ACKNOWLEDGEMENT

First and foremost, we would like to sincerely thank our advisor and mentor Prof. H.C Taneja for the continuous support during our research and study, for his enthusiasm, motivation and for the freedom we were granted throughout the preparation of the major project. He offered us useful advice whenever we asked him and also guided us towards our goal whenever we needed help. His wisdom and experience helped us tremendously towards achieving our goals.

Our sincere thanks also goes to the Department of Applied Mathematics for their support, guidance, and encouragement from the research point of view as well as giving us a platform so that we could perform to the best of our abilities and learn new, advanced techniques.

Deepest gratitude are also due to all members of the panel for their interest in this work and for taking the time to evaluate this project report.

We would also like to thank our families for the constant motivation and encouragement throughout this project timeline. Without their unwavering encouragement, our path towards the completion of the project would have been significantly more difficult.

ADITYA PAREEK	**SHIKHAR BHARDWAJ**	**TUSHAR PRASAD**
2K15/MC/006	**2K15/MC/078**	**2K15/MC/093**

CONTENTS

ABSTRACT

An automatic program that generates constant profit from the financial market is lucrative for every market practitioner. The financial trading task can be described as an agent that interacts with the market to try to achieve some intrinsic goal. It is not required for the agent needs to be human as in modern financial markets, algorithmic trading accounts for large amount of trading activities.

Success is defined in terms of the degree the agent is reaching its intrinsic goal. One of the most fundamental hypotheses of reinforcement learning is that goals of an agent can be expressed through maximizing long-term future rewards. Reward is a single scalar feedback signal that reflects the "goodness" of an agent's action in some state. This is called the reward hypothesis which states that *"All goals can be described by maximization of expected future rewards".*

We try to model such a trading agent leveraging the recent advances in deep reinforcement learning. A Partially Observable Markov Decision Process suitable for financial trading task is modelled and solved with the help of state-of-the-art Deep Recurrent Q-Learning (DRQN) Algorithm. The work is inspired from the paper: "Financial Trading as a Game: A Deep Reinforcement Learning Approach" by Chien-Yi Huang [2018]. Several modifications to the existing learning algorithm are implemented, that make it more suited for financial trading task –

1. A substantially small replay memory compared to the ones used in modern Deep Reinforcement learning algorithms is implemented.
2. An action augmentation technique is implemented that mitigates the need for random exploration by giving extra reward feedback signals for all possible

actions in a particular state to the agent. This enables the model to use greedy exploration policy instead of the commonly used ε-greedy approach. However, this technique is specific to financial trading under a few market assumptions.

3. A longer sequence is sampled for the training of recurrent neural network. As a result, we now can train the agent every T steps, which greatly reduces the time required to train the model as the overall computation required is brought down by a factor of T.

All of the above points are combined into an online learning algorithm and is validated on the spot foreign exchange market.

1. We propose a Partially Observable Markov Decision Process (PO-MDP) model for general signal-based financial trading task solvable by state-of-the-art deep reinforcement learning algorithm with publicly accessible data only.
2. Several modifications to the existing learning algorithm are implemented, that make it more suited for financial trading task. This involves using a substantially smaller replay memory and sampling a longer sequence for training. We also employ a novel action augmentation technique to mitigate the need for random exploration in the financial trading environment.

CHAPTER 1

OUTLINE OF THE REPORT

This report is constructed in a way to explain how the financial domain works and the kind of algorithms we have applied on the data procured from these domains. This flow of contents is made to help the reader understand the work and the pre-requisites required to understand our research and implementation.

Chapter 1 indicates the outline of the report as well as an overview about the work he have done and the research we have put forward.

Chapter 2 lies as an introduction towards our project as well as providing some pre-requisite knowledge. Chapter 2.1 shows our motivation towards taking this project as well as the scope of implementing reinforcement learning on various domains, along with a brief literature review. Chapter 2.2 gives an overview of the financial trade domain, where Chapter 2.2.1 represents Price Chart, Chapter 2.2.2 represents Trade History and Chapter 2.2.3 represents Order Book.

Chapter 3 familiarizes the basics of reinforcement learning to the reader which is a pre-requisite required to understand our model and its implementation which is further introduced in Chapter 3.1. We understand how the model learns to optimize rewards in Chapter 3.1.1. Policy Search is introduced in Chapter 3.2 and the Policy Gradient algorithm is initiated in Chapter 3.2.1. Markov Chains present in Chapter 3.3.1 is present under the subset of Markov Decision Processes present in Chapter 3.3, which is further explained in Chapter 3.3.2. Temporal Difference Learning is introduced in Chapter 3.4 with Q-Learning Algorithm and SARSA Algorithm being a subset of it in

Chapter 3.4.1 and Chapter 3.4.2 respectively. Chapter 3.4.3 introduces the concept of Exploration Policies.

Chapter 4 introduces the idea of Deep Reinforcement Learning. We present a brief introduction of this topic in Chapter 4.1 and then move on to introduce the Deep Q-Network algorithm in Chapter 4.2. Chapter 4.2.1 explores the problems and shortcomings of Deep Q-Networks and the solutions to these problems are mentioned in Chapter 4.2.2. We briefly give an overview of the implementation details of Deep Reinforcement Learning algorithms in Chapter 4.2.3. Double DQNs are explored in the next Chapter 4.3. The concept of Deep Recurrent Q-Networks is introduced in Chapter 4.4. We give an overview of Recurrent Neural Networks (RNN) and Long Short Term Memory (LSTM) in Chapter 4.4.1 and Chapter 4.4.2 respectively. Finally, we explore the usefulness of DRQN in Chapter 4.4.3.

CHAPTER 2

INTRODUCTION

2.1 MOTIVATION AND LITERATURE REVIEW

The motivation for considering this project comes from the fact that the Deep Learning researchers and community has stayed away from modelling on Financial Markets. Some of the reasons causing this problem seems to be disinterest towards financial markets from research perspective and the difficult in obtaining data. We believe that this subject has not received enough attention from the researchers despite the potential to create state-of-the art models.

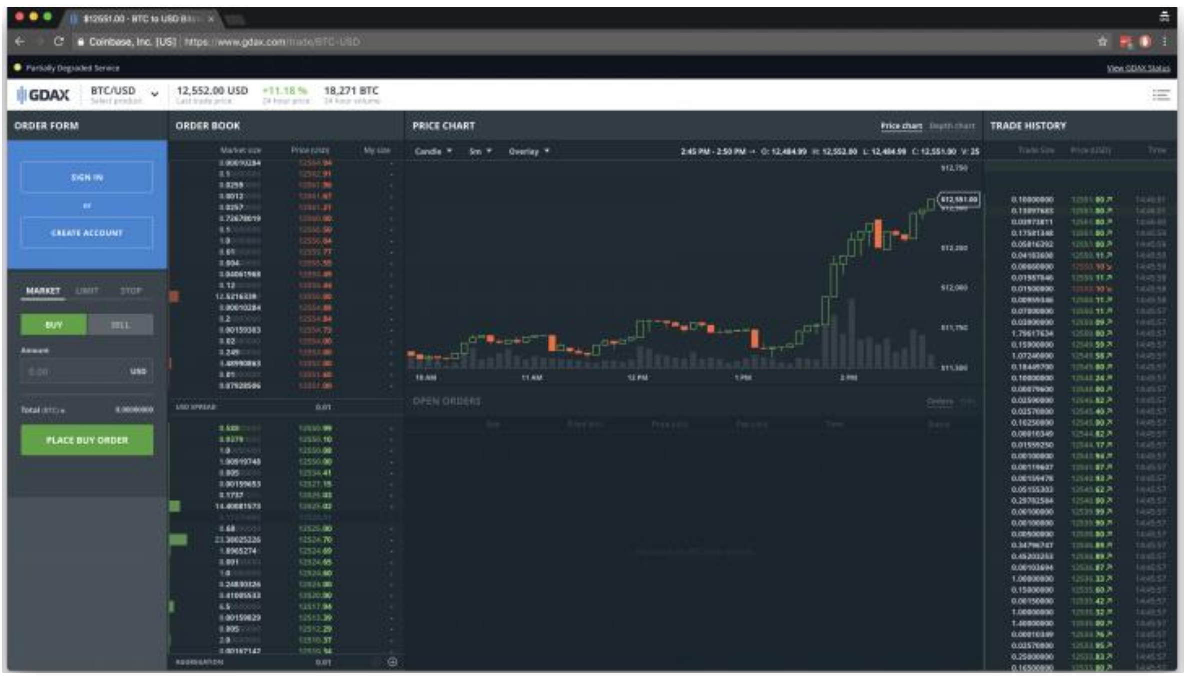
Another fact that leads to us working on this project is the algorithm we are going to implement on the financial market. Among the three dominant machine learning techniques, reinforcement learning is the most sparsely used within various domains, other techniques being supervised learning and unsupervised learning. Taking this project helps us to learn more about the field of reinforcement learning and its applications towards financial markets as well as other domains.

The scope of reinforcement learning in financial domain is vast. Since this is the least applied out of all the major machine learning techniques, a lot of research work can be done to bring better results on domains primarily working on supervised and unsupervised learning.

Work done by Huang, Chien-Yi [1] is one of the only papers which have a baseline model running in the trading domain for reinforcement learning algorithms. Chiao-Ting; An-Pin; Szu-Hao also implemented a agent-based reinforcement learning algorithm [2] to produce cloning strategies from trading records. Yue Deng et al formed a Deep Direct Reinforcement Learning Algorithm [3] for representation of financial signals and its trading. Zhengyao Jiang; Jinjun Liang worked on Cryptocurrency data for Portfolio Management [4].

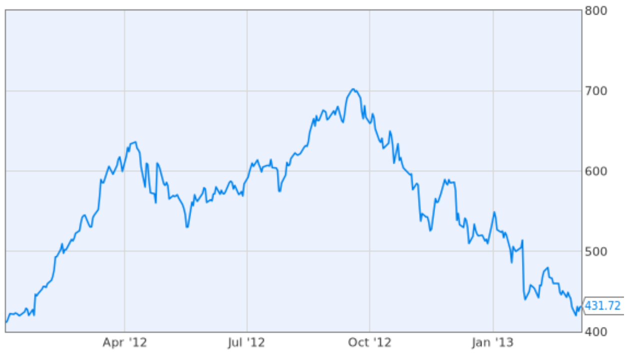
2.2. FINANCIAL MARKETS

Financial trading as well as cryptocurrency trading in most markets happen in an auction called continuous double auction with an open order book on an exchange. This also means that there are buyers and sellers present who get matched and can trade assets amongst themselves. There are various exchanges present in the financial markets which may have different products like U.S Dollar vs Euros or Bitcoin vs U.S Dollars. Apart from that, their interface and their data is very similar.



