DM-Gym: A Reinforcement Learning Python Library for Data Mining Problems

A Thesis Presented
by

Ashwin Devanga

to

The Department of Mechanical and Industrial Engineering

in partial fulfillment of the requirements
for the degree of

Master of Science

in

Data Analytics Engineering

Northeastern University
Boston, Massachusetts

August 2022

To my family.

Contents

List of Figures

List of Tables

List of Acronyms

A2C Advantage Actor-Critic

A3C Asynchronous Advantage Actor-Critic

AI Artificial Intelligence

DBSCAN Density-Based Spatial Clustering of Applications with Noise

Density-based non-parametric clustering algorithm.

DDQN Double Deep Q Networks

DM Data Mining

DM-Gym Data Mining Gym

DQN Deep Q Networks

KNN K Nearest neighbors

K Nearest neighbors is a non-parametric classification and prediction algorithm.

MDP Markov Decision Process

This can stand for either Markov Decision Process or Markov Decision Problem based on context.

ML Machine Learning

RL Reinforcement Learning

SAC Soft Actor-Critic

SOM Self Organising Maps

Neural Network architecture to solve clustering problems.

Acknowledgments

Here I wish to thank those who have supported me during the process of the thesis work. I want to thank Prof. Mohammad Dehghani for his support and belief in me throughout the project. I would like to thank Sahil Belsare, with whom I spent multiple hours conversing about the issues faced in this endeavor. I want to thank Northeastern University, notably the College of Engineering, for allowing me to conduct this research. Finally, I would like to thank my family, without whose moral support, none of this would have been possible.

Abstract of the Thesis

DM-Gym: A Reinforcement Learning Python Library for Data Mining Problems

by

Ashwin Devanga

Master of Science in Data Analytics Engineering

Northeastern University, August 2022

Mohammad Dehghani, Adviser

Reinforcement Learning (RL) is a comparatively new machine learning method based on decision making through experience. Since natural learning always occurs through experience, it is not difficult to understand that RL is the next step toward Turing-Complete machines. Every problem can be defined as a decision-making problem if structured properly. Hence, we believe that RL can solve any problem. This project shows that classical data-mining problems such as classification, clustering, and regression can be solved using RL. Data Mining is the practical aspect of Machine Learning which focuses on problems that require an understanding of data to make predictions.

Chapter 1

Introduction

”I predict that, because of artificial intelligence and its ability to automate certain tasks that in the past were impossible to automate, not only will we have a much wealthier civilization, but the quality of work will go up very significantly and a higher fraction of people will have callings and careers relative to today.” - Jeff Bezos, Amazon.com LLC (Founder and executive chairman), Blue Origin LLC (Founder)

Reinforcement Learning (RL) is a concept of sequential decision-making to obtain maximum rewards. It is a branch of Machine Learning (ML) that is relatively new and is based on the concept of continuous learning. It has the ability to learn from mistakes and correct them in the future. The algorithms have shown that with experience, they can outperform humans in the task they are trained to do with ease. Such an example was showcased famously by AlphaGo [25], an RL agent trained in the game of Go that beat the best human player on the planet.

However, this branch of ML has not received attention in industrial use-cases. The reason for this is designing reward functions that are optimal for a problem is a difficult task. This is the reason most RL algorithms are built and tested on games, as it is easier to reward a decision in that scenario.

Using RL for Data Mining (DM) problems is a new idea that has many advantages over traditional DM methods. The major advantage is self-healing. This stems from the property of RL that it is a sequential learning algorithm. That means over time, the RL agent would be able to solve the problems without errors, and if it is exposed to outliers, it learns how to work with such datapoints too, thereby self-correcting itself.

There is a concept that RL is the future of Artificial Intelligence (AI) for the simple reason that any problem can be designed to be a sequential decision-making problem. This would make

RL a universal concept that can solve any type of problem if posed to the agent in a proper format. In nature, when it comes to intelligent life, all our actions are based on sequential decisions. We have two major inputs to factor in, one is the immediate reward for our action, and the second is our long-term gains from that action which are stored as knowledge in our memory. This is comparable to the rewards we receive from the environment and the Bellman equation trying to predict the long-term gains. Since natural intelligence can solve problems other than blatant decision-making, we propose RL can do the same.

To prove this, we have designed this python library or package known as Data Mining Gym (DM-Gym) to showcase the ability of RL to solve classical data-mining tasks such as Clustering, Classification, and Regression. The work involves designing the Markov Decision Process (MDP) and designing the reward function to provide feedback to the agent at each timestep.

The idea for this package was born when we noticed OR-Gym is unintentionally trying to solve the same problem. They built the package to bring RL into the spotlight for industrial use-cases. They also inadvertently showcased that RL can be used for problems other than games and in actual industrial real-world optimization problems.

Our library relies upon the standard OpenAI gym [6] interface for RL and contains multiple environments for solving Clustering and Classification problems. The MDP is also structured in a way that they are usable by the optimization community. OpenAI gym has multiple environments, which can mostly be solved using the model-free RL techniques. Most optimization problems perform well with model-based approaches, so the optimization community relies on this more, even though there is a loss of generality in the models.

Our library can be found along with python notebooks to test these environments on GitHub and is downloadable from PyPI as well.

- Github: https://github.com/ashwin-M-D/DM-Gym
- PyPI: https://pypi.org/project/dm-gym/

Chapter 2

Literature Review

In this section, we discuss RL and how it works. We will also see the type of problems it can solve and how DM problems can be configured so that RL can solve them.

2.1 Reinforcement Learning

RL as we know it today is modeled in the book ”Reinforcement Learning: An introduction” [26]. They go over the basics of RL and how it has improved over the years. They focus on the core ideas of RL and how they work.

RL is a novel ML approach that depends on experience and rewards to train itself in an environment. The agent’s objective is to maximize the reward it gains throughout an episode. The agent uses a Monte-Carlo or a Dynamic Programming based approach to solve the environments.

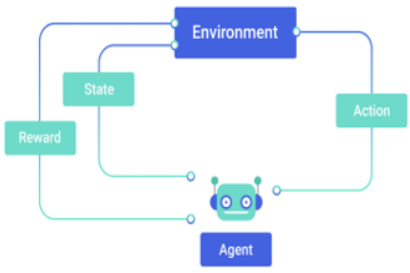


Figure 2.1: Basic Architecture for an RL Training Setup

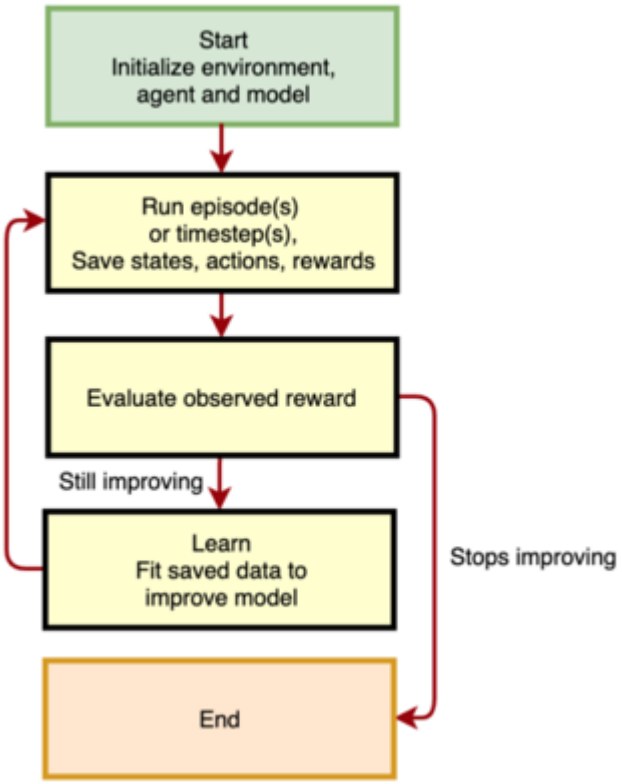


Figure 2.2: Basic Algorithm for Training an RL Agent in an Environment

The backbone for the concept of RL is known as the MDP. This is where the problem is poised as a sequence of decision-making situations, and based on agent's decision in that scenario, feedback in the form of a reward is provided to the agent. The agent, however, should not look at short-term reward maximization but should be able to maximize reward over the entire simulation. This could mean it takes a lower reward action in one situation to get a very large reward later. This is known as the Markov Chain [12].

$$G_t = R_t + \gamma G_{t+1} \tag{2.1}$$

The Bellman equation (Equation 2.1) [3] is the mathematical manifestation of the Markov chain. It considers not only the current reward but also all possible future rewards. The Bellman equation is used to obtain policy values with experience for RL. These policy values for each state can be approximated into a function, and currently, the best function approximator is a concept known as neural networks. This is the basis of Deep Q Networks (DQN) based RL.

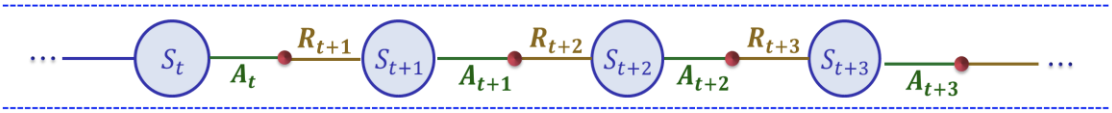


Figure 2.3: A Representation of the Markov Decision Process

Below is the derivation of the Bellman Equation from the theoretical concept of the Markov Chain.

Proof. let G_t and R_t be value of state S and reward at timestep t respectively

$$\begin{aligned} G_t &= R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \gamma^3 R_{t+3} + ... \\ &= R_t + \gamma(R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...) \\ &= R_t + \gamma G_{t+1} \end{aligned}$$

□

2.1.1 Q Learning

Q-Learning is RL in its most basic form. Q-learning is an off-policy reinforcement learning algorithm that seeks to find the best action to take in the current state. It's considered off-policy because the q-learning function learns from actions outside the current policy, like taking random actions, and therefore a policy is not needed. More specifically, q-learning seeks to learn a policy that maximizes the total reward [30].

