April 2025 - September 2025

CIMNE
International Centre
for Numerical Methods in Engineering

Open Arms project: internship report

Improvements in the neural network code, migration of the code to Julia and study of the possibility of optimizing with reinforcement learning.

Joel Sendra Revuelta

Tutors: Àlex Ferrer Ferre, Xavier Martínez Garcia

CIMNE - UPC

Abstract

This report summarizes the work carried out during my internship, providing a record of the progress achieved and serving as a reference for future contributors to the project, to facilitate continuity. It focuses on three main areas: the development of a predictive neural network for fuel consumption, the migration of existing Matlab code to Julia, and the initial exploration of reinforcement learning for fuel optimization. The report discusses the current state of these components, challenges encountered, and potential directions for further work. It is intended as a review of project advances rather than a code tutorial, and assumes the reader has some familiarity with the codes used.

Contents

1 Neural network code

Before analyzing the performance of the predictive neural network, it is useful to understand how the code is structured. For this purpose, one of the first steps done was to create a UML diagram of the implementation. The diagram provides a clear overview of the main components of the code, their public methods, and how they interact with each other. Having this holistic view is important, since the code is a complex net of classes and functions that weave together to determine how data is processed, how training is carried out, and how results are stored. The UML diagram is a structural map that serves as a reference throughout any analysis of the code, helping visualize it as a whole and identify how one issue may influence more than one branch of the network. Figure 1.1 shows a general picture of it, but the picture can be found in the OpenArmsTutorial folder in Swan-Tutorials.

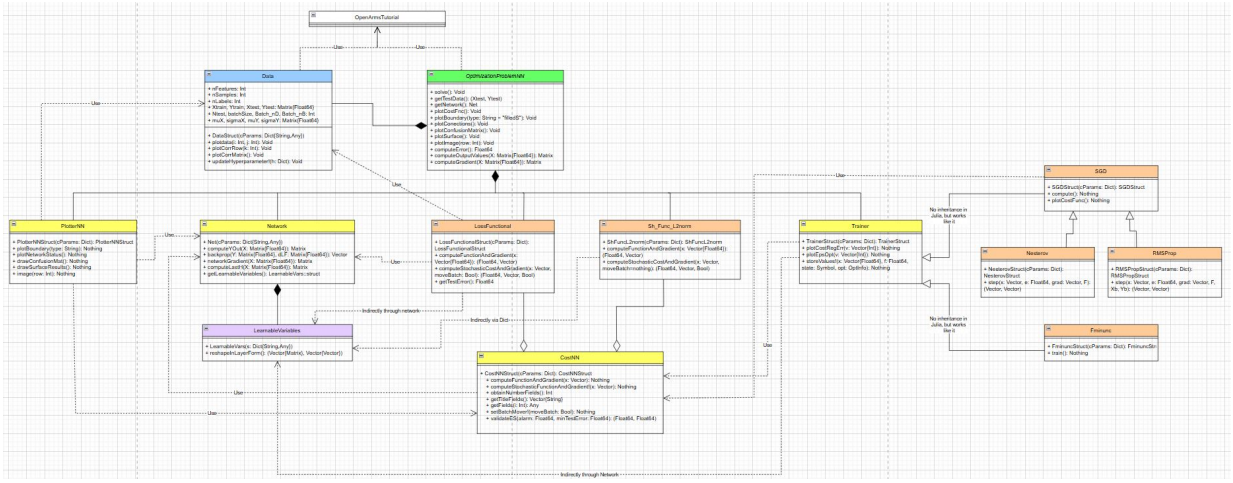


Figure 1.1: UML diagram of the neural network

The predictive neural network is, as of now, the most important asset of the project. It is, essentially, a black box that takes environment data like vessel speed or wind direction as inputs, and returns a prediction of what the fuel consumption of the ship should be under those conditions. The way a predictive neural network like this one works is the following: it is given a dataset to train with and, once training is finished, it is tested with cases that it has not seen during training to test its performance. The code was already written when I joined the project, but it was not returning good predictions in the testing part, as figure 1.2 shows.

It has to be noted that we worked with a fairly limited dataset. The data used during this phase was measured at the Barcelona vessel, not at the Open Arms ship. Only so many data points were obtained (around 300), and they did not spread out uniformly across the whole range of some inputs. For instance, rpm values made jumps of 200 rpm; there were some samples at 1200 rpm, then some at 1400 rpm, and so on, but nothing in the middle. As of now, the data processing of a new dataset, with values measured at the Open Arms ship, has just been finished by Desirée.

Rpm	Windy (deg)	Windy (m/s)	Speed (m/s)	Yaw	Pitch	Roll	Cons. real	Cons. prediction	Difference
1600	133	1.4404	16.387	31.926	0.46644	1.4478	1.013	0.38617	0.62683
1400	118	2.3664	10.648	-28.32	-0.67399	-0.68811	0.62338	1.0222	-0.39883
1600	118	2.8294	12.649	-32.319	2.2165	2.0447	1.013	0.38617	0.62683
1800	118	1.8006	17.576	-31.406	1.2544	2.8216	2.1818	0.29962	1.8822
1400	133	0.926	12.009	33.336	0.9133	1.3427	0.62338	1.0222	-0.39883
1600	118	2.5722	13.652	-31.415	1.1061	2.3899	1.013	0.38617	0.62683
1200	133	1.5948	8.6151	29.044	-0.34359	4.6013	0.7013	0.87072	-0.16942
2000	118	1.6462	30.08	-31.795	2.4424	3.6005	3.7403	4.4347	-0.69442
2000	118	2.3664	29.504	30.602	0.59111	0.39906	3.7403	4.4347	-0.69442
1800	118	0	17.174	33.077	-0.30307	2.4898	2.1818	0.29962	1.8822
1200	118	1.9549	4.4921	-24.21	2.1483	4.0357	0.7013	0.87072	-0.16942
1600	133	0.72022	14.349	33.078	-0.39943	6.0665	1.013	0.38617	0.62683
1600	133	0.25722	12.649	31.493	0.80972	1.4388	1.013	0.38617	0.62683
1400	118	2.3664	5.0884	-30.168	0.5278	-0.56208	0.62338	1.0222	-0.39883
1400	133	1.8006	15.625	35.489	0.60817	4.8449	0.62338	1.0222	-0.39883
1600	118	2.778	12.649	-31.915	-0.41661	0.69118	1.013	0.38617	0.62683
1800	118	0	12.009	30.254	-0.14446	-1.4138	2.1818	0.29962	1.8822
2000	118	2.0063	26.731	-34.085	0.1249	0.93481	3.7403	4.4347	-0.69442

Figure 1.2: Consumption predictions done by the neural network before any improvements. Starting from the right, column 3 shows the real, measured consumption in each case; column 2 shows what the neural network predicts it to be.

In light of such a discrepancy between real and predicted values, one may speculate that the issue lies not only in aspects to improve — such as normalizing the data or ensuring the right inputs are being used — but also in the possibility of a major bug in the code.

All the results discussed in this section have been achieved using the network architecture and setup shown in figure 1.3.

```
%% Initialization of hyperparameters
pol_deg         = 1;
testratio       = 20;
lambda          = 0.0;
learningRate    = 0.2;
hiddenLayers    = 128 .* ones(1, 6);
```

```
%% INITIALIZATION
% Store dataset file name
s.fileName = 'Resultados2.csv';

% Load model parameters
s.polynomialOrder = pol_deg;
s.testRatio       = testratio;
s.networkParams.hiddenLayers    = hiddenLayers;
s.optimizerParams.learningRate  = learningRate;
s.optimizerParams.maxEpochs = 1000;
s.costParams.lambda             = lambda;
s.costParams.costType           = 'L2';

s.networkParams.HUtype = 'ReLU';
s.networkParams.OUtype = 'linear';
```

Figure 1.3: Network architecture used.

1.1 Normalization of the data

One likely problem causing bad results in a predictive neural network is data not being normalized. This causes the model difficulty finding the correct weights, since a big weight for one variable may be small for another one. At the end of the day, neural networks are just a huge generalization of linear maps. Think of the following example:

Example 1.1: Jack's house pricing Jack is a property agent, and uses a neural network to predict the price of a house in € as a function of two inputs: house size in m^2 and number of bedrooms. But Jack is frustrated; the predictions of his network are not too precise. One day, his colleague Richard comes to help Jack on his work, and he immediately identifies the issue. He asks Jack:

- Jack, what is the average surface of the houses you sell?

- Around $200\ m^2$ — he answers

- And how many bedrooms do these houses usually have?

- Normally 4 bedrooms.

Richard smiles and explains:

- Here is the problem: the two inputs are on completely different scales. House size is usually around $200\ m^2$, while the number of bedrooms is only about 4. Because the network is not scale-invariant, it gives much more importance to the input with larger numerical values, in this case the surface area, and almost ignores the number of bedrooms.

- The solution is simple: normalize both inputs so that they are on a comparable scale (for example, between 0 and 1, or with zero mean and unit variance). Once this is done, the neural network can learn the contribution of each feature properly, and your predictions will improve significantly.

In the Open Arms neural network, there are 7 inputs. One of them is rpm, of the order of 1000 rpm. Another one is pitch angle, of the order of 1º. Since the scale of rpm is much larger, the model would nearly ignore other inputs like pitch angle unless the data gets normalized. The chosen normalization method was zero mean and unit variance. That was done by adding the corresponding lines of code to the *splitData()* function in the *Data* class, as shown in figure 1.4.

```
% Normalize X
[obj.Xtrain, obj.muX, obj.sigmaX] = zscore(obj.Xtrain);
obj.Xtest = (obj.Xtest - obj.muX) ./ obj.sigmaX;

% Normalize Y
[obj.Ytrain, obj.muY, obj.sigmaY] = zscore(obj.Ytrain);
obj.Ytest = (obj.Ytest - obj.muY) ./ obj.sigmaY;
```

Figure 1.4: Normalization of the 7 inputs (X) and the output (Y) before training.

Note that the dataset is normalized before training. After testing, it will be denormalized, in order to obtain real-world values instead of normalized units. Denormalization simply applies the inverse of the normalization transformation. These are the corresponding lines in the main script:

```
% Denormalization
Xtest = Xtest .* data.sigmaX + data.muX;
Ypred = Ypred .* data.sigmaY + data.muY;
Ytest = Ytest .* data.sigmaY + data.muY;
```

Figure 1.5: Code for denormalization of the output for a clear presentation of the results.

1.2 Discovery of the major bug

During the early weeks of the internship, the focus was on carrying out an extensive inspection of the code, class by class, in order to identify the issues leading to inaccurate results. To do so, two types of plots were used for diagnosis: cost function evolution plots and prediction histograms. See figures 1.6 and 1.7.

Cost function evolution plots make it possible to observe how cost is decreasing throughout epochs, and whether or not it is converging to a small value — not necessarily 0. Prediction histograms give us an image of the distribution of predictions done by the network. Comparing them to real consumption histograms, one can get a general vision of whether predictions are biased or overly variable.

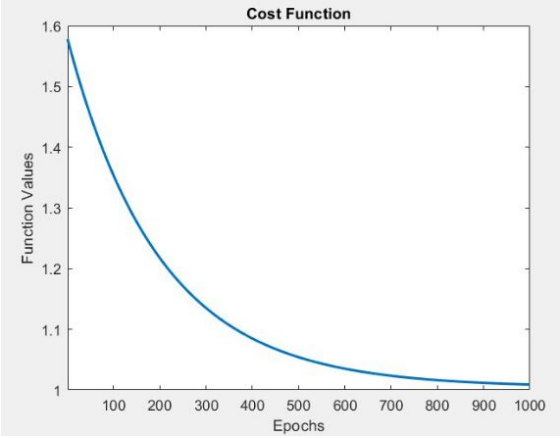


Figure 1.6: Example of a cost function evolution plot.

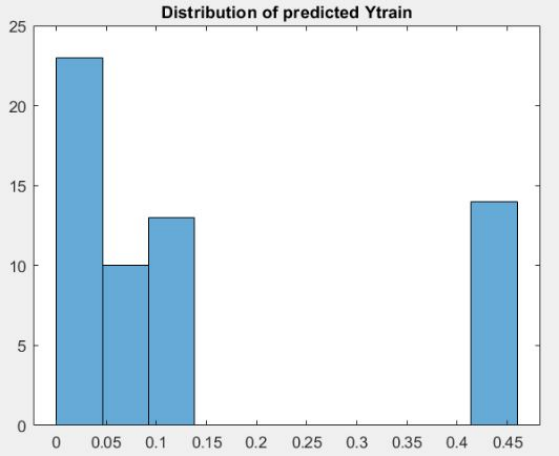


Figure 1.7: Example of a prediction histogram. X values are normalized (see section 1.1).

Despite normalization, both plots indicated that a significant problem persisted. Figure 1.6 shows a cost function converging to a value that is too large, while figure 1.7 shows a distribution that does not match the dataset. Moreover, the mean square error (MSE) of the prediction was around 0.9 — which is huge for values of the order of 1. These results strongly suggested the presence of a major bug in the code.

The issue was eventually identified: a single line in the main script was calling the wrong function during prediction. Instead of retrieving the neural network's output — the predicted consumption — the code was retrieving the cost function. Although predictions were being computed, they were not correctly passed to the main script. The problem was quickly resolved by eliminating the wrong function call and replacing it with the correct one, *computeYOut()*.

```
% Pass every Xtest sample to the neural network and store predictions
for i = 1:size(Xtest, 1)
    Ypred(i) = network.forwardprop(Xtest(i, :), Ytest(i, :));  % Prediction for each case
end
```

Figure 1.8: Major bug: mistaken call of function *forwardprop()* in the main script.

```
% Pass every Xtest sample to the neural network and store predictions
for i = 1:size(Xtest, 1)
    Ypred(i) = network.computeYOut(Xtest(i, :));  % Prediction for each case
end
```

Figure 1.9: Major bug solved

Once this was found, predictions being so off suddenly made sense. The code was not predicting anything, but returing values of the cost function and assigning each of them to one sample. Using the proper function *computeYOut()*, and without introducing any other changes or optimizations in network architecture, an immense improvement in the predictions' precision was achieved, as shown in

figure 1.10. The cost function also improved noticeably, as shown in figure 1.11. Finally, the distribution of predictions rightly resembled the distribution of real consumptions, as shown in figure . MSE was 0.02, compared to the 0.9 before resolving the bug.

Cons. real	Cons. prediction
1.013	0.949
2.1818	2.3098
0.7013	0.71129
3.7403	3.6361
0.7013	0.71129
0.7013	0.71129
1.013	0.89748
3.7403	3.673
0.62338	0.71129
0.7013	0.71129
1.013	0.7504

Figure 1.10: Real and predicted consumptions right after the major bug was solved.

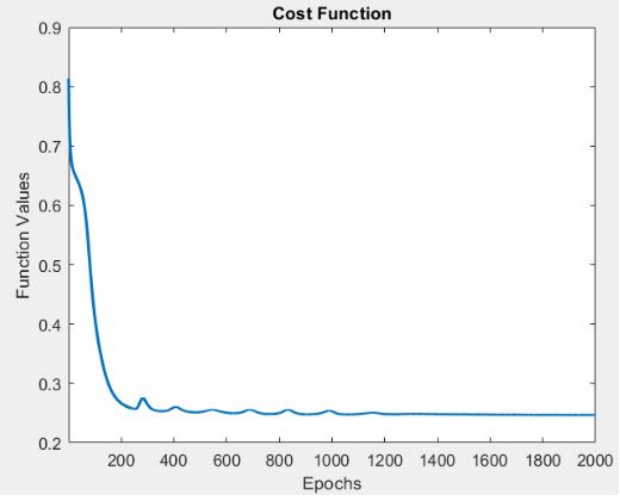


Figure 1.11: Cost function evolution right after the major bug was solved.

