CURRENCY TRADING OPTIMIZATION

Software Agents Coursework 2019

City University of London

Translating Forex trading into a Markov Decision process and thus building a Reinforcement Learning algorithm using Q-learning.

UG - Trevor Oluotch, UG- Sai Uppinakuduru

1. Define the Domain & Task

Currencies around the world are traded on the global foreign exchange (FX) market. The FX market is the largest in the world with trillions of dollars being exchanged in trade volume each day (Drożdż, Górski and Kwapień, 2007).
Since a currency is simply a store of value, the way to account for a gain or a loss in value is through measuring one currency against another – pair trading. When applying pair trading the value of a base currency – such as the GBP is relative to the currency its being traded against – such as the USD. As a result, the aim of currency trading is to make a profit each day based upon a base currency relative value increasing.

The majority of currency trading volume comes from trading in 'major' currency pairs which are known as the world's largest and most developed economies (Galant and Dolan, 2007). These major pairs include currencies such as GBP (Great British Pound), USD (United States Dollar), EUR (Euro). Despite trade volumes going to major pairs the FX market also offers trading options for currencies such as AUD (Australian Dollar), CAD (Canadian Dollar), NZD (New Zealand Dollar) which are known as commodity pairs due to these economies holding natural resources and offering a relatively de-regulated market for currency exchanges (Perry, 2019). Furthermore, due to the relatively high growth in emerging market economies there is also an FX market provided to trade currency pairs such as MXN (Mexican peso) and SGD (Singapore Dollar) to name a few. The currencies from each of these economic tiers are paired in many-to-many relationships and act as a transmission function of market participants sentiment to developments in each economy and the effects they will have in the world. As a result, the global FX market is made from a diverse set of currency pairs representing varying economies, and the participants in the market use these pairs in an attempt to realise a profit from the current macro-economic data and future outlook.

As discussed, the FX market has a wide range of currency pairs available. In order to provide a detailed analysis of the agent through the Q-learning algorithm implemented in the reinforcement learning problem, the domain is represented as a graph with currencies as nodes. In the environment there are 13 currencies included from economies in each tier: major, commodities and emerging. The agent's task is to find the optimal term currency to trade against the base currency – GBP (see figure 1).

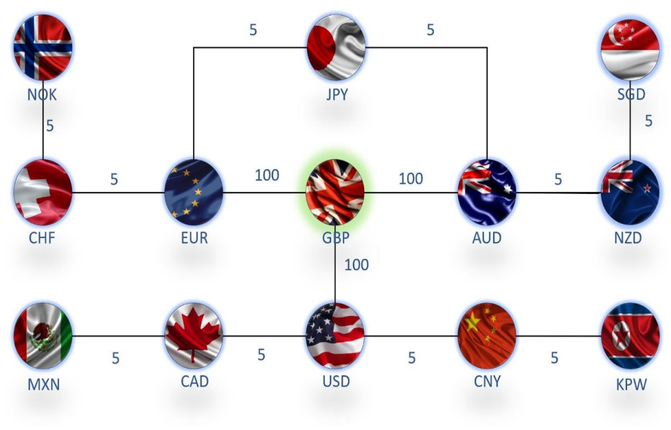


Figure 1. Agent's Environment

In an attempt to model the real-world closely the currencies that are paired with the GBP – EUR, AUD, USD are the currency pairs that represent the largest trade volumes on the FX market since these are the closest linked economies in trade to the UK. In addition to this, the environment also represents the interconnectivity of the FX markets so a currency such as the USD is closely linked to the CAD, MXN currencies (see figure 1).

2. Define a State transition function and the Reward function

2.1 State transition function

A reinforcement learning problem can be formalised as a finite Markov decision process. A Markov decision process must adhere to three criteria as described by Sutton and Barto (2018): the "learning agent should be able to understand the state of its environment", take actions that affect the state, and have rewards relating to the state of the environment (see figure 2).

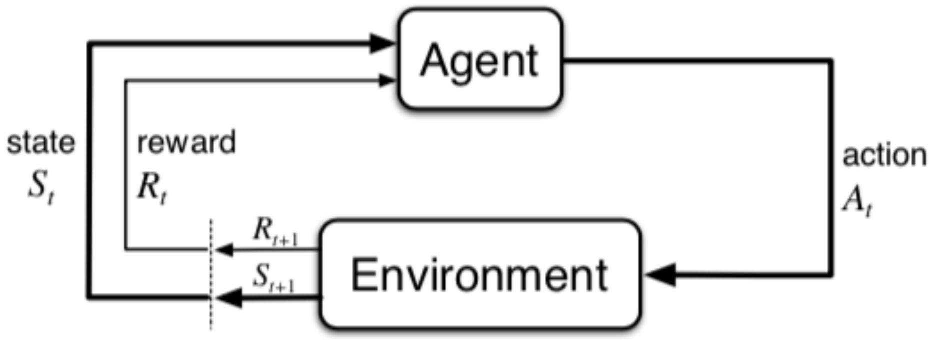


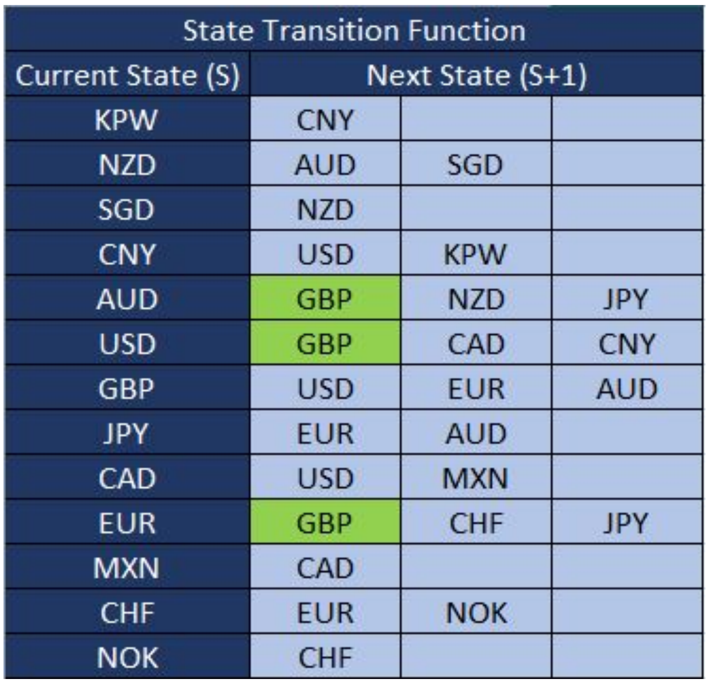
Figure 2. Markov Decision Process

At any given time, the agent observes $s_t \in S$ and chooses $a_t \in A$, this causes the current state to change to s_{t+1} with probability $P(s_{t+1}|\, s_t, a_t)$.
As a result, the domain the agent is in can be formalised with a state transition function $s_{t+1} = \delta(s_t, a_t)$.

Within the environment the agent has been placed in to solve the problem of the optimal currency pair to trade with the GBP, there are 13 currencies which are represented as states/nodes (see figure 1). For all given states there is a finite set of actions from which the agent can choose to take (see table 1 below). From table 1 it can be seen that the agent will have at most three actions to choose from in any given state in the environment.

2.2 Reward Function

As discussed, the agent is given an immediate reward for each action taken from a particular state, this is formalised as $r_t = r(s_t, a_t)$. As seen in figure 1 the agent gets a reward value (5) for each state transition except the transitions from USD, EUR, AUD -> GBP.
These 3 state transitions each give a reward value of (100) as the goal of the agent is to optimize against the base currency GBP.
The environment is modelled with consistent rewards of (100) for each currency adjacent to the GBP in order to remove bias and let the agent learn which currencies are best rather than being guided. These rewards are encoded into an R-matrix which will be discussed in further detail below.



