14 Instruction Manual

The final part of this thesis is to generalize the RL-agent — which was previously specially tailored to play the Iterated Prisoner's Dilemma — such that it can be used to learn almost any task. This agent should provide a useful tool for those who are only little familiar with Reinforcement and Machine Learning techniques. This part of the thesis will provide no new scientific results. It will rather demonstrate the agents abilities and provide a guide on how to use and tune the agent. Furthermore, we will mostly refrain from using technical terms the user might be unfamiliar with. The agent is implemented in Python. The following code snippets and the implemented agent can be found in the following github repository:

`https://github.com/id428/RL-agent`

14.1 General Usage

The learning procedure is comprised of two repeating steps. The first step is to play the chosen game in order to gather experience. During this step, the agent applies previously gained knowledge to the game and explores how deviations from already known strategies turn out. The second step is to learn from these newly acquired experiences by updating the internal decision-making system.

14.1.1 Prerequisites

The agent is designed to adapt to any environment. It decides autonomously when to deviate from its learned strategies and how to expand its knowledge in an efficient way. Also, the entire adaptation process runs automatically. However, there are still certain pieces the user has to provide and think about:

- The user has to provide the whole **environment**, meaning the game or the simulation.
- One of two things the user has to think about is the **game state**. The game state is a numerical array which contains informations about the situation the agent is in. The user has to figure out which pieces of information are crucial or which might help the agent to make informed decisions[1]. More on that in section 14.3.2.
- The second thing the user has to think about are the **payoffs**. The payoffs are the rewards, points or whatever quantity is fit to quantify and assess the performance of the agent. The agent will seek to maximize the payoff. Consequently, the user has to figure out which outcomes should be rewarded or penalized in order to convey an idea of the goal. Section 14.3.3 elaborates on the subject of payoffs.

[1] A possible composition of the game state for the IPD is described in section 6.1

14.1.2 Example: Tic Tac Toe

The github repository contains a file `tic_tac_toe.py` which is an implementation of the game *Tic Tac Toe*. The Jupyter-Notebook `Learn_Tic_Tac_Toe` shows how to teach a RL-agent to play the game. The following sections will go through this example and explain each step.

Preceding imports

```
import numpy as np # for numerical operations
from tic_tac_toe import Tic_Tac_Toe # load game class
from tic_tac_toe import Random_Player, Preventive_Player
from RL_agent import RL_agent # Load RL-agent class
import pickle # for storing the agent for later use
import torch # for reproducability (RL-agent employs this package)
```

First, the necessary classes and libraries are imported. The file `tic_tac_toe.py` also contains two classes `Random_Player` and `Preventive_Player`. `Random_Player` plays by choosing randomly from all possible moves. `Preventive_Player` prioritizes preventing the winning option by choosing its action such that it prevents the third opposing symbol in a row. Against it, one has to create a dilemma (german *Zwickmühle*) from which the `Preventive_Player` cannot emerge without loosing the game. Both players can be used as a sparring partner for the RL-agent. One can already run a game between these two players to see how the `Tic_Tac_Toe` class works:

```
Game = Tic_Tac_Toe(print_board=False)
Game.run(Random_Player(),Preventive_Player())
winner = Game.winner
```

In the first line, an instance of a Tic Tac Toe game is created. Setting the `print_board` option to `True` causes the board to be printed out during the game so that one can retrace the single moves. The second line runs the game with the two players passed as arguments. The first player will get the first turn. After the game is finished, the winner is denoted by setting the variable `Game.winner` to 0, 1 or `None` if the first, second or no player won, respectively.