2.2.1 Price Chart

The current price represents the price of the most recent trade. It depends on the kind of action taken by the trade like buy or sell. The price chart is usually shown as a candlestick chart which for a given time window shows 4 different types of prices; Open (O), High (H), Low (L) and Close(C) prices. Volume (V), which is the total volume of the trades happened within the given time period, is represented by the bars below the given price chart. The volume of a trade represents the market liquidity. It also indicates the quality and veracity of the trend prices, as high volume is usually the consensus of trade participants.

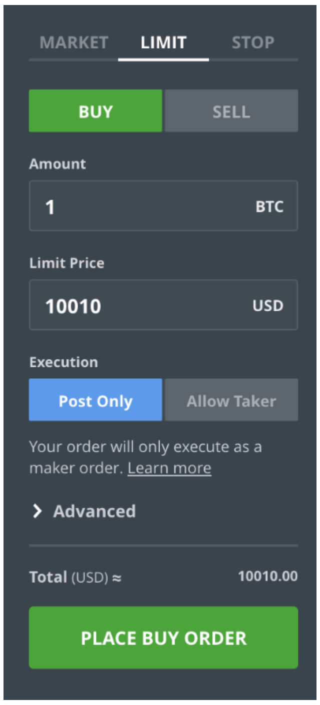


2.2.2 Trade History

The history of all the trades can be viewed on the right hand side. Each trade is represented by a direction (sell or buy), timestamp, price and size. The two parties in the trade are referred as maker and taker.

2.2.3 Order Book

The order book contains information about the selling and buying prices which people are willing to pay. It contains two sides, Bids and Asks. Bids are people who want to buy and Asks are people who want to sell. Best bid is the highest price one is willing to buy and the best sell is the lowest price one is willing to sell. Generally, the best ask is larger than the best bid. Spread is the difference between the best ask and the best bid.



Amount: 1 BTC

Limit Price: 10010 USD

Execution: Post Only

Your order will only execute as a maker order. Learn more

Total (USD) ≈ 10010.00

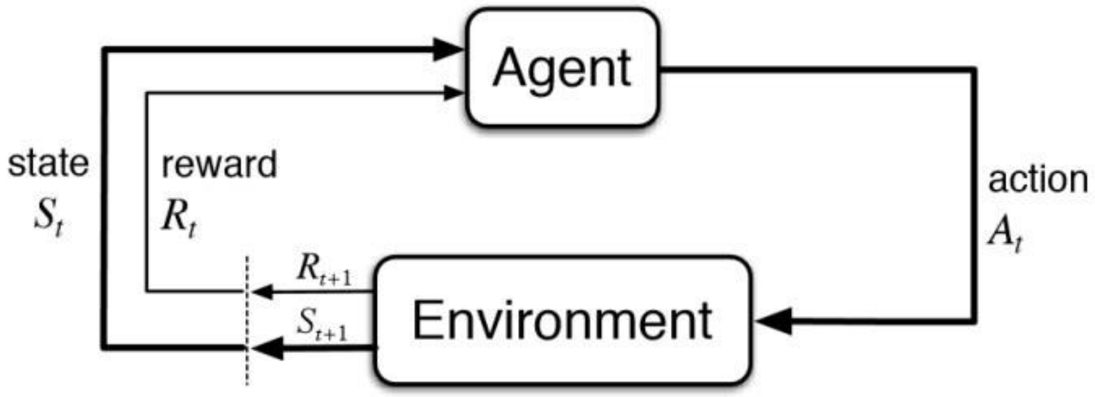
CHAPTER 3

BASICS OF REINFORCEMENT LEARNING

3.1. INTRODUCTION

Reinforcement Learning is one of the oldest as well as one of the most exciting field of machine learning at this moment. It started around 1950s, with various applications over the years, typically related to games like TD-Gammon which was created to solve Backgammon as well as machine control, but it was never highly utilised. In 2013, a start-up called DeepMind made a system which could learn to play any Atari games, revolutionised the field of reinforcement learning. It only required raw pixels as inputs and could beat most humans without any prior knowledge about the rules of the games.

This was followed by a series of remarkable models, including one of the most famous in the domain of reinforcement learning, the victory of AlphaGo over Lee Sedol, who at that time was the world champion of the game Go. Before that, no program had ever come close to beating the champion of a game at his own speciality. In 2014, Google bought DeepMind for 500 million dollars signifying the success of this new wave of Reinforcement Learning.



3.1.1 Learning to optimise rewards

In reinforcement learning, actions are taken by a software agent which makes observations within an environment, which returns rewards back to the agent. The main objective is to learn actions that will increase and hence maximize the expected long-term rewards. Therefore, to summarize, the agent; through trial and error, learns to maximize rewards in an environment.

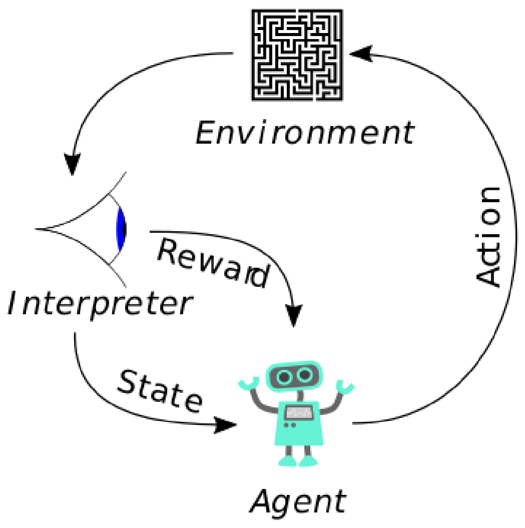
Reinforcement learning can be applied to a wide domain of work. Some of the examples are:-

1. The stock market prices are observed by the agent, which decides when to sell or buy, like the implementation by Jae Won Lee.[5]
2. A board game agent can be programmed like Go.
3. Ms Pac Man controlling agent can be programmed. Here, the simulation of the game is the environment of the game, screenshots are the observations, the joystick positions are the rewards and the game points are rewards.
4. The agent can be the walking robot controller. Here, the agent can observe the environment using sensors like touch sensors and cameras, the signals sent in order to activate monitors will be its actions. A positive reward would occur

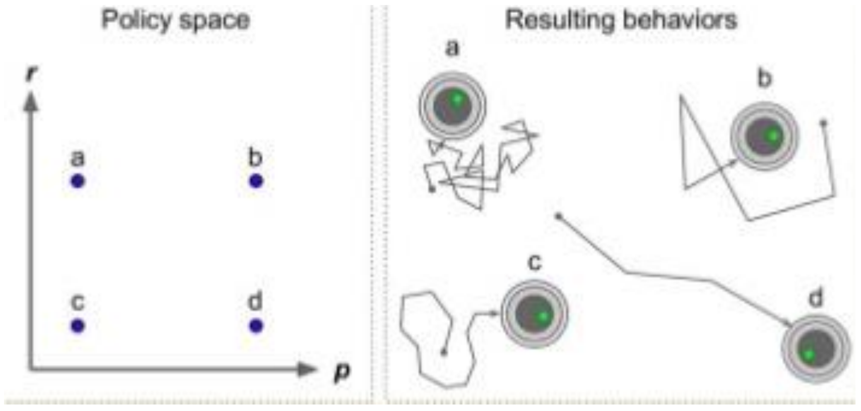
when the robot approaches the target destination and a negative rewards would occur in the opposite case.

3.2 POLICY SEARCH

The algorithm which is used by the agent to determine actions is known as Policy. For example, observations could act as inputs to a policy which outputs the actions which the agent has to take. It is not necessary that the policy has to be deterministic. If we consider a robotic vacumn cleaner where the reward would be the amount of dirt it cleans in 15 minutes, its policy could be to rotate right or left with probability ‘p’ or move forward with a probabbility ‘p-1’. The rotation would lie between -r and +r degrees, due to which it is called a stochastic policy.



In the above mentioned case, there are two ways to train the robot; by tweaking either the angle range r or the probability p. One way of exploring the policy space, is to use the brute force policy search method in which the algorithm tries as many different values of the parameters as possible and pick the most rewarding combination. This is not a good method for parameter exploration since its complexity would be very high. Genetic Algorithms can be another way to to explore the policy space where we generate a population of policies, then kill the worst policies as well as crossover the best policies to create offsprings. This way we can iterate through various generations in order to get the best policy.



Using optimization techniques is another approach to do policy search. This is done by evaluating the gradients of the rewards with regards to the policy parameters and changing these parameters by following the gradient towards higher rewards (gradients ascent). This technique is known as Policy Gradients (PG).

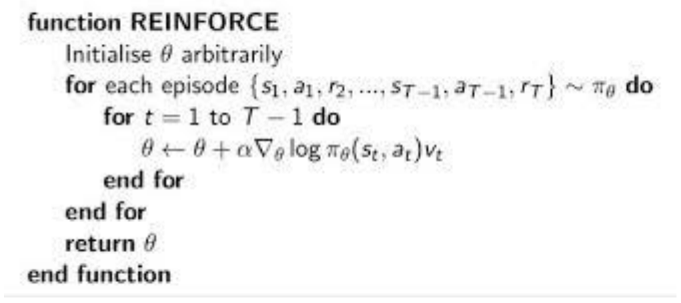
3.2.1 Policy Gradients

As mentioned earlier, Policy Gradient optimizes the policy parameters by following the gradients towards higher rewards. REINFORCE algorithm is one of the most popular policy gradient algorithm. There are several steps we have to follow in order to optimize the parameters.

1. We will let the policy play the game several times and the compute the gradient which would make the chosen action even more likely without actually applying the gradients.
2. After running several episodes, compute the score of each action.

3. If the score of the action is negative, then it represents a bad action and we should apply gradients in the opposite direction to make sure that this action occurs less likely in the future. On the other hand, if the score of the action is positive, then it represents a good action and we should apply gradients in the same direction as the rewards to make the action even more likely to be selected in the future.
4. Finally, we will compute the mean of all the resulting gradient vectors and use these vectors to perform a Gradient Descent iteration.

