for any of those 10 rounds is better than our previous record, we save the model into the file. Those files will be used later for fine-tuning.
- **-db-path:** The directory name where random episode segments will be stored during the training. This data will be used for the labeling and training of the reward model later.

Let's discuss the episode segments database (DB for short). Its structure is very simple: every environment used for training (in total, we have 16 of them) has an identifier from 0 to 15, which is used as a subdirectory under the directory given in the -db-path command-line argument. So, every environment stores random segments independently in its own directory. The storage logic is implemented in a Gym API Wrapper subclass, which is called `EpisodeRecorderWrapper` and is in the `lib/rlhf.py` module.

Let's take a look at the source code of the wrapper. Initially, we declare two hyperparameters, `EPISODE_STEPS`, which defines the length of segments, and `START_PROB`, which is the probability of starting the episode recording:

```
# how many transitions to store in episode
EPISODE_STEPS = 50
# probability to start episode recording
START_PROB = 0.00005

@dataclass(frozen=True)
class EpisodeStep:
    obs: np.ndarray
    act: int

class EpisodeRecorderWrapper(gym.Wrapper):
    def __init__(self, env: gym.Env, db_path: pathlib.Path, env_idx: int,
                 start_prob: float = START_PROB, steps_count: int = EPISODE_STEPS):
        super().__init__(env)
        self._store_path = db_path / f"{env_idx:02d}"
        self._store_path.mkdir(parents=True, exist_ok=True)
        self._start_prob = start_prob
        self._steps_count = steps_count
        self._is_storing = False
        self._steps: tt.List[EpisodeStep] = []
        self._prev_obs = None
        self._step_idx = 0
```

We store the episode segment as a list of `EpisodeStep` objects, which is just an observation and the action we're taking at this step.

The method that resets the environment is very simple – it updates the wrapper’s `_step_idx` field (which is a counter of the steps we’ve done in this environment) and stores the observation in the `_prev_obs` field, depending on the `_is_storing` field. This field is `True` if we’re in the middle of segment recording.

Our segments have a fixed number of environment steps (50 by default) and they are recorded independent of episode boundaries (in other words, if we started the segment recording shortly before the submarine’s death, we’ll record the beginning of the next episode after the `reset()` method):

```
def reset(self, *, seed: int | None = None, options: dict[str, tt.Any] | None = None)
\
    -> tuple[WrapperObsType, dict[str, tt.Any]]:
    self._step_idx += 1
    res = super().reset(seed=seed, options=options)
    if self._is_storing:
        self._prev_obs = deepcopy(res[0])
    return res
```

If you want, you can experiment with this logic as, in principle, observations after the end of the episode are independent from observations and actions before the end of the episode. But it will make the handling of episode segment data more complicated, as the length will become variable.

The main logic of the wrapper is in the `step()` method and it is also not very complicated. On every action, we store the step if we’re in the middle of recording; otherwise, we generate a random number to make the decision to start the recording:

```
def step(self, action: WrapperActType) -> tuple[
    WrapperObsType, SupportsFloat, bool, bool, dict[str, tt.Any]
]:
    self._step_idx += 1
    obs, r, is_done, is_tr, extra = super().step(action)
    if self._is_storing:
        self._steps.append(EpisodeStep(self._prev_obs, int(action)))
        self._prev_obs = deepcopy(obs)

    if len(self._steps) >= self._steps_count:
        store_segment(self._store_path, self._step_idx, self._steps)
        self._is_storing = False
        self._steps.clear()
    elif random.random() <= self._start_prob:
        # start recording
        self._is_storing = True
        self._prev_obs = deepcopy(obs)
    return obs, r, is_done, is_tr, extra
```

By default, the probability of starting the recording is small (`START_PROB = 0.00005`, which is a 0.005% chance), but because of the large number of steps we're doing during the training, we still have plenty of segments to label. For example, after 12M environment steps (about 5 hours of training), the database contains 2,500 recorded segments, which occupy 12GB of disk.

The method `step()` uses the function `store_segment()` to store the list of `EpisodeStep` objects, and it is just the `pickle.dumps()` call for the list of steps:

```
def store_segment(root_path: pathlib.Path, step_idx: int, steps: tt.List[EpisodeStep]):  
    out_path = root_path / f"{step_idx:08d}.dat"  
    dat = pickle.dumps(steps)  
    out_path.write_bytes(dat)  
    print(f"Stored {out_path}")
```

Before we get to the training results, I need to mention one small but important detail about the wrapper's usage. To make the labeling easier, the observations we store in the DB are taken *before* the standard Atari wrappers. This increases the size of the data we have to store, but human labelers will see the original colorful Atari screen in the original resolution (160×192) instead of a downscaled picture in shades of gray.

To achieve that, the wrapper is applied right after the original Gymnasium environment before the Atari wrappers. The following is the relevant piece of code in the `01_a2c.py` module:

```
def make_env() -> gym.Env:  
    e = gym.make("SeaquestNoFrameskip-v4")  
    if reward_path is not None:  
        p = pathlib.Path(reward_path)  
        e = rlhf.RewardModelWrapper(e, p, dev=dev, metrics_queue=metrics_queue)  
    if db_path is not None:  
        p = pathlib.Path(db_path)  
        p.mkdir(parents=True, exist_ok=True)  
        e = rlhf.EpisodeRecorderWrapper(e, p, env_idx=env_idx)  
    e = ptan.common.wrappers.wrap_dqn(e)  
    # add time limit after all wrappers  
    e = gym.wrappers.TimeLimit(e, TIME_LIMIT)  
    return e
```

The training process hyperparameters were taken from the paper (LR decrease schedule, network architecture, count of environments, etc). I let it train for 5 hours and 12M observations. The charts with testing results are shown in *Figure 19.2*.

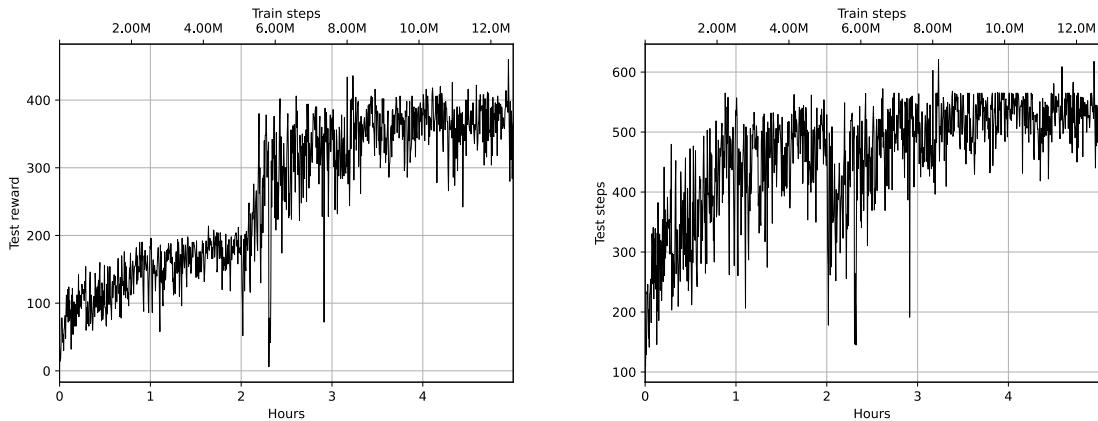


Figure 19.2: The reward (left) and steps (right) during the A2C training

The best model was able to reach the reward level of 460 (without reward clipping in the environment), which is good but is much worse than the results that could be achieved if you refill the oxygen from time to time.

The video of this model's gameplay is available at https://youtu.be/R_H3pXu-7cw. As you can see from the video, our agent mastered shooting the fish almost perfectly, but got stuck on the local optima of floating at the bottom (maybe because it is safer, as enemy submarines are not present there) and has no idea about the oxygen refilling.

You can record your own video from the model file using the tool `01_play.py`, which takes the model filename.

Labeling process

During the A2C training, we got 12GB of 2,500 random episode segments. Each segment contains 50 steps with screen observations and actions the agent took on every step. Now we are ready for the labeling process of the RLHF pipeline.

During the labeling, we need to randomly sample pairs of episode segments and show them to the human, asking the question “Which one is better?”. The answer should be stored for reward model training. Exactly this logic is implemented in `02_label_ui.py`.

The UI of the labeling process is implemented as a web application that uses the NiceGUI library (<https://nicegui.io/>). NiceGUI allows a modern web application UI to be implemented in Python and provides a rich set of interactive UI widgets, like buttons, lists, pop-up dialogs, etc. In principle, you don't need to know JavaScript and CSS (but it won't harm if you're familiar with them). If you have never used NiceGUI before, that's not a problem; you just need to install it with the following command in your Python environment:

```
pip install nicegui==1.4.26
```

To start the labeling UI (after installing the NiceGUI package), you need to specify the path to the DB with stored episode segments:

```
Chapter19$ ./02_label_ui.py -d db-v0
NiceGUI ready to go on http://localhost:8080, http://172.17.0.1:8080,
http://172.18.0.1:8080, and http://192.168.10.8:8080
```

The interface is available via HTTP (so, open it in your browser) and listens on port 8080 on all machine interfaces, which is convenient if you start it on a remote server (but you need to be aware of the possible risk of external access, as the labeling UI has no authentication and authorization at all). If you want to change the port or limit the scope to the specific network interface, just tweak `02_label_ui.py`. Let's look at a screenshot of the labelling interface:

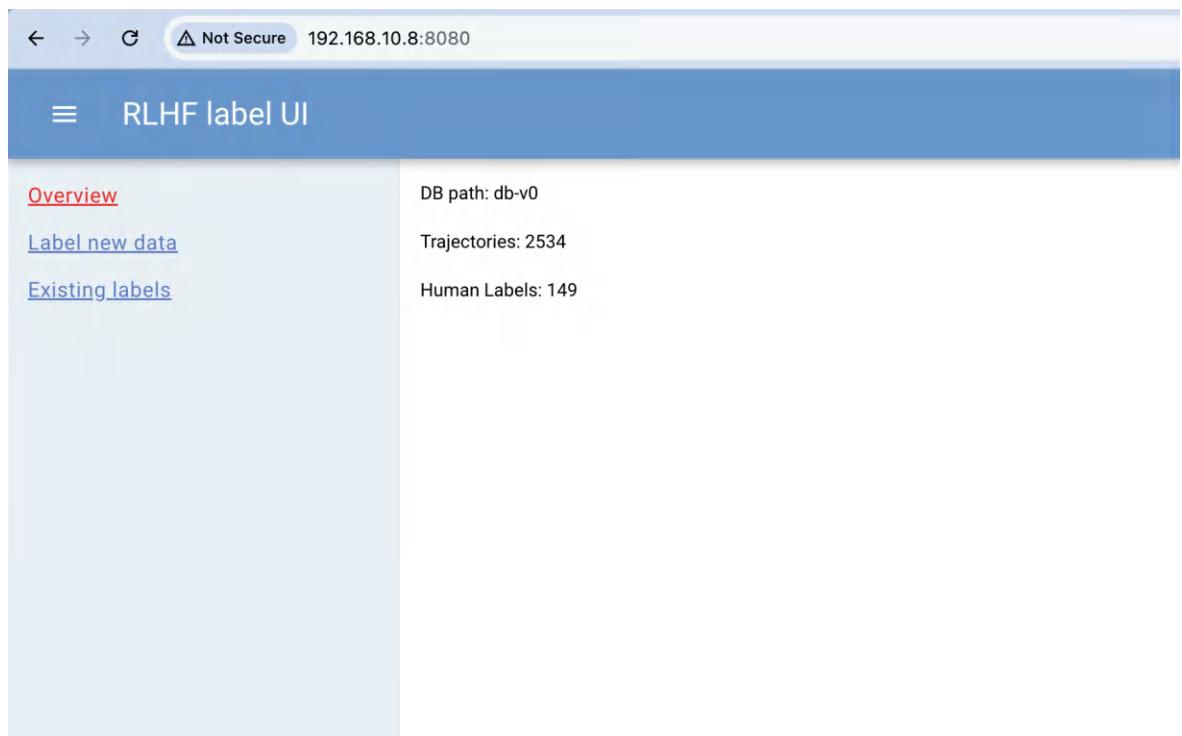


Figure 19.3: The labeling UI section with DB information

This interface is very basic: on the left, there are three links to different sections of the UI functionality:

- **Overview** shows the path to the database, the total count of segments it contains, and the amount of labels already created.
- **Label new data** samples random pairs of segments and allows you to label them.
- **Existing labels** shows all the labels and allows you to modify the labels if needed.

If needed, the list with links could be hidden or shown by clicking on the top-left button (with three horizontal lines). The most time has been spent on the **Label new data** section, shown in *Figure ??*:

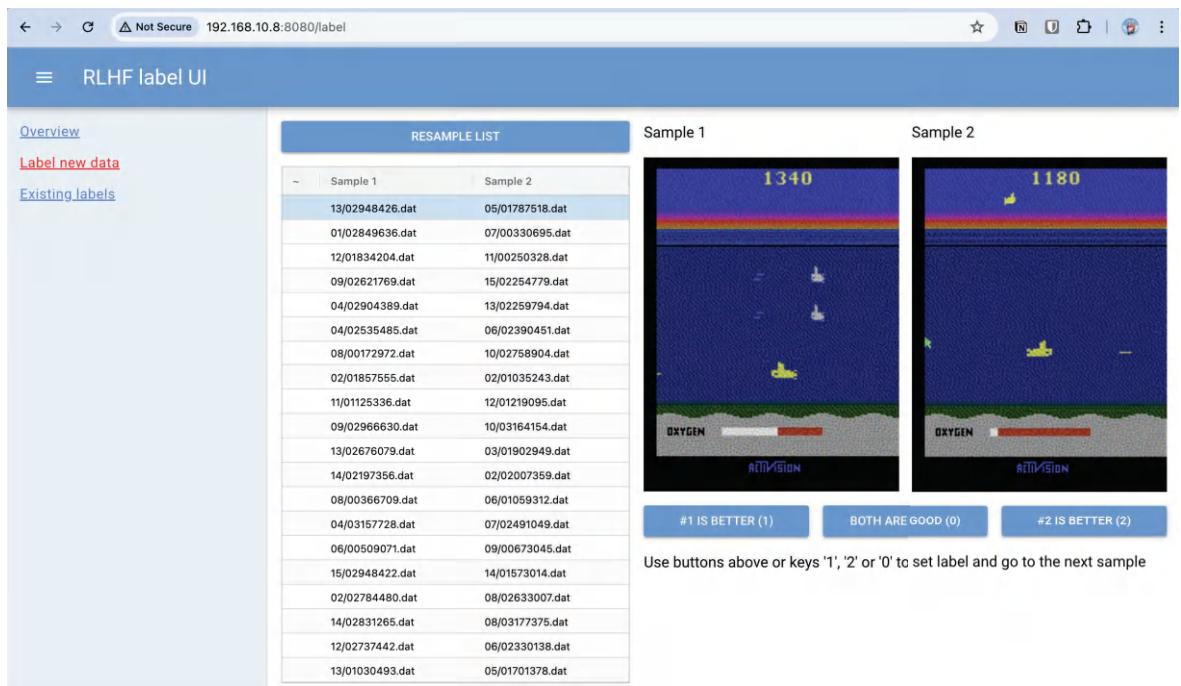


Figure 19.4: The interface for adding new labels (for better visualization, refer to <https://packt.link/gbp/9781835882702>)

Here, we have a list of 20 randomly sampled pairs of episode segments we can label. When the entry in the list is selected, the interface shows both segments (as animated GIFs generated by the code on the fly). The user can click one of three buttons to add the label:

- **#1 IS BETTER (1)**: Marks the first segment as preferred. Such entries will have $\mu_1 = 1.0$ and $\mu_2 = 0.0$ during the reward model training.
- **BOTH ARE GOOD (0)**: Marks both segments as equally good (or bad), assigning $\mu_1 = 0.5$ and $\mu_2 = 0.5$.
- **#2 IS BETTER (2)**: Marks the second segment as preferred ($\mu_1 = 0.0$ and $\mu_2 = 1.0$).

Instead of clicking the UI buttons, you can use the keyboard keys 0 (“both are good”), 1 (“the first is better”), or 2 (“the second is better”) to assign the label. Once the label is assigned, the UI automatically selects the next unlabeled entry in the list, so the labeling process could be done with the keyboard only. When you’re done with all the labels in the list, you can click the **RESAMPLE LIST** button to load 20 new samples for labeling.

After every label has been assigned (with UI button clicks or key presses), the labels are stored in the JSON file `labels.json` in the root of the DB directory. The file has a trivial JSON-line format where every line is an entry containing paths to both segments (relative to the DB root) and assigned labels:

```
Chapter19$ head db-v0/labels.json
{"sample1": "14/00023925.dat", "sample2": "10/00606788.dat", "label": 0}
{"sample1": "02/01966114.dat", "sample2": "10/01667833.dat", "label": 2}
{"sample1": "00/02432057.dat", "sample2": "06/01410909.dat", "label": 1}
{"sample1": "01/02293138.dat", "sample2": "11/00997214.dat", "label": 0}
 {"sample1": "10/00091149.dat", "sample2": "11/01262679.dat", "label": 2}
 {"sample1": "12/01394239.dat", "sample2": "04/01792088.dat", "label": 2}
 {"sample1": "10/01390371.dat", "sample2": "09/00077676.dat", "label": 0}
 {"sample1": "10/01390371.dat", "sample2": "09/00077676.dat", "label": 1}
 {"sample1": "12/02339611.dat", "sample2": "00/02755898.dat", "label": 2}
 {"sample1": "06/00301623.dat", "sample2": "06/00112361.dat", "label": 2}
```

If needed, existing labels could be reviewed using the **Existing labels** link (shown in *Figure 19.5*), which shows almost the same interface as **Label new data**, but instead of sampling 20 fresh pairs, it shows already labeled pairs.

Those pairs could be changed by clicking the buttons or using the keyboard shortcuts described earlier.

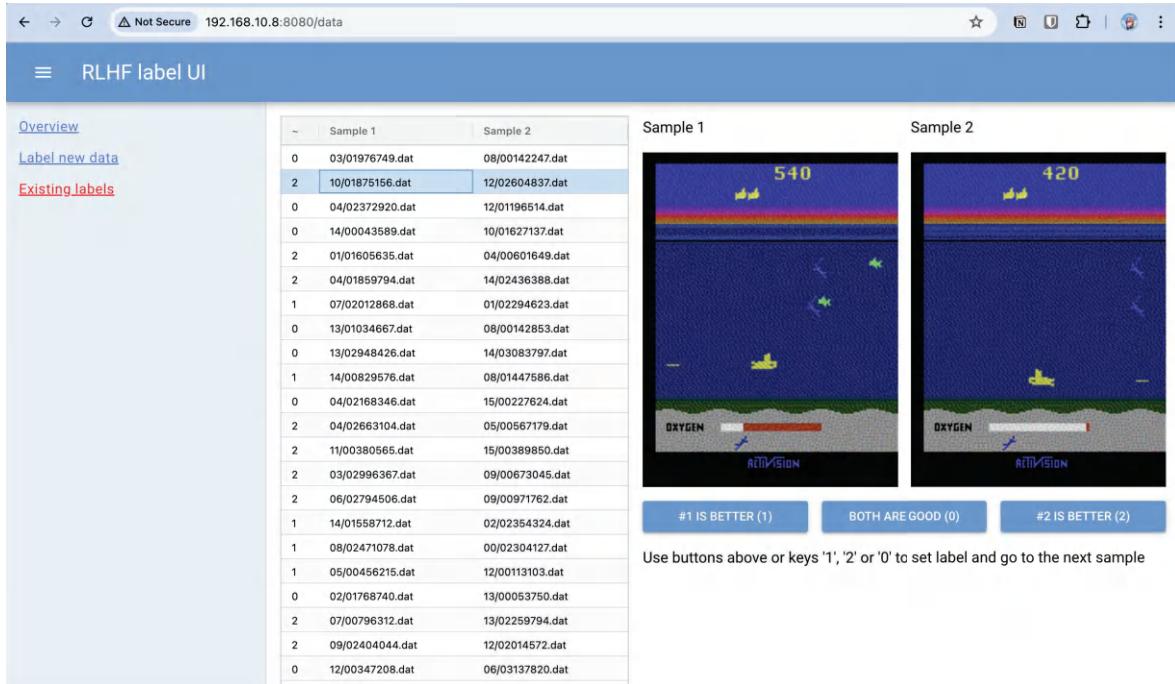


Figure 19.5: The interface for reviewing and editing old labels (for better visualization, refer to <https://packt.link/gbp/9781835882702>)

During my experiments, I did the first round of labeling 100 pairs paying most attention to the rare cases when the submarine was on the surface (marking them as good) and more frequent situations when oxygen was low (marking them as bad). In other situations, I prefer the segments where fish were properly hit. With some labels at hand, we're ready to go on to the next step: reward model training.

Reward model training

The reward model network has most of the structure taken from the paper, with the only difference in handling actions. In the paper, the authors do not specify how actions are taken into account besides stating “*For the reward predictor, we use 84×84 images as inputs (the same as the inputs to the policy), and stack 4 frames for a total $84 \times 84 \times 4$ input tensor.*” From that, I made an assumption that the reward model deducts actions “implicitly” from the dynamics between the frames. I haven’t tried this approach in my experiment and instead decided to show the actions to the network explicitly by concatenating one-hot encoding to the vectors obtained from the convolution layers. As an exercise, you can change my code to use the approach from the paper and compare the results.

The rest of the architecture and training parameters are the same as in the paper. Let's take a look at the reward model network code:

```
class RewardModel(nn.Module):
    def __init__(self, input_shape: tt.Tuple[int, ...], n_actions: int):
        super().__init__()

        self.conv = nn.Sequential(
            nn.Conv2d(input_shape[0], 16, kernel_size=7, stride=3),
            nn.BatchNorm2d(16),
            nn.Dropout(p=0.5),
            nn.LeakyReLU(),
            nn.Conv2d(16, 16, kernel_size=5, stride=2),
            nn.BatchNorm2d(16),
            nn.Dropout(p=0.5),
            nn.LeakyReLU(),
            nn.Conv2d(16, 16, kernel_size=3, stride=1),
            nn.BatchNorm2d(16),
            nn.Dropout(p=0.5),
            nn.LeakyReLU(),
            nn.Conv2d(16, 16, kernel_size=3, stride=1),
            nn.BatchNorm2d(16),
            nn.Dropout(p=0.5),
            nn.LeakyReLU(),
            nn.Flatten(),
        )
        size = self.conv(torch.zeros(1, *input_shape)).size()[-1]
        self.out = nn.Sequential(
            nn.Linear(size + n_actions, 64),
            nn.LeakyReLU(),
            nn.Linear(64, 1),
        )

    def forward(self, obs: torch.ByteTensor, acts: torch.Tensor) -> torch.Tensor:
        conv_out = self.conv(obs / 255)
        comb = torch.hstack((conv_out, acts))
        out = self.out(comb)
        return out
```

As you can see, convolution layers are combined with batch normalization, dropout, and the leaky ReLU activation function.

The training of the reward model is implemented in `03_reward_train.py` and has nothing complicated. We load labeled data from JSON files (you can pass several databases in the command line to use for the training), use 20% of the data for the testing, and compute the binary cross entropy objective, which is implemented in the `calc_loss()` function:

```
def calc_loss(model: rlhf.RewardModel, s1_obs: torch.ByteTensor,
              s1_acts: torch.Tensor, s2_obs: torch.ByteTensor,
              s2_acts: torch.Tensor, mu: torch.Tensor) -> torch.Tensor:
    batch_size, steps = s1_obs.size()[:2]

    s1_obs_flat = s1_obs.flatten(0, 1)
    s1_acts_flat = s1_acts.flatten(0, 1)
    r1_flat = model(s1_obs_flat, s1_acts_flat)
    r1 = r1_flat.view((batch_size, steps))
    R1 = torch.sum(r1, 1)

    s2_obs_flat = s2_obs.flatten(0, 1)
    s2_acts_flat = s2_acts.flatten(0, 1)
    r2_flat = model(s2_obs_flat, s2_acts_flat)
    r2 = r2_flat.view((batch_size, steps))
    R2 = torch.sum(r2, 1)
    R = torch.hstack((R1.unsqueeze(-1), R2.unsqueeze(-1)))
    loss_t = F.binary_cross_entropy_with_logits(R, mu)
    return loss_t
```

Initially, our observations and actions tensors had the following structure: $(batch, time, colors, height, width)$ for observations and $(batch, time, actions)$ for actions, where $time$ is the sequence's time dimension. More concretely, observation tensors had the size $64 \times 50 \times 3 \times 210 \times 160$ and actions had the size $64 \times 50 \times 18$.

As the first step in loss calculation, we flatten the first two dimensions, getting rid of the time dimension and applying the model to compute the reward value $\hat{r}(o, a)$. After that, we return the time dimension and sum along it according to the paper's formula we've already discussed. Then our computation of loss is the application of the `torch` function to compute the binary cross entropy.

On every epoch of the training, we compute the test loss (on 20% of the data) and save the reward model if the new loss is lower than the previous minimum of the test loss. If the train loss grows for four epochs in a row, we stop the training.

With the number of labels set in the previous section (a couple of hundred), the training is very quick – it takes about a dozen epochs and several minutes. The following is the example training process. The command-line argument `-o` specifies the directory name where the best model will be saved:

```
Chapter19$ ./03_reward_train.py --dev cuda -n v0-rw -o rw db-v0
Namespace(dev='cuda', name='v0-rw', out='rw', dbs=['db-v0'])
Loaded DB from db-v0 with 149 labels and 2534 paths
RewardModel(
  (conv): Sequential(
```

```
(0): Conv2d(3, 16, kernel_size=(7, 7), stride=(3, 3))
(1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(2): Dropout(p=0.5, inplace=False)
(3): LeakyReLU(negative_slope=0.01)
(4): Conv2d(16, 16, kernel_size=(5, 5), stride=(2, 2))
(5): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(6): Dropout(p=0.5, inplace=False)
(7): LeakyReLU(negative_slope=0.01)
(8): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1))
(9): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(10): Dropout(p=0.5, inplace=False)
(11): LeakyReLU(negative_slope=0.01)
(12): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1))
(13): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(14): Dropout(p=0.5, inplace=False)
(15): LeakyReLU(negative_slope=0.01)
(16): Flatten(start_dim=1, end_dim=-1)
)
(out): Sequential(
    (0): Linear(in_features=8978, out_features=64, bias=True)
    (1): LeakyReLU(negative_slope=0.01)
    (2): Linear(in_features=64, out_features=1, bias=True)
)
)
Epoch 0 done, train loss 0.131852, test loss 0.132976
Save model for 0.13298 test loss
Epoch 1 done, train loss 0.104426, test loss 0.354560
Epoch 2 done, train loss 0.159513, test loss 0.170160
Epoch 3 done, train loss 0.054362, test loss 0.066557
Save model for 0.06656 test loss
Epoch 4 done, train loss 0.046695, test loss 0.121662
Epoch 5 done, train loss 0.055446, test loss 0.064895
Save model for 0.06490 test loss
Epoch 6 done, train loss 0.024505, test loss 0.025308
Save model for 0.02531 test loss
Epoch 7 done, train loss 0.015864, test loss 0.045814
Epoch 8 done, train loss 0.024745, test loss 0.054631
Epoch 9 done, train loss 0.027670, test loss 0.054107
Epoch 10 done, train loss 0.025979, test loss 0.048673
Best test loss was less than current for 4 epoches, stop
```

Combining A2C with the reward model

Once the reward model is trained, we can finally try it for use in RL training. To do that, we use the same tool `01_a2c.py` but give it a couple of extra arguments:

- `-r` or `-reward`: This gives the path to the reward model to be loaded and used. With this option, we don't use the environment reward but, instead, use the model to get the reward from the observation and action we decided to take. This is implemented as an additional environment wrapper; we'll take a look shortly.
- `-m` or `-model`: This is the path to the actor model (stored on the previous A2C round of training) to be loaded. As I'm doing fine-tuning with RLHF instead of training with the reward model from scratch, the actor model is needed. In principle, you can try to use the reward model to train from scratch, but my experiments were not very successful.
- `-finetune`: This enables the fine-tuning mode: convolution layers are frozen and LR is decreased 10 times. Without those modifications, the actor very quickly unlearns all the prior knowledge and the reward drops to almost zero.

So, to use the reward model we've just trained, the command line will look like this:

```
./01_a2c.py -dev cuda -n v1 -r rw/reward-v0.dat -save save/v1 -m save/v0/model_rw=460-steps=580.dat  
-finetune
```

Before checking the experiment results, let's take a look at how the reward model is used in the RL training process. To minimize the changes needed, I implemented an environment wrapper, which is added between the original environment and Atari wrappers, because the reward model needs an unscaled full-color game image.

The code of the wrapper is in `lib/r1hf.py` and is called `RewardModelWrapper`. The constructor of the wrapper loads the model from the data file and assigns a couple of fields. According to the paper, the reward predicted by the reward model is normalized to have zero mean and unit variance, so to do the normalization, the wrapper maintains the last 100 rewards in `collections.deque`. Besides normalization, the wrapper can have a queue for metrics to be sent. The metrics contain information about normalization values and the real sum from the underlying environment:

```
class RewardModelWrapper(gym.Wrapper):
    KEY_REAL_REWARD_SUM = "real_reward_sum"
    KEY_REWARD_MU = "reward_mu"
    KEY_REWARD_STD = "reward_std"

    def __init__(self, env: gym.Env, model_path: pathlib.Path, dev: torch.device,
```

```

        reward_window: int = 100, metrics_queue: tt.Optional[queue.Queue] =
        None):
    super().__init__(env)
    self.device = dev
    assert isinstance(env.action_space, gym.spaces.Discrete)
    s = env.observation_space.shape
    self.total_actions = env.action_space.n
    self.model = RewardModel(
        input_shape=(s[2], s[0], s[1]), n_actions=self.total_actions)
    self.model.load_state_dict(torch.load(model_path,
                                         map_location=torch.device('cpu'),
                                         weights_only=True))
    self.model.eval()
    self.model.to(dev)
    self._prev_obs = None
    self._reward_window = collections.deque(maxlen=reward_window)
    self._real_reward_sum = 0.0
    self._metrics_queue = metrics_queue

```

In the `reset()` method, we just remember the observation and reset the reward counter:

```

def reset(self, *, seed: int | None = None, options: dict[str, tt.Any] | None = None) \
:
    -> tuple[WrapperObsType, dict[str, tt.Any]]:
    res = super().reset(seed=seed, options=options)
    self._prev_obs = deepcopy(res[0])
    self._real_reward_sum = 0.0
    return res

```

The main logic of the wrapper is in the `step()` function, but it is not very complicated: we apply the model to the observation and the action, normalize the reward, and return it instead of the real one. The model application is not very efficient from a performance perspective and could be optimized (as we have several environments working in parallel), but I decided to implement the simple version first, leaving optimizations as an exercise for you:

```

def step(self, action: WrapperActType) -> tuple[
    WrapperObsType, SupportsFloat, bool, bool, dict[str, tt.Any]
]:
    obs, r, is_done, is_tr, extra = super().step(action)
    self._real_reward_sum += r
    p_obs = np.moveaxis(self._prev_obs, (2, ), (0, ))
    p_obs_t = torch.as_tensor(p_obs).to(self.device)
    p_obs_t.unsqueeze_(0)

```

```

act = np.eye(self.total_actions)[[action]]
act_t = torch.as_tensor(act, dtype=torch.float32).to(self.device)
new_r_t = self.model(p_obs_t, act_t)
new_r = float(new_r_t.item())

# track reward for normalization
self._reward_window.append(new_r)
if len(self._reward_window) == self._reward_window maxlen:
    mu = np.mean(self._reward_window)
    std = np.std(self._reward_window)
    new_r -= mu
    new_r /= std
    self._metrics_queue.put((self.KEY_REWARD_MU, mu))
    self._metrics_queue.put((self.KEY_REWARD_STD, std))

if is_done or is_tr:
    self._metrics_queue.put((self.KEY_REAL_REWARD_SUM, self._real_reward_sum))
self._prev_obs = deepcopy(obs)
return obs, new_r, is_done, is_tr, extra

```

The rest of the training is the same. We just inject the new wrapper into the environment-creating function if the reward model file is given in the command line:

```

def make_env() -> gym.Env:
    e = gym.make("SeaquestNoFrameskip-v4")
    if reward_path is not None:
        p = pathlib.Path(reward_path)
        e = rlhf.RewardModelWrapper(e, p, dev=dev, metrics_queue=metrics_queue)
    if db_path is not None:
        p = pathlib.Path(db_path)
        p.mkdir(parents=True, exist_ok=True)
        e = rlhf.EpisodeRecorderWrapper(e, p, env_idx=env_idx)
    e = ptan.common.wrappers.wrap_dqn(e)
    # add time limit after all wrappers
    e = gym.wrappers.TimeLimit(e, TIME_LIMIT)
    return e

```

With this code, we can now combine the previous model with the labels we made before.