Creating RL-agent

```
seed = 42
np.random.seed(seed) # for reproduction
torch.manual_seed(seed) # for reproduction

# Create agent
# 9 input features, 100 hidden nodes, always one output
```

```
architecture = [9,100,1]
n_actions = 9 # Number of actions available
agent = RL_agent(architecture,
    n_actions,
    expl_rate=0.05,
    memory_size=1000*9, # at most 1000 matches
    batch_size=32)
```

Next, we create an instance of the learning agent. The first three lines are just for reproduction purposes meaning they ensure that everyone setting the `seed` to 42 gets exactly the same result as reported here. In line 5 we define the architecture of the network. The first entry of the list denotes the number of input features, i.e. the size of the game state. For Tic Tac Toe, the game state has nine entries, one for each field. The second entry denotes the number of nodes in the hidden layer, quantifying how complex and layered the decision-making algorithm will be. Higher numbers allow for more complex decisions but also need more training. The last entry is always 1, denoting that the outcome/payoff of the game is one-dimensional, i.e. 1 or 0 for win or loose (alternatively, the number of points achieved in other games). In line 7 the number of available actions is defined. For Tic Tac Toe, the number of available actions is 9, one for each field even though not every move is possible at any given time. The remaining lines are one single command which creates an instance of a RL-agent. The lines 10 to 12 specify the exploration rate, memory size and batch size. These learning parameters will be elaborated in section 14.3.1.

Training RL-agent

```
# run 10 training generations
for i in range(10):

    # PERFORMANCE EVALUATION
    agent.evaluation_mode() # Put agent into evaluation mode
    test_games(agent,Random_Player()) # test agents performance

    agent.training_mode() # Put agent into training mode

    # Create sparring partner
    sparring_partner = Random_Player()

    # RUN TRAINING GAMES
    for j in range(100):

        # RUN GAME
        Game = Tic_Tac_Toe(print_board=False)
        Game.run(agent,sparring_partner)
        winner = Game.winner

```

```
21          # ASSIGN PAYOFFS
22          if winner==0: # if RL-agent is winner
23              payoffs = np.zeros(Game.player_turns[0])
24              payoffs[-1] = 1 # the last turn led to victory
25              agent.assign_payoffs(payoffs)
26
27          if winner==1: # if opponent is winner
28              payoffs = np.zeros(Game.player_turns[0])
29              payoffs[-1] = -1 # the last turn led to defeat
30              agent.assign_payoffs(payoffs)
31
32          if winner==None: # draw
33              payoffs = np.zeros(Game.player_turns[0])
34              agent.assign_payoffs(payoffs)
35
36      # ADAPT AGENT
37      agent.train()
38
39  agent.evaluation_mode() # back to evaluation mode
```

Finally, the agent is trained. The program enters a loop which consists of three parts:

1. Playing test games to evaluate the agents performance and observe its progress. The agent does not draw any experience from these games.

2. Playing 100 training games during which the agent collects experience and explores new tactics.

3. Learning from the experiences made previously.

Line 5 puts the agent into evaluation mode. In evaluation mode, the agent does not record the games and always chooses the action which seems most beneficial to it, i.e. no exploration occurs. In line 6 a predefined function runs 100 test games of the agent against the Random Player and prints out how many the agent won and lost[2]. This way one can follow the progress the agent makes from generation to generation. Line 8 puts the agent into training mode. Now the agent records every move and explores new strategies. In line 11 an instance of the Random Player is defined as the agents sparring partner. We then enter the loop during which the agent will play 100 training games.

The lines 17 to 19 run a game exactly as shown previously, only with the agent and the sparring partner as players. In line 22 to 34 the payoffs are assigned. This means that the agent cannot know the consequences of a move right away but only after the game has ended. The agent has to be told how it did after each game. This is done by creating a list which assigns the immediate rewards (points) to every action. Since only the last turn produces immediate payoffs (victory or defeat), the list is all zeros except for the last entry which is 1 or -1 for victory or defeat, respectively. In case of a victory, line

[2]The predefined function can be found in the appendix in section A.5

23 creates a list of payoffs which contains only zeros, line 24 changes the last entry to 1 and line 25 passes the payoffs to the agent which then automatically assigns them to the moves made previously. The same is done in case of a defeat, only with -1 as the last entry and in case of a draw with all zeros.

One may think that this leads to the agent interpreting every move as worthless except for the last one. However, this is not the case. After passing the immediate payoffs to the agent, it internally backpropagates the payoffs to all previous moves so that it knows which moves finally led to victory or defeat. This makes it easier for the user since the user only has to submit the immediate payoffs at the respective point in time.