State Transition Function			
Current State (S)	Next State (S+1)		
KPW	CNY		
NZD	AUD	SGD	
SGD	NZD		
CNY	USD	KPW	
AUD	GBP	NZD	JPY
USD	GBP	CAD	CNY
GBP	USD	EUR	AUD
JPY	EUR	AUD	
CAD	USD	MXN	
EUR	GBP	CHF	JPY
MXN	CAD		
CHF	EUR	NOK	
NOK	CHF		

Table 1. State Transition function showing all possible action the agent can take within the environment.

3. Choose a Policy

The learning policy that the agent uses is formally denoted as π, where π at a particular time dictates the action and state the agent is in - $\pi = (a|s)$. Within reinforcement learning π is not static and so can change depending on the agent's experience (Sutton and Barto, 2018).

Within the environment set for the agent, the learning policy implemented is an epsilon greedy policy. This policy is adopted as Q-learning is an on-line algorithm due to not collecting any data before computing, hence the Q-matrix values being (0) initially. it is best suited to address the exploration vs. exploitation dilemma because the agent has no initial understanding of the environment and as such will be looking to explore the environment. This is implemented by setting a relatively high ε value of 0.9 which is known formally as the exploration factor, then generating a uniform random number - $0 \leq rd \ \leq 1$ which dictates whether the agent will explore or exploit. At each iteration of the algorithm ε decays when multiplied by 0.99999 if it is equal to or higher than 0.5, or by 0.9999 if lower than 0.5. This ensures that at the beginning, the agent has a higher chance of exploring the environment but after a significant amount of iterations it has a higher chance of exploiting – maximizing rewards.

4. Represent the problem with a Graph and set the original R-matrix

The agent Is placed in an environment with 13 currencies as states. As described the task of the agent is to learn which term currency Is optimal to trade against the base currency GBP (see figure 3 below). The currencies that are closer to the GBP have more edges as they are impacted by both smaller economy currencies and other 'major' economy currencies.

The reward that the agent receives for each transition between the states representing developing currencies and commodity currencies is (5), however the reward that the agent receives each adjacent currency/state to the GBP state is (100) as this is the state the agent is attempting to reach each time. As a result, the reward function is represented as an R-matrix (see table 2 below). The R-matrix is representation of the state transition function and the graph representation of the RL problem the agent is attempting to solve as the structure followed is:

-- denotes an illegal action the agent cannot perform in a given state
5 denotes a reward of 5 for a legal transition from one state to another
100 denotes a reward of 100 for a legal transition from one state to another.

As seen from the R matrix, rows are states which represent currencies and columns are states which represent currencies, and this all together produces the rewards which an agent receives from an action taken in a given state.

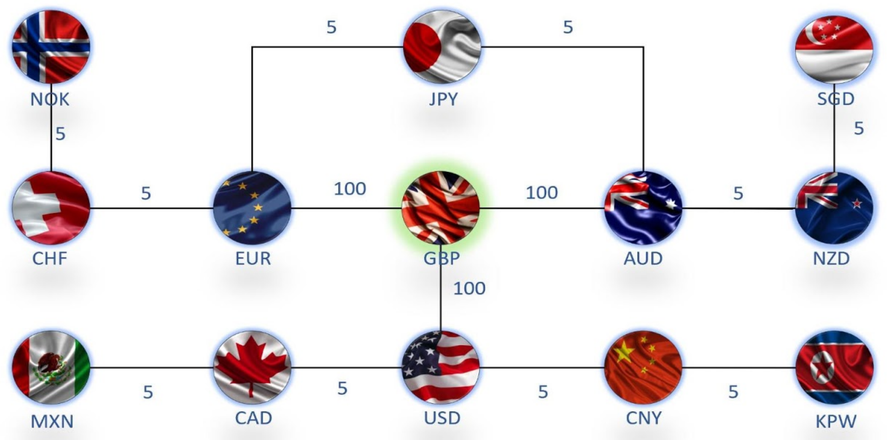


Figure 3. The Graph Representation of Environment

BASE CURRENCY \ TERM CURRENCY R - MATRIX													
BASE\TERM	KPW	NZD	SGD	CNY	AUD	USD	GBP	JPY	CAD	EUR	MXN	CHF	NOK
KPW	--	--	--	5	--	--	--	--	--	--	--	--	--
NZD	--	--	5	--	5	--	--	--	--	--	--	--	--
SGD	--	5	--	--	--	--	--	--	--	--	--	--	--
CNY	5	--	--	--	--	5	--	--	--	--	--	--	--
AUD	--	5	--	--	--	--	100	5	--	--	--	--	--
USD	--	--	--	5	--	--	100	--	5	--	--	--	--
GBP	--	--	--	--	100	100	--	--	--	100	--	--	--
JPY	--	--	--	--	5	--	--	--	--	5	--	--	--
CAD	--	--	--	--	--	5	--	--	--	--	5	--	--
EUR	--	--	--	--	--	--	100	5	--	--	--	5	--
MXN	--	--	--	--	--	--	--	--	5	--	--	--	--
CHF	--	--	--	--	--	--	--	--	--	5	--	--	5
NOK	--	--	--	--	--	--	--	--	--	--	--	5	--

Table 2. R-matrix

5. Set the parameter values for Q-learning

The formal representation of the Q-learning algorithm as shown by Sutton and Barto

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

Figure 4. Q-learning Algorithm by Watkins

(2019) (see figure 4 above) shows that there are three parameters that affect the agent's behaviour and learning in within the environment: α (alpha), γ (gamma), ε (epsilon).

Alpha is known as the agents learning rate because its multiplied by the error which is the current estimate of the Q-value minus the agent's old Q-value. This value dictates how fast the agents learning takes place as without alpha the old Q-values will cancel each other out and the update will simply be the new Q-value. The designer must attempt to find the balance between setting a high alpha where learning may occur quickly but not accurately, and a low alpha such as (0) where Q-values are never updated.

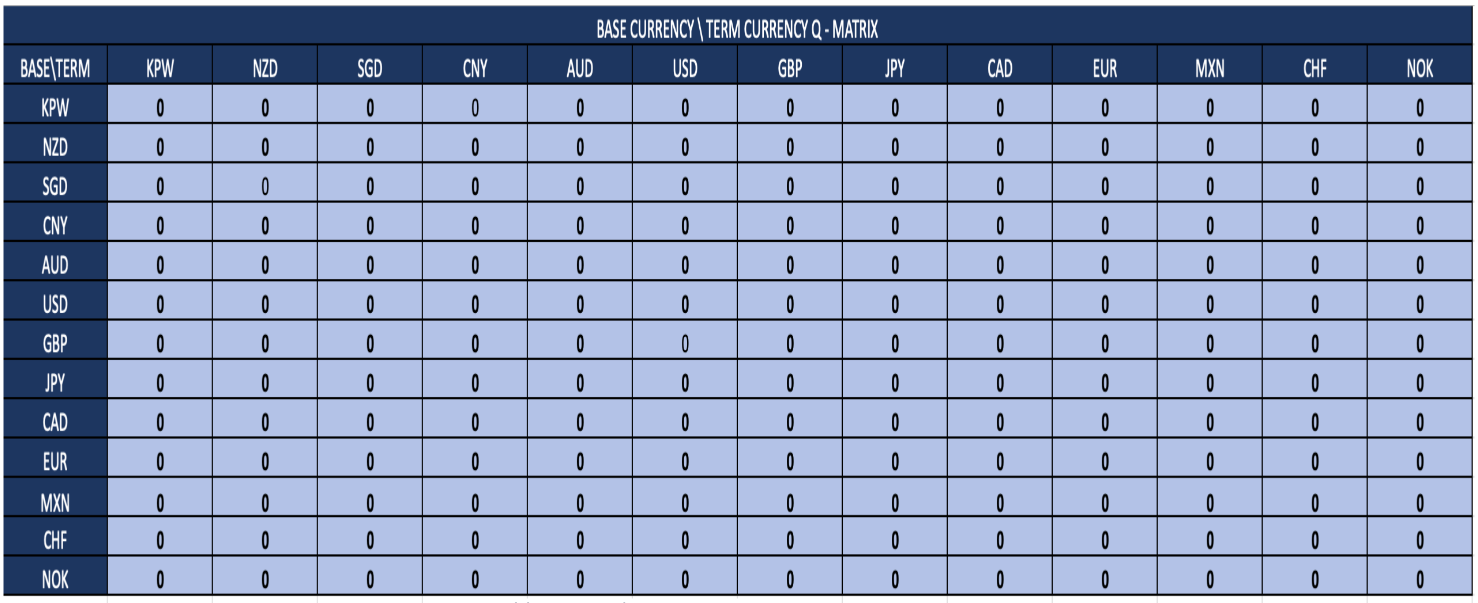
Gamma is known as the discount factor for future rewards. When gamma is (0), the only immediate rewards matter to the agent, however when gamma is (1) the agent views all rewards as being equally weighted.