Fine-tuning with 100 labels

I ran the training with the best model from the basic A2C training, which, on testing, achieved a reward of 460 in 580 steps. In addition, I enabled sampling of episode segments into the new DB directory (v1 in this case), so the full command line was the following:

```
./01_a2c.py -dev cuda -n v1 -r rw/reward-v0.dat -save save/v1 -m save/v0/model_rw=460-steps=580.dat  
-finetune -db-path v1
```

This model started to overfit quite quickly and after 2M steps (3 hours), I stopped the training. *Figure 19.6* shows the test results (reward and count of steps):

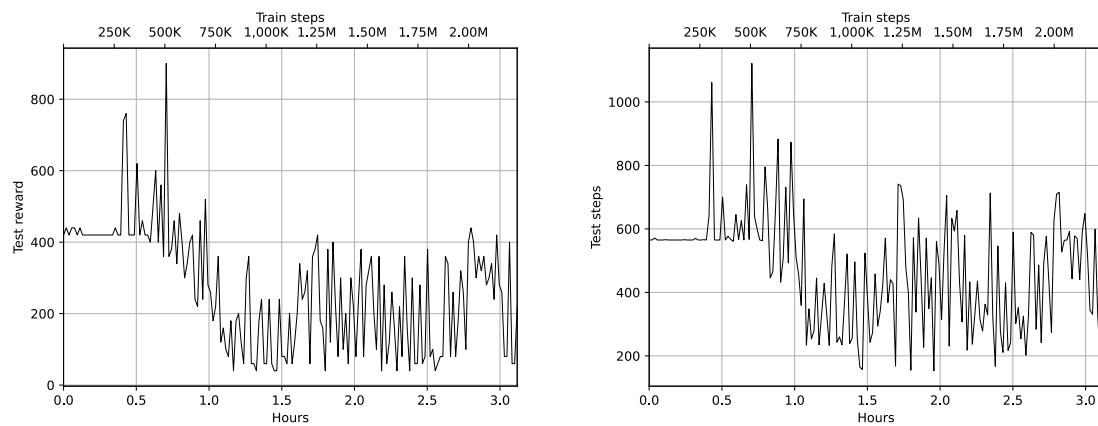


Figure 19.6: Test reward (left) and steps (right) during the fine-tuning

Figure 19.7 shows the training reward (predicted by the model) and the total loss:

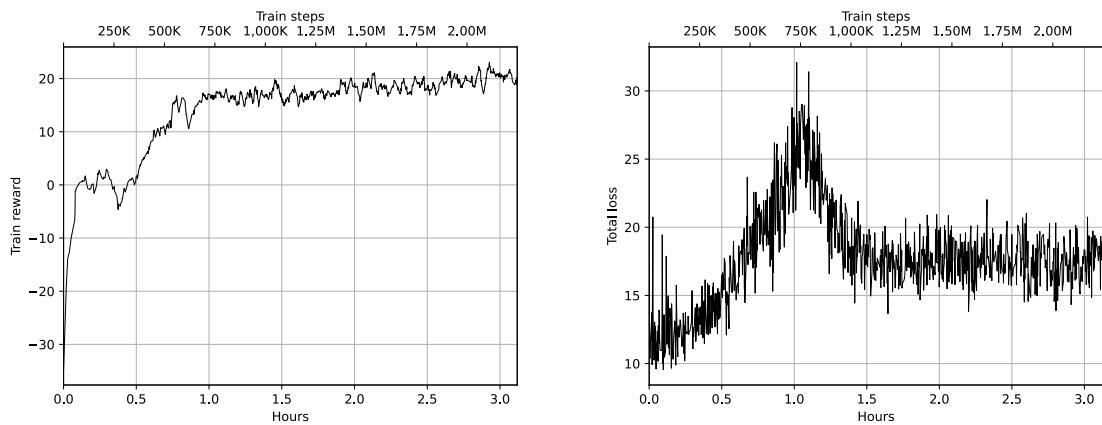


Figure 19.7: Training reward (left) and total loss (right) during the fine-tuning

The best model was stored at the 500K training step and it was able to get a reward of 900 in 1,120 steps. In comparison to the original model, this is quite an improvement.

A video recording of this model is available here: <https://youtu.be/LnPwuyVrj9g>. From the gameplay, we see that the agent learned how to refill the oxygen and is now spending some time in the middle of the screen. I also had the impression that it picked divers more intentionally (but I haven't done specific labeling for this behavior). So, overall, the method works and it is quite impressive that we can teach the agent something new with just 100 labels.

Let's try to improve the model further with more labeling.

The second round of the experiment

On the second round, I did more labeling: 50 pairs from the v0 DB and 50 pairs from segments stored during the fine-tuning (v1 DB). The database generated during the fine-tuning (v1) contains many more segments with the submarine floating on the surface, which confirms that our pipeline is working as expected. During the labeling, I also put more emphasis on oxygen refill segments.

After labeling, I retrained the reward model, which only took several minutes. Then, fine-tuning of the best v1 model (with a reward of 900 and 1,120 steps) was performed using the reward model.

Figure 19.8 and Figure 19.9 contain charts with test results, training the reward, and the loss:

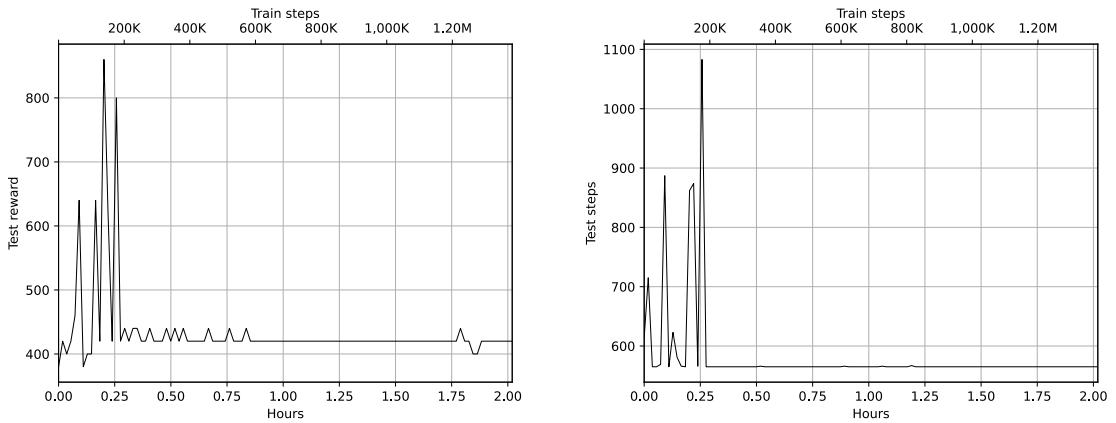


Figure 19.8: Test reward (left) and steps (right) during the fine-tuning

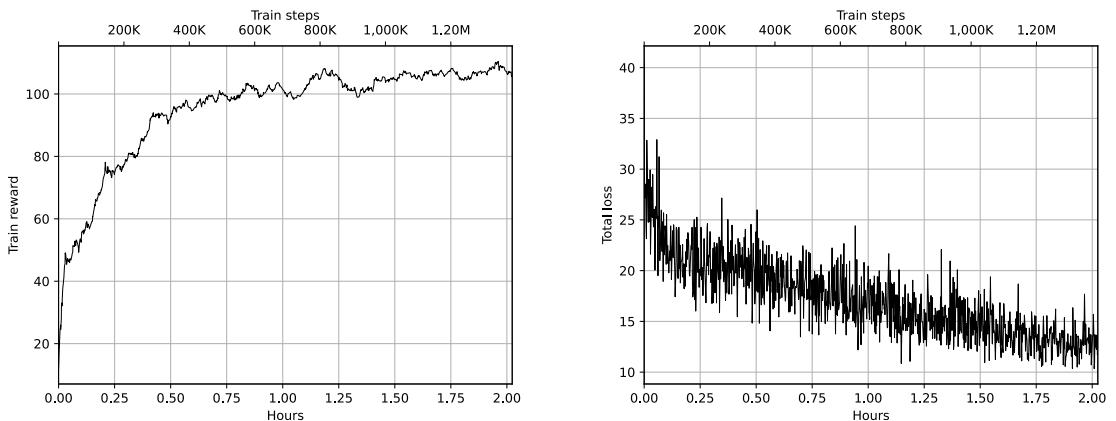


Figure 19.9: Training reward (left) and total loss (right) during the fine-tuning

After 1.5M steps (2 hours), the training got stuck, but the best model wasn't better than the best model of v1: the best model got a reward of 860 in 1,084 steps.

The third round of the experiment

Here, I paid more attention during the labeling, trying to prioritize not just oxygen refill, but also better fish shooting and diver pickup. Unfortunately, 100 pairs gave just a couple of examples of divers, so more labeling is needed to teach the agent this behavior.

Regarding the divers, it might be that the agent doesn't pick them up just because they are very hard to distinguish from the background, so on a grayscale image, they are invisible. To fix that, we can tweak the contrast in our Atari wrappers.

After the reward model retraining, A2C fine-tuning was started. I also ran it for almost 2M steps for 3 hours and the results were interesting. At the end of the training (check *Figure 19.10* and *Figure 19.11*), the boat during the testing reached 5,000 steps (which is the limit I set in the environment), but the score was fairly low. Most likely, the submarine just stayed on the surface, which is very safe, but not what we want – this could be because of labeled samples. Strangely, when I tried to record the video of those later models, their behavior was different and the number of steps was much lower, which could be an indication of some testing bug.

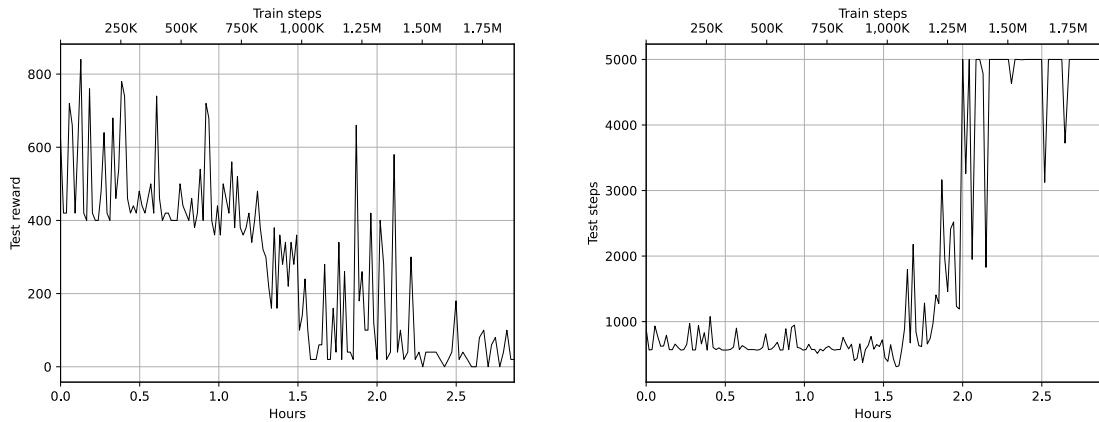


Figure 19.10: Test reward (left) and steps (right) during the fine-tuning

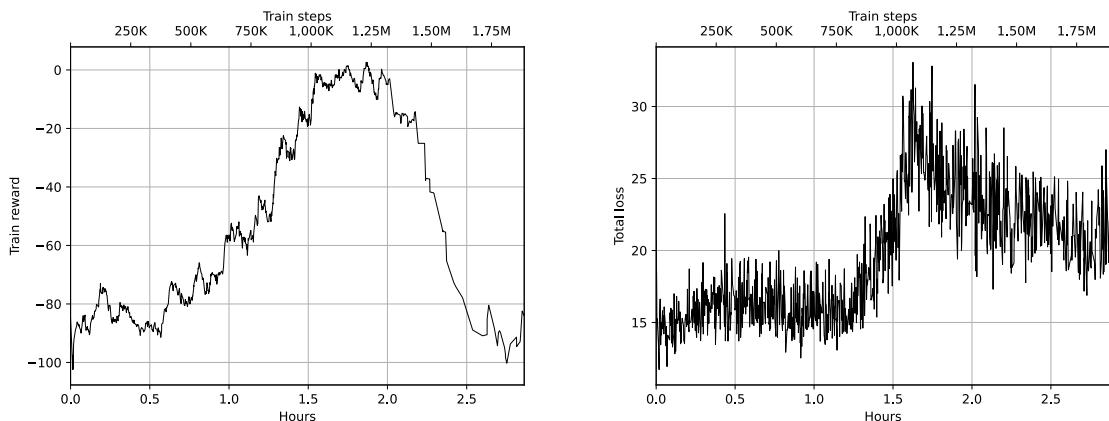


Figure 19.11: Training reward (left) and total loss (right) during the fine-tuning

Before the overfitting, the training generated several policies that were better than the v2 models. For example, in this recording, the agent refilled the oxygen twice and got a score of 1,820 during the 1,613 steps: https://youtu.be/DVe_9b3gdxU.

Overall results

In the following table, I have summarized the information about the experiment rounds and the results we got.

Step	Labels	Reward	Steps	Video
Initial	None	460	580	https://youtu.be/R_H3pXu-7cw
v1	100	900	1120	https://youtu.be/LnPwuyVrj9g
v2	200	860	1083	
v3	300	1820	1613	https://youtu.be/DVe_9b3gdxU

Table 19.1: Summary of experiment rounds

As you can see, with just 300 labels, we were able to increase the scoring by almost 4 times. As an exercise, you can try to teach the agent to pick up divers, which might result in a much better score if done properly.

Another experiment that might be worth doing is to fine-tune the original v0 model, instead of the best models from the previous step. It might lead to better results, as the training has more time before overfitting.

Summary

In this chapter, we've taken a look at the recent addition of RLHF to the RL toolbox. This method, at the core of the LLM training pipeline, allows you to increase the quality of models. In the chapter, we implemented RLHF and applied it to the SeaQuest Atari game, which should have illustrated to you how this method could be used in RL pipelines for model improvement.

In the next chapter, we'll discuss a different family of RL methods: AlphaGo, AlphaZero, and MuZero.

20

AlphaGo Zero and MuZero

Model-based methods allow us to decrease the amount of communication with the environment by building a model of the environment and using it during training. In this chapter, we take a look at model-based methods by exploring cases where we have a model of the environment, but this environment is being used by two competing parties. This situation is very common in board games, where the rules of the game are fixed and the full position is observable, but we have an opponent who has the primary goal of preventing us from winning the game.

A few years ago, DeepMind proposed a very elegant approach to solving such problems. No prior domain knowledge is required, but the agent improves its policy only via self-play. This method is called **AlphaGo Zero** and was introduced in 2017. Later, in 2020, they extended this method by removing the requirement for an environment model, which allowed it to apply to a much wider range of RL problems (including Atari games). The method is called MuZero and we will also look at this in detail. As you'll see in this chapter, MuZero is more general than AlphaGo Zero, which comes with the price of more networks to be trained and might lead to longer training times and worse results. From that perspective, we'll discuss both methods in detail, as AlphaGo Zero may be more applicable in some situations.

In this chapter, we will:

- Discuss the structure of the AlphaGo Zero method
- Implement the method for playing Connect 4
- Implement MuZero and compare it to AlphaGo Zero

Comparing model-based and model-free methods

In *Chapter 4*, we saw several different ways in which we can classify RL methods. We distinguished three main categories:

- Value-based and policy-based
- On-policy and off-policy
- Model-free and model-based

So far, we have covered enough examples of methods of both types in the first and second categories, but all the methods that we have covered so far have been 100% model-free. However, this doesn't mean that model-free methods are more important or better than their model-based counterparts. Historically, due to their **sample efficiency**, model-based methods have been used in the robotics field and for other industrial controls. This has also happened because of the cost of the hardware and the physical limitations of samples that can be obtained from a real robot. Robots with a large degree of freedom are not widely accessible, so RL researchers are more focused on computer games and other environments where samples are relatively cheap. However, ideas from robotics are infiltrating RL, so, who knows, maybe the model-based methods will become more of a focus quite soon. To begin, let's discuss the difference between the model-free approach that we have used in the book and model-based methods, including their strong and weak points and where they might be applicable.

In the names of both classes, "model" means the model of the environment, which could have various forms, for example, providing us with a new state and reward from the current state and action. All the methods covered so far put zero effort into predicting, understanding, or simulating the environment. What we were interested in was proper behavior (in terms of the final reward), specified directly (a policy) or indirectly (a value) given the observation. The source of the observations and reward was the environment itself, which in some cases could be very slow and inefficient.

In a model-based approach, we're trying to learn the model of the environment to reduce this "real environment" dependency. At a high level, the model is some kind of black box that approximates the real environment that we talked about in *Chapter 1*. If we have an accurate environment model, our agent can produce any number of trajectories that it needs simply by using this model instead of executing the actions in the real world.

To some degree, the common playground of RL research is also just models of the real world; for example, MuJoCo and PyBullet are simulators of physics used so we don't need to build real robots with real actuators, sensors, and cameras to train our agents. The story is the same with Atari games or TORCS (The Open Racing Car Simulator): we use computer programs that model some processes, and these models can be executed quickly and cheaply.

Even our CartPole example is a simplified approximation of a real cart with a stick attached. (By the way, in PyBullet and MuJoCo, there are more realistic CartPole versions with 3D actions and more accurate simulation.)

There are two motivations for using the model-based approach as opposed to model-free:

- The first and the most important one is sample efficiency caused by less dependency on the real environment. Ideally, by having an accurate model, we can avoid touching the real world and use only the trained model. In real applications, it is almost never possible to have a precise model of the environment, but even an imperfect model can significantly reduce the number of samples needed. For example, in real life, you don't need an absolutely precise mental picture of some action (such as tying shoelaces or crossing the road), but this picture helps you plan and predict the outcome.
- The second reason for a model-based approach is the **transferability** of the environment model across goals. If you have a good model for a robot manipulator, you can use it for a wide variety of goals without retraining everything from scratch.

There are a lot of details in this class of methods, but the aim of this chapter is to give you an overview and take a closer look at the model-based approach applied to board games.

Model-based methods for board games

Most board games provide a setup that is different from an arcade scenario. The Atari game suite assumes that one player is making decisions in some environment with complex dynamics. By generalizing and learning from the outcome of their actions, the player improves their skills, increasing their final score. In a board game setup, however, the rules of the game are usually quite simple and compact. What makes the game complicated is the number of different positions on the board and the presence of an opponent with an unknown strategy who tries to win the game.

With board games, the ability to observe the game state and the presence of explicit rules opens up the possibility of analyzing the current position, which isn't the case for Atari. This analysis means taking the current state of the game, evaluating all the possible moves that we can make, and then choosing the best move as our action. To be able to evaluate all the moves, we need some kind of model of the game, capturing the game rules.

The simplest approach to evaluation is to iterate over the possible actions and recursively evaluate the position after the action has been taken. Eventually, this process will lead us to the final position, when no more moves are possible. By propagating the game result back, we can estimate the expected value of any action in any position.

One possible variation of this method is called **minimax**, which is when we are trying to make the strongest move, but our opponent is trying to make the worst move for us, so we are iteratively minimizing and maximizing the final game objective of walking down the tree of game states (which will be described in detail later).

If the number of different positions is small enough to be analyzed entirely, like in the tic-tac-toe game (which has only 138 terminal states), it's not a problem to walk down this game tree from any state that we have and figure out the best move to make. Unfortunately, this brute-force approach doesn't work even for medium-complexity games, as the number of configurations grows exponentially. For example, in the game of draughts (also known as checkers), the total game tree has $5 \cdot 10^{20}$ nodes, which is quite a challenge even for modern hardware. In the case of more complex games, like Chess or Go, this number is much larger, so it's just not possible to analyze all the positions reachable from every state. To handle this, usually some kind of approximation is used, where we analyze the tree up to some depth. With a combination of careful search and stop criteria, called **tree pruning**, and the smart predefined evaluation of positions, we can make a computer program that plays complex games at a fairly good level.

The AlphaGo Zero method

In late 2017, DeepMind published an article titled *Mastering the game of Go without human knowledge* in the journal *Nature* by Silver et al. [SSa17] presenting a novel approach called AlphaGo Zero, which was able to achieve a superhuman level of playing complex games, like Go and chess, without any prior knowledge except the rules. The agent was able to improve its policy by constantly playing against itself and reflecting on the outcomes. No large game databases, handmade features, or pretrained models were needed. Another nice property of the method is its simplicity and elegance.

In the example of this chapter, we will try to understand and implement this approach for the game Connect 4 (also known as “four in a row” or “four in a line”) to evaluate it ourselves.

First, we will discuss the structure of the method. The whole system contains several parts that need to be understood before we can implement them.

Overview

At a high level, the method consists of three components, all of which will be explained in detail later, so don't worry if something is not completely clear from this section:

- We constantly traverse the game tree using the **Monte Carlo tree search (MCTS)** algorithm, the core idea of which is to semi-randomly walk down the game states, expanding them and gathering statistics about the frequency of moves and underlying game outcomes.

As the game tree is huge, both in terms of the depth and width, we don't try to build the full tree; we just randomly sample its most promising paths (that's the source of the method's name).

- At every moment, we have the current *best player*, which is the model used to generate the data via **self-play** (this concept will be discussed in detail later, but for now it is enough for you to know that it refers to the usage of the same model against itself). Initially, this model has random weights, so it makes moves randomly, like a four-year-old learning how chess pieces move. However, over time, we replace this best player with better variations of it, which generate more and more meaningful and sophisticated game scenarios. Self-play means that the same *current best* model is used on both sides of the board. This might not look very useful, as having the same model play against itself has an approximately 50% chance outcome, but that's actually what we need: samples of the games where our best model can demonstrate its best skills. The analogy is simple: it's usually not very interesting to watch a match between the outsider and the leader; the leader will win easily. What is much more fun and intriguing to see is when players of roughly equal skill compete. That's why the final in any championship attracts much more attention than the preceding matches: both teams or players in the final usually excel in the game, so they will need to play their best game to win.
- The third component in the method is the **training** process of the *apprentice* model, which is trained on the data gathered by the best model during self-play. This model can be likened to a kid sitting and constantly analyzing the chess games played by two adults. Periodically, we play several matches between this trained model and our current best model. When the trainee is able to beat the best model in the majority of games, we announce the trained model as the new best model and the process continues.

Despite the simplicity and even naïvety of this, AlphaGo Zero was able to beat all the previous AlphaGo versions and became the best Go player in the world, without any prior knowledge except the rules. After the paper by Silver et al. [SSa17] was published, DeepMind adapted the method for chess and published the paper called *Mastering chess and shogi by self-play with a general reinforcement learning algorithm* [Sil+17], where the model trained from scratch beat Stockfish, which was the best chess program at the time and took more than a decade for human experts to develop.

Now, let's take a look at all three components of the method in detail.

MCTS

To understand what MCTS does, let's consider a simple subtree of the tic-tac-toe game, as shown in *Figure 20.1*. At the beginning, the game field is empty and the cross player (X) needs to choose where to move.

There are nine different options for the first move, so our root state has nine different branches leading to the corresponding states.

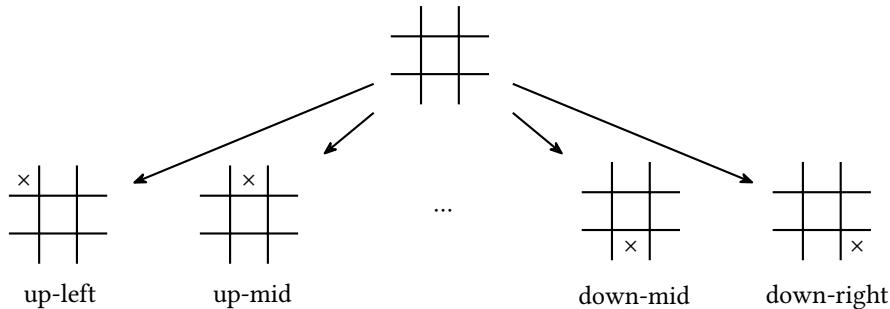


Figure 20.1: The game tree of tic-tac-toe

The number of possible actions at any game state is called the **branching factor**, and it shows the bushiness of the game tree. In general, this is not constant and may vary, as some moves are not always doable. In the case of tic-tac-toe, the number of available actions could vary from nine at the beginning of the game to zero at the leaf nodes. The branching factor allows us to estimate how quickly the game tree grows, as every available action leads to another set of actions that could be taken.

For our example, after the cross player has made their move, the nought (0) has eight alternatives at every nine positions, which makes 9×8 total positions at the second level of the tree. The total number of nodes in the tree can be up to $9! = 362880$, but the actual number is less, as not all the games could be played to the maximum depth.

Tic-tac-toe is tiny, but if we consider larger games and, for example, think about the number of first moves that white could make at the beginning of a chess game (which is 20) or the number of spots that the white stone could be placed at in Go (361 in total for a 19×19 game field), the number of game positions in the complete tree quickly becomes enormous. With every new level, the number of states is multiplied by the average number of actions that we can perform on the previous level.

To deal with this combinatorial explosion, random sampling comes into play. In a general MCTS, we perform many iterations of depth-first search, starting at the current game state and either selecting the actions randomly or with some strategy, which should include enough randomness in its decisions. Every search is continued until the end state of the game, and then it is followed by updating the weights of the visited tree branches according to the game's outcome. This process is similar to the value iteration method, when we played the episodes and the final step of the episode influenced the value estimation of all the previous steps.

This is a general MCTS, and there are many variants of this method related to expansion strategy, branch selection policy, and other details.

In AlphaGo Zero, a variant of MCTS is used. For every edge (representing the move from some position), this set of statistics is stored:

- A prior probability, $P(s, a)$, of the edge
- A visit count, $N(s, a)$
- An action value, $Q(s, a)$

Each search starts from the root state following the most promising actions, selected using the utility value, $U(s, a)$, proportional to

$$Q(s, a) + \frac{P(s, a)}{1 + N(s, a)}$$

Randomness is added to the selection process to ensure enough exploration of the game tree. Every search could end up with two outcomes: the end state of the game is reached, or we face a state that hasn't been explored yet (in other words, has no known values). In the latter case, the policy **neural network (NN)** is used to obtain the prior probabilities and the value of the state estimation, and the new tree node with $N(s, a) \leftarrow 0$, $P(s, a) \leftarrow p_{\text{net}}$ (which is a probability of the move returned by the network) and $Q(s, a) \leftarrow 0$ is created. Besides the prior probability of the actions, the network returns the estimation of the game's outcome (or the value of the state) as seen from the current player.

Once we have obtained the value (by reaching the final game state or by expanding the node using the NN), a process called the backup of value is performed. During the process, we traverse the game path and update statistics for every visited intermediate node; in particular, the visit count, $N(s, a)$, is incremented by one and $Q(s, a)$ is updated to include the game's outcome from the perspective of the current state. As two players are exchanging moves, the final game outcome changes the sign in every backup step.

This search process is performed several times (in AlphaGo Zero's case, 1,000-2,000 searches are performed), gathering enough statistics about the action to use the $N(s, a)$ counter as an action probability to be taken in the root node.

Self-play

In AlphaGo Zero, the NN is used to approximate the prior probabilities of the actions and evaluate the position, which is very similar to the **advantage actor-critic (A2C)** two-headed setup. In the input of the network, we pass the current game position (augmented with several previous positions) and return two values:

- The policy head returns the probability distribution over the actions.

- The value head estimates the game outcome as seen from the player’s perspective. This value is undiscounted, as moves in Go are deterministic. Of course, if you have stochasticity in a game, like in backgammon, some discounting should be used.

As has already been described, we’re maintaining the current best network, which constantly self-plays to gather the training data for our apprentice network. Every step in each self-play game starts with several MCTSs from the current position to gather enough statistics about the game subtree to select the best action. The selection depends on the move and our settings. For self-play games, which are supposed to produce enough variance in the training data, the first moves are selected in a stochastic way. However, after some number of steps (which is a hyperparameter in the method), action selection becomes deterministic, and we select the action with the largest visit counter, $N(s, a)$. In evaluation games (when we check the network being trained versus the current best model), all the steps are deterministic and selected solely on the largest visit counter.

Once the self-play game has been finished and the final outcome has become known, every step of the game is added to the training dataset, which is a list of tuples (s_t, π_t, r_t) , where s_t is the game state, π_t is the action probabilities calculated from MCTS sampling, and r_t is the game’s outcome from the perspective of the player at step t .

Training and evaluation

The self-play process between two clones of the current best network provides us with a stream of training data consisting of states, action probabilities, and position values obtained from the self-play games. With this at hand, for training we sample mini-batches from the replay buffer of training examples and minimize the **mean squared error (MSE)** between the value head prediction and the actual position value, as well as the cross-entropy loss between predicted probabilities and sampled probabilities, π .

As mentioned earlier, once in several training steps the trained network is evaluated, which consists of playing several games between the current best and trained networks. Once the trained network becomes significantly better than the current best network, we copy the trained network into the best network and continue the process.

Connect 4 with AlphaGo Zero

To see the method in action, let’s implement AlphaGo Zero for a relatively simple game, Connect 4. The game is for two players with a field size of 6×7 . Each player has disks of a certain color, which they drop in turn into any of the seven columns. The disks fall to the bottom, stacking vertically. The game objective is to be the first to form a horizontal, vertical, or diagonal line of four disks of the same color. To illustrate the game, two positions are shown in *Figure 20.2*.

In the first situation, the first player has just won, while in the second, the second player is going to form a group.

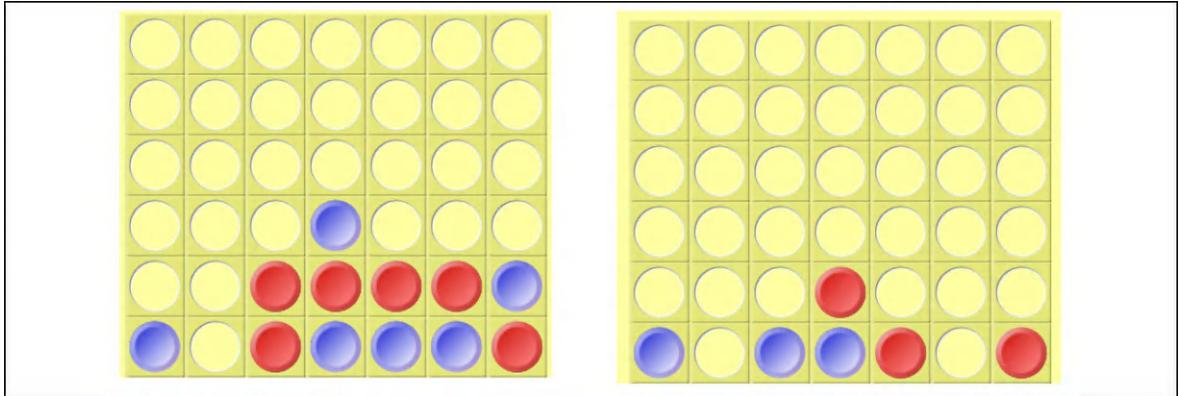


Figure 20.2: Two game positions in Connect 4

Despite its simplicity, this game has $\approx 4.5 \cdot 10^{12}$ different game states, which is challenging for computers to solve with brute force. This example consists of several tools and library modules:

- `Chapter20/lib/game.py`: A low-level game representation that contains functions for making moves, encoding and decoding the game state, and other game-related utilities.
- `Chapter20/lib/mcts.py`: The MCTS implementation that allows GPU-accelerated expansion of leaves and node backup. The central class here is also responsible for keeping the game node statistics, which are reused between the searches.
- `Chapter20/lib/model.py`: The NN and other model-related functions, such as the conversion between game states and the model's input and the playing of a single game.
- `Chapter20/train.py`: The main training utility that glues everything together and produces the model checkpoints of the new best networks.
- `Chapter20/play.py`: The tool that organizes the automated tournament between the model checkpoints. This accepts several model files and plays the given number of games against each other to form a leaderboard.
- `Chapter20/telegram-bot.py`: The bot for the Telegram chat platform that allows the user to play against any model file and keep a record of the statistics. This bot was used for human verification of the example's results.