This algorithm does not use any neural network to approximate functions. It stores values in a table to refer to later. It does not build the Q function but refers to the table to get the values. The table is updated at each step using the Bellman Equation.

The Q-Learning algorithm is shown below (Algorithm 2.1.1).

2.1.2 Deep Reinforcement Learning

In Q learning, since the Q function is not generated but rather referred to from a table, the q values for unknown states are impossible to calculate. This means, that in scenarios where there is a continuous state space, Q-Learning cannot be used. It also becomes impractical if the state space is too large. The larger the state space, the larger the q table would be. This would consume a lot of memory.

Algorithm 2.1.1 : Q-learning (off-policy TD control) for estimating Policy Values.

Require:

step size $\alpha \in (0, 1]$,

$\epsilon > 0$

Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(\text{terminal}, \cdot) = 0$

for episode **do**

Initialize S

for step of episode **do**

Choose A from S using policy derived from Q (e.g., ϵ-greedy)

Take action A, observe R, S'

$Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$

$S \leftarrow S'$

end for

end for

The best way to combat this is to use a function approximator. As mentioned earlier, the best function approximator known are neural networks, and so using this, values for unknown states can be obtained as well.

Once the agent can work on unknown states, the algorithms can focus on other problem aspects in RL, such as accuracy, efficiency, and stability. For these reasons, many different RL algorithms were born. A few of them are discussed here.

2.1.2.1 Deep Q Learning

Deep Q Learning is based on the original Q learning algorithm. The only difference is that instead of a q table, there is a neural network, which takes the state as input and returns the policies for each action in that state (π). This network is known as DQN [21]. The algorithm is shown below (Algorithm 2.1.2).

2.1.2.2 Double Deep Q Learning

DQN solves the issue of handling a large and continuous state space. However, the algorithm is not stable and tends to face a problem known as catastrophic forgetting [17]. It also fails

Algorithm 2.1.2 : Deep Q-learning: Learn function $Q : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$

Require:

States $\mathcal{X} = \{1, \ldots, n_x\}$

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 $T : \mathcal{X} \times \mathcal{A} \to \mathcal{X}$

Discounting factor $\gamma \in [0, 1]$

procedure DEEP Q LEARNING($\mathcal{X}$, A, R, T, γ)

while Q is not converged **do**

Start in state $s \in \mathcal{X}$

while s is not terminal **do**

Get π from DQN and exploration strategy

(e.g. $\pi(x) \leftarrow \arg\max_a Q(x, a)$ or ϵ greedy)

$a \leftarrow \pi(s)$

$r \leftarrow R(s, a)$ ▷ Receive the reward

$s' \leftarrow T(s, a)$ ▷ Receive the new state

$Q'(s, a) \leftarrow r + \gamma \cdot \max_{a'} Q(s', a')$

Train DQN with $(Q'(s, a), Q(s, a))$

$s \leftarrow s'$

end while

end while

return Q

end procedure

to train due to not being able to minimize error. The reason this happens is that Q' and Q are both calculated using the same network. This results in a problem similar to a dog trying to catch its tail.

To overcome this, Double Deep Q Networks (DDQN) uses two networks that work synchronously. One network is used to calculate Q' and the other to calculate Q. The main network, which is being trained, is periodically copied into the other network (Algorithm 2.1.3) [27].

Other than DDQN, there is another way two networks can be trained so that the training is stable and reaches completion. This method, in its raw form, takes longer to train than DDQN but, once combined with advanced RL methods, performs better. This is known as Dueling DQN (Algorithm 2.1.4) [29].

Algorithm 2.1.3 : Double Deep Q-learning: Learn function $Q : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$

Require:

States $\mathcal{X} = \{1, \ldots, n_x\}$

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 $T : \mathcal{X} \times \mathcal{A} \to \mathcal{X}$

Discounting factor $\gamma \in [0, 1]$

procedure DOUBLE DEEP Q LEARNING($\mathcal{X}$, A, R, T, γ)

while $Q1$ is not converged **do**

Start in state $s \in \mathcal{X}$

while s is not terminal **do**

Get π from DQN1 and exploration strategy

(e.g. $\pi(x) \leftarrow \arg\max_a Q1(x, a)$ or ϵ greedy)

$a \leftarrow \pi(s)$

$r \leftarrow R(s, a)$ ▷ Receive the reward

$s' \leftarrow T(s, a)$ ▷ Receive the new state

$Q1'(s, a) \leftarrow r + \gamma \cdot \max_{a'} Q2(s', a')$ ▷ Q2 is calculated using DQN2

Train DQN1 with $(Q1'(s, a), Q1(s, a))$

$s \leftarrow s'$

end while

DQN2 $\leftarrow$ DQN1 ▷ DQN2 is overwritten with the parameters from DQN1

end while

return Q

end procedure

There are multiple implementations, One of which is shown below. In this case, two networks train simultaneously in the same environment, and we end up with two trained networks. Both networks predict the action, and whichever action has a higher Q value will be chosen.

This is better than DQN because having two networks train in this manner removes the possibility of having correlated states coming in for training continuously. Having correlated states while training leads to unwanted bias building up inside the network.

Algorithm 2.1.4 : Dueling Deep Q-learning Network.

Require:

Initialize Q^A, Q^B, s

repeat

$Choose\ a, based\ on\ Q^A(s,\cdot) and Q^B(s,\cdot), observer, s$

Choose (e.g. random or better) either UPDATE(A) or UPDATE(B)

if $UPDATE(A)$ **then**

$Define\ a^* = argmax_a Q^A(s', a)$

$Q^A(s,a) \leftarrow Q^A(s,a) + \lambda(s,a)(r + \gamma Q^B(s', a^*) - Q^A(s,a))$

else if UPDATE(B) **then**

$Define\ b^* = argmax_a Q^B(s', a)$

$Q^B(s,a) \leftarrow Q^B(s,a) + \lambda(s,a)(r + \gamma Q^A(s', b^*) - Q^B(s,a))$

end if

$s \leftarrow s'$

until End

This is, however, not the original Dueling DQN algorithm [29]. The original Dueling DQN algorithm is based on two networks, one which calculates the state value and the other which calculates the action advantage or action policies for that state. Then an algorithm similar to DDQN is run.

2.1.2.3 Actor-Critic Methods

Actor-Critic Methods [16] were developed to combat the problem that DDQN methods cannot handle a continuous action-space. This is not a single algorithm but rather a family of different algorithms. They have a minimum of two networks. An actor-network provides the action for a given state, and a critic-network provides the state value, action policies, or any other evaluation based on the action taken by the actor for a given state.

The base algorithm is simple (Algorithm 2.1.5). It is known as Q Actor-Critic. It has two networks. w is the Q Network, also known as the Critic, and θ is the actor-network.

More complex Actor-Critic algorithms exist, such as Advantage Actor-Critic (A2C) [20], Asynchronous Advantage Actor-Critic (A3C) [20], and Soft Actor-Critic (SAC) [13]. They can have more than two neural networks and they might use other techniques such as experience replay to improve training capabilities and performance.

Algorithm 2.1.5 : Q Actor-Critic

Require:

Networks: θ, w

learning rates: α_θ, α_w

for episode **do**

Initialize s

Sample $a \sim \pi_\theta(a|s)$

for step of episode **do**

$r_t \leftarrow R(s, a)$

$s' \leftarrow P(s'|s, a)$

$a' \leftarrow \pi_\theta(a'|s')$

$\theta \leftarrow \theta + \alpha_\theta Q_w(s, a) \nabla_\theta log \pi_\theta(a|s)$: Update the Policy Parameters

$\delta_t = r_t + \gamma Q_w(s', a') - Q_w(s, a)$: TD error calculation at time t

$w \leftarrow w + \alpha_w \delta_t \nabla_w Q_w(s, a)$: Update Q Function

$a \leftarrow a'$ and $s \leftarrow s'$

end for

end for

2.2 Data Mining

The problems being solved in this paper are classical DM problems. These include clustering, classification, and regression for value prediction. These problems can be solved by multiple classical approaches. To compare our model's performance, we use the K-Means clustering algorithm for clustering problems, and logistic regression for classification,. We have not converted linear regression into an MDP yet.

2.2.1 Clustering

Clustering is an unsupervised ML problem where datapoints are grouped into clusters based on their proximity to one another. It is a very important problem that can be solved in many ways. Each solution to the clustering problem has its advantages and disadvantages.

The most famous method for solving clustering problems is K-Means clustering. It is popular because it is not computationally expensive. However, it is dependent on knowing beforehand the number of clusters present in our data. It is also dependant on the clusters being centroidal

clusters.

An algorithm that counters the first problem of knowing the number of clusters beforehand is the Mean-Shift algorithm. This is computationally expensive and is still dependent on data containing centroidal clusters.

An algorithm that solves all the problems faced by the above algorithms is Density-Based Spatial Clustering of Applications with Noise (DBSCAN). This algorithm is not dependent on knowing the number of clusters beforehand and can also identify non-centroidal clusters. However, this requires a lot more parameters than the other algorithms. It is also computationally expensive. It does not perform well with clusters of varying density and can also report datapoints as not belonging to any cluster.

Our clustering MDP models are based on the K-Means algorithm. While comparing the results of our environment with classical data mining techniques, we have used K-Means and Mean-Shift algorithms.

2.2.1.1 K-Means Clustering

K-Means clustering is a popular clustering method as mentioned above. In real life, in many cases, we already know the number of clusters in a dataset, and most data follow a normal distribution making this algorithm very desirable.

The algorithm works by iteratively updating the centroids of the clusters up until the centroids no longer move. It is computationally inexpensive as well.

The algorithm, as was originally thought of by James MacQueen [19], starts with

$$S_i^{(t)} = \left\{ x_p : \left\| x_p - \mu_i^{(t)} \right\|^2 \leq \left\| x_p - \mu_j^{(t)} \right\|^2 \forall j, 1 \leq j \leq k \right\} \tag{2.2}$$

The initial μs or centroids are some randomly selected points in the subspace of the data. S_i is the set of the closest points to μ_i.