Epsilon as described earlier is the exploration factor, and upon each iteration of the algorithm decays meaning that the agent is more likely to explore the environment initially and exploit after many iterations.

The initial values set for the agent in the environment are:
$a = 0.8$
$\gamma = 0.75$
$\varepsilon = 0.9$

6. Show how the Q-matrix is updated in a learning episode

The Q-matrix is the agent's memory and represents what the agent has learnt from past experiences in the environment. This Q-matrix has the same dimensions as the reward matrix and at the start is a zero matrix as the agent has no prior information of the environment or any data (see table 3).



BASE CURRENCY \ TERM CURRENCY Q - MATRIX													
BASE\TERM	KPW	NZD	SGD	CNY	AUD	USD	GBP	JPY	CAD	EUR	MXN	CHF	NOK
KPW	0	0	0	0	0	0	0	0	0	0	0	0	0
NZD	0	0	0	0	0	0	0	0	0	0	0	0	0
SGD	0	0	0	0	0	0	0	0	0	0	0	0	0
CNY	0	0	0	0	0	0	0	0	0	0	0	0	0
AUD	0	0	0	0	0	0	0	0	0	0	0	0	0
USD	0	0	0	0	0	0	0	0	0	0	0	0	0
GBP	0	0	0	0	0	0	0	0	0	0	0	0	0
JPY	0	0	0	0	0	0	0	0	0	0	0	0	0
CAD	0	0	0	0	0	0	0	0	0	0	0	0	0
EUR	0	0	0	0	0	0	0	0	0	0	0	0	0
MXN	0	0	0	0	0	0	0	0	0	0	0	0	0
CHF	0	0	0	0	0	0	0	0	0	0	0	0	0
NOK	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 3. Initial Q-matrix

The agent will start in a particular initial state and will move from state to state given the Q-learning algorithm parameters set until the goal state is reached. Once the goal state – GBP is reached, this signals the end of an episode and another begins. The agent is to explore the

R-matrix environment each episode and the aim at the end is to have an optimized Q-matrix which holds values representing the agent's memory. The aim is to build an ever-optimized Q-matrix through episodes which act as learning experiences similar to how humans and animals learn. Upon optimizing Q-matrix values the agent is expected to converge to the goal state with the optimal currencies – in this case the base currency GBP is the optimal state.
The algorithm can be modelled in pseudo code like so:

```
1. Initialise $\alpha, \gamma, \varepsilon$
2. Initialise Q-matrix to 0
3. For Each Episode
    a. Select an Initial state
    b. While GBP state hasn't been reached
        i. Choose an Action a from State s using , $\varepsilon$ greedy policy
        ii. Take Action a, observe Reward r, look at the next possible states
            actions
        iii. Select highest Q-value for next state considering all possible actions
        iv. $Q_{new}(s,a) = Q_{old}(s,a) + \alpha\ [(r(s,a) + \gamma \max Q(s_{t+1}, a_{all})) - Q_{old}(s,a)]$
        v. Set the next state s' as current state s
End While.
End For.
```

In a observing a demonstration of a learning episode from the agent following this Q-learning algorithm, let us set $a = 0.8, \gamma = 0.75, \varepsilon = 0.9$. Episode:

Step 1

1. With the Initial state set to CNY
2. Check to see GBP hasn't been reached
3. The Agent looks at the fourth row in the R-matrix and can see that there are only 2 options – KPW, USD.
4. By Random selection the agent chooses state USD as the action.
5. Now the agent is in state USD it looks at the sixth row in the R-matrix, where it has 3 possible actions go to states: CNY, GBP, CAD.
6. Q_{new}(CNY,USD) = Q_{old}(CNY, USD) + α [r(CNY,USD) + γ max [Q(USD, CNY),(USD,GBP),(USD,CAD)] - Q_{old}(CNY,USD)]
7. Q_{new}(CNY,USD) = 0 + 0.8[5 + 0.75*max[0,0,0] - 0] = 4.
8. Next State USD becomes current state.

Since the goal state has not been reached we loop through this algorithm again:

Step 2

1. Current State is USD
2. Agent looks at sixth row in the R-matrix and can see that there are 3 options – CNY, GBP, CAD.
3. By random selection the agent chooses state GBP as the action.
4. Now the agent is in state GBP it looks at the 7th row in the R-matrix, where it has three possible actions: AUD, USD, EUR.
5. Q_{new}(USD, GBP) = Q_{old}(USD, GBP) + α [r(USD, GBP) + γ max [Q(GBP,AUD),(GBP, USD),(GBP,EUR)] - Q_{old}(USD,GBP)]
6. Q_{new}(USD, GBP) = 0 + 0.8[100 + 0.75*max[0,0,0] - 0] = 80

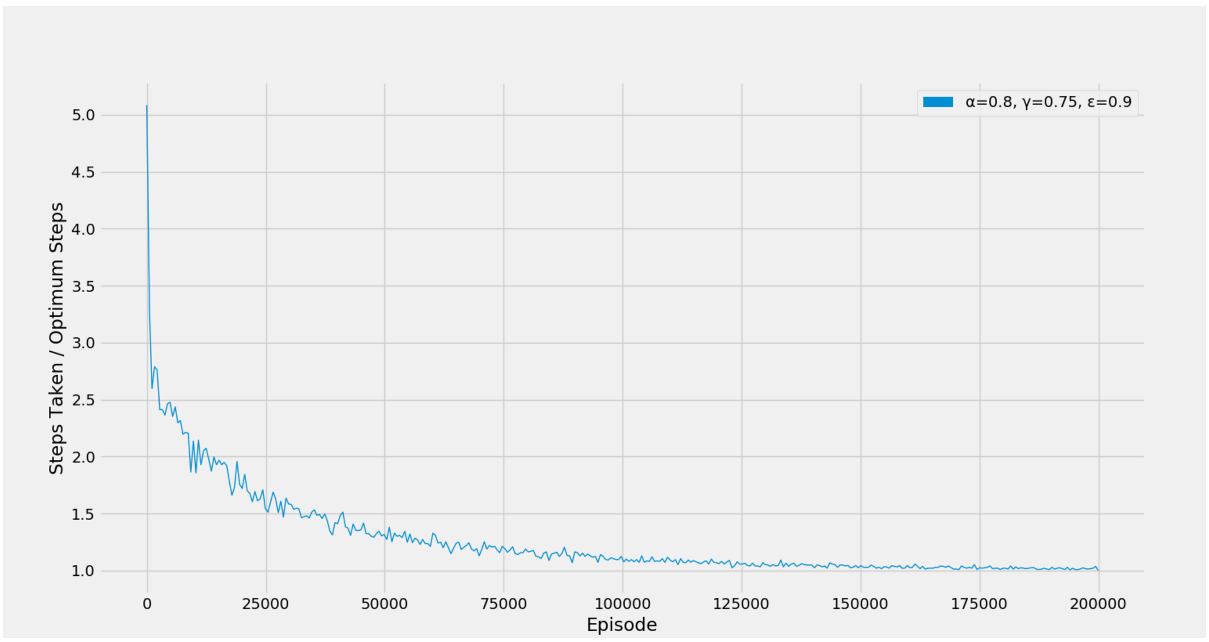
7. Next State GBP becomes current state
8. Complete episode since GBP goal state is reached.