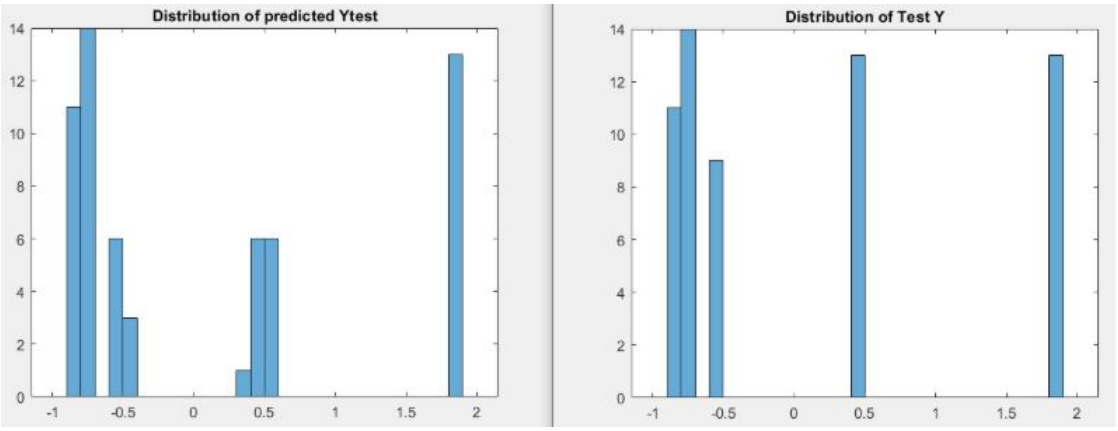


Figure 1.12: Comparison of the distribution of predictions (left side histogram) to the real distribution (right side histogram).

1.3 Are we using the right inputs?

1.3.1 Pitch, roll and yaw

Three out of the seven inputs that the model receives are pitch, roll and yaw. Even though these three do likely have a significant effect on consumption, the dependency on them may not be as direct as for other magnitudes. The ship oscillates continuously, with oscillation frequencies containing several harmonics that span a wide range of values. The main oscillation period is generally on the order of 5 to 10 seconds, possibly somewhat longer for roll than for pitch, as suggested by observations during the trip with the Open Arms.

This means that, over 5-10 seconds, a full period of oscillation is completed. Consequently, pitch and roll values change significantly every few seconds. Such rapid variations are unlikely to directly affect fuel consumption; rather, consumption is expected to be influenced by the oscillation frequencies themselves. A possible approach would be to translate the data into the frequency domain. Using discrete Fourier transforms (DFT or FFT), one can obtain a spectrum of the data. That spectrum would ideally have peaks at the dominant oscillation frequencies. Those frequencies could be used as inputs instead of the raw pitch, roll and yaw.

In principle, data will be available from the accelerometers at a rate of at least one sample per second, allowing this approach. The data could be divided in sets of 1 minute each, then transformed via an FFT, to identify the dominant oscillation frequencies during that period. This would create a new dataset of pitch, roll and yaw frequencies with one sample per minute — likely better aligned with the timescale of changes in fuel consumption — rather than angles updating every second.

This was an idea that came up during our analysis of the inputs of the model, but it has not been implemented. I leave the idea documented here for future participants of the project to explore it in case they find it interesting.

1.3.2 Vessel speed

Keeping an eye on theoretical, physical models of the subject one is studying is always a good practice when working with neural networks and computational algorithms in general. Consulting physics books, one can identify a direct relation between engine power and speed in any classic moving vehicle:

$$P = \frac{1}{2}\rho v^3 A C_{TS} \tag{1.1}$$

Where ρ is the density of the medium that the vehicle is moving through, A is usually a projected area, and C_{TS} is a friction coefficient. This formula provides crucial insight: it tells us that power — and therefore consumption — depends (close to) cubically on speed. While a neural network is, in

principle, capable of learning such nonlinear relationships from raw inputs, its performance and efficiency can often be improved by incorporating prior knowledge. In this case, that means providing the network with speed cubed as an input, instead of raw speed, thus reducing the complexity of the relations that it needs to learn from the data.

```
% Velocity cubed
obj.Xtrain(:,4) = obj.Xtrain(:,4).^3;
obj.Xtest(:,4) = obj.Xtest(:,4).^3;
```

Figure 1.13: Code addition to *splitData()*, in the Data class, to implement this idea.

1.3.3 Wind direction

The model used wind direction in degrees as input, likely with respect to the direction of movement of the ship. This is not ideal because degrees are circular. A small change in the direction of wind, for example from 1º to 359º (a 2º shift), would appear huge to the network, which would interpret the two values as being at opposite extremes. Using raw wind direction as an input is therefore problematic.

The highest consumption happens when wind comes from the front (0º), and the lowest when wind comes from the rear (180º). Dependence on lateral wind may be more complex. With this in mind, and in the same effort to help the network capture the relationship between input and output more effectively, the better approach is to convert wind direction into its cosine, and use that as the input.

```
% Wind direction as a cosine
obj.Xtrain(:,2) = cos(obj.Xtrain(:,2)*pi/180);
obj.Xtest(:,2) = cos(obj.Xtest(:,2)*pi/180);
```

Figure 1.14: Code addition to *splitData()*, in the Data class, to implement this idea.

These additions improved the performance of the network, yielding even more accurate prediccions and reducing the MSE to less than 0.001 in the best runs.

Cons. real	Cons. prediction
3.7403	3.7404
1.013	0.98139
0.62338	0.60922
3.7403	3.7404
0.62338	0.61498
0.62338	0.61411
0.7013	0.69197
0.7013	0.69714
0.62338	0.62581
1.013	1.0464
0.62338	0.61319
0.62338	0.62036
3.7403	3.7404
0.7013	0.71269
2.1818	2.1142
2.1818	2.1637
2.1818	2.2338

Figure 1.15: Real and predicted consumptions after the cubed speed and cosine of wind direction additions.

1.3.4 Revolutions per minute

A fundamental problem was identified at the very beginning of the analysis of the code: in the dataset used, fuel consumption is calculated directly from rpm, using the engine manufacturer curve. Despite the relationship being nonlinear, this results in the two values being perfectly correlated.. With this dataset, using rpm as one of the inputs is redundant, and reduces the network's task essentially to fitting a cubic function.

However, that issue only arises from the limited dataset that we have been using. It can be easily resolved by measuring consumption independently of rpm, with a flowmeter. Until that is done, though, it should be noted that using rpm as an input is inconsistent.

2 Migration of the code to Julia

The second segment of tasks carried out this internship consisted on the migration of the neural network code from Matlab to Julia. The idea of translating the code to Julia is not original from this project, but part of a bigger plan of slowly migrating all the research group's codes from Matlab to Julia. This was the first full trial of that plan.

The motivation to translate the whole code to Julia comes from various potential benefits of the language with respect to Matlab. Firstly, its speed of execution, which is comparable to C in many cases, and ideal for intensive calculations (machine learning, optimization). Secondly, its simple and modern syntax, comparable to Python and Matlab itself, and very intuitive for mathematical formulation and engineering. Finally, the fact that it is open source, unlike Matlab which needs a license, and increasingly being used in the scientific community.

2.1 Class by class migration: method 1

The modus operandi of the this first method of translation is the following. We translate the class to a Julia module. Translation in and of itself is pretty clean. However, to check that everything works correctly, one needs to run the original Matlab with the new module from Julia. That way, debugging can be done class by class and not all at the same time at the end.

This approach allows for a clean and structured migration, but it requires an additional layer: Matlab to Julia communication, i.e. Matlab being able to call a Julia script in real time. That significantly increases the complexity, since Matlab does not provide a built-in way to call Julia — like it does for other languages like Python. As a result, communication has to be achieved in a manual manner. After doing some research on the topic —– and confirming that all approaches would be nontrivial —–, using JSON (JavaScript Object Notation) as an intermediary emerged as the most suitable solution. Communication via JSON is achieved through three steps.

2.1.1 Step 1: JSON auxiliary files, one per public method

The first one consists of creating an intermediate, JSON-using file for each method that needs to be called. Those auxiliary files are all named using a unified structure: *call_CLASS_METHOD*. Here is an example of how one of those would look like:

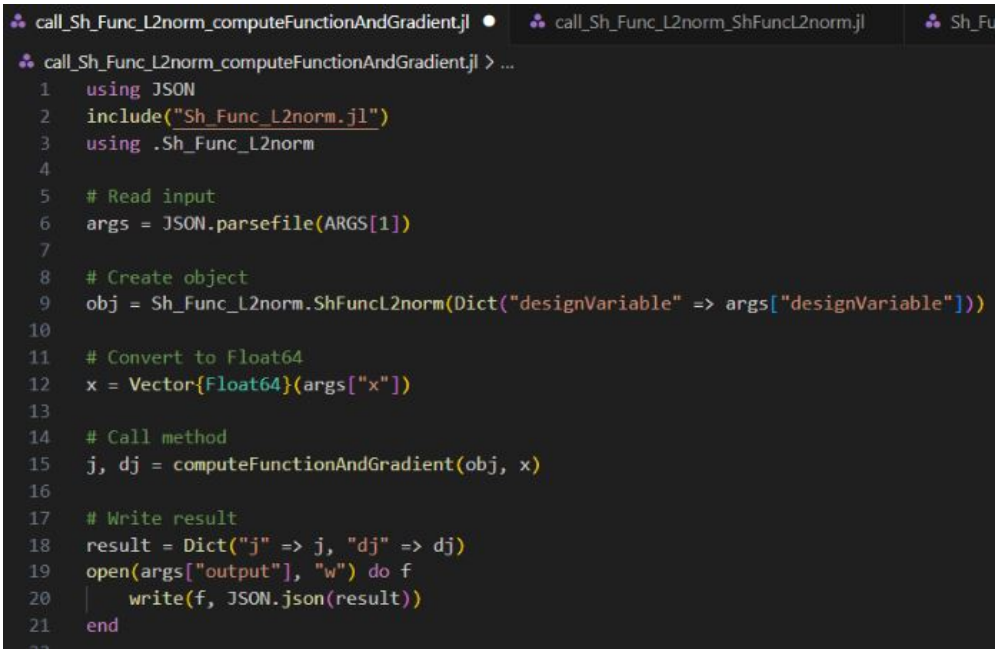


Figure 2.1: JSON auxiliary file for method *computeFunctionAndGradient()* of class *Sh_Func_L2norm*.

Here is what one of these files does, step by step:

1. Imports the JSON package.
2. Loads the module that defines our Julia class (in this example, the Sh_Func_L2norm class), and its associated methods. Think of this as importing a source file.
3. Brings the module into the namespace so its methods can be used directly. The dot indicates it's defined locally, not as a system package.
4. Reads a JSON input file passed as a command-line argument from MATLAB. ARGS[1] is the filename (e.g., *julia_input.json*, as we will see later on). The resulting variable *args* is a Julia dictionary holding fields like *designVariable*, i.e. an equivalent to the properties of the Matlab class.
5. Constructs a ShFuncL2norm object. That's the equivalent in MATLAB language to the obj. structure of the class.
6. Converts the input vector x — which is read as a generic array by JSON — into a vector of floats, which is the type that the methods in the Julia module expect.
7. Calls the method.
8. Creates a dictionary with the results.
9. Writes the result into an output JSON file
10. This file is later read by MATLAB to retrieve the information saved in the result dictionary.

2.1.2 Step 2: Function to call Julia methods from Matlab

Once the JSON auxiliary files have been created, their information needs to be retrieved from Matlab. That process is done by a single Matlab function, *callJuliaClass*, which decodes the information stored in the output JSON file and stores it in Matlab language. Function *callJuliaClass* takes, as inputs, the name of the class, the name of the method and the input parameters of that method.

```
function output = callJuliaClass(className, methodName, params)
    % Path to Julia scripts folder
    currentScriptPath = fileparts(mfilename('fullpath'));
    juliaScriptDir = fullfile(currentScriptPath, className);

    % Create filenames
    inputFile = 'julia_input.json';
    outputFile = 'julia_output.json';

    % Add output field so Julia knows where to write
    params.output = outputFile;

    % Write input JSON
    fid = fopen(inputFile, 'w');
    fwrite(fid, jsonencode(params));
    fclose(fid);

    % Build Julia command
    juliaScriptName = ['call_' className '_' methodName '.jl'];
    juliaScript = fullfile(juliaScriptDir, juliaScriptName);
    command = ['julia ' juliaScript ' ' inputFile];
    system(command);

    % Read output JSON
    fid = fopen(outputFile, 'r');
    raw = fread(fid, '*char')';
    fclose(fid);

    output = jsondecode(raw);

    % Delete JSON files
    delete(inputFile);
    delete(outputFile);
end
```

Figure 2.2: Function *callJuliaClass*

1. First 2 lines determine the folder of the class and create a path to it.
2. Next 2 lines define the filenames for the input/output JSON files used to communicate with Julia.
3. Next line adds a field to the params struct so Julia knows the name of the file where it should write the result.
4. The params struct is converted to a JSON string using *jsonencode*. It is written into a file (julia_input.json) so Julia can read it.
5. The name of the JSON auxiliary script is constructed, following the unified structure based on the class and method names.
6. The full system command is built.

7. The script is executed using MATLAB's system command (calls Julia script through the terminal).
8. After Julia has run, the output file created by Julia is openned. It is read into Matlab as a string.
9. The raw JSON string is parsed into a Matlab array, and returned as *output*.
10. The temporary files are deleted.

2.1.3 Step 3: Equivalent Matlab class that uses the Julia module

One would want to keep the MATLAB code as unchanged as possible, so that switching back an forth from the MATLAB class to the Julia class can be done with ease. To do so, the method is to create an equivalent class file *JuliaCLASS.m* which does the same job as the original but uses the Julia methods instead.

```
classdef JuliaShFuncL2norm < handle
    properties (Access = private)
        data  % Struct returned from Julia constructor (holds designVariable info)
    end

    methods (Access = public)
        function obj = JuliaShFuncL2norm(params)
            % Call Julia constructor and save returned data
            obj.data = callJuliaClass('Sh_Func_L2norm', 'ShFuncL2norm', params);
        end

        function [j, dj, isBD] = computeStochasticCostAndGradient(obj, x, moveBatch)
            % Optional moveBatch argument
            if nargin < 3
                moveBatch = [];
            end

            % Package input parameters
            params.designVariable = struct('thetavec', obj.data.thetavec);
            params.x = double(x(:));
            params.moveBatch = moveBatch;

            % Call Julia method
            result = callJuliaClass('Sh_Func_L2norm', 'computeStochasticCostAndGradient', params);

            % Return outputs
            j    = result.j;
            dj   = result.dj';
            isBD = result.isBD;
        end

        function [j, dj] = computeFunctionAndGradient(obj, x)
            params.designVariable = struct('thetavec', obj.data.thetavec);
            params.x = double(x(:));

            result = callJuliaClass('Sh_Func_L2norm', 'computeFunctionAndGradient', params);

            j  = result.j;
            dj = result.dj;
        end
    end
end
```

Figure 2.3: Equivalent Matlab class for calling Julia's *Sh_Func_L2norm* from Matlab.

This is not complex. It's basically the same as the original Matlab class, but here each method calls its Julia equivalent through *callJuliaClass*.

Method 1 is very effective and allows the user to do a clean debugging process. However, it is slow both to set up — it requires a considerable number of scripts to be created — and to execute, as Matlab–Julia communication takes a couple of seconds each time. It is therefore recommended to use

this method only for pivotal classes, where all errors must be eliminated because they would propagate to many other parts of the code.