Algorithm

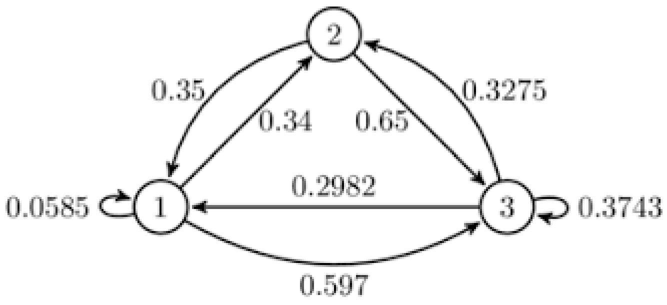


```
function REINFORCE
    Initialise $\theta$ arbitrarily
    for each episode $\{s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ do
        for $t = 1$ to $T - 1$ do
            $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$
        end for
    end for
    return $\theta$
end function
```

3.3 MARKOV DECISION PROCESS

3.3.1 Markov Chains

In the early 20th century, Markov chains were discovered by Andrey Markov, who studied stochastic processes having no memory. This process has a fixed number of states where it randomly evolves one state to another state at each step. This transition from a state s to s' has a fixed probability and it only depends on the pair(s,s') not on the past states, i.e. memoryless system.

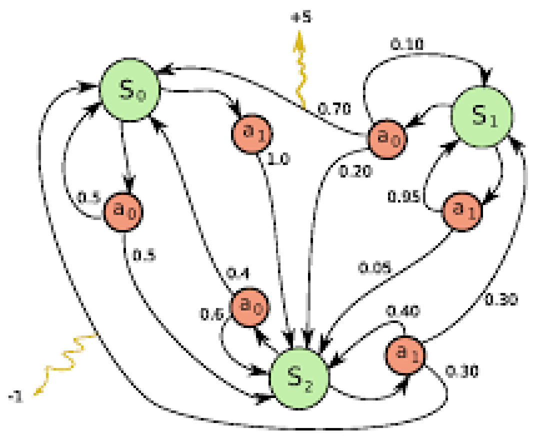


0.0585	0.34	0.597
0.35	0	0.65
0.2982	0.3275	0.3743

The first figure represents the Markov chain as a computational graph having 3 states with probabilities of transforming into different states. The table shows the transition probability matrix.

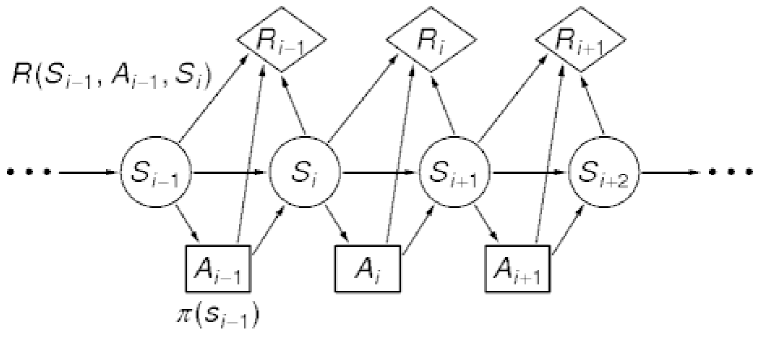
3.3.2 Markov Decision Processes (MDPs)

Markov Decision Process were first described by Richard Bellman in the 1950s. They were very similar to Markov chains. The agent could one of the possible actions at a step where the transition probabilities would also depend on the actions. The only difference would be that the environment returns positive or negative rewards for each action and the agent's goal would be to maximize these rewards over a long period of time.



In the above figure, there are 3 states (S_0,S_1,S_2) and 2 actions (a_0,a_1). If the agent starts at S_0 then it can choose among the two actions. If it chooses a_0 then it has equal probability to stay at S_0 or to move towards S_2 otherwise it will always move to S_2 by choosing the action a_1. At state S_1, if it choose action a_0, then there is a 70% chance to move into state S_0 by getting a reward of +5.

Similarly in state S_2, if it selects action a_1, there is a 30% chance to move towards S_0 and getting a reward of -1 from the environment.



V*(s) is known as the optimal state value of any state s which was estimated by Bellman. He formed the Bellman Optimality Equation where V*(s) is the sum of all discounted future rewards the agent can expect on average after it reaches a state s if it acts optimally.

$$V^*(s) = max_a \left\{ \sum_{s`} T(s,a,s`)[R(s,a,s`) + \gamma . V^*(s`)] \right\} \;\; \forall s$$

- T(s,a,s’) is referred as the transition probability from state s to s’ when the chosen action is a.
- R(s,a,s’) is the reward which the agent gets due to choosing action a which causes a transition from state s to s’.
- γ is the discount rate.

This recursive equation states that the optimal value of the present state is equal to the reward it will get on average after taking one optimal action, plus the expected optimal value of all possible next states that this action can lead to, provided that the agent acts optimally.

This equation directly leads to an algorithm known as Value Iteration algorithm which can precisely estimate the optimal state value of each and every state. We first initialize all the state value estimates to zero and then update them using the above mentioned algorithm iteratively. These estimates always converge to the optimal state values depending on the optimal policy, given enough time.

Algorithm

Output: $\pi^*, the\ optimal\ policy$

```
1.  Initialize v(s) arbitrarily for all s ∈ S
2.  **repeat**
3.      Δ← 0
4.      **for** each s ∈ S do
5.          $v_{old} \leftarrow v(s)$
6.          $v(s) = max_a\{\sum_{s`} T(s,a,s`)[R(s,a,s`) + \gamma . v(s`)]\}$
7.          Δ← max{Δ, $|v_{old} - v(s)|$}
8.      **end for**
9.  **until** Δ<∈
10. **for** each s ∈ S do
11.     $\pi(s) \leftarrow argmax_{a\in A}\{\sum_{s`} T(s,a,s`)[R(s,a,s`) + \gamma . v(s`)]\}$
12. **end for**
13. return $\pi$
```

It is very useful to know the optimal state values especially in the case of policy evaluation, but it doesn’t explicitly tell the agent what to do. To counter this problem, Bellman also found another similar algorithm to estimate the optimum state-action values, called the Q-values. Q*(s,a) is known as the optimal Q-Value of the state-action pair which is the sum of discounted future rewards the agent can expect on average once it chooses the action a after reaching the state s. A difference is that the agent chooses the action without actually seeing its outcome given that it acts optimally after the action.

Algorithm

```
Initialize $V(s)$ to arbitrary values
Repeat
    For all $s \in S$
        For all $a \in \mathcal{A}$
            $Q(s,a) \leftarrow E[r|s,a] + \gamma \sum_{s' \in S} P(s'|s,a)V(s')$
        $V(s) \leftarrow \max_a Q(s,a)$
Until $V(s)$ converge
```

3.4 TEMPORAL DIFFERENCE LEARNING

As seen before, Reinforcement Learning problems with discretized actions can be modelled as Markov Decision processes. There is one problem with this process, the transition probabilities (T(s,a,s’)) are unknown in the beginning of the process. Also, the rewards (R(s,a,s’)) are unknown as well. Therefore, the model should experience each transition and state to know the rewards at least one time before the iterations. In order to estimate the transition probabilities well, it must experience these states and transitions multiple times.

Just like the Value Iteration algorithm, the Temporal Difference algorithm is also very similar except it takes care of the problems we mentioned above. Usually, we assume that the software agent only knows about the possible actions and states initially. Exploration policy is then used by the agent to explore the Markov Decision Process. Then, based on the rewards and transitions that are actually observed, the Temporal Difference Learning algorithm updates the estimates of the state values.

Algorithm

```
Input: the policy $\pi$ to be evaluated
Initialize $V(s)$ arbitrarily (e.g., $V(s) = 0, \forall s \in \mathcal{S}^+$)
Repeat (for each episode):
    Initialize $S$
    Repeat (for each step of episode):
        $A \leftarrow$ action given by $\pi$ for $S$
        Take action $A$; observe reward, $R$, and next state, $S'$
        $V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$
        $S \leftarrow S'$
    until $S$ is terminal
```

From the algorithm we can see that it keeps the running average of the immediate rewards which the agent receives when it leaves the state, plus the rewards it expects to get later.

3.4.1 Q-LEARNING

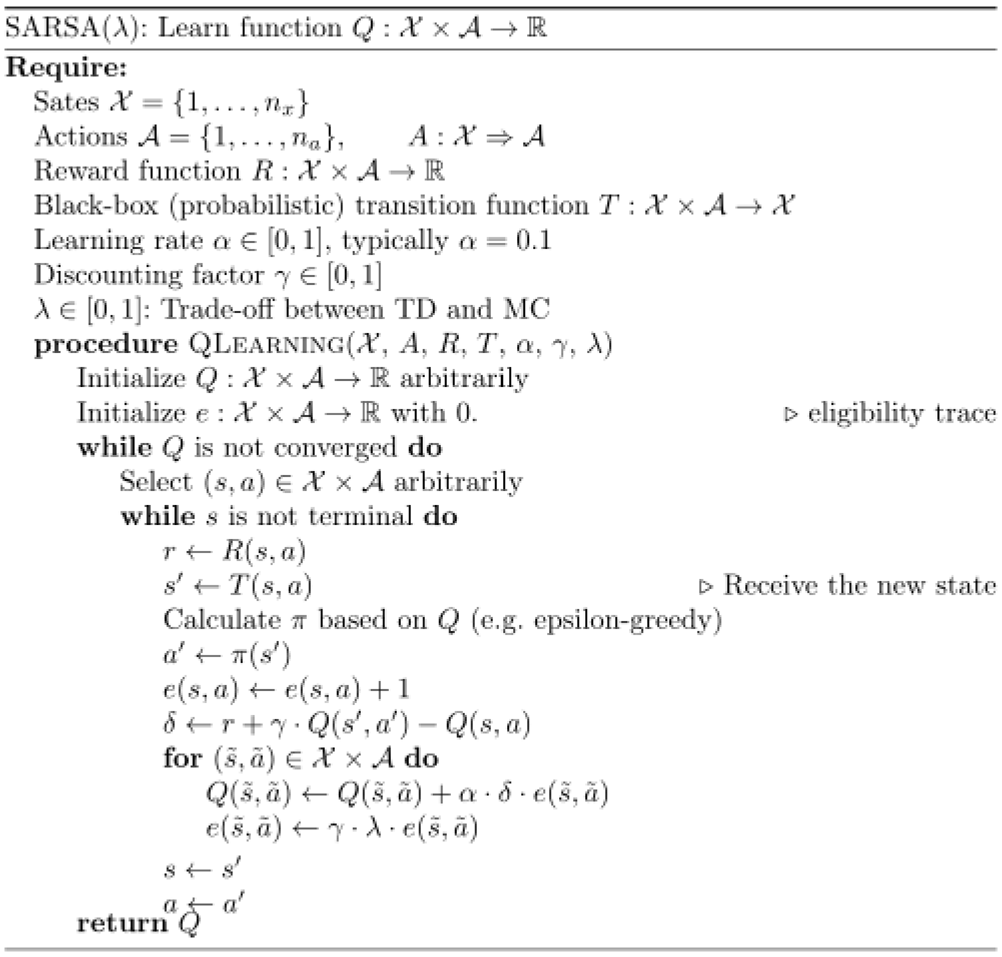
Similar to TD Learning algorithm, the Q-learning algorithm is a likewise adaptation of the Q-value iteration algorithm. Here, the rewards as well as the transition probabilities are initially not given.

```
$Q$-learning: Learn function $Q : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$
Require:
    Sates $\mathcal{X} = \{1, \ldots, n_x\}$
    Actions $\mathcal{A} = \{1, \ldots, n_a\}$,        $A : \mathcal{X} \Rightarrow \mathcal{A}$
    Reward function $R : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$
    Black-box (probabilistic) transition function $T : \mathcal{X} \times \mathcal{A} \to \mathcal{X}$
    Learning rate $\alpha \in [0, 1]$, typically $\alpha = 0.1$
    Discounting factor $\gamma \in [0, 1]$
    procedure QLEARNING($\mathcal{X}$, $A$, $R$, $T$, $\alpha$, $\gamma$)
        Initialize $Q : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$ arbitrarily
        while $Q$ is not converged do
            Start in state $s \in \mathcal{X}$
            while $s$ is not terminal do
                Calculate $\pi$ according to Q and exploration strategy (e.g. $\pi(x) \leftarrow$
    $\arg\max_a Q(x, a)$)
                $a \leftarrow \pi(s)$
                $r \leftarrow R(s, a)$                              ▷ Receive the reward
                $s' \leftarrow T(s, a)$                             ▷ Receive the new state
                $Q(s', a) \leftarrow (1 - \alpha) \cdot Q(s, a) + \alpha \cdot (r + \gamma \cdot \max_{a'} Q(s', a'))$
                $s \leftarrow s'$
        return $Q$
```