The agent has completed a single episode and has updated the information it has learned in the Q-matrix from both steps (see Table 4 below).

BASE CURRENCY \ TERM CURRENCY Q - MATRIX													
BASE\TERM	KPW	NZD	SGD	CNY	AUD	USD	GBP	JPY	CAD	EUR	MXN	CHF	NOK
KPW	0	0	0	0	0	0	0	0	0	0	0	0	0
NZD	0	0	0	0	0	0	0	0	0	0	0	0	0
SGD	0	0	0	0	0	0	0	0	0	0	0	0	0
CNY	0	0	0	0	0	4	0	0	0	0	0	0	0
AUD	0	0	0	0	0	0	0	0	0	0	0	0	0
USD	0	0	0	0	0	0	80	0	0	0	0	0	0
GBP	0	0	0	0	0	0	0	0	0	0	0	0	0
JPY	0	0	0	0	0	0	0	0	0	0	0	0	0
CAD	0	0	0	0	0	0	0	0	0	0	0	0	0
EUR	0	0	0	0	0	0	0	0	0	0	0	0	0
MXN	0	0	0	0	0	0	0	0	0	0	0	0	0
CHF	0	0	0	0	0	0	0	0	0	0	0	0	0
NOK	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 4. Completed Episode Q matrix

7. Represent Performance vs Episodes with Qualitative and Quantitative analysis



Graph 1: Episode vs Steps Taken / Optimum Steps

In any given environment with our agent having, learning rate (α) 0.8 and discount rate (γ) 0.75, we can observe a relative behaviour between exploration factor(ε) and our agent

Behaviour of agent with respect to exploration factor(ε)
.
Proposition:

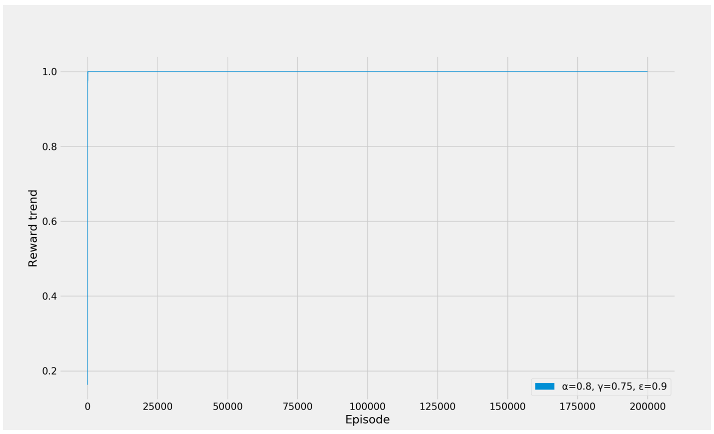
When exploration factor (ε) = 1, agent is concentrated towards exploration.
When exploration factor (ε) = 0, agent is concentrated towards exploitation.

Observation:

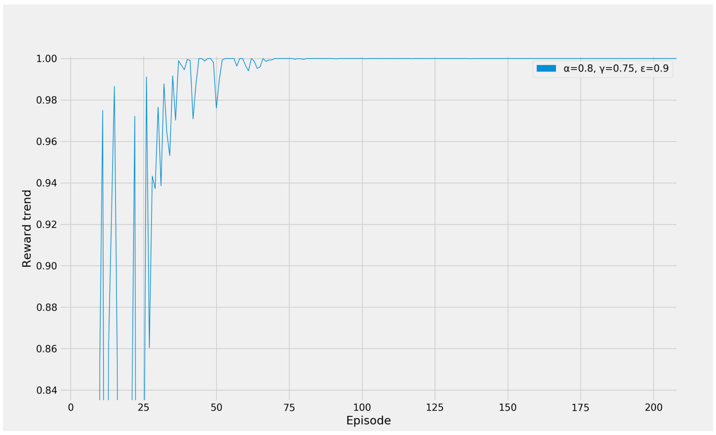
We can observe this behaviour of our agent on the *graph 1.0*. We initially start with the exploration factor (ε) 0.9 and it tends decrease exponentially as we go through the episodes. Similarly, on average, our agents take the highest number of steps per episode for about first twenty episodes, which represents that, our agent is spending more time in exploring.

As we go through episodes, we can see the exponential decrease in average steps taken by our agents. In other words, our agent is spending more time towards exploiting. This is because, the exploration factor (ε) is exponentially tending towards 0.

Ideally, at an n^{th} episode where n being infinite, exploration factor (ε) will be zero. As a result, our agent will spend 100% of its time in exploiting. But practically, exploration factor (ε) will be nearly equal to zero but will never reach zero. That means, our agent will have negligibly small probability of exploring.



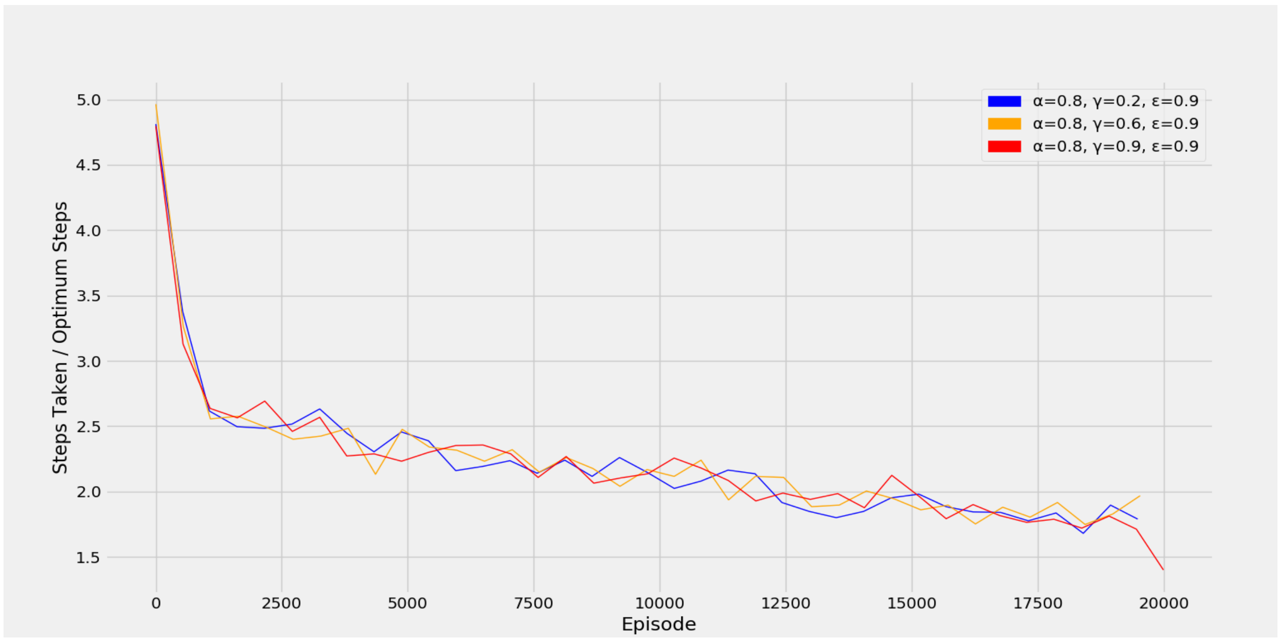
Graph 2.0: Episode vs Reward Trend



Graph 2.1: Zoomed version of Graph 2.0

Even though our agent starts exploration factor (ε) from 0.9, the reward only comes to play when we are exploiting. The reward navigates the agent towards the goal states. $\max_a Q(S_{t+1}, a)$ in $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]$ picks the maximum reward from agent's memory(Q-Matrix). Meaning, while exploiting, agent takes the path which yields maximum reward from the Q-Matrix. But it is not the case while agent is exploring. In the process of exploring, our agent moves randomly irrespective of the state's reward. As a result, there is very high probability of going to the states; otherwise, there is no way of to reach to those states. We can observe this in both on graph 1 and graph 2.1(Zoomed version of graph 2.0). During the process of exploring, reward graph points (graph 2.1) are unstable and quite noisy. But as exponentially tending towards 0, we can see the stabilisation of rewards. Stabilisation of rewards not necessarily mean that the agent is optimum in reaching the goal state. But to reach the optimum steps, an agent must have stabilised reward.

