Name:	Jack Travis
ID:	---
Username:	---
Module Name:	Deep Learning
Assessment Title:	Assignment 2: Reinforcement Learning and Sequence nets
CRN:	56630

Contents

1 – Reinforcement Learning

1.1 – What is Deep Q-Learning

Deep Q-learning is a method of reinforcement learning that approximates the state-value of a Q function that represents all action values in all states, by utilizing a neural network. (Hu, Wang and Yu, 2021) Over many iterations, this function will begin to calculate the optimal solution for the given problem. This process is conducted via trial and error, receiving rewards and punishments for doing the correct or wrong thing respectively.

The concept of Q-Learning was first proposed by C. Watkins in 1989 (Kröse, 1995), which saw a successful implementation of it playing Atari 2600 games in 2014 (Watkins, 2014).

1.2 – The issues with Deep Q-Learning

Deep Q-learning does suffer for a few issues, some of which make its application less favoured and others that can negatively affect its performance. This section details some of these issues.

One issue that DQL suffers from is its computational heavy nature, leading to long learning times that may not be suitable for every situation (Pina et al., 2021). This is due to the nature of Q-learning, requiring the reward for every possible combination of state and actions choices to be stored and calculated. (Kumar, 2020) Deep Q-learning does alleviate some of this computational heaviness by estimating Q-values. However, it still remains computationally heavy.

Another issue of DQL is its tendency to create unrealistically high action values (Kumar, 2020). This is due to a maximisation step which often leads to overestimating the action values, which are favoured over underestimated values. This effect occurs non-uniformly, generally effecting values of which more information has been gathered more severely. This will affect the chances of which decision is chosen, since the values are now more weighted than they should be. This sometimes can be beneficial or have no performance impact. However, sometimes it has a negative effect, making DQL less consistent during training.

These potential inconsistencies when training, along with long computational periods, may see this solution as less favourable, preferring a more reliable or faster solution.

1.3 – Double Deep Q-Learning

Double Deep Q-Learning was introduced by (van Hasselt, Guez and Silver, 2015) and is an adaptation of Deep Q-Learning, with its core functionality remaining the same. However, a second value function, with its own neural network, is utilized, changing the max operation into two parts: The Deep Neural Network and a Target Network.

Like with DQL, the Deep Q Network is used to calculate the values of each action, with a greedy policy opting for the one with the greatest value. However, the Target Network is also used to estimate the Q-value of the selected action. The value created by the Target Network is used to update Deep Q Network.

This decomposition of the max function reduces overestimations, creating more realistic values. Since the Target Network isn't affected by actions that have more information, any overestimations that are calculated occur more uniformly, reducing the impact said overestimates have. This results in a more consistent outcome for this network, potentially leading to improved performance and decreased training time (van Hasselt, Guez and Silver, 2015).

1.4 – An empirical evaluation of Deep Q-Learning and Double Deep Q-Learning

1.4.1 – Overview of the task

To evaluate both of these models, Taxi was used on Gym (www.gymlibrary.dev, n.d.). Taxi is a static map, split into a 5x5 grid, whereby the goal is to pickup the passenger and drop them off at their desired location. The person may spawn on any one of four colour coded tiles, with their destination being a different one of these tiles.