Finally, in line 37, the agent adapts its internal decision-making algorithm to its experiences. Calling this function `train()`, when the training games are already played might seem confusing to those not familiar with machine learning. This is an artefact from the machine learning vocabulary where training means to adapt an algorithm to the data.

Attention! Line 39 puts the agent back into evaluation mode. We advise the user to always put the agent back into evaluation mode if its not playing training matches. Notwithstanding the fact that the agent would still try to explore new strategies, the real problem is that it still records every move it does, waiting for the respective payoffs to be assigned. This will lead to an error during the training procedure since the number of recorded moves does not match with the number of assigned payoffs if those have not been assigned.

After this training procedure, the RL-agent has played 1000 training matches against the Random Player with the result that it wins over 90% of the test matches against the Random Player. This is a very good result since even a Random Player occasionally (or accidentally) thwarts the agents plans and causes a draw or sometimes even a defeat. Note that the agent is only trained in being the player which makes the first move and is therefore most suited as the beginning party in a Tic Tac Toe match.

14.2 Requirements for the Environment

In order to learn how to get along in a certain environment, the agent has to interact with the environment and vice versa. Apart from the initialization and the `agent.train()` function, there are two essential functions which have to be called by the environment:

Choosing actions

`agent.choose_action(game_state, possible_actions=[], return_payoffs=False)` is the function used when the agent has to make a move. The first argument is the game state, a numerical vector of the size specified when the agent was created, which is needed for the agent to make an informed decision and to record the game state. The function then returns the index of the action it would like to perform. That means the user has to specify beforehand which index belongs to which action. Imagine the agent plays a spatial game and there are four actions: moving up, down, left and right. One can then decide

that index 0 corresponds to up, 1 to down, 2 to left and 3 to right (indexing starts at 0). As long as this is done consistently, the agent will make sense of it.

The `choose_action()` function has two optional arguments. One can pass a list of indices representing the possible actions. This ensures that the agent makes no invalid moves. Another way to teach the agent which moves are invalid is by adding penalties for invalid moves to the payoffs. The second optional argument `return_payoffs` decides whether the method returns the payoff the agent expects. If possible moves are provided, it only returns those of the possible moves. This way one can analyze how much value the agent assigns to each move.

Assigning Payoffs

The second function is the already mentioned `agent.assign_payoffs(payoffs)` function. This is needed for the agent to relate the recorded game states to the payoffs, i.e. the consequences the agents actions had in a specific situation. The number of payoffs in the list has to exactly match the number of moves the agent made. It can therefore be advisable to create a function which monitors the agents moves.

When to assign payoffs?

If a move can influence the course of the game, the future rewards are just as relevant as the immediate one. The prospect of future rewards has to incorporated into the value (the immediate payoff plus the expected future payoffs). This is why payoffs should only be assigned once the whole chain of events is known, i.e. when the game has finished. Independent payoffs have to be assigned seperately, meaning that one should not play several training games and then assign all payoffs at once. The agent would interpret the games as interdependent and backpropagate the payoffs from later games to earlier games which is not useful.

When to train the agent?

Theoretically, one could train the agent each time payoffs have been assigned. However we advise to run many training games between two training procedures so that the agent has enough opportunities to test and explore its newly won knowledge. This reduces training time by only training once a considerable amount of possibly novel experiences has been made.

On the other hand, if the agent is in an environment where quickly adapting to new knowledge is crucial (on-the-fly updating, see appendix section A.2), one may train the agent very often but with very few epochs so that the time needed for a single training procedure is reduced.

14.3 Fine tuning

14.3.1 Learning parameters

The learning parameters determine how well the agent can train. More specifically, it determines how quickly the agent learns, how accurately it adapts and how complex and layered the decision-making algorithm is. Every single of these learning parameters can be set when initializing the agent just like that:

```
agent = RL_agent(architecture=[9,100,1],
    n_actions=9,
    gamma=0.9,
    expl_rate=0.1,
    epochs=5,
    batch_size=32,
    memory_size=9000)
```

Architecture

As already mentioned above, the architecture determines how complex the decision-making algorithm is. The first entry is always the length of the game state passed to the agent, the last entry is always 1. The second entry denotes the complexity of the algorithm. A good rule of thumb is to set the second entry to a value higher than the first entry plus the number of possible actions but not more than two orders of magnitude larger (exceptions can be made). For the previous parts of the thesis we successfully used agents with the architecture `[18,30,1]` where the first entry is larger but the second one smaller than in our Tic Tac Toe example (architecture `[9,100,1]`), assuming that the strategies do not need to be as complex in the IPD. As with most hyperparameters in machine learning, there is no one size fits all approach. Finding the ideal setting is often the result of trial-and-error.