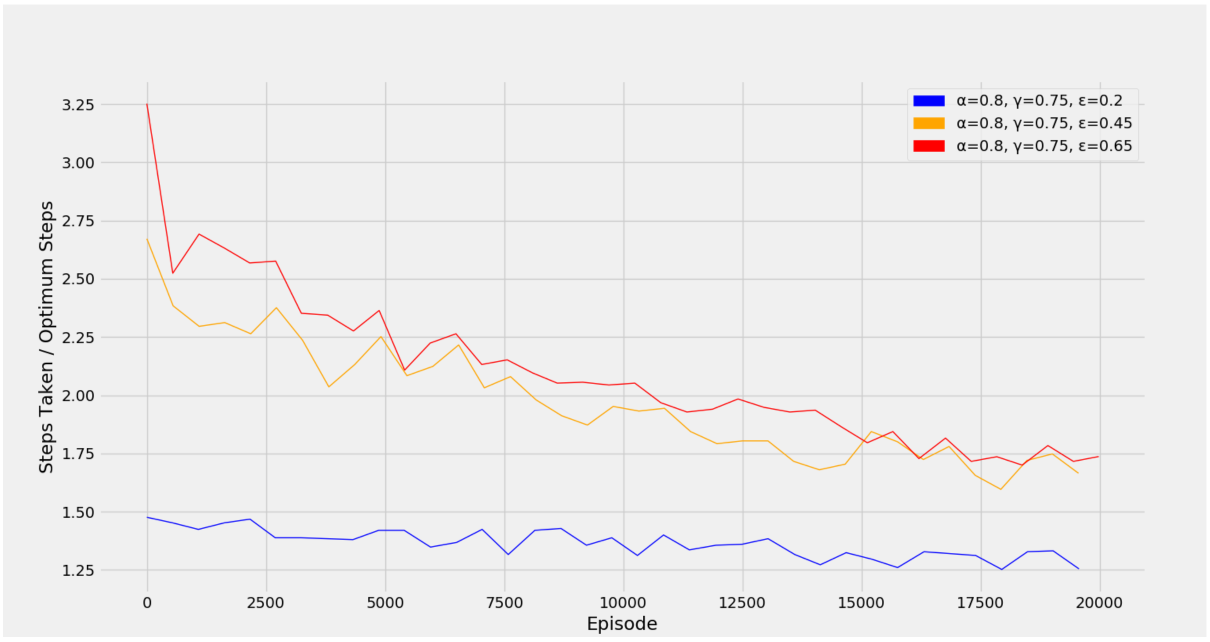
Experimenting with Different parameter values – Alpha, Gamma, Epsilon



Graph 3.Varying gamma Values

<u>Gamma Values Experimentation</u>

In table 5 above we run the agent through 20,000 episodes to find the optimal gamma rate. As discussed earlier gamma is discount factor for future rewards. Within the first 1250 episodes there is real distinction between the various gamma values:
0.2
0.6
0.9.
Using a gamma value of 0.9 is slightly more beneficial in the first 1250 episodes, however over the 2000-episode mark there is a sudden spike in the curve's gradient.
The results in table 5 show a clear relationship between a higher gamma rate and the convergence to optimal steps.
From the results we can see that when the agent placed more equal value on rewards it converged to optimality at a quicker rate hence gamma (0.9) finishing with a step taken to optimal steps ratio of approximately 1.2.
One surprise in the results however is the rate of volatility among each of the gamma values. The overall trend shows a convergence at around 7500 episodes which is also striking considering the agent had a wide range of 0.7 between the lowest gamma value and the highest gamma value.



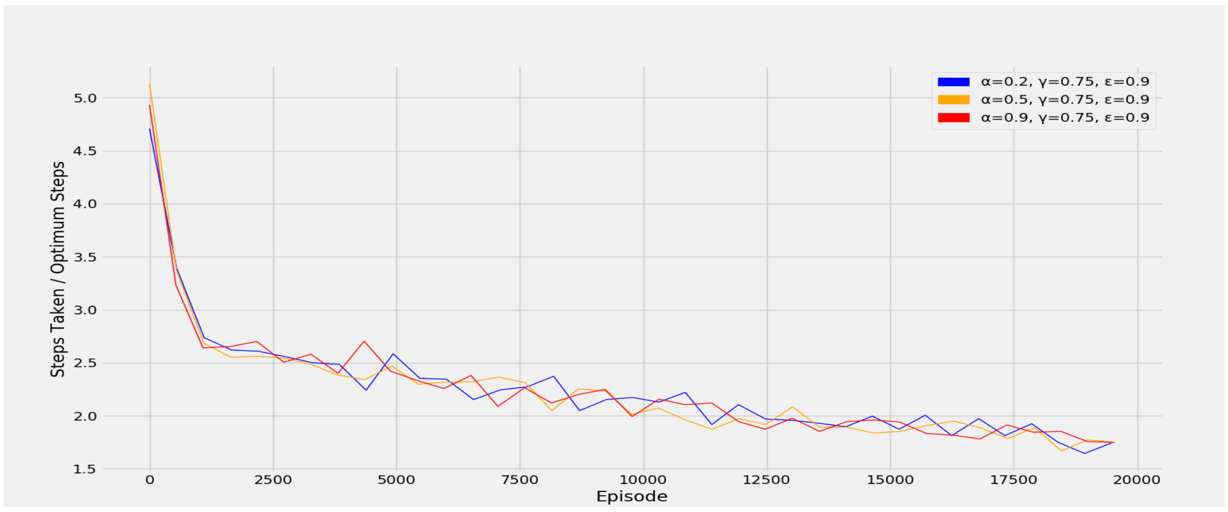
Graph 4. Varying Epsilon Values

Epsilon Values Experimentation

Within the epsilon values is where there was the greatest variation to be seen.
The epsilon values the agent was set with include 0.2, 0.45, 0.65.
The agent still implemented an epsilon greedy policy, with varying degrees of success.
The best parameter for the agent in the environment was setting epsilon to 0.2.
From the very beginning the agent took the 1.5 steps taken/optimal steps. There was some volatility, but the curve shows the agent maintaining efficiency.

Looking at epsilon holding a value of 0.65, the agent initially had the steeps decay to optimality out of all the three values as there was a quick positive learning period of 0.7: 3.25- 2.55.
The agent at epsilon value 0.65 and 0.45 had a very similar trend to optimality, however the greatest separation came at approximately 2100 episodes, when epsilon 0.45 was significantly more optimal.