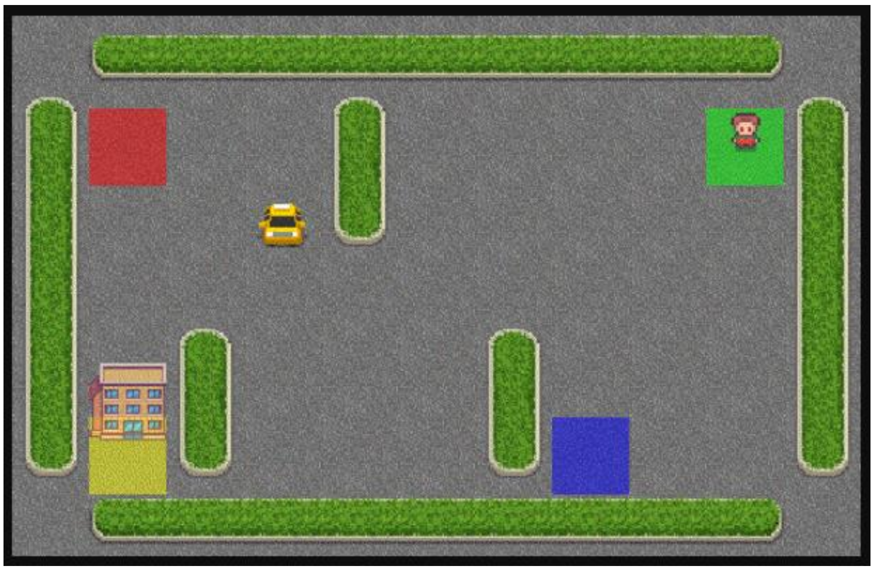


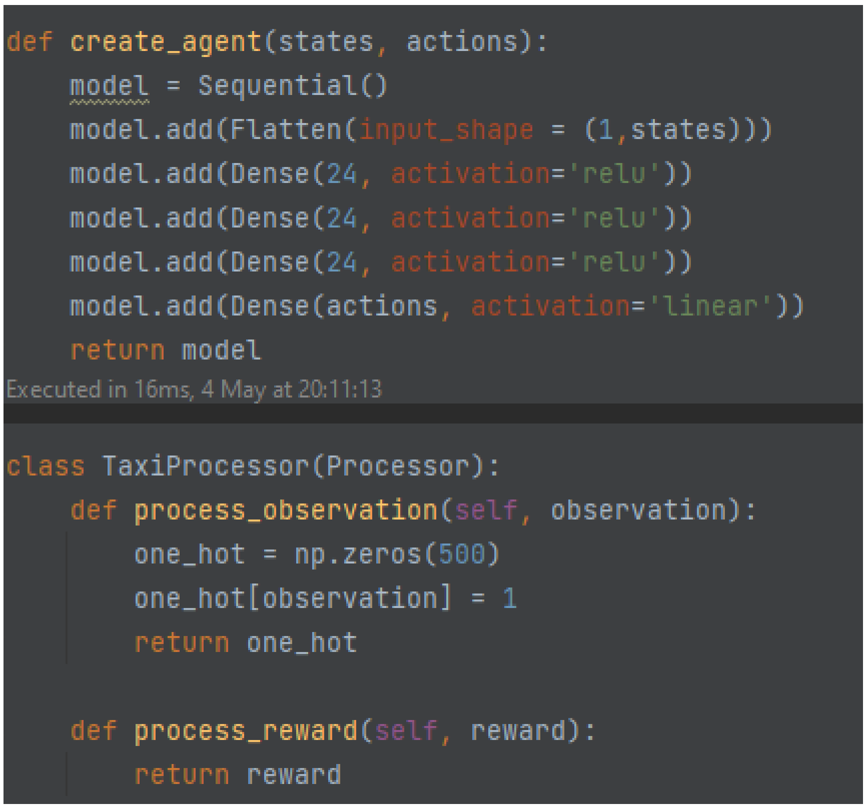
Figure 1: A physical representation of the taxi game (www.gymlibrary.dev, n.d.).

The taxi is spawned in a random position every episode, and has the goal to complete its objectives in the least amount of moves as possible. The network, which controls the taxi, has a choice of 6 actions every step, creating 500 possible states. It is rewarded for successfully dropping off the passenger, and penalised for every step taken that doesn't reward a point, or if an illegal attempt to pick up or drop off the passenger is made.

Successfully completing its objectives in less than 20 moves will result in a positive score. The maximum score that can be gained each round, obtained by taking the optimal path, ranges between 4 and 15 points, depending on the cars start position and the passengers start position and desired destination.

1.4.2 – Development of the Deep Q-Learning Network

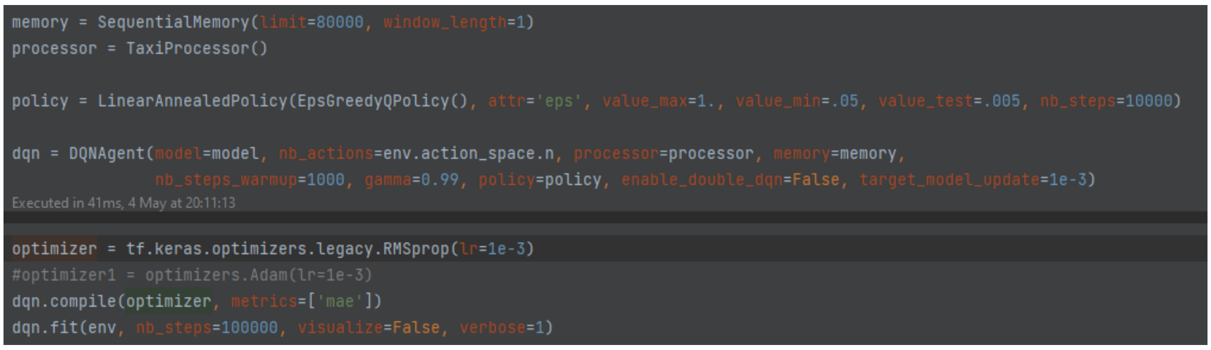
To begin with, I created the agent and Processor that is shown in the image below:



```
def create_agent(states, actions):
    model = Sequential()
    model.add(Flatten(input_shape = (1,states)))
    model.add(Dense(24, activation='relu'))
    model.add(Dense(24, activation='relu'))
    model.add(Dense(24, activation='relu'))
    model.add(Dense(actions, activation='linear'))
    return model
Executed in 16ms, 4 May at 20:11:13

class TaxiProcessor(Processor):
    def process_observation(self, observation):
        one_hot = np.zeros(500)
        one_hot[observation] = 1
        return one_hot

    def process_reward(self, reward):
        return reward
```

Figure 2: The Agent and Processor.

