Chapter 1: Introduction

Reinforcement learning (RL) is one of the phenomenal applications of deep learning, which is about an agent interacting with the environment. RL algorithms can make sequential decisions and understand from their experience. These characteristics differentiate them from traditional machine learning models. Reinforcement learning is a powerful tool, and it has a wide range of applications, such as computer games, industrial automation, robotics, traffic control systems, chatbots, self-driving cars, etc [4].

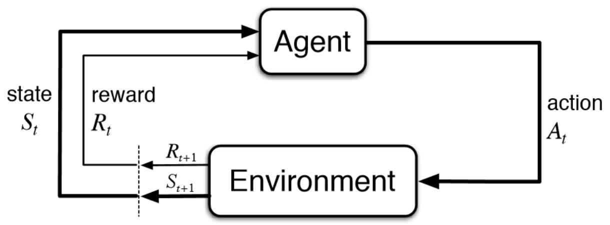
Chapter one solves the first part of the assignment, the OpenAI Gym Mountain Car environment problem utilising RL algorithms. More details about the Epsilon Greedy and Boltzmann Policies for selecting actions in Deep Q-Learning will be discussed.

Chapter two focuses on the second part of the assignment. The sequence networks exploiting GloVe embeddings and Word2Vec will be developed to predict the ratings of customer reviews from Trip Advisor. More detail about the difference between embeddings will be considered.

The Python programming language with its libraries has been selected to implement the project. Numpy, Pandas, Scikit-learn, TensorFlow, Gensim, Keras, Keras-rl2 module, and Plotly are the core libraries and packages that have been utilised to implement the project.

Chapter 2: Reinforcement learning

Reinforcement learning is machine learning training method that uses rewards and punishments as indications for positive and negative behaviour. Learning occurs as an agent perceives and interprets the environment, takes actions, and learns by trial and error. Figure 2.1 illustrates the feedback loop for action and reward in generic RL model.



2.1 Epsilon Greedy

As the idea of a random selection of actions is not an optimal solution, it is better to learn by exploration. The Epsilon-Greedy Algorithm is an exploration strategy that takes an exploratory effort with probability epsilon (ϵ) and a greedy action with probability 1−ϵ. It tackles the exploration-exploitation trade-off by instructing the computer to investigate (i.e., choosing a random option with probability epsilon). Next, it exploits (i.e., select the option that seems to be the best) the remainder of the time. This way, as the model continues to choose different options, it will be able to determine which choices will provide the best reward.

2.2 The Mountain Car Problem

One of the interesting OpenAI Gym environment is Mountain Car problem [] which need to be solve using Q-learning in the first part of the assignment. Figure 2.2 illustrates the car which starts in between two hills. The car must gain momentum by going backward and forward until it reaches the flag with enough speed. In every state, there are only three possible actions. The vehicle may accelerate forward, reverse, or stay in the same place.

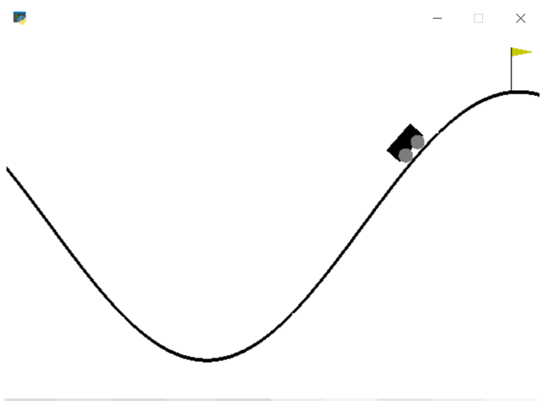
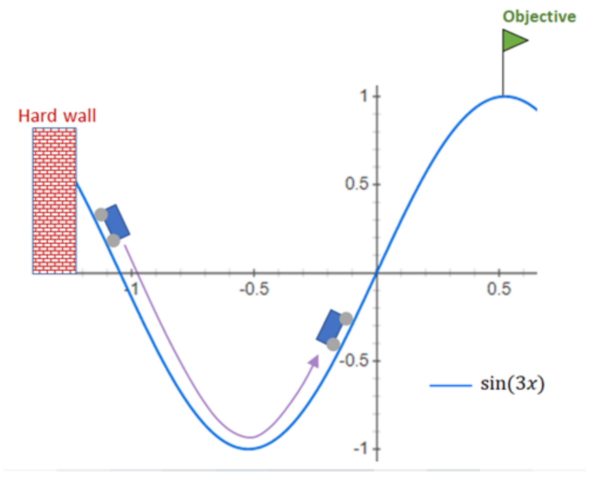


Figure 2.3 illustrates the car position in x-axis range [-1.2, 0.6]. the vehicle reaches the target at 0.5 x-axis.



2.2.1 Mountain Car Experiment One

This experiment will solve the problem using only Q-Learning with the Epsilon Greedy strategy, which encountered many challenges that need to be mentioned.