This algorithm keeps track of a running average of the rewards r for each state-action pair(s,a) which the agent gets by deploying action a and leaving the state s, plus the rewards it expects to get later. Maximum of the Q-Value estimates are taken for the next state, given that the target policy would act optimally. This algorithm will always converge to the optimal Q-values if ran for a long time. In this case, the policy being executed and the policy being trained are not the same. This is known as off-policy algorithm.

3.4.2 STATE-ACTION-REWARD-STATE-ACTION (SARSA)

SARSA is very similar to Q-learning. The main difference between Q-Learning and SARSA is that Q-learning is an off-policy algorithm whereas SARSA is an on-policy algorithm. It means that SARSA learns the Q-value based on the action detemined by the current policy rather than the greedy policy.



```
SARSA($\lambda$): Learn function $Q : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$
Require:
  Sates $\mathcal{X} = \{1, \ldots, n_x\}$
  Actions $\mathcal{A} = \{1, \ldots, n_a\}$,        $A : \mathcal{X} \Rightarrow \mathcal{A}$
  Reward function $R : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$
  Black-box (probabilistic) transition function $T : \mathcal{X} \times \mathcal{A} \to \mathcal{X}$
  Learning rate $\alpha \in [0, 1]$, typically $\alpha = 0.1$
  Discounting factor $\gamma \in [0, 1]$
  $\lambda \in [0, 1]$: Trade-off between TD and MC
  procedure QLearning($\mathcal{X}$, $A$, $R$, $T$, $\alpha$, $\gamma$, $\lambda$)
    Initialize $Q : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$ arbitrarily
    Initialize $e : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$ with 0.          ▷ eligibility trace
    while $Q$ is not converged do
      Select $(s, a) \in \mathcal{X} \times \mathcal{A}$ arbitrarily
      while $s$ is not terminal do
        $r \leftarrow R(s, a)$
        $s' \leftarrow T(s, a)$                                  ▷ Receive the new state
        Calculate $\pi$ based on $Q$ (e.g. epsilon-greedy)
        $a' \leftarrow \pi(s')$
        $e(s, a) \leftarrow e(s, a) + 1$
        $\delta \leftarrow r + \gamma \cdot Q(s', a') - Q(s, a)$
        for $(\tilde{s}, \tilde{a}) \in \mathcal{X} \times \mathcal{A}$ do
          $Q(\tilde{s}, \tilde{a}) \leftarrow Q(\tilde{s}, \tilde{a}) + \alpha \cdot \delta \cdot e(\tilde{s}, \tilde{a})$
          $e(\tilde{s}, \tilde{a}) \leftarrow \gamma \cdot \lambda \cdot e(\tilde{s}, \tilde{a})$
        $s \leftarrow s'$
        $a \leftarrow a'$
    return $Q$
```

From the above algorithm, we can see that the two action selections always follow the current policy unlike the Q-learning algorithm in which the algorithm has no constraint over the next action, as long as the next state Q-value is maximized.

3.4.3 EXPLORATION POLICIES

Exploration policies are very important since the Q-learning models depends a lot on the exploration of the Markov Decision Process. We can use a random policy as our exploration policy since it is guaranteed to visit all the transitions and states many times.

But, there is a serious problem with this policy and that is its complexity. Therefore, we use a different option which is known as the Ɛ-greedy policy in which the policy acts randomly with a probability Ɛ or chooses the action with the highest Q-value (greedily) with the probability 1- Ɛ. The main advantage of Ɛ-greedy policy is the fact that it spends more time exploring the useful parts of the MDP while still exploring some unknown parts of the environment as the Q-estimates improve.

Another approach which can be used as exploration policy is to increase the chances of trying actions it hasn't tried much instead of relying on chance for exploration. This can be implemented as a small tweak to the Q-Value estimates.

$$Q(s`, a`) \leftarrow (1-\alpha).Q(s,a) + \alpha.(r + \gamma.max_{a`}\{f(Q(s`,a`), N(s`,a))\})$$

- N(s',a') counts the number of times state s' chose action a'.
- f(q,n) is an exploration function.

CHAPTER 4

DEEP REINFORCEMENT LEARNING

4.1 INTRODUCTION

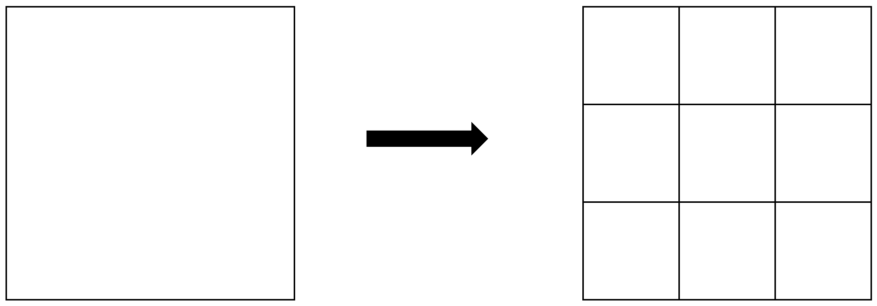
Q-Learning is an extremely powerful algorithm that provides a robust solution to Reinforcement Learning problems modeled as MDPs. However, there is a major drawback to this approach. In Q Learning, we maintain a Q-value table that maps <state, action> pairs to real valued probable rewards.

$$Q: S \times A \rightarrow R$$

These values are stores in a Q-table and iteratively tuned during the training phase. This very much resembles dynamic programming. The space complexity of the algorithm increases in direct proportion to the state space of the environment. Hence, for situations where the state space or the action space is very large, Q-Learning becomes quite difficult to implement in practice.

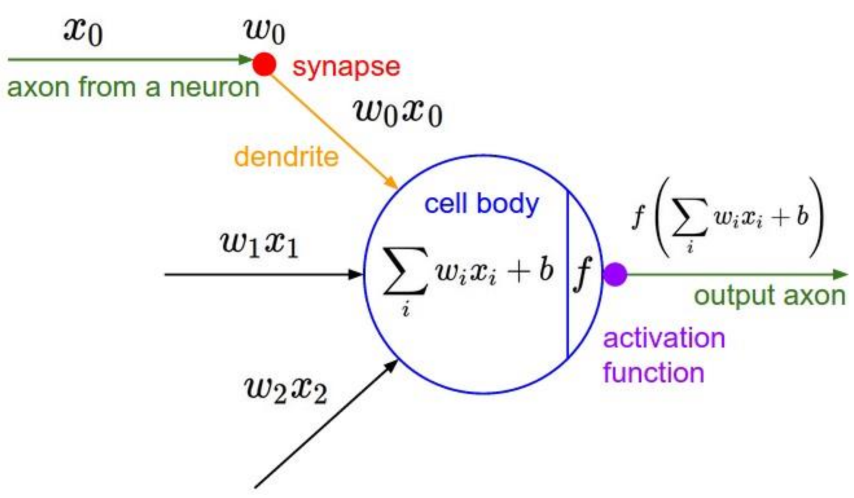
Also, since the agent makes decision after referring to the Q-table, for states which are not present in the Q-table, the agent will have no clue what to do. The agent will make no assumptions and no approximations. Scenarios where the state space is not discrete are not uncommon in practical world setup. For example, most robotic applications of reinforcement learning possess continuous state spaces defined by the means of continuous variables such as position, angular displacement, velocity, torque, etc.

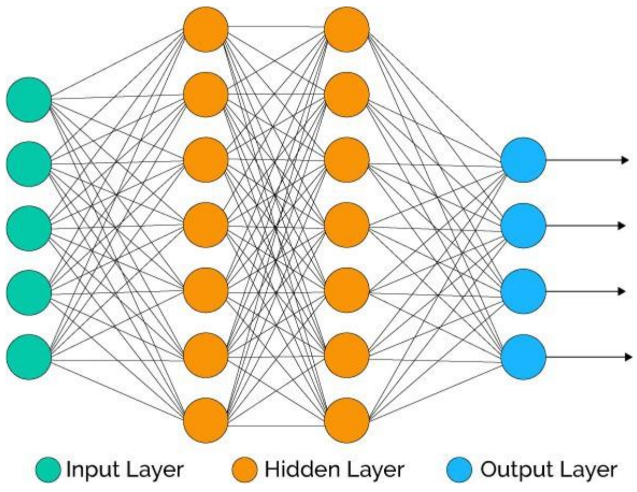
The usual approach has been to discretize the continuous variables, which quickly leads to combinatorial explosion and the well-known “Curse of Dimensionality”.



The above figure shows the discretization of a continuous state space. It has been observed that Q-learning with discretised states and actions scale poorly. As the number of state and action variables increase, the size of the table used to store Q-values grows exponentially. Accurate control requires that variables be quantised finely, but as these systems fail to generalise between similar states and actions, they require large quantities of training data. We can avoid such problems that arise due to the discretization of state space by using methods that can directly deal with continuous states.