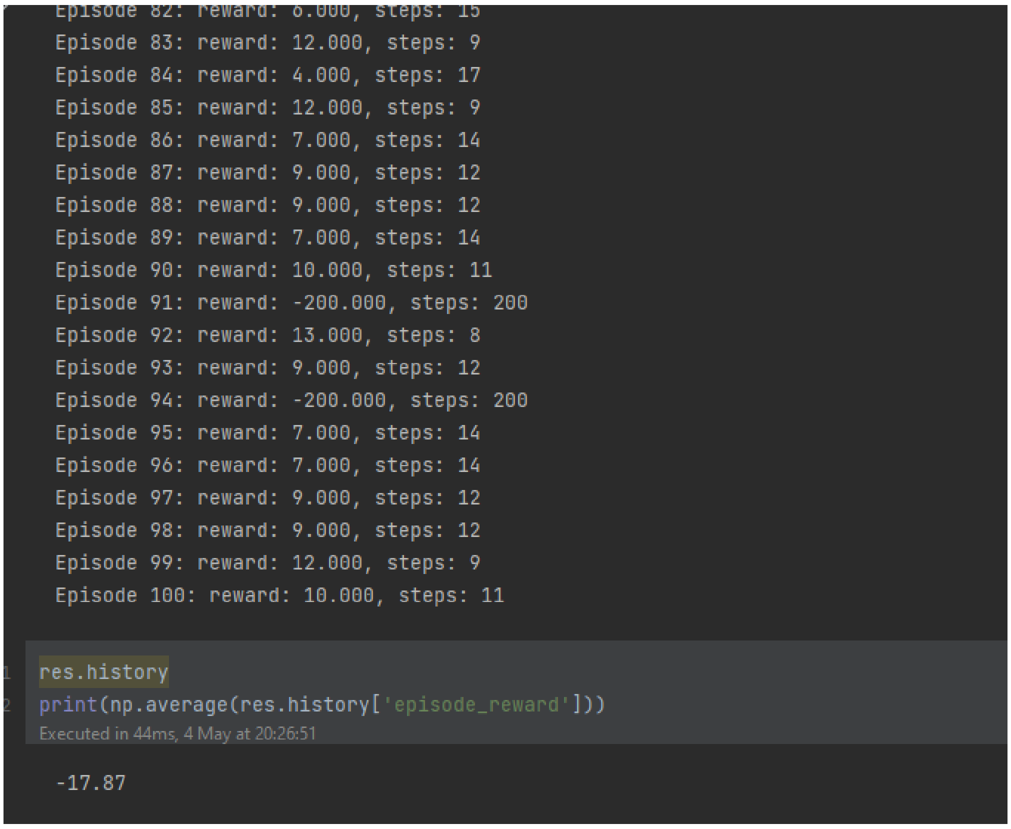
The used model structure was chosen due to its relative success in a different project, which will allow for a good starting ground for the development of this network. I then employed the LinearAnnealedPolicy and the EpsGreedyPolicy, in combination with the RMSprop optimiser:



```
memory = SequentialMemory(limit=80000, window_length=1)
processor = TaxiProcessor()

policy = LinearAnnealedPolicy(EpsGreedyQPolicy(), attr='eps', value_max=1., value_min=.05, value_test=.005, nb_steps=10000)

dqn = DQNAgent(model=model, nb_actions=env.action_space.n, processor=processor, memory=memory,
               nb_steps_warmup=1000, gamma=0.99, policy=policy, enable_double_dqn=False, target_model_update=1e-3)
Executed in 41ms, 4 May at 20:11:13

optimizer = tf.keras.optimizers.legacy.RMSprop(lr=1e-3)
#optimizer1 = optimizers.Adam(lr=1e-3)
dqn.compile(optimizer, metrics=['mae'])
dqn.fit(env, nb_steps=100000, visualize=False, verbose=1)
```

Figure 3: The policy, DQNAgent instantiation and optimiser.

A DQNAgent was also utilized, to create the Deep Q-Learning network, and it was run over 100,000 steps. However, this didn't perform well, producing a result of -200 points. This was found to be due to RMSprop not being a suitable optimizer for this model. Instead, Adam was used, yielding the following results:

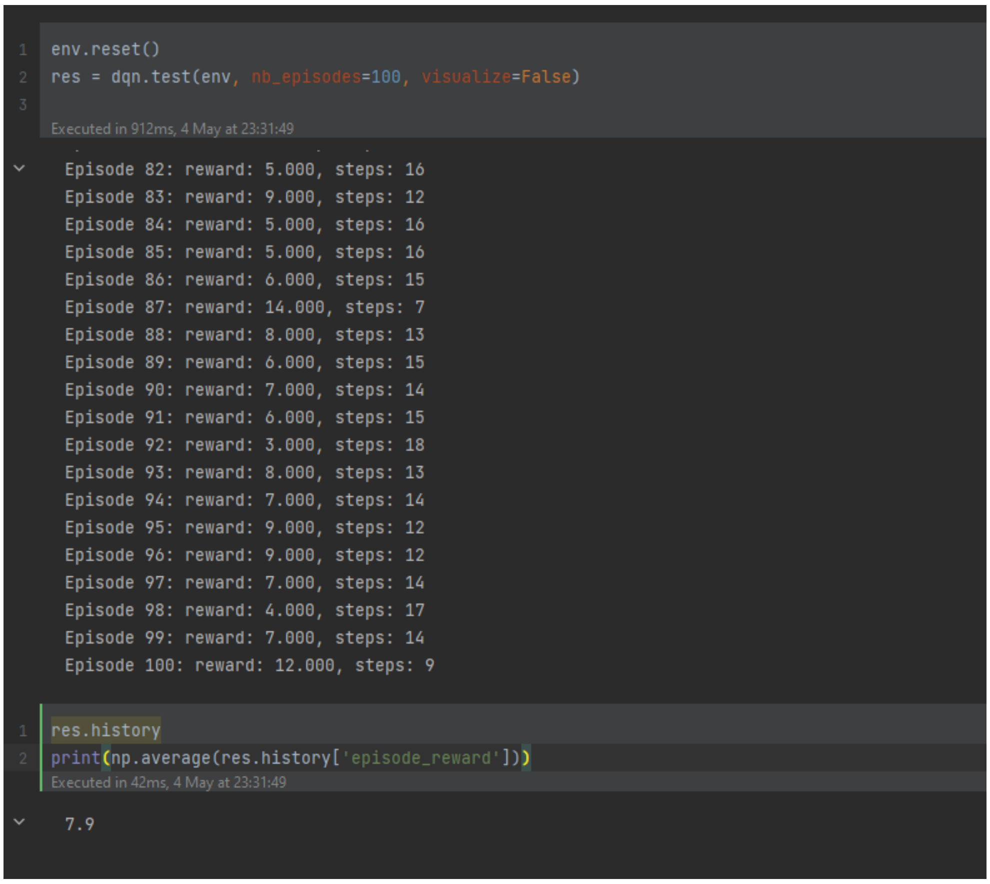


```
Episode 82: reward: 6.000, steps: 15
Episode 83: reward: 12.000, steps: 9
Episode 84: reward: 4.000, steps: 17
Episode 85: reward: 12.000, steps: 9
Episode 86: reward: 7.000, steps: 14
Episode 87: reward: 9.000, steps: 12
Episode 88: reward: 9.000, steps: 12
Episode 89: reward: 7.000, steps: 14
Episode 90: reward: 10.000, steps: 11
Episode 91: reward: -200.000, steps: 200
Episode 92: reward: 13.000, steps: 8
Episode 93: reward: 9.000, steps: 12
Episode 94: reward: -200.000, steps: 200
Episode 95: reward: 7.000, steps: 14
Episode 96: reward: 7.000, steps: 14
Episode 97: reward: 9.000, steps: 12
Episode 98: reward: 9.000, steps: 12
Episode 99: reward: 12.000, steps: 9
Episode 100: reward: 10.000, steps: 11

res.history
print(np.average(res.history['episode_reward']))
Executed in 44ms, 4 May at 20:26:51

-17.87
```

Figure 4: The results of the first version of DQL.

As figure 4 shows, an average reward of -17.87 points was acquired, based off the trained model. Following this, many changes were experimented with the agent model, such as changing the number of dense layers and the quantity of nodes they have, and changing the policy, utilizing versions of the BoltzmannQPolicy and the GreedyQPolicy. However, these changes resulted in a negative impact on performance, resulting in said changes being reverted.

After further experimentation, the best results were obtained by altering the DQNAgents target_model_update parameter to 5e-4, and altering the amount of steps to 250,000. Below shows the results of that network, when tested with 100 episodes:



```
env.reset()
res = dqn.test(env, nb_episodes=100, visualize=False)

Executed in 912ms, 4 May at 23:31:49
```

```
Episode 82: reward: 5.000, steps: 16
Episode 83: reward: 9.000, steps: 12
Episode 84: reward: 5.000, steps: 16
Episode 85: reward: 5.000, steps: 16
Episode 86: reward: 6.000, steps: 15
Episode 87: reward: 14.000, steps: 7
Episode 88: reward: 8.000, steps: 13
Episode 89: reward: 6.000, steps: 15
Episode 90: reward: 7.000, steps: 14
Episode 91: reward: 6.000, steps: 15
Episode 92: reward: 3.000, steps: 18
Episode 93: reward: 8.000, steps: 13
Episode 94: reward: 7.000, steps: 14
Episode 95: reward: 9.000, steps: 12
Episode 96: reward: 9.000, steps: 12
Episode 97: reward: 7.000, steps: 14
Episode 98: reward: 4.000, steps: 17
Episode 99: reward: 7.000, steps: 14
Episode 100: reward: 12.000, steps: 9
```

```
res.history
print(np.average(res.history['episode_reward']))
Executed in 42ms, 4 May at 23:31:49
```

```
7.9
```