The centroids are then updated to be the mean of the closest points.

$$\mu_i^{(t+1)} = \frac{1}{|S_i^{(t)}|} \sum_{x_j \in S_i^{(t)}} x_j \tag{2.3}$$

This update occurs up until the update equation produces no change in the values of μ_i.

2.2.1.2 Mean Shift Clustering

Mean shift is more expensive than K-Means Clustering (Section 2.2.1.1) but solves the problem of not knowing the number of clusters beforehand. The algorithm is officially introduced by Fukunaga and Hostetler [11]. The algorithm starts with selecting a large number of centroids (N). Points are selected such that

$$S_i^{(t)} = \left\{ x_p : \left\| x_p - \mu_i^{(t)} \right\|^2 \leq W \right\} \tag{2.4}$$

Where W is a scalar known as a window. It is user-defined. Once the points within a certain distance (W) are chosen, centroids are updated as follows

$$\mu_i^{(t+1)} = \frac{1}{|S_i^{(t)}|} \sum_{x_j \in S_i^{(t)}} x_j \tag{2.5}$$

If any centroids overlap, duplicate points are deleted.

This update occurs until the update equation produces no change in the values of μ_i. After this, we are left with the centroids for all the clusters, and we also know the number of clusters in the system.

2.2.2 Classification

Classification is a supervised ML problem where datapoints are predicted to belong to a certain group based on knowledge obtained from provided data and labels.

Our classification MDP models are based on the K-nearest algorithm. While comparing the results of our environment with classical data mining techniques, we will be using K Nearest neighbors (KNN) and logistic regression.

2.2.2.1 K-Nearest Neighbors

The algorithm is computationally very cheap. However, the algorithm does require a lot of memory. The entire dataset needs to be stored on the deployed system's RAM (Random Access Memory) because, in the case of KNN, the dataset is essentially the model itself. This makes it impractical in the case of large training datasets.

The non-parametric method used by KNN was first described by Evelyn Fix and Joseph Hodges in 1951 [10]. The algorithm was utilized for classification and regression by Thomas Cover [1].

The algorithm itself is simple and is as follows.

Key idea: store all training examples $\langle x_i, f(x_i)\rangle$

k-Nearest neighbor:

Given x_q,

1. Take vote among its k nearest neighbors (if discrete-valued target function)

2. Take the mean of f values of k nearest neighbors (if real-valued)

$$\hat{f}(x_q) = \frac{\sum_{i=1}^{k} f(x_i)}{k} \tag{2.6}$$

The nearest neighbor can be calculated using any metric. Despite Euclidian Distance (Equation 2.7) being the most popular, Manhattan Distance (Equation 2.8) and L_m distance (Equation 2.9) can also be used.

$$Euclidian\, distance\; d = \sqrt{\sum_{i=1}^{n}(X1_i - X2_i)^2} \tag{2.7}$$

$$Manhattan\, distance\; d = \sum_{i=1}^{n}(|X1_i - X2_i|) \tag{2.8}$$

$$L_m\, distance\; d = (\sum_{i=1}^{n}(|X1_i - X2_i|)^m)^{1/m} \tag{2.9}$$

This is an elegant solution and works effectively. However, as stated before, it is memory intensive and therefore not highly scalable. There are workarounds for this. The model could be limited by the number of datapoints in it and choosing those datapoints carefully. The downside is that the model performance is affected as the entire dataset would not be used to calculate the test datapoint's value.

2.2.2.2 Logistic Regression

Logistic regression estimates the probability of an event occurring, such as being or not being in a certain class, based on a given dataset of independent variables. Since the outcome is a probability, the dependent variable is bounded between 0 and 1.

The odds are calculated using logit (Equation 2.10). It is a linear equation of all variables in a datapoint. During training, the coefficients (β_i) of this logit are varied to give the best possible odds. The logit is passed through a sigmoid function (Equation 2.11) to keep the dependant variable bounded between 0 and 1.

The logistic regression as a general statistical model was originally developed and popularized primarily by Joseph Berkson where he coined "logit" [4] [23]

$$z = \beta_0 + (\sum_{i=1}^{n} \beta_i X1_i) \tag{2.10}$$

$$p = \frac{1}{1 + e^{-z}} \tag{2.11}$$

There are multiple models, one for each possible class. The value of p, which is the highest among all classes, is chosen as the class for the datapoint.

The coefficients can be updated during training by various methods, but the simplest and most popular method is Gradient Descent [8] [24]. The algorithm is as follows (Algorithm 2.2.1).

Algorithm 2.2.1 : The general gradient descent algorithm

Require:

initial weights $\beta^{(0)}$, number of iterations T

final weights $\beta^{(T)}$

learning rate α

different choices of the learning rate α and the estimation technique for $\nabla\mathcal{L}(\beta)$ may lead to different implementations.

for do$t = 0$ to $T - 1$

estimate $\nabla\mathcal{L}(w^{(t)})$

compute $\Delta w^{(t)} = -\nabla\mathcal{L}(\beta^{(t)})$

$\beta^{(t+1)} := \beta^{(t)} + \alpha\Delta\beta^{(t)}$

end for

return $w^{(T)}$

This algorithm has been modified and improved upon to produce multiple variations. The most popular is the Adam optimizer [14].

2.2.3 Prediction

Prediction is one of the most basic DM problems. It is looking at data and extrapolating it into test data to predict its dependent variable. The most common algorithm used is linear regression.

2.2.3.1 Linear Regression

Linear regression is the prediction of a dependent variable using other independent variables in the test datapoint. It was first introduced in its now recognizable form by Yule and G Udny [32] in the late 19th century. It has been worked upon since then, and once combined with gradient descent (Algorithm 2.2.1), it is a very effective predictor.

It works similar to the logit in Logistic Regression (Section 2.2.2.2). The equation and algorithm are the same. However, the logit is not passed through the sigmoid function since the dependent variable here is not bounded.

The equation is the same (Equation 2.12) and is passed through the gradient descent algorithm (Algorithm 2.2.1) to improve the coefficients (β_i). It can also be solved using other optimizer algorithms.

$$z = \beta_0 + (\sum_{i=1}^{n} \beta_i X1_i) \tag{2.12}$$

2.2.4 Neural Networks

Neural networks [28] can be modified to solve any of the mentioned problems. At the base, they are essentially the same as linear regression but applied multiple times in a series of layers. The output can be a single value or multiple values. The values can be modified using different functions such as sigmoid or softmax to work on classification.

It is difficult to modify neural nets to solve clustering problems as neural nets are inherently supervised machine learning algorithms. However, people have worked on this and developed algorithms such as Self Organising Maps (SOM) [15]. It is widely different from the normal understanding of neural networks as it does not have a fixed number of neurons. It has a set of neurons connected to form a topological grid (usually rectangular). When some pattern is presented to a SOM, the neuron with the closest weight vector is considered a winner, and its weights are adapted to the pattern, as well as the weights of its neighborhood. In this way, a SOM naturally finds data clusters.

During the training of a neural network, we use the same algorithm as we use in the case of logistic regression or linear regression. Gradient Descent (Algorithm 2.2.1) is still popular, but its variations are much more popular as its higher computational resource requirements would be justified. Adam [14] is the most popular optimizer for neural networks.

2.2.5 Using Reinforcement Learning for Data Mining Problems

RL is a new and evolving field. It has shown promise to be the first step towards general artificial intelligence. To do so, it must be able to solve DM problems too. There have been attempts to do this.

Aristidis Likas tried to solve this in his paper "A Reinforcement Learning Approach to Online Clustering" [18]. He tried to solve clustering with the assumption that data would appear in a stream and the agent would have to cluster it in each timestep. The paper "Reinforcement learning algorithms for solving classification problems" [31] tries to solve classification using RL. Many others have worked on solving these problems too [22] [7] [5].

All RL researchers require a simulation environment to test their work. This project aims to provide the researchers with just that. This project provides different DM problems formulated and built as RL environments.

Chapter 3

DM-Gym: The Python Package

The DM-Gym package/library is written in python and can be used without modifications with python. It has been tested with a python version greater than 3.6. It is incompatible with python 2. It requires the following packages:

- gym >=0.15.0
- numpy >=1.16.1
- scipy >=1.0
- matplotlib >=3.1
- networkx >=2.3
- scikit-learn >=1.0

To use the package, you need to install it on your system. you can install it onto your system using pip.

```
pip install dm-gym
```

You can also download the package from GitHub directly and use it by placing the *dm_gym* folder inside your project folder.

The library is gym complete. This means that it follows the standard OpenAI gym structure and functions. The library is also Ray complete. This means that is compatible with Ray and

all its libraries for building an agent and tuning its hyperparameters, such as Ray RLlib and Ray Tune.

The library includes multiple versions of Clustering and Classification environments. Once the MDP for regression environments are developed, they will be added to the library as well. The environments can handle any dataset and other parameters which may be required can be passed into the environment through a config dictionary.

All the environments have been showcased using the Ray package in multiple python notebooks to demonstrate the environments' success in replicating results from classical ML techniques.

3.1 Structure of the package

This section is to help the users of this package with using it effortlessly. We have multiple prebuilt functions to help with that.

3.1.1 OpenAI Gym

As mentioned above, the package is gym complete. After importing or_gym into the python environment, one can create different environments. This can be done using the create_env() function.

```
import dm_gym
from dm_gym.create_env import create_env

env_name = "<env_name>"
env_config = {...<"envconfig">...}

env = create_env(env_name, env_config=env_config)

## This creates an initiated environment.
## It is important to note that you need to specify
## env_config=env_config i.e **kwargs and not *args.
```

One can use the gym.make() function but will need to import gym first and need to make sure to specify dm_gym as the source of the environment inside gym.make().

The environment has the basic structure of any other gym based environment. It has 3 functions: reset(), step() and render(). Depending on the environment, the render function defers. However, the reset function resets the environment and returns the first observation. The step function takes the action from the previous state and returns the next_observation, reward, done_state, and info_dict in that order.