Now let's discuss the core of our game – the game model.

The game model

The whole approach is based on our ability to predict the outcome of our actions; in other words, we need to be able to get the resulting game state after we execute a move. This is a much stronger requirement than we had in the Atari environments and Gym in general, where you can't specify a state that you want to act from. So, we need a model of the game that encapsulates the game's rules and dynamics. Luckily, most board games have a simple and compact set of rules, which makes the model implementation a straightforward task.

In our case, the full game state of Connect 4 is represented by the state of the 6×7 game field cells and the indicator of who is going to move. What is important for our example is to make the game state representation occupy as little memory as possible, but still allow it to work efficiently. The memory requirement is dictated by the necessity of storing large numbers of game states during the MCTS. As our game tree is huge, the more nodes we're able to keep during the MCTS, the better our final approximation of move probabilities will be. So, potentially, we'd like to be able to keep millions, or maybe even billions, of game states in memory.

With this in mind, the compactness of the game state representation could have a huge impact on memory requirements and the performance of our training process. However, the game state representation has to be convenient to work with, for example, when checking the board for a winning position, making a move, or finding all the valid moves from some state.

To keep this balance, two representations of the game field were implemented in `Chapter20/lib/game.py`:

- The first encoded form is very memory-efficient and takes only 63 bits to encode the full field, which makes it extremely fast and lightweight, as it fits in a machine word on 64-bit architectures.
- Another decoded game field representation has the form of a list with a length of 7, where each entry is a list of integers representing the disks in a particular column. This form takes much more memory, but it is convenient to work with.

I'm not going to show the full code of `Chapter20/lib/game.py`, but if you need it, it's available in the repository. Here, let's just take a look at the list of the constants and functions that it provides:

```
GAME_ROWS = 6
GAME_COLS = 7
BITS_IN_LEN = 3
PLAYER_BLACK = 1
PLAYER_WHITE = 0
COUNT_TO_WIN = 4

INITIAL_STATE = encode_lists([[[]] * GAME_COLS])
```

The first two constants in the preceding code define the dimensionality of the game field and are used everywhere in the code, so you can try to change them and experiment with larger or smaller game variants. The `BITS_IN_LEN` value is used in state encoding functions and specifies how many bits are used to encode the height of the column (the number of disks present). In the 6×7 game, we could have up to six disks in every column, so three bits is enough to keep values from zero to seven. If you change the number of rows, you will need to adjust `BITS_IN_LEN` accordingly.

The `PLAYER_BLACK` and `PLAYER_WHITE` values define the values used in the *decoded* game representation and, finally, `COUNT_TO_WIN` sets the length of the group that needs to be formed to win the game. So, in theory, you can try to experiment with the code and train the agent for, say, five in a row on a 20×40 field by just changing four numbers in `game.py`.

The `INITIAL_STATE` value contains the encoded representation for an initial game state, which has `GAME_COLS` empty lists.

The rest of the code is made up of functions. Some of them are used internally, but some make an interface of the game used everywhere in the example. Let's list them quickly:

- `encode_lists(state_lists)`: This converts from a decoded to an encoded representation of the game state. The argument has to be a list of `GAME_COLS` lists, with the contents of the column specified in bottom-to-top order. In other words, to drop a new disk at the top of the stack, we just need to append it to the corresponding list. The result of the function is an integer with 63 bits representing the game state.
- `decode_binary(state_int)`: This converts the integer representation of the field back into the list form.
- `possible_moves(state_int)`: This returns a list with indices of columns that can be used for moving from the given encoded game state. The columns are numbered from zero to six, left to right.
- `move(state_int, col, player)`: The central function of the file, which provides game dynamics combined with a win/lose check. In arguments, it accepts the game state in the encoded form, the column to place the disk in, and the index of the player that moves. The column index has to be valid (that is, be present in the result of `possible_moves(state_int)`), otherwise an exception will be raised. The function returns a tuple with two elements: a new game state in the encoded form after the move has been performed and a Boolean indicating the move leading to the win of the player. As a player can win only after their move, a single Boolean is enough. Of course, there is a chance of getting a draw state (when nobody has won, but there are no possible moves remaining). Such situations have to be checked by calling the `possible_moves()` function after the `move()` function.
- `render(state_int)`: This returns a list of strings representing the field's state. This function is used in the Telegram bot to send the field state to the user.

Implementing MCTS

MCTS is implemented in [Chapter20/lib/mcts.py](#) and represented by a single class, `MCTS`, which is responsible for performing a batch of MCTSs and keeping the statistics gathered during it. The code is not very large, but it still has several tricky pieces, so let's check it in detail.

The constructor has no arguments except the `c_puct` constant, which is used in the node selection process. Silver et al. [SSa17] mentioned that it could be tweaked to increase exploration, but I'm not redefining it anywhere and haven't experimented with it. The body of the constructor creates an empty container to keep statistics about the states:

```
class MCTS:
    def __init__(self, c_puct: float = 1.0):
        self.c_puct = c_puct
        # count of visits, state_int -> [N(s, a)]
        self.visit_count: tt.Dict[int, tt.List[int]] = {}
        # total value of the state's act, state_int -> [W(s, a)]
        self.value: tt.Dict[int, tt.List[float]] = {}
        # average value of actions, state_int -> [Q(s, a)]
        self.value_avg: tt.Dict[int, tt.List[float]] = {}
        # prior probability of actions, state_int -> [P(s,a)]
        self.probs: tt.Dict[int, tt.List[float]] = {}
```

The key in all the dicts is the encoded game state (an integer), and values are lists, keeping the various parameters of actions that we have. The comments above every container have the same notations of values as in the AlphaGo Zero paper.

The `clear()` method clears the state without destroying the `MCTS` object, which happens when we switch the current best model to the new one and the gathered statistics become obsolete:

```
def clear(self):
    self.visit_count.clear()
    self.value.clear()
    self.value_avg.clear()
    self.probs.clear()
```

The `find_leaf()` method is used during the search to perform a single traversal of the game tree, starting from the root node given by the `state_int` argument and continuing to walk down until one of the following two situations has been faced: we reach the final game state or an as yet unexplored leaf has been found. During the search, we keep track of the visited states and the executed actions so we can update the nodes' statistics later:

```
def find_leaf(self, state_int: int, player: int):
    states = []
    actions = []
    cur_state = state_int
    cur_player = player
    value = None
```

Every iteration of the loop processes the game state that we're currently at. For this state, we extract the statistics that we need to make the decision about the action:

```
while not self.is_leaf(cur_state):
    states.append(cur_state)

    counts = self.visit_count[cur_state]
    total_sqrt = m.sqrt(sum(counts))
    probs = self.probs[cur_state]
    values_avg = self.value_avg[cur_state]
```

The decision about the action is based on the action utility, which is a sum of $Q(s, a)$ and the prior probabilities scaled to the visit count. The root node of the search process has extra noise added to the probabilities to improve the exploration of the search process. As we perform the MCTS from different game states along the self-play trajectories, this extra Dirichlet noise (according to the parameters used in the paper) ensures that we have tried different actions along the path:

```
if cur_state == state_int:
    noises = np.random.dirichlet([0.03] * game.GAME_COLS)
    probs = [0.75 * prob + 0.25 * noise for prob, noise in zip(probs,
        noises)]
    score = [
        value + self.c_puct*prob*total_sqrt/(1+count)
        for value, prob, count in zip(values_avg, probs, counts)
    ]
```

As we have calculated the score for the actions, we need to mask invalid actions for the state. (For example, when the column is full, we can't place another disk on the top.) After that, the action with the maximum score is selected and recorded:

```
invalid_actions = set(range(game.GAME_COLS)) - \
                  set(game.possible_moves(cur_state))
for invalid in invalid_actions:
```

```

        score[invalid] = -np.inf
action = int(np.argmax(score))
actions.append(action)

```

To finish the loop, we ask our game engine to make the move, returning the new state and the indication of whether the player won the game. The final game states (win, lose, or draw) are never added to the MCTS statistics, so they will always be leaf nodes. The function returns the game's value for the leaf player (or `None` if the final state hasn't been reached), the current player at the leaf state, the list of states we have visited during the search, and the list of the actions taken:

```

cur_state, won = game.move(cur_state, action, cur_player)
if won:
    value = -1.0
cur_player = 1-cur_player
# check for the draw
moves_count = len(game.possible_moves(cur_state))
if value is None and moves_count == 0:
    value = 0.0

return value, cur_state, cur_player, states, actions

```

The main entry point to the `MCTS` class is the `search_batch()` function, which performs several batches of searches. Every search consists of finding the leaf of the tree, optionally expanding the leaf, and doing backup. The main bottleneck here is the expand operation, which requires the NN to be used to get the prior probabilities of the actions and the estimated game value. To make this expansion more efficient, we use mini-batches when we search for several leaves, but then perform expansion in a single NN execution. This approach has one disadvantage: as several MCTSSs are performed in one batch, we don't get the same outcome as when they are executed serially.

Indeed, initially, when we have no nodes stored in the `MCTS` class, our first search will expand the root node, the second will expand some of its child nodes, and so on. However, one single batch of searches can expand only one root node at first. Of course, later, individual searches in the batch could follow the different game paths and expand more, but at first, mini-batch expansion is much less efficient in terms of exploration than a sequential MCTS.

To compensate for this, I still use mini-batches, but perform several of them:

```

def is_leaf(self, state_int):
    return state_int not in self.probs

def search_batch(self, count, batch_size, state_int, player, net, device="cpu"):
    for _ in range(count):
        self.search_minibatch(batch_size, state_int, player, net, device)

```

In the mini-batch search, we first perform the leaf search, starting from the same state. If the search has found a final game state (in that case, the returned value will not be equal to `None`), no expansion is required and we save the result for a backup operation. Otherwise, we store the leaf for later expansion:

```

def search_minibatch(self, count, state_int, player, net, device="cpu"):
    backup_queue = []
    expand_states = []
    expand_players = []
    expand_queue = []
    planned = set()
    for _ in range(count):
        value, leaf_state, leaf_player, states, actions = \
            self.find_leaf(state_int, player)
        if value is not None:
            backup_queue.append((value, states, actions))
        else:
            if leaf_state not in planned:
                planned.add(leaf_state)
                leaf_state_lists = game.decode_binary(leaf_state)
                expand_states.append(leaf_state_lists)
                expand_players.append(leaf_player)
                expand_queue.append((leaf_state, states, actions))

```

To expand, we convert the states into the form required by the model (there is a special function in the `model.py` library) and ask our network to return the prior probabilities and values for the batch of states. We will use those probabilities to create nodes, and the values will be backed up in a final statistics update:

```

if expand_queue:
    batch_v = model.state_lists_to_batch(expand_states, expand_players, device)
    logits_v, values_v = net(batch_v)
    probs_v = F.softmax(logits_v, dim=1)
    values = values_v.data.cpu().numpy()[:, 0]
    probs = probs_v.data.cpu().numpy()

```

Node creation is just storing zeros for every action in the visit count and action values (total and average). In prior probabilities, we store values obtained from the network:

```
for (leaf_state, states, actions), value, prob in \
    zip(expand_queue, values, probs):
    self.visit_count[leaf_state] = [0]*game.GAME_COLS
    self.value[leaf_state] = [0.0]*game.GAME_COLS
    self.value_avg[leaf_state] = [0.0]*game.GAME_COLS
    self.probs[leaf_state] = prob
    backup_queue.append((value, states, actions))
```

The backup operation is the core process in MCTS, and it updates the statistics for a state visited during the search. The visit count of the taken actions is incremented, the total values are summed, and the average values are normalized using visit counts.

It's very important to properly track the value of the game during the backup because we have two opponents, and in every turn, the value changes the sign (because a winning position for the current player is a losing game state for the opponent):

```
for value, states, actions in backup_queue:
    cur_value = -value
    for state_int, action in zip(states[::-1], actions[::-1]):
        self.visit_count[state_int][action] += 1
        self.value[state_int][action] += cur_value
        self.value_avg[state_int][action] = self.value[state_int][action] / \
            self.visit_count[state_int][action]
    cur_value = -cur_value
```

The final function in the class returns the probability of actions and the action values for the game state, using the statistics gathered during the MCTS:

```
def get_policy_value(self, state_int, tau=1):
    counts = self.visit_count[state_int]
    if tau == 0:
        probs = [0.0] * game.GAME_COLS
        probs[np.argmax(counts)] = 1.0
    else:
        counts = [count ** (1.0 / tau) for count in counts]
        total = sum(counts)
        probs = [count / total for count in counts]
    values = self.value_avg[state_int]
    return probs, values
```

Here, there are two modes of probability calculation, specified by the τ parameter. If it equals zero, the selection becomes deterministic, as we select the most frequently visited action. In other cases, the distribution given by

$$\frac{N(s, a)^{\frac{1}{\tau}}}{\sum_k N(s, k)^{\frac{1}{\tau}}}$$

is used, which, again, improves exploration.

The model

The NN used is a residual convolutional network with six layers, which is a simplified version of the network used in the original AlphaGo Zero method. For the input, we pass the encoded game state, which consists of two 6×7 channels. The first channel contains the places with the current player's disks, and the second channel has a value of 1.0 where the opponent has their disks. This representation allows us to make the network player invariant and analyze the position from the perspective of the current player.

The network consists of the common body with residual convolution filters. The features produced by them are passed to the policy and the value heads, which are a combination of a convolution layer and a fully connected layer. The policy head returns the logits for every possible action (the column in which a disk is dropped) and a single-value float. The details are available in the `lib/model.py` file.

Besides the model, this file contains two functions. The first, with the name `state_lists_to_batch()`, converts the batch of game states represented in lists into the model's input form. This function uses a utility function, `_encode_list_state`, which converts the states into a NumPy array:

```
def _encode_list_state(dest_np, state_list, who_move):
    assert dest_np.shape == OBS_SHAPE
    for col_idx, col in enumerate(state_list):
        for rev_row_idx, cell in enumerate(col):
            row_idx = game.GAME_ROWS - rev_row_idx - 1
            if cell == who_move:
                dest_np[0, row_idx, col_idx] = 1.0
            else:
                dest_np[1, row_idx, col_idx] = 1.0

def state_lists_to_batch(state_lists, who_moves_lists, device="cpu"):
    assert isinstance(state_lists, list)
    batch_size = len(state_lists)
    batch = np.zeros((batch_size,) + OBS_SHAPE, dtype=np.float32)
    for idx, (state, who_move) in enumerate(zip(state_lists, who_moves_lists)):
        _encode_list_state(batch[idx], state, who_move)
    return torch.tensor(batch).to(device)
```

The second method is called `play_game` and is very important for both the training and testing processes. Its purpose is to simulate the game between two NNs, perform the MCTS, and optionally store the taken moves in a replay buffer:

```
def play_game(mcts_stores: tt.Optional[mcts.MCTS | tt.List[mcts.MCTS]],
              replay_buffer: tt.Optional[collections.deque], net1: Net, net2: Net,
              steps_before_tau_0: int, mcts_searches: int, mcts_batch_size: int,
              net1_plays_first: tt.Optional[bool] = None,
              device: torch.device = torch.device("cpu")):
    if mcts_stores is None:
        mcts_stores = [mcts.MCTS(), mcts.MCTS()]
    elif isinstance(mcts_stores, mcts.MCTS):
        mcts_stores = [mcts_stores, mcts_stores]
```

As you can see in the preceding code, the function accepts a lot of parameters:

- The MCTS class instance, which could be a single instance, a list of two instances, or None. We need to be flexible there to cover different usages of this function.
- An optional replay buffer.
- NNs to be used during the game.
- The number of game steps that need to be taken before the parameter used for the action probability calculation will be changed from 1 to 0.
- The number of MCTSs to perform.
- The MCTS batch size.
- Which player acts first.

Before the game loop, we initialize the game state and select the first player. If there is no information given about who will make the first move, this is chosen randomly:

```
state = game.INITIAL_STATE
nets = [net1, net2]
if net1_plays_first is None:
    cur_player = np.random.choice(2)
else:
    cur_player = 0 if net1_plays_first else 1
step = 0
tau = 1 if steps_before_tau_0 > 0 else 0
game_history = []
```

In every turn, we perform the MCTS to populate the statistics and then obtain the probability of actions, which will be sampled to get the action:

```

result = None
net1_result = None

while result is None:
    mcts_stores[cur_player].search_batch(
        mcts_searches, mcts_batch_size, state,
        cur_player, nets[cur_player], device=device)
    probs, _ = mcts_stores[cur_player].get_policy_value(state, tau=tau)
    game_history.append((state, cur_player, probs))
    action = np.random.choice(game.GAME_COLS, p=probs)

```

Then, the game state is updated using the function in the game engine module, and the handling of different end-of-game situations (such as a win or a draw) is performed:

```

if action not in game.possible_moves(state):
    print("Impossible action selected")
state, won = game.move(state, action, cur_player)
if won:
    result = 1
    net1_result = 1 if cur_player == 0 else -1
    break
cur_player = 1-cur_player
# check the draw case
if len(game.possible_moves(state)) == 0:
    result = 0
    net1_result = 0
    break
step += 1
if step >= steps_before_tau_0:
    tau = 0

```

At the end of the function, we populate the replay buffer with probabilities for the action and the game result from the perspective of the current player. This data will be used to train the network:

```

if replay_buffer is not None:
    for state, cur_player, probs in reversed(game_history):
        replay_buffer.append((state, cur_player, probs, result))
        result = -result

return net1_result, step

```

Training

With all those functions in hand, the training process is a simple combination of them in the correct order. The training program is available in `train.py`, and it has logic that has already been described: in the loop, our current best model constantly plays against itself, saving the steps in the replay buffer. Another network is trained on this data, minimizing the cross-entropy between the probabilities of actions sampled from MCTS and the result of the policy head. MSE between the value head predictions, about the game and the actual game result, is also added to the total loss.

Periodically, the network being trained and the current best network play 100 matches, and if the current network is able to win in more than 60% of them, the network's weights are synced. This process continues infinitely, hopefully, finding models that are more and more proficient at the game.

Testing and comparison

During the training process, the model's weights are saved every time the current best model is replaced with the trained model. As a result, we get multiple agents of various strengths. In theory, the later models should be better than the preceding ones, but we would like to check this ourselves. To do this, there is a tool, `play.py`, that takes several model files and plays a tournament in which every model plays a specified number of rounds with all the others. The results table, with the number of wins achieved by every model, will represent the relative model's strength.

Results

To make the training fast, I intentionally set the hyperparameters of the training process to small values. For example, at every step of the self-play process, only 10 MCTSs were performed, each with a mini-batch size of eight. This, in combination with efficient mini-batch MCTS and the fast game engine, made training very fast.

Basically, after just one hour of training and 2,500 games played in self-play mode, the produced model was sophisticated enough to be enjoyable to play against. Of course, the level of its play was well below even a child's level, but it showed some rudimentary strategies and made mistakes in only every other move, which was good progress.

I've done two rounds of training, the first with a learning rate of 0.1 and the second with a learning rate of 0.001. Every experiment was trained for 10 hours and 40K game rounds. In *Figure 20.3*, you can see charts with the win ratio (win/loss for the current evaluated policy versus the current best policy). As you can see, both learning rate values are oscillating around 0.5, sometimes spiking to 0.8-0.9:

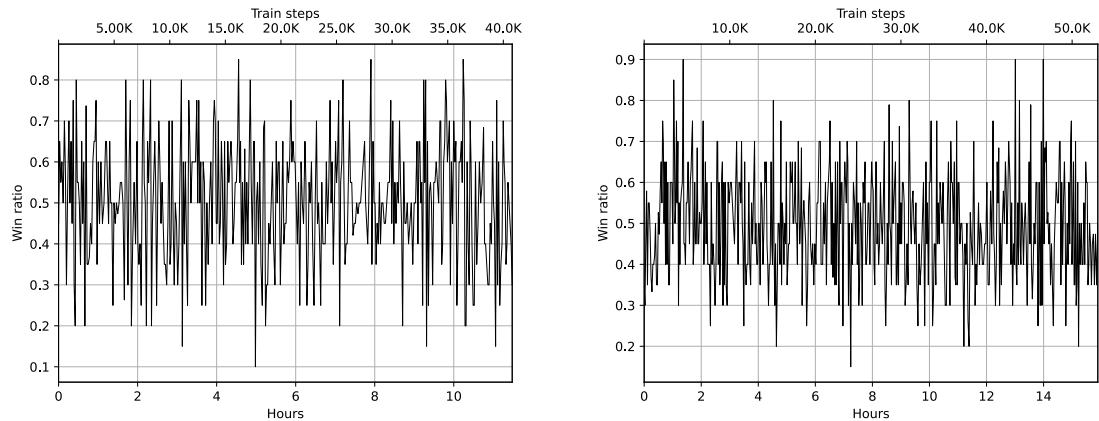


Figure 20.3: The win ratio for training with two learning rates; learning rate=0.1 (left) and learning rate=0.001 (right)

Figure 20.4 shows the total loss for both experiments, and there is no clear trend. This is due to constant switches of the current best policy, which leads to constant retraining of the trained model.

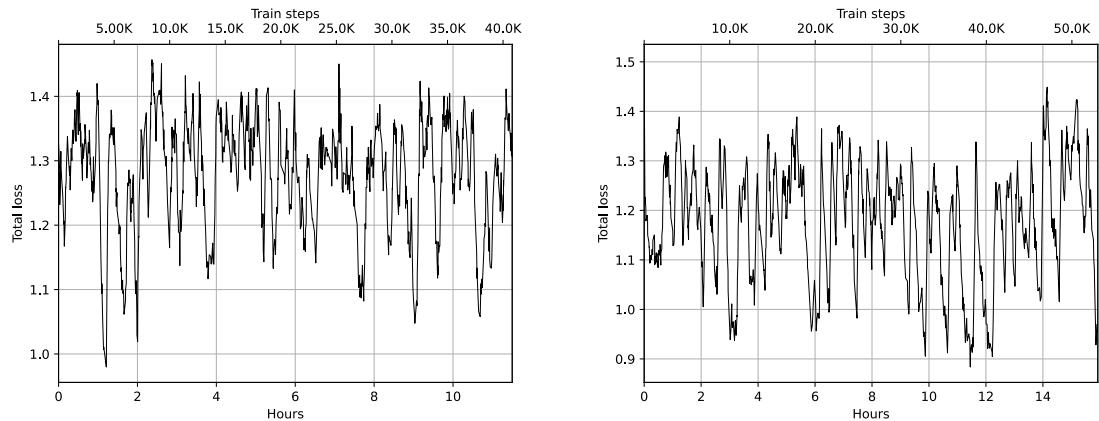


Figure 20.4: The total loss for training with two learning rates; learning rate=0.1 (left) and learning rate=0.001 (right)

The tournament verification was complicated by the number of different models, as several games needed to be played by each pair to estimate their strength. At the beginning, I ran 10 rounds for each model stored during every experiment (separately). To do this, you can run the `play.py` utility like this:

```
./play.py --cuda -r 10 saves/v2/best\_* > semi-v2.txt
```

But for 100 models, it will take a while, as every model plays 10 rounds with all the other models.

After all the testing, the utility prints on the console the result of all the games and the leaderboard of models. The following is the top 10 for experiment 1 (learning rate=0.1):

```
saves/t1/best_088_39300.dat:      w=1027, l=732, d=1
saves/t1/best_025_09900.dat:      w=1024, l=735, d=1
saves/t1/best_022_08200.dat:      w=1023, l=737, d=0
saves/t1/best_021_08100.dat:      w=1017, l=743, d=0
saves/t1/best_009_03400.dat:      w=1010, l=749, d=1
saves/t1/best_014_04700.dat:      w=1003, l=757, d=0
saves/t1/best_008_02700.dat:      w=998, l=760, d=2
saves/t1/best_010_03500.dat:      w=997, l=762, d=1
saves/t1/best_029_11800.dat:      w=991, l=768, d=1
saves/t1/best_007_02300.dat:      w=980, l=779, d=1
```

Here's the top 10 for experiment 2 (learning rate=0.001):

```
saves/t2/best_069_41500.dat:      w=1023, l=757, d=0
saves/t2/best_070_42200.dat:      w=1016, l=764, d=0
saves/t2/best_066_38900.dat:      w=1005, l=775, d=0
saves/t2/best_071_42600.dat:      w=1003, l=777, d=0
saves/t2/best_059_33700.dat:      w=999, l=781, d=0
saves/t2/best_049_27500.dat:      w=990, l=790, d=0
saves/t2/best_068_41300.dat:      w=990, l=789, d=1
saves/t2/best_048_26700.dat:      w=983, l=796, d=1
saves/t2/best_058_32100.dat:      w=982, l=797, d=1
saves/t2/best_076_45200.dat:      w=982, l=795, d=3
```

To check that our training generates better models, I have plotted the win ratio of the models versus their index in *Figure 20.5*. The Y axis is the relative win ratio and the X axis is the index (which is increased during training). As you can see, the quality of models in each experiment is increased, but experiments with smaller learning rates have more consistent behavior.

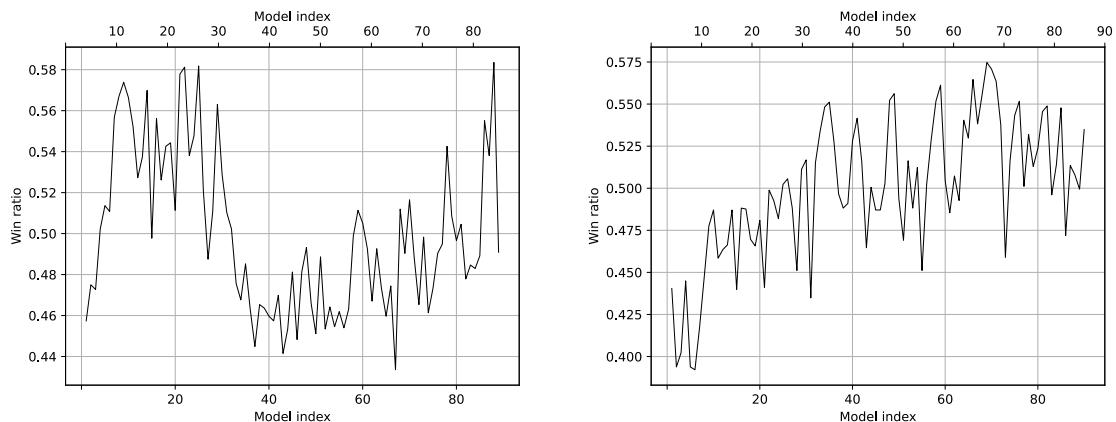


Figure 20.5: Win ratio for best models during the training, learning rate=0.1 (left) and learning rate=0.001 (right)

I haven't done much hyperparameter tuning of the training, so they definitely could be improved. You can try experimenting with this yourself.

It was also interesting to compare the results with different learning rates. To do that, I've taken 10 best models for each experiment and run 10 rounds of games. Here is the top 10 leaderboard for this tournament:

saves/t2/best_059_33700.dat:	w=242, l=138, d=0
saves/t2/best_058_32100.dat:	w=223, l=157, d=0
saves/t2/best_071_42600.dat:	w=217, l=163, d=0
saves/t2/best_068_41300.dat:	w=210, l=170, d=0
saves/t2/best_076_45200.dat:	w=208, l=171, d=1
saves/t2/best_048_26700.dat:	w=202, l=178, d=0
saves/t2/best_069_41500.dat:	w=201, l=179, d=0
saves/t2/best_049_27500.dat:	w=199, l=181, d=0
saves/t2/best_070_42200.dat:	w=197, l=183, d=0
saves/t1/best_021_08100.dat:	w=192, l=188, d=0

As you can see, models trained with a learning rate of 0.001 are winning in the joint tournament by a significant margin.

MuZero

The successor of AlphaGo Zero (published in 2017) was a method called MuZero, described by Schrittwieser et al. from DeepMind in the paper *Mastering Atari, Go, chess and shogi by planning with a learned model* [Sch+20] published in 2020. In this method, the authors made an attempt to generalize the method by removing the requirement of the precise game model, but still keeping the method in the model-based family.

As we saw in the description of Alpha Go Zero, the game model is heavily used during the training process: in the MCTS phase, we use the game model to obtain the available actions in the current state and the new state of the game after applying the action. In addition, the game model provides the final game outcome: whether we have won or lost the game.

At first glance, it looks almost impossible to get rid of the model from the training process, but MuZero not only demonstrated how it could be done, but has also beaten the previous AlphaGo Zero records in Go, chess, and shogi, and established state-of-the-art results in 57 Atari games.

In this part of the chapter, we'll discuss the method in detail, implement it, and compare it to AlphaGo Zero using Connect 4.

High-level model

First, let's take a look at MuZero from a high level. As in AlphaGo Zero, the core is MCTS, which is performed many times to calculate statistics about possible future outcomes of the game state we currently have at the root of the tree. After this search, we calculate visit counters, indicating how frequently the actions have been executed.

But instead of using the game model to answer the question “what state do I get if I execute this action from this state?”, MuZero introduces two extra neural networks:

1. **representation** $h_\theta(o) \rightarrow s$: To compute the hidden state of the game observation
2. **dynamics** $g_\theta(s, a) \rightarrow r, s'$: To apply the action a to the hidden state s , transforming it into the next state s' (and obtaining the immediate reward r)

As you may remember, in AlphaGo Zero, only one network $f_\theta(s) \rightarrow \pi, v$ was used, which predicted the policy π and the value v of the current state s . MuZero uses three networks for its operation, which are trained simultaneously. I'll explain how the training is done a bit later, but for now let's focus on MCTS.

In *Figure 20.6*, the MCTS process is shown schematically, indicating the values we compute using our neural networks. As the first step, we compute the hidden state s^0 for the current game observation o using our representation network h_θ .

Having the hidden state, we can use the network f_θ to compute the policy π^0 and values v^0 of this state – quantities that indicate what action we should take (π^0) and the expected outcome of those actions (v^0).

We use the policy and the value (with visit count statistics for the actions) to compute the utility value for the action $U(s, a)$ in a similar way to in AlphaGo Zero. Then, the action with the maximum utility value is selected for the tree descent.

If it is the first time we select this action from this state node (in other words, the node has not been expanded yet), we use the neural network $g_\theta(s^0, a) \rightarrow r^1, s^1$ to obtain immediate reward r^1 and the next hidden state s^1 .