Graph 5. Alpha Values experimentation



Alpha Values Experimentation

The alpha values experimented with for the software agent were: 0.2, 0.5, 0.9

Of each of the experiments this proved the most inconclusive as each of the different learning rates resulted in the agent finishing approximately at 1.75 steps taken to optimum steps ratio.
The greatest difference among learning rates occurred at 4,600 episodes when the agent with the alpha value of 0.9 was greater by (0.6) compared to the alpha value 0.2.
However, in looking at the time spent with the most optimal steps ratio the alpha rate of 0.9 was more optimal for longer periods of the 20,000 episodes.

Final Analysis

From the Experiments run one can conclude that the way to reach the optimal term currency GBP is with a relatively high learning rate which is at least 0.8, a relatively low epsilon value of no greater than 0.4 and relatively high gamma rate of at least 0.8.

The agents environment is relatively simple however, and with a future attempt the designers could look to adding more value to the environment through the R-matrix, and decreasing the difference between the reward state values and other values.

Establish parallelisms between the Q-learning algorithm and error correction models in psychology

As mentioned Q-learning is an ‘off policy temporal difference algorithm’ as described by Sutton & Barto (2018). Q-learning algorithms can learn from experience without the need of a model and can update estimates based on based on previous learned experience without waiting for a final outcome.

Ivan Pavlov’s experimentation by measuring dog’s salivary reflex resulted in the discovery of classical conditioning. His experiments demonstrated associative learning (AL) by creating a conditioned stimulus (CS) each time an unconditional stimulus was presented (US) and thus the unconditioned response (UR) of the dog’s salivation became a conditioned response (CR) to the CS – the bell ring. Classical conditioning produces blocking/error when the animal doesn’t learn the CR when a potential CS is presented. As a result, the Rescorla Wagner (RW) model was produced as an error correction model for AL.

The RW model aims to calculate how much a stimulus predicts a given outcome. Temporal difference and hence Q-learning is a real-time implementation of the RW model as the RW model deals with changes as on a single trial basis.

The RW model has an error term in the learning algorithm similar to Q-learning. The error term in the RW model is: $\lambda^n - \sum_{i=A}^{Z} V_i^{n-1} * X_i^n$. The asymptote λ^n represents the maximum amount of learning that a US can obtain – 1. The total value of the last outcome prediction is multiplied by the presence of stimulus i. As long as a stimulus is present the error is exponentially decaying at the rate of the outcome of the last prediction. Likewise, the Q-learning error is the difference between the current estimate and the previous Q-value which is the agent’s past estimation. The learning rate α is also multiplied by the error as in Q-learning, however the learning rate can be arbitrarily set whereas in RW it depends on the stimulus' characteristics.

The RW model also implements an eligibility trace which can also be found in Q-learning methods as well. In the RW model the eligibility trace $e_{i,j}^{t}$, is multiplied by gamma and a parameter that regulates the decay. Due to this the learning rate of the stimulus is exponentially growing as time increases.

The RW model is also similar to the Q-learning algorithm in that the stimulus consists of unique components that have their own associative strength (V). The components are activated then decay. This is similar to the exploration factor - epsilon which if implementing a greedy policy decays as time increases.

Propose how error correction models could be implemented as (advanced) reinforcement learning architectures

A type of advanced reinforcement learning architecture is Nash Q-learning. This architecture is useful applied to a problem spaces such as repeated or stochastic games, where each agents reward depends on the joint action of all agents and the current state (Hu and Wellman, 2019). In this scenario decision makers take actions following a strategy and at the end of the game each of the players receive a reward. This algorithm is useful for agents looking to maximize reward against opponents.

As previously discussed the Rescorla Wagner (RW) model aims to calculate how much a stimulus predicts a given outcome.

One proposal of how error correction could be implanted in Nash Q-learning is by representing competing agents as different competing stimuli within the same animal. These different stimuli would attempt to maximise their own utility at the expense of other stimuli, each aiming to gain the highest associative strength (V). The implementation of this may produce a Nash equilibrium among the competing stimuli. This would produce declining errors for some stimuli however due to their competing interests the errors may actually increase for each time step causing a longer curve to the Asymptote - λ^{n}.

Another type of advanced reinforcement learning architecture is serial and simultaneous configural-cue compound stimuli representation for temporal difference (SSCC). SSCC TD is more advanced than regular TD as its algorithms show simultaneous and serial compound stimulus. It uses configural cues and compound effects and as a result gives a wider range of outcomes the TD algorithm can successfully predict. This architecture can implement an error correction model as the temporal difference can represent the co-occurrence of multiple stimulus at each time period and compute the associative strength of a compound component as the total strength of its sub-elements. Each time step is a cue initialised for any given stimulus component.

8.0 Double Q - Learning

The future maximum action values in the Q-learning is estimated using the same Q-Matrix. This sometimes overestimates the action values due to the maximum bias and decrease the performance of learning. This can be avoided by having a variant Q-Matrix. This Double Q-Learning has off-policy where different policies can be used for action value evaluation.

To implement this, we take 2 Q-Matrix and train separately as seen from the image below (Wikipedia, 2019).

$$Q_{t+1}^A(s_t,a_t) = Q_t^A(s_t,a_t) + \alpha_t(s_t,a_t)\left(r_t + \gamma Q_t^B\left(s_{t+1}, \arg\max_a Q_t^A(s_{t+1},a)\right) - Q_t^A(s_t,a_t)\right)$$

$$Q_{t+1}^B(s_t,a_t) = Q_t^B(s_t,a_t) + \alpha_t(s_t,a_t)\left(r_t + \gamma Q_t^A\left(s_{t+1}, \arg\max_a Q_t^B(s_{t+1},a)\right) - Q_t^B(s_t,a_t)\right)$$

Code implementation is given here

```
# Double Q Learning Algorithm

if switch <= 0.5:
    # Q1[s,a] = Q1[s,a] + ALPHA * (reward + GAMMA * Q2[s_,a_] - Q1[s,a])
    config["QM1"][current_state][next_state] += learning_rate * (
            (config["RM"][current_state][next_state] + gamma *
config["QM2"][next_state][
             maximum_reward_state_val(get_states(next_state, config["QM1"]))[0]]
             ) - config["QM1"][current_state][next_state])
else:
    # Q2[s,a] = Q2[s,a] + ALPHA * (reward + GAMMA * Q1[s_,a_] - Q2[s,a])
    config["QM2"][current_state][next_state] += learning_rate * (
            (config["RM"][current_state][next_state] + gamma *
config["QM1"][next_state][
             maximum_reward_state_val(get_states(next_state, config["QM2"]))[0]]
             ) - config["QM2"][current_state][next_state])
```

Full code of Double Q-Learning is given in appendix

8.1 Analysing the result Double Q - Learning and comparing with Q - Learning

The graph 3.0, 4.0 is a steps vs episode graph. We can observe that it has levels where, the top level being the most steps taken and the bottom level (1st level) being optimum steps taken. For the representation purpose we are normalising steps where the optimum steps taken to a goal state from any given initial state being 1 step. So, bottom level will be a single straight line parallel to x-axis from the step 1.