3.1.2 Ray RLlib and Ray Tune Compatibility

The environments are ray compatible but the environments cannot be created outside ray. For this purpose, another function is used to create the environment: ray_create_env(). This returns the class itself and not an instance of it. This is can be used inside the register_env() function required for ray tune.

```
import ray
from ray.rllib import agents
from ray import tune

from dm_gym.create_env import ray_create_env

def register_env(env_name, env_config={}):
    env = ray_create_env(env_name)
    tune.register_env(env_name,
    lambda env_name: env(env_name,
    env_config=env_config))

env_name = "<env_name>"
env_config = {<"env config">}

register_env(env_name, env_config)
```

This registers the environment with ray tune. For more details look into the ray project's documentation.

3.2 Environments

The project contains multiple environments to solve both classification and clustering problems. Prediction problems that are generally solved by linear regression have not been converted into an MDP yet.

3.2.1 Clustering

There are four different clustering environments in the package. The only difference between them is the reward function. The environment configs for the environments are as below.

```
env_name = "clustering-v0"
env_config = {  'data': data,
                'k': k
             }

env_name = "clustering-v1"
env_config = {  'data': data,
                'k': k
             }

env_name = "clustering-v2"
env_config = {  'data': data,
                'k': k,
                'max_steps': max_steps,
                'lr': learning_rate
             }

env_name = "clustering-v3"
env_config = {  'data': data,
                'k': k
             }
```

'data' is the dataset in the form of a pandas dataframe. 'k' refers to the number of clusters.

3.2.2 Classification

There is one classification environment in the package. The environment configuration for the environment is as below.

```
env_name = "classification-v0"
env_config = {  'data': data,
                'target': target,
                'num_classes': k
             }
```

'data' is the dataset in the form of a pandas dataframe. Target refers to the target class of the data. It is expected to be a numpy array. 'k' refers to the number of possible classes.

3.3 Datasets

DM-Gym has the ability to generate random data to test your environments. The function in DM-Gym is called data_gen().

```
from dm_gym.utils.data_gen import *

### For Clustering type dataset
datagen = data_gen_clustering()

### For Classification type dataset
datagen = data_gen_clustering()

datagen.param_init(n=<num_features>, k=<num_clusters>,
num_records=<num_records>, parameter_means=[], parameter_sd=[])

### For Clustering type dataset
data = datagen.gen_data()

### For Classification type dataset
data, target = datagen.gen_data()
```

An example of how the required parameters are assigned is as follows. The parameters n, k, and $num\_records$ are required. The others are generated randomly if not specified

```
### Example of parameters
n = 5 ## Five features
k = 3 ## Three classes

parameter_means = [
                [ftr-1-means],
                [ftr-2-means],
                [ftr-3-means],
                [ftr-4-means],
                [ftr-5-means]
                ] ## is n x k matrix

parameter_means = [[1,5,9], [1,6,4], [1,7,3], [2,2,2], [7,8,9]]
## So the centroids would equate to
## (1,1,1,2,7) (5,6,7,2,8) (9,4,3,2,9)

parameter_sd = [
            [ftr-1-std],
            [ftr-2-std],
            [ftr-3-std],
            [ftr-4-std],
            [ftr-5-std]
            ] ## is n x k matrix

parameter_sd = [[1,1,1], [1,1,1], [1,1,1], [1,1,1], [1,1,1]]
## This is the default value for parameter_sd given n=5 and k=3)
```

The package also has inbuilt test models. This means that one can test the dataset with traditional algorithms and then compare the results with the RL algorithm. These functions are imported from sklearn and so are very fast. The package does not require any modeling or building from the user's side.

For clustering, we have k-means clustering and mean shift algorithms built in. These are included inside dm_gym.data_gen and are imported with it.

```
from dm_gym.utils.data_gen import *

datagen = data_gen_clustering()

datagen.param_init(n=<num_features>,
                   k=<num_clusters>,
                   num_records=<num_records>,
                   parameter_means=[],
                   parameter_sd=[])

data = datagen.gen_data()

## Mean Shift Algorithm (Scikit-learn)
final_data, centroids = datagen.gen_model(data=data)

## K-Means Clustering Algorithm (Scikit-learn)
final_data, centroids = datagen.gen_model_Kmeans(data=data,k=k)

## data can be from anywhere and does not need to be generated.

### if you generated the data using datagen, then you need
### not specify k. Otherwise, the value of k is required.

### If k is not specified, it defaults to 2.
```

The Outputs are as follows:

- final_data: is a dataset that contains the original data along with an additional column called ”Class” which contains a label stating which cluster it belongs to.
- centroids: is a 2d python list that contains tuples of the centroids of the clusters. The order is the same as the label itself.

For classification, we have logistic regression built in.

```
from dm_gym.utils.data_gen import *

datagen = data_gen_classification()

datagen.param_init(n=<num_features>, k=<num_clusters>,
num_records=<num_records>, parameter_means=[],
parameter_sd=[])

data = datagen.gen_data()

## Logistic regression (Scikit-learn)
final_data, training_accuracy, classifier = \\
datagen.gen_model(data=data, target=target)

## data can be from anywhere and does not need to be generated.

### if you generated the data using datagen, then you need
### not specify k. Otherwise, the value of k is required.

### If k is not specified, it defaults to 2.
```

Detailed documentation on the package can be found on GitHub at https://github.com/ashwin-M-D/DM-Gym.

Chapter 4

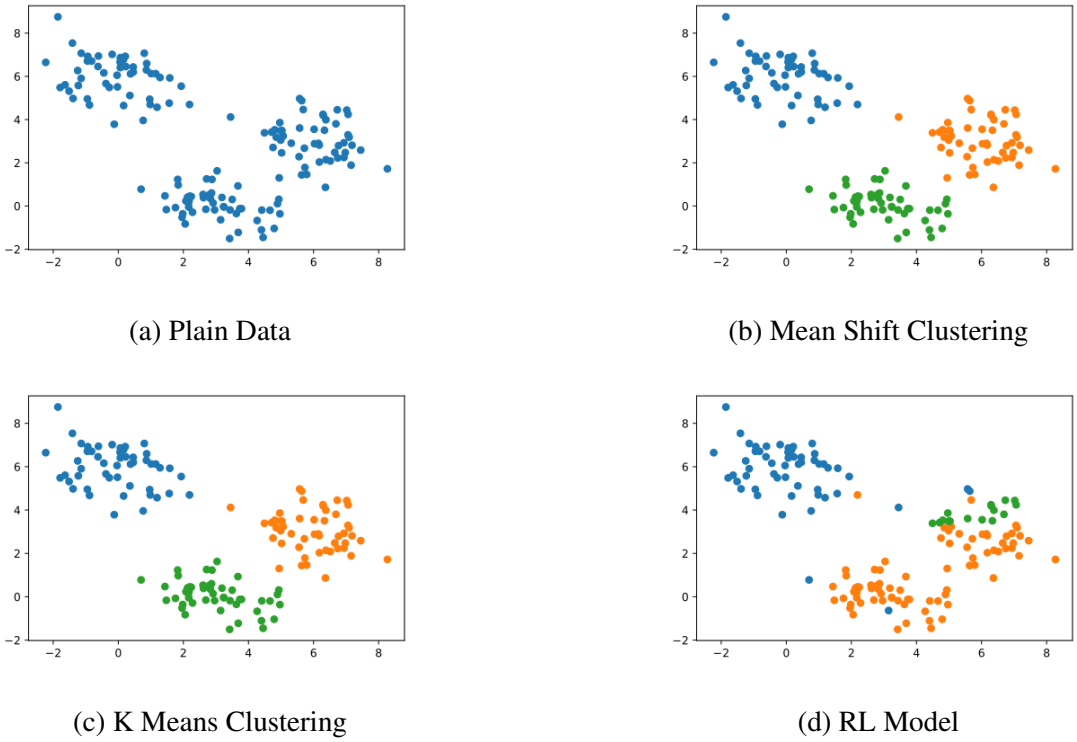
Problems and Results

The DM problems converted into MDPs here are Clustering (Section 4.1) and Classification (Section 4.2). We demonstrate the use of these environments using Ray RLlib. We also compare the results with traditional data mining techniques which are generally employed to solve these problems.

Though the models perform better if the parameters are fine tuned for each envrionment, we kept the parameters nearly identical for all environments to be able to compare the results of the environments. Each environment was run on a DDQN model built in Ray RLlib. All models were trained for 500 iterations. Datasets were generated for each model using the data_gen() function. We used the default parameters. For clustering environments, we generated one hundred fifty datapoints with two features each. For the classification environment, we generated one thousand datapoints with five features each. All environments had three clusters or classes. We also ran the models on the IRIS dataset [9] [2] to provide a better comparison of our agent's performance.

4.1 Clustering

There are four different clustering environments in the package. They are labeled as clustering-v0 (Section 4.1.1), clustering-v1 (Section 4.1.2), clustering-v2 (Section 4.1.3), and clustering-v3 (Section 4.1.4). They differ by the parameters they take and the reward function they utilize. The order in which the points are provided by the environment is random after each reset. However, no datapoints are repeated in a single iteration.