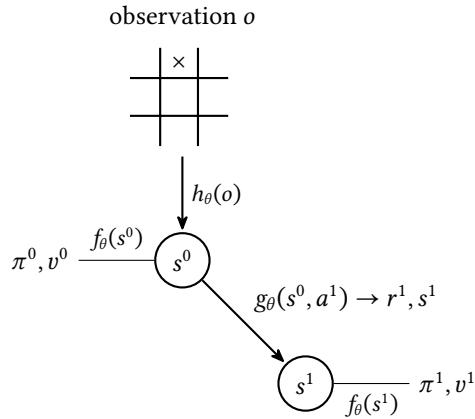


Figure 20.6: Monte-Carlo Tree Search in MuZero

This process is repeated over and over again hundreds of times, accumulating visit counters for actions, expanding more and more nodes in the tree. In every node expansion, the value of the node, obtained from f_θ , is added to all the nodes along the search path until the tree's root. In the AlphaGo Zero paper, this process was called “backup,” while in the MuZero paper, the term “backpropagation” was used. But essentially, the meaning is the same – adding value from the expanded node to the root of the tree, altering the sign.

After some time (800 searches in the original MuZero method), the actions’ visit counts are accurate enough (or we believe they are accurate enough) to be used as an approximation of the policy for the action selection and for the training.

Training process

MCTS, as described above, is used for the single game state (at the root of the tree). After all the search rounds, we select the action from this root state based on the frequency of actions performed during the search. Then, the selected action is executed in the environment and the next state and the reward are obtained. After that, another MCTS is performed using the next state as the root of the search tree.

This process allows us to generate episodes. We’re storing them in the replay buffer and using them for the training. To prepare the training batch, we sample an episode from the replay buffer and randomly select an offset in the episode. Then, starting from this position in the episode, we unroll the episode to the fixed number of steps (in the MuZero paper, a five-step unroll was used).

On every step of unroll, the following data is accumulated:

- Action frequencies from MCTS are used as policy targets (trained using cross-entropy loss).
- A discounted sum of rewards until the end of the episode is used as a value target (trained with MSE loss).
- Immediate rewards are used as targets for the reward value predicted by the dynamics network (also trained with MSE loss).

Besides that, we remember the action taken in every unroll step, which will be used as input for the dynamics network, $g_\theta(s, a) \rightarrow r, s'$.

Once the batch is generated, we apply the representation network $h_\theta(o)$ to the game observations (the first step of the unrolled episode's segments). Then, we repeat the unrolling by computing the policy π and the value v from the current hidden state, compute their loss, and perform the dynamics network step to obtain the next hidden state. This process is repeated for five steps (the length of the unroll). Schrittwieser et al. used gradient scaling by a factor of 0.5 for unrolled steps, but in my implementation, I just multiplied the loss with this constant to get the same effect.

Connect 4 with MuZero

Now that we have discussed the method, let's check its implementation and the results in Connect 4. The implementaton consists of several modules:

- `lib/muzero.py`: Contains MCTS data structures and functions, neural networks, and batch generation logic
- `train-mu.py`: The training loop, implementing self-play for episode generation, training, and periodic validation of the currently trained model versus the best model (the same as the AlphaGo Zero method)
- `play-mu.py`: Performs a series of games between the list of models to get their rankings

Hyperparameters and MCTS tree nodes

Most MuZero hyperparameters are put in a separate dataclass to simplify passing them around the code:

```
@dataclass
class MuZeroParams:
    actions_count: int = game.GAME_COLS
    max_moves: int = game.GAME_COLS * game.GAME_ROWS >> 2 + 1
    dirichlet_alpha: float = 0.3
    discount: float = 1.0
    unroll_steps: int = 5
```

```
pb_c_base: int = 19652
pb_c_init: float = 1.25

dev: torch.device = torch.device("cpu")
```

I'm not going to explain these parameters here. I will do that when we discuss the relevant pieces of the code.

MCTS for MuZero is implemented differently than the AlphaGo Zero implementation. In our AlphaGo Zero implementation, every MCTS node had a unique identifier of the game state, which was an integer. As a result, we kept the whole tree in several dictionaries, mapping the game state to the node's attributes, such as visit counters, the states of the child nodes, and so on. Every time we saw the game state, we simply updated those dictionaries.

However, in MuZero, every MCTS node is now identified by a hidden state, which is a list of floats (since the hidden state is produced by the neural network). As a result, we cannot compare two hidden states to check whether they are the same or not. To get around this, we're now storing the tree "properly" – as nodes referencing child nodes, which is less efficient from a memory point of view.

The following is the core MCTS data structure: an object representing a tree node. For the constructor, we just create an empty unexpanded node:

```
class MCTSNode:
    def __init__(self, prior: float, first_plays: bool):
        self.first_plays: bool = first_plays
        self.visit_count = 0
        self.value_sum = 0.0
        self.prior = prior
        self.children: Dict[Action, MCTSNode] = {}
        # node is not expanded, so has no hidden state
        self.h = None
        # predicted reward
        self.r = 0.0
```

The expansion of the node is implemented in the `expand_node` method, which will be shown later, after introducing the models. For now, the node is expanded if it has child nodes (actions) and has a hidden state, policy, and value calculated using NNs.

The value of the node is computed as the sum of all the values from the children divided by the number of visits:

```
@property
def is_expanded(self) -> bool:
    return bool(self.children)

@property
def value(self) -> float:
    return 0 if not self.visit_count else self.value_sum / self.visit_count
```

The `select_child` method performs the action selection during the MCTS search. This is done by selecting the child with the largest value returned by the `ucb_value` function, which will be shown shortly:

```
def select_child(self, params: MuZeroParams, min_max: MinMaxStats) -> \
    tt.Tuple[Action, "MCTSNode"]:
    max_ucb, best_action, best_node = None, None, None
    for action, node in self.children.items():
        ucb = ucb_value(params, self, node, min_max)
        if max_ucb is None or max_ucb < ucb:
            max_ucb = ucb
            best_action = action
            best_node = node
    return best_action, best_node
```

The `ucb_value` method implements the **Upper Confidence Bound (UCB)** calculation for the node, and it is very similar to the formula we discussed for AlphaGo Zero. The UCB is calculated from the node's value and the prior multiplied by a coefficient:

```
def ucb_value(params: MuZeroParams, parent: MCTSNode, child: MCTSNode,
              min_max: MinMaxStats) -> float:
    pb_c = m.log((parent.visit_count + params.pb_c_base + 1) /
                  params.pb_c_base) + params.pb_c_init
    pb_c *= m.sqrt(parent.visit_count) / (child.visit_count + 1)
    prior_score = pb_c * child.prior
    value_score = 0.0
    if child.visit_count > 0:
        value_score = min_max.normalize(child.value + child.r)
    return prior_score + value_score
```

Another method of the `MCTSNode` class is `get_act_probs()`, which returns approximated probabilities from visit counters. Those probabilities are used as targets for the policy network training. This method has a special “temperature coefficient” that allows us to vary the entropy in different stages of training: if the temperature is close to zero, we assign a higher probability to the action with the highest number of visits. If the temperature is high, the distribution becomes more uniform:

```
def get_act_probs(self, t: float = 1) -> List[float]:
    child_visits = sum(map(lambda n: n.visit_count, self.children.values()))
    p = np.array([(child.visit_count / child_visits) ** (1 / t)
                  for _, child in sorted(self.children.items())])
    p /= sum(p)
    return list(p)
```

The last method of `MCTSNode` is `select_action()`, which uses the `get_act_probs()` method to select the action, handling several corner cases as follows:

- If we have no children in the node, the action is done randomly
- If the temperature coefficient is too small, we take the action with the largest visit count
- Otherwise, we use `get_act_probs()` to get the probabilities for every action based on the temperature coefficient and select the action based on those probabilities

```
def select_action(self, t: float, params: MuZeroParams) -> Action:
    act_vals = list(sorted(self.children.keys()))

    if not act_vals:
        res = np.random.choice(params.actions_count)
    elif t < 0.0001:
        res, _ = max(self.children.items(), key=lambda p: p[1].visit_count)
    else:
        p = self.get_act_probs(t)
        res = int(np.random.choice(act_vals, p=p))
    return res
```

The preceding code might look tricky and non-relevant, but it will fit together when we discuss the MuZero models and the MCTS search procedure.

Models

As we have mentioned, MuZero uses three NNs for various purposes. Let's take a look at them. You'll find all the code in the GitHub repository in the `lib/muzero.py` module.

The first model is the representation model, $h_\theta(o) \rightarrow s$, which maps game observations into the hidden state. The observations are exactly the same as in the AlphaGo Zero code – we have a tensor of size $2 \times 6 \times 7$, where 6×7 is the board size and two planes are the one-hot encoded position of the current player's and the opponent's disks. The dimension of the hidden state is given by the `HIDDEN_STATE_SIZE=64` hyperparameter:

```
class ReprModel(nn.Module):
    def __init__(self, input_shape: tt.Tuple[int, ...]):
        super(ReprModel, self).__init__()
        self.conv_in = nn.Sequential(
            nn.Conv2d(input_shape[0], NUM_FILTERS, kernel_size=3, padding=1),
            nn.BatchNorm2d(NUM_FILTERS),
            nn.LeakyReLU()
        )
        # layers with residual
        self.conv_1 = nn.Sequential(
            nn.Conv2d(NUM_FILTERS, NUM_FILTERS, kernel_size=3, padding=1),
            nn.BatchNorm2d(NUM_FILTERS),
            nn.LeakyReLU()
        )
        self.conv_2 = nn.Sequential(
            nn.Conv2d(NUM_FILTERS, NUM_FILTERS, kernel_size=3, padding=1),
            nn.BatchNorm2d(NUM_FILTERS),
            nn.LeakyReLU()
        )
        self.conv_3 = nn.Sequential(
            nn.Conv2d(NUM_FILTERS, NUM_FILTERS, kernel_size=3, padding=1),
            nn.BatchNorm2d(NUM_FILTERS),
            nn.LeakyReLU()
        )
        self.conv_4 = nn.Sequential(
            nn.Conv2d(NUM_FILTERS, NUM_FILTERS, kernel_size=3, padding=1),
            nn.BatchNorm2d(NUM_FILTERS),
            nn.LeakyReLU()
        )
        self.conv_5 = nn.Sequential(
            nn.Conv2d(NUM_FILTERS, NUM_FILTERS, kernel_size=3, padding=1),
            nn.BatchNorm2d(NUM_FILTERS),
            nn.LeakyReLU(),
        )
        self.conv_out = nn.Sequential(
            nn.Conv2d(NUM_FILTERS, 16, kernel_size=1),
```

```

        nn.BatchNorm2d(16),
        nn.LeakyReLU(),
        nn.Flatten()
    )

body_shape = (NUM_FILTERS,) + input_shape[1:]
size = self.conv_out(torch.zeros(1, *body_shape)).size()[-1]
self.out = nn.Sequential(
    nn.Linear(size, 128),
    nn.ReLU(),
    nn.Linear(128, HIDDEN_STATE_SIZE),
)

```

The structure of the network is almost the same as in the AlphaGo Zero example, with the difference that it returns a hidden state vector instead of the policy and values.

As the network blocks are residual, special handling of every layer is required:

```

def forward(self, x):
    v = self.conv_in(x)
    v = v + self.conv_1(v)
    v = v + self.conv_2(v)
    v = v + self.conv_3(v)
    v = v + self.conv_4(v)
    v = v + self.conv_5(v)
    c_out = self.conv_out(v)
    out = self.out(c_out)
    return out

```

The second model is the prediction model, $f_\theta(s) \rightarrow \pi, v$, which takes the hidden state and returns the policy and the value. In my example, I used two-layer heads for the policy and the value:

```

class PredModel(nn.Module):
    def __init__(self, actions: int):
        super(PredModel, self).__init__()
        self.policy = nn.Sequential(
            nn.Linear(HIDDEN_STATE_SIZE, 128),
            nn.ReLU(),
            nn.Linear(128, actions),
        )

        self.value = nn.Sequential(
            nn.Linear(HIDDEN_STATE_SIZE, 128),

```

```

        nn.ReLU(),
        nn.Linear(128, 1),
    )

def forward(self, x) -> tt.Tuple[torch.Tensor, torch.Tensor]:
    return self.policy(x), self.value(x).squeeze(1)

```

The third model we have is the dynamics model, $g_\theta(s, a) \rightarrow r, s'$, which takes the hidden state and one-hot encoded actions and returns the immediate reward and the next state:

```

class DynamicsModel(nn.Module):
    def __init__(self, actions: int):
        super(DynamicsModel, self).__init__()
        self.reward = nn.Sequential(
            nn.Linear(HIDDEN_STATE_SIZE + actions, 128),
            nn.ReLU(),
            nn.Linear(128, 1),
        )

        self.hidden = nn.Sequential(
            nn.Linear(HIDDEN_STATE_SIZE + actions, 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, HIDDEN_STATE_SIZE),
        )

    def forward(self, h: torch.Tensor, a: torch.Tensor) -> \
        tt.Tuple[torch.Tensor, torch.Tensor]:
        x = torch.hstack((h, a))
        return self.reward(x).squeeze(1), self.hidden(x)

```

For convenience, all three networks are kept in the `MuZeroModels` class, which provides the required functionality:

```

class MuZeroModels:
    def __init__(self, input_shape: tt.Tuple[int, ...], actions: int):
        self.repr = ReprModel(input_shape)
        self.pred = PredModel(actions)
        self.dynamics = DynamicsModel(actions)

    def to(self, dev: torch.device):
        self.repr.to(dev)
        self.pred.to(dev)

```

```
    self.dynamics.to(dev)
```

The class provides methods for syncing networks from the other instance. We will use it to store the best model after validation.

In addition, there are two methods for storing and loading the networks' weights:

```
def sync(self, src: "MuZeroModels"):
    self.repr.load_state_dict(src.repr.state_dict())
    self.pred.load_state_dict(src.pred.state_dict())
    self.dynamics.load_state_dict(src.dynamics.state_dict())

def get_state_dict(self) -> tt.Dict[str, dict]:
    return {
        "repr": self.repr.state_dict(),
        "pred": self.pred.state_dict(),
        "dynamics": self.dynamics.state_dict(),
    }

def set_state_dict(self, d: dict):
    self.repr.load_state_dict(d['repr'])
    self.pred.load_state_dict(d['pred'])
    self.dynamics.load_state_dict(d['dynamics'])
```

Now that we've seen the models, we're ready to get to the functions that implement the MCTS logic and the gameplay loop.

MCTS search

First, we have two functions doing similar tasks, but in different situations:

- `make_expanded_root()` creates the MCTS tree root from the given game state. For the root, we have no parent node, so we don't need to apply the dynamic NN; instead, we obtain the node hidden state from the encoded game observation with the representation network.
- `expand_node()` expands the non-root MCTS node. In this case, we perform the dynamics step using the NN to take the parent's hidden state and generate the hidden state for the child node.

At the beginning of the first function, we create a new `MCTSNode`, decode the game state into a list representation, and convert it into a tensor. Then, the representation network is used to obtain the node's hidden state:

```
def make_expanded_root(player_idx: int, game_state_int: int, params: MuZeroParams,
    models: MuZeroModels, min_max: MinMaxStats) -> MCTSNode:
    root = MCTSNode(1.0, player_idx == 0)
    state_list = game.decode_binary(game_state_int)
    state_t = state_lists_to_batch([state_list], [player_idx], device=params.dev)
    h_t = models.repr(state_t)
    root.h = h_t[0].cpu().numpy()
```

Using the hidden state, we get the policy and the value of the node and convert the policy logits into probabilities, after which some random noise is added to increase exploration:

```
p_t, v_t = models.pred(h_t)
# logits to probs
p_t.exp_()
probs_t = p_t.squeeze(0) / p_t.sum()
probs = probs_t.cpu().numpy()
# add dirichlet noise
noises = np.random.dirichlet([params.dirichlet_alpha] * params.actions_count)
probs = probs * 0.75 + noises * 0.25
```

As we've got probabilities, we create child nodes and backpropagate the value of the node. The `backpropagate()` method will be discussed a bit later; it adds the node value along the search path. For the root node, our search path has only the root, so it will be just one step (in the next method, `expand_node()`, the path could be much longer):

```
for a, prob in enumerate(probs):
    root.children[a] = MCTSNode(prob, not root.first_plays)
v = v_t.cpu().item()
backpropagate([root], v, root.first_plays, params, min_max)
return root
```

The `expand_node()` method is similar, but is used for non-root nodes, so it performs the dynamics step using the parent's hidden state:

```
def expand_node(parent: MCTSNode, node: MCTSNode, last_action: Action,
    params: MuZeroParams, models: MuZeroModels) -> float:
    h_t = torch.as_tensor(parent.h, dtype=torch.float32, device=params.dev)
```

```

    h_t.unsqueeze_(0)
    p_t, v_t = models.pred(h_t)
    a_t = torch.zeros(params.actions_count, dtype=torch.float32, device=params.dev)
    a_t[last_action] = 1.0
    a_t.unsqueeze_(0)
    r_t, h_next_t = models.dynamics(h_t, a_t)
    node.h = h_next_t[0].cpu().numpy()
    node.r = float(r_t[0].cpu().item())

```

The rest of the logic is the same, with the exception that noise is not added to the non-root nodes:

```

    p_t.squeeze_(0)
    p_t.exp_()
    probs_t = p_t / p_t.sum()
    probs = probs_t.cpu().numpy()
    for a, prob in enumerate(probs):
        node.children[a] = MCTSNode(prob, not node.first_plays)
    return float(v_t.cpu().item())

```

The `backpropagate()` function is used to add discounted values to the nodes along the search path. The signs of the values are changed at every level to indicate that player's turn is changing. So, a positive value for us means a negative value for the opponent and vice versa:

```

def backpropagate(search_path: tt.List[MCTSNode], value: float, first_plays: bool,
                  params: MuZeroParams, min_max: MinMaxStats):
    for node in reversed(search_path):
        node.value_sum += value if node.first_plays == first_plays else -value
        node.visit_count += 1
        value = node.r + params.discount * value
        min_max.update(value)

```

The instance of the `MinMaxStats` class is used to keep the minimal and maximal value for the tree during the search. Then, those extremes are used to normalize the resulting values.

With all those functions, let's now look at the logic of actual MCTS search. At first, we create a root node, then perform several search rounds. In every round, we traverse the tree by following the UCB value function. When we find a node that is not expanded, we expand it and backpropagate the value to the root of the tree:

```

@torch.no_grad()
def run_mcts(player_idx: int, root_state_int: int, params: MuZeroParams,
            models: MuZeroModels, min_max: MinMaxStats,

```

```

        search_rounds: int = 800) -> MCTSNode:
    root = make_expanded_root(player_idx, root_state_int, params, models, min_max)
    for _ in range(search_rounds):
        search_path = [root]
        parent_node = None
        last_action = 0
        node = root
        while node.is_expanded:
            action, new_node = node.select_child(params, min_max)
            last_action = action
            parent_node = node
            node = new_node
            search_path.append(new_node)
        value = expand_node(parent_node, node, last_action, params, models)
        backpropagate(search_path, value, node.first_plays, params, min_max)
    return root

```

As you can see, this implementation uses NNs without processing nodes in batches. The problem with the MuZero MCTS process is that the search process is deterministic and driven by nodes' values (which are updated when a node is expanded) and visit counters. As a result, batching has no effect because repeating the search without expanding the node will lead to the same path in the tree, so expansions have to be done one by one. This is a very inefficient way of using NNs, which negatively impacts the overall performance. Here, my intention was not to implement the most efficient possible version of MuZero, but rather to demonstrate a working prototype for you, so I did no optimization. As an exercise, you can change the implementation to perform MCTS searches in parallel from several processes. As an alternative (or in addition), you could add noise during the MCTS search and use batching similarly to when we discussed AlphaGo Zero.

Training data and gameplay

To store the data for training, we have the `Episode` class, which keeps the sequence of `EpisodeStep` objects with additional information:

```

@dataclass
class EpisodeStep:
    state: int
    player_idx: int
    action: int
    reward: int

class Episode:
    def __init__(self):

```

```

    self.steps: tt.List[EpisodeStep] = []
    self.action_probs: tt.List[tt.List[float]] = []
    self.root_values: tt.List[float] = []

    def __len__(self):
        return len(self.steps)

    def add_step(self, step: EpisodeStep, node: MCTSNode):
        self.steps.append(step)
        self.action_probs.append(node.get_act_probs())
        self.root_values.append(node.value)

```

Now, let's take a look at the `play_game()` function, which uses MCTS search several times to play the full episode. At the beginning of the function, we create the game state and the required objects:

```

@torch.no_grad()
def play_game(
    player1: MuZeroModels, player2: MuZeroModels, params: MuZeroParams,
    temperature: float, init_state: tt.Optional[int] = None
) -> tt.Tuple[int, Episode]:
    episode = Episode()
    state = game.INITIAL_STATE if init_state is None else init_state
    players = [player1, player2]
    player_idx = 0
    reward = 0
    min_max = MinMaxStats()

```

At the beginning of the game loop, we check if the game is a draw and then run the MCTS search to accumulate statistics. After that, we select an action using random sampling from the actions' frequencies (not UCB values):

```

while True:
    possible_actions = game.possible_moves(state)
    if not possible_actions:
        break

    root_node = run_mcts(player_idx, state, params, players[player_idx], min_max)
    action = root_node.select_action(temperature, params)

    # act randomly on wrong move
    if action not in possible_actions:
        action = int(np.random.choice(possible_actions))

```

Once the action has been selected, we perform a move in our game environment and check for win/lose situations. Then, the process is repeated:

```

new_state, won = game.move(state, action, player_idx)
if won:
    if player_idx == 0:
        reward = 1
    else:
        reward = -1
    step = EpisodeStep(state, player_idx, action, reward)
    episode.add_step(step, root_node)
    if won:
        break
    player_idx = (player_idx + 1) % 2
    state = new_state
return reward, episode

```

Finally, we have the method that samples the batch of training data from the replay buffer (which is a list of `Episode` objects). If you remember, the training data is created by unrolling from a random position in a random episode. This is needed to apply the dynamics network and optimize it with actual data. So, our batch is not a tensor, but a list of tensors, where every tensor is a step in the unroll process.

In preparation for the batch sampling, we create empty lists of the required size:

```

def sample_batch(
    episode_buffer: tt.Deque[Episode], batch_size: int, params: MuZeroParams,
) -> tt.Tuple[
    torch.Tensor, tt.Tuple[torch.Tensor, ...], tt.Tuple[torch.Tensor, ...],
    tt.Tuple[torch.Tensor, ...], tt.Tuple[torch.Tensor, ...],
]:
    states = []
    player_indices = []
    actions = [[] for _ in range(params.unroll_steps)]
    policy_targets = [[] for _ in range(params.unroll_steps)]
    rewards = [[] for _ in range(params.unroll_steps)]
    values = [[] for _ in range(params.unroll_steps)]

```

Then we sample a random episode and select an offset in this episode:

```

for episode in np.random.choice(episode_buffer, batch_size):
    assert isinstance(episode, Episode)
    ofs = np.random.choice(len(episode) - params.unroll_steps)
    state = game.decode_binary(episode.steps[ofs].state)

```

```

    states.append(state)
    player_indices.append(episode.steps[ofs].player_idx)

```

After that, the unroll for a specific number of steps (five, as in the paper) is performed. At every step, we remember the action, the immediate reward, and the actions' probabilities. After that, we compute the value target by summing the discounted rewards until the end of the episode:

```

for s in range(params.unroll_steps):
    full_ofs = ofs + s
    actions[s].append(episode.steps[full_ofs].action)
    rewards[s].append(episode.steps[full_ofs].reward)
    policy_targets[s].append(episode.action_probs[full_ofs])

    value = 0.0
    for step in reversed(episode.steps[full_ofs:]):
        value *= params.discount
        value += step.reward
    values[s].append(value)

```

With this preparation data aggregated, we convert it into tensors. Actions are one-hot encoded using the `eye()` NumPy function with indexing:

```

states_t = state_lists_to_batch(states, player_indices, device=params.dev)
res_actions = tuple(
    torch.as_tensor(np.eye(params.actions_count)[a],
                  dtype=torch.float32, device=params.dev)
    for a in actions
)
res_policies = tuple(
    torch.as_tensor(p, dtype=torch.float32, device=params.dev)
    for p in policy_targets
)
res_rewards = tuple(
    torch.as_tensor(r, dtype=torch.float32, device=params.dev)
    for r in rewards
)
res_values = tuple(
    torch.as_tensor(v, dtype=torch.float32, device=params.dev)
    for v in values
)
return states_t, res_actions, res_policies, res_rewards, res_values

```

I'm not going to show the full training loop here; we perform the self-play with the current best model to populate the replay buffer. The full training code is in the `train-mu.py` module. The following code optimizes the network:

```
states_t, actions, policy_tgt, rewards_tgt, values_tgt = \
    mu.sample_batch(replay_buffer, BATCH_SIZE, params)

optimizer.zero_grad()
h_t = net.repr(states_t)
loss_p_full_t = None
loss_v_full_t = None
loss_r_full_t = None
for step in range(params.unroll_steps):
    policy_t, values_t = net.pred(h_t)
    loss_p_t = F.cross_entropy(policy_t, policy_tgt[step])
    loss_v_t = F.mse_loss(values_t, values_tgt[step])
    # dynamic step
    rewards_t, h_t = net.dynamics(h_t, actions[step])
    loss_r_t = F.mse_loss(rewards_t, rewards_tgt[step])
    if step == 0:
        loss_p_full_t = loss_p_t
        loss_v_full_t = loss_v_t
        loss_r_full_t = loss_r_t
    else:
        loss_p_full_t += loss_p_t * 0.5
        loss_v_full_t += loss_v_t * 0.5
        loss_r_full_t += loss_r_t * 0.5
loss_full_t = loss_v_full_t + loss_p_full_t + loss_r_full_t
loss_full_t.backward()
optimizer.step()
```

MuZero results

I ran the training for 15 hours and it played 3,400 episodes (you see, the training is not very fast). The policy and value losses are shown in *Figure 20.7*. As often happens with self-play training, the charts have no obvious trend:

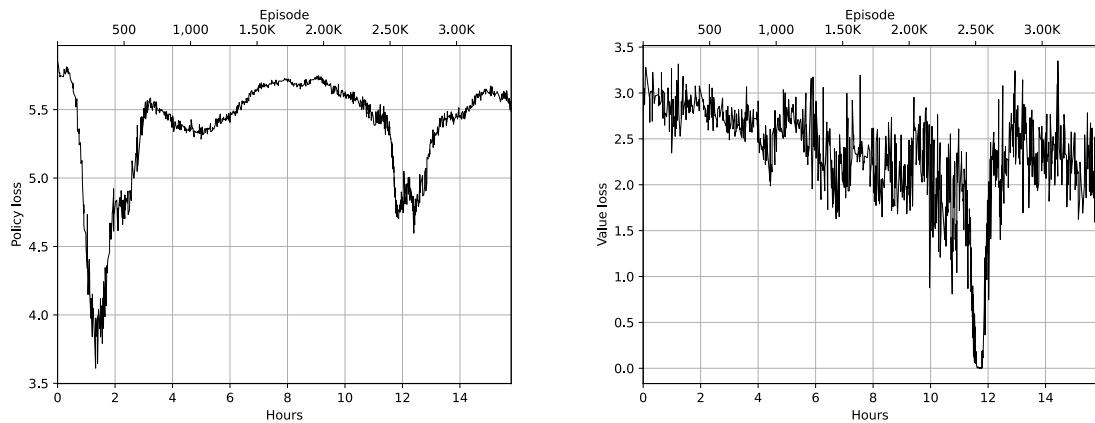


Figure 20.7: Policy (left) and value (right) losses for the MuZero training

During the training, almost 200 current best models were stored, which I checked in tournament mode using the `play-mu.py` script. Here are the top 10 models:

saves/mu-t5-6/best_010_00210.dat:	w=339, l=41, d=0
saves/mu-t5-6/best_015_00260.dat:	w=298, l=82, d=0
saves/mu-t5-6/best_155_02510.dat:	w=287, l=93, d=0
saves/mu-t5-6/best_150_02460.dat:	w=273, l=107, d=0
saves/mu-t5-6/best_140_02360.dat:	w=267, l=113, d=0
saves/mu-t5-6/best_145_02410.dat:	w=266, l=114, d=0
saves/mu-t5-6/best_165_02640.dat:	w=253, l=127, d=0
saves/mu-t5-6/best_005_00100.dat:	w=250, l=130, d=0
saves/mu-t5-6/best_160_02560.dat:	w=236, l=144, d=0
saves/mu-t5-6/best_135_02310.dat:	w=220, l=160, d=0

As you can see, the best models are models stored at the beginning of the training, which might be an indication of bad convergence (as I haven't tuned the hyperparameters much).

Figure 20.8 shows the plot with the model's winning ratio versus the model index, and this plot correlates a lot with policy loss, which is understandable because lower policy loss should lead to better gameplay:

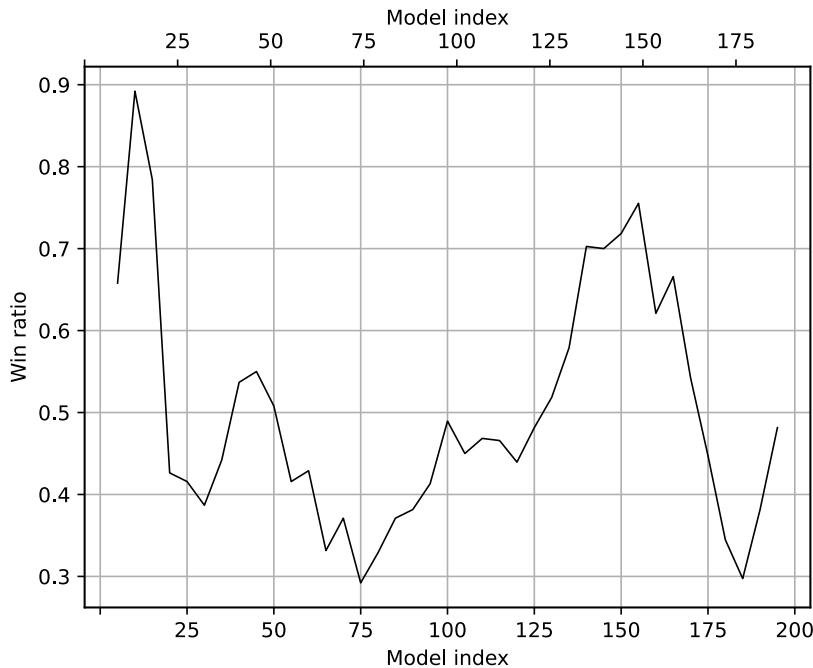


Figure 20.8: Winning ratio of the best models stored during the training

MuZero and Atari

In our example, we used Connect 4, which is a two-player board game, but we shouldn't miss the fact that MuZero's generalization (usage of hidden state) makes it possible to apply it to more classical RL scenarios. In the paper by Schrittwieser et al. [Sch+20], the authors successfully applied the method to 57 Atari games. Of course, the method requires tuning and adaptation to such scenarios, but the core is the same. This has been left as an exercise for you to try by yourself.

Summary

In this chapter, we implemented the AlphaGo Zero and MuZero model-based methods, which were created by DeepMind to solve board games. The primary point of this method is to allow agents to improve their strength via self-play, without any prior knowledge from human games or other data sources. This family of methods has real-world applications in several domains, such as healthcare (protein folding), finance, and energy management.

In the next chapter, we will discuss another direction of practical RL: discrete optimization problems, which play an important role in various real-life problems, from schedule optimization to protein folding.

Join our community on Discord

Read this book alongside other users, Deep Learning experts, and the author himself.

Ask questions, provide solutions to other readers, chat with the author via Ask Me Anything sessions, and much more.

Scan the QR code or visit the link to join the community.

<https://packt.link/rl>



21

RL in Discrete Optimization

The perception of deep **reinforcement learning (RL)** is that it is a tool to be used mostly for playing games. This is not surprising given the fact that, historically, the first success in the field was achieved on the Atari game suite by DeepMind in 2015 (<https://deepmind.com/research/dqn/>). The Atari benchmark suite turned out to be very successful for RL problems and, even now, lots of research papers use it to demonstrate the efficiency of their methods. As the RL field progresses, the classic 53 Atari games continue to become less and less challenging (at the time of writing, almost all the games have been solved with superhuman accuracy) and researchers are turning to more complex games, like StarCraft and Dota 2.