2.2 Class by class migration: method 2

A second migration method exists, avoiding the main drawback of Method 1, namely Matlab–Julia communication. This simpler approach consists of translating the code class by class and then debugging each class directly with a test script in Julia. No interaction with Matlab is required.

Notice that, in practice, most classes in object-oriented programming (OOP) cannot be isolated, since they depend on other classes. Therefore, order of translation becomes crucial in this method. The first classes to be translated should be those that have no dependencies themselves, but upon which many other parts of the code rely. The UML diagram of the code can serve as a valuable guide in this process.

There may also be cases where no hierarchy allows classes to be translated one by one, no matter what the order is. In those cases, two or more classes may be migrated and tested together, as a unit. Keep in mind that translating one class at a time is not a requirement, but rather a strategy to simplify debugging.

Lastly, notice that, even when crucial information is lacking — likely because of a needed class not having yet been migrated — dummy data[1] can be created and used for testing. Classes that have been tested using dummy data shall be tested back once the whole migration is complete, since some errors may remain. Generally, though, around 90 % of bugs are found already with this trick.

Method 2 offers clear advantages and disadvantages. Its main benefit is that it is substantially faster than method 1, both to set up and to execute. It also still allows for a localized debugging process, and avoids an overload of auxiliary archives. In the negative side, testing is less precise and may miss some bugs that only appear once the full code is run. This method also requires the user to have good knowledge of the dependencies between classes. It is therefore recommended to use method 2 for the majority of classes, except for those that are central for the code and must be debugged with precision. Test scripts can be found in the *Tests* directory, in *OpenArmsTutorial - Julia*.

[1] **Dummy data** refers to benign data that does not contain any useful information, but can be used as a placeholder for both testing and operational purposes.

2.3 Gridap-style code

It must be mentioned that a second version of the neural network in Julia was coded. This second version tries to follow the style of Gridap[2], is faster and has arguably a better legibility. Both versions of the code can be used indistinctively. The initial one, in the *Julia* directory, mirrors the Matlab code better, and has an OOP-heavy style. Users that are unfamiliarized with Julia may understand that version better. The new version, in the *Gridap* directory, is more optimized and written in a more Julian manner.

3 Optimization: reinforcement learning

The third and final phase of this internship was devoted to learning an optimization method to find a strategy for minimizing fuel consumption in the Open Arms. The method chosen was reinforcement learning, a machine learning (ML) technique that trains an agent to make decisions that maximize performance. The work carried out on this topic was primarily formative: a well known reference on the field, Sutton & Barto, was followed, and several of the book's examples were coded in Matlab as a learning exercise. Some of the information in this section directly cites Sutton & Barto, and the vast majority of it is based on the book.

The purpose of this section is to provide a summary of the main concepts learned and of the examples implemented, so that readers can grasp the essentials of the topic without having to study the entire book.

3.1 Diccionary of key reinforcement learning concepts

- **Agent:** The learner or decision maker that interacts with the environment to maximize cumulative reward.
- **Environment:** The world with which the agent interacts. Provides observations, rewards, and transitions in response to agent actions.
- **State (s):** A representation of the current situation of the agent within the environment. Can be fully observable (Markov) or partially observable.
- **Action (a):** The choice made by the agent at each state. Determines how the agent affects the environment.

[2] **Gridap** is a an extensible finite element toolbox, exclusively written in the Julia programming language, for the numerical simulation of a wide range of mathematical models governed by partial differential equations (PDEs). It is characterized by allowing very clear mathematical formulations.

- **Policy (π):** A mapping from states to actions (deterministic) or to probabilities of actions (stochastic). Defines the "strategy" of the agent.
- **Reward (r):** A scalar feedback signal from the environment after each action. Guides learning by indicating how good a certain state-action pair is.
- **Return (R):** The total accumulated reward from time t onwards. Often defined as the sum of discounted rewards:
$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots$$
- **Discount factor (γ):** Determines the importance of future rewards compared to immediate ones. Satisfies that $0 \leq \gamma \leq 1$.
- **Value function (v_π):** Expected return starting from state s and following a certain policy. Indicates how good it is to be in a state.
- **Episode**: A complete sequence of states, actions, and rewards in an environment, starting from an initial state and ending in a terminal state.

3.2 What is reinforcement learning

Reinforcement learning (RL) problems involve learning what to do so as to maximize a numerical reward signal. The learner wants to do good to obtain a prize / avoid a punishment. In an essential way they are closed-loop[3] problems because the learning system's actions influence its later inputs. At a certain moment, the system is in a state s_t and the agent performs an action a_t, thus transitioning to a new state s_{t+1} and receiving a reward r_{t+1} according to how good that new state is. The agent's action affects what situation it will face next and therefore conditions not only the current reward, but also the next (and, subsequently, all the ones that follow).

Example 3.1: The balancing pole: In this problem, a pole is standing on a kart (see figure 3.1). The agent can apply a force to move the kart left or right any desired amount and, after each move, it receives 1 point if the pole is still standing.

If the agent only cares about immediate reward, it will consider any action acceptable as long as it keeps the pole is standing for one more step. This short-sighted strategy may lead it to push the cart into awkward positions where the pole is excessively tilted or the cart is close to the track's edge. Although such moves earn an instant reward, they make failure inevitable in the next few steps. By contrast, taking future rewards into account encourages the agent to take actions that not only keep the pole upright now, but also preserve stability over time — for example, making smaller corrective moves.

[3] In a **closed-loop system**, the system's actions affect its environment, and then the environment's response (feedback) affects the system's future decisions.

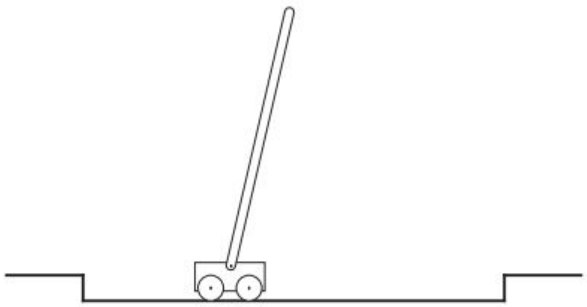


Figure 3.1: The pole balancing task. Illustration from Sutton & Barto.

In RL, the learner is not told which actions to take, as in many forms of machine learning, but instead must discover which actions yield the most reward by trying them out. It does have this reward system which guides it towards improvement, but it does not have access to real data to compare its results with. This characteristic places RL in a different branch from supervised learning — the technique used in the predictive neural network —, in which performance is evaluated against ground-truth data rather than through interaction with an environment.

3.3 The trade-off between exploration and exploitation

To obtain a lot of reward, an RL agent must prefer action that it has tried in the past and found to be effective in producing reward. Nonetheless, to discover such actions, it has to try actions that it has not selected before. In other words, the model has to *exploit* what it has already found in order to make forward progress; however, *greedily* following its knowledge may cause it not to *explore* other possibilities or actions that would provide better results.

Exploring always causes more failure (less reward), in average, since for exploration to happen suboptimal actions need to be tried out. For that reason, an agent that is programmed to achieve the highest possible reward may be tempted to stay in a known path and stop exploring. The dilemma is that neither exploration nor exploitation can be pursued exclusively without failing at the task. The agent must try a variety of actions and progressively favor those that appear to be best.

The exploration vs exploitation dilemma has been studied by mathematicians for decades. There is no universally optimal proportion between exploration and exploitation; the right balance is highly problem-dependent. What is clear is that both must be present to some degree, and that a balance between the two must be found.

3.4 Multi-arm bandits

Multi-arm bandits are the simplest form of reinforcement learning problems: a gambler is faced repeatedly with a choice among n different options, or actions. After each choice, he receives a numerical reward chosen from a stationary probability distribution that depends on the action he selected. His objective is to maximize the expected total reward over some time period, for example, over 1000 action selections, or time steps.

Figure 3.2: The multi-arm bandit problem.

Each of the n options has a value, which is unknown but can estimated. Here raises the exploit vs explore conflict. At every time step, there will be an option which has the highest estimated value. If the gambler chooses that one, he is taking the "best" option with the information he has (he's *exploiting* that information). However, he is being greedy, since he is not gathering any more information about the other options. If, instead, he chooses one of the apparently suboptimal actions, it is said that he is exploring, because he is trading off immediate reward for an improvement in his estimate of the value of that action.

3.4.1 Action-value methods

The true (actual) value of action a is denoted $q(a)$, and the estimated value on the t-th time step as $Q_t(a)$. Recall that value is the mean reward obtained when choosing an action. For now, let's take the most simple of estimation methods: averaging of past results.

$$Q_t(a) = \frac{R_1 + R_2 + \ldots + R_{N_t(a)}}{N_t(a)} \tag{3.1}$$

with $N_t(a)$ being the number of times that action a has been taken before the t-th time step. With this, the greedy action selection method would be:

$$A_t = \arg\max_a Q_t(a) \tag{3.2}$$

A near-greedy or ϵ-greedy action selection method would use this formula most of the time, but once in a while, say with a probability ϵ, choose another of the options randomly. This is a simple way to ensure a balance between exploration and exploitation.

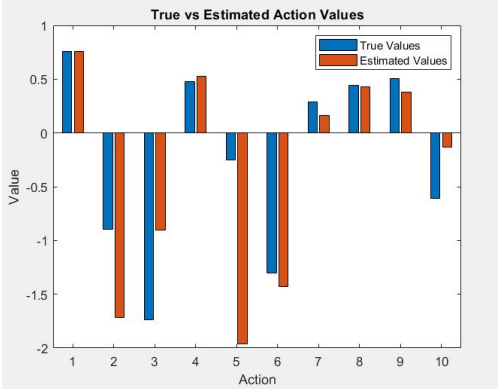


Figure 3.3: Real value vs agent-estimated value in a 10-armed bandit problem, after 1000 time steps, using an ϵ-greedy policy with $\epsilon = 0.05$. Notice how the more positive a value is, the more precise the estimation. That is because the agent picks the better options more often, and therefore the variance of the estimation of their value is continuously reduced.

To roughly assess the relative effectiveness of the greedy and ϵ-greedy methods, they were compared numerically on a set of 2000 randomly generated n-armed bandit tasks with $n = 10$. For each bandit, the action values, $q(a)$, $a = 1, \ldots, 10$, were selected according to a normal (Gaussian) distribution with mean 0 and variance 1. On the t-th time step with a given bandit, the actual reward R_t was the real $q(A_t)$ for the bandit (A_t was the action selected) plus a normally distributed noise term that was mean 0 and variance 1. Averaging over bandits, one can plot the performance and behavior of various methods as they improve with experience over 2000 steps.

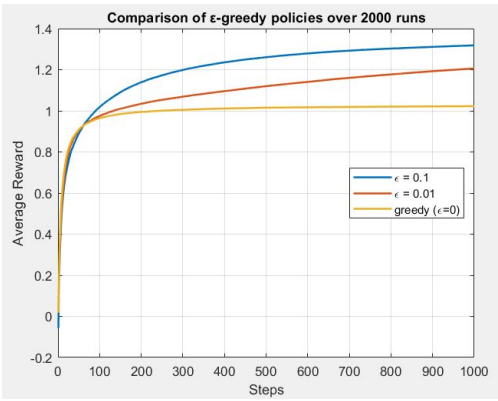


Figure 3.4: Evolution of average reward over 1000 time steps for ϵ-greedy policies with different values of ϵ. $\epsilon = 0.01$ ends up catching up and surpassing $\epsilon = 0.1$ once a large enough number of time steps have occurred. These results are the average over 2000 runs.

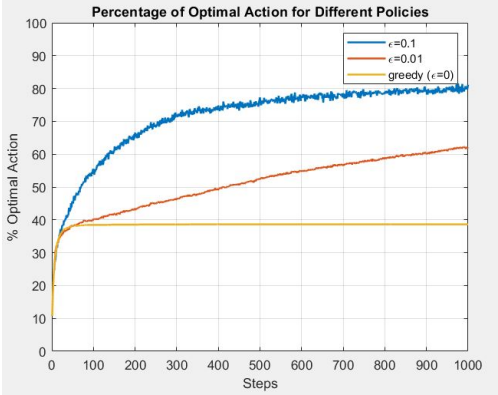


Figure 3.5: Evolution, over 1000 time steps, of the percentage of runs where the optimal action has been selected, for ϵ-greedy policies with different values of ϵ.

3.4.2 Optimistic initial values

Another simple technique to encourage exploration is to use *optimistic initial values*. Instead of initializing all action-value estimates $Q(a)$ to zero (or some neutral guess), we deliberately set them to large, optimistic values. At the beginning, every action appears promising, which forces the agent to try each of them. Once one has been tried, its value estimate gets smaller, since it was initially set to an overly optimistic number. In this way, exploration emerges naturally from the learning process without explicitly randomizing action selection.

This approach is only one among many methods to address the exploration–exploitation dilemma in the n-armed bandit problem. We highlight it here because the same idea will be useful in more complex reinforcement learning tasks later on.

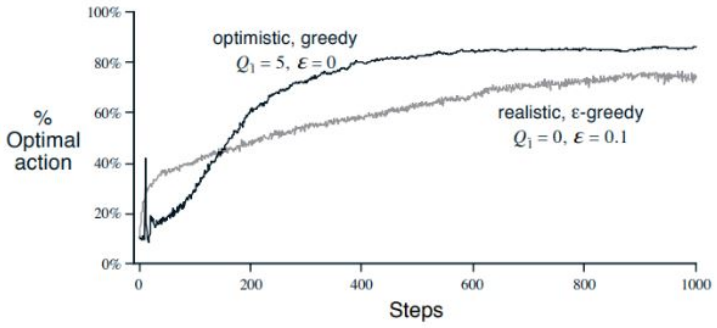


Figure 3.6: The effect of using optimistic initial values in the percentage optimal action choice. From Sutton & Barto

3.5 Finite Markov decision processes

3.5.1 Discounting

By now, it should alread be clear to the reader that the agent's objective is to maximize the total reward it receives. However, if the task is not episodic (that is, if it does not end in a well-defined terminal state), then the total summation of rewards could very easily go to infinity. To adress this issue, we introduce discounting. The agent then seeks to maximize the sum of discounted rewards, which places more emphasis on immediate — and therefore more controllable — outcomes. Discounting allows us to build a new definition of return:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{3.3}$$

3.5.2 The Markov property

What we would like, ideally, is a state signal that summarizes past sensations compactly, yet in such a way that all relevant information is retained. This normally requires more than the immediate sensations, but never more than the complete history of all past sensations. A state signal that succeeds in retaining all relevant information is said to be Markov, or to have the Markov property.

Example 3.2: A game of checkers A position in checkers serves as a Markov state because it fully summarizes all the relevant information about what happened earlier in the game. Although it does not explicitly contain the information of the previous moves (several move sequences can end on the same position), it encodes everything necessary to make decisions about the future of the game. In other words, seeing a position is enough for a player to make a decision on his next move, without the need of knowing what his opponent's last move was.

Example 3.3: Is the ground wet? Imagine trying to predict whether the ground is wet just by looking at the current sky. If it's cloudy, one might guess it's wet. If it's sunny, one might guess it's not. But whether it's actually wet depends on whether it rained earlier, information that is not contained in the current sky alone. The present state (sky condition) isn't enough — past information is needed, so the Markov property doesn't hold.

The Markov property is important in reinforcement learning because decisions and values are assumed to be a function only of the current state. In order for these to be effective, the state representation must be fully informative. However, in real-life problems, one can usually not account for all influential factor, therefore making a lot of problems not strictly Markov. In most practical cases, though, states can be considered *approximately* Markov by neglecting factors that, while influential, are not essential.