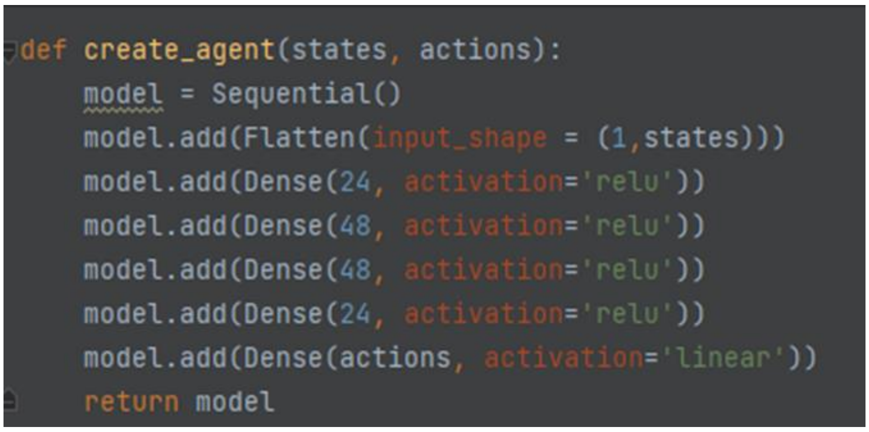
Figure 5: The results of the final version of DQL

As figure 5 shows, and accuracy of around 7.9 was found. Out of the 100 episodes tested, the most steps taken was 18, suggesting that the created network provided a 'near optimal' solution.

At an accuracy level of this degree, it is very hard to work out how close to an optimal solution the created model provides, since a taxi following the optimal path doesn't always yield the same points. Based on the number of steps each episode took, I would conclude that this model is 'near optimal.'

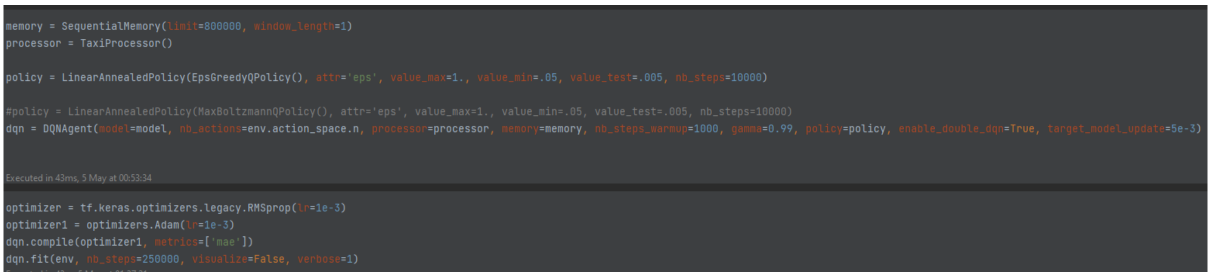
1.4.3 – Double Deep Q-Learning

To develop a Double Deep Q-Learning network, I used the DQL network mentioned above as a base. I then experimented with different values and agent models, to see what improved performance. Below shows the kept changes to the model:



```
def create_agent(states, actions):
    model = Sequential()
    model.add(Flatten(input_shape = (1,states)))
    model.add(Dense(24, activation='relu'))
    model.add(Dense(48, activation='relu'))
    model.add(Dense(48, activation='relu'))
    model.add(Dense(24, activation='relu'))
    model.add(Dense(actions, activation='linear'))
    return model
```

Figure 6: The DDQL's agent.



```
memory = SequentialMemory(limit=800000, window_length=1)
processor = TaxiProcessor()

policy = LinearAnnealedPolicy(EpsGreedyQPolicy(), attr='eps', value_max=1., value_min=.05, value_test=.005, nb_steps=10000)

#policy = LinearAnnealedPolicy(MaxBoltzmannQPolicy(), attr='eps', value_max=1., value_min=.05, value_test=.005, nb_steps=10000)
dqn = DQNAgent(model=model, nb_actions=env.action_space.n, processor=processor, memory=memory, nb_steps_warmup=1000, gamma=0.99, policy=policy, enable_double_dqn=True, target_model_update=5e-3)


Executed in 43ms, 5 May at 00:53:34

optimizer = tf.keras.optimizers.legacy.RMSprop(lr=1e-3)
optimizer1 = optimizers.Adam(lr=1e-3)
dqn.compile(optimizer1, metrics=['mae'])
dqn.fit(env, nb_steps=250000, visualize=False, verbose=1)
```

Figure 7: The DDQL agent.

This model also preformed well for the given task, scoring an average of 8.46 points, with 17 being the longest path taken.

1.4.4 – A Comparison of the Models

Both models performed well for this application, successfully achieving around 8 points per episode. Due to the points range for episodes of different length optimal paths, it is very hard to definitively state that one model is better than the other, since both achieve similar values in performance. Unfortunately, I do not know the true average for the maximum points received for every starting possibility, so it is hard to say how far these solutions are from being optimal.

However, I do know that the optimal path for the worst-case scenario is 18 steps, which both models didn't exceed. This suggests that the models appear to be 'near optimal.'

Regarding total execution time, they were both very similar, taking 2535 and 2627 seconds for DQL and DDQL respectively. However, this isn't a fair comparison, since alterations were made to the model.

To conclude, the DQL's main issue of overestimating values did not affect its performance for this application, with DDQL being as suitable of a model. Further testing should be carried out to investigate whether DDQL provides a model of the same quality reliably, and to test whether some trainings of DQL end up being affected by its overestimation of values for this particular game.