This perception, which is especially prevalent in the media, is something that I've tried to counterbalance in this book by accompanying Atari games with examples from other domains, including stock trading and **natural language processing (NLP)** problems, web navigation automation, continuous control, board games, and robotics. In fact, RL's very flexible **Markov decision process (MDP)** model potentially could be applied to a wide variety of domains; computer games are just one convenient and attention-grabbing example of complicated decision-making.

In this chapter, we will explore a new field in RL application: discrete optimization (which is a branch of mathematics that studies optimization problems on discrete structures), which will be showcased using the famous Rubik's cube puzzle. I've tried to provide a detailed description of the process followed in the paper titled *Solving the Rubik's cube without human knowledge*, by UCI researchers McAleer et al. [McA+18].

In addition, we will cover my implementation of the method described in the paper (which is in the `Chapter21` directory of the book's GitHub repository) and discuss directions to improve the method. I've combined the description of the paper's method with code pieces from my version to illustrate the concepts with concrete implementation.

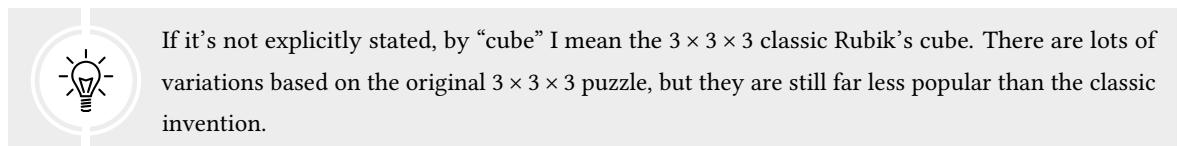
More specifically, in this chapter, we will:

- Briefly discuss the basics of discrete optimization
- Cover step by step the process followed by McAleer et al. [McA+18], who apply RL methods to the Rubik's cube optimization problem
- Explore experiments that I've done in an attempt to reproduce the paper's results and directions for future method improvement

Let's start with an overview of the Rubik's cube and discrete optimization in general.

The Rubik's cube and discrete optimization

I'm sure you are aware of what a Rubik's cube is, so I'm not going to go over the general description (https://en.wikipedia.org/wiki/Rubik%27s_Cube) of this puzzle, but rather focus on the connections it has to mathematics and computer science.



Despite being quite simple in terms of mechanics and the task at hand, the cube is quite a tricky object in terms of all the transformations we can make by possible rotations of its sides. It was calculated that in total, the cube has $\approx 4.33 \cdot 10^{19}$ distinct states reachable by rotating it. That's only the states that are reachable without disassembling the cube; by taking it apart and then assembling it, you can get 12 times more states in total: $\approx 5.19 \cdot 10^{20}$, but those "extra" states make the cube unsolvable without disassembling it.

All those states are quite intimately intertwined with each other through rotations of the cube's sides. For example, if we rotate the left side clockwise in some state, we get to the state from which rotation of the same side counterclockwise will destroy the effect of the transformation, and we will get into the original state.

But if we apply the left-side clockwise rotation three times in a row, the shortest path to the original state will be just a single rotation of the left side clockwise, but not three times counterclockwise (which is also possible, but just not optimal).

As the cube has 6 edges and each edge can be rotated in 2 directions, we have 12 possible rotations in total. Sometimes, half turns (which are two consecutive rotations in the same direction) are also included as distinct rotations, but for simplicity, we will treat them as two distinct transformations of the cube.

In mathematics, there are several areas that study objects of this kind. One of these is abstract algebra, a very broad division of math that studies abstract sets of objects with operations on top of them. In these terms, the Rubik's cube is an example of quite a complicated group (https://en.wikipedia.org/wiki/Group_theory) with lots of interesting properties.

The cube is not just states and transformations; it's a puzzle, with the primary goal to find a sequence of rotations with the solved cube as the end point. Problems of this kind are studied using combinatorial optimization, which is a subfield of applied math and theoretical computer science. This discipline has lots of famous problems of high practical value, for example:

- The traveling salesman problem (https://en.wikipedia.org/wiki/Travelling_salesman_problem): Finds the shortest closed path in a graph
- The protein folding simulation (https://en.wikipedia.org/wiki/Protein_folding): Finds possible 3D structures of protein
- Resource allocation: How to spread a fixed set of resources among consumers to get the best objective

What those problems have in common is a huge state space, which makes it infeasible to just check all possible combinations to find the best solution. Our “toy cube problem” also falls into the same category, because a state space of $4.33 \cdot 10^{19}$ makes a brute-force approach very impractical.

Optimality and God's number

What makes the combinatorial optimization problem tricky is that we're not looking for *any solution*; we're in fact interested in the *optimal solution* of the problem. So, what is the difference? Right after the Rubik's cube was invented, it was known how to reach the goal state (but it took Ernő Rubik about a month to figure out the first method of solving his own invention, which I expect was a frustrating experience). Nowadays, there are lots of different ways or *schemes* of cube solving: the beginner's method (layer by layer), the method by Jessica Fridrich (very popular among speedcubers), and so on.

All of them vary by the number of moves to be taken. For example, a very simple beginner's method requires about 100 rotations to solve the cube using just 5 ... 7 sequences of rotations to be memorized. In contrast, the current world record in the speedcubing competition is solving the cube in 3.13 seconds, which requires much fewer steps, but more sequences need to be memorized. The method by Fridrich requires about 55 moves on average, but you need to familiarize yourself with 120 different sequences of moves.

Of course, the big question is: what is the shortest sequence of actions to solve any given state of the cube? Surprisingly, after 50 years since the invention of the cube, humanity still doesn't know the full answer to this question. Only in 2010 did a group of researchers from Google prove that the minimum number of moves needed to solve *any* cube state is 20. This number is also known as *God's number* (not to be confused with the “golden ratio” or “divine proportion,” which is found everywhere in nature). Of course, on average, the optimal solution is shorter, as only a bunch of states require 20 moves and one single state doesn't require any moves at all (the solved state). This result only proves the minimal amount of moves; it does not find the solution itself. How to find the optimal solution for any given state is still an open question.

Approaches to cube solving

Before the paper by McAleer et al. was published, there were two major directions for solving the Rubik's cube:

- By using group theory, it is possible to significantly reduce the state space to be checked. One of the most popular solutions using this approach is Kociemba's algorithm (https://en.wikipedia.org/wiki/Optimal_solutions_for_Rubik%27s_Cube#Kociemba's_algorithm).
- By using brute-force search accompanied by manually crafted heuristics, we can direct the search in the most promising direction. A vivid example of this is Korf's algorithm (https://en.wikipedia.org/wiki/Optimal_solutions_for_Rubik%27s_Cube#Korf's_algorithm), which uses A* search with a large database of patterns to cut out bad directions.

McAleer et al. [McA+18] introduced a third approach (called autodidactic iteration, or ADI): by training the **neural network (NN)** on lots of randomly shuffled cubes, it is possible to get the policy that will show us the direction to take toward the solved state. The training is done without any prior knowledge about the domain; the only thing needed is the cube itself (not the physical one, but the computer model of it). This is in contrast with the two preceding methods, which require lots of human knowledge about the domain and labor to implement them in the form of computer code.

This method has lots of similarities to the AlphaGo Zero method we discussed in the previous chapter: we need a model of the environment and use **Monte Carlo tree search (MCTS)** to avoid full-state space exploration.

In the subsequent sections, we will take a detailed look at this approach; we'll start with data representation. In our cube problem, we have two entities that need to be encoded: actions and states.

Actions

Actions are possible rotations that we can do from any given cube state and, as has already been mentioned, we have only 12 actions in total. For every side, we have two different actions, corresponding to the clockwise and counterclockwise rotation of the side (90° or -90°). One small, but very important, detail is that a rotation is performed from the position when the desired side is facing toward you. This is obvious for the *front* side, for example, but for the *back* side, it might be confusing due to the mirroring of the rotation.

The names of the actions are taken from the cube sides that we're rotating: *left*, *right*, *top*, *bottom*, *front*, and *back*. The first letter of a side's name is used. For instance, the rotation of the *right* side clockwise is named as *R*. There are different notations for counterclockwise actions; sometimes they are denoted with an apostrophe (*R'*), with a lowercase letter (*r*), or even with a tilde (*Ŕ*). The first and last notations are not very practical in computer code, so in my implementation, I've used lowercase actions to denote counterclockwise rotations. For the right side, we have two actions: *R* and *r*, and we have another two for the left side: *L* and *l*, and so on.

In my code, the action space is implemented using a Python enum in `libcube/cubes/cube3x3.py`, in the `Action` class, where each action is mapped into the unique integer value:

```
class Action(enum.Enum):
    R = 0
    L = 1
    T = 2
    D = 3
    F = 4
    B = 5
    r = 6
    l = 7
    t = 8
    d = 9
    f = 10
    b = 11
```

In addition, we describe the dictionary with reverse actions:

```
_inverse_action = {
    Action.R: Action.r,    Action.r: Action.R,
    Action.L: Action.l,    Action.l: Action.L,
    Action.T: Action.t,    Action.t: Action.T,
    Action.D: Action.d,    Action.d: Action.D,
    Action.F: Action.f,    Action.f: Action.F,
    Action.B: Action.b,    Action.b: Action.B,
}
```

States

A state is a particular configuration of the cube's colored stickers and, as discussed earlier, the size of our state space is very large ($4.33 \cdot 10^{19}$ different states). But the number of states is not the only complication we have; in addition, we have different objectives that we would like to meet when we choose a particular representation of a state:

- **Avoid redundancy:** In the extreme case, we can represent a state of the cube by just recording the colors of every sticker on every side. But if we just count the number of such combinations, we get $6^{6 \cdot 8} = 6^{48} \approx 2.25 \cdot 10^{37}$, which is significantly larger than our cube's state space size, which just means that this representation is highly redundant; for example, it allows all sides of the cube to have one single color (except the center cubelets). If you're curious to know how I got $6^{6 \cdot 8}$, this is simple: we have six sides of a cube, each having eight small cubes (we're not counting centers), so we have 48 stickers in total and each of them could be colored in one of six colors.
- **Memory efficiency:** As you will see shortly, during the training and, even more so, during the model application, we will need to keep in our computer's memory a large amount of different states of the cube, which might influence the performance of the process. So, we would like the representation to be as compact as possible.
- **Performance of the transformations:** On the other hand, we need to implement all the actions applied to the state, and those actions need to be taken quickly. If our representation is very compact in terms of memory (uses bit-encoding, for example), but requires us to do a lengthy unpacking process for every rotation of the cube's side, our training might become too slow.
- **NN friendliness:** Not every data representation is equally as good as input for the NN. This is true not only for our case but for machine learning in general. For example, in NLP, it is common to use bag of words or word embeddings; in computer vision, images are decoded from JPEG into raw pixels; random forests require data to be heavily feature-engineered; and so on.

In the paper, every state of the cube is represented as a 20×24 tensor with one-hot encoding. To understand how this is done and why it has this shape, let's start with the picture taken from the paper shown in *Figure 21.1*:

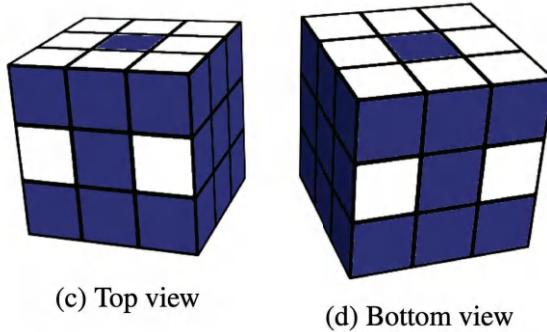


Figure 21.1: Stickers we need to track in the cube are marked in a lighter color

Here, the light color marks the stickers of the cubelets that we need to track; the rest of the stickers (shown in a darker color) are redundant and there is no need to track them. As you know, a cube consists of three types of cubelets: 8 corner cubelets with 3 stickers, 12 side cubelets with 2 stickers, and 6 central ones with a single sticker. It might not be obvious from first sight, but the central cubelets do not need to be tracked at all, as they can't change their relative position and can only rotate. So, in terms of the central cubelets, we need only to agree on the *cube alignment* (how the cube is oriented in space) and stick to it.

In my implementation, the white side is always on the top, the front is red, the left is green, and so on. This makes our state *rotation-invariant*, which basically means that all possible rotations of the cube as a whole are considered as the same state.

As the central cubelets are not tracked at all, on the figure, they are marked with the darker color. What about the rest? Obviously, every cubelet of a particular kind (corner or side) has a unique color combination of its stickers. For example, the assembled cube in my orientation (white on top, red on the front, and so on) has a top-left cubelet facing us with the following colors: green, white, and red. There are no other corner cubelets with those colors (please check in case of any doubt). The same is true for the side cubelets.

Due to this, to find the position of some particular cubelet, we need to know the position of only one of its stickers. The selection of such stickers is completely arbitrary, but once they are selected, you need to stick to this. As shown in the preceding figure, we track eight stickers from the top side, eight stickers from the bottom, and four additional side stickers: two on the front face and two on the back. This gives us 20 stickers to be tracked.

Now, let's discuss where 24 in the tensor dimension comes from. In total, we have 20 different stickers to track, but in which positions could they appear due to cube transformations? It depends on the kind of cubelet we're tracking. Let's start with corner cubelets. In total, there are eight corner cubelets and cube transformations can reshuffle them in any order. So, any particular cubelet could end up in any of eight possible corners.

In addition, every corner cubelet could be rotated, so our “green, white, and red” cubelet could end up in three possible orientations:

- White on top, green left, and red front
- Green on top, red left, and white front
- Red on top, white left, and green front

So, to precisely indicate the position and orientation of the corner cubelet, we have $8 \times 3 = 24$ different combinations. In the case of the 12-side cubelets, they have only two stickers, so there are only two orientations possible, which, again, gives us 24 combinations, but they are obtained from a different calculation: $12 \times 2 = 24$. Finally, we have 20 cubelets to be tracked, 8 corners and 12 sides, each having 24 positions that it could end up in.

A very popular option to feed such data into an NN is one-hot encoding, when the concrete position of the object has 1, with other positions filled with 0. This gives us the final representation of the state as a tensor with the shape 20×24 .

From a redundancy point of view, this representation is much closer to the total state space; the amount of possible combinations equals $24^{20} \approx 4.02 \cdot 10^{27}$. It is still larger than the cube state space (it could be said that it is *significantly* larger, as the factor of 10^8 is a lot), but it is better than encoding all the colors of every sticker. This redundancy comes from tricky properties of cube transformations; for example, it is not possible to rotate one single corner cubelet (or flip one side cubelet) *leaving all others in their places*. Mathematical properties are well beyond the scope of this book, but if you’re interested, I recommend the wonderful book by Alexander Frey and David Singmaster called *Handbook of Cubik Math* [FS20].

You might have noticed that the tensor representation of the cube state has one significant drawback: memory inefficiency. Indeed, by keeping the state as a floating-point tensor of 20×24 , we’re using $4 \times 20 \times 24 = 1,920$ bytes of memory, which is a lot given the requirement to keep thousands of states during the training process and millions of them during the cube solving (as you will get to know shortly). To overcome this, in my implementation, I used two representations: one tensor is intended for NN input and another, more compact, representation is needed to store different states for longer. This compact state is saved as a bunch of lists, encoding the permutations of corner and side cubelets, and their orientation. This representation is not only much more memory efficient (160 bytes) but also much more convenient for transformation implementation.

To illustrate this, what follows is the piece of the cube 3×3 library, `libcube/cubes/cube3x3.py`, which is responsible for compact representation.

The variable `initial_state` is the encoding of the solved state of the cube. In it, corner and side stickers that we're tracking are in their original positions, and both orientation lists are set to 0, indicating the initial orientation of the cubelets:

```
State = collections.namedtuple("State", field_names=[  
    'corner_pos', 'side_pos', 'corner_ort', 'side_ort'])  
  
initial_state = State(corner_pos=tuple(range(8)), side_pos=tuple(range(12)),  
                      corner_ort=tuple([0]*8), side_ort=tuple([0]*12))
```

The transformation of the cube is a bit complex and includes lots of tables holding the rearrangements of cubelets after different rotations are applied. I'm not going to put this code here; if you're curious, you can start with the function `transform(state, action)` in `libcube/cubes/cube3x3.py`. It might also be helpful to check unit tests of this code.

Besides the actions and compact state representation and transformation, the module `cube3x3.py` includes a function that converts the compact representation of the cube state (as the `State` named tuple) into the tensor form. This functionality is provided by the `encode_inplace()` method.

Another functionality implemented is the ability to render the compact state into human-friendly form by applying the `render()` function. It is very useful for debugging the transformation of the cube, but it's not used in the training code.

The training process

Now that you know how the state of the cube is encoded in a 20×24 tensor, let's explore the NN architecture and understand how it is trained.

The NN architecture

Figure 21.2, from the paper by McAleer et al., shows the network architecture:

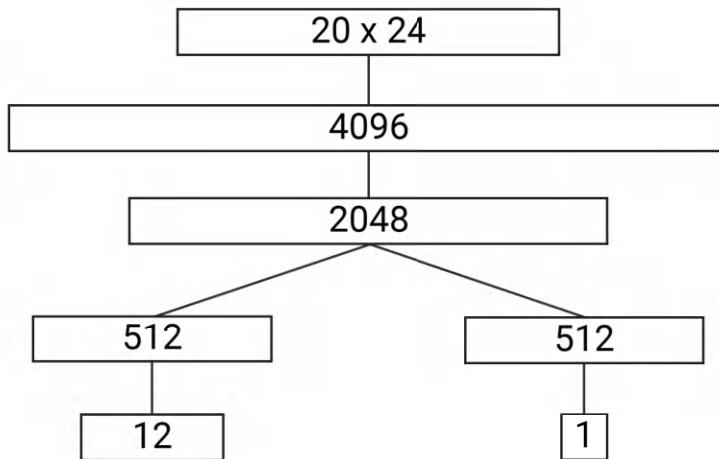


Figure 21.2: The NN architecture transforming the observation (top) to the action and value (bottom)

As the input, it accepts the already familiar cube state representation as a 20×24 tensor and produces two outputs:

- The policy, which is a vector of 12 numbers, representing the probability distribution over our actions.
- The value, a single scalar estimating the “goodness” of the state passed. The concrete meaning of a value will be discussed in the next section.

In my implementation, the architecture is exactly the same as in the paper, and the model is in the module `libcube/model.py`. Between the input and output, the network has several fully connected layers with **exponential linear unit (ELU)** activations, as discussed in the paper:

```

class Net(nn.Module):
    def __init__(self, input_shape, actions_count):
        super(Net, self).__init__()

        self.input_size = int(np.prod(input_shape))
        self.body = nn.Sequential(
            nn.Linear(self.input_size, 4096),
            nn.ELU(),
            nn.Linear(4096, 2048),
            nn.ELU()
        )
  
```

```

        )
        self.policy = nn.Sequential(
            nn.Linear(2048, 512),
            nn.ELU(),
            nn.Linear(512, actions_count)
        )
        self.value = nn.Sequential(
            nn.Linear(2048, 512),
            nn.ELU(),
            nn.Linear(512, 1)
        )

    def forward(self, batch, value_only=False):
        x = batch.view((-1, self.input_size))
        body_out = self.body(x)
        value_out = self.value(body_out)
        if value_only:
            return value_out
        policy_out = self.policy(body_out)
        return policy_out, value_out

```

The `forward()` call can be used in two modes: to get both the policy and the value, or whenever `value_only=True`, only the value. This saves us some computations in the case when only the value head's result is of interest.

The training

In this network, the policy tells us what transformation we should apply to the state, and the value estimates how good the state is. But the big question still remains: how do we train the network?

As discussed earlier, the training method proposed in the paper is called **autodidactic iterations (ADI)**. Let's look at its structure. We start with the goal state (the assembled cube) and apply the sequence of random transformations of some predefined length, N . This gives us a sequence of N states.

For each state, s , in this sequence, we carry out the following procedure:

1. Apply every possible transformation (12 in total) to s .
2. Pass those 12 states to our current NN, asking for the value output. This gives us 12 values for every substate of s .
3. The target value for s is calculated as $y_{v_i} = \max_a(v_s(a) + R(A(s, a)))$, where $A(s, a)$ is the state after the action, a , is applied to s and $R(s)$ equals 1 if s is the goal state and -1 otherwise.
4. The target policy for s is calculated using the same formula, but instead of max, we take argmax: $y_{p_i} = \arg \max_a(v_s(a) + R(A(s, a)))$. This just means that our target policy will have 1 at the position of the maximum value for the substate and 0 on all other positions.

This process is shown in *Figure 21.3*, taken from the paper. The sequence of scrambles, x_0, x_1, \dots, x_N , is generated, where the cube, x_i , is shown expanded. For this state, x_i , we make targets for the policy and value heads from the expanded states by applying the preceding formulas.

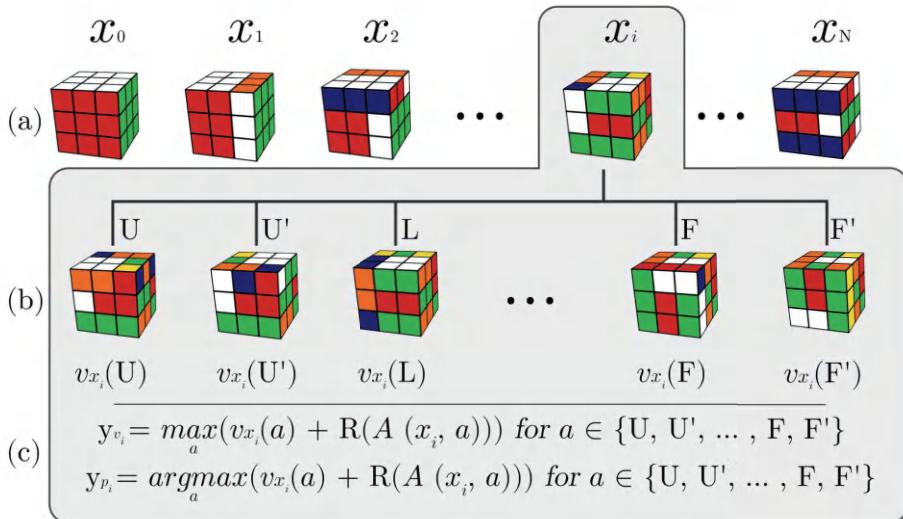


Figure 21.3: Data generation for training

Using this process, we can generate any amounts of training data that we want.

The model application

Okay, imagine that we have trained the model using the process just described. How should we use it to solve the scrambled cube? From the network's structure, you might imagine the obvious, but not very successful, way:

1. Feed the model the current state of the cube that we want to solve.
2. From the policy head, get the largest action to perform (or sample it from the resulting distribution).
3. Apply the action to the cube.
4. Repeat the process until the solved state has been reached.

On paper, this method should work, but in practice, it has one serious issue: it doesn't! The main reason for that is our model's quality. Due to the size of the state space and the nature of the NNs, it just isn't possible to train an NN to return the exact optimal action for *any* input state all of the time. Rather than telling us what to do to get the solved state, our model shows us promising directions to explore.

Those directions could bring us closer to the solution, but sometimes they could be misleading, just from the fact that this particular state has never been seen during the training. Don't forget, there are $4.33 \cdot 10^{19}$ of them, so even with a **graphics processing unit (GPU)** training speed of hundreds of thousands of states per second, and after a month of training, we will only see a tiny portion of the state space, about 0.0000005%. So, a more sophisticated approach has to be used.

There is a family of very popular methods, called MCTS, and one of these methods was covered in the last chapter. There are lots of variants of those methods, but the overall idea can be described in comparison with the well-known brute-force search methods, like **breadth-first search (BFS)** or **depth-first search (DFS)**. In BFS and DFS, we perform an exhaustive search of our state space by trying all the possible actions and exploring all the states that we get from those actions. That behavior is the other extreme of the procedure described previously (when we have something that tells us where to go at every state). But MCTS offers something in between those extremes: we want to perform the search and we have some information about where we should go, but this information could be unreliable, noisy, or just wrong in some situations. However, sometimes, this information could show us the promising directions that could speed up the search process.

As I've mentioned, MCTS is a family of methods and they vary in their particular details and characteristics. In the paper, a method called **Upper Confidence Bound 1** is used. This method operates on the tree, where the nodes are the states and the edges are actions connecting those states. The whole tree is enormous in most cases, so we can't try to build the whole tree, just some tiny portion of it.

In the beginning, we start with a tree consisting of a single node, which is our current state. At every step of the MCTS, we walk down the tree, exploring some path in the tree, and there are two options we can face:

- Our current node is a leaf node (we haven't explored this direction yet)
- Our current node is in the middle of the tree and has children

In the case of a leaf node, we "expand" it by applying all the possible actions to the state. All the resulting states are checked for being the goal state (if the goal state of the solved cube has been found, our search is done). The leaf state is passed to the model and the outputs from both the value and policy heads are stored for later use.

If the node is not the leaf, we know about its children (reachable states), and we have value and policy outputs from the network. So, we need to make the decision about which path to follow (in other words, which action is more promising to explore). This decision is not a trivial one and this is the exploration versus exploitation problem that we have covered previously in this book. On the one hand, our policy from the network says what to do. But what if it is wrong? This could be solved by exploring surrounding states, but we don't want to explore all the time (as the state space is enormous). So, we should keep the balance, and this has a direct influence on the performance and the outcome of the search process.

To solve this, for every state, we keep the counter for every possible action (there are 12 of them), which is incremented every time the action has been chosen during the search. To make the decision to follow a particular action, we use this counter; the more an action has been taken, the less likely it is to be chosen in the future.

In addition, the value returned by the model is also used in this decision-making. The value is tracked as the maximum from the current state's value and the value from its children. This allows the most promising paths (from the model perspective) to be seen from the parent's states.

To summarize, the action to follow from a non-leaf tree is chosen by using the following formula:

$$A_t = \arg \max_a (U_{s_t}(a) + W_{s_t}(a)), \text{ where}$$

$$U_{s_t}(a) = cP_{s_t}(a) \frac{\sqrt{\sum_{a'} N_{s_t}(a')}}{1 + N_{s_t}(a)}$$

Here, $N_{s_t}(a)$ is a count of times that action a has been chosen in state s_t . $P_{s_t}(a)$ is the policy returned by the model for state s_t and $W_{s_t}(a)$ is the maximum value returned by the model for all children states of s_t under the branch a .

This procedure is repeated until the solution has been found or our time budget has been exhausted. To speed up the process, MCTS is very frequently implemented in a parallel way, where several searches are performed by multiple threads. In that case, some extra loss could be subtracted from A_t to prevent multiple threads from exploring the same paths of the tree.

The final piece in solving the process puzzle is how to get the solution from the MCTS tree once we have reached the goal state. The authors of the paper experimented with two approaches:

- **Naïve:** Once we have faced the goal state, we use our path from the root state as the solution
- **The BFS way:** After reaching the goal state, BFS is performed on the MCTS tree to find the shortest path from the root to this state

According to the authors, the second method finds shorter solutions than the naïve version, which is not surprising, as the stochastic nature of the MCTS process can introduce cycles to the solution path.

Results

The final result published in the paper is quite impressive. After 44 hours of training on a machine with three GPUs, the network learned how to solve cubes at the same level as (and sometimes better than) human-crafted solvers. The final model has been compared against the two solvers described earlier: the Kociemba two-stage solver and Korf. The method proposed in the paper is named DeepCube.

To compare efficiency, 640 randomly scrambled cubes were used in all the methods. The depth of the scramble was 1,000 moves. The time limit for the solution was an hour and both the DeepCube and Kociemba solvers were able to solve all of the cubes within the limit. The Kociemba solver is very fast, and its median solution time is just one second, but due to the hardcoded rules implemented in the method, its solutions are not always the shortest ones.

The DeepCube method took much more time, with the median time being about 10 minutes, but it was able to match the length of the Kociemba solutions or do better in 55% of cases. From my personal perspective, 55% is not enough to say that NNs are significantly better, but at least they are not worse.

In *Figure 21.4*, taken from the paper, the length distributions for all the solvers are shown. As you can see, the Korf solver wasn't compared in 1,000 scramble test cases, due to the very long time needed to solve the cube. To compare the performance of DeepCube against the Korf solver, a much easier 15-step scramble test set was created:

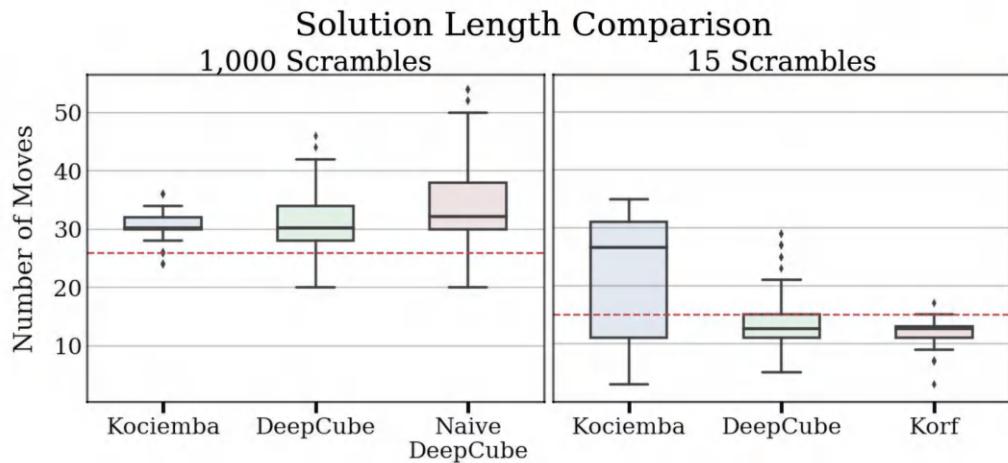


Figure 21.4: The length of solutions found by various solvers

The code outline

Now that you have some context, let's switch to the code, which is in the `Chapter21` directory in the book's GitHub repository. In this section, I'm going to give a quick outline of my implementation and the key design decisions, but before that, I have to emphasize the important points about the code to set up the correct expectations:

- I'm not a researcher, so the original goal of this code was just to reimplement the paper's method. Unfortunately, the paper has very few details about the exact hyperparameters used, so I had to experiment a lot, and still, my results are very different from those published in the paper.
- At the same time, I've tried to implement everything in a general way to simplify further experiments. For example, the exact details about the cube state and transformations are abstracted away, which allows us to implement more puzzles similar to the 3×3 cube just by adding a new module. In my code, two cubes are implemented: 2×2 and 3×3 , but any fully observable environment with a fixed set of predictable actions can be implemented and experimented with. The details are given later in this section (in the *Cube environments* subsection).
- Code clarity and simplicity were put ahead of performance. Of course, when it was possible to improve performance without introducing much overhead, I did so. For example, the training process was sped up by a factor of five by just splitting the generation of the scrambled cubes and the forward network pass. But if the performance required refactoring everything into multi-GPU and multithreaded mode, I preferred to keep things simple. A very clear example is the MCTS process, which is normally implemented as multithreaded code sharing the tree. It usually gets sped up several times, but requires tricky synchronization between processes. So, my version of MCTS is serial, with only trivial optimization of the batched search.

Overall, the code consists of the following parts:

1. The cube environment, which defines the observation space, the possible actions, and the exact representation of the state to the network. This part is implemented in the `libcube/cubes` module.
2. The NN part, which describes the model that we will train, the generation of training samples, and the training loop. It includes the training tool `train.py` and the module `libcube/model.py`.
3. The solver of cubes or the search process, including the `solver.py` utility and the `libcube/mcts.py` module, which implements MCTS.
4. Various tools used to glue up other parts, like configuration files with hyperparameters and tools used to generate cube problem sets.