(a) Plain Data

(b) Mean Shift Clustering

(c) K Means Clustering

(d) RL Model

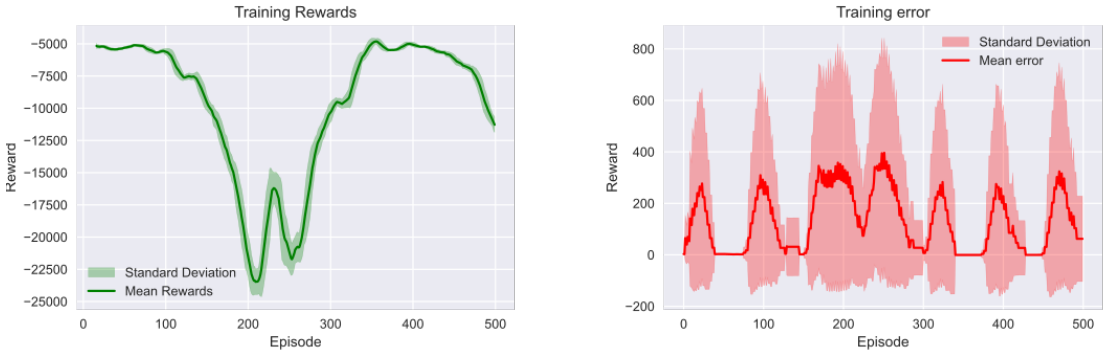
Figure 4.1: Clustering V0 Comparison

4.1.1 Clustering-V0

Clustering-V0 takes the data and number of clusters (k) as its parameters. It works by taking in the datapoint as input and providing the q-values for each cluster as output. The reward function is designed as follows:

$$Reward = \begin{cases} -1000 & num\_clusters \leq k \\ -log_{10}(dbindex(allPreviousObservations)) & num\_clusters = k \end{cases} \tag{4.1}$$

From the training graphs (Figure 4.2) one can see that the learning seems to be unstable. This however is expected as the reward function has high variation in its values. Looking at the comparison plots (Figure 4.1) we see that the agent is able to cluster the data but certain clusters are overpowering the entire data. To fix this, we need to add a regularization term into our reward function. This has been implemented in clustering-v1 (Section 4.1.2).



(a) Rewards

(b) TD Error

Figure 4.2: Clustering V0 Training Plots

4.1.2 Clustering-V1

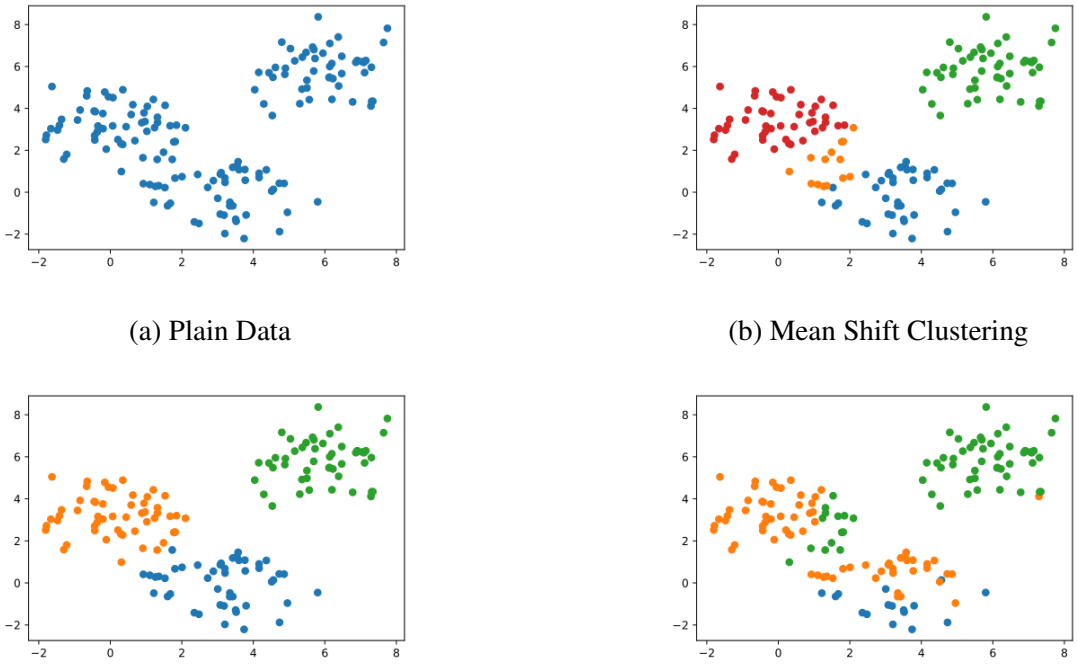
This works very similar to clustering-V0 however, the reward function is different. It provides a reward at the end of each iteration. We rely on the Markov Chain to propagate the rewards to earlier decisions.

$$Reward = \begin{cases} 0 & done \neq True \\ -2 * log_{10}(dbindex(allPreviousObservations)) & \\ +regularization\_term & done = True \end{cases} \tag{4.2}$$

The regularization term is a minute quantity that tries to make sure that no one cluster fully takes over the entire dataset if the clusters are close by.

Looking at the training plots (Figure 4.4) one can infer that this algorithm is a bit more stable than clustering-v0. The problem of forgetting at the end of training can be corrected by fine-tuning the parameters based on the type of data being handled. The error plot is also converging towards zero without a high deviation.

The comparison plots (Figure 4.3) show the performance increase just by adding a regularization term. The clusters are more distinct and are correctly recognized. The regularization term can be modified to obtain better results.



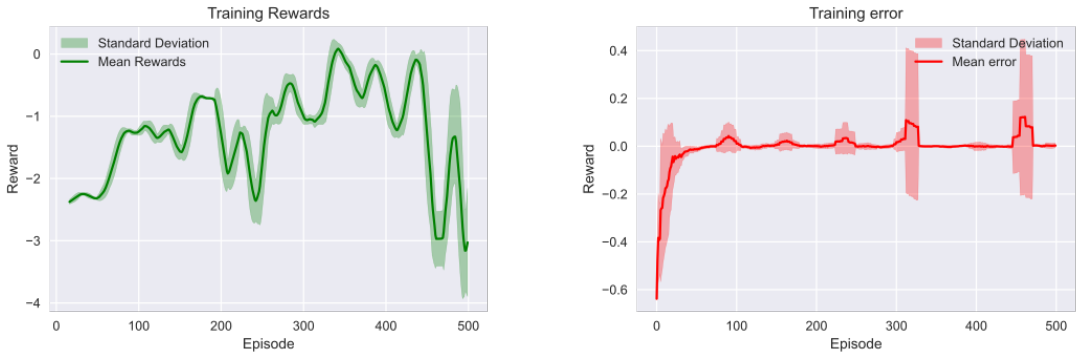
(a) Plain Data

(b) Mean Shift Clustering

(c) K Means Clustering

(d) RL Model

Figure 4.3: Clustering V1 Comparison



(a) Rewards

(b) TD Error

Figure 4.4: Clustering V1 Training Plots

4.1.3 Clustering-V2

This environment works a bit differently compared to the other environments. It takes in more parameters than clustering v0 and v1. In addition to data and k, it also takes in a parameter known as max_steps and learning rate (lr).

The environment keeps track of the centroids of the clusters and updates them on each step. The learning rate parameter is required to state how quickly these centroids are to be updated.

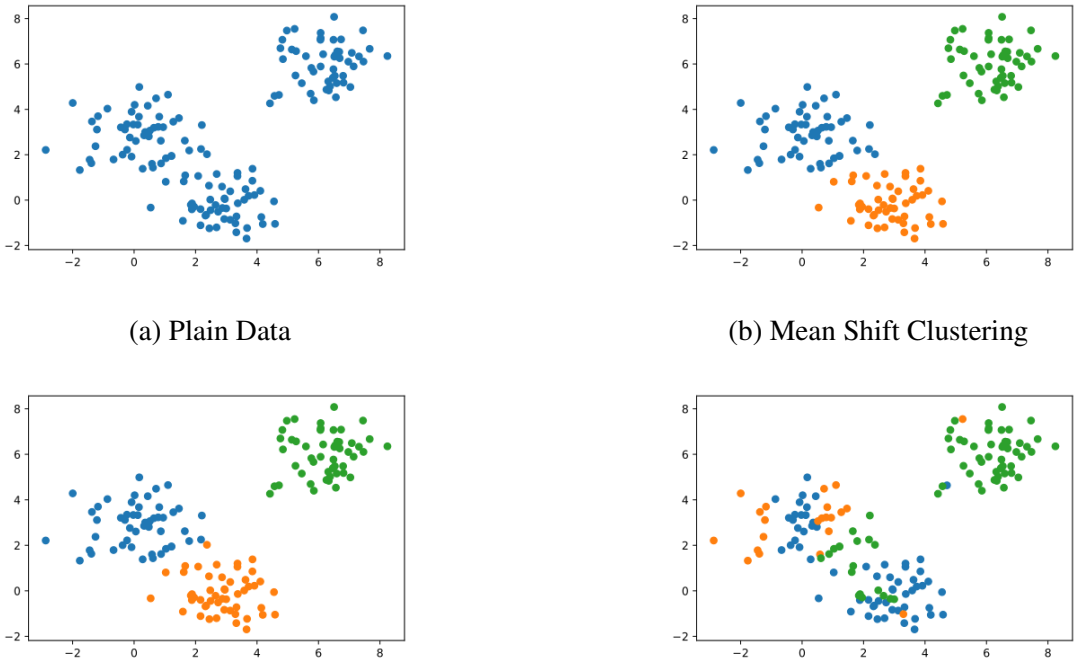
The max_steps parameter is an optional parameter that tells the environment to provide just those many points in each iteration before done is set to true. It is a value that should not be greater than the size of the dataset. If it is not specified, the parameter is automatically set to be the size of the dataset.

$$Reward = \begin{cases} p_i & y_i = 1 \\ -1 * (1 - p_i) & y_i = 0 \end{cases} \tag{4.3}$$

p_i is a value which is calculated based on the distance between centroid(i) and the datapoint. It is bounded between 0 and 1. y_i is 1 if centroid(i) is the closest centroid to the datapoint. Else, the value is set to 0.

The performance of clustering-v2 is not as good as clustering-v1. During training, the rewards are decreasing and the error is not consistent (Figure 4.6). To train using clustering-v2, one needs to run the model with a low learning rate for a large number of iterations. However, looking at the comparison plots (Figure 4.5), we see that the clusters are identified with a lot of noise.

Running this model on the iris dataset, we get results comparable to the mean shift algorithm (Figure 4.7) (Figure 4.9). If two clusters are in very close proximity, the model struggles to find the independent clusters. Also, we see that with higher dimensional data such as iris, the environment is stable (Figure 4.8).