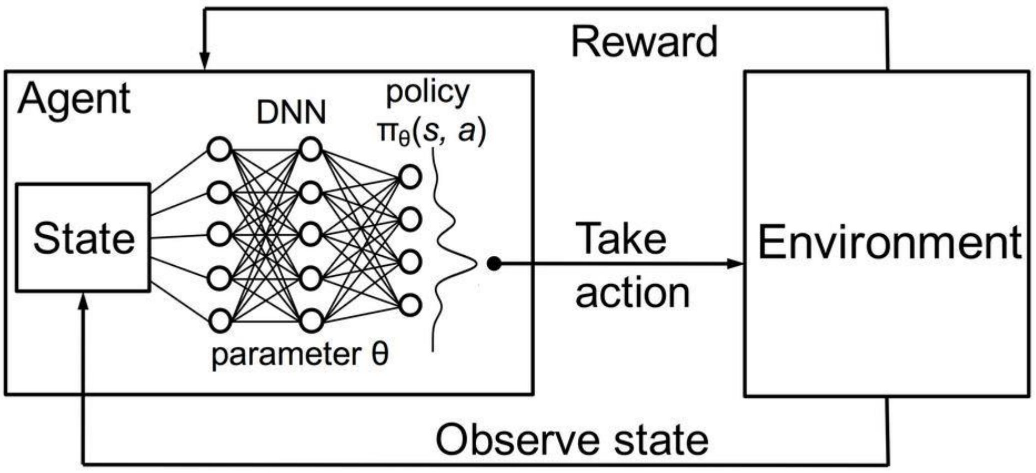
4.2 DEEP REINFORCEMENT LEARNING



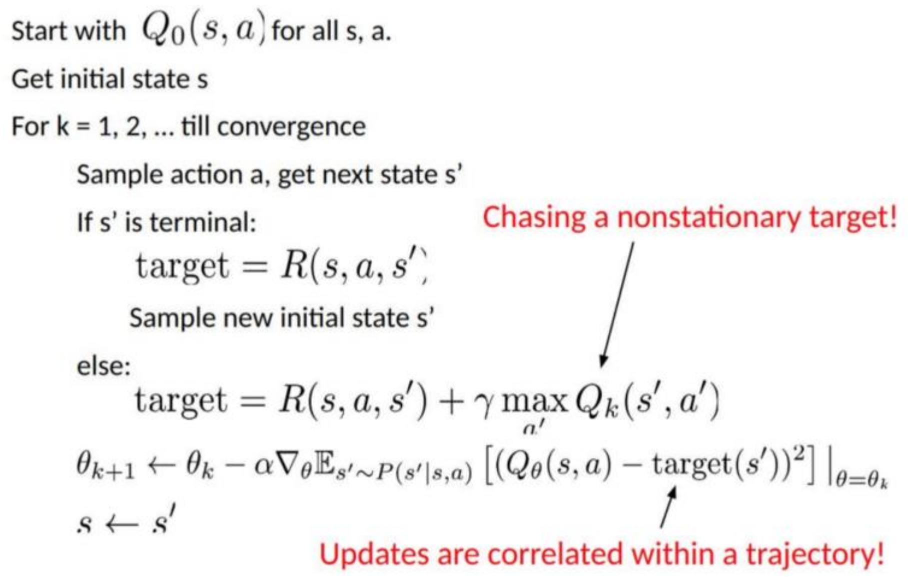


One of the most popular and effective approach to overcome this shortcoming has been to use neural networks to approximate the Q-values of <state, action> pairs for continuous state space. This is strongly supported by the Universal Approximation Theorem. In the mathematical theory of artificial neural networks, the universal approximation theorem states that under mild assumptions on the activation function, a neural network containing a finite number of neurons can approximate continuous functions on compact subsets of R^n.

A neural network architecture is designed to estimate the Q-values in case of continuous state space. The input to the network is the current state of the agent and the output is an n-dimensional vector where each dimension represents the Q-value of that particular action; given there are 'n' possible actions. The algorithm is called Deep Q-Network (DQN).[6]



Algorithm



```
Start with $Q_0(s, a)$ for all s, a.
Get initial state s
For k = 1, 2, ... till convergence
    Sample action a, get next state s'
    If s' is terminal:
        $\text{target} = R(s, a, s')$
        Sample new initial state s'
    else:
        $\text{target} = R(s, a, s') + \gamma \max_{a'} Q_k(s', a')$
    $\theta_{k+1} \leftarrow \theta_k - \alpha \nabla_\theta \mathbb{E}_{s' \sim P(s'|s,a)} \left[ (Q_\theta(s, a) - \text{target}(s'))^2 \right] \big|_{\theta=\theta_k}$
    $s \leftarrow s'$
```

Chasing a nonstationary target!

Updates are correlated within a trajectory!

4.2.1 Problems with DQN

Using target values, the above equations seem to be similar to those encountered in supervised learning. However, in supervised learning we want the input data to be I.I.D (Independently and Identically Distributed), i.e.

- Samples should be randomized among batches and each batch should have similar data distribution.
- Samples should be independent of each other in the same batch.

If not, the model may overfit for some class of samples at different time and the solution will not be generalized. In addition, for the same input, its label should not change over time. These stability conditions are required for the supervised learning algorithms to work well.

In reinforcement learning, both the input and the target change constantly during the process and hence make training unstable.

- **Target unstable**: We are using the same parameters (weights) for estimating the target **and** the Q value. As a consequence, there is a big correlation between the target and the parameters that are being changed. Therefore, at every step of training, **our Q values change but the target values also change.** So, we're getting closer to our target but the target is also moving. It's like chasing a moving target! This leads to amplified oscillations in training.

- **Non I.I.D.:** Another problem related to the correlations within a trajectory is that within a training iteration, we update model parameters to move *Q(s, a)* closer to the ground truth. These updates will impact other estimations. When we pull up the Q-values in the deep network, the Q-values in the surrounding states will be pulled up also like a net. Let's say we just score a reward and adjust the *Q*-network to reflect it. Next, we make another move. The new state will look similar to the last one in particular since that environment does not change drastically and hence the states are correlated. The newly estimated *Q (s', a')* will be higher and our new target for *Q* will be higher, regardless of the merit of the new action taken. If we update the network with a sequence of actions in the same trajectory, it magnifies the updates greatly. This destabilizes the learning process.

In Reinforcement Learning, we often depend on the value functions or the policy to sample actions. However, this is frequently changing as we get to know better what to explore. As we play out the game, we know better about the ground truth values of actions and states. So, the target outputs are changing also. Hence, we are trying to learn a mapping *f* for a constantly changing input and output!

Luckily, both input and output can converge if we slow down the changes in both the input and output enough, we may have a chance to model *f* while allowing it to evolve.

4.2.2 Solutions

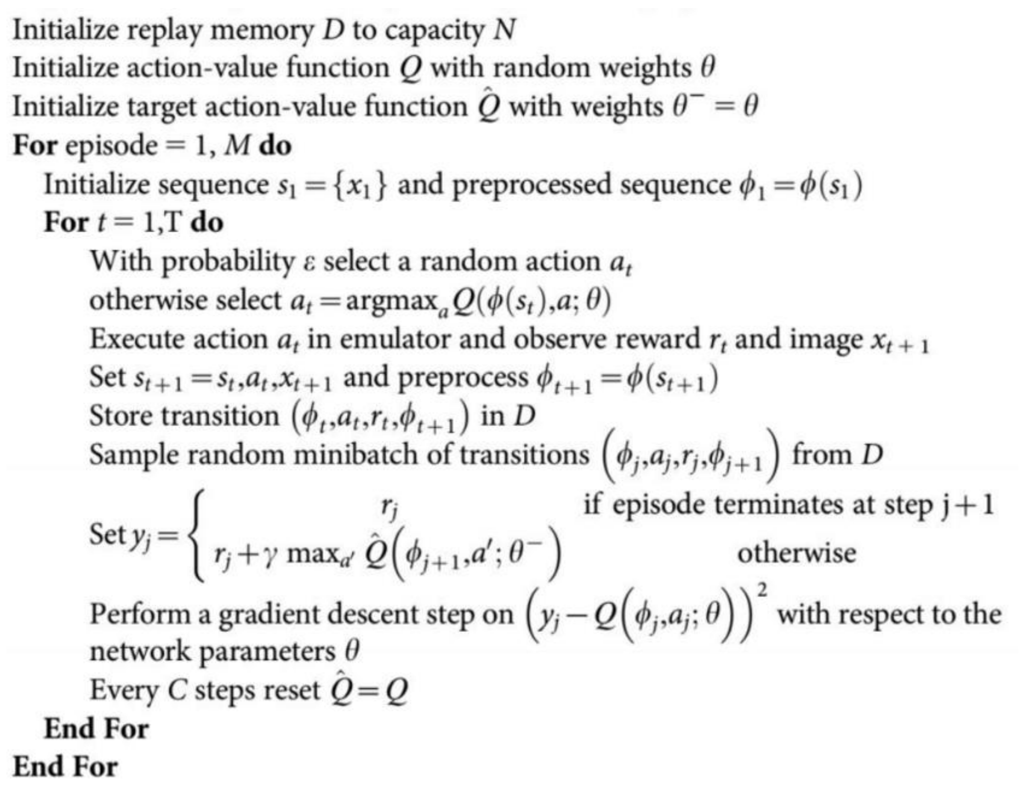
- **Experience replay**: We put the past few transitions into a buffer and sample a mini-batch of samples from this buffer to train the deep network. This forms an input dataset which is stable enough for training. As we randomly sample from the replay buffer, the data is more independent of each other and closer to i.i.d.

- **Target network**: We create two deep networks θ^{-} and θ. The first one is used to retrieve the Q values while the all the updates during the training process are made in the second network. After a fixed number of updates, we synchronize θ^{-} and θ. The objective is to fix the Q-value targets temporarily so that we don't have to chase a moving target. In addition, parameter changes do not impact θ^{-} immediately and therefore even the input may not be 100% i.i.d., it will not incorrectly magnify its effect as mentioned before.

$$L_i(\theta_i) = \mathbb{E}_{s,a,s',r\sim D}\left(\underbrace{r + \gamma \max_{a'} Q(s', a'; \theta_i^{-})}_{\text{target}} - Q(s, a; \theta_i)\right)^2$$

With both experience replay and the target network, we have a more stable input and output to train the network and behaves more like supervised training.