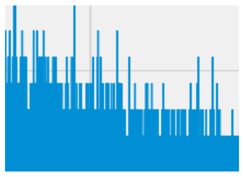
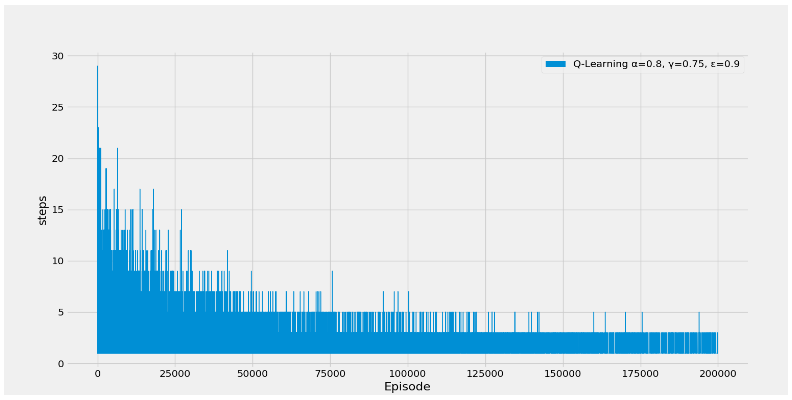


Figure 5: 4, 3 and 2nd level

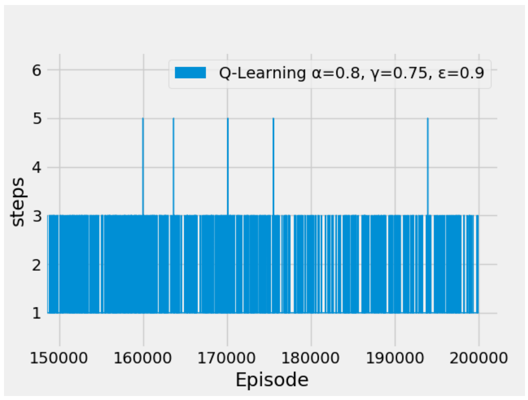
As we go throughout the episodes, it will deprecate each layer from the top. This shows the trend of our agent. Time taken to deprecate each layer is exponential to the layer it is in. For instance, on *graph 3.0* we can observe that it takes nearly 40,000 episodes deprecate 5th level and nearly 65,000 episodes to deprecate 4th level and nearly 115,000 episodes to deprecate 3rd level and so on. Here, higher the deprecation rate on any given average episode, higher the performance.

Figure 6: No Deprecation

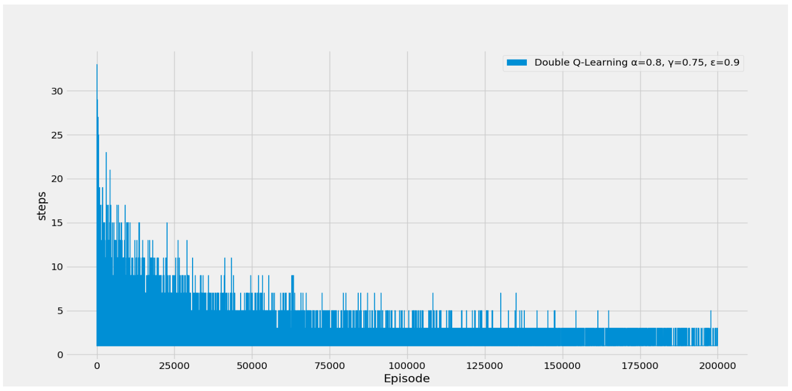
Figure 7: Some ratio of deprecation



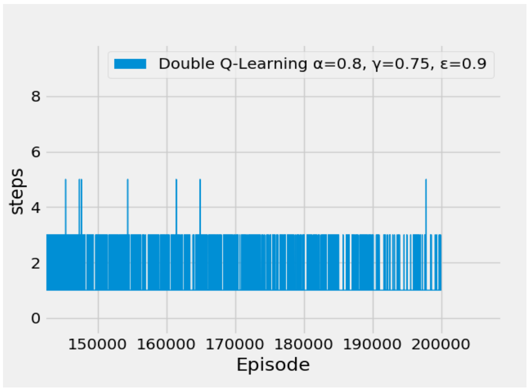
Graph 3.0: Q-Learning Episode vs Steps



Graph 3.1: Snippet of Graph 3.0



Graph 4.0: Double Q-Learning Episode vs Steps



Graph 4.1: Snippet of Graph 4.0

Comparison of Q-Learning to Double Q-Learning shows that the Double Q-Learning has the higher deprecation rate on any given average episodes is higher than Q-Learning. For instance, *graph 4.1* represents the zoomed version of 2nd level of the Double Q-Learning (*graph 4.0)* and *graph 3.1* represents the zoomed version of 2nd level of the Q-Learning (*graph 3.0)*. If we compare *graph 3.1* and *graph 4.1*, we can observe that the Double Q-Learning (*graph 4.1*) has higher deprecation rate on given average episodes from 150,000 to 200,000.

References

Drożdż, S., Górski, A. & Kwapień, J. Eur. Phys. J. B (2007) 58: 499. https://doi.org/10.1140/epjb/e2007-00246-8

Galant, M. and Dolan, B. (2007). *Currency trading for dummies*. Hoboken, NJ: Wiley Publishing, p.21.

Perry, B. (2019). *Forex Currencies: Commodity Pairs (USD/CAD, USD/AUD, USD/NZD)*. [online] Investopedia.com. Available at: https://www.investopedia.com/university/forex-currencies/currencies8.asp [Accessed 15 Mar. 2019].

Sutton, R. and Barto, A. (2018). *Reinforcement Learning, An Introduction*. 2nd ed. Cambridge, Massachusetts: MIT Press.

Hu, J. and Wellman, M. (2019). Nash Q-Learning for General-Sum Stochastic Games. *Journal of Machine Learning Research 4 (2003)*, [online] pp.1039-1069. Available at:

https://www.readkong.com/page/nash-q-learning-for-general-sum-stochastic-games-5772044?p=1 [Accessed 28 Mar. 2019].
Wikipedia. (2019). *Q-learning*. [online] Available at: https://en.wikipedia.org/wiki/Q-learning [Accessed 4 Apr. 2019].

Appendix

Figure 1 - Q-Learning.py

```
import datetime
from random import randint, uniform, choice
from libs.fileio import fileio
from Raw_input import STF, RM, QM, OPG, QFINAL


def display_matrix(list):
    for lst in list:
        print(lst)


def optimum_states(state):
    return len(OPG[state]) - 1


def optimum_rewards(cord1, cord2):
    return QFINAL[cord1][cord2]


def write_for_graph(graph_input, filename):
    fileio().set_File_Name(fileio().root_path("/data/{}.DAT".format(filename)))
    print("{},{},{},{},{}".format(graph_input["episode"], graph_input["steps"],
graph_input["cumulative_rewards"], graph_input["epsilon"],
graph_input["initial_state"]))
    fileio().write_file("{},{},{},{},{}\n".format(graph_input["episode"],
graph_input["steps"], graph_input["cumulative_rewards"], graph_input["epsilon"],
graph_input["initial_state"]))


def write_cache(graph_input):
    fileio().set_File_Name(fileio().root_path("/tool/{}".format("graph.cache")))
    fileio().write_file("{},{},{},{},{}\n".format(graph_input["episode"],
graph_input["steps"], graph_input["cumulative_rewards"], graph_input["epsilon"],
graph_input["initial_state"]))


def flush_cache():
    fileio().set_File_Name(fileio().root_path("/tool/{}".format("graph.cache")))
    fileio().file_flush()


def get_states(state, QMatrix):
    state_val = []
    for action in STF[state]:
        state_val.insert(len(state_val), [action, QMatrix[state][action]])
    return state_val

def maximum_reward_state_val(state_val):
    val = 0
    state = choice(state_val)[0]
    for couple in state_val:
        if val < couple[1]:
            val = couple[1]
            state = couple[0]
    return [state, val]


def random_state_val(state_val):
    lst = []
    for state in state_val:
        lst.insert(len(lst), state[0])
    return choice(lst)


def random_state(ranges):
    for i in range(ranges):
        state = randint(0, 12)
        yield state


def step(config):

    # Initialization -----------------------------------------------------------------
    epsilon = config["epsilon"]
    learning_rate = config["learning_rate"]
    gamma = config["gamma"]
    current_state = config["state"]

    # Explore VS Exploit -------------------------------------------------------------
    if uniform(0.0, 1.0) >= epsilon:
        # Exploit
        next_state = maximum_reward_state_val(get_states(current_state,
config["QM"]))[0]
    else:
        # Explore
        next_state = random_state_val(get_states(current_state, config["QM"]))

    # Q Learning Algorithm -----------------------------------------------------------
    config["QM"][current_state][next_state] += learning_rate * (
            (config["RM"][current_state][next_state] + gamma *
maximum_reward_state_val(get_states(next_state, config["QM"]))[1]
             ) - config["QM"][current_state][next_state])

    # Epsilon Variation --------------------------------------------------------------
    if config["epsilon"] <= 0.5:
        config["epsilon"] = config["epsilon"] * 0.99999
    else:
        config["epsilon"] = config["epsilon"] * 0.9999

    config["state"] = next_state
    return config,
config["QM"][current_state][next_state]/optimum_rewards(current_state, next_state)


def episode(Config):
    rewards = 0
    steps = 0
    initial_state = Config["state"]
    while Config["state"] != Config["goal"]:
        results = step(Config)
        steps = steps + 1
        rewards = rewards + results[1]
        Config = results[0]
    return Config, ((steps - optimum_states(initial_state)) + 1), rewards/steps,
initial_state


if __name__ == '__main__':

    flush_cache()
    filename = str(datetime.datetime.now()).replace(":", ".")
    config = {
        "epsilon"       :   0.9,
        "learning_rate" :   0.8,
        "gamma"         :   0.75,
        "state"         :   [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12],
        "goal"          :   6,
        "QM"            :   QM,
        "RM"            :   RM
    }

    graph = {
        "steps"             :   0,
        "cumulative_rewards":   0,
        "episode"           :   0,
        "epsilon"           :   0,
        "initial_state"     :   0
    }

    Episode = 20000
    episode_count = 0
    for state in random_state(Episode):
        episode_count = episode_count + 1
        if state != config["goal"]:
            config["state"] = state
            results = episode(config)
            config = results[0]

            if results[1] != 0 and results[2] != 0:
                graph["steps"] = results[1]
                graph["cumulative_rewards"] = results[2]
                graph["episode"] = episode_count
                graph["epsilon"] = config["epsilon"]
                graph["initial_state"] = results[3]

                write_for_graph(graph, filename)
                write_cache(graph)

    display_matrix(config["QM"])
```