2 – Sequence Learning

2.1 – Recurrent Neural Network

Recurrent Neural Networks were suggested in 1925 by (Ising, 1925), which allow the connections between nodes to create a cycle, allowing outputs from the nodes to affect the subsequent input to the same nodes. The network uses sentiment analysis, based off the provided embeddings and weights, to decide what affect the current information, such as a word, will have on the result of the output. (Shadrina, Sutoyo and Widartha, 2021) When this has been decided, the value is passed back to the beginning of the Sequential model, to be used along side the next word. This new word/information undergoes the same process, but instead affects the value that was passed into the function.

This process repeats until all the information has been covered, resulting in a value that represents all the passed information. This value is used to decide the classification for the passed information.

However, this process does come with a potential issue. Data that is read early on is often forgotten about by the end, with later data sometimes having a greater effect on the chosen classification. (Shadrina, Sutoyo and Widartha, 2021). This is known as the vanishing gradient problem and is problematic, since it can lead to important information being forgotten about, leading to incorrect classifications. This also suffers from the opposite effect, whereby a word has too much affect on the classification, which is know as the exploding gradient problem.

2.2 – Long Short-Term Memory

LSTM was invented in 1997 (Hochreiter and Schmidhuber, 1997) and attempts to fix the vanishing gradient problem of RNN. This is done by remembering important key words that are encountered when the data is passing through, helping to not forget useful information. This information is used to evaluate preceding words, taking better into account the context of said word than the RNN model. This often has a great affect on natural language processing, since related words that can vastly alter each other's meaning are not always found next to each other. By remembering important information, such as the context of a word, a more accurate value may be calculated at later stages of a given sentence.

However, LSTM still suffers from the exploding gradient problem, whereby certain words or combination of words will have too great of an impact on the classification, resulting in a wrong classification.

2.3 – An Empirical Evaluation of RNN and LSTM

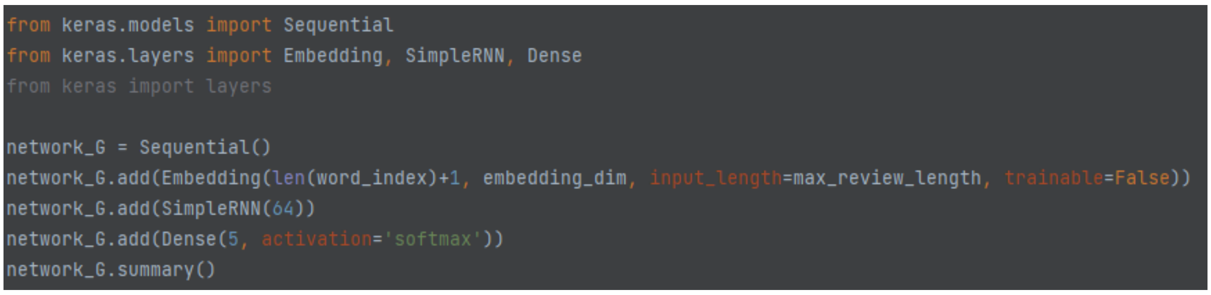
2.3.1 – Overview

A data file containing 20,491 reviews from Trip Adviser was used to test the accuracy of the RNN and LSTM when performing sentiment analysis. The models will be fed up to the first 500 words of each review and will attempt to predict the rating that user provided, on a scale of 1 to 5.

The reviews were split into training, validation, and testing sets of 15,200, 3,800 and 1,491 respectively.

2.3.2 – RNN

For this model, I initially created a very simple network, as shown in the image below:



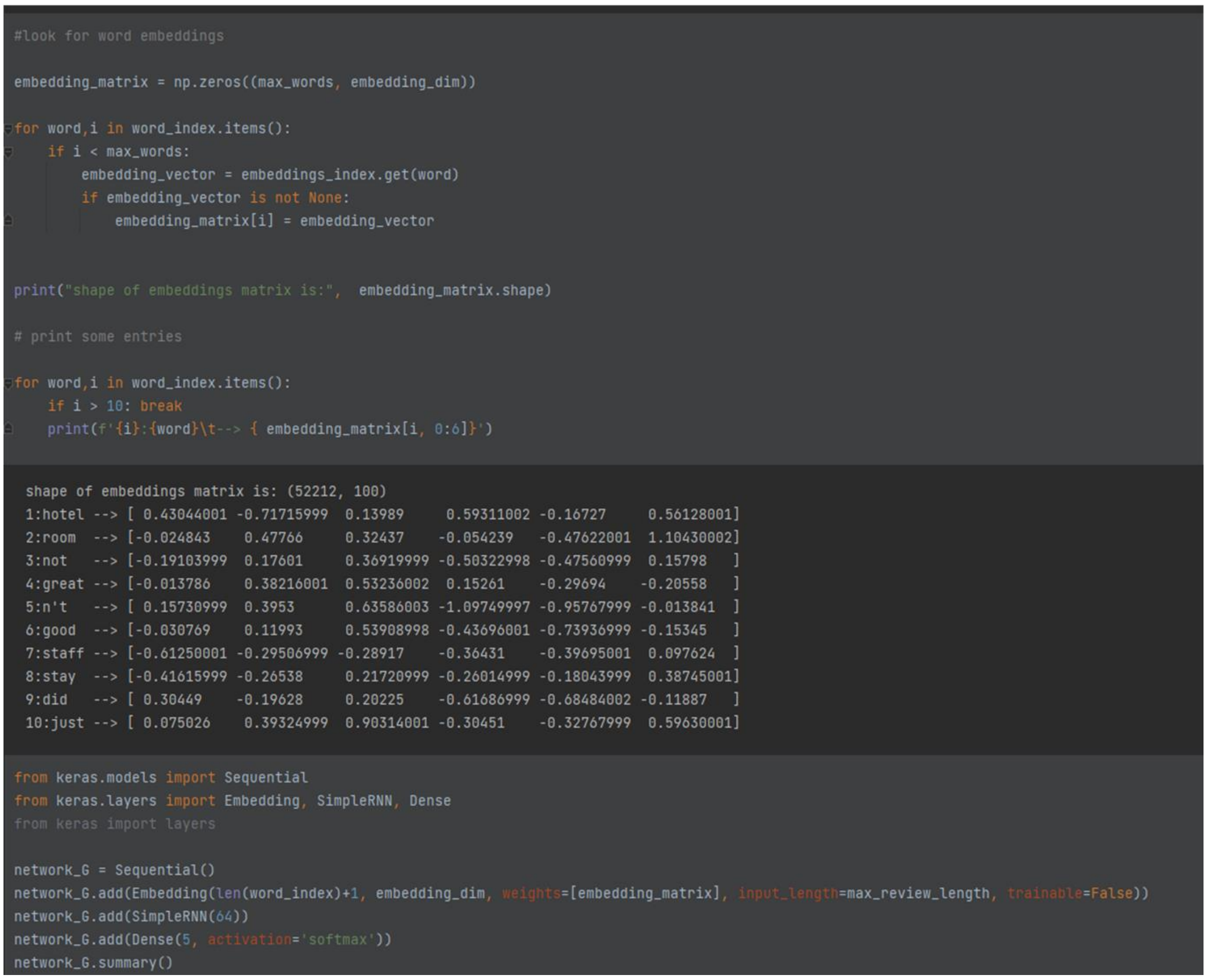
```
from keras.models import Sequential
from keras.layers import Embedding, SimpleRNN, Dense
from keras import layers

network_G = Sequential()
network_G.add(Embedding(len(word_index)+1, embedding_dim, input_length=max_review_length, trainable=False))
network_G.add(SimpleRNN(64))
network_G.add(Dense(5, activation='softmax'))
network_G.summary()
```

Figure 8: The starting point for the RRN model.

This utilizes the default embedding weights to be used with a RNN layer with 64 nodes, finishing with a 5 node Dense layer to provide the 5 output options, using the Adam optimizer.

This model, run in batches of 64 over 10 epochs achieved an accuracy of 0.479. To increase this further, lots of experimentation was conducted, changing the batch size, total nodes of SimpleRNN, the optimiser, adding dropout and recurrent_dropout and using a bidirectional RNN. However, these changes all lead to a negligible or negative change, whilst significantly increasing the training time.

However, some changes did remain, specifically the amount of epochs, which was set to 20, and the use of an external embedding matrix, globe.6B.100d. These changes resulted in the accuracy now reaching 0.488.



```
#look for word embeddings

embedding_matrix = np.zeros((max_words, embedding_dim))

for word,i in word_index.items():
    if i < max_words:
        embedding_vector = embeddings_index.get(word)
        if embedding_vector is not None:
            embedding_matrix[i] = embedding_vector


print("shape of embeddings matrix is:",  embedding_matrix.shape)

# print some entries

for word,i in word_index.items():
    if i > 10: break
    print(f'{i}:{word}\t--> { embedding_matrix[i, 0:6]}')
```

```
 shape of embeddings matrix is: (52212, 100)
 1:hotel --> [ 0.43044001 -0.71715999  0.13989     0.59311002 -0.16727     0.56128001]
 2:room  --> [-0.024843    0.47766     0.32437    -0.054239   -0.47622001  1.10430002]
 3:not   --> [-0.19103999  0.17601     0.36919999 -0.50322998 -0.47560999  0.15798   ]
 4:great --> [-0.013786    0.38216001  0.53236002  0.15261    -0.29694    -0.20558   ]
 5:n't   --> [ 0.15730999  0.3953      0.63586003 -1.09749997 -0.95767999 -0.013841  ]
 6:good  --> [-0.030769    0.11993     0.53908998 -0.43696001 -0.73936999 -0.15345   ]
 7:staff --> [-0.61250001 -0.29506999 -0.28917    -0.36431    -0.39695001  0.097624  ]
 8:stay  --> [-0.41615999 -0.26538     0.21720999 -0.26014999 -0.18043999  0.38745001]
 9:did   --> [ 0.30449    -0.19628     0.20225    -0.61686999 -0.68484002 -0.11887   ]
 10:just --> [ 0.075026    0.39324999  0.90314001 -0.30451    -0.32767999  0.59630001]
```

```
from keras.models import Sequential
from keras.layers import Embedding, SimpleRNN, Dense
from keras import layers

network_G = Sequential()
network_G.add(Embedding(len(word_index)+1, embedding_dim, weights=[embedding_matrix], input_length=max_review_length, trainable=False))
network_G.add(SimpleRNN(64))
network_G.add(Dense(5, activation='softmax'))
network_G.summary()
```

Figure 9: The use of Glove for RNN.