A reinforcement learning task that satisfies the Markov property is called a Markov Decision Process (MDP). The book focuses on this type of problems almost exclusively.

3.5.3 Value functions

Almost all reinforcement learning algorithms involve estimating value functions. Informally, the value of a state s under a policy π, denoted $v_\pi(s)$, is the expected return when starting in s and following π thereafter. This function is called the step-value function for policy π.

$$v_\pi(s) = \mathbb{E}\big[G_t \mid S_t = s\big] = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}\right] \tag{3.4}$$

Similarly, we define the value of taking action a in state s under a policy π, denoted $q_\pi(s, a)$, as the expected return starting from s, taking the action a, and thereafter following policy π. We call this the action-value function for policy π.

$$q_\pi(s, a) = \mathbb{E}\big[G_t \;\big|\; S_t = s,\ A_t = a\big] = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \;\middle|\; S_t = s,\ A_t = a\right] \tag{3.5}$$

If an agent follows policy π and maintains an average, for each state encountered, of the actual returns that have followed that state, then the average will converge to the state's value, $v_\pi(s)$, as the number of times that state is encountered approaches infinity.

3.5.4 The Bellman equation

Value functions used throughout reinforcement learning and dynamic programming satisfy particular recursive relationships. For any policy π and any state s, the following consistency condition holds between the value of s and the value of its possible successor states:

$$v_\pi(s) = \sum_a \pi(a \mid s) \sum_{s', r} p(s', r \mid s, a)\Big[r + \gamma v_\pi(s')\Big] \tag{3.6}$$

This is called the **Bellman equation** for v_π. The Bellman equation expresses the value of a state as the reward you expect now plus the discounted value of the next states you might reach. Think of it as breaking the total expected reward into 'what I get now' + 'what I expect to get in the future.

The Bellman equation is a recursive system of equations that, in theory, can be solved to find the value v_π of every state. It accounts for the most general case possible, that in which:

- Policy is stochastic: in a given state, two or more actions have a nonzero probability to be chosen. Each action has an assigned probability $\pi(a \,|s)$. In the Bellman equation, a sweep over all possible actions is done.
- The new state and the transition reward are stochastic: if the environment is in state s and the agent takes action a, there is a probability $p(s', r \mid s, a)$ that the new state is s' and the reward received is r. In the Bellman equation, a sweep over all possible new states and transition rewards is done.

In the Bellman equation (and in general in RL), the total discounted return G_t is represented recursively as $r + \gamma v_\pi(s')$. This term gives the expected return: the immediate reward r received after taking action a in state s, plus the discounted value of the next state s'. By definition, the value of a state s is the expected return when starting in s and following π thereafter. Thus, the Bellman equation replaces the infinite summation $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$ with its recursive equivalent.

An extensive analysis of the Bellman equation is performed in the book. What the reader needs to know for now is that solving the Bellman equation analitically as a system of equations is close to impossible in most cases. Therefore, in practice, RL algorithms rely on iterative, approximate methods to estimate value functions, rather than attempting an exact analytical solution.

3.6 Dynamic Programming

Dynamic programming is the first of those iterative methods. The term dynamic programming (DP) refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment is a Markov decision process (MDP). Classical DP algorithms are of limited utility in reinforcement learning both because of their assumption of a perfect model and because of their great computational expense, but they are still important theoretically.

3.6.1 Policy Evaluation

Imagine you want to know the value of each state (i.e. how good it is to be in each state) if you follow a certain policy. In theory, you could solve a giant system of equations all at once using the Bellman equation, but for most problems, that's too complicated. Instead, you can start with a guess for the value of each state ——for example, assume all states are worth 0. Then, you improve your guess step by step:

1. Take one state (s).
2. Sweep through all the actions that the policy might actually take from that state (that is, the ones with nonzero probability)
3. Use the Bellman equation as an update rule:

$$v_{k+1}(s) = \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a) \Big[r + \gamma v_k(s') \Big] \tag{3.7}$$

4. Now, your old estimate of the value of state s, $v_k(s)$, has been replaced by a new estimate $v_{k+1}(s)$.
5. Repeat for all states.

Each pass over all states is called a **full backup** because you update every state by sweeping over all the possible next states and rewards, not by using just one sample (see sections 3.7 and 3.8). Over time, these guesses get closer and closer to the true values v_π, until they converge in the limit and the Bellman equation is satisfied. This iterative process is called **Policy Evaluation**. Its purpose is to find the value of each state of a problem under a given policy.

Policy Evaluation, as DP in general, use a what is known as **bootstrapping**: estimating the value of a state on the basis of the estimated values of other states. Instead of waiting until the final return is observed, the current estimate is refined by combining the immediate reward with the estimated value of the next state ($r + \gamma v_l(s')$). Bootstrapping allows learning to happen in a more immediate manner, without requiring complete trajectories or full sums of rewards.

3.6.2 Policy Improvement

Suppose we have already determined the value function v_π for a given policy π. This function tells us how good it is to follow that policy from each state s, that is $v_\pi(s)$. But how can we not only *evaluate* the values under our current policy but *improve* to a better policy, and ultimately find the optimal one?

One way to investigate this is to consider taking a specific action a in state s and then following the existing policy π from that point onward. The expected return of this strategy is given by the action-value function

$$q_\pi(s, a) = \mathbb{E}\big[G_t \;\big|\; S_t = s,\ A_t = a\big] \tag{3.8}$$

The key question is whether $q_\pi(s, a)$ is greater than $v_\pi(s)$. If it is, it suggests that choosing action a whenever the agent encounters state s would improve the policy. Selecting the best action in this way for each state produces a new policy that is guaranteed to be at least as good as the original one. This result is formalized in what is known as the Policy Improvement Theorem.

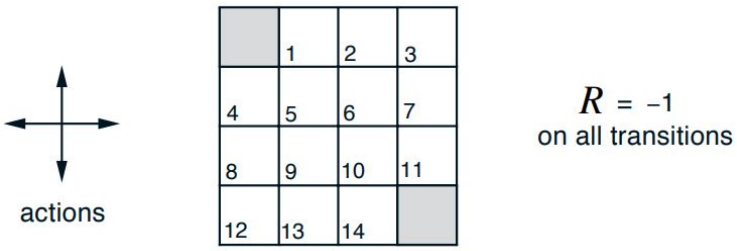
3.6.3 Policy Iteration

Combining these two concepts, an optimal-policy finding algorithm can be built. This algorithm is known as **Policy Iteration**. The procedure starts with an arbitrary policy π_0 and alternates between two main steps:

1. **Policy Evaluation:** Compute the value function, v_k, corresponding to the current policy π_k. That gives an estimate of how good it is to follow the current policy from each state.
2. **Policy Imporvement:** Construct a new policy π_{k+1} by choosing, in each state, the action that maximizes the expected return according to your previously done evaluation, v_k. That is, act *greedily* with respect to the current value function.

These steps are periodically repeated until the policy no longer changes between iterations. At convergence, the algorithm is guaranteed to produce an optimal policy π^* and its corresponding optimal value function v_π, as long as the problem is a Markov Decision Process (MDP).

Example 3.4: 4x4 Gridworld Consider the 4x4 gridworld shown below.



The nonterminal states are $S = \{1, 2, ..., 14\}$. There are four actions possible in each state, A $= \{$up, down, right, left$\}$, which deterministically cause the corresponding state transitions. Those actions that would take the agent off the grid leave the state unchanged. The reward is -1 on all transitions until the terminal state is reached.

Suppose the agent starts following, as π_0, the equiprobable random policy (all actions equally likely). The left side of figure 3.7 shows the value of each state after evaluation convergence has been achieved, while the right side shows the new, improved policy that is constructed by acting greedily with respect to those values. The policy shown in the lower right figure is the optimal policy for the problem. That means that, if the agent follows the steps recommended by the arrows, he will reach one of the terminal states the fastest way possible.

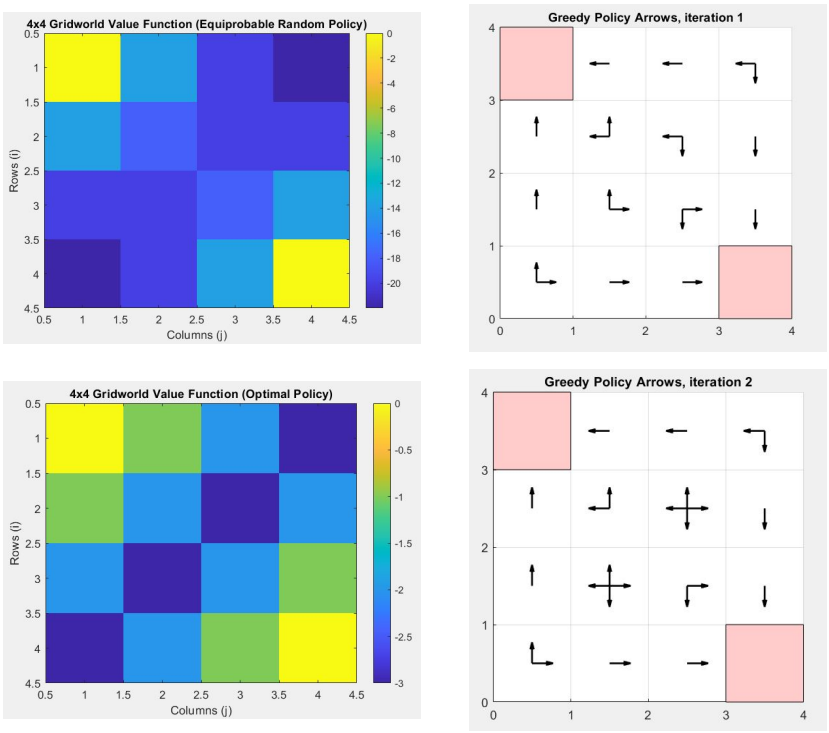


Figure 3.7: Convergence of the Policy Iteration algorithm in just 2 iterations for the 4x4 gridworld problem. Each row of the figure is an iteration of the algotythm. Left side of the figure shows the results of Policy Evaluation, while right side shows the policy found by value-greedy Policy Improvement.

DP algorithms, mainly Policy Iteration and its variations, are great at solving small tabular problems like a 4x4 Gridworld. However, the full-backup approach used in these algorithms quickly becomes impractical as the problem size grows. For each state update, a full backup requires summing over all possible next states and rewards, which scales poorly when the number of states or actions increases. In larger environments, the number of computations per iteration explodes, and storing all state values in a table may become impossible.

This limitation illustrates why DP is not suitable for large or continuous problems, and motivates the need for approximate methods and sample-based algorithms such as Monte Carlo or Temporal-Difference methods, which update values using individual experiences rather than sweeping over all possible outcomes.

3.7 Temporal-Difference learning (TD learning)

TD learning is a combination of Monte Carlo ideas and Dynamic Programming (DP) ideas. Like Monte Carlo methods[4], TD methods can learn directly from raw experience without a model of the

[4]**Monte Carlo** methods are a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results. In the RL context, they estimate value functions by averaging returns obtained from many sampled

environment's dynamics. Like DP, TD methods update estimates based in part on other learned estimates, without waiting for a final outcome (they bootstrap).

TD learning differce from DP in the fact that it does not require a model of the environment. Notice that DP does not interact with the environment; it uses a perfect model of it to compute exact expectations and updates. As opposed to that, a TD methods learns purely from experience by sampling. In other words, a TD agent effectively "plays the game" itself, improving its estimates through the rewards and transitions it encounters, while a DP agent remains outside the game, relying on its knowledge of the rules to calculate outcomes.

3.7.1 The TD(0) algorithm

The simplest TD *evaluation* algorithm, known as TD(0), is characterized by the following update rule:

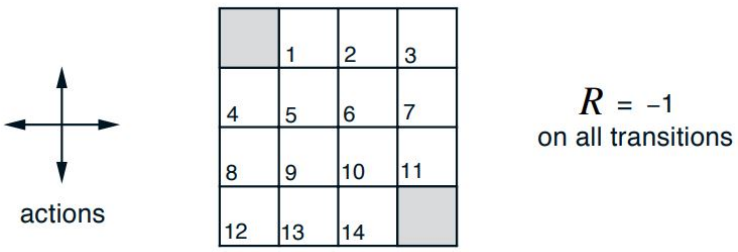
$$V(S_t) \leftarrow V(S_t) + \alpha\Big[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\Big] \tag{3.9}$$

In any TD learning update, the value of the currently visited state S_t is adjusted toward a *target*. In TD(0), the target is the sum of the immediate reward R_{t+1} plus the discounted value of the next state, $\gamma V(S_{t+1})$. The difference between the target and the current estimate,

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \tag{3.10}$$

is called the **TD error**. It measures how surprising or different from the expected the outcome was, and serves as the director of learning. Compared to Monte Carlo methods, TD(0) updates its estimates after every step, without having to wait until the end of an episode. This allows it to learn in continuing tasks and to improve its estimates much more quickly.

Example 3.5: TD(0) 4x4 Gridworld To illustrate how TD(0) works, consider once again a simple 4×4 gridworld (see section 3.6.3 for the full problem statement).



Suppose the agent is in state S_t (a particular square), takes an action, and moves to a neighboring square S_{t+1}. It then reveives the reward $R_{t+1} = 1$. With TD(0), the estimate of $V(S_t)$ is updated

episodes, without relying on bootstrapping from other estimates nor requiring a model of the environment.

immediately using:

$$V(S_t) \leftarrow V(S_t) + \alpha\Big[-1 + \gamma V(S_{t+1}) - V(S_t)\Big] \tag{3.11}$$

For example, if at the t-th time step $V(S_t) = -15$, $V(S_{t+1}) = -13$, $\alpha = 0.1$ and $\gamma = 1$, the update would be:

$$V(S_t) \leftarrow -15 + 0.1\Big[-1 + (-13) - (-15)\Big] = -15 + 0.1\Big[1\Big] = -14.9$$

Here the value of S_t shifts slightly up, towards the target of -14. By repeatedly applying this update as the agent explores the grid, the values of all states gradually converge toward the true expected returns under the current policy.

3.7.2 SARSA: On-policy TD control

In optimization tasks, it is not enough to know how valuable a state is; what ultimately matters is which action to take in each state. To capture this, we extend the concept of state values $V(s)$ to action values, $Q(s, a)$ (see section 3.5.3). By learning $Q(s, a)$ directly, an agent can choose actions greedily with respect to these estimates, allowing it to make optimal decisions without having to separately evaluate the consequences of all possible actions in each state, like a DP algorithm would.

We turn now to the use of TD prediction methods for the control problem, and again we face the need to trade off exploration and exploitation. The approaches fall into two main classes: on-policy and off-policy. On-policy methods attempt to evaluate or improve the policy that is used to make decisions, whereas off-policy methods evaluate or improve a policy different from that used to generate the data. In this section an on-policy TD control method is presented.

In the previous section we considered transitions from state to state and learned the values of states. Now we consider transitions from state–action pair to state-action pair, and learn the value of state–action pairs. In line with that, the SARSA update is

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\Big] \tag{3.12}$$

which is reminiscent of equation 3.9, the TD(0) update. A SARSA agent estimates action values through this update rule, and follows an ϵ-greedy policy with respect to those estimates to decide its next actions. Figure 3.8 shows pseudocode of a SARSA algorithm.

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Figure 3.8: Pseudocode of a SARSA algorithm, from Sutton & Barto.

SARSA updates only the action value of the action it has taken, no matter if it was or randomly picked by the ϵ-greedy policy. This has the implication of SARSA converging to the optimal policy for ϵ-greedy, which may not be the optimal one for a perfectly greedy agent. SARSA implicitly assumes that the agent will take random steps every while, and builds an optimal policy for that behavior. See section **??** for a bigger insight on this.