Firstly, numerous trials and errors revealed that the number of states must be manually discretised. According to the Frozen Lake workshop example in class, there are only 16 state numbers. However, in the Mountain Car problem, the X values are not discrete. Consequently, code 2.1 overcome this challenge.

```
# Determine size of discretized state space
no_states = (env.observation_space.high - env.observation_space.low)*\
                np.array([10, 100])
no_states = np.round(no_states, 0).astype(int) + 1

print('Number of State= ', no_states )
```

Code 2.1:Discritized the State Space

Secondly, Code 2.2 populates the Q-table with the uniform distribution because the environment space X-axis range is [-1.2 -> 0.6]; hence the distribution is considered from -1 to +1.

```
q_table = np.random.uniform(low = -1, high = 1,
                        size = (no_states[0], no_states[1],
                                env.action_space.n))
```

Code 2.2: Define Q-table

Thirdly, code 2.3 enables Epsilon Greedy Algorithm to decide whether to keep exploring a new random position or exploit the Q-table to choose a corresponding position with the best reward. The dynamic epsilon reduces from its initial high value as the episodes progress. At the beginning of the game, there is more likelihood of exploration, and at the higher levels, there is a greater likelihood of exploitation or greedy behaviour.

```
def eps_greedy(current_state,step_no,trainingOn):

    eps = max(min_eps,  max_eps -  eps_a*step_no)

    if np.random.uniform() < eps and trainingOn:
        action = np.random.randint(0, no_actions)
    else:
        action = np.argmax(q_table[current_state[0],current_state[1]])
    return action
```

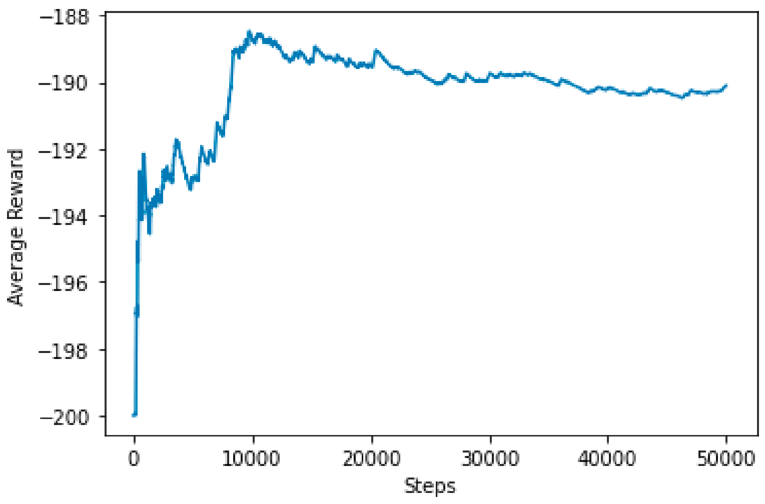
Code 2.3: Define Epsilon Greedy Function

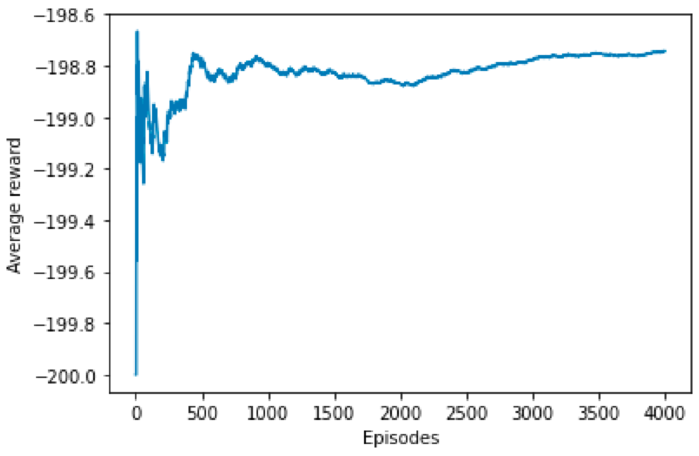
Table 2.1 represent the essential hyperparameter that have been set for this experiment.

Table 2.1: Mountain Car Q-Leaning Hyperparameter

Parameter	Value
Learning Rate Alpha	0.6
Discount factor Gamma	0.99
Number of Episodes	50000
Starting Epsilon	0.9

Figures 2.4 and 2.5 demonstrate the trend of the average rewards in training and testing process. It reached -186 after 10000 steps in training process. However, after this point, agent learning performance reduced slightly over each episode. Unfortunately, game performance was poor in testing as well. The most problematic was that it always rewards negatively, regardless of the random action taken. Hence, a different approach will be used to improve learning performance in the next step.





2.3 Deep-Q-learning Network (DQN)

Deep-Q learning is a mixture of deep learning and reinforcement learning. Neural networks are used to estimate the Q-values of all possible actions in each state.