Algorithm

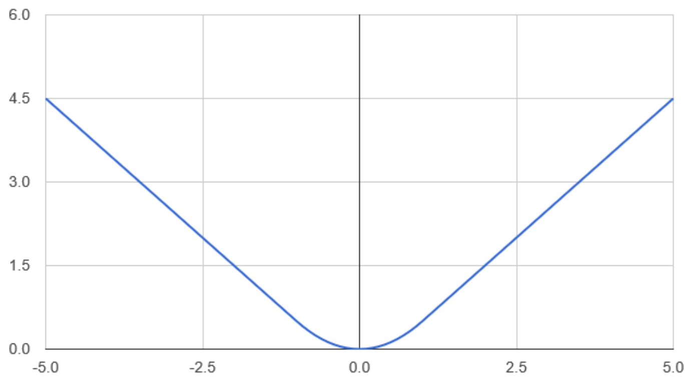


```
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
**For** episode = 1, $M$ **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  **For** $t = 1,T$ **do**
    With probability $\varepsilon$ select a random action $a_t$
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the
    network parameters $\theta$
    Every $C$ steps reset $\hat{Q} = Q$
  **End For**
**End For**
```

4.2.3 Implementation Details

- **Loss Function**

  DQN uses Huber Loss function, where the loss is quadratic for small values of *x*, and linear for large values.



$$L_\delta(a) = \begin{cases} \frac{1}{2}a^2 & \text{for } |a| \le \delta, \\ \delta(|a| - \frac{1}{2}\delta), & \text{otherwise.} \end{cases}$$

- **Optimization**

  Training of reinforcement learning agents is sensitive to optimization methods used. To handle the changes in the input during the training, a simple learning rate schedule does not suffice because it is not dynamic enough. Therefore, RMSProp or Adam optimizers are often used while training a Deep Q-Network.

4.3 DOUBLE DQN

In certain stochastic environments, the well-known Q-learning reinforcement learning algorithm performs somewhat poorly. This poor performance is caused by large overestimations of action values. These overestimations are resulted from a positive bias that is introduced into the system because Q-learning uses the maximum action value as an approximation for the maximum expected action value (Q-value).

Double Deep Q-Networks, or double Learning, was introduced by Hado van Hasselt, which **handles the problem of the overestimation of Q-values.** This is how we used to calculate the Temporal Difference (TD) Target -

$$\underbrace{Q(s,a)}_{\text{Q target}} = \underbrace{r(s,a)}_{\text{Reward of taking that action at that state}} + \underbrace{\gamma max_a Q(s',a)}_{\text{Discounted max q value among all. possibles actions from next state.}}$$

However, we cannot be sure whether the best action for the next state is the action with the highest Q-value. The accuracy of Q-values depends on what actions were tried and what neighbouring states were explored.

As a consequence, at the beginning of the training we don't have enough information with us to know about the best action to take. Therefore, taking the maximum Q-value (which is noisy due to random initializations) as the best action to take may lead to false positives. If non-optimal actions are regularly given a higher Q-value than the optimal best action, the learning will be complicated.[7]

The solution is to use two networks to decouple the action selection from the target Q-value generation when we compute the Q target.

- Use DQN network to select what is the best action to take for the next state (the action with the highest Q value).
- Use target network to calculate the target Q value of taking that action at the next state.

$$\underbrace{Q(s,a)}_{\text{TD target}} = r(s,a) + \gamma \underbrace{Q(s', \underbrace{argmax_a Q(s',a)}_{\text{DQN Network choose action for next state}})}_{\text{Target network calculates the Q value of taking that action at that state}}$$

Therefore, Double DQN helps us reduce the overestimation of Q-values and, as a consequence, helps us train faster and have more stable learning process.

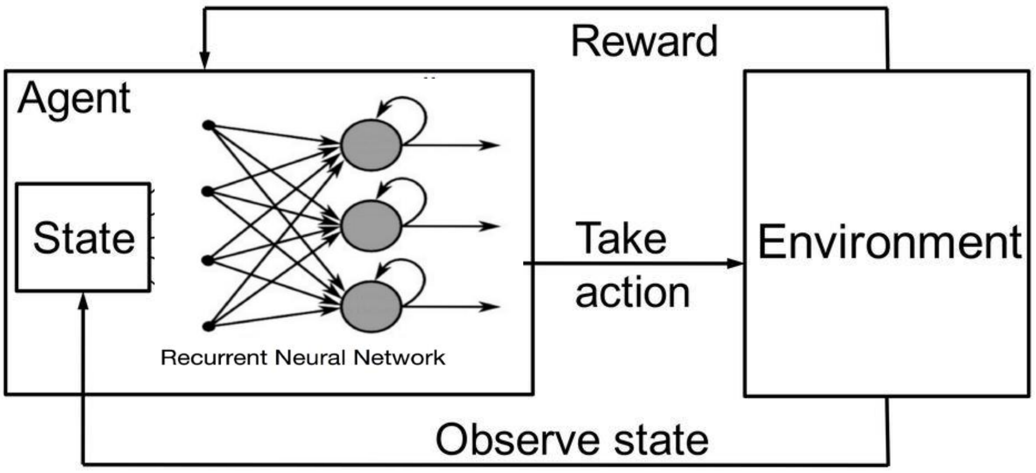
Algorithm

```
Initialize $Q^A$,$Q^B$,$s$
repeat
    Choose $a$, based on $Q^A(s,\cdot)$ and $Q^B(s,\cdot)$, observe $r$, $s'$
    Choose (e.g. random) either UPDATE(A) or UPDATE(B)
    if UPDATE(A) then
        Define $a^* = \arg\max_a Q^A(s',a)$
        $Q^A(s,a) \leftarrow Q^A(s,a) + \alpha(s,a)\left(r + \gamma Q^B(s',a^*) - Q^A(s,a)\right)$
    else if UPDATE(B) then
        Define $b^* = \arg\max_a Q^B(s',a)$
        $Q^B(s,a) \leftarrow Q^B(s,a) + \alpha(s,a)(r + \gamma Q^A(s',b^*) - Q^B(s,a))$
    end if
    $s \leftarrow s'$
until end
```

4.4 DEEP RECURRENT Q-NETWORKS

Deep Q-Networks are limited in the sense that they learn a mapping from a limited number of past states. Thus they will be unable to master the processes that require them to remember even more distant events in the past. Such processes do not only depend on the current state of the agent. Hence, instead of a Markov Decision Process (MDP), the process becomes Partially Observable Markov Decision Process (POMDP). Many real world tasks often feature such incomplete and noisy state information resulting from partial observability of the states.

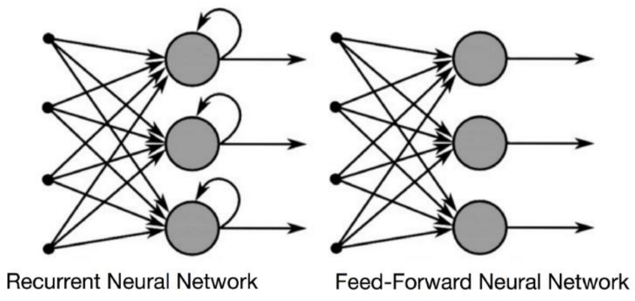
Deep Q-Networks' performance declines when it is made to estimate Q-functions from partially observable states, due to the lack of memory. However, the existing DQN can be updated and modified to better deal with POMDP by leveraging the power of Recurrent Neural Networks (RNN).



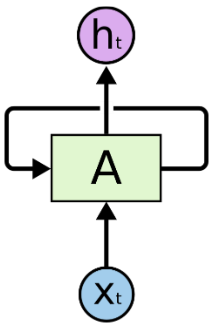
4.4.1 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are an improvement over the existing neural network architecture that possess the ability to capture the past trends in the input. This property is of utmost importance when the data that we have is sequential in format. Some problems that require such an architecture are image captioning, financial data modelling, named entity recognition, etc. Famous applications include Apple's digital personal voice assistant Siri, Google Translate, etc.

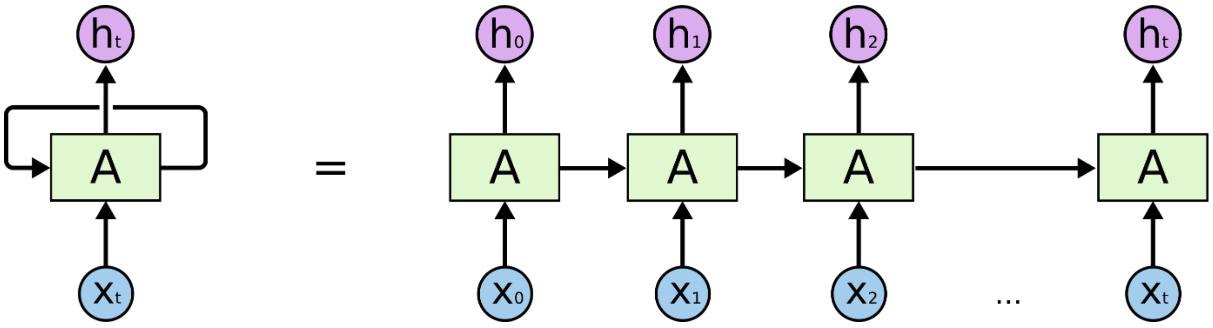
RNNs are a very powerful class of algorithms that are built on top of the existing neural networks by introducing the concept of network 'memory', that remembers the past states as well.



The information cycles through a loop in an RNN. It takes into consideration the inputs that it has received what it has received previously, along with the current input when it makes a decision.



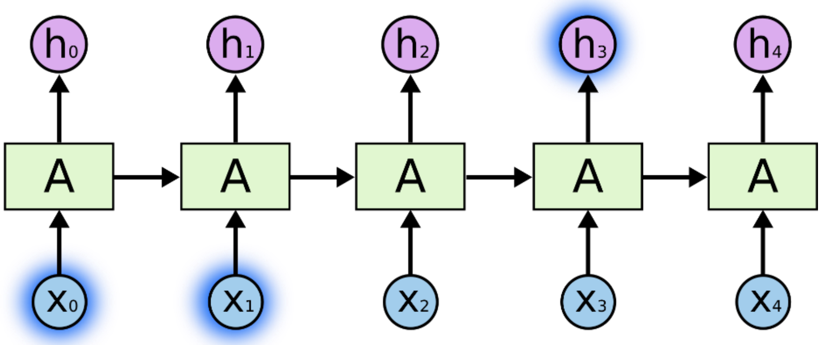
In the above diagram, a collection of neural networks, A, takes some input x_t and outputs the value h_t. A loop allows information to be passed from one step of the network to the next. The loops may make recurrent neural networks seem sort of strange and mysterious. However, on careful observation, it can be seen that they aren't much different from a normal neural network. A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor over time. Consider what happens if we unroll the loop of the RNN:



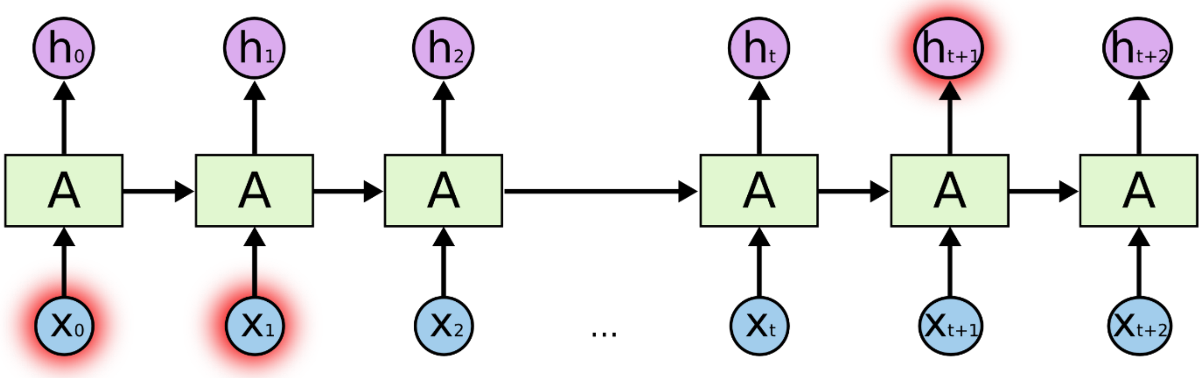
This appears to be a stacked layers of various neural networks, with the exception that the layers are stacked across time, not across space. The values h_t are the hidden units that represent the memory of the recurrent neural network. This allows the network to capture and extract information from historical input data and use it further down the time.