Gamma

`gamma` is the so-called discount factor. It determines how many future payoffs are taken into account. If `gamma` is 0, the agent will always choose the action with the highest immediate reward not considering potentially higher future payoffs if it decided to go for a lower immediate reward. If `gamma` is 1, the agent takes all future rewards into account. For games like Tic Tac Toe where the only thing that matters is the final result, high (long term) values for `gamma` are appropriate. `gamma` can only assume values from 0 to 1.

Exploration rate

The exploration rate also only assumes values from 0 to 1. It denotes the probability with which the agent chooses not to make the most promising choice but a random move which might put the agent in a so far unknown situation. The exploration rate is usually set very low, i.e. below 0.1. This way, the agent mostly follows the presumably most

promising path and only occasionally deviates from it. In short games, a good rule of thumb is to choose the exploration rate such that one can expect exploration to occur at most once in a game, so that the agent also experiences perfect games. In longer games (e.g. chess), explorations do not spoil the outcome of the match as much as they do in short matches like Tic Tac Toe, where one exploration might prevent any possibility of winning the match. Longer games allow to make up for a mistake which is why multiple exploration can be allowed each match.

Epochs

This is simply how often the agent iterates through the dataset during training. Too few epochs might not be enough to fully adapt to the data. In terms of adaptation quality, more is always better. However, the adaptation time is directly proportional to the number of epochs. Generally, up to 100 epochs are reasonable for this implementation. In our Tic Tac Toe example, five were already enough.

Batch size

This is how many datapoints are used at once to further adapt to the experiences. The number of updating steps per epoch is given by the number of datapoints divided by the batch size. One epoch should include at least some hundred updating steps. One should therefore choose the batch size small enough to ensure that. However, smaller batches cause more updating steps which cause more adaptation time. For a dataset containing hundreds of thousands of datapoints, the batch size can very well be between 100 and 1000. One usually chooses powers of two (i.e. 128, 256, ...) because this way the allocated memory is most efficiently used.

Memory size

This denotes how many datapoints the agent can remember. If the memory size is exceeded, older datapoints are replaced by newer ones. Often, a changing environment causes the first experiences the agent made to become obsolete at some point. A limited memory size can help the agent to adapt to new conditions and be less conservative.

14.3.2 Game State

The game state is the information passed to the agent. It should give a good overview of the situation as well as the details needed to recognize immediate dangers or possibilities. For Tic Tac Toe it seems sufficient to just pass the state of the nine fields to the agent. In spatial games, it might be more relevant to pass the information of what is going on around the agent rather than reporting every information possible. When selecting the important features, the signal-to-noise ratio should be high. Rather limit the game state to the most crucial pieces of information than passing lots of irrelevant features. The more features the user selects, the more complex the decision-making algorithm needs to be which also leads to a longer training time.

The game state can be comprised of very detailed pieces of information, e.g. what is going on on a particular field, or it can contain derived quantities which summarize general trends, e.g. the fraction of cooperations in case of the IPD.

14.3.3 Payoffs

Payoffs are the means by which the user communicates the goal of the game to the agent. In the ideal case, payoffs directly represent the goal of the game. If the goal is to win, like in Tic Tac Toe, then the only payoff should be the one assigned at the end of the game.

Sometimes, rewards can be used to nudge the agent towards the real goal. Imagine letting an uninformed agent figure out to play chess by providing it a sparring partner. How long would it take until this agent finally experiences something different than a defeat? How long until the agent figures out anything that improves its gameplay? When anything the agent does leads to the same unsatisfying result, no move seems to be better than another. This problem is why payoffs are sometimes used to convey an idea of the goal of the game by rewarding something like intermediate goals (like capturing opponent's pieces, not getting one's own pieces captured).