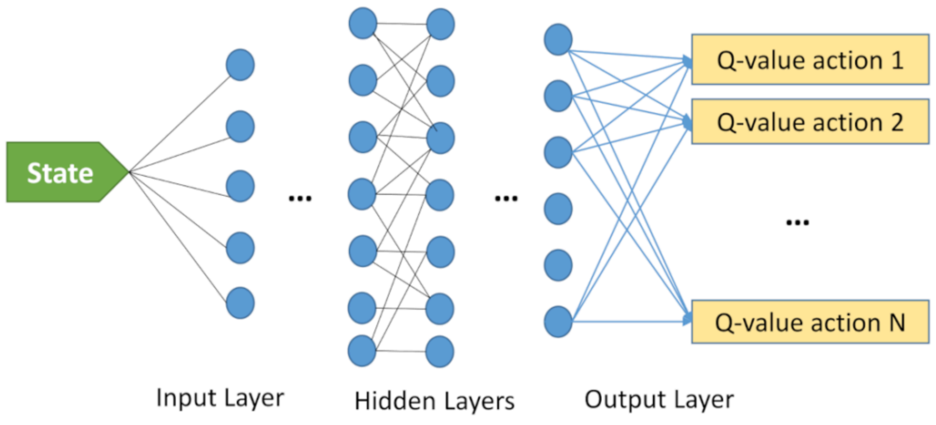
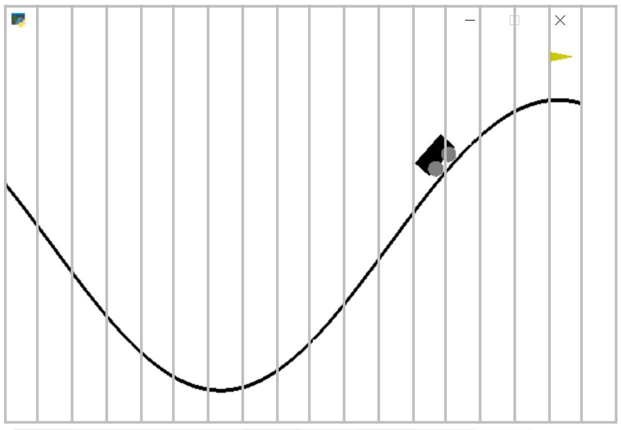


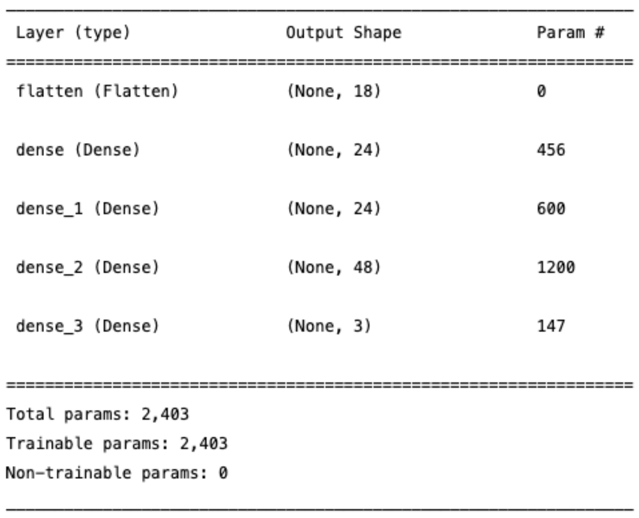
Figure 2.6 illustrates the Deep Q-Learning architecture design. Firstly, the input data need to be feed to the model in hot encoded format. Next the hidden layers minimise the error estimated by the loss function by optimising the weights.

2.3.1 Mountain Car Experiment Two

In this experiment, the problem will be solved utilising DQN learning. Therefore, the input data, which is the state of the car in the Gym environment space, need to be pre-processed. To interpret the vehicle's position, which by default is continues values, the X-axis area has been divided into 18 parts. Figure 2.7 illustrates the division area. Consequently, the network input data will have 18 values or states.



The experiment two DQN sequential network model consists of five layers. The first one is only flattening the input data, and the last layer will drive the output, which has three neurons representing three actions (accelerate forward, reverse, or stay in the same place). All extra details of the model summary, such as the total parameter numberers, can be seen in Figure 2.8.



```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten (Flatten)           (None, 18)                0

 dense (Dense)               (None, 24)                456

 dense_1 (Dense)             (None, 24)                600

 dense_2 (Dense)             (None, 48)                1200

 dense_3 (Dense)             (None, 3)                 147

=================================================================
Total params: 2,403
Trainable params: 2,403
Non-trainable params: 0
_________________________________________________________________
```

Figure 2.8: Experiment Two DQN Model Summary

One of the biggest challenges of the problem is that the Gym environment reward is not as expected, and experiment one outcome proved that the problem requires a more sophisticated solution. Looking into Mountain Car environment documentation [3], the agent penalised with a reward of -1 for each timestep is not at the goal and is not penalised (reward = 0) for when it reaches the destination. Another approach will be tested by increasing the reward to two when the car climbs a steep hill to reach the flagged goal.