4.4.2 Long Short Term Memory (LSTM)

One of the most appealing ideas of RNNs is that they might be able to connect previous information to the present task, such as utilizing the previous stock movement ticks might inform the understanding of the scenario. In cases where the relative time gap between the relevant information and the place that it's required is small, RNNs can learn to use the past information.



Unfortunately, as that gap grows, RNNs become unable to learn to use the past information and connect it with the present scenario. In theory, RNNs are proven to be capable of handling such *"long-term dependencies."* Parameters could be carefully chosen for them to solve problems of this form. Sadly, it has been observed that in practice, RNNs don't seem to be able to learn them. This was explored in depth by Hochreiter (1991) and Bengio, et al. (1994), who found some pretty fundamental reasons why it might be difficult.

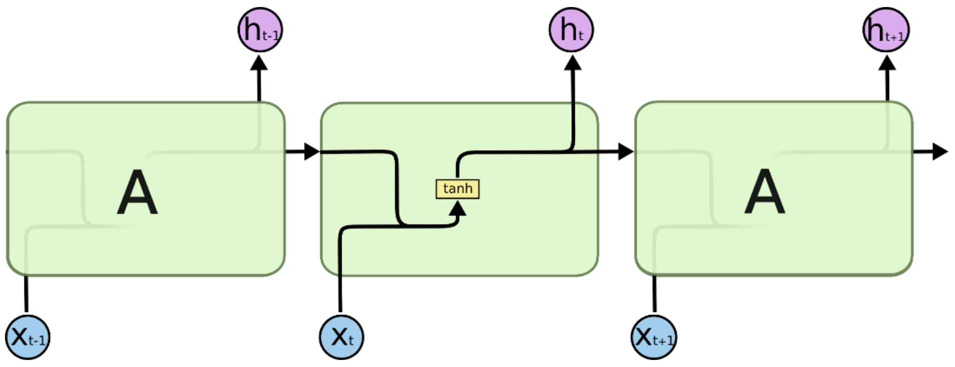


Long Short Term Memory (LSTM) networks, are used to solve this problem.

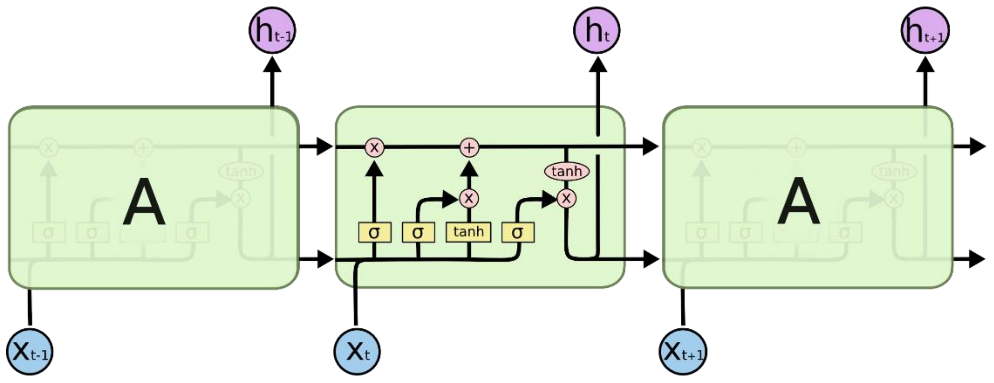
“LSTMs” are a special kind of RNN which are capable of learning such long-term dependencies. They were first introduced by Hochreiter & Schmidhuber in 1997, and were popularized and refined by many people over time. They work extremely well on a large variety of problems, and are now widely used.

LSTMs are explicitly designed to avoid the long-term dependency problem. It’s their default behaviour to remember information over long periods of time, not something they have to struggle to learn!

All recurrent neural networks have the structure of a chain of repeating neural network modules over time. In standard RNNs, this repeating module has a very simple structure, such as a single tanh layer.



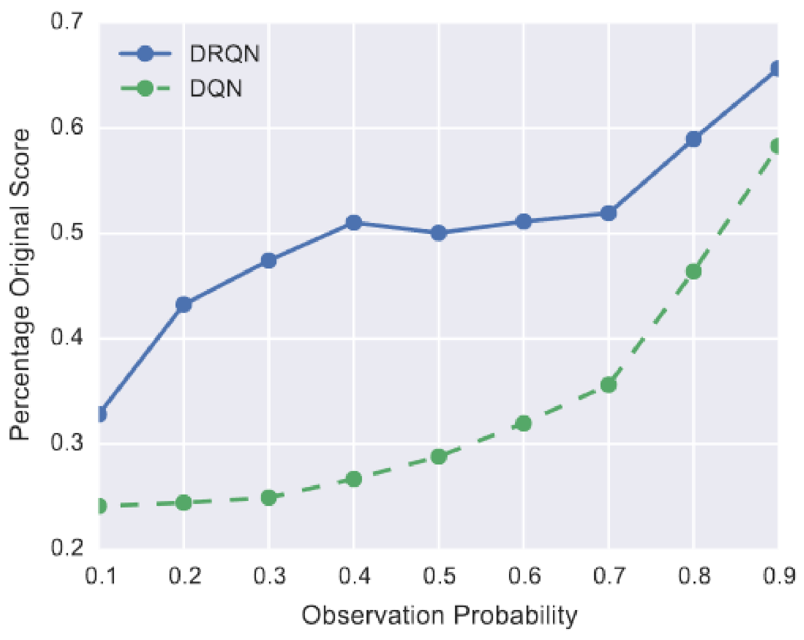
LSTMs also have this chain like structure, but the repeating module has a different structure. Instead of having a single neural network layer, there are four, interacting in a very special way.



4.4.3 Deep Recurrent Q-Networks (DRQN)

We modify the existing DQN network and append an LSTM layer in the end of the dense layers to capture the long term dependencies.

A recurrent Q-network could also be trained on a standard Markov Decision Process environment and then generalized to a Partially Observable Markov Decision Process at runtime. Matthew Hausknecht and Peter Stone showed in 2017 that it can indeed be done and the model developed by incorporating RNN in Deep Q-Learning algorithm.[8]



CHAPTER 5

EXPERIMENTAL IMPLEMENTATION

5.3 AGENT

We seek to completely exploit the environment with the help of action augmentation. Random exploration is deemed unsatisfactory, especially in the financial trading domain since transaction costs occur with a change of position. Hence, we shall use a simple technique to mitigate the need for random exploration by providing the agent with reward signals for every action. This is easily possible since the reward is easily computable after the price of the current timestamp is revealed.

For example, if the unrealized PnL for the current step is +10 after we execute action +1, then we immediate know that if we were to execute action -1, we would get a reward of -10 and 0 for action 0. Therefore the portfolio value v_t can be computed (therefore the reward signal) for all actions.

On the other hand, the only part of the state that would be altered if we were to take other actions is the agent's position. This is known as the zero market impact hypothesis which states that the action taken from the market participator has no influence on the current market condition. We also assume order issued by the agent always executes at the next opening price. That is, we always know the position for the next step if the output action is determined.

Now we can update Q-values for all actions. The loss function is defined by –

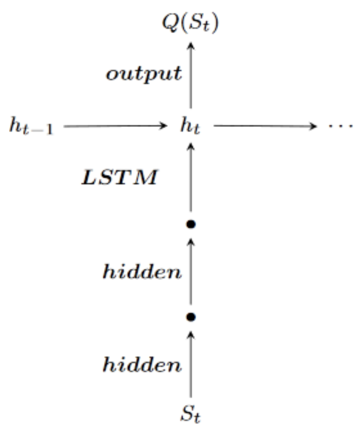
$$L(\theta) = E_{(s,a,r,s`)\sim D}[|| \, r + \gamma . Q_{\theta^-}(s`, argmax_{a`}\{Q_\theta(s`, a`)\}) - Q_\theta(s,a) \, ||^2]$$

Where Q_{θ^-} denotes the target network.

5.4 MODEL ARCHITECTURE

5.4.1 Weight/Parameter Initialisation

Proper weight initialization is extremely critical for successful training of deep neural networks. The initialization scheme presented in He et al. (2015) for weight matrices in both hidden layers and input-to-hidden layer in LSTM is followed in our approach. We follow Le et al. (2015) to initialize all hidden-to- hidden weight matrices to be identity. All biases in the network are set to be zero except for the forget gate in the LSTM which are set to be 1. We sparsely initialize the output layer weight matrix with Gaussian distribution N (0, 0.001).