Example 3.6: Windy Gridworld Figure 3.20 shows a standard gridworld, with start and goal states, but with one difference: there is a crosswind upward through the middle of the grid. The actions are the standard four—up, down, right, and left—but in the middle region the resultant next states are shifted upward by a "wind," the strength of which varies from column to column. The strength of the wind is given below each column, in number of cells shifted.

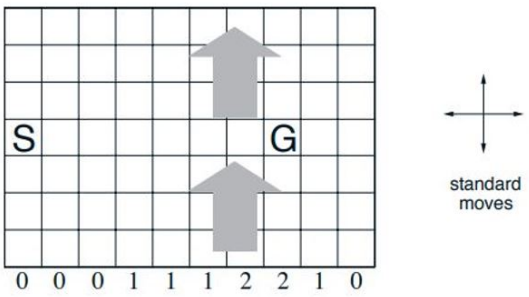


Figure 3.9: Windy Gridworld, from Sutton & Barto.

Figure 3.10 shows the result of applying ϵ-greedy Sarsa to this task, with $\epsilon = 0.01$, $\alpha = 0.1$, and the initial values $Q(s,a) = 0$ for all s, a.

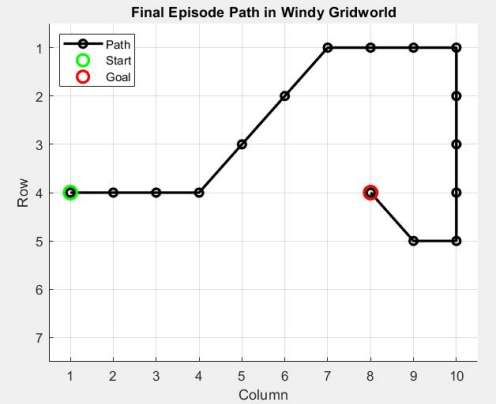


Figure 3.10: Optimal path of the Gridworld problem in example 3.6, found in 750 episodes using SARSA.

Figure 3.24 (left) shows the max action value (Q) at each grid location (recall that there is one action value per state and action, so each grid location holds 4 of them), while figure 3.24 (right) shows the evolution of the path length over episodes.

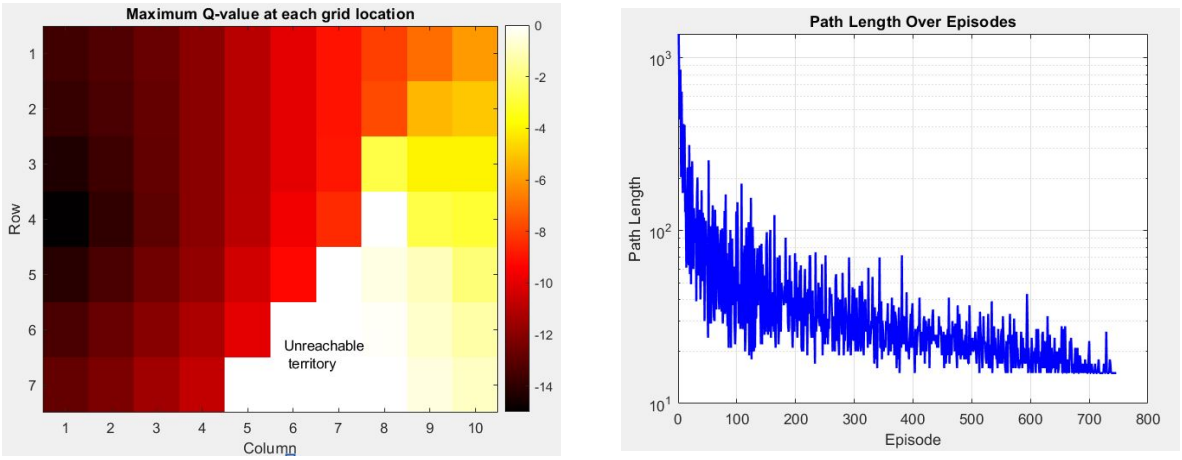


Figure 3.11: Additional plots for the Windy Gridworld problem. The left-hand plot shows the maximum state–action value for each state of the grid. The right-hand plot illustrates the evolution of path length over episodes, displayed on a logarithmic scale. Notice the fact that progress is very noisy. Notice as well the cutting line at the bottom right part of the plot, as if the noise tried to go deeper but was not able to. That is a sign of the optimal path being found.

3.7.3 Q-learning: Off-policy TD control

We have just studied SARSA, an on-policy TD learning control method that updates action values based on the actions actually taken by the agent. As opposed to that, there is Q-learning, an **off-policy** method. In Q-learning, the agent learns the value of the optimal action in each state, independently of the actions chosen by the behavior policy. This allows Q-learning to converge to the optimal action-value function $Q^*(s, a)$ even despite the agent occasionally exploring or behaving suboptimally during learning.

The Q-learning update is

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big] \tag{3.13}$$

where the main and only difference with SARSA relies on the *max* operator over possible next actions. This ensures that updates always move toward a target corresponding to the optimal action from the current state, independently of the action actually taken. Q-learning acts following an ϵ-greedy policy, just like SARSA, but it updates as if it were following a 100 % greedy one. This differentiation is what makes Q-learning an off-policy method. Figure 3.12 shows pseudocode of a Q-learning algorithm.

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$;
  until S is terminal

Figure 3.12: Pseudocode of a Q-learning algorithm, from Sutton & Barto.

Example 3.7: Cliff walking This example pretends to illustrate the differences between on-policy (SARSA) and off-policy (Q-learning) methods, and show how they can lead to different solutions in certain cases. Recall that off-policy methods assume that the future agent will be perfect, i.e. it will always follow the optimal action. Meanwhile, on-policy methods assume that the future agent will act as it does during learning: every now and then, it will explore.

Consider the gridworld shown in figure 3.13. This is a standard undiscounted, episodic task, with start and goal states, and the usual actions causing movement up, down, right, and left. Reward is -1 on all transitions except those into the darker region, which we will call "The Cliff." Stepping into this region incurs a reward of -100 and sends the agent instantly back to the start.

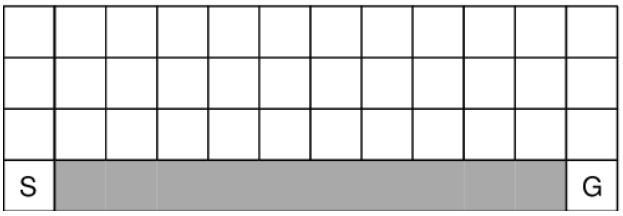


Figure 3.13: The cliff-walking gridworld, from Sutton & Barto.

Figure 3.14 shows the results of applying both SARSA and Q-learning, with $\epsilon = 0.1$ and $\alpha = 0.5$, to the cliff-walking problem. Notice how the path chosen by each of the methods is different. Q-learning finds the true fastest path from the starting block to the goal. SARSA, however, optimizes under the assumption that the agent is imperfect and might take suboptimal actions, and therefore chooses a path that walks as far from the cliff as possible to avoid any risk of falling.

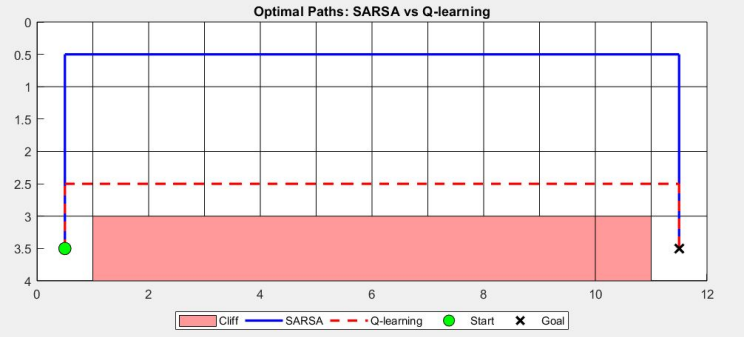


Figure 3.14: SARSA's solution and Q-learning's solution to the cliff-walking problem.

No solution is inherently better than the other; the choice depends on the context and the goals of the optimization. Q-learning assumes the agent behaves perfectly and therefore may take the fastest but riskier path, whereas SARSA accounts for the possibility of mistakes, resulting in a safer, slightly longer route.

Extrapolate this to a real-life situation: suppose you want to drive from point A to point B. A standard path-finding algorithm may give you the fastest route, but would you take it if it included narrow, winding roads with a high risk of accidents? Depending on your priorities — speed versus safety — you might prefer a slightly longer, more reliable path, just as SARSA does in the cliff-walking example.

3.8 Eligibility traces

3.8.1 n-step TD methods

So far, for RL problems with no model of the environment, we have seen TD methods and are aware of Monte Carlo (MC) methods (not explained in detail here for simplicity). Backups in simple TD methods — like the ones we have seen — rely only on the immediate reward R_{t+1}, using the value of the next state-action pair $Q(S_{t+1}, A_{t+1})$ to approximate the remaining rewards. In other words, they bootstrap. Monte Carlo methods, in contrast, perform backups based on the entire sequence of rewards until termination, also called full-return backups, and do not bootstrap. An intermediate approach is to perform backups based on the next n rewards: more than one, but not the full episode. Methods in which the temporal difference extends over n steps are called n-step TD methods.

At the end of the day, both MC methods and TD methods use the same target: total reward from the current state to the end of the episode. MC methods do it explicitly, by waiting until the end of each episode and taking the reward

$$G_t = R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{T-t-1} R_T \tag{3.14}$$

as the target, where T denotes the terminal state. Meanwhile, TD methods do it implicitly, through a

bootstrapped one-step target

$$R_{t+1} + \gamma V_t(S_{t+1}) \tag{3.15}$$

The point now is that this idea makes just as much sense after two steps as it does after one. The target for a two-step backup may be

$$G_t^{t+2}(V_t(S_{t+2})) = R_{t+1} + \gamma R_{t+2} + \gamma^2 V_t(S_{t+2}) \tag{3.16}$$

where now $\gamma^2 V_t(S_{t+2})$ corrects for the absence of the terms $\gamma^2 r_{t+3} + \gamma^3 R_{t+4} + \ldots + \gamma^{T-t-1} R_T$. Similarly, the target for an arbitrary n-step backup may be

$$G_t^{t+n}(V_t(S_{t+n})) = R_{t+1} + \gamma R_{t+2} + \ldots \gamma^{n-1} R_{t+n} + \gamma^n V_t(S_{t+n}) \tag{3.17}$$

Nevertheless, pure n-step TD methods are rarely used because they are inconvenient to implement. Computing n-step returns requires waiting n steps to observe the resultant rewards and states. For large n, this can become problematic, particularly in control applications. However, the concept of n-step methods is used to define other, more complex TD methods which are more conveniently implemented.

3.8.2 The forward view of TD(λ) methods

Backups can be done not only through an n-step return, but through any average of n-step returns. For instance, a backup can be done toward a target that is half of a two-step return and half of a four-step return:

$$\frac{1}{2} G_t^{t+2}(V_t(S_{t+2})) + \frac{1}{2} G_t^{t+4}(V_t(S_{t+4})) \tag{3.18}$$

The TD(λ) algorithm can be understood as one particular way of averaging n-step backups. This average contains all the possible n-step backups (from $n = 1$ to $n = T - t - 1$), each weighted proportional to λ^{n-1} and normalized by a factor of $1 - \lambda$. Here, $\lambda \in [0, 1]$ is called the **trace-decay parameter**, and is a measure of how much credit earlier states get in the update of value estimates.

The *target* for the TD(λ) algorithm update is

$$L_t = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^{t+n}\big(V_t(S_{t+n})\big) \tag{3.19}$$

which can be rewritten as

$$L_t = (1 - \lambda) \sum_{n=1}^{T-t-1} \lambda^{n-1} G_t^{t+n}\big(V_t(S_{t+n})\big) + \lambda^{T-t-1} G_t \tag{3.20}$$

This separation gives a clearer view of what happens in the limits $\lambda = 0$ and $\lambda = 1$. For $\lambda = 0$, the only surviving term is the first term of the summation, and thus we get a one-step TD method or

TD(0). For $\lambda = 1$, the summation term disapears and only the term G_t — the target of MC methods — survives.

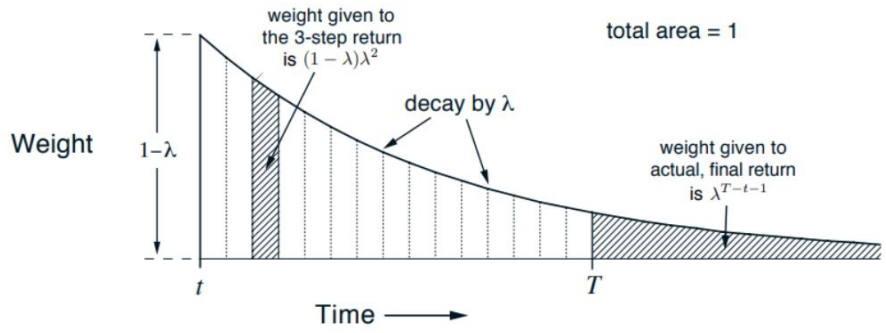


Figure 3.15: Weighting given in TD(λ) to each of the n-step returns.

The formulation we have just presented is called the **forward view** of TD(λ). It is conceptually useful because it clearly shows how the algorithm averages over all possible n-step returns to compute the update target for each state. Each n-step return is weighted by λ^{n-1}, which reflects the idea of assigning more credit to more recent states and gradually discounting earlier ones.

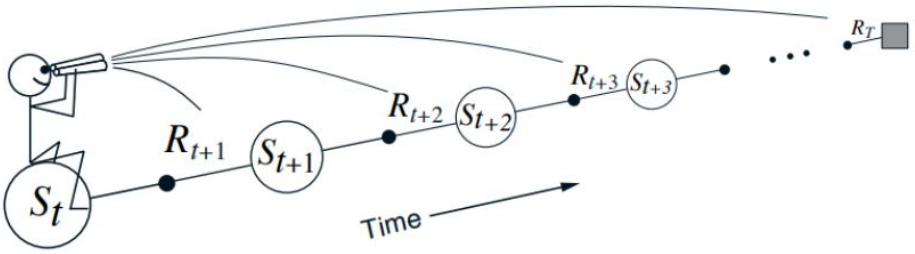


Figure 3.16: Illustration of the forward view of TD(λ).

From a practical standpoint, though, the forward view is not directly implementable. This is because computing the target L_t requires knowledge of rewards and states that will be encountered in the future, up to the end of the episode. In other words, it is *non-causal*: the algorithm would need to see events that have not yet happened in order to update the current state. For this reason, TD(λ) is usually implemented in its **backward view**, using *eligibility traces*, a tool that allows updates to propagate backward from observed rewards in real time, without needing to wait for the full future sequence.

3.8.3 The backward view of TD(λ) methods

The backward view of TD(λ) is the popular view from a practical standpoint because it is simple both conceptually and computationally. In the backward view, a new memory variable, associated to each state, is introduced: the eligibility trace. Eligibility traces tell us how blamed (or credited) a state should be for the recent rewards.

The eligibility trace for state s at time t is a random variable denoted $E_t(s) \in \mathbb{R}^+$. On each time step, the eligibility traces of all non-visited states decay by $\gamma\lambda$:

$$E_t(s) = \gamma\lambda E_{t-1}(s), \qquad \forall s \in S, s \neq S_t, \tag{3.21}$$

where γ is the discount rate and λ is the trace-decay parameter. The eligibility trace for S_t decays just like for any state, but is then incremented by 1:

$$E_t(S_t) = \gamma\lambda E_{t-1}(S_t) + 1 \tag{3.22}$$

This kind of eligibility trace is called an accumulating trace because it accumulates each time the state is visited, then fades away gradually when the state is not visited, as illustrated below.