There is a need to create a link between the gym environment and the agent to accomplish this. Code 2.4 defines a MountainCarProcessor class which subclass of Processor will be given as a parameter. Two methods construct a connection. The first one is to change the state or position of the car into 16 possible hot codding formats. The process_observation method takes the observation and returns corresponding hot codding. The fundamental key is to translate the state before it feeds into a neural network.

The second method, called process_reward, adjusts the reward returned from the environment as a parameter and does the calculation, and returns a new reward. This function is crucial to boost the average reward in this assignment.

```python
class MountainCarProcessor(Processor):
    def process_observation(self, observation):
        #print(observation)
        one_hot = np.zeros(18)

        # identify the index of observation state in x-axis:
        i = int(np.round(( observation[0] + 1.2 / 0.1 )))
        one_hot[i] = 1
        return one_hot

    def process_reward(self, reward):
        if (env.state[0] >= 0.5):
            new_reward = 2
        else:
            new_reward = (env.state[0] + 1.2) / 1.8 - 1
        return new_reward
```

Code 2.4: MountainCarProcessor Class Implementation

Policy

EpsGreedyQPolicy from the Keras-rl package has been used in this experiment. The main reason is to balance exploration and exploitation by randomly choosing between exploration and exploitation. The detail of the policy strategy has been explained in section 2.2.

Code 2.5 defines LinearAnnealedPolicy that calculates a threshold value that decreases linearly over the steps and passes this value to EpsGreedyQPolicy. Additionally, extra hypermeter such as epsilon value represent the probability of exploring the other random options. The EpsGreedyQPolicy will choose the arbitrary action or use Q values to find the best move.

DQN agent

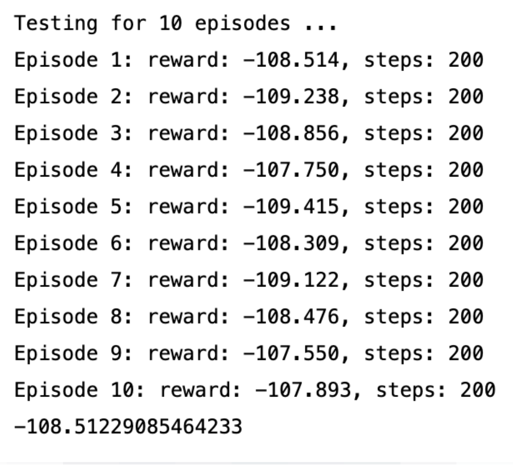
Lastly, the DQN agent has been created and referenced in code 2.5. The memory has been specified to create the sequential memory with 100k steps.

```
memory = SequentialMemory(limit= 100000, window_length= 1 )

processor = MountainCarProcessor()

policy = LinearAnnealedPolicy(EpsGreedyQPolicy(), attr='eps',
                              value_max=1., value_min=.1, value_test=.05,
nb_steps=10000)

dqn = DQNAgent(model=model, nb_actions= env.action_space.n, processor = processor ,
               memory= memory, nb_steps_warmup=100, gamma=0.99, policy=policy,
               enable_double_dqn= True,target_model_update= 1e-3 )
```

Code 2.5: Defining DQN Agent with EpsGreedyQPolicy

Result

The agent performance reached an average reward of -117.63 in the training process and almost - 108.51 in the testing process on ten episodes. Figure 2.9 represents the reward values in each episode.



```
Testing for 10 episodes ...
Episode 1: reward: -108.514, steps: 200
Episode 2: reward: -109.238, steps: 200
Episode 3: reward: -108.856, steps: 200
Episode 4: reward: -107.750, steps: 200
Episode 5: reward: -109.415, steps: 200
Episode 6: reward: -108.309, steps: 200
Episode 7: reward: -109.122, steps: 200
Episode 8: reward: -108.476, steps: 200
Episode 9: reward: -107.550, steps: 200
Episode 10: reward: -107.893, steps: 200
-108.51229085464233
```

2.3.2 Mountain Car Experiment Three

In this experiment, the Mountain Car problem will be solved utilising DQN learning, same as experiment two; however, the only modification is the Boltzmann exploration policy will be employed to select the best action in each step.

Boltzmann policy

Despite Epsilon's greedy policy always taking a random or optimal action to select the next move, the BoltzmannQPolicy works differently by choosing an activity with weighted probabilities. As a result, each movement is estimated using a SoftMax algorithm over the network. It means that the agent will likely choose the action that it counts to be the optimal one (but not necessarily). Compared to the e-greedy algorithm, this policy can also incorporate information about the likely value of other actions. [1] Dua et al. clarifies the difference between two policies in the OpenAI Frozen Lake example[2]. As four steps available to an agent, in e-greedy, the three actioned are estimated to be non-optimal, and all considered equally. However, in the Boltzmann exploration, they are weighted by their relative values. With this approach, the agent can ignore actions that it estimates to be sub-optimal and give more attention promising, but necessarily ideal actions.