2.3.3 – LSTM

The same model created for RNN was repurposed for LSTM, which successfully achieved 0.641 accuracy after 10 epochs. However, just like the RNN model, most changes did not help to improve the result of this model. For example, running twice the amount of epochs yielded 0.639 and 0.638 accuracy for a LSTM and bidirectional LSTM network.

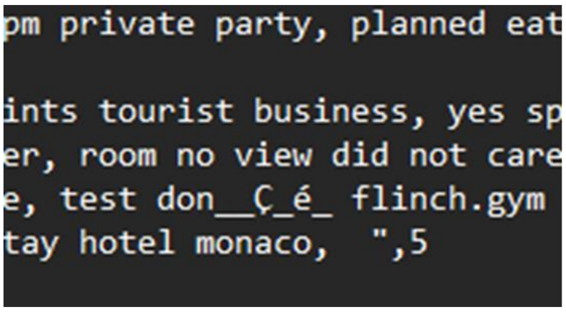
2.3.4 – A Comparison of the Models

Both models did not perform that well with the provided data, producing an error rate that I would deem to be too great for any actual implementation. With that aside, the experiment still showed that LSTM can outperform RNN, with its ability to retain important information for longer.

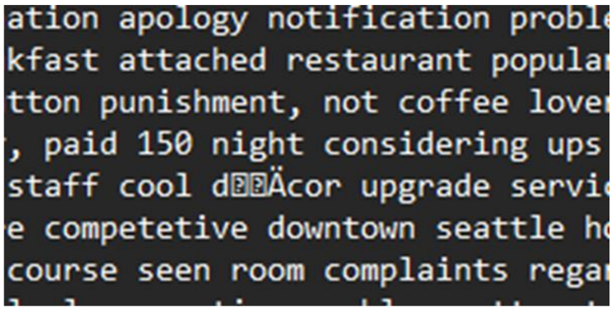
2.3.5 – Issues Affecting Accuracy

The poor performance of these models is most likely due to the used data set, which wasn't pre-processed. As such, there were many words containing spelling mistakes, which the models wouldn't have understood, resulting in the being skipped. If these words were adjectives or verbs, its possible that the whole tone of the sentence would be misunderstood.

Similarly, there are lots of words that are contain spaces after each letter, which the model would attempt to interpret as individual words. There were also the use of non utf-8 characters and seemingly random characters throughout the .csv file. All of these would not be interpretable by the network, reducing the total information collected.



```
pm private party, planned eat

ints tourist business, yes sp
er, room no view did not care
e, test don__Ç_é_ flinch.gym
tay hotel monaco,  ",5
```



```
ation apology notification probl
kfast attached restaurant popula
tton punishment, not coffee love
, paid 150 night considering ups
staff cool d��Äcor upgrade servi
e competetive downtown seattle h
course seen room complaints rega
```

Figure 10: Examples of issues within the Trip Advisor reviews files.

References

Hu, W., Wang, J. and Yu, Y. (2021). Performance Analysis and Improvement of Deep Q Network in Pommerman Agent Based on Compound Training Method. *2021 3rd International Conference on Machine Learning, Big Data and Business Intelligence (MLBDBI)*. doi:https://doi.org/10.1109/mlbdbi54094.2021.00132.

Kröse, B.J.A. (1995). Learning from delayed rewards. *Robotics and Autonomous Systems*, 15(4), pp.233–235. doi:https://doi.org/10.1016/0921-8890(95)00026-c.

Watkins, C. (2014). *Q Learning*. [online] cml.rhul.ac.uk. Available at: https://cml.rhul.ac.uk/qlearning.html#:~:text=An%20application%20of%20Q%2Dlearning [Accessed 4 May 2023].

Pina, R., Tibebu, H., Hook, J., De Silva, V. and Kondoz, A. (2021). Overcoming Challenges of Applying Reinforcement Learning for Intelligent Vehicle Control. *Sensors*, 21(23), p.7829. doi:https://doi.org/10.3390/s21237829.

Kumar, S. (2020). Balancing a CartPole System with Reinforcement Learning -- A Tutorial. doi:https://doi.org/10.48550/arXiv.2006.04938.

van Hasselt, H., Guez, A. and Silver, D. (2015). Deep Reinforcement Learning with Double Q-learning. (v3). doi:https://doi.org/10.48550/arXiv.1509.06461.

www.gymlibrary.dev. (n.d.). *Taxi - Gym Documentation*. [online] Available at: https://www.gymlibrary.dev/environments/toy_text/taxi/.

Ising, E. (1925). Beitrag zur Theorie des Ferromagnetismus. *Zeitschrift für Physik*, 31