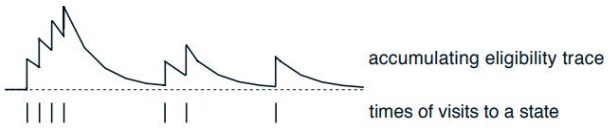


Figure 3.17: Evolution of an accumulating trace.

In the TD(λ) algorithm, when a state is visited and a reward is obtained, **all previously visited states have their values updated to some extent**. That general update is done by computing the current-state TD error

$$\delta_t = R_{t+1} + \gamma V_t(S_{t+1}) - V(S_t) \tag{3.23}$$

and triggering updates in all states proportionally to their eligibility traces:

$$Q(s, A_t) \leftarrow Q(S_t, A_t) + \alpha\delta_t E_t(s) \qquad \forall s \in S \tag{3.24}$$

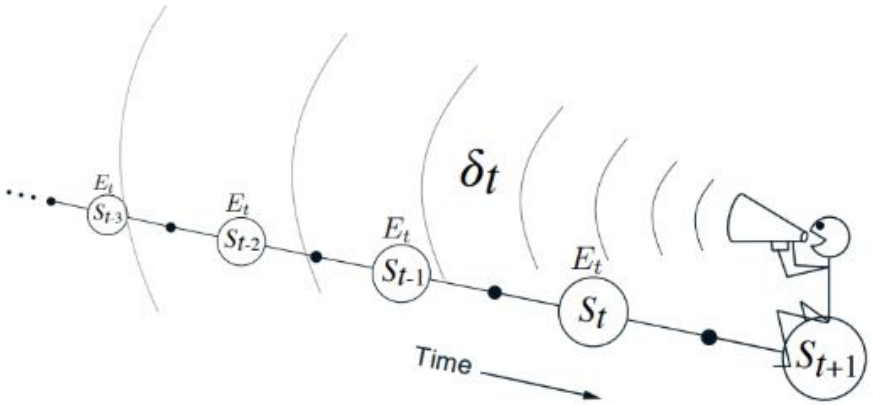


Figure 3.18: Illustration of the backward view of TD(λ).

3.8.4 SARSA(λ)

SARSA(λ) and TD(λ) are straightforward generalizations of the SARSA and Q-learning algorithms to the ideas of TD(λ).

For SARSA(λ), the update rule includes, as usual, the TD error

$$\delta_t = R_{t+1} + \gamma Q_t(S_{t+1}, A_{t+1}) - Q_t(S_t, A_t) \tag{3.25}$$

now weighted by the eligibility trace of the corresponding state-action pair:

$$Q(s,a) = Q_t(s,a) + \alpha \delta_t E_t(s,a) \qquad \forall s, a \tag{3.26}$$

Figure 3.19, an example from Sutton & Barto, perfecly illustrates the difference between regular SARSA updates and SARSA(λ) updates. The first board shows the path taken by the agent, from the start up to the terminal state marked as $*$. The next two boards show which which action values were updated as a result of this path, both by one-step Sarsa and by Sarsa(λ), respectively,

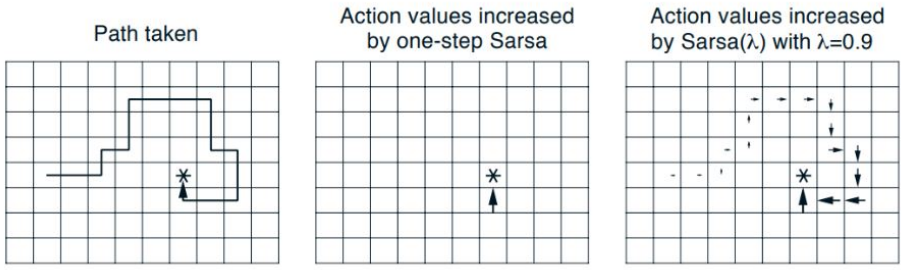


Figure 3.19: Gridworld example of the speedup of policy learning due to the use of eligibility traces. From Sutton & Barto.

Example 3.8: SARSA(λ) Windy Gridworld The SARSA(λ) algorithm was implemented to the Windy Gridworld problem (see section 3.7.2 for the full problem statement).

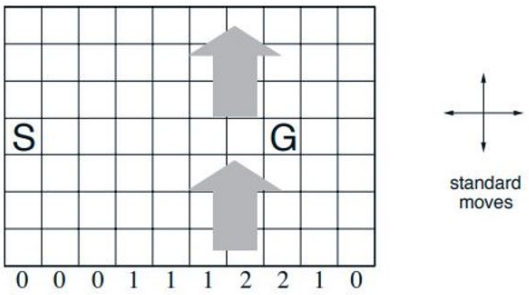


Figure 3.20: Windy Gridworld, from Sutton & Barto.

Various values of the trace-decay parameter (λ) were tested to compare their effect on learning. Figure **??** shows the state of learning at episode 200 for three values, $\lambda = 0, 0.5$, and 0.8. The corresponding

evolution of path length over episodes is shown in plot 3.22. Notice how the algorithm converges significantly faster to the optimal path in the $\lambda = 0.8$ case compared to the pure SARSA baseline. This improvement arises because higher λ values enable faster propagation of information through states, thanks to eligibility traces.

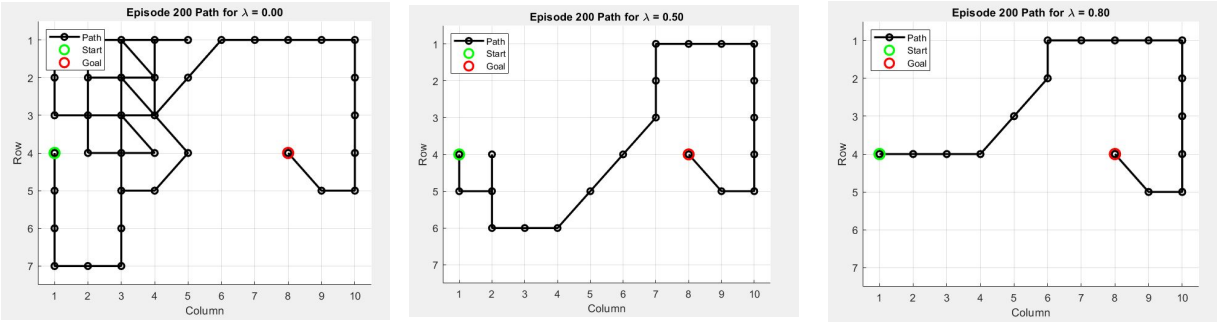


Figure 3.21: Full path at episode 200 for $\lambda = 0,\ 0.5$ and 0.8, respectively.

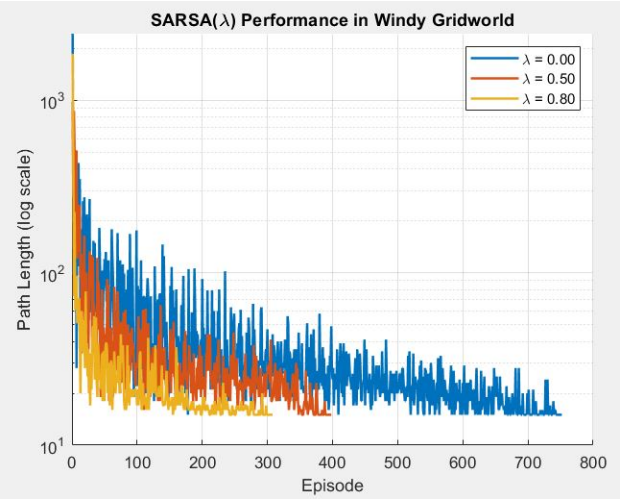


Figure 3.22: Path length evolution over episodes for $\lambda = 0,\ 0.5$ and 0.8

It's important to mention that learning speed generally improves as λ increases. Nevertheless, increasing λ to values that are very close to 1 can actually harm performance. The reason is that TD(λ) begins to behave almost like a Monte Carlo method: instead of relying mainly on short-term estimates, it bases its updates on nearly complete returns from entire episodes. While this allows information to propagate very far back, it also makes each update heavily dependent on long, noisy sequences of rewards. As a result, the updates become highly variable and less stable, often requiring more data to average out the noise.

In practice, this means that although large λ values speed up credit assignment, they can also make learning slower to converge or even unstable. For this reason, values of λ close to 1 are usually avoided, and intermediate values (e.g. between 0.6 and 0.9) are often preferred.

3.8.5 Watkins' Q(λ)

In off-policy methods, the look-ahead strategy is not that straightforward to implement. That is because, every time the ϵ-greedy policy takes an exploratory action, there is a **bifurcation between the learning path and the walking path**. Once that bifurcation happens, there is no longer a correlation between previous actions and current rewards, and therefore action values before the bifurcation should no longer be updated.

Thus, unlike TD(λ) or Sarsa(λ), Watkins's Q(λ) does not look ahead all the way to the end of the episode in its backup. It only **looks ahead as far as the next exploratory action**. Aside from this difference, however, Watkins's Q(λ) is much like TD(λ) and Sarsa(λ). Their lookahead stops at episode's end, whereas Q(λ)'s lookahead stops at the first exploratory action, or at episode's end if there are no exploratory actions before that.

It is worth noting that we speak of *look-ahead* rather than *look-back*, even though in practice we often implement Q(λ) using the backward view with eligibility traces. The reason is that, from a theoretical standpoint, the algorithm is defined in terms of how far ahead in the future the backup target extends. The backward view is simply an efficient computational trick, but conceptually the method is governed by its look-ahead horizon.

The mechanistic or backward view of Watkins's Q(λ) is also very simple. Eligibility traces are used just as in Sarsa(λ), except that they are set to zero whenever an exploratory (nongreedy) action is taken:

$$E_t(s,a) = \begin{cases} \gamma\lambda E_{t-1}(s,a) + I_{sS_t} \cdot I_{aA_t} & \text{if } Q_{t-1}(S_t, A_t) = \max_a Q(S_t, a); \\ I_{sS_t} \cdot I_{aA_t} & \text{otherwise.} \end{cases} \tag{3.27}$$

Here I_{xy} denotes a Kronecker-delta type function, i.e. a function that equals 1 if $x = y$ and 0 if $x \neq y$. Pseudocode for Watkins' Q(λ) is given in figure 3.23.

Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
Repeat (for each episode):
  $E(s,a) = 0$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
  Initialize S, A
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $A^* \leftarrow \arg\max_a Q(S', a)$ (if A' ties for the max, then $A^* \leftarrow A'$)
    $\delta \leftarrow R + \gamma Q(S', A^*) - Q(S, A)$
    $E(S, A) \leftarrow E(S, A) + 1$ (accumulating traces)
    or $E(S, A) \leftarrow (1 - \alpha)E(S, A) + 1$ (dutch traces)
    or $E(S, A) \leftarrow 1$ (replacing traces)
    For all $s \in \mathcal{S}, a \in \mathcal{A}(s)$:
      $Q(s,a) \leftarrow Q(s,a) + \alpha\delta E(s,a)$
      If $A' = A^*$, then $E(s,a) \leftarrow \gamma\lambda E(s,a)$
                else $E(s,a) \leftarrow 0$
    $S \leftarrow S'$; $A \leftarrow A'$
  until S is terminal

Figure 3.23: Pseudocode of Watkins' Q(λ) algorithm.

3.9 RL in continuous problems

Up to this point, we have focused on tabular methods, where the estimated value function is represented explicitly in a table with one entry per state or state–action pair. While straightforward, this approach is practical only for small problems: it requires both enough memory to store the table and enough experience to fill it accurately. As a result, the use of tabular methods is limited to discrete tasks with a finite and manageable number of states, like Gridworld.

This raises an important question: how can experience with a limited subset of the state space be generalized to produce useful approximations across a much larger or continuous space? In many tasks — especially continuous tasks — most states the agent encounters will never have been experienced exactly before. Learning in such environments requires the ability to extrapolate from past experience to new, unseen states.

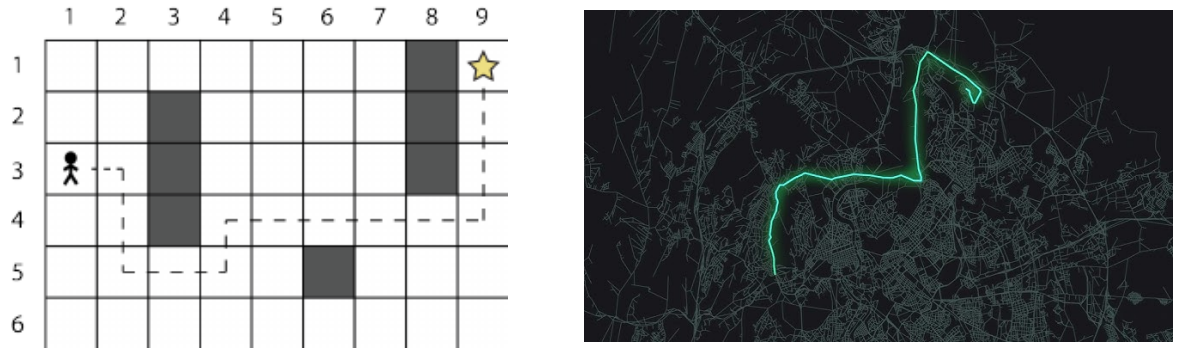


Figure 3.24: A gridworld, discrete path-finding problem (left). A real-life, continuous path-finding problem in Rome (right).

3.9.1 Value prediction through function approximation

Instead of storing state values $V(s)$ in a table, as we have been doing so far, we will now approximate them through a parametrized function

$$\hat{V}(s, w) \approx V(s) \tag{3.28}$$

where w is a vector of weights — for example, neural network weights. This is called **function approximation**. Typically, the number of components of w is much smaller than the number of states, and changing one component influences the estimated value of many other states.

3.9.2 Linear function approximation

A variety of function approximators can be used. The simplest case is **linear function approximation**, where the value of a state is expressed as a weighted combination of features:

$$\hat{V}(s, w) = \mathbf{w}^\top \mathbf{x}(s), \tag{3.29}$$

with $\mathbf{x}(s) = [x_1(s),\ x_2(s),\ ...,\ x_n(s)]$ a feature vector that describes the state. More powerful but complex approaches employ nonlinear approximators, such as neural networks. Learning then consists of adjusting the parameter vector w to minimize the discrepancy between predicted and actual returns. Typically, this is formulated as minimizing a squared-error objective:

$$J(w) = \big(G_t - \hat{V}(S_t, w)\big)^2, \tag{3.30}$$

where G_t is the return observed from state S_t. Parameter updates are performed by stochastic gradient descent:

$$w \leftarrow w + \alpha\big(G_t - \hat{V}(S_t, w)\big)\nabla_w \hat{V}(S_t, w), \tag{3.31}$$

with α the learning rate. This framework extends naturally to state–action value functions $Q(s, a)$, enabling function approximation to be integrated with control algorithms such as SARSA, Q-learning, or actor–critic methods.

3.9.3 Tile coding

In continuous state spaces, it is desirable for value estimations to vary smoothly across states: nearby states should have similar values. Consider, for instance, a 2D navigation task in which the agent moves in $\mathbb{R}^2$ from $A = (0, 1)$ to $B = (5, 3)$. If two states correspond to positions that are very close in space, then their values should also be close; otherwise the function approximation would give a discontinuous or inconsistent estimate of the true value function.