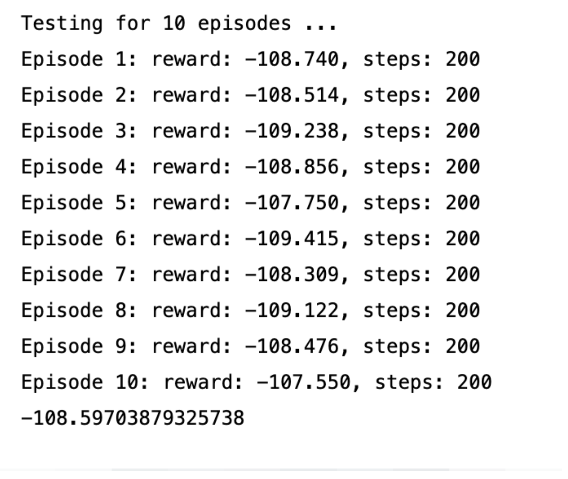
Code 2.6 creates the DQN agent with specific hyperparameters that have been declared in the code.

```
memory = SequentialMemory(limit= 100000, window_length= 1 )

processor = MountainCarProcessor()

policy = BoltzmannQPolicy()

dqn = DQNAgent(model=model, nb_actions= env.action_space.n, processor = processor ,
               memory= memory, nb_steps_warmup=100, gamma=0.99, policy=policy,
               enable_double_dqn= True,target_model_update= 1e-3 )
```

Code 2.6: Defining DQN Agent with Boltzmann policy

Result

The agent performance reached an average reward of -124.97 in the training process and almost - 108.59 in the testing process on ten episodes. Figure 2.10 represents the reward values in each episode.



```
Testing for 10 episodes ...
Episode 1: reward: -108.740, steps: 200
Episode 2: reward: -108.514, steps: 200
Episode 3: reward: -109.238, steps: 200
Episode 4: reward: -108.856, steps: 200
Episode 5: reward: -107.750, steps: 200
Episode 6: reward: -109.415, steps: 200
Episode 7: reward: -108.309, steps: 200
Episode 8: reward: -109.122, steps: 200
Episode 9: reward: -108.476, steps: 200
Episode 10: reward: -107.550, steps: 200
-108.59703879325738
```

2.4 Comparison

Table 2.2 presents the comparison of the evaluation indexes in three experiments. It is evident from the second and third experiments that using Neural Networks in Reinforcement Learning can obtain the best performance. Both agents achieved average awards of -108.51 in the testing process. However, in more detail, the DQN agent with Epsilon Greedy Policy gained the best average rewards of -117.626 in training for climbing the mountain. Also, from Experiment one, it can be understood that the Mountain Car problem cannot be solved by Q-Reinformance learning itself, and it needs more sophisticated solutions with Neural Network contribution.

Table 2.2: Mountain Car Experiments Comparison

#	**Experiments**	**Average Reward**	
		Train	Test
1	Q-Learning	-190.111	-198.742
2	DQN with Epsilon Greedy Policy	-117.626	-108.51
3	DQN with Epsilon Boltzmann Policy	-124.974	-108.597

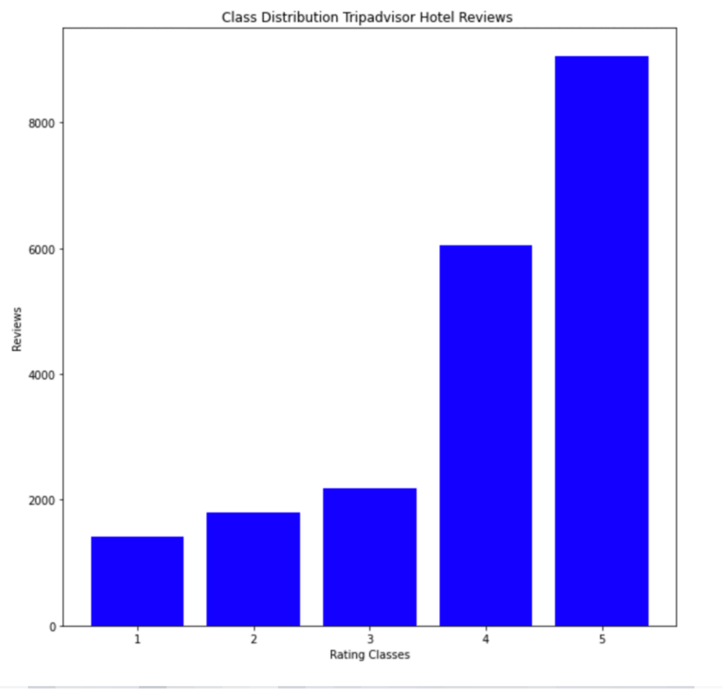
Comprehensibly, DQN has more potential to train agents. Unfortunately, as the deadline is approaching, I could not assign more time to enhance the neural network and tweak the hyperparameters.