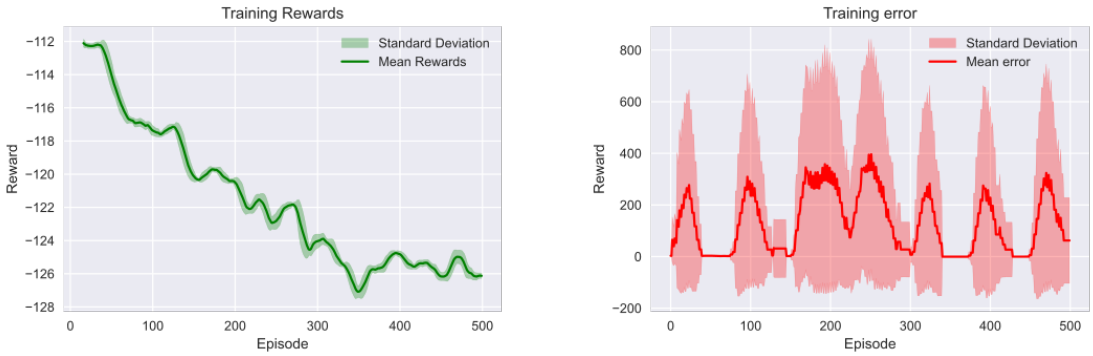
(a) Plain Data

(b) Mean Shift Clustering

(c) K Means Clustering

(d) RL Model

Figure 4.5: Clustering V2 Comparison



(a) Rewards

(b) TD Error

Figure 4.6: Clustering V2 Training Plots

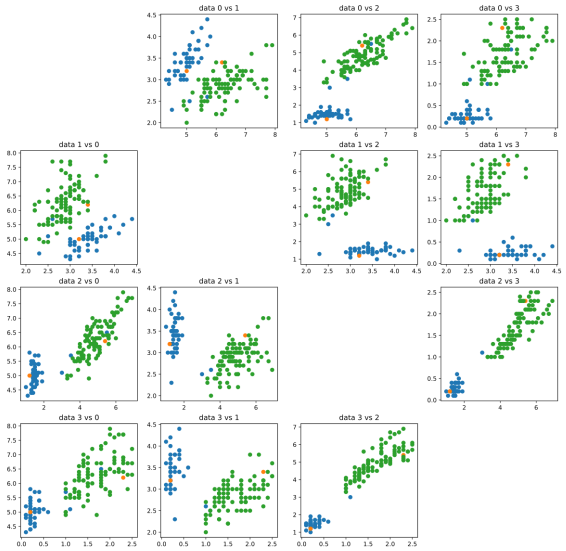
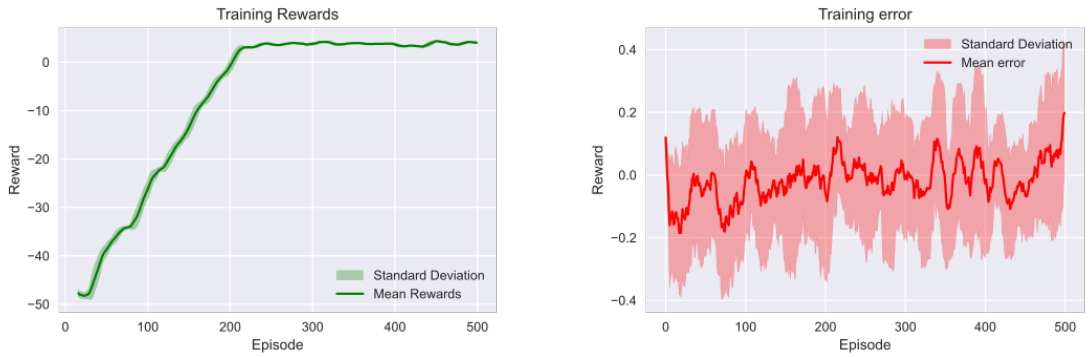


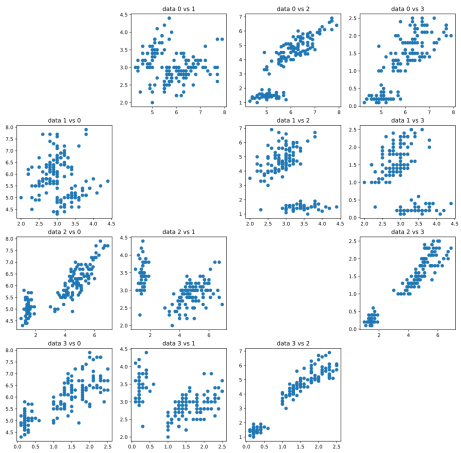
Figure 4.7: RL Model using Clustering V2 on IRIS Dataset



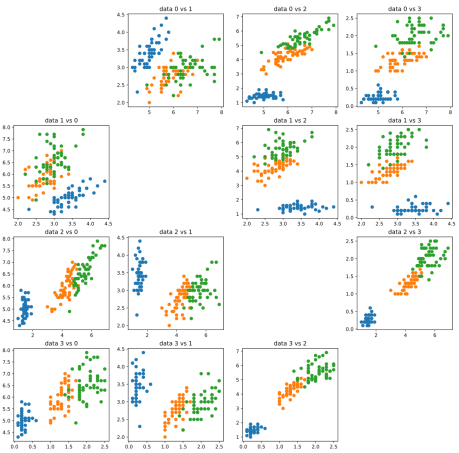
(a) Rewards

(b) TD Error

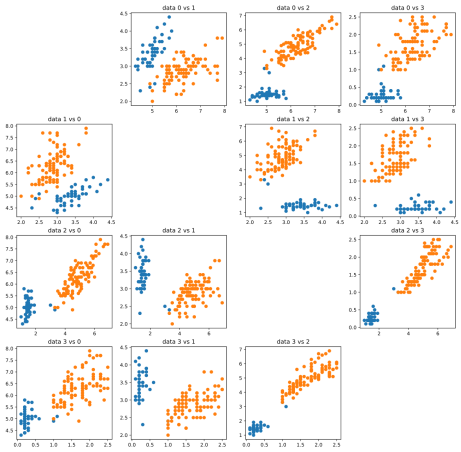
Figure 4.8: Clustering V2 Training Plots for IRIS Dataset



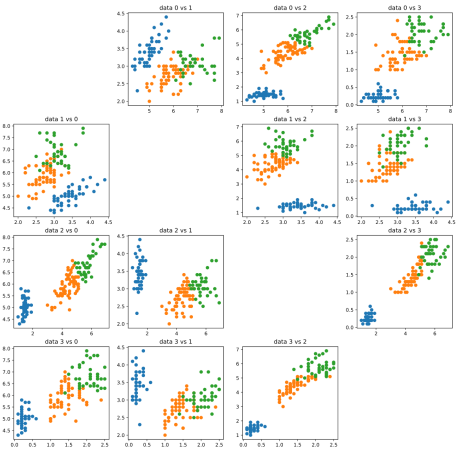
(a) Unlabeled Iris Dataset



(b) Actual labels - Iris Dataset



(c) Iris Dataset solved by Mean Shift Clustering



(d) Iris Dataset solved by K Means Clustering

Figure 4.9: IRIS Dataset

4.1.4 Clustering-V3

Clustering-v3 is based on k-means clustering and we use it to generate our reward. It takes the data and k as parameters. From the provided data, we generate a k-means clustering model and obtain the centroids of the cluster. We check if the selected cluster is the right cluster according to the k-means model and provide the reward accordingly.

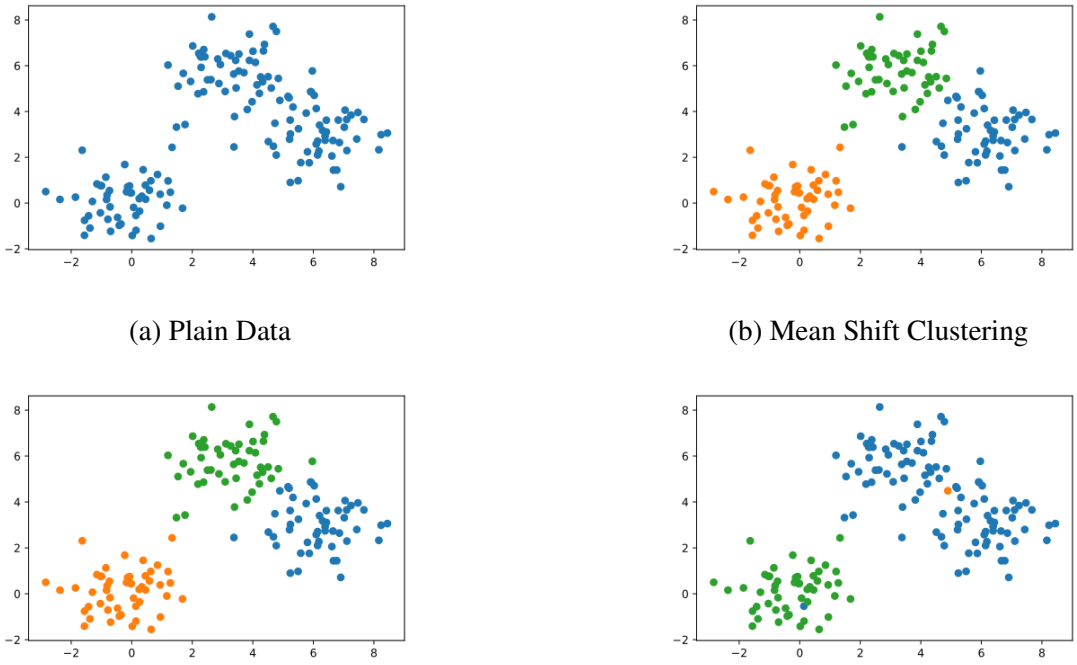
$$Reward = \begin{cases} 1 & y_i = 1 \\ -1 & y_i = 0 \end{cases} \tag{4.4}$$

y_i is 1 if the chosen centroid(i) is the closest centroid to the datapoint. Else, the value is set to 0.