Tile coding is a function approximation technique designed for continuous or large state spaces. The basic idea is to map a state s in a continuous space into a high-dimensional but sparse binary (1s and 0s) feature vector $\mathbf{x}(s)$. The method works by creating a set of *tilings*, each of which partitions the space into disjoint *tiles*. A state s belongs to exactly one tile in each tiling, and the feature vector $\mathbf{x}(s)$ is constructed by setting to 1 the components corresponding to those active tiles (and 0 elsewhere).

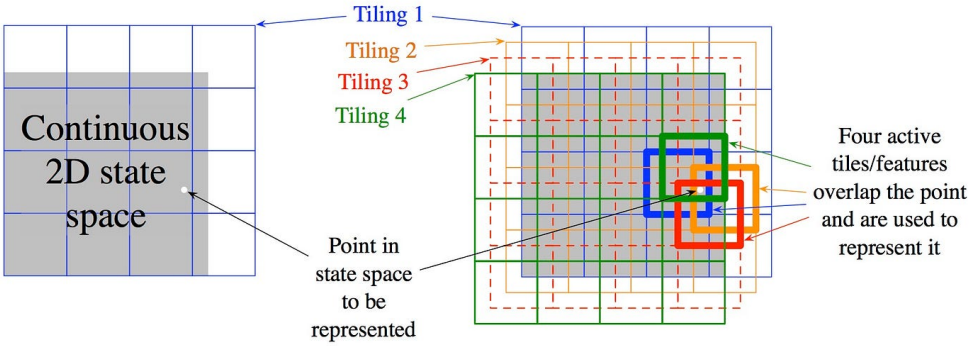


Figure 3.25: Visual representation of multiple, overlapping tilings.

Notice in figure 3.25 how the information obtained from one point in state space spreads in all directions to other nearby states, thanks to the overlapping tilings. Using a single tiling would lead to coarse and discontinuous approximations, since moving across a tile boundary could cause a sudden jump in the

value estimate. To overcome this, tile coding employs multiple overlapping tilings, each offset slightly from the others. When the state changes to a nearby one, typically only one of the n active tiles changes at a time, while the other $n-1$ remain the same. As a result, value estimates vary more smoothly across states, providing the desired generalization property.

The denser the tiling, the finer and more accurately the desired function can be approximated, but the greater the computational costs. Also, it is important to note that tilings can be arbitrary and need not be uniform grids. Not only can the tiles be strangely shaped, as in figure 3.26a, but they can be shaped and distributed to give particular kinds of generalization. For example, the stripe tiling in figure 3.26b will promote generalization along the vertical dimension and discrimination along the horizontal dimension, particularly on the left

Figure 3.26: Other kinds of tilings.

3.9.4 Control with function approximation

So far we have considered function approximation only in the context of *value prediction*. However, reinforcement learning is usually concerned with *control*: learning a policy that maximizes the expected return. In control settings, as we have seen repeatedly in this report, the agent must not estimate state values $V(s)$ but action values $Q(s,a)$, which guide the choice of actions.

Nonetheless, the same issue with generalization to continuous spaces arises, so, in practice, tabular action-value function must be replaced with a parameterized one:

$$\hat{Q}(s,a,w) \approx Q(s,a), \tag{3.32}$$

where w is the parameter vector. Control algorithms such as SARSA or Q-learning can then be adapted to this setting by updating w rather than individual state-action entries. The Gradient-Descent SARSA(λ) algorithm uses the TD error

$$\delta_t = R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t) \tag{3.33}$$

and an adapted form of eligibility traces

$$\mathbf{e}_t = \gamma\lambda \mathbf{e}_{t-1} + \nabla \hat{q}(S_t, A_t, \mathbf{w}_t) \tag{3.34}$$

to create an update rule for the weight vector

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \delta_t \mathbf{e}_t \tag{3.35}$$

Similarly, Watkins' Q(λ) algorithm can be extended to continuous problems. Pseudocode for both methods is given in figures 3.27 and 3.28.

Let $\mathbf{w}$ and $\mathbf{e}$ be vectors with one component for each possible feature
Let $\mathcal{F}_a$, for every possible action a, be a set of feature indices, initially empty
Initialize $\mathbf{w}$ as appropriate for the problem, e.g., $\mathbf{w} = \mathbf{0}$
Repeat (for each episode):
    $\mathbf{e} = \mathbf{0}$
    $S, A \leftarrow$ initial state and action of episode (e.g., ε-greedy)
    $\mathcal{F}_A \leftarrow$ set of features present in S, A
    Repeat (for each step of episode):
        For all $i \in \mathcal{F}_A$:
            $e_i \leftarrow e_i + 1$ (accumulating traces)
            or $e_i \leftarrow 1$ (replacing traces)
        Take action A, observe reward, R, and next state, S'
        $\delta \leftarrow R - \sum_{i \in \mathcal{F}_A} w_i$
        If S' is terminal, then $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\mathbf{e}$; go to next episode
        For all $a \in \mathcal{A}(S')$:
            $\mathcal{F}_a \leftarrow$ set of features present in S', a
            $Q_a \leftarrow \sum_{i \in \mathcal{F}_a} w_i$
        $A' \leftarrow$ new action in S' (e.g., ε-greedy)
        $\delta \leftarrow \delta + \gamma Q_{A'}$
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\mathbf{e}$
        $\mathbf{e} \leftarrow \gamma\lambda\mathbf{e}$
        $S \leftarrow S'$
        $A \leftarrow A'$

Figure 3.27: Pseudocode for a linear, gradient-descent Sarsa(λ) with binary features and ϵ-greedy policy. From Sutton & Barto

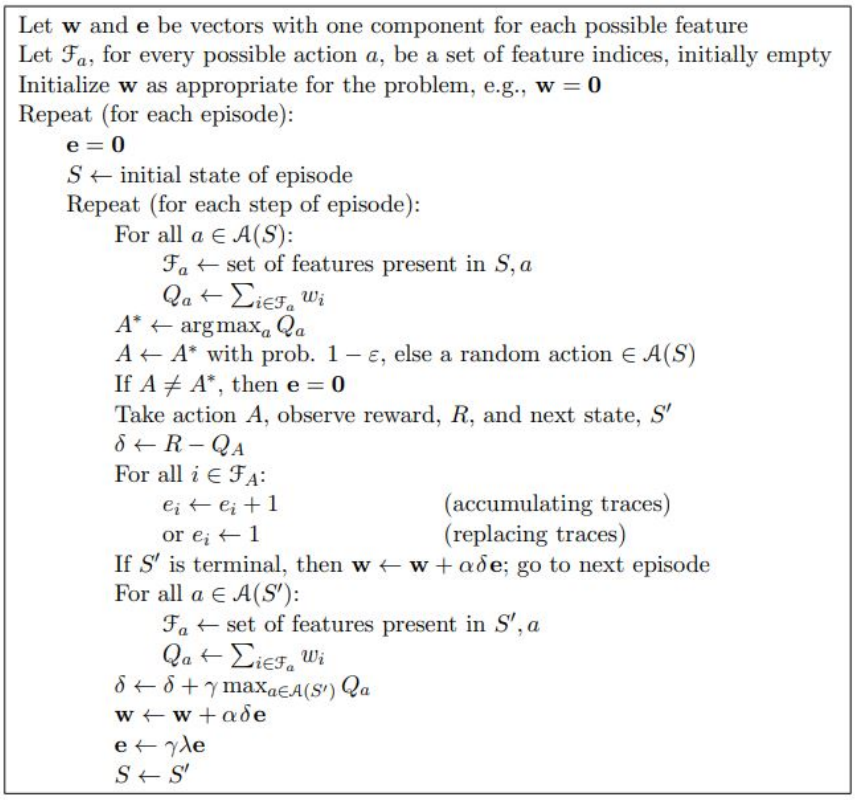


Figure 3.28: Pseudocode for a linear, gradient-descent version of Watkins's Q(λ) with binary features and ϵ-greedy policy. From Sutton & Barto

Notes on eligibility traces:

Notice that eligibility traces are working differently here, as there are no longer tabular values to which more or less importance can be assigned via eligibility traces. What we have now is an eligibility trace per weight, which gets larger or smaller according to the gradient of $\hat{q}$ with respect to the given weight.

The information carried by eligibility traces is now directional: they can take positive or negative values, depending on the sign of the gradient. With function approximation, eligibility traces are no longer just a scalar weighting factor indicating how much influence a state had on the current reward. Instead, they accumulate gradient information, making them a vector that reflects both the *degree* and the *direction* in which each parameter should be adjusted.

For example, if $\delta < 0$, the agent's estimate of $\hat{q}$ is too small; we need make it bigger:

- If the **gradient is negative**: $\hat{q}$ gets larger when w_t gets smaller. Therefore, decreasing w_{t+1} would make $\hat{q}$ increase. That is ensured through trace e_t being negative (see equations 3.34 and 3.35).
- If the **gradient is positive**: $\hat{q}$ gets larger when w_t gets larger. Therefore, increasing w_{t+1} would make $\hat{q}$ increase. That is ensured through trace e_t being positive.

Example 3.9: Mountain Car Consider the task of driving an underpowered car up a steep mountain road, as suggested by the diagram in figure 3.29. The difficulty is that gravity is stronger than the car's engine, and even at full throttle the car cannot accelerate up the steep slope. The only solution is to first move away from the goal and up the opposite slope on the left. Then, by applying full throttle the car can build up enough inertia to carry it up the steep slope even though it is slowing down the whole way. This is a simple example of a continuous control task where things have to get worse in a sense (farther from the goal) before they can get better. Many control methodologies have great difficulties with tasks of this kind unless explicitly aided by a human designer.

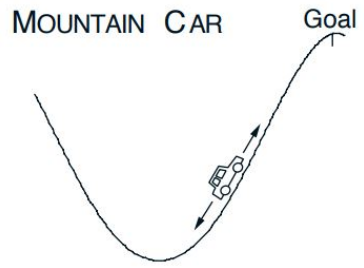


Figure 3.29: The mountain car task, from Sutton & Barto.

The reward in this problem is -1 on all time steps until the car moves past its goal position at the top of the mountain, which ends the episode. There are three possible actions: full throttle forward (+1), full throttle reverse (-1), and zero throttle (0). The car moves according to simplified physics. Its position, p_t, and velocity, $\dot{p}_t$, are updated by

$$p_{t+1} = \text{bound}(p_t + \dot{p}_t)$$

$$\dot{p}_{t+1} = \text{bound}(\dot{p}_t + 0.001A_t + 0.0025\cos(3p_t))$$

where $A_t \in \{-1, 0, 1\}$ are the three possible actions and $0.0025\cos(3p_t)$ is the term corresponding to gravity in the simplified phyisics of the problem. The *bound* operation enforces position and velocity to stay within the boundaries of the problem: $p \in [-1.2, 0.5]$ and $\dot{p} \in [-0.07, 0.07]$. Each episode starts from a random position and velocity uniformly chosen from these ranges. An episode ends when the goal state is reached. The bottom of the valley is at position $p = -0.5$.

The Mountain-Car task is a 2D (position and velocity) continuous reinforcement learning problem. As such, it must be studied in a two-dimensional state space. It can be solved with either of the two control algorithms that have been explained: SARSA(λ) or Watkins' Q(λ). Both algorithms yield similar results. In order to apply these algorithms, the continuous state space must be discretized through function approximation. For that, we employ **tile coding**, using 4 overlapping tilings of 16x16 tiles each. Each action $a \in \{-1, 0, 1\}$ is associated with its own set of features, resulting in the approximate action-value functions $Q(s, a, \mathbf{w})$.

The Sarsa algorithm in Figure 3.27 (using replace traces) readily solved this task. Figure 3.34 shows the evolution of the negative of the max action-value function (the cost-to-go function) throughout the state space, using the parameters $\lambda = 0.9$, $\epsilon = 0$ and $\alpha = 0.05$. The initial action values were all zero, which was optimistic (all true values are negative in this task), causing extensive exploration to occur despite the exploration parameter, ϵ, being 0.

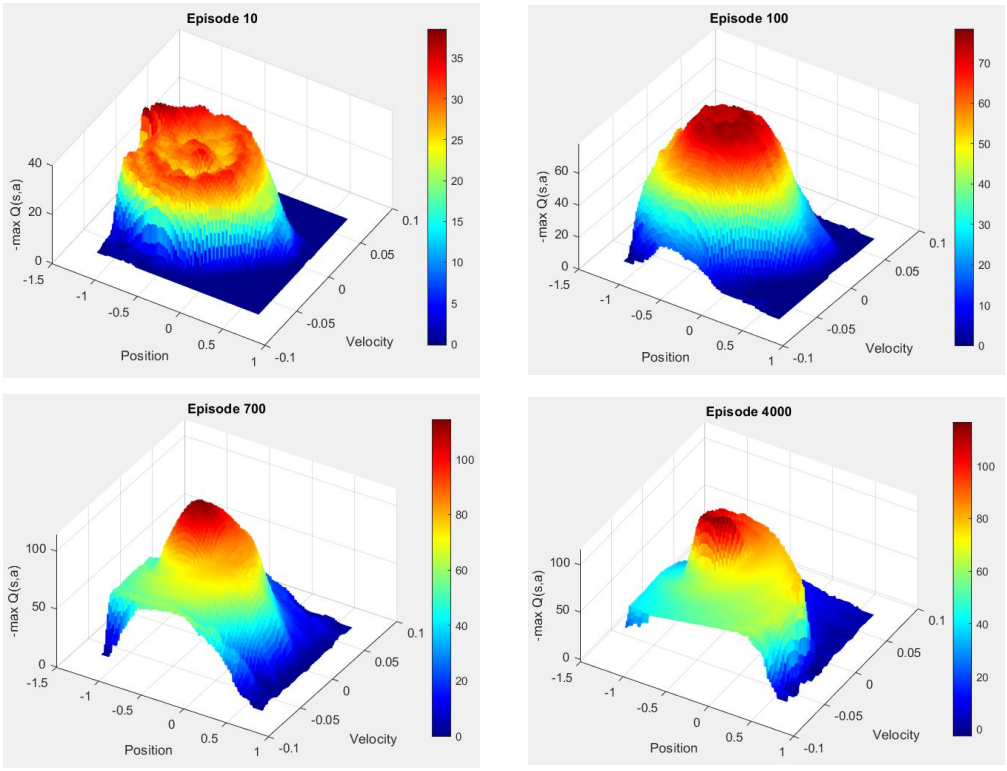


Figure 3.30: The cost-to-go function ($-\hat{q}(s, a, \mathbf{w})$) learned during one run. $p - 0.5$ represents the bottom of the valley.

At the beginning of learning (episode 10, top left), the estimated cost-to-go function is still close to zero in the majority of the state space, since many of those states have not yet been visited. During the first few episodes, the agent still has very limited information about the problem. Therefore, the car oscillates back and forth a lot in the valley, without finding a way to reach the top of the mountain. This causes the states in the valley to get updates very frequently, thus making their cost-to-go functions grow quickly.

The volcano-like shape observed in the top left plot arises directly from the dynamics of the car's early behavior. When the agent applies actions almost at random, the car gets stuck oscillating back and forth in the valley without ever reaching the goal. In the state space, such oscillations trace out a closed loop: as the car moves forward up the right slope, its velocity decreases to zero; then it rolls backward, with its velocity becoming increasingly negative, until it starts climbing the left slope and again slows down to a stop; finally, it accelerates forward once more, repeating the cycle. Because the agent repeatedly visits states along this loop, those states are updated far more frequently than states outside it, which causes their cost-to-go values to rise

faster. This explains why the surface at episode 10 has a raised circular ridge with a depressed center: the "ring" corresponds to the highly visited oscillatory path, while the middle and the edges remain relatively unexplored at this early stage.

By episode 100, the volcano-like ring begins to smooth out, and a clearer gradient structure emerges. By episode 700, this trend becomes even more pronounced. The surface clearly shows that the hardest states are those located deep in the valley with near-zero velocity, since from these positions the car requires the maximum number of steps to build momentum and reach the top.