Chapter 3: Sequence Learning and Embeddings

As Natural Language Processing (NLP) texts are random strings, building a sequential model can be challenging. A word embedding is a technique that represents words as real-valued vectors in the context of a predefined space of vectors. Different embeddings such as Word2Vec [5] and GloVe [6] have been designed for the text, transforming a word into an n-dimensional vector. This chapter's primary focus is to employ these embeddings to build models to predict the ratings of user reviews. The Trip Advisor Review dataset will be used in this section.

3.1 Data Preparation

The Trip Advisor Hotel Reviews dataset has been utilised to solve part B of the assignment. The dataset consists of 20,491 samples of review text and their ratings.
The distribution of rating class is demonstrated in Figure 3.1. The classes are fairly imbalance.



3.2 Word2Vec Embeddings

A shallow neural network has been used to create one of the most famous techniques to learn and understand word embeddings called Word2Vec. It was developed by Tomas Mikolov in 2013 at Google[5].

The word embedding has been generated by employing two different learning models. One of them is the Continuous Bag-of-Words or CBOW model, and the other is the Continuous Skip-Gram Model.

The CBOW model understands the embedding by forecasting the current word based on its context. The continuous skip-gram model comprehends by forecasting the surrounding words given an existing word. A window of nearby words defines the context in both models, and each model focuses on learning about words based on their local use. The window is a hyperparameter that can be tweaked.

3.3 GloVe Embeddings

Pennington, et al. at Stanford developed the Global Vectors for Word Representation, or GloVe, an extension to the word2vec method for learning word vectors efficiently[6].

Latent Semantic Analysis (LSA) is an effective way to use global text statistics when developing classical vector space model representations of words has been used in this approach. However, in some cases, like measuring analogies, it does not understand the meaning as well as Word2Vec.

GloVe aims to match both global statistics derived from matrix factorization techniques such as LSA and context-based learning found in Word2Vec. As a substitute for operating a window to determine local context, GloVe creates an explicit matrix of word contexts, or co-occurrences, across the entire text corpus. As a result, better word embeddings have been achieved.

3.4 Modelling

The entire dataset has been divided into 18441 samples for training and 2050 samples for testing models via the train_test_split method from Sklearn.model_selection python library. Also, most of the ratings are 5-star, and the least are 1-star ratings. Thus, stratify option has been considered in the division process. It will ensure that the distribution of all the classes is even between the train, validation, and test set.

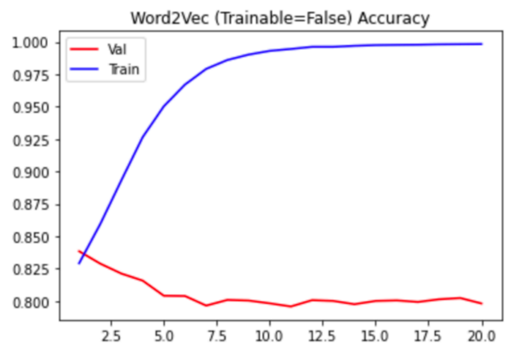
Different networks such as LSTM and RNN have experimented with Word2Vec and GloVe embeddings. In total, 20 epochs were used to train the network models, with 1894 as the maximum length of the review. Several evaluation metrics were employed to evaluate the efficiency of the models. More details will be discussed in the next section.

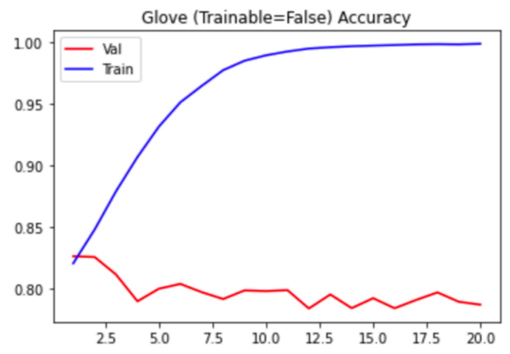
3.5 Comparison

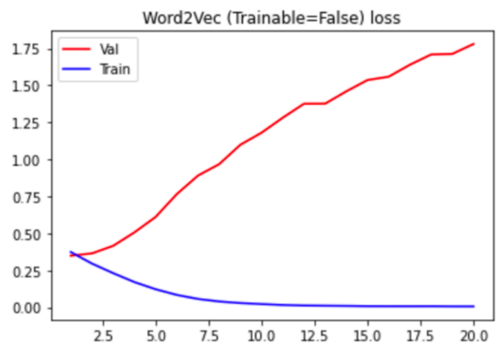
Table 3.1: Comparison of models evaluating result