Cube environments

As you have already seen, combinatorial optimization problems are quite large and diverse. Even the narrow area of cube-like puzzles includes a couple of dozen variations. The most popular ones are $2 \times 2 \times 2$, $3 \times 3 \times 3$, and $4 \times 4 \times 4$ Rubik's cubes, Square-1, and Pyraminx (<https://ruwix.com/twisty-puzzles/>). At the same time, the method presented in the paper is quite general and doesn't depend on prior domain knowledge, the amount of actions, and the state space size. The critical assumptions imposed on the problem include:

- States of the environment need to be fully observable and observations need to distinguish states from each other. That's the case for the cube when we can see all the sides' states, but it doesn't hold true for most variants of poker, for example, when we can't see the cards of our opponent.
- The number of actions needs to be discrete and finite. There is a limited number of actions we can take with the cube, but if our action space is "rotate the steering wheel on angle $\alpha \in [-120^\circ \dots 120^\circ]$," we have a different problem domain here, as you have already seen in chapters devoted to the continuous control problems.
- We need to have a reliable model of the environment; in other words, we have to be able to answer questions like "What will be the result of applying action a_i to the state s_j ?" Without this, both ADI and MCTS become non-applicable. This is a strong requirement and, for most problems, we don't have such a model or its outputs are quite noisy. On the other hand, in games like chess or Go, we have such a model: the rules of the game.

At the same time, as we've seen in the previous chapter (about the MuZero method), you can approximate the model with neural networks, but paying the price of lower performance.

- In addition, our domain is deterministic, as the same action applied to the same state always ends up in the same final state. The counter example might be backgammon, when, on each turn, players roll the dice to get the amount of moves they can possibly make. Most likely, this method could be generalized to this case as well.

To simplify the application of the methods to domains different from the 3×3 cube, all concrete environment details are moved to separate modules, communicating with the rest of the code via the abstract interface `CubeEnv`, which is described in the `libcube/cubes/_env.py` module. Let's go through its interface.

As shown in the following code snippet, the constructor of the class takes a bunch of arguments:

- The name of the environment.
- The type of the environment state.
- The instance of the initial (assembled) state of the cube.

- The predicate function to check that a particular state represents the assembled cube. For 3×3 cubes, this might look like an overhead, as we possibly could just compare this with the initial state passed in the `initial_state` argument, but cubes of size 2×2 and 4×4 , for instance, might have multiple final states, so a separate predicate is needed to cover such cases.
- The enumeration of actions that we can apply to the state.
- The transformation function, which takes the state and the action and returns the resulting state.
- The inverse function, which maps every action into its inverse.
- The render function to represent the state in human-readable form.
- The shape of the encoded state tensor.
- The function to encode the compact state representation into an NN-friendly form.

```
class CubeEnv:
    def __init__(self, name, state_type, initial_state, is_goal_pred,
                 action_enum, transform_func, inverse_action_func,
                 render_func, encoded_shape, encode_func):
        self.name = name
        self._state_type = state_type
        self.initial_state = initial_state
        self._is_goal_pred = is_goal_pred
        self.action_enum = action_enum
        self._transform_func = transform_func
        self._inverse_action_func = inverse_action_func
        self._render_func = render_func
        self.encoded_shape = encoded_shape
        self._encode_func = encode_func
```

As you can see, cube environments are not compatible with the Gym API; I used this example intentionally to illustrate how you can step beyond Gym.

Some of the methods in the `CubeEnv` API are just wrappers around functions passed to the constructor. This allows the new environment to be implemented in a separate module, register itself in the environment registry, and provide a consistent interface to the rest of the code:

```
def __repr__(self):
    return "CubeEnv(%r)" % self.name

def is_goal(self, state):
    assert isinstance(state, self._state_type)
    return self._is_goal_pred(state)

def transform(self, state, action):
```

```

    assert isinstance(state, self._state_type)
    assert isinstance(action, self.action_enum)
    return self._transform_func(state, action)

def inverse_action(self, action):
    return self._inverse_action_func(action)

def render(self, state):
    assert isinstance(state, self._state_type)
    return self._render_func(state)

def encode_inplace(self, target, state):
    assert isinstance(state, self._state_type)
    return self._encode_func(target, state)

```

All the other methods in the class provide extended uniform functionality based on those primitive operations.

The `sample_action()` method provides the functionality of randomly sampling the random action. If the `prev_action` argument is passed, we exclude the reverse action from possible results, which is handy to avoid a generation of short loops, like $R \rightarrow r$ or $L \rightarrow l$, for example:

```

def sample_action(self, prev_action=None):
    while True:
        res = self.action_enum(random.randrange(len(self.action_enum)))
        if prev_action is None or self.inverse_action(res) != prev_action:
            return res

```

The method `scramble()` applies the list of actions (passed as an argument) to the initial state of the cube, returning the final state:

```

def scramble(self, actions):
    s = self.initial_state
    for action in actions:
        s = self.transform(s, action)
    return s

```

The method `scramble_cube()` provides the functionality of randomly scrambling the cube, returning all the intermediate states. In the case of the `return_inverse` argument being `False`, the function returns the list of tuples with `(depth, state)` for every step of the scrambling process. If the argument is `True`, it returns a tuple with three values: `(depth, state, inv_action)`, which are needed in some situations:

```

def scramble_cube(self, scrambles_count, return_inverse=False,
include_initial=False):
    assert isinstance(scrambles_count, int)
    assert scrambles_count > 0

    state = self.initial_state
    result = []
    if include_initial:
        assert not return_inverse
        result.append((1, state))
    prev_action = None
    for depth in range(scrambles_count):
        action = self.sample_action(prev_action=prev_action)
        state = self.transform(state, action)
        prev_action = action
        if return_inverse:
            inv_action = self.inverse_action(action)
            res = (depth+1, state, inv_action)
        else:
            res = (depth+1, state)
        result.append(res)
    return result

```

The method `explore_states()` implements functionality for ADI and applies all the possible actions to the given cube state. The result is a tuple of lists in which the first list contains the expanded states, and the second has flags of those states as the goal state:

```

def explore_state(self, state):
    res_states, res_flags = [], []
    for action in self.action_enum:
        new_state = self.transform(state, action)
        is_init = self.is_goal(new_state)
        res_states.append(new_state)
        res_flags.append(is_init)
    return res_states, res_flags

```

With this generic functionality, a similar environment might be implemented and plugged into the existing training and testing methods with very little boilerplate code. As an example, I have provided both the $2 \times 2 \times 2$ cube and the $3 \times 3 \times 3$ cube that I used in my experiments. Their internals reside in `libcube/cubes/cube2x2.py` and `libcube/cubes/cube3x3.py`, which you can use as a base to implement your own environments of this kind.

Every environment needs to register itself by creating the instance of the `CubeEnv` class and passing the instance into the function `register()`, defined in `libcube/cubes/_env.py`. The following is the relevant piece of code from the `cube2x2.py` module:

```
_env.register(_env.CubeEnv(name="cube2x2", state_type=State, initial_state=initial_state,
                           is_goal_pred=is_initial, action_enum=Action,
                           transform_func=transform, inverse_action_func=inverse_action,
                           render_func=render, encoded_shape=encoded_shape,
                           encode_func=encode_inplace))
```

Once this is done, the cube environment can be obtained by using the `libcube.cubes.get()` method, which takes the environment name as an argument. The rest of the code uses only the public interface of the `CubeEnv` class, which makes the code cube-type agnostic and simplifies the extensibility.

Training

The training process is implemented in the tool `train.py` and the module `libcube/model.py`, and it is a straightforward implementation of the training process described in the paper, with one difference: the code supports two methods of calculating the target values for the value head of the network. One of the methods is exactly how it was described in the paper and the other is my modification, which I'll explain in detail in the subsequent section.

To simplify the experimentation and make the results reproducible, all the parameters of the training are specified in a separate `.ini` file, which gives the following options for training:

- The name of the environment to be used; currently, `cube2x2` and `cube3x3` are available.
- The name of the run, which is used in TensorBoard names and directories to save models.
- What target value calculation method in ADI will be used. I implemented two of them: one is described in the paper and then there is my modification, which, from my experiments, has more stable convergence.
- The training parameters: the batch size, the usage of CUDA, the learning rate, the learning rate decay, and others.

You can find the examples of my experiments in the `ini` folder in the repo. During the training, TensorBoard metrics of the parameters are written in the `runs` folder. Models with the best loss value are saved in the `saves` directory.

To give you an idea of what the configuration file looks like, the following is `ini/cube2x2-paper-d200.ini`, which defines the experiment for a 2×2 cube, using the value calculation method from the paper and a scramble depth of 200:

```
[general]
cube_type=cube2x2
run_name=paper

[train]
cuda=True
lr=1e-5
batch_size=10000
scramble_depth=200
report_batches=10
checkpoint_batches=100
lr_decay=True
lr_decay_gamma=0.95
lr_decay_batches=1000
```

To start the training, you need to pass the .ini file to the `train.py` utility; for example, this is how the preceding .ini file could be used to train the model:

```
$ ./train.py -i ini/cube2x2-paper-d200.ini -n t1
```

The extra argument `-n` gives the name of the run, which will be combined with the name in the .ini file to be used as the name of a TensorBoard series.

The search process

The result of the training is a model file with the network's weights. The file could be used to solve cubes using MCTS, which is implemented in the tool `solver.py` and the module `libcube/mcts.py`.

The solver tool is quite flexible and could be used in various modes:

1. To solve a single scrambled cube given as a comma-separated list of action indices, passed in the `-p` option. For example, `-p 1,6,1` is a cube scrambled by applying the second action, then the seventh action, and finally, the second action again. The concrete meaning of the actions is environment-specific, which is passed with the `-e` option. You can find actions with their indices in the cube environment module. For example, the actions `1,6,1` for a 2×2 cube mean an $L \rightarrow R' \rightarrow L$ transformation.
2. To read permutations from a text file (one cube per line) and solve them. The file name is passed with the `-i` option. There are several sample problems available in the folder `cubes_tests`. You can generate your own random problem sets using the `gen_cubes.py` tool, which allows you to set the random seed, the depth of the scramble, and other options.
3. To generate a random scramble of the given depth and solve it.

4. To run a series of tests with increasing complexity (scramble depth), solve them, and write a CSV file with the result. This mode is enabled by passing the `-o` option and is very useful for evaluating the quality of the trained model, but it can take lots of time to complete. Optionally, plots with those test results are produced.

In all cases, you need to pass the environment name with the `-e` option and the file with the model's weights (the `-m` option). In addition, there are other parameters, allowing you to tweak MCTS options and time or search step limits. You can find the names of those options in the code of `solver.py`.

The experiment results

Unfortunately, the paper provided no details about very important aspects of the method, like training hyperparameters, how deeply cubes were scrambled during the training, and the obtained convergence. To fill in the missing blanks, I experimented with various values of hyperparameters (`.ini` files are available in the GitHub repo), but still my results are very different from those published in the paper. I observed that the training convergence of the original method is very unstable. Even with a small learning rate and a large batch size, the training eventually diverges, with the value loss component growing exponentially. Examples of this behavior are shown in *Figure 21.5* and *Figure 21.6* (obtained from the 2×2 environment):

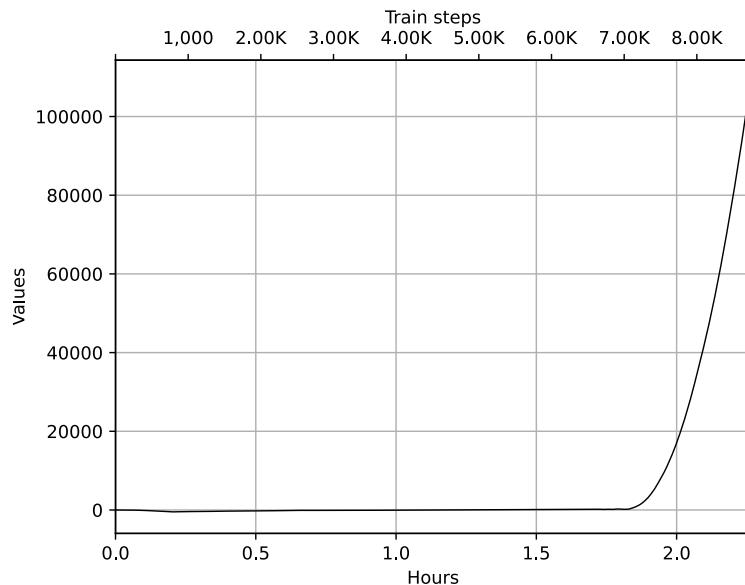


Figure 21.5: Values predicted by the value head during training on the paper's method

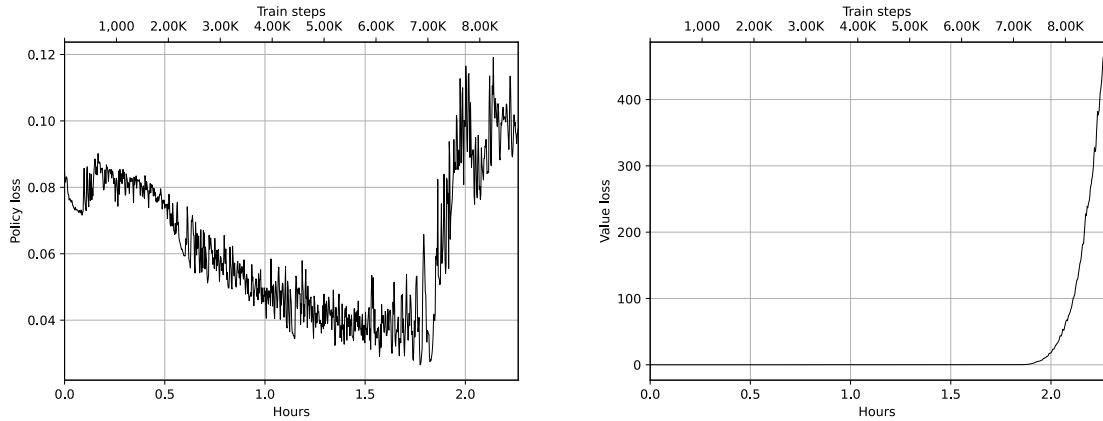


Figure 21.6: The policy loss (left) and value loss (right) during the typical run of the paper’s method

After several experiments with this problem, I came to the conclusion that this behavior is a result of the wrong value objective being proposed in the method. Indeed, in the formula $y_{v_i} = \max_a(v_s(a) + R(A(s, a)))$, the value $v_s(a)$ returned by the network is always added to the actual reward, $R(s)$, even for the goal state. With this, the actual values returned by the network could be anything: -100 , 10^6 , or 3.1415 . This is not a great situation for NN training, especially with the **mean squared error (MSE)** objective.

To check this, I modified the method of the target value calculation by assigning a 0 target for the goal state:

$$y_{v_i} = \begin{cases} \max_a(v_s(a) + R(A(s, a))) & \text{if } s \text{ is not the goal} \\ 0 & \text{if } s \text{ is the goal state} \end{cases}$$

This target could be enabled in the `.ini` file by specifying the parameter `value_targets_method` to be `zero_goal_value`, instead of the default `value_targets_method=paper`.

With this simple modification, the training process converged much quicker to stable values returned by the value head of the network. An example of this convergence is shown in *Figure 21.7* and *Figure 21.8*:

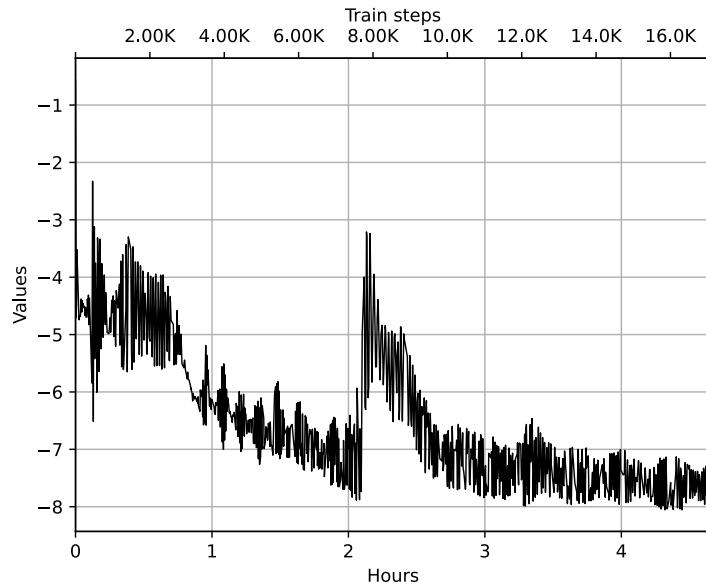


Figure 21.7: Values predicted by the value head during training

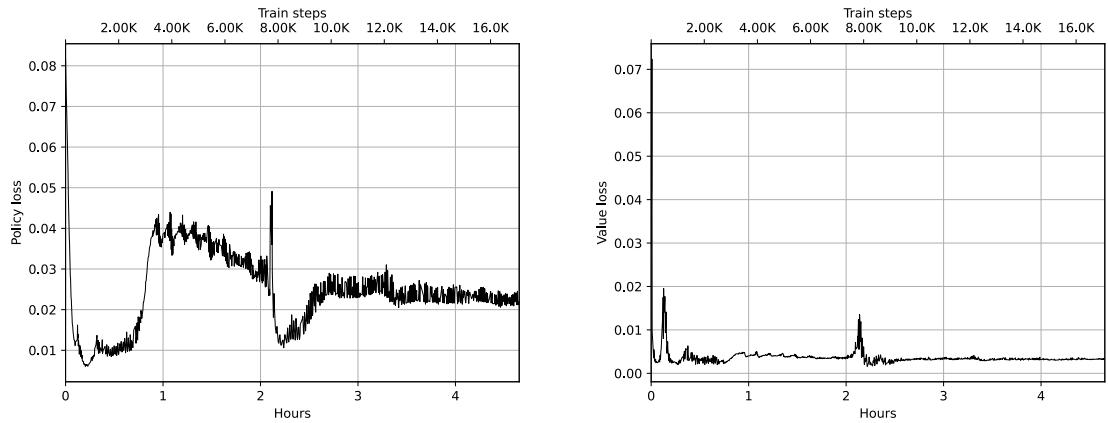


Figure 21.8: The policy loss (left) and value loss (right) after modifications in value calculation

The 2×2 cube

In the paper, the authors reported training for 44 hours on a machine with three Titan Xp GPUs. During the training, their model saw 8 billion cube states. Those numbers correspond to the training speed 50,000 cubes/second. My implementation shows 15,000 cubes/second on a single GTX 1080 Ti, which is comparable.

So, to repeat the training process on a single GPU, we need to wait for almost six days, which is not very practical for experimentation and hyperparameter tuning.

To overcome this, I implemented a much simpler 2×2 cube environment, which takes just an hour or two to train. To reproduce my training, there are two .ini files in the repo:

- `ini/cube2x2-paper-d200.ini`: This uses the value target method described in the paper
- `ini/cube2x2-zero-goal-d200.ini`: The value target is set to 0 for goal states

Both configurations use batches of 10k states and a scramble depth of 200, and the training parameters are the same. After the training, using both configurations, two models were produced:

- The paper's method: loss 0.032572
- The zero-goal method: loss 0.012226

To perform a fair comparison, I generated 20 test scrambles for depths 1 ... 50 (1,000 test cubes in total), which are available in `cubes_test/3ed`, and ran the `solver.py` utility on the best model produced by each method. For every test scramble, the limit for searches was set to 30,000. This utility produced CSV files (available in `csvs/3ed`) with details about every test outcome.

My experiments have shown that the model described in the paper was able to solve 55% of test cubes, while the model with zero-goal modification solved 100%. The results for both models depending on scramble depth are shown in *Figure 21.9*. On the left plot, the ratio of solved cubes is shown. On the right plot, the average MCTS search steps per scramble depth are displayed. As you can see, the modified version requires significantly (3x-5x) fewer MCTS searches to find a solution, so the learned policy is better.

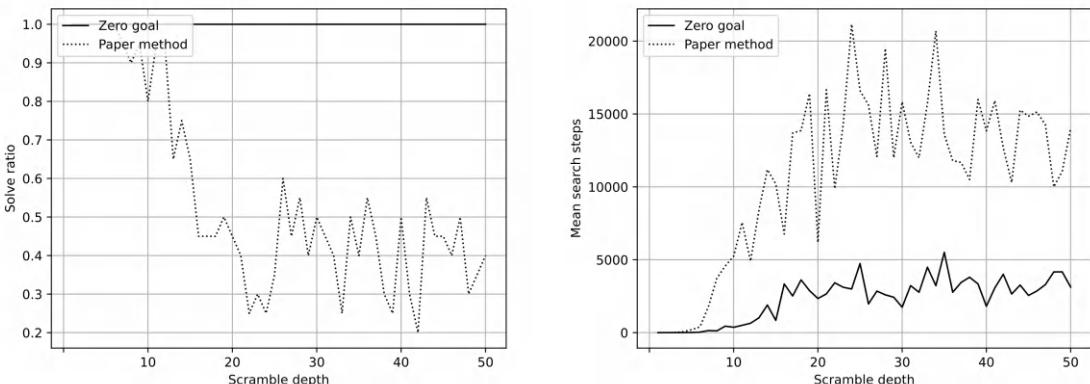


Figure 21.9: The ratio of solved 2×2 cubes (left) and average count of MCTS searches needed for various scramble depths

Finally, let's check the length of the found solutions. In *Figure 21.10*, both the naïve and BFS solution lengths are plotted. From those plots, it can be seen that the naïve solutions are much longer (by a factor of 10) than solutions found by BFS. Such a difference might be an indication of untuned MCTS parameters, which could be improved. In naïve solutions, the zero goal finds shorter solutions (which might again be an indication of a better policy).

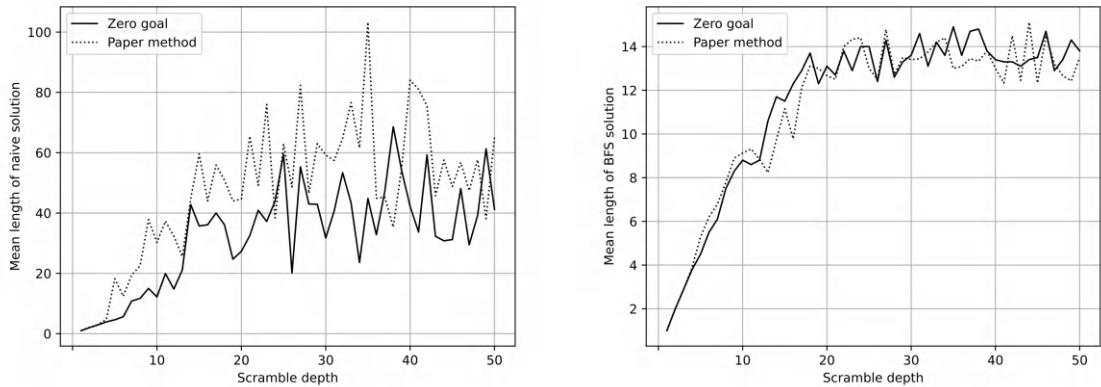


Figure 21.10: A comparison of naïve (left) and BFS (right) ways of solution generation

The 3×3 cube

The training of the 3×3 cube model is much more heavy; we've just scratched the surface here. But my limited experiments show that zero-goal modifications to the training method greatly improve the training stability and resulting model quality. Training requires about 20 hours, so running lots of experiments requires time and patience.

My results are not as shiny as those reported in the paper: the best model I was able to obtain can solve cubes up to a scrambling depth of 12 ... 15, but consistently fails at more complicated problems. Probably, those numbers could be improved with more **central processing unit (CPU)** cores plus parallel MCTS. To get the data, the search process was limited to 100k steps and, for every scramble depth, five random scrambles were generated (available in `cubes_tests/3ed` in the repo). But again, the modified version shows better results – the model trained using the paper's method was able to solve only problems with a scramble depth of 9, but the modified version was able to reach a depth of 13.

Figure 21.11 shows a comparison of solution rates (left plot) for the method presented in the paper and the modified version with the zero-value target. On the right part of the figure, the average number of MCTS searches is shown.

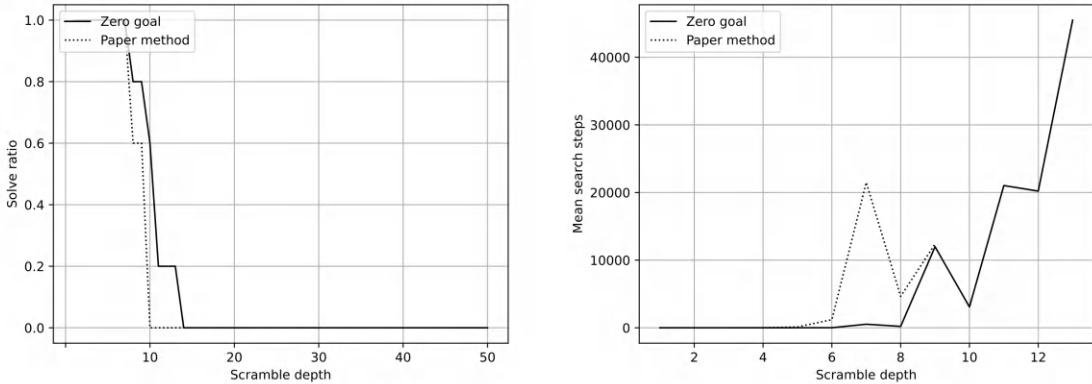


Figure 21.11: The ratio of solved 3×3 cubes by both methods (left) and the average number of MCTS searches

Figure 21.12 shows the length of the optimal solution found. As before, naïve search produces longer results than the BFS-optimized one. The BFS length almost perfectly aligned with the scramble depth:

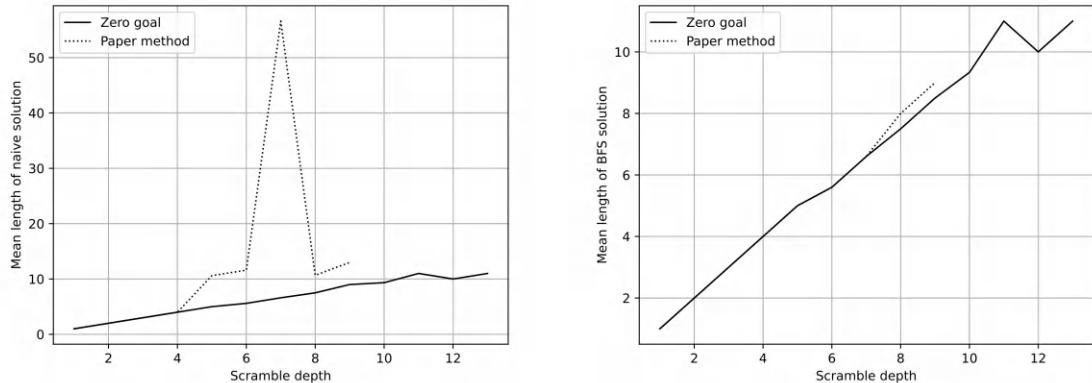


Figure 21.12: A comparison of naïve (left) and BFS (right) solution lengths for 3×3 cube

In theory, after a depth of 20, it should saturate (because “the God number” is 20), but my version wasn’t able to solve any cubes with a scramble longer than 13, so it is hard to tell.

Further improvements and experiments

There are lots of directions and things that could be tried:

- More input and network engineering: The cube is a complicated thing, so simple feed-forward NNs may not be the best model. Probably, the network could greatly benefit from convolutions.
- Oscillations and instability during training might be a sign of a common RL issue with inter-step correlations. The usual approach is the target network, when we use the old version of the network to get bootstrapped values.
- The priority replay buffer might help the training speed.
- My experiments show that the samples' weighting (inversely proportional to the scramble depth) helps to get a better policy that knows how to solve slightly scrambled cubes, but might slow down the learning of deeper states. Probably, this weighting could be made adaptive to make it less aggressive in later training stages.
- Entropy loss could be added to the training to regularize our policy.
- The 2×2 cube model doesn't take into account the fact that the cube doesn't have central cubelets, so the whole cube could be rotated. This might not be very important for a 2×2 cube, as the state space is small, but the same observation will be critical for 4×4 cubes.
- More experiments are needed to get better training and MCTS parameters.

Summary

In this chapter, we discussed discrete optimization problems – a subfield of the optimization domain that deals with discrete structures like graphs or sets. We checked RL's applicability using the Rubik's cube as a well-known, but still challenging, problem. But in general, this topic is much wider than just puzzles – the same methods could be used in optimizing schedules, optimal route planning, and other practical topics.

In the final chapter of the book, we will talk about multi-agent problems in RL.

22

Multi-Agent RL

In the last chapter, we discussed discrete optimization problems. In this final chapter, we will introduce multi-agent reinforcement learning (sometimes abbreviated to MARL), a relatively new direction of **reinforcement learning (RL)** and deep RL, which is related to situations when multiple agents communicate in an environment. In real life, such problems appear in auctions, broadband communication networks, Internet of Things, and other scenarios.

In this chapter, we will just take a quick glance at MARL and experiment a bit with simple environments; but, of course, if you find it interesting, there are lots of things you can experiment with. In our experiments, we will use a straightforward approach, with agents sharing the policy that we are optimizing, but the observation will be given from the agent's standpoint and include information about the other agent's location. With that simplification, our RL methods will stay the same, and just the environment will require preprocessing and must handle the presence of multiple agents.

More specifically, we will:

- Start with an overview of the similarities and differences between the classical single-agent RL problem and MARL
- Explore the MAgent environment, which was implemented and open sourced by the Geek.AI UK/China research group and later adopted by The Farama Foundation
- Use MAgent to train models in different environments with several groups of agents

What is multi-agent RL?

The multi-agent setup is a natural extension of the familiar RL model that we covered in *Chapter 1*. In the classical RL setup, we have one agent communicating with the environment using observations, rewards, and actions. But in some problems that often arise in real life, we have several agents involved in the environment interaction. To give some concrete examples:

- A chess game, when our program tries to beat the opponent
- A market simulation, like product advertisements or price changes, when our actions might lead to counter-actions from other participants
- Multiplayer games, like Dota 2 or StarCraft II, when the agent needs to control several units competing with other players' units (in this scenario, several units controlled by a single player might also cooperate to reach the goal)

If other agents are outside of our control, we can treat them as part of the environment and still stick to the normal RL model with the single agent. As you saw in *Chapter 20*, training via self-play is a very powerful technique, which might lead to good policies without much sophistication on the environment side. But in some situations, that's too limited and not exactly what we want.

In addition, as research shows, a group of simple agents might demonstrate collaborative behavior that is way more complex than expected. Some examples are the OpenAI blog post at <https://openai.com/blog/emergent-tool-use/> and the paper *Emergent tool use from multi-agent autocurricula* by Baker et al. [Bak+20] about the “hide-and-seek” game, where a group of agents collaborate and develop more and more sophisticated strategies and counter-strategies to win against another group of agents, for example, “build a fence from available objects” and “use a trampoline to catch agents behind the fence.”

In terms of different ways that agents might communicate, they can be separated into two groups:

- **Competitive:** When two or more agents try to beat each other in order to maximize their reward. The simplest setup is a two-player game, like chess, backgammon, or Atari Pong.
- **Collaborative:** When a group of agents needs to use joint efforts to reach some goal.

There are lots of examples that fall into one of these groups, but the most interesting and close to real-life scenarios are normally a mixture of both behaviors. There are tons of examples, starting from some board games that allow you to form allies and going up to modern corporations, where 100% collaboration is assumed, but real life is normally much more complicated than that.

From a theoretical point of view, game theory has quite a developed foundation for both communication forms, but for the sake of brevity, we're not going to deep dive into the field, which is large and has different terminology. If you're curious, you can find lots of books and courses that explore it in great depth.

To give an example, the minimax algorithm is a well-known result of game theory, and you saw it used in *Chapter 20*.

MARL is a relatively young field, but activity has been growing over time; so, it might be interesting to keep an eye on it.

Getting started with the environment

Before we jump into our first MARL example, let's look at the environment we can use. If you want to play with MARL, your choice is a bit limited. All the environments that come with Gym support only one agent. There are some patches for Atari Pong, to switch it into two-player mode, but they are not standard and are an exception rather than the rule.