At episode 4000, the surface has essentially converged to a stable shape. The structure now resembles a steep mountain in which the optimal policy can be visualized as if water were flowing downhill from the highest regions toward the lowest point. This "river" follows the direction of steepest descent in the cost-to-go surface, outlining the strategy that minimizes the number of steps to termination. Geometrically, the optimal path is a spiral in the two-dimensional state space (see figure 3.31): the car first moves forward up the slope, then rolls backward to gain speed, and finally accelerates forward again with sufficient momentum to crest the hill, following the policy shown in figure 3.32.

Figure 3.31: Trajectory of the car from the initial state $[-0.5, 0]$ (bottom of the valley with 0 speed) to the goal state (top of the mountain), represented in the state space. Notice the two abrupt changes in the trajectoty around $[-0.35, 0.01]$ and $[-0.9, -0.025]$. These sudden changes in direction are caused by the car shifting from forward acceleration to backward acceleration, and viceversa.

Figure 3.32: Policy followed by the agent after 4000 episodes. Notice the change of policy from right thrust to left thrust (green to red), and viceversa, around $[-0.35, 0.01]$ and $[-0.9, -0.025]$, directly related to the trajectory in the previous figure. The central area of the state spaced is nearly optimized, and policy is less consistent close to the sides, since those states are rarely visited. A larger number of episodes would perfect the sides too.

3.10 Actor-Critic

Up to this point, the focus has been on methods that estimate the values of states or state–action pairs. In such methods, like SARSA or Q-learning, control is done by learning action values and then deriving a policy from them, for example by applying an ϵ-greedy rule. These approaches are therefore known as *action-value methods*.

Actor–Critic methods follow a different structure. Although they may still involve the estimation of state or action values, these values are not used directly for action selection. Instead, the policy is represented explicitly, with its own set of parameters independent of any value function. The value estimates serve as a baseline or "critic" that evaluates the policy's actions, while the "actor" adjusts the policy parameters in the direction suggested by the critic.

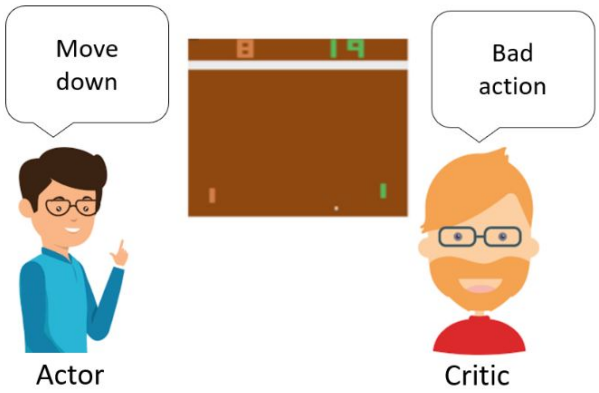


Figure 3.33: The Actor-Critic method.

Actor-Critic is a wide family of methods that use this structure for action selection. Here, only the most basic version of that family is explained. We will refer to it as Actor-Critic from now on, for simplicity, but the reader may keep in mind that it is only one of the many versions of the method.

Now, Actor-Critic, learning is still guided by action values (Q), but they are used more softly — as an evaluation tool rather than as a direct driver of decisions. Learning is on-policy: the critic must evaluate and provide feedback on the very policy that the actor is currently following. This evaluation is done via TD error:

$$\delta_t = R_{t+1} + \gamma V_t(S_{t+1}) - V_t(S_t) \tag{3.36}$$

3.10.1 The discrete case

Suppose actions are generated by the Gibbs softmax method:

$$\pi_t(a|s) = \Pr\Big\{A_t = a \;\Big|\; S_t = s\Big\} = \frac{e^{H_t(s,a)}}{\sum_b e^{H_t(s,b)}} \tag{3.37}$$

Where $\pi_t(a|s)$ is the probability of choosing action a from state s. $H_t(s,a)$ are usually called *preferences* and they are the trainable parameters that define the policy. They are not related to the expected return from a given state s, unlike action values. Instead, they measure of how often each action a shall be selected by the agent. Roughly speaking, in actor-critic methods, the actor chooses its actions via $H_t(s,a)$ and the critic evaluates them via $Q(s,a)$. The preferences are updated by

$$H_{t+1}(S_t, A_t) = H_t(A_t, S_t) + \beta\delta_t \tag{3.38}$$

3.10.2 The continuous case

An extension of this to continuous problems can be easily done, in a similar manner to the extension of SARSA and Q-learning to continuous problems showed in section 3.9. The main difference is that, for Actor-Critic, two sets of weights are required: one for the critic and one for the actor.

The **critc** estimates the state-value function $V(s)$ using linear approximation:

$$V(s,w) = \sum_i w_i x_i = \mathbf{w}^T\mathbf{x}(s) \tag{3.39}$$

Where $\mathbf{x}(s)$ is the feature vector for state s (recall tile coding, section 3.9.3), and $\mathbf{w}$ is the weight vector that the critic has to learn.

The **actor** follows a policy $\pi(a \mid s;\theta)$, which is parametrized by a second set of weights θ

$$\pi_t(a \mid s;\theta) = \frac{e^{H_t(s,a;\theta)}}{\sum_b e^{H_t(s,b,\theta)}} \tag{3.40}$$

Preferences $H(s,a;\theta)$ now depend on θ, linearly with respect to the feature vector

$$H(a,s;\theta) = \sum_i \theta_i x_i = \theta^T \mathbf{x}(s) \tag{3.41}$$

3.10.3 Eligibility traces and weight updates

The extension of Actor–Critic methods to incorporate eligibility traces is conceptually straightforward. The critic component corresponds to on-policy learning of the state-value function v_π, which can be implemented using the TD(λ) algorithm with one eligibility trace per state. The actor component, in turn, requires an eligibility trace for each state–action pair. Consequently, an Actor–Critic method equipped with eligibility traces requires two sets of traces: one for states (critic) and one for state–action pairs (actor). These traces are updated according to the following rules:

$$\mathbf{e}_w \leftarrow \gamma\lambda\mathbf{e}_w + \mathbf{x}(s_t) \tag{3.42}$$

$$\mathbf{e}_{\theta_a} \leftarrow \gamma\lambda\mathbf{e}_{\theta a} + \nabla_{\theta_a} \ln \pi(a_t|s_t) \tag{3.43}$$

Where e_w and e_{θ_a} are the eligibility traces for critic and actor, respectively. Meanwhile, the two sets of weights are updates as follows:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha_c \delta_t \mathbf{e}_w \tag{3.44}$$

$$\theta_a \leftarrow \theta_a + \alpha_a \delta_t \mathbf{e}_{\theta_a} \tag{3.45}$$

An Actor-Critic method that uses these rules, or any other variation of them, can be applied just like SARSA and Watkins' Q(λ) to solve a wide group of continuous problems. For example, Actor-Critic can be used to solve the Mountain Car task, obtaining similar results to the previously mentioned methods.

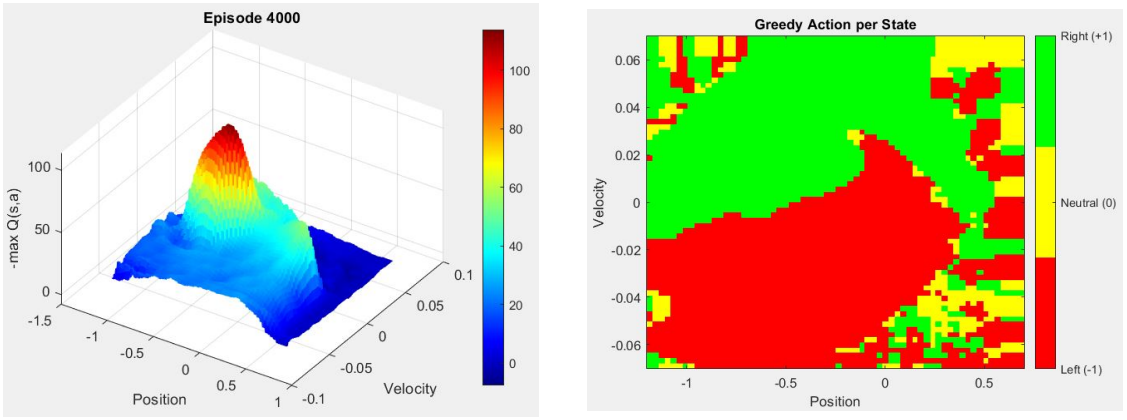


Figure 3.34: Results of the Actor-Critic method for the Mountain Car task.

4 The reinforcement learning code

As a result of the formation process, a functional reinforcement learning code was created towards the end of this internship. The code is compartmentalized, general, and applicable to a wide variety of continuous problems. Nonetheless, it is still young and can be complemented in several ways, with a refactoring phase foreseen to improve readability, maintainability, and extensibility.

The objective of this short section is not to explain the code in detail. The reader can inspect the code themselves and become familiar with it more effectively than through a written description. Rather, the purpose here is simply to inform the reader of its existence and encourage them, now that they have studied many theoretical reinforcement learning concepts and algorithms, to explore the code and identify the correspondences between theory and implementation.

That being said, the structure of the code can be somewhat harder to understand through a pure inspection. That is why an UML diagram is provided. This same diagram can be found in the *OpenArmsTutorial* directory in *Swan*.

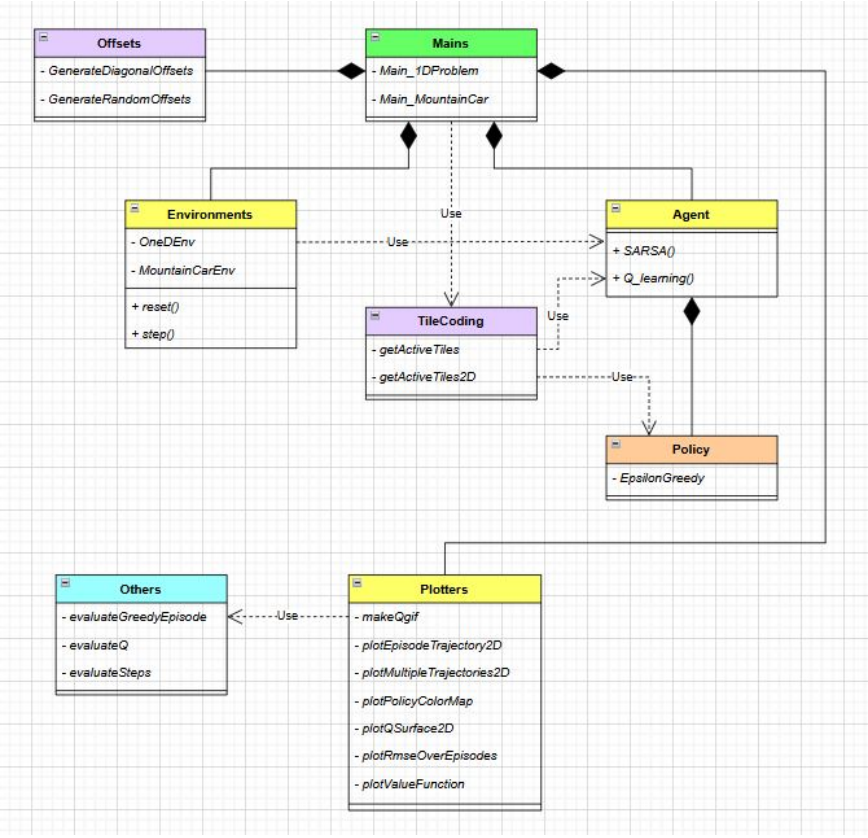


Figure 4.1: UML diagram of the reinforcement learning code.

5 Future lines of action

During this internship, three main topics of research and implementation were addressed: the predictive neural network, the migration of that code to Julia, and the exploration of reinforcement learning as a method for optimization. For future contributors to the project, it may be useful to review the current state of the three topics, and outline possible directions for continuing each of these lines of work. This final section aims to do just that.

5.1 Code migration to Julia

Although this part of the internship was the least directly related to the Open Arms project, it serves as a proof-of-concept for migrating a full Matlab codebase to Julia. The methods and workflow developed during this migration are general and modular, making them applicable to other projects that aim to transition from Matlab to Julia. In particular, the approach demonstrates how to maintain compatibility with existing Matlab functions, systematically replace classes and functions with Julia implementations, and validate results at each step. As such, this component of the internship can be seen as a template for future code migrations, providing a practical guide and reference for other teams considering a similar transition.

5.2 Predictive neural network

The predictive neural network is, as of now, one of the most advanced parts of the project. The major bugs were identified and resolved, and once several improvements were implemented, its predictions came rather close to the real consumption values. This was achieved even with a limited dataset; now that the Open Arms dataset is available, the neural network is expected to reach a much higher level of precision. Consequently, this component is relatively mature, and may not require as much immediate attention as other parts of the project. The only thing that might require improvement in the neural network is the choice of inputs, as discussed in section 1.3.

5.3 Neural network + reinforcement learning

Undoubtedly, the main future line of action regarding the predictive neural network relies on how to combine it with the fuel-optimization branch of the project. One possible direction is to integrate the neural network as a core component of a larger optimization framework, where its consumption predictions serve as inputs for decision-making algorithms, such as reinforcement learning agents (see section 3) or classical optimization techniques.

For instance, **states** of a reinforcement learning environment could be provided by the inputs of the neural network (wind, vessel speed, etc.); n-dimensional tile coding would need to be implemented there (see section 3.9.3). Then, **actions** could be related to the amount of thrust applied by the ship; thrust could be a continuous action space, but simplifying it to a discretized version — where there are only k possible thrust positions — might be the better option for computational efficiency or stability in training. Continuous-action RL methods (like DDPG or PPO) could be explored later. Finally, **rewards** could be given by a combination of predicted consumption and distance travelled by the ship. A model for computing the latter should be developed, but it would be strongly related to vessel speed, hence building that model should not be too complex. The reward clearly must grow with distance travelled, and decrease the more fuel is consumed. Therefore, any positive polynomial expression of

$$x = \frac{\text{distance}}{\text{consumption}} \tag{5.1}$$

could be used as reward-giver. The simplest case would be the linear one,

$$R_{t+1} = x, \tag{5.2}$$

but more complex expressions that implement non-linear relations between reward (R_{t+1}) and x may be studied. The specific form of the reward function could be further refined if a more effective relationship between distance and consumption is identified.

A major challenge in this approach is the **transition between states**, since, as of now, we do not have a model that returns a new state S_{t+1} after action A_t is taken from state S_t. Therefore, research should be done in that direction.

Going even further, once this were implemented, a validation and evaluation issue would arise. It would be essential to assess the performance of the reinforcement learning agent in terms of fuel efficiency, and its extensibility from the simulated environment to the real case. A natural approach would involve running simulations over a range of realistic scenarios, including variations in wind, sea state, vessel load, and route characteristics, and comparing the predicted fuel consumption with actual measurements. Key performance metrics could include total fuel consumed, distance traveled per unit of fuel, and the stability of the RL agent's policy over time.

5.4 Closure

Overall, this internship has been a highly rewarding experience, giving me the opportunity to explore, learn, and implement new concepts every day. I hope that the work presented here, from predictive modeling to code migration and initial reinforcement learning experiments, can serve as a foundation for future contributors to build upon and take further. It has been rewarding to

see the progress made, and I am excited by the potential these lines of work have for the future of the project.

Finally, I would like to thank the people who have guided me throughout this work, researchers Xavier Martínez and Àlex Ferrer. Their dedication to the project and constant willingness to provide guidance have been extremely valuable to me during my time conducting research alongside them.