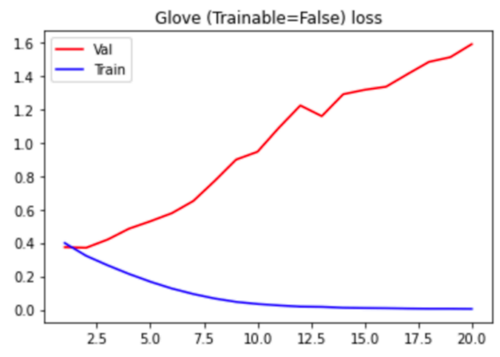
#	Model Name	Embeddings	Evaluation Metrics			
			Train Accuracy	Validation Accuracy	Train Loss	Validation Loss
1	Sequential Network	Word2Vec	0.9984	0.7981	0.0061	1.7800
2	LSTM (Trainable = false)	Word2Vec	0.724	0.6346	0.078	0.0986
3	LSTM (Trainable = True)	Word2Vec	0.9206	0.6034	0.0282	0.1444
4	Sequential Network	GloVe	0.9990	0.7870	0.0048	1.5895
5	LSTM (Trainable = false)	GloVe	0.7315	0.6288	0.6288	0.0998
6	LSTM (Trainable = True)	GloVe	0.9241	0.5932	0.0274	0.1492
7	RNN Network	GloVe	0.4456	0.4434	0.1351	0.1340

Table 3.1 compares the evaluation indexes of the seven models during the training and testing process. The highest validation accuracy is 0.7981, reached with Sequential Network using Word2Vec embeddings. Nevertheless, identical network using GloVe embeddings perform slightly better in the training process. Figures 3.1 and 3.2 illustrate that both embeddings have similar performance trends. Yet, there is an indication of exponential growth in loss which could be a sign of overfitting the models.

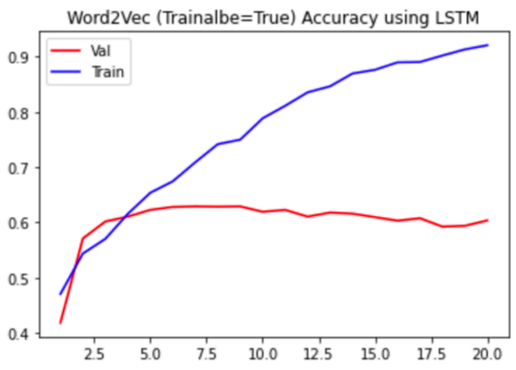


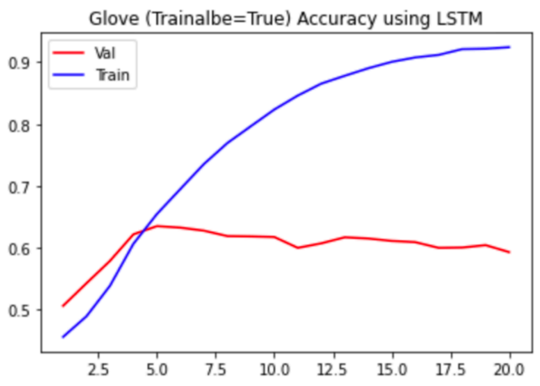


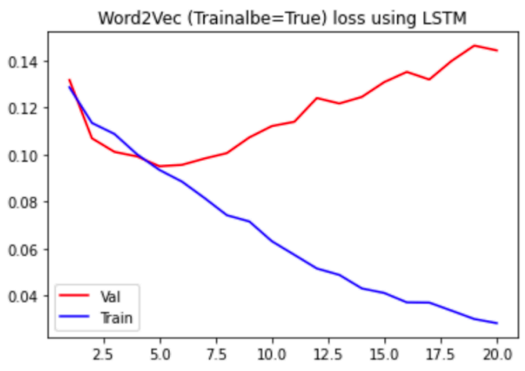


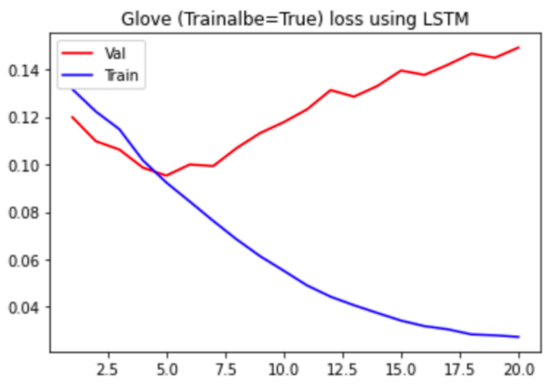


The lowest loss is 0.0986 achieved by LSTM (Trainable = false) network model; regardless, its performance on validation accuracy is not as expected. Figures 3.3 and 3.4 demonstrate the training and testing trend clearly and clearly that the validation trend aligns with training up to epoch five and after it begins overfitting the model.