In this case, training is very stable (Figure 4.11). The rewards increase almost linearly until it reaches convergence. The error plot also converges towards zero. There is a significant deviation in the error plot. This can be fixed by reducing the learning rate of the model.

The comparison plot (Figure 4.10) shows us that the algorithm has a hard time separating the close clusters. This can also be fixed by fine-tuning the parameters of the agent and training it over a larger number of iterations.

When this model is run over the iris dataset, we see really good results. The training plots are still stable (Figure 4.13). The clusters are very clearly identified as well (Figure 4.12) (Figure 4.9b).



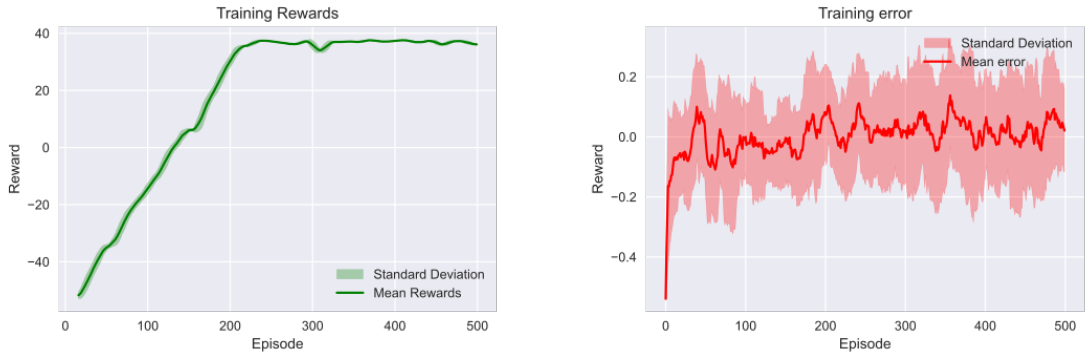
(a) Plain Data

(b) Mean Shift Clustering

(c) K Means Clustering

(d) RL Model

Figure 4.10: Clustering V3 Comparison



(a) Rewards

(b) TD Error

Figure 4.11: Clustering V3 Training Plots

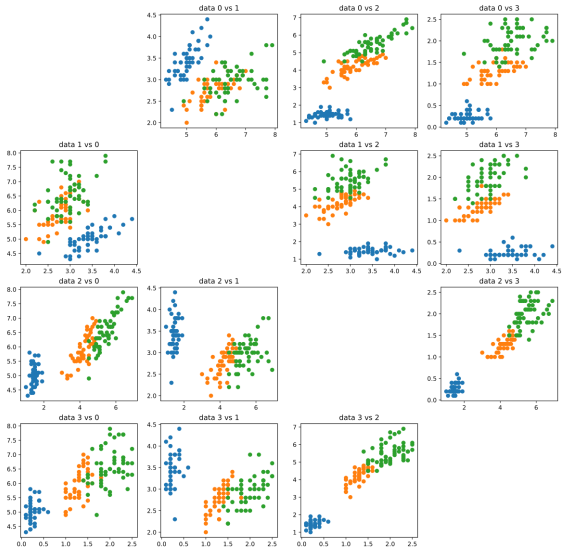
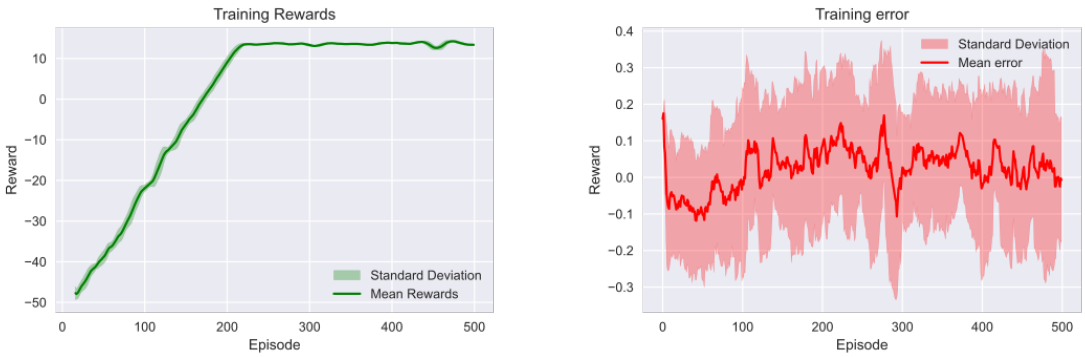


Figure 4.12: RL Model using Clustering V3 on IRIS Dataset



(a) Rewards (b) TD Error

Figure 4.13: Clustering V3 Training Plots for IRIS Dataset

Accuracy	Generated Data		Iris Dataset	
	Logistic Regression	RL Model	Logistic Regression	RL Model
Training Data	1.0	1.0	0.975	0.95
Test Data	1.0	1.0	0.967	1.0

Table 4.1: Results of RL Model Using Classification V0

4.2 Classification

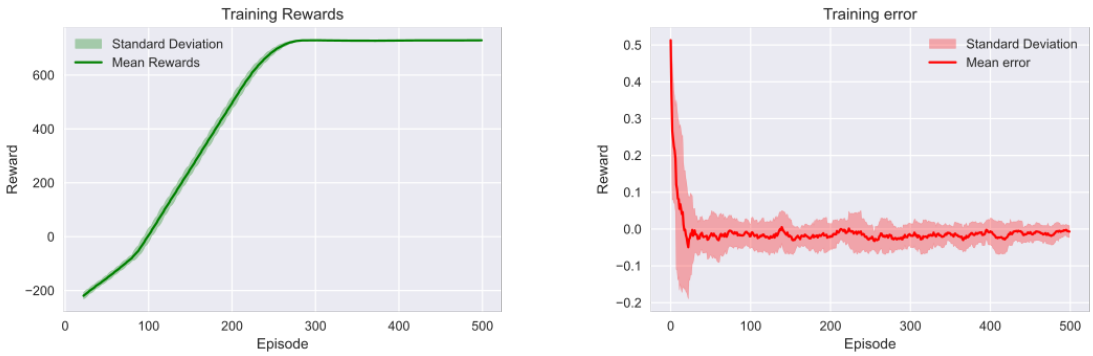
Classification is much simpler than clustering as it contains labeled data. We have a single environment set up to solve this model: classification-v0.

4.2.1 Classification-V0

This works similarly to clustering-v3. However, we do not need to use a k-means clustering model to obtain the labels. It takes in the data, labels (target), and the number of classes as input. The reward is provided at each step as follows:

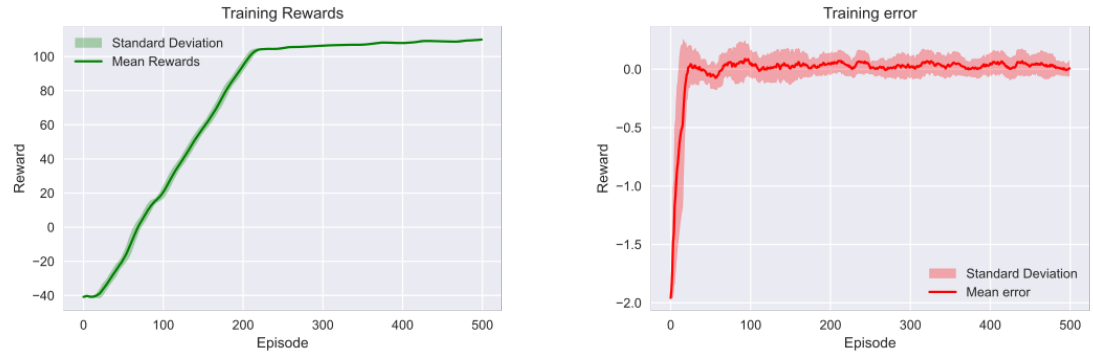
$$Reward = \begin{cases} 1 & expected\_class = predicted\_class \\ -1 & expected\_class \neq predicted\_class \end{cases} \tag{4.5}$$

Since the environment works very similar to clustering-v3, we expect similar results. The data is classified very well (Table 4.1). The training plots (Figure 4.14) (Figure 4.15) show very stable learning as well. The errors converge to zero and the standard deviation is very low. The reward function also stabilizes smoothly.



(a) Rewards (b) TD Error

Figure 4.14: Classification v0 Training Plots



(a) Rewards (b) TD Error

Figure 4.15: Classification v0 Training Plots for IRIS Dataset

Chapter 5

Conclusion

RL is a sequential decision-making algorithm that can solve any problem modeled as a Markov chain. We decided to showcase this ability using DM problems. Since most problems already exist as either a DM problem or a sequential decision-making problem, we theorize that RL can solve any problem posed to it if modeled correctly.

The advantages of having this ability are immense. Any problem can be solved using a single algorithm. This saves man hours and resources which would have gone into analyzing data to understand the type of problem being solved and then building that model. It also brings us one step closer to general AI. The same program may not solve all problems, but the same algorithm does.

From the results we obtained, we see that RL models perform well when the data is complex with a large number of dimensions. Else, the rewards tend to decrease with time. This could be fixed by varying the reward function to be more optimal.

The current clustering environments can only work with centroidal clusters and not ambiguous clusters. Those can be solved using traditional DM techniques such as DBSCAN. Generating such an environment requires us to rethink the entire concept of clustering from scratch.

We also plan on building an MDP on prediction. Unlike Clustering and Classification, Prediction is done using a single algorithm, linear regression. However, the same results can be obtained by using a simple neural net. This poses a unique problem as all neural nets, while training, is basically a minimization optimization problem. i.e they try to minimize the error over time. RL on the other hand is a maximization problem. We try to maximize the rewards obtained over time. Converting a minimization problem into a maximization problem is simple. However, in this case, if we use any major RL algorithm, we tend to use a neural net within it. This means that we

convert the minimization problem into a maximization problem which is inherently converted into a minimization problem again. This is inefficient and needs to be worked on.