DeepMind, together with Blizzard, has made StarCraft II publicly available (<https://github.com/deepmind/pysc2>) and makes for a very interesting and challenging environment for experimentation. However, for somebody who is taking their first steps in MARL, it might be too complex. In that regard, I have found the MAgent environment, originally developed by Geek.AI, to be perfectly suitable; it is simple and fast and has minimal dependency, but it still allows you to simulate different multi-agent scenarios for experimentation. It doesn't provide a Gym-compatible API, but we will implement it on our own.

If you're interested in MARL, you might also check out the PettingZoo package from The Farama Foundation: <https://pettingzoo.farama.org>. It includes more environments and a unified API for communication, but in this chapter, we're focusing only on the MAgent environment.

An overview of MAgent

Let's look at MAgent at a high level. It provides the simulation of a grid world that 2D agents inhabit. These agents can observe things around them (according to their perception length), move to some distance from their location, and attack other agents around them.

There might be different groups of agents with various characteristics and interaction parameters. For example, the first environment that we will consider is a predator-prey model, where "tigers" hunt "deer" and obtain a reward for that. In the environment configuration, you can specify lots of aspects of the group, like perception, movement, attack distance, the initial health of every agent in the group, how much health they spend on movement and attack, and so on. Aside from the agents, the environment might contain walls that are not crossable by the agents.

The nice thing about MAgent is that it is very scalable, as it is implemented in C++ internally, just exposing the Python interface. This means that the environment can have thousands of agents in the group, providing you with observations and processing the agents' actions.

Installing MAgent

As it often happens, the original version of MAgent hasn't been maintained for some time. Luckily for us, The Farama Foundation forked the original repo and are currently maintaining it, providing most of the original functionality. Their version is called MAgent2 and the documentation can be found here: <https://magent2.farama.org/>. The GitHub repository is available here: <https://github.com/Farama-Foundation/magent2>. To install MAgent2, you need to run the following command:

```
pip install magent2==0.3.3
```

Setting up a random environment

To quickly understand the MAgent API and logic, I've implemented a simple environment with "tiger" and "deer" agents, where both groups are driven by the random policy. It might not be very interesting from an RL perspective, but it will allow us to quickly learn enough about the API to implement the Gym environment wrapper. The example can be found in `Chapter22/forest_random.py` and we'll walk through it here.

We start with `ForestEnv`, defined in `lib/data.py`, which defines the environment. This class is inherited from `magent_parallel_env` (yes, the name of the class is in lowercase, in contravention of the Python style guide, but that's how it has been defined in the library), the base class for MAgent environments:

```
class ForestEnv(magent_parallel_env, EzPickle):
    metadata = {
        "render_modes": ["human", "rgb_array"],
        "name": "forest_v4",
        "render_fps": 5,
    }
```

This class mimics the Gym API, but it is not 100% compatible, so we will need to deal with this in our code later.

In the constructor, we instantiate the `GridWorld` class, which works as a Python adapter around the low-level MAgent C++ library API:

```
def __init__(self, map_size: int = MAP_SIZE, max_cycles: int = MAX_CYCLES,
            extra_features: bool = False, render_mode: tt.Optional[str] = None,
            seed: tt.Optional[int] = None, count_walls: int = COUNT_WALLS,
            count_deer: int = COUNT_DEER, count_tigers: int = COUNT_TIGERS):
    EzPickle.__init__(self, map_size, max_cycles, extra_features, render_mode, seed)
    env = GridWorld(self.get_config(map_size), map_size=map_size)
```

```
handles = env.get_handles()
self.count_walls = count_walls
self.count_deer = count_deer
self.count_tigers = count_tigers

names = ["deer", "tiger"]
super().__init__(env, handles, names, map_size, max_cycles, [-1, 1],
                 False, extra_features, render_mode)
```

In the preceding code, we instantiate the `GridWorld` class, which implements most of the logic of our environment.

The `GridWorld` class is configured by the `Config` instance returned by the `get_config` function:

```
@classmethod
def get_config(cls, map_size: int):
    # Standard forest config, but deer get reward after every step
    cfg = forest_config(map_size)
    cfg.agent_type_dict["deer"]["step_reward"] = 1
    return cfg
```

This function uses the `forest_config` function, which is imported from the `magent.builtin.config.forest` package and tweaks the configuration, adding the reward for deer on every step. This will be important when we start training the deer model, so a reward of 1 on every step will incentivize the agent to live longer.

The rest of the configuration hasn't been included here as it is largely unchanged and defines lots of details about the environment, including the following:

- How many groups of agents do we have in the environment? In our case, we have two groups: "deer" and "tigers."
- What are the properties of each group – how far can they see from their location? An example of this could be that the deer can see as far as one cell, but the tigers have can see as far as four. Can they attack others and how far? What is the initial health of each agent? How fast can they recover from damage? There are lots of parameters you can specify.
- How can they attack other groups and what damage does it do? There is lots of flexibility – for example, you can model the scenario when predators hunt only in pairs (we'll do this experiment later in the chapter). In our current setup, the situation is simple – any tiger can attack any deer without restrictions.

The last function in the ForestEnv class is generate_map, which places walls, deer, and tigers randomly on the map:

```
def generate_map(self):
    env, map_size = self.env, self.map_size
    handles = env.get_handles()

    env.add_walls(method="random", n=self.count_walls)
    env.add_agents(handles[0], method="random", n=self.count_deer)
    env.add_agents(handles[1], method="random", n=self.count_tigers)
```

Now let's get to the forest_random.py source code. In the beginning, we import the lib.data package and the VideoRecorder class from Gymnasium:

```
from gymnasium.wrappers.monitoring.video_recorder import VideoRecorder
from lib import data

RENDER_DIR = "render"
```

In *Chapter 2*, we used the RecordVideo wrapper to capture environment observations automatically, but in the case of MAgent environments, it is not possible, due to different return values (all methods are returning dictionaries for all agents at once instead of single values). To get around this, we'll use the VideoRecorder class to capture videos and write into the RENDER_DIR directory.

First, we create a ForestEnv instance and video recorder. An environment object contains the property agents, which keeps string identifiers for all the agents in the environment. In our case, it will be a list of values like deer_12 or tiger_3. With the default configuration, on the map 64×64 , we have 204 deer agents and 40 tigers, so the env.agents list has 244 items:

```
if __name__ == "__main__":
    env = data.ForestEnv(render_mode="rgb_array")
    recorder = VideoRecorder(env, RENDER_DIR + "/forest-random.mp4")
    sum_rewards = {agent_id: 0.0 for agent_id in env.agents}
    sum_steps = {agent_id: 0 for agent_id in env.agents}
```

We reset the environment using the reset() method, but now it returns one value (instead of the two in the Gym API). The returned value is a dict with agent IDs as keys and observation tensors as values:

```
obs = env.reset()
recorder.capture_frame()
assert isinstance(obs, dict)
print(f"tiger_0: obs {obs['tiger_0'].shape}, act: {env.action_space('tiger_0')}")
print(f"deer_0: obs {obs['deer_0'].shape}, act: {env.action_space('deer_0')}\n")
step = 0
```

The preceding code produces the following output:

```
tiger_0: obs (9, 9, 5), act: Discrete(9)
deer_0: obs (3, 3, 5), act: Discrete(5)
```

The action space contains five mutually exclusive actions for deer (four directions + a “do nothing” action). Tigers can do the same, but in addition can attack in four directions.

In terms of observations, every tiger gets a 9×9 matrix with five different planes of information. Deer are more short-sighted, so their observation is just 3×3 . The observation always contains the agent in the center, so it shows the grid around this specific agent. The five planes of information are:

- Walls: 1 if this cell contains the wall and 0 otherwise
- Group 1 (the group that the agent belongs to): 1 if the cell contains agents from the agent’s group and 0 otherwise
- Group 1 health: The relative health of the agent in this cell
- Group 2 (the group with enemy agents): 1 if there is an enemy in this cell and 0 otherwise
- Group 2 health: The relative health of the enemy or 0 if nothing is there

If more groups are configured, the observation will contain more planes in the observation tensor. In addition, MAgent has a “minimap” functionality that adds the “zoomed-out” location of agents of every group. This minimap feature is disabled in my examples, but you can experiment with it to check the effect on the training. Without this feature, every agent sees only a limited range of cells around itself, but minimap allows them to have a more global view of the environment.

Groups 1 and 2 are relative to the agent’s group; so, in the second plane, deer have information about other deer and for tigers, this plane includes other tigers. This makes observations group-independent and allows us to train a single policy for both groups if needed.

Another optional part of the observation is the so-called “extra features,” which includes the agent’s ID, last action, last reward, and normalized position. Concrete details could be found in the MAgent source code, but we’re not going to use this functionality in our examples.

Let’s continue describing our code. We have a loop that is repeated until we have alive agents in the environment.

On every iteration, we sample random actions for all the agents and execute them in the environment:

```
while env.agents:
    actions = {agent_id: env.action_space(agent_id).sample() for agent_id in
               env.agents}
    all_obs, all_rewards, all_dones, all_trunc, all_info = env.step(actions)
    recorder.capture_frame()
```

All values returned from the `env.step()` function are dictionaries with `agent_id` as the key. Another very important detail about the MAgent environment is that the set of agents is volatile: agents can disappear from the environment (when they die, for example). In our “forest” environment, tigers are losing 0.1 points of health every step, which could be increased after eating deer. Deer lose health only after an attack and gain it on every step (likely from eating grass).

When the agent dies (a tiger from starvation or a deer from a tiger’s attack), the corresponding entry in the `all_dones` dict is set to `True` and on the next iteration, the agent disappears from all the dictionaries and the `env.agents` list. So, after the death of one agent, the whole episode continues and we need to take this into account during the training.

In the preceding example, the loop is executed until no more agents are alive. As both tigers and deer are behaving randomly (and tigers are losing health at every step), it is very likely that all tigers will die from starvation and the surviving deer will live happily infinitely long. But the environment is configured to automatically remove all deer when no more tigers are left, so our program ends after 30-40 steps.

At the end of the loop, we sum up the reward obtained by agents and track the amount of steps for which they were alive:

```
for agent_id, r in all_rewards.items():
    sum_rewards[agent_id] += r
    sum_steps[agent_id] += 1
step += 1
```

After the loop, we show the top 20 agents sorted by their reward:

```
final_rewards = list(sum_rewards.items())
final_rewards.sort(key=lambda p: p[1], reverse=True)
for agent_id, r in final_rewards[:20]:
    print(f"{agent_id}: got {r:.2f} in {sum_steps[agent_id]} steps")
recorder.close()
```

The output from this tool might look like this:

```
$ ./forest_random.py
tiger_0: obs (9, 9, 5), act: Discrete(9)
deer_0: obs (3, 3, 5), act: Discrete(5)

tiger_5: got 34.80 in 37 steps
tiger_37: got 19.70 in 21 steps
tiger_31: got 19.60 in 21 steps
tiger_9: got 19.50 in 21 steps
tiger_24: got 19.40 in 21 steps
tiger_36: got 19.40 in 21 steps
tiger_38: got 19.40 in 21 steps
tiger_1: got 19.30 in 21 steps
tiger_3: got 19.30 in 21 steps
tiger_11: got 19.30 in 21 steps
tiger_12: got 19.30 in 21 steps
tiger_17: got 19.30 in 21 steps
tiger_19: got 19.30 in 21 steps
tiger_26: got 19.30 in 21 steps
tiger_32: got 19.30 in 21 steps
tiger_2: got 19.20 in 21 steps
tiger_8: got 19.20 in 21 steps
tiger_10: got 19.20 in 21 steps
tiger_23: got 19.20 in 21 steps
tiger_25: got 19.20 in 21 steps
Moviepy - Building video render/forest-random.mp4.
Moviepy - Writing video render/forest-random.mp4
```

In my simulation, one agent (`tiger_5`) was especially lucky and lived longer than others. At the end, the program saves a video of the episode. The result of my run is available here: <https://youtube.com/shorts/pH-Rz9Q4yrl>.

In *Figure 22.1*, two different states are shown: at the beginning of the game and close to the end. Tigers are shown with blue dots (or the darker ones if you’re reading this in grayscale), the red dots are deer, and the gray dots are walls (you can refer to the digital version of the book to see the colors in the screenshot).

The attack direction is shown with small black arrows.

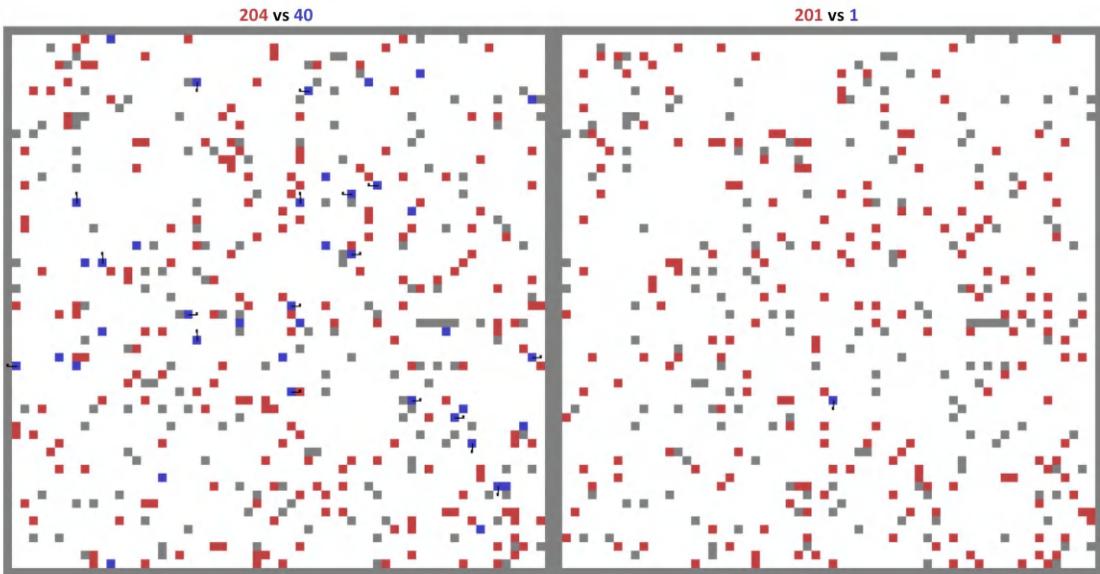


Figure 22.1: Two states of the forest environment: at the beginning of the episode (left) and close to the end (right)

In this example, both groups of agents were behaving randomly, which is not very interesting. In the next section, we'll apply a **deep Q-network (DQN)** to improve the tiger's hunting skills.

Deep Q-network for tigers

Here, we will apply the DQN model to the tiger group of agents to check whether they can learn how to hunt better. All of the agents share the network, so their behavior will be the same. The deer group will keep random behavior in this example to keep things simple for now; we'll train them later in the chapter.

The training code can be found in `Chapter22/forest_tigers_dqn.py`; it doesn't differ much from the other DQN versions from the previous chapters.

Understanding the code

To make the `MAgent` environment work with our classes, a specialized version of `ExperienceSourceFirstLast` was implemented to handle the specifics of the environment. This class is called `MAgentExperienceSourceFirstLast` and can be found in `lib/data.py`. Let's check it out to understand how it fits into the rest of the code.

The first class we define is the items produced by our `ExperienceSource`. As we discussed in *Chapter 7*, instances of the class `ExperienceFirstLast` contain the following fields:

- `state`: Observations from the environment at the current step
- `action`: The action we executed
- `reward`: The amount of reward we obtained
- `last_state`: Observation after executing the action

In a multi-agent setup, every agent is producing the same set of data, but we also have to be able to identify which group this agent belongs to (in this tiger-deer example, does this experience correspond to the tiger or the deer's trajectory?). To retain this information, we define a subclass, `ExperienceFirstLastMARL`, with a new field keeping the group's name:

```
@dataclass(frozen=True)
class ExperienceFirstLastMARL(ExperienceFirstLast):
    group: str

class MAgentExperienceSourceFirstLast:
    def __init__(self, env: magent_parallel_env, agents_by_group: tt.Dict[str,
        BaseAgent],
                 track_reward_group: str, env_seed: tt.Optional[int] = None,
                 filter_group: tt.Optional[str] = None):
```

In the constructor of `MAgentExperienceSourceFirstLast`, we pass the following arguments:

- `magent_parallel_env`: The MAgent parallel environment (we experimented with this in the previous section).
- `agents_by_group`: The PTAN `BaseAgent` object for every group of agents. In our tiger DQN example, tigers will be controlled by a neural network (`ptan.agent.DQNAgent`), but deer behave randomly.
- `track_reward_group`: The parameter that specifies the group for which we're tracking the episode's reward.
- `filter_group`: An optional filter for the group for which we want to generate an experience. In our current example, we need only observations from tigers (as we train only tigers), but in the next section, we'll train a DQN for both tigers and deer, so the filter will be disabled.

In the subsequent constructor code, we store arguments and create two useful mappings for agents: from the agent ID to the group name and back:

```

self.env = env
self.agents_by_group = agents_by_group
self.track_reward_group = track_reward_group
self.env_seed = env_seed
self.filter_group = filter_group
self.total_rewards = []
self.total_steps = []

# forward and inverse map of agent_id -> group
self.agent_groups = {
    agent_id: self.agent_group(agent_id)
    for agent_id in self.env.agents
}
self.group_agents = collections.defaultdict(list)
for agent_id, group in self.agent_groups.items():
    self.group_agents[group].append(agent_id)

@classmethod
def agent_group(cls, agent_id: str) -> str:
    a, _ = agent_id.split("_", maxsplit=1)
    return a

```

We also define a utility method to strip the agent's numerical ID to get the group name (in our case, it will be tiger or deer).

Now comes the main method of the class: the iterator interface, which produces experience items from the environment:

```

def __iter__(self) -> tt.Generator[ExperienceFirstLastMARL, None, None]:
    # iterate episodes
    while True:
        # initial observation
        cur_obs = self.env.reset(self.env_seed)

        # agent states are kept in groups
        agent_states = {
            prefix: [self.agents_by_group[prefix].initial_state() for _ in group]
            for prefix, group in self.group_agents.items()
        }

```

Here, we reset the environment in the beginning and create initial states for agents (in case our agents will keep some state, but in this chapter's examples, they are stateless).

Then, we iterate the episode until we have living agents (the same way we did in the example in the previous section). In this loop, we fill actions in the dictionary, mapping the agent ID to the action. To do this, we use PTAN BaseAgent instances, which work with batches of observations, so actions will be produced very efficiently for the whole group of agents:

```
episode_steps = 0
episode_rewards = 0.0
# steps while we have alive agents
while self.env.agents:
    # calculate actions for the whole group and unpack
    actions = {}
    for prefix, group in self.group_agents.items():
        gr_obs = [
            cur_obs[agent_id]
            for agent_id in group if agent_id in cur_obs
        ]
        gr_actions, gr_states = self.agents_by_group[prefix](
            gr_obs, agent_states[prefix])
        agent_states[prefix] = gr_states
        idx = 0
        for agent_id in group:
            if agent_id not in cur_obs:
                continue
            actions[agent_id] = gr_actions[idx]
            idx += 1
```

Once we have actions to be executed, we send them to the environment and obtain dictionaries with new observations, rewards, and done and truncation flags. Then, we generate experience items for every alive agent we currently have:

```
new_obs, rewards, dones, trunks, _ = self.env.step(actions)

for agent_id, reward in rewards.items():
    group = self.agent_groups[agent_id]
    if group == self.track_reward_group:
        episode_rewards += reward
    if self.filter_group is not None:
        if group != self.filter_group:
            continue
    last_state = new_obs[agent_id]
    if dones[agent_id] or trunks[agent_id]:
        last_state = None
    yield ExperienceFirstLastMARL(
        state=cur_obs[agent_id], action=actions[agent_id],
```

```

            reward=reward, last_state=last_state, group=group
        )
cur_obs = new_obs
episode_steps += 1

```

At the end of the episode, we remember the number of steps and the average reward obtained by the group:

```

self.total_steps.append(episode_steps)
tr_group = self.group_agents[self.track_reward_group]
self.total_rewards.append(episode_rewards / len(tr_group))

```

With this class at hand, our DQN training code stays almost the same as in the single-agent RL case. The full source code of this example can be found in `forest_tigers_dqn.py`. Here, I'm going to show only part of the code where PTAN agents and the experience source are created (to illustrate how `MAgentExperienceSourceFirstLast` is being used):

```

action_selector =
    ptan.actions.EpsilonGreedyActionSelector(epsilon=PARAMS.epsilon_start)
    epsilon_tracker = common.EpsilonTracker(action_selector, PARAMS)
    tiger_agent = ptan.agent.DQNAgent(net, action_selector, device)
    deer_agent = data.RandomMAgent(env, env.handles[0])
    exp_source = data.MAgentExperienceSourceFirstLast(
        env,
        agents_by_group={'deer': deer_agent, 'tiger': tiger_agent},
        track_reward_group="tiger",
        filter_group="tiger",
    )
    buffer = ptan.experience.ExperienceReplayBuffer(exp_source, PARAMS.replay_size)

```

As you can see, tigers are controlled by the neural network (which is a very simple two-layer convolution plus a two-layer fully connected net). A group of deer is controlled by a random number generator. The experience replay buffer will be populated only with the tiger experience because of the `filter_group="tiger"` argument.

Training and results

To start the training, run `./forest_tigers_dqn.py -n run_name -dev cuda`. In one hour of training, the tiger's test reward has reached the best score of 82, which is a significant improvement over the random baseline. Acting randomly, most tigers die after 20 steps and only a few lucky ones can live longer.

Let's calculate how many deer were eaten to get this score. Initially, each tiger has a health of 10 and spends 0.5 of their health on each step. In total, we have 40 tigers and 204 deer on the map (you can change this amount with command-line arguments). For every eaten deer, the tigers obtain 8 health points, which allows them to survive for an extra 16 steps. For every step, each tiger obtains a reward of 1, so the "excess reward" from the deer eaten by 40 tigers is $82 \cdot 40 - 20 \cdot 40 = 2480$. Every deer gives 8 health points, which is converted into an extra 16 steps of a tiger's life, so the number of deer eaten is $2480/16 = 155$. So, almost 76% of deer were hunted by the best policy we've got. Not bad given that deer are randomly placed on the map and tigers need to get to them to attack.

It's quite likely that the policy stopped improving just because of the limited view of the tigers. If you are curious, you can enable the minimap in the environment settings and experiment. With more information about the food's location, it's likely that the policy could be improved even more.

In *Figure 22.2*, the average reward and number of steps during the training is shown. From it, you can see that the main growth was during the first 300 episodes and later, the training progress was almost 0:

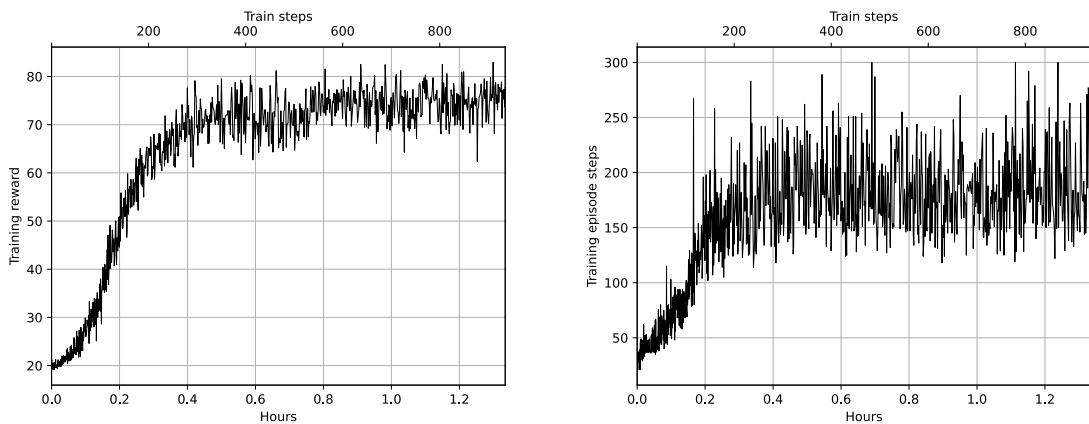


Figure 22.2: Average reward (left) and count of steps (right) from training episodes

However, *Figure 22.3* shows the plots for the test reward and steps, which demonstrate that the policy continued to improve even after 300 episodes (≈ 0.4 hours of training):

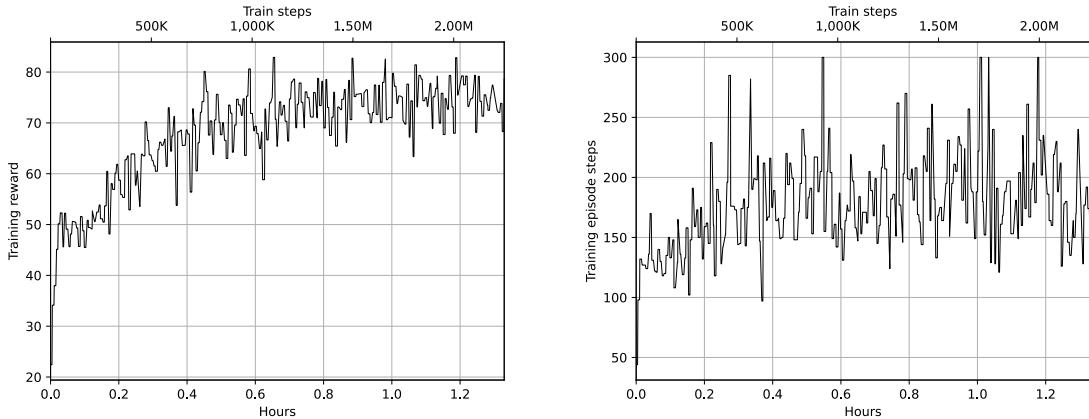


Figure 22.3: Average reward (left) and count of steps (right) from test episodes

The final pair of charts in *Figure 22.4* shows the training loss and epsilon during the training. Both plots are correlated, which indicates that most of the novelty during the training was obtained in the exploration phase (as the loss value is high, which means that new situations arise during training). This might be an indication that better exploration methods might be beneficial for the final policy.

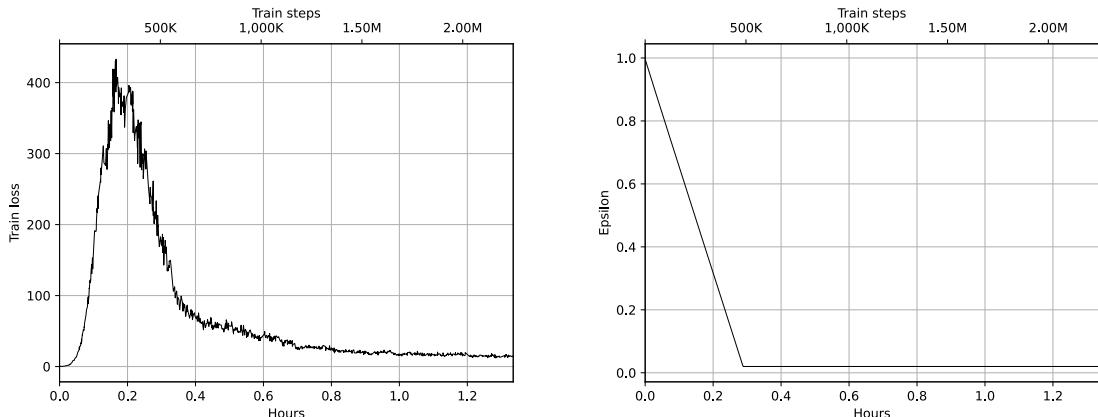


Figure 22.4: Average loss (left) and epsilon (right) during the training

As usual, besides the training, I implemented a tool to check the models in action. It's called `forest_tigers_play.py` and it loads the trained model and uses it during the episode, producing a video recording of the observations. The video from the best model (with a test score of 82.89) is available here: <https://www.youtube.com/shorts/ZZf80AHk538>. As you can see, tigers' hunting skills are now significantly better than the random policy: at the end of the episode, just 53 deer were left from the initial 204.

Collaboration by the tigers

The second experiment that I implemented was designed to make the tigers' lives more complicated and encourage collaboration between them. The training and play code are the same; the only difference is in the MAgent environment's configuration.

If you pass the argument `--mode double_attack` to the training utility, the environment `data.DoubleAttackEnv` will be used. The only difference is the configuration object, which sets additional constraints on tigers' attacks. In the new setup, they can attack deer only in pairs and have to do this at the same time. A single tiger's attack doesn't have any effect. This definitely complicates the training and hunting, as obtaining the reward from eating the deer is now much harder for tigers.

To start the training, you can run the same train utility, but with an extra command-line argument:

```
./forest\_tigers\_dqn.py -n run-name --dev cuda --mode  
double\_attack
```

Let's take a look at the results.

In *Figure 22.5*, the reward and step plots for the training episodes are shown. As you can see, even after 2 hours of training, the reward is still improving. At the same time, the count of steps in the episode never exceeds 300, which might be an indication that tigers just don't have nearby deer to eat and die from starvation (it also might just be an internal limit of steps in the environment).

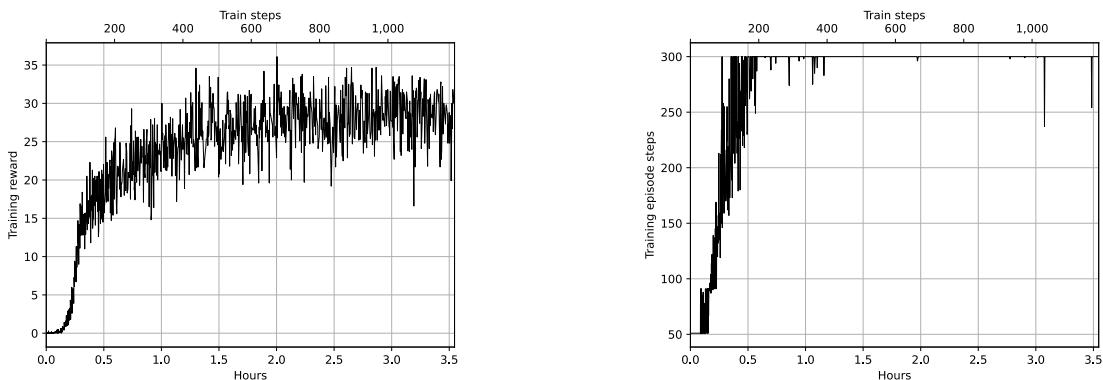


Figure 22.5: Average reward (left) and count of steps (right) from training episodes in double_attack mode

In contrast to single-tiger hunting mode, the loss during the training is not decreasing (as shown in *Figure 22.6*), which might indicate that training hyperparameters could be improved:

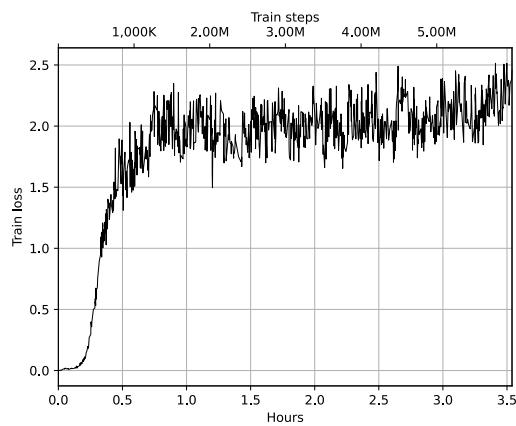


Figure 22.6: Average loss during training

To test the model in action, you can use the same utility as before; just pass it the -mode `double_attack` argument. A video recording of the best model I got is available here: <https://youtu.be/VjGbzP1r7HY>. As you can see, tigers are now moving in pairs, attacking the deer together.

Training both tigers and deer

The next example is the scenario when both tigers and deer are controlled by different DQN models being trained simultaneously. Tigers are rewarded for living longer, which stimulates them to eat more deer, as at every step in the simulation, they lose health points. Deer are also rewarded on every timestamp.