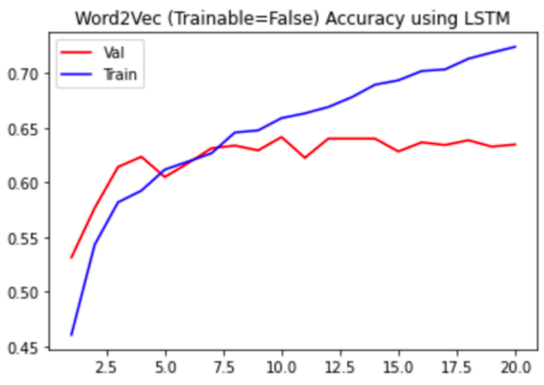


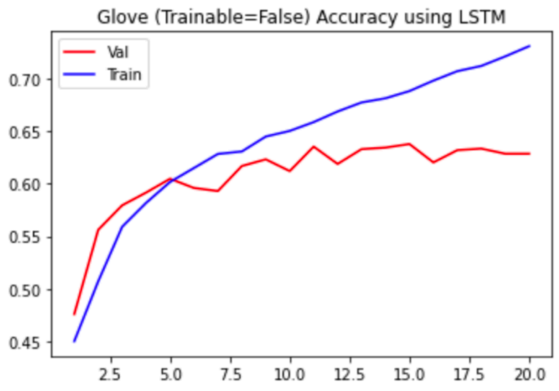


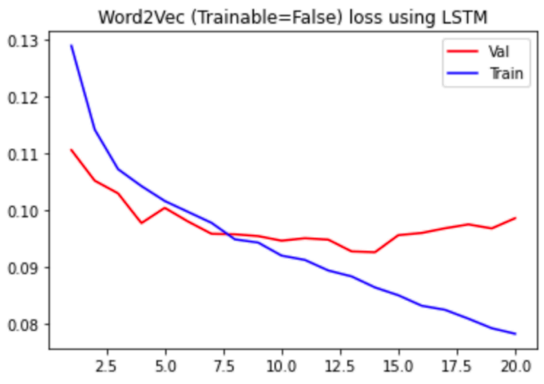


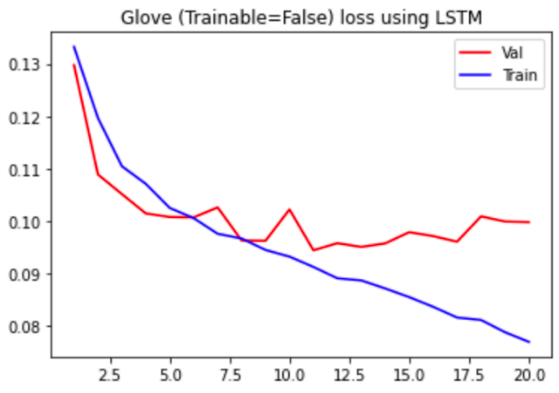
Figures 3.6 and 3.6 reveal that LSTM (Trainable = True) networks using Word2Vec and GloVe embeddings have almost equivalent performance. However, the building structure of the GloVe is claimed to be more sophisticated than Word2Vec, but in a practical sense, it has been evidenced that it has a similar performance to Word2Vec.

As the RNN network in experiment seven result is insufficient, it has not experimented further.









Chapter 4: Conclusions

As a final note, the core concepts of the deep learning approach, such as simplicity, scalability, and reusability, have been experienced in the developing process of this project. It demonstrated that blindingly adding novel layers to the neural network is not beneficial, and it can also increase the cost.

On the other side, using pre-trained embeddings can be beneficial, such as less training and less effort in building model architecture. However, it needs to be mentioned there is always a risk of transferring a bias from the pre-trained models.

Investigating worst cases classification in assignment's part B is worth spending time enhancing the exciting models.

Chapter 5: Reference:

[1] Dua, R., & Ghotra, M. S. (2018). Keras Deep Learning Cookbook: Over 30 recipes for implementing deep neural networks in Python. Packt Publishing Limited.

[2] gym/frozen_lake.py at master · openai/gym. (n.d.). Github. Retrieved May 6, 2022, from https://github.com/openai/gym/blob/master/gym/envs/toy_text/frozen_lake.py

[3] gym/mountain_car.py at master · openai/gym. (n.d.). Github. Retrieved May 15, 2022, from https://github.com/openai/gym/blob/master/gym/envs/classic_control/mountain_car.py

[4] Li, Y. (2017). Deep reinforcement learning: An overview. arXiv preprint arXiv:1701.07274.

[5] Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*

[6] Pennington, J., Socher, R., & Manning, C. D. (2014, October). Glove: Global vectors for word representation. In Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP) (pp. 1532-1543)