The codes used to run these experiments, along with results and preset parameters, can be found on GitHub at https://github.com/ashwin-M-D/DM-Gym

Bibliography

[1] Naomi S Altman. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician*, 46(3):175–185, 1992.

[2] Edgar Anderson. The species problem in iris. *Annals of the Missouri Botanical Garden*, 23(3):457–509, 1936.

[3] EN Barron and H Ishii. The bellman equation for minimizing the maximum cost. *Nonlinear Analysis: Theory, Methods & Applications*, 13(9):1067–1090, 1989.

[4] Joseph Berkson. Application of the logistic function to bio-assay. *Journal of the American statistical association*, 39(227):357–365, 1944.

[5] Sourabh Bose and Manfred Huber. Semi-unsupervised clustering using reinforcement learning. In *The Twenty-Ninth International Flairs Conference*, 2016.

[6] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[7] Duo Chai, Wei Wu, Qinghong Han, Fei Wu, and Jiwei Li. Description based text classification with reinforcement learning. In *International Conference on Machine Learning*, pages 1371–1382. PMLR, 2020.

[8] Haskell B Curry. The method of steepest descent for non-linear minimization problems. *Quarterly of Applied Mathematics*, 2(3):258–261, 1944.

[9] Ronald A Fisher. The use of multiple measurements in taxonomic problems. *Annals of eugenics*, 7(2):179–188, 1936.

[10] Evelyn Fix and Joseph Lawson Hodges. Discriminatory analysis. nonparametric discrimination: Consistency properties. *International Statistical Review/Revue Internationale de Statistique*, 57(3):238–247, 1989.

[11] K. Fukunaga and L. Hostetler. The estimation of the gradient of a density function, with applications in pattern recognition. *IEEE Transactions on Information Theory*, 21(1):32–40, 1975.

[12] Charles J Geyer. Practical markov chain monte carlo. *Statistical science*, pages 473–483, 1992.

[13] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *CoRR*, abs/1801.01290, 2018.

[14] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization.

[15] Teuvo Kohonen. *Self-organizing maps*, volume 30. Springer Science & Business Media, 2012.

[16] Vijay Konda and John Tsitsiklis. Actor-critic algorithms. *Advances in neural information processing systems*, 12, 1999.

[17] Stephan Lewandowsky and Shu-Chen Li. Catastrophic interference in neural networks: Causes, solutions, and data. In *Interference and inhibition in cognition*, pages 329–361. Elsevier, 1995.

[18] Aristidis Likas. A reinforcement learning approach to online clustering. *Neural computation*, 11(8):1915–1932, 1999.

[19] J MacQueen. Some methods for classification and analysis of multivariate observations. In *Proc. Fifth Berkeley Symp. Math. Stat. Probab. Vol. 1 Stat.*, pages 281–297, Berkeley, Calif., 1967. University of California Press.

[20] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016.

[21] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[22] Hossein K Mousavi, Mohammadreza Nazari, Martin Takáč, and Nader Motee. Multi-agent image classification via reinforcement learning. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5020–5027. IEEE, 2019.

[23] L. J. Reed and Joseph Berkson. The application of the logistic function to experimental data. *The Journal of Physical Chemistry*, 33(5):760–779, 1929.

[24] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

[25] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[26] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[27] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015.

[28] Sun-Chong Wang. Artificial neural network. In *Interdisciplinary computing in java programming*, pages 81–100. Springer, 2003.

[29] Ziyu Wang, Nando de Freitas, and Marc Lanctot. Dueling network architectures for deep reinforcement learning. *CoRR*, abs/1511.06581, 2015.

[30] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3):279–292, 1992.

[31] Marco A Wiering, Hado Van Hasselt, Auke-Dirk Pietersma, and Lambert Schomaker. Reinforcement learning algorithms for solving classification problems. In *2011 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*, pages 91–96. IEEE, 2011.

[32] G Udny Yule. On the theory of correlation. *Journal of the Royal Statistical Society*, 60(4):812–854, 1897.

Appendix A

Python Codes: Experiment parameters

A.1 Data Generation Parameters for Clustering Algorithms

```
from dm_gym.utils.data_gen import data_gen_clustering

n = 2 ###Number of dimentions in the data
k = 3 ###Number of clusters we want in the data
num_records = 150
parameter_means = []
parameter_sd = []

data_gen = data_gen_clustering()

error, error_code, pm, psd = data_gen.param_init(n=n, k=k,
                                    num_records=num_records,
                                    parameter_means=parameter_means,
                                    parameter_sd=parameter_sd)
data = data_gen.gen_data()
```

A.2 Data Generation Parameters for Classification Algorithms

```
from dm_gym.utils.data_gen import data_gen_classification

n = 5 ###Number of dimentions in the data
k = 3 ###Number of classes we want in the data
num_records = 1000
parameter_means = []
parameter_sd = []

data_gen = data_gen_classification()

error, error_code, pm, psd = data_gen.param_init(n=n, k=k,
                                    num_records=num_records,
                                    parameter_means=parameter_means,
                                    parameter_sd=parameter_sd)
data, target = data_gen.gen_data()
```

A.3 Ray Config parameters for clustering-v0

```
env_name = "clustering-v0"
epochs = 500
num_timesteps = 1000

env_config = { 'data': data, 'k': k}

rl_config = dict( log_level = "ERROR",
    env=env_name,
    num_workers=10,
    num_gpus=1,
    env_config=env_config,
    double_q=True,
    model=dict(
        vf_share_layers=False,
        fcnet_activation='relu',
        fcnet_hiddens=[128, 64]),
    exploration_config={
        "type": "EpsilonGreedy",
        "initial_epsilon": 1.0,
        "final_epsilon": 0.02,
        "epsilon_timesteps": 0.4*num_timesteps*epochs},
    evaluation_config={ "explore": False},
    gamma = 1,
    target_network_update_freq=500,
    buffer_size=100,
    #adam_epsilon=1e-8,
    #grad_clip=40,
    train_batch_size=32,
    framework='torch',
    lr=1e-5
)
```

A.4 Ray Config parameters for clustering-v1

```
env_name = "clustering-v1"
epochs = 500
num_timesteps = 1000

env_config = { 'data': data, 'k': k}

rl_config = dict( log_level = "ERROR",
    env=env_name,
    num_workers=10,
    num_gpus=1,
    env_config=env_config,
    double_q=True,
    model=dict(
        vf_share_layers=False,
        fcnet_activation='relu',
        fcnet_hiddens=[128, 64]),
    exploration_config={
        "type": "EpsilonGreedy",
        "initial_epsilon": 1.0,
        "final_epsilon": 0.02,
        "epsilon_timesteps": 0.4*num_timesteps*num_records},
    evaluation_config={"explore": False},
    gamma = 1,
    target_network_update_freq=500,
    buffer_size=100,
    #adam_epsilon=1e-8,
    #grad_clip=40,
    train_batch_size=32,
    framework='torch',
    lr=1e-5
)
```

A.5 Ray Config parameters for clustering-v2

```
env_name = "clustering-v2"
epochs = 500
num_timesteps = 1000
max_steps = min(num_timesteps, num_records)
env_config = { 'data': data, 'k': k,
            'max_steps': max_steps, 'lr': 0.001 }

rl_config = dict( log_level = "ERROR",
    env=env_name,
    num_workers=10,
    num_gpus=1,
    env_config=env_config,
    double_q=True,
    model=dict(
        vf_share_layers=False,
        fcnet_activation='relu',
        fcnet_hiddens=[128, 64]),
    exploration_config={
        "type": "EpsilonGreedy",
        "initial_epsilon": 1.0,
        "final_epsilon": 0.02,
        "epsilon_timesteps": 0.4*num_timesteps*epochs},
    evaluation_config={ "explore": False },
    gamma = 0.4,
    target_network_update_freq=500,
    buffer_size=100,
    #adam_epsilon=1e-8,
    #grad_clip=40,
    train_batch_size=32,
    framework='torch',
    lr=5e-5)
```

A.6 Ray Config parameters for clustering-v3

```
env_name = "clustering-v3"
epochs = 500
num_timesteps = 1000

env_config = { 'data': data, 'k': k }

rl_config = dict( log_level = "ERROR",
    env=env_name,
    num_workers=10,
    num_gpus=1,
    env_config=env_config,
    double_q=True,
    model=dict(
        vf_share_layers=False,
        fcnet_activation='relu',
        fcnet_hiddens=[128, 64] ),
    exploration_config={
        "type": "EpsilonGreedy",
        "initial_epsilon": 1.0,
        "final_epsilon": 0.02,
        "epsilon_timesteps": 0.4*num_timesteps*epochs },
    evaluation_config={ "explore": False },
    gamma = 0.4,
    target_network_update_freq=500,
    buffer_size=100,
    #adam_epsilon=1e-8,
    #grad_clip=40,
    train_batch_size=32,
    framework='torch',
    lr=5e-5
)
```

A.7 Ray Config parameters for classification-v0

```
env_name = "classification-v0"
epochs = 500
num_timesteps = 1000

env_config = { 'data': X_train, 'target': y_train, 'num_classes': k }

rl_config = dict( log_level = "ERROR",
    env=env_name,
    num_workers=10,
    num_gpus=1,
    env_config=env_config,
    double_q=True,
    model=dict(
        vf_share_layers=False,
        fcnet_activation='relu',
        fcnet_hiddens=[128, 64] ),
    exploration_config={
        "type": "EpsilonGreedy",
        "initial_epsilon": 1.0,
        "final_epsilon": 0.02,
        "epsilon_timesteps": 0.4*num_timesteps*epochs },
    evaluation_config={ "explore": False },
    gamma = 0.4,
    target_network_update_freq=500,
    buffer_size=100,
    #adam_epsilon=1e-8,
    #grad_clip=40,
    train_batch_size=32,
    framework='torch',
    lr=1e-5
)
```

Appendix B

License

BSD 3-Clause License