The code is in `forest_both_dqn.py` and it is an extension of the previous example. For both groups of agents, we have a separate `DQNAgent` class instance, which uses separate neural networks to convert observations into actions. The experience source is the same, but now we're not filtering on a tiger's group experience (with the parameter `filter_group=None`). Because of this, our replay buffer now contains observations from all the agents in the environment, not just from tigers as in the previous example. During the training, we sample a batch and split examples from deer and tigers into two separate batches to be used for training their networks.

I'm not going to include all the code here, as it differs from the previous example only in small details. If you are curious, you can take a look at the source code in the GitHub repository. *Figure 22.7* shows the training reward and steps for tigers. You can see that initially, tigers were able to consistently increase their reward, but later, the growth stopped:

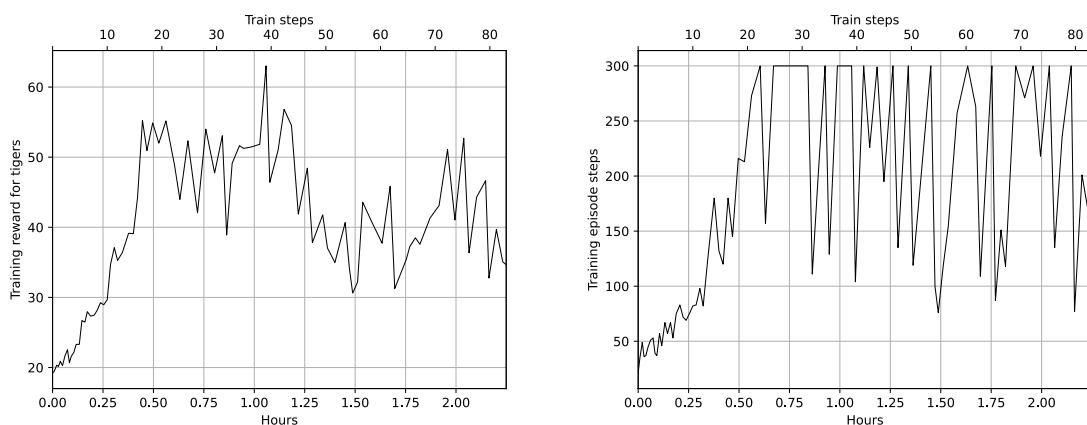


Figure 22.7: Average reward for tigers (left) and count of steps (right) from training episodes

In the next two plots in *Figure 22.8*, the reward for tigers and deer during the testing is shown.

There is no clear trend here; both groups are competing and trying to beat their opponent:

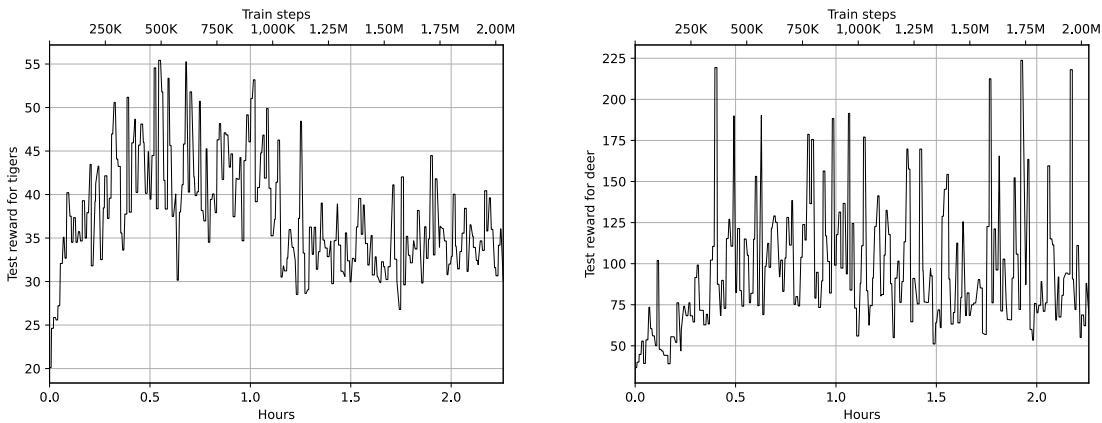


Figure 22.8: Test reward for tigers (left) and deer (right)

As you can see from *Figure 22.8*, deer are way more successful than tigers, which is not surprising, as the speed of both is the same, so the deer just need to move all the time and wait for the tigers to die from starvation. If you want, you can experiment with the environment settings by increasing either the tigers' speed or the wall density.

As before, it is possible to visualize the learned policies with the utility `forest_both_play.py`, but now you need to pass two model files. Here is a video comparing the best model for deer and the best model for tigers: <https://youtube.com/shorts/vuVL1e26KqY>. In the video, all deer are just moving to the left of the field. Most likely, tigers can exploit this simple policy for their benefit.

The battle environment

Besides the tiger-deer environment, MAgent contains several other predefined configurations you can find in the `magent2.builtin.config` and `magent2.environment` packages. As a final example in this chapter, we'll take a look at the "battle" configuration, where two groups of agents are fighting each other (without eating, thank goodness). Both agents have health points of 10 and every attack takes 2 health points, so 5 consecutive attacks are required to get the reward for the agent.

You can find the code in `battle_dqn.py`. In this setup, one group is behaving randomly and another is using the DQN to improve the policy. Training took two hours and the DQN was able to find a decent policy, but at the end, the training process diverged. In *Figure 22.9*, the training and test reward plots are shown:

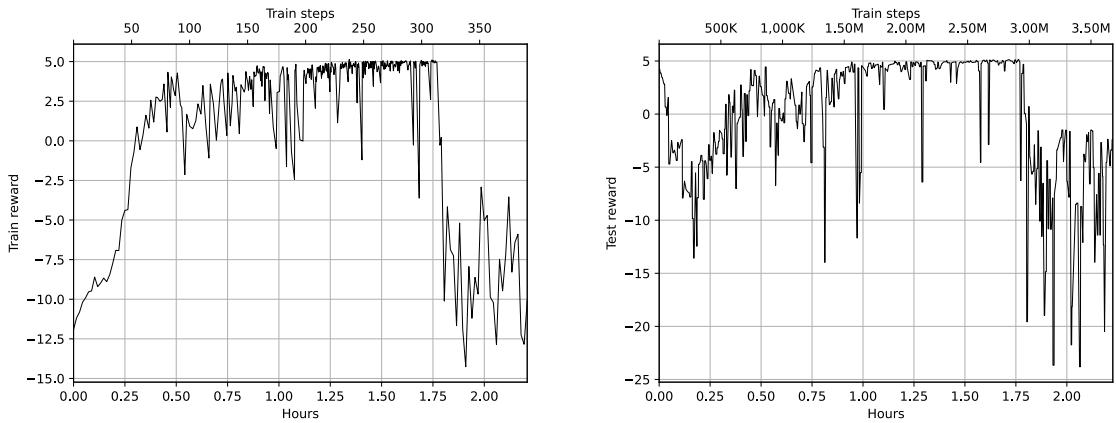


Figure 22.9: Average reward during training (left) and test (right) in the battle scenario

The video recording (produced by the tool `battle_play.py`) is available here: <https://youtube.com/shorts/ayfCa8xGY2k>. The blue team is random and the red is controlled by the DQN.

Summary

In this chapter, we just touched a bit on the very interesting and dynamic field of MARL, which has several practical applications in trading simulation, communication networks, and others. There are lots of things that you can try on your own using the MAgent environment or other environments (like PySC2).

My congratulations on reaching the end of the book! I hope that the book was useful and you enjoyed reading it as much as I enjoyed gathering the material and writing all the chapters. As a final word, I would like to wish you good luck in this exciting and dynamic area of RL. The domain is developing very rapidly, but with an understanding of the basics, it will become much simpler for you to keep track of the new developments and research in this field.

There are many very interesting topics left uncovered, such as partially observable Markov decision processes (where environment observations don't fulfill the Markov property) or recent approaches to exploration, such as the count-based methods. There has been a lot of recent activity around multi-agent methods, where many agents need to learn how to coordinate to solve a common problem. I also haven't mentioned the memory-based RL approach, where your agent can maintain some sort of memory to keep its knowledge and experience. A great deal of effort is being put into increasing the RL sample efficiency, which will ideally be close to human learning performance one day, but this is still a far-off goal at the moment. Of course, it's not possible to cover the full domain in just one book, because new ideas appear almost every day.

However, the goal of this book was to give you a practical foundation in the field, simplifying your own learning of the common methods.

I'd like to end by quoting Volodymyr Mnih's words from his talk, *Recent Advances, Frontiers and Future of Deep RL*, from the Deep RL Bootcamp, Berkeley, 2017, which are still very relevant: "The field of deep RL is very new and everything is still exciting. Literally, nothing is solved yet!"

Leave a Review!

Thank you for purchasing this book from Packt Publishing—we hope you enjoyed it! Your feedback is invaluable and helps us improve and grow. Please take a moment to leave an Amazon review; it will only take a minute, but it makes a big difference for readers like you.

Scan the QR code below to receive a free ebook of your choice.



<https://packt.link/NzOWQ>

Bibliography

- [Sut88] Richard S Sutton. “Learning to predict by the methods of temporal differences”. In: *Machine learning* 3 (1988), pp. 9–44.
- [HS96] Sepp Hochreiter and Jürgen Schmidhuber. “LSTM can solve hard long time lag problems”. In: *Advances in neural information processing systems* 9 (1996).
- [RK04] Reuven Y Rubinstein and Dirk P Kroese. *The cross-entropy method: a unified approach to combinatorial optimization, Monte-Carlo simulation, and machine learning*. Vol. 133. Springer, 2004.
- [SL08] Alexander L Strehl and Michael L Littman. “An analysis of model-based interval estimation for Markov decision processes”. In: *Journal of Computer and System Sciences* 74.8 (2008), pp. 1309–1331.
- [Kro+11] Dirk P Kroese et al. “Cross-entropy method”. In: *European Journal of Operational Research* 31 (2011), pp. 276–283.
- [LS11] Joel Lehman and Kenneth O Stanley. “Abandoning objectives: Evolution through the search for novelty alone”. In: *Evolutionary computation* 19.2 (2011), pp. 189–223.
- [Mni13] Volodymyr Mnih. “Playing atari with deep reinforcement learning”. In: *arXiv preprint arXiv:1312.5602* (2013).
- [Sil+14] David Silver et al. “Deterministic policy gradient algorithms”. In: *International conference on machine learning*. Pmlr. 2014, pp. 387–395.
- [Lil15] TP Lillicrap. “Continuous control with deep reinforcement learning”. In: *arXiv preprint arXiv:1509.02971* (2015).
- [MG15] James Martens and Roger Grosse. “Optimizing neural networks with kronecker-factored approximate curvature”. In: *International conference on machine learning*. PMLR. 2015, pp. 2408–2417.

- [Mni+15] Volodymyr Mnih et al. “Human-level control through deep reinforcement learning”. In: *nature* 518.7540 (2015), pp. 529–533.
- [Sch+15] Tom Schaul et al. “Prioritized Experience Replay”. In: (2015). arXiv: 1511 . 05952 [cs.LG]. URL: <https://arxiv.org/abs/1511.05952>.
- [Sch15] John Schulman. “Trust Region Policy Optimization”. In: *arXiv preprint arXiv:1502.05477* (2015).
- [VGS16] Hado Van Hasselt, Arthur Guez, and David Silver. “Deep reinforcement learning with double q-learning”. In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 30. 1. 2016.
- [Wan+16] Ziyu Wang et al. “Dueling network architectures for deep reinforcement learning”. In: *International conference on machine learning*. PMLR. 2016, pp. 1995–2003.
- [BDM17] Marc G Bellemare, Will Dabney, and Rémi Munos. “A distributional perspective on reinforcement learning”. In: *International conference on machine learning*. PMLR. 2017, pp. 449–458.
- [Chr+17] Paul Christiano et al. *Deep reinforcement learning from human preferences*. 2017. eprint: arXiv:1706 . 03741.
- [For+17] Meire Fortunato et al. “Noisy Networks for Exploration”. In: (2017). arXiv: 1706 . 10295 [cs.LG]. URL: <https://arxiv.org/abs/1706.10295>.
- [Mar+17] Jarryd Martin et al. “Count-based exploration in feature space for reinforcement learning”. In: *arXiv preprint arXiv:1706.08090* (2017).
- [Ost+17] Georg Ostrovski et al. “Count-based exploration with neural density models”. In: *International conference on machine learning*. PMLR. 2017, pp. 2721–2730.
- [Sal+17] Tim Salimans et al. “Evolution strategies as a scalable alternative to reinforcement learning”. In: *arXiv preprint arXiv:1703.03864* (2017).
- [Sch+17] John Schulman et al. “Proximal policy optimization algorithms”. In: *arXiv preprint arXiv:1707.06347* (2017).
- [SSa17] David Silver, Julian Schrittwieser, and Karen Simonyan et al. *Mastering the game of Go without human knowledge*. 2017. eprint: 10 . 1038/nature24270.
- [Sil+17] David Silver et al. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*. 2017. arXiv: 1712 . 01815 [cs.AI]. URL: <https://arxiv.org/abs/1712.01815>.

- [Suc+17] Felipe Petroski Such et al. “Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning”. In: *arXiv preprint arXiv:1712.06567* (2017).
- [Vas17] A Vaswani. “Attention is all you need”. In: *Advances in Neural Information Processing Systems* (2017).
- [Wu+17] Yuhuai Wu et al. “Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation”. In: *Advances in neural information processing systems* 30 (2017).
- [Bar+18] Gabriel Barth-Maron et al. “Distributed distributional deterministic policy gradients”. In: *arXiv preprint arXiv:1804.08617* (2018).
- [Bur+18] Yuri Burda et al. “Exploration by random network distillation”. In: *arXiv preprint arXiv:1810.12894* (2018).
- [Haa+18] Tuomas Haarnoja et al. “Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor”. In: *International conference on machine learning*. PMLR. 2018, pp. 1861–1870.
- [Hes+18] Matteo Hessel et al. “Rainbow: Combining improvements in deep reinforcement learning”. In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 32. 1. 2018.
- [McA+18] Stephen McAleer et al. “Solving the Rubik’s cube without human knowledge”. In: *arXiv preprint arXiv:1805.07470* (2018).
- [Bak+20] Bowen Baker et al. *Emergent Tool Use From Multi-Agent Autocurricula*. 2020. arXiv: 1909.07528 [cs.LG]. URL: <https://arxiv.org/abs/1909.07528>.
- [FS20] Alexander H Frey Jr and David Singmaster. “Handbook of cubik math”. In: (2020).
- [Sch+20] Julian Schrittwieser et al. “Mastering Atari, Go, chess and shogi by planning with a learned model”. In: *Nature* 588.7839 (Dec. 2020), pp. 604–609. ISSN: 1476-4687. DOI: 10.1038/s41586-020-03051-4. URL: <http://dx.doi.org/10.1038/s41586-020-03051-4>.
- [BDR23] Marc G Bellemare, Will Dabney, and Mark Rowland. *Distributional reinforcement learning*. MIT Press, 2023.



www.packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

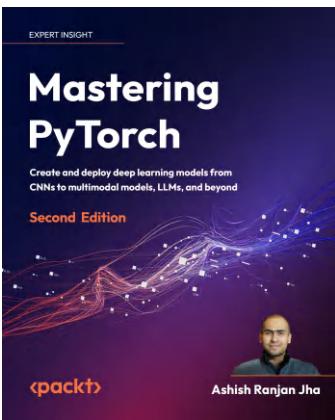
Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

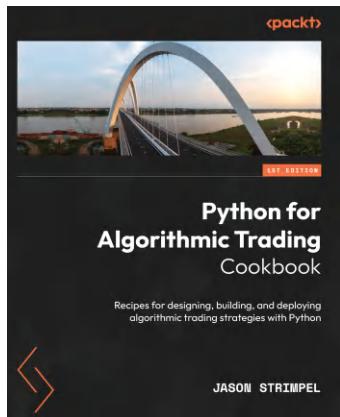


Mastering PyTorch

Ashish Ranjan Jha

ISBN: 9781801074308

- Implement text, vision, and music generation models using PyTorch
- Build a deep Q-network (DQN) model in PyTorch
- Deploy PyTorch models on mobile devices (Android and iOS)
- Become well versed in rapid prototyping using PyTorch with fastai
- Perform neural architecture search effectively using AutoML
- Easily interpret machine learning models using Captum
- Design ResNets, LSTMs, and graph neural networks (GNNs)
- Create language and vision transformer models using Hugging Face



Python for Algorithmic Trading Cookbook

Jason Strimpel

ISBN: 9781835084700

- Acquire and process freely available market data with the OpenBB Platform
- Build a research environment and populate it with financial market data
- Use machine learning to identify alpha factors and engineer them into signals
- Use VectorBT to find strategy parameters using walk-forward optimization
- Build production-ready backtests with Zipline Reloaded and evaluate factor performance
- Set up the code framework to connect and send an order to Interactive Brokers

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Index

Symbols

ϵ -greedy 518–522
 2×2 cube model 637
 3×3 cube model 639

A

A2C baseline 463
 implementation 463, 464
 results 465–468
 video recording 468
A2C method 435, 436
 implementing 436–440
 models, using 441, 442
 results 440, 441
 videos, recording 441, 442
A2C on Pong 331–337
A3C, with data parallelism 346
 results 346
A3C, with gradient parallelism 347
 implementation 347, 348, 350, 352
 results 353
ACKTR 481
 implementation 481
 results 482
action selector 174, 176
actions 34

continuous action 34
discrete actions 34
actor-critic parallelization
 data parallelism 344
 gradients parallelism 344, 345
advantage actor-critic (A2C) 87, 328–330,
 575
 combining, with reward model
 560–562
 results 338–341
 used, for initial training 548–552
agent 176, 177
 anatomy 28
 DQNAgent 177, 178
 implementing, in Python 28–30
 PolicyAgent 179
AlphaGo Zero method 569, 572
 evaluation 576
 implementing, for Connect 4 576,
 577
 MCTS 573–575
 overview 572, 573
 results 588–591
 self-play 575, 576
 training 576
artificial intelligence (AI) 3

- asynchronous advantage actor-critic (A3C)
 342
- correlation and sample efficiency
 342, 343
- Atari experiments 537
- DQN + epsilon 538
- DQN + noisy networks 539
- PPO 539
- Atari images
- GAN training 74–80
- GAN training, with PyTorch Ignite on
 82, 84, 85
- autodidactic iterations (ADI) 623
- B**
- bandits exploration approach 523
- baseline DQN 209, 210, 256, 257
- battle environment 662
- behavior characteristic (BC) 510
- Bellman equation
- of optimality 116–118
- black-box optimizations methods 491,
 492
- evolution strategies 492, 493
- genetic algorithms 507
- bootstrapping 143
- branching factor 574
- breadth-first search (BFS) 625
- browser automation 398, 399
- challenges 399
- C**
- candlestick chart 276
- CartPole
- ES, implementing on 493–498
- GA, implementing on 508, 509
- results 499, 500
- CartPole environment
- example 304–308
- policy gradient method on 313
- CartPole variance 325–328
- categorical DQN 237–240
- hyperparameter tuning 248
- implementation 240–246
- results 246, 248
- central processing unit (CPU) 639
- channel, height, width (CHW)format 151
- ChatGPT 389
- ChatGPT API 392, 393, 395
- interactive mode 389–392
- setup 389
- code outline, Rubik's cube 628
- cube environments 629–632
- search process 634
- training process 633
- Connect 4 game, with AlphaGo Zero 576
- comparison 588
- game model 578, 579
- MCTS implementation 580–585
- model 585–587
- testing 588
- training 588
- Connect 4, with MuZero 594
- hyperparameters 594
- MCTS search 601–604
- MCTS tree nodes 595–597
- models 598–601
- training data and gameplay 604,
 606–608
- continuous action 34

- continuous action space 432, 433
environments 433–435
- convolution model 293, 294
- covariance matrix adaptation evolution strategy (CMA-ES) 493
- cross-entropy method 88
in practice 89–91
on CartPole 91, 93–98, 100, 101
on FrozenLake 101–107
theoretical background 109, 110
- cube solving approaches 616
actions 617
states 618–621
- custom layers 64–67
- D**
- decoder 368
- deconvolution 75
- deep deterministic policy gradient (DDPG)
173, 443, 444, 461
exploring 444
implementing 445–450
results and video 450, 452, 453
- deep GA 510
- deep learning (DL) 3
- deep neural networks 5
- deep NLP 364
Encoder-Decoder architecture 368
RNNs 365, 366
transformers 369
word embedding 367, 368
- deep Q-learning 140, 141
correlation between steps 143
DQN algorithm 144, 145
environment, interacting with 142
Markov property 144
SGD optimization 143
- deep Q-network (DQN) 87, 171, 355
- deep Q-network, for tigers 652
code 652–656
training 656, 658
- depth-first search (DFS) 625
- Dilbert reward process (DRP) 20
- discrete actions 34
- discrete optimization 614
- Distributed Distributional Deep Deterministic Policy Gradients (D4PG) 453
architecture 453
implementing 454–456
results 457, 458
- dopamine library 194
- double DQN 215
hyperparameter tuning 219
implementation 216–218
results 218, 219
- DQN extensions 195
categorical DQN 237–240
double DQN 215
dueling DQN 233, 234
N-step DQN 210–212
noisy networks 219, 220
prioritized replay buffer 225, 226
- DQN model 378
- DQN training 144, 145
- DQN, basic implementation 196, 202, 203
common library 196–201
hyperparameter tuning 203–207
results, with common parameters

- 207–209
 tuned baseline DQN 209, 210
DQN, on Pong 145, 146
 DQN model 151, 153
 model, working with 167–169
 performance 164–167
 training 153, 154, 156–158, 160–164
 wrappers 146–150
DQNAgent 177, 178
dueling DQN 233, 234
 hyperparameter tuning 237
 implementation 235, 236
 results 236
- E**
- encoder 368
Encoder-Decoder architecture 368
environment 28
 implementing, in Python 28–30
epsilon-greedy method 142
evolution strategies (ES) 492, 493
 CartPole results 499, 500
 HalfCheetah results 505, 506
 implementing, on CartPole 493–498
 implementing, on HalfCheetah
 500–505
experience replay buffers 186, 187
ExperienceSource class 180, 182, 183
 Gym environment, implementing
 181, 182
 implementing 184, 186
exploration 517
 approaches 522
 count-based methods 523
- noisy networks 522, 523
 prediction-based methods 524
 significance 518
exponential linear unit (ELU) activations
 622
- F**
- feed-forward model 288–290
 fine-tuning 563, 564
 fitness function 491
FrozenLake
 Q-iteration 129–131
functional operations 54
- G**
- GA algorithm**
 tweaks 510
GA algorithm, tweaks
 deep GA 510
 novelty search 510
GA, on HalfCheetah 511
 implementing 511–513
 results 514
GAN training
 on Atari images, with PyTorch Ignite
 82, 84, 85
generative adversarial network (GANs)
 70
generative adversarial networks (GANs) 4
 on Atari images 74–80
generator 74
genetic algorithms (GA) 507
 implementing, on CartPole 508, 509
GPU tensors 56–58
gradients 58–63

- approaches 59
- graphics processing unit (GPU)** 145, 356
 training 625
- Gym API functionality**
- environment, rendering** 47, 49
 - wrappers 44–47
- Gymnasium** 27, 31, 33
- actions 34
 - CartPole session 40–42
 - environment 36–38
 - environment, creating 38–40
 - observations 34–36
 - random CartPole agent 43
- H**
- HalfCheetah**
- ES, implementing on 500–505
 - results 505, 506
- height, width, channel (HWC)** 151
- human demonstrations** 420, 421
- recording 421, 422
 - results 425, 426
 - training with 423–425
- I**
- Ignite helpers** 190
- independent and identically distributed (iid)** 6, 143
- inplace operations** 54
- interactive fiction** 355–357, 359
- Internet Engineering Task Force (IETF)** 397
- K**
- Keras-RL library** 194
- Kullback-Leibler (KL) divergence** 462
- L**
- large language models (LLMs)** 145, 541, 546
- training, stages 546
- long short-term memory (LSTM) model** 366
- loss function** 68
- M**
- machine learning (ML)** 3
- MAgent** 645
- installing 646
 - random environment, setting up 646–652
- Markov chain** 14
- Markov decision process (MDP)** 14, 144, 299
- actions, adding 22–24
 - Markov process 14–18
 - Markov reward processes 19–22
- Markov decision processes (MDP)** 122
- minimax** 572
- MiniWoB benchmark** 400, 401
- MiniWoB++** 401
- actions 402
 - example 402, 404, 405
 - installation 401
 - observations 402
- mirrored sampling** 495
- model-based methods** 88
- for board games 571, 572
 - versus model-free methods 570, 571

Monte Carlo sampling 497
 Monte Carlo tree search (MCTS) 572, 616
 MountainCar experiments 524, 525
 DQN + epsilon greedy 525, 527
 DQN + noisy networks 527
 DQN + state counts 528, 529
 PPO + network distillation 534–536
 PPO + noisy networks 532, 533
 PPO + state counts 533, 534
 PPO method 530, 531
 MuJoCo 462
 multi-agent RL 644
 environment 645
 MuZero 591, 592
 high-level model 592, 593
 implementing, for Connect 4 594
 results 609, 610
 training process 593, 594

N

N-step DQN 210–212
 hyperparameter tuning 215
 implementation 213
 results 213, 214
 natural language processing (NLP) 364
 Neural Information Processing Systems (NIPS) 366
 neural network (NN) 365, 616
 architecture 622
 building blocks 63, 64
 noisy networks 219, 220, 522
 hyperparameter tuning 225
 implementation 220, 221, 223
 results 223, 224
 non-stationarity 18

novelty search 510
 NumPy 31

O

observations, tweaking 382
 objective 386
 relative actions 384, 385
 visited rooms, tracking 382, 383

off-policy methods 89
 on-policy methods 89
 OpenAI Gym API 33
 OpenCV Python bindings 31
 optimality 114–116
 Bellman equation 116–118
 optimizers 68–70
 Ornstein-Uhlenbeck (OU) process 438

P

partially observable Markov decision process (POMDP) 144, 363, 412
 policy gradient methods 299, 302
 CartPole environment 313
 on Pong 319, 320
 REINFORCE issues 310
 REINFORCE method 302, 303
 representation 301
 significance 300, 301
 policy gradient methods, CartPole
 implementing 313, 314
 policy gradient methods, on CartPole
 implementing 315
 results 316–318
 policy gradient methods, on Pong
 implementing 320, 321

results 321
policy-based methods 88
 versus value-based methods 310
PolicyAgent 179, 180
Pong
 policy gradient 319, 320
Pong environment 144
prediction-based methods 524
prioritized replay buffer 225, 226
 hyperparameter tuning 232, 233
 implementation 226–230
 results 231, 232
probability distribution 301
proximal policy optimization (PPO) 469,
 470
 implementation 470–473
 results 474–476
PTAN 32
PTAN CartPole solver 190–193
PTAN library 172, 173
 entities 173
PTAN library, entities
 action selector 174, 176
 agent entity 176, 177
 experience replay buffers 186, 187
 experience source classes 180, 181
 Ignite helpers 190
 TargetNet class 188, 189
PyBullet 462
PyTorch 31
 graph computation 258–261
PyTorch Ignite 31, 80, 81
 concepts 81, 82
 used, for GAN training on Atari 82,
 84, 85

Q
Q-iteration
 for FrozenLake 130, 131
R
random CartPole agent 43, 44
Ray library 194
ReAgent library 194
rectified linear unit (ReLU) 64, 151
recurrent neural network (RNNs) 69, 365,
 366
REINFORCE method 299, 302, 303
 CartPole example 304–308
 issues 310–312
 policy-based methods, versus
 value-based methods
 310
 results 308–310
reinforcement learning (RL) 3, 5, 6
 baseline 256–258
 benchmark results 271
 complications 6, 7
 graph computation, in PyTorch
 258–261
 playing and training, in separate
 processes 264–268
 several environments 261–263
 speed, significance 253–256
 wrappers, tweaking 268–271
reinforcement learning (RL) algorithm 113

reinforcement learning with human
 feedback (RLHF) 541,
 546, 547
replay buffer 143

-
- residual NN (ResNet)** 65
reward functions
 in complex environments 542, 543
reward model
 A2C, combining with 560–562
 training 556–558
RL communication channels
 actions 10, 11
 observations 11–14
 reward 8, 9
RL entities
 agent 9
 environment 10
RL libraries
 list, considerations 193, 194
 need for 172
RL methods
 model-free or model-based 88
 taxonomy 88
 value-based or policy based 88
RL, theoretical foundations
 Markov decision process (MDP) 14
 Markov decision process (MDPs) 14
 policy 25
RLHF experiments 547
 A2C, combining with reward model
 560–562
 fine-tuning, with 100 labels 563, 564
 initial training, with A2C 548–552
 process, labeling 552, 553, 555, 556
 results 567
 reward model training 556–558
 second round of 564, 565
 third round of 565–567
RLHF method
 overview 544–546
 theoretical background 544
Rubik's cube 614, 615
 2 × 2 cube, experiment results 637, 638
 3 × 3 cube, experiment results 639, 640
 code outline 628
 cube solving approaches 616
 experiment results 635, 636
 improvements 641
 model application 624–626
 optimality 615
 results 626
 training process 621
- S**
- sample efficiency** 128
scalar tensors 55, 56
self-play 575
simple clicking approach 406
 grid actions 406–408
 limitations 412–414
 model and training code 410
 RL part 409
 training results 410, 411
soft actor-critic (SAC) 461, 483, 484
 implementation 484–486
 results 486–488
speed 253–256
Spinning Up library 194
Stable Baselines3 library 146, 193
state 114–116
state space 14

- stationarity 18
- stochastic gradient descent (SGD) 142, 461
- supervised learning 4
- T**
- tabular Q-learning 133, 135–139
- target network 144
- TargetNet class 188, 189
- tensor 52
- creating 52, 53, 55
 - GPU tensors 56–58
 - operations 54, 56
 - scalar tensors 55, 56
- TensorBoard
- metrics, plotting 72–74
 - monitoring with 70, 71
- TensorBoard 101 71, 72
- tensors 60–63
- text description
- adding 414
 - implementation 414–418
 - results 419, 420
- TextWorld environment 359
- game generation 360, 361
 - game information 362–364
 - installation 360
 - observation and action spaces 361, 362
- TextWorld environment, problem solving 369–371
- agent 378
 - code, training 379, 380
 - DQN model 378
 - embedding and encoders 376–378
- observation preprocessing 371–376
- results, training 380–382
- TF-Agents library 194
- TorchRL 193
- trading 273, 274
- convolution model 293, 294
 - data 276, 277
 - environment 277–285
 - feed-forward model 288–291, 293
 - key decisions 274, 275
 - models 285–287
 - problem statement 274, 275
 - training code 287
- training process, Rubik’s cube
- architecture 622
 - training 623
- training process, Rubik’s cube architecture 622
- transformer models 369
- transformers 387–389
- transition matrix 15
- trust region optimization extension 462
- Trust region policy optimization (TRPO) 476, 477
- implementation 477, 478
 - results 479, 480
- U**
- unsupervised learning 4
- V**
- value 114–116
- of action 118–121
 - of state 20
- value iteration 121–123, 134

value iteration method
 best practices 123–129

value-based methods
 versus policy-based methods 310

variance reduction 324, 325

Virtual Network Computing (VNC) 400

W

web navigation
 evolution 397, 398

web scraping 398

website testing 398

word embeddings 367, 368

word2vec 367

Wrapper class 44–46

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere? Is your eBook purchase not compatible with the device of your choice?

Don't worry; with every Packt book, you now get a DRM-free PDF version of that book at no cost.

Read anywhere, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there! You can get exclusive access to discounts, newsletters, and great free content in your inbox daily.

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below:



<https://packt.link/free-ebook/9781835882702>

2. Submit your proof of purchase.
3. That's it! We'll send your free PDF and other benefits to your email address directly.