5.4.2 Training Scheme

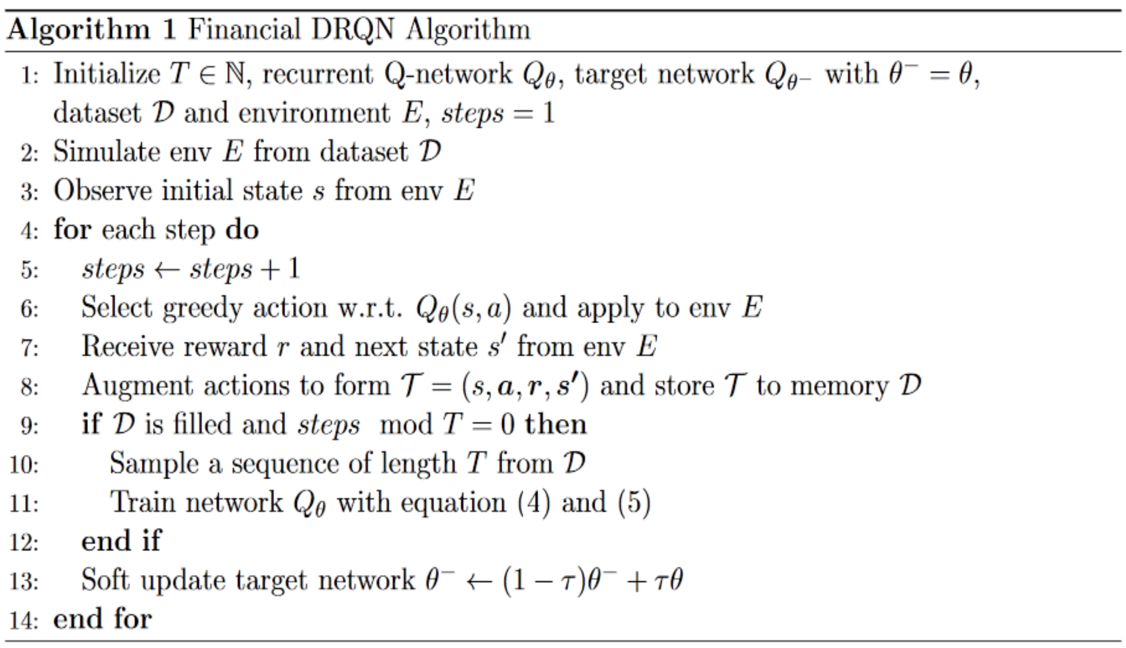
1. A smaller replay memory is used as it is proven to be more effective in DRQN for financial domain. The reason is that in financial trading, recent data points are more important than the points which are further in the past.

2. A longer sequence is sampled from the replay memory than the number of steps used in the DRQN paper. The reason behind this modification is that, a successful trading strategy involves opening a position at the right time and holding the position for a sufficiently long period of time then exiting the position. Sampling a short sequence can’t effectively train the network to learn the desired long-term dependency.

3. It is unnecessary to train the network for each step since we are sampling a longer sequence. Hence we train the network for every T time steps only. This significantly reduces computation since the number of backward passes are reduced by a factor of T.

5.4.3 Complete Learning Algorithm

Adopting the above learning scheme, the following algorithm is finally used for the training of the Deep Recurrent Q-Network. The algorithm has been named as “Financial Deep Recurrent Q-Network (FDRQN)”. The code for the training is written in Python. The algorithm is optimised to capture the nuances of Financial Trading.



Algorithm 1 Financial DRQN Algorithm

```
1:  Initialize $T \in \mathbb{N}$, recurrent Q-network $Q_\theta$, target network $Q_{\theta^-}$ with $\theta^- = \theta$,
    dataset $\mathcal{D}$ and environment $E$, $steps = 1$
2:  Simulate env $E$ from dataset $\mathcal{D}$
3:  Observe initial state $s$ from env $E$
4:  for each step do
5:      $steps \leftarrow steps + 1$
6:      Select greedy action w.r.t. $Q_\theta(s, a)$ and apply to env $E$
7:      Receive reward $r$ and next state $s'$ from env $E$
8:      Augment actions to form $\mathcal{T} = (s, \boldsymbol{a}, \boldsymbol{r}, \boldsymbol{s'})$ and store $\mathcal{T}$ to memory $\mathcal{D}$
9:      if $\mathcal{D}$ is filled and $steps \mod T = 0$ then
10:         Sample a sequence of length $T$ from $\mathcal{D}$
11:         Train network $Q_\theta$ with equation (4) and (5)
12:     end if
13:     Soft update target network $\theta^- \leftarrow (1 - \tau)\theta^- + \tau\theta$
14: end for
```

5.4.4 Hyper-parameters

Hyper-parameters	**Value**
Learning time-step T	96
Replay memory size N	480
Learning rate	0.00025
Optimizer	Adam
Discount factor γ	0.99
Target network τ	0.001

APPENDIX-A

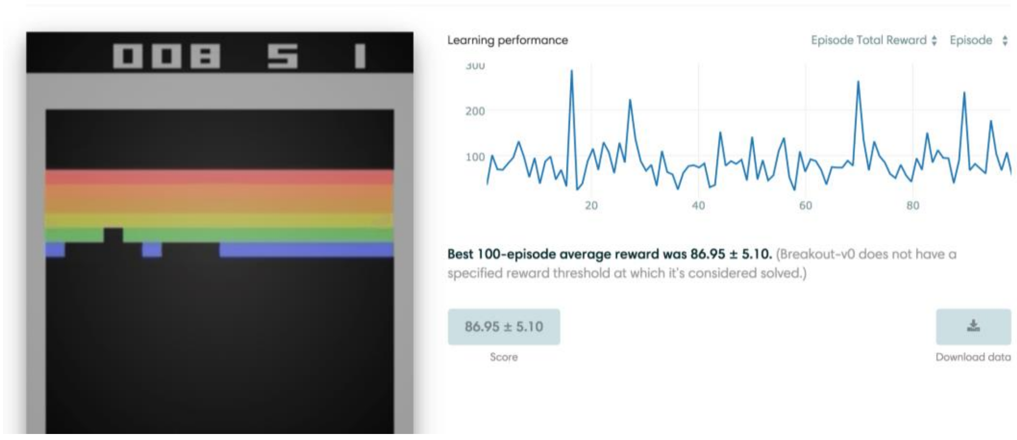
1. OPEN AI GYM

OpenAI is a non-profit artificial intelligence research company that seeks to develop and develop friendly AI in such a way as to benefit humanity as a whole. The company was founded in late 2015 in San Francisco by Elon Musk and Sam Altman. The organization aims to “freely collaborate” with other researchers and organisations by making its research and open to the people.

The most important product released by OpenAI is the OpenAI Gym (or simply Gym). Gym strives to provide a general-intelligence benchmark with a wide variety of different environments - somewhat similar to, but broader than, the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) used in supervised learning research – hoping to standardize the way in which environments are defined in AI research publications, so that published research becomes more easily reproducible. The environment is fairly easy to set-up and the user is provided with a simple interface. Currently, Gym can only be used with Python.

Gym comes with a diverse suite of environments that range from easy to difficult and involve many different kinds of data -

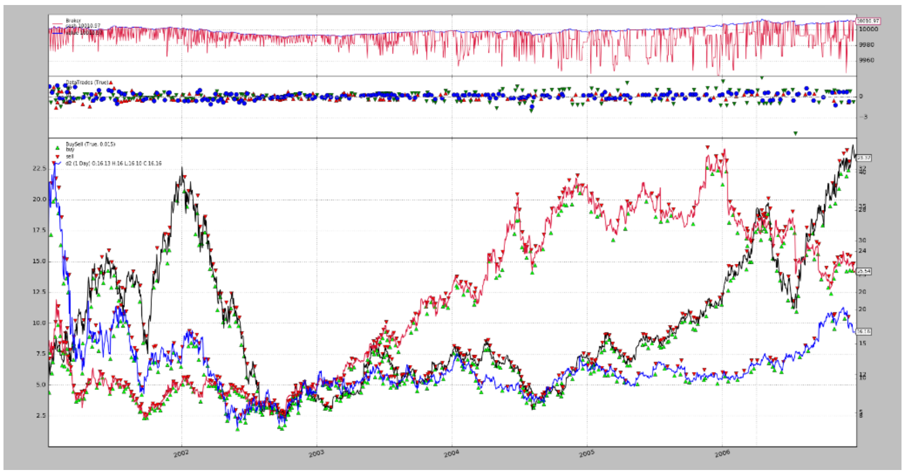
- Classic control and toy text: complete small-scale tasks, mostly from the RL literature.
- Algorithmic: perform computations such as adding multi-digit numbers and reversing sequences.
- Atari: play classic Atari games.
- 2D and 3D robots: control a robot in simulation.



Best 100-episode average reward was 86.95 ± 5.10. (Breakout-v0 does not have a specified reward threshold at which it's considered solved.)

2. BACKTRADER

Backtrader is a feature-rich Python framework for back-testing and trading. Backtrader allows the user to focus on writing reusable trading strategies, indicators and analysers instead of having to spend time building infrastructure. The platform has two main features – ease of use and reproducibility. The user can create a stock trading strategy by deciding the potential adjustable parameters, instantiating the indicators required in the strategy and then writing down the logic for entering/exiting the market.



3. BT GYM

BTGym is an OpenAI Gym-compatible environment for Backtrader back-testing/trading library, designed to provide gym-integrated framework for running reinforcement learning experiments in [close to] real world algorithmic trading environments.

Market Position: Flat
Unrealized P&L: $0.00
Realized P&L Today: $675.00
P&L from 4/16/2017: $675.00

REFERRENCES

[1]. Chien Yi Huang , “Financial Trading as a Game: A Deep Reinforcement Learning Approach”, arXiv:1807.02787[q-finTR], 8 July 2018

[2]. C. T. Chen, A. Chen and S. Huang, "Cloning Strategies from Trading Records using Agent-based Reinforcement Learning Algorithm," *2018 IEEE International Conference on Agents (ICA)* , Singapore, 2018, pp. 34-37.

[3]. Y. Deng, F. Bao, Y. Kong, Z. Ren and Q. Dai, "Deep Direct Reinforcement Learning for Financial Signal Representation and Trading," in *IEEE Transactions on Neural Networks and Learning Systems* , vol. 28, no. 3, pp. 653-664, March 2017.

[4]. Z. Jiang and J. Liang, "Cryptocurrency portfolio management with deep reinforcement learning," *2017 Intelligent Systems Conference (IntelliSys)* , London, 2017, pp. 905-913.

[5]. Jae Won Lee, "Stock price prediction using reinforcement learning," *ISIE 2001. 2001 IEEE International Symposium on Industrial Electronics Proceedings (Cat. No.01TH8570)* , Pusan, South Korea, 2001, pp. 690-695 vol.1.

[6]. K. Kashihara, "Deep Q learning for traffic simulation in autonomous driving at a highway junction," *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)* , Banff, AB, 2017, pp. 984-988.

[7]. Y. Zhang, P. Sun, Y. Yin, L. Lin and X. Wang, "Human-like Autonomous Vehicle Speed Control by Deep Reinforcement Learning with Double Q-Learning," *2018 IEEE Intelligent Vehicles Symposium (IV)* , Changshu, 2018, pp. 1251-1256.

[8]. J. Zeng, J. Hu and Y. Zhang, "Adaptive Traffic Signal Control with Deep Recurrent Q-learning," *2018 IEEE Intelligent Vehicles Symposium (IV)* , Changshu, 2018, pp. 1215-1220.