Figure 2 - Double Q-Learning.py

```
import datetime
from random import randint, uniform, choice

import math

from libs.fileio import fileio
from Raw_input import STF, RM, QM, OPG, QFINAL


def display_matrix(list):
    for lst in list:
        print(lst)


def optimum_states(state):
    return len(OPG[state]) - 1


def optimum_rewards(cord1, cord2):
    return QFINAL[cord1][cord2]


def write_for_graph(graph_input, filename):
    fileio().set_File_Name(fileio().root_path("/data/{}.DAT".format(filename)))
    print("{},{},{},{},{}".format(graph_input["episode"], graph_input["steps"],
graph_input["cumulative_rewards"],
                                  graph_input["epsilon"],
graph_input["initial_state"]))
    fileio().write_file(
        "{},{},{},{},{}\n".format(graph_input["episode"], graph_input["steps"],
graph_input["cumulative_rewards"],
                                  graph_input["epsilon"],
graph_input["initial_state"]))


def write_cache(graph_input):
    fileio().set_File_Name(fileio().root_path("/tool/{}".format("graph.cache")))
    fileio().write_file(
        "{},{},{},{},{}\n".format(graph_input["episode"], graph_input["steps"],
graph_input["cumulative_rewards"],
                                  graph_input["epsilon"],
graph_input["initial_state"]))


def flush_cache():
    fileio().set_File_Name(fileio().root_path("/tool/{}".format("graph.cache")))
    fileio().file_flush()


def get_states(state, QMatrix):
    state_val = []
    for action in STF[state]:
        state_val.insert(len(state_val), [action, QMatrix[state][action]])
    return state_val


def maximum_reward_state_val(state_val):
    val = 0
    state = choice(state_val)[0]
    for couple in state_val:
        if val < couple[1]:
            val = couple[1]
            state = couple[0]

    return [state, val]


def random_state_val(state_val):
    lst = []
    for state in state_val:
        lst.insert(len(lst), state[0])
    return choice(lst)


def random_state(ranges):
    for i in range(ranges):
        state = randint(0, 12)
        yield state


def step(config):

    # Initialization ----------------------------------------------------------
    epsilon = config["epsilon"]
    learning_rate = config["learning_rate"]
    gamma = config["gamma"]
    current_state = config["state"]
    switch = uniform(0.0, 1.0)

    # Explore VS Exploit ------------------------------------------------------
    if uniform(0.0, 1.0) >= epsilon:
        # Exploit
        if switch <= 0.5:
            next_state = maximum_reward_state_val(get_states(current_state,
config["QM1"]))[0]
        else:
            next_state = maximum_reward_state_val(get_states(current_state,
config["QM2"]))[0]
    else:
        # Explore
        next_state = random_state_val(get_states(current_state, config["QM1"]))

    # Double Q Learning Algorithm -----------------------------------------------
    if switch <= 0.5:
        # Q1[s,a] = Q1[s,a] + ALPHA * (reward + GAMMA * Q2[s_,a_] - Q1[s,a])
        config["QM1"][current_state][next_state] += learning_rate * (
                (config["RM"][current_state][next_state] + gamma *
config["QM2"][next_state][
                 maximum_reward_state_val(get_states(next_state,
config["QM1"]))[0]]
                 ) - config["QM1"][current_state][next_state])
    else:
        # Q2[s,a] = Q2[s,a] + ALPHA * (reward + GAMMA * Q1[s_,a_] - Q2[s,a])
        config["QM2"][current_state][next_state] += learning_rate * (
                (config["RM"][current_state][next_state] + gamma *
config["QM1"][next_state][
                 maximum_reward_state_val(get_states(next_state,
config["QM2"]))[0]]
                 ) - config["QM2"][current_state][next_state])

    # Epsilon Variation ---------------------------------------------------------
    if config["epsilon"] <= 0.5:
        config["epsilon"] *= 0.99999
    else:
        config["epsilon"] *= 0.9999

    # -------------------------------------------------------------------------
    config["state"] = next_state
    if switch <= 0.5:
        normalised_reward = config["QM1"][current_state][next_state] /
optimum_rewards(current_state, next_state)
    else:
        normalised_reward = config["QM2"][current_state][next_state] /
optimum_rewards(current_state, next_state)
    # -------------------------------------------------------------------------

    return config, normalised_reward


def episode(Config):
    rewards = 0
    steps = 0
    initial_state = Config["state"]
    while Config["state"] != Config["goal"]:
        results = step(Config)
        steps = steps + 1
        rewards = rewards + results[1]
        Config = results[0]
    return Config, ((steps - optimum_states(initial_state)) + 1), rewards / steps,
initial_state


if __name__ == '__main__':

    flush_cache()
    filename = str(datetime.datetime.now()).replace(":", ".")

    config = {
        "epsilon": 0.9,
        "learning_rate": 0.3,
        "gamma": 0.75,
        "state": [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12],
        "goal": 6,
        "QM1": QM,
        "QM2": QM,
        "RM": RM
    }

    graph = {
        "steps": 0,
        "cumulative_rewards": 0,
        "episode": 0,
        "epsilon": 0,
        "initial_state": 0
    }

    Episode = 200000
    episode_count = 0
    for state in random_state(Episode):
        episode_count = episode_count + 1
        if state != config["goal"]:
            config["state"] = state
            results = episode(config)
            config = results[0]

            if results[1] != 0 and results[2] != 0:
                graph["steps"] = results[1]
                graph["cumulative_rewards"] = results[2]
                graph["episode"] = episode_count
                graph["epsilon"] = config["epsilon"]
                graph["initial_state"] = results[3]

                write_for_graph(graph, filename)
                write_cache(graph)

    print()
    display_matrix(config["QM1"])
    print()
    display_matrix(config["QM2"])
```