The latter should be used with caution. An agent capturing enemy pieces is worth nothing if this does not help him winning. If the intermediate goals are rewarded about as much as the final goal, the agent might seek to achieve the intermediate goals instead. For this reason, intermediate goals should be rewarded much less than the real goals. They should only be used to get the agent going.

14.3.4 Sparring Partners

An agent needs an environment to interact with, but not all environments require an opponent, e.g. when training an agent to escape a maze. On the other hand, in games like Tic Tac Toe, another agent (the sparring partner) is part of the environment and how much the agent improves depends on the agents sparring partner. How else would the agent learn to play in the first place? A good way to get an agent to learn the game is to start with a relatively simple, equally uninformed sparring partner, i.e. a random agent, until the agent mostly succeeds and then train the agent with a more advanced sparring partner.

At some point, there will be no more sparring partners left. From that point on (maybe even from the beginning on, if you will), the agent could use a copy of itself as its sparring partner. But beware, there are some difficulties involved. As already mentioned, the agent trained to play Tic Tac Toe only knows how to play the game if it makes the first move. It is expected to perform worse as the second party. Consider this when using a copy as a sparring partner. Putting a trained agent in a different position makes it weaker and therefore less fit as a sparring partner.

14.4 Limitations

This RL-agent should prove to be quite versatile. However, the very general implementation imposes some constraints one the flexibility:

- The agent can only make discrete decisions. If the agent had to specify how much of a specific action it would do (like how much money to invest), one would have to offer the agent several options.
- The agent has no convolutional layers. Those were useful if neighbouring fields of a game board should be related to each other. A partial workaround is to provide features relative to the agent with the game state, like what is happening around it.
- The learning parameters are set at the beginning and fixed for the rest of the game. But sometimes one would like to have a very high exploration rate at the beginning which gradually decreases. A possible workaround is to directly access the agents member variables `agent.expl_rate`, `agent.learning_rate`, `agent.epochs`, `agent.memory_size` and `agent.gamma` and change their values.

14.5 Tips and Tricks

Ensemble agents

Each agent is randomly initialized (except when `torch.manual_seed()` is called before) and therefore has an individual starting point. It may happen that the initial values are so unfortunate that the agent has a hard time finding any patterns which is why it is advisable to train multiple agents and see which one performs best. Another possibility is to train multiple agents and use the expected payoffs returned by the `agent.choose_action()` function to 'democratically' decide what to do next.

As already said, it can be difficult to train a single agent in every aspect of the game. Assigning tasks to specially trained agents might be a better option to create a great ensemble agent. In the case of Tic Tac Toe, one could train one agent for matches where the agent starts and another agent for those where the opponent makes the first move.

Exploiting symmetries

The game board of Tic Tac Toe is rotationally invariant. That means that two different game states strategically correspond to the same game state and only differ by a rotation. Exploiting such symmetries can drastically decrease the number of possible game states while keeping the amount of experience constant. This can increase learning speed and success a lot.

- **Tit For 2 Tats:** Starts by cooperating and then defects only after two successive defects by opponent.
- **TFT:** See *Tit For Tat*
- **Win-Shift-Lose-Stay:** (Don't confuse with *Win-Stay-Loose-Shift* or *WSLS*) Starts with defection. Changes action if the opponent cooperated in the previous round.
- **Win-Stay-Lose-Shift:** Starts with cooperation. Changes action if the opponent defected in the previous round.
- **WSLS**: See *Win-Stay-Lose-Shift*.

A.5 Predefined test game function

This is the predefined test game function:

```
def test_games(agent, opponent,n_test_games=100):

    # run test games
    games_won = 0
    games_lost = 0

    # agent begins
    for n in range(n_test_games):
        # Create and run game
        Game = Tic_Tac_Toe(print_board=False)
        Game.run(agent,opponent)
        winner = Game.winner
        if winner==0:
            games_won += 1
        if winner==1:
            games_lost += 1

    print('Generation',i+1,'\tGames won:',games_won,'\tGames lost:',games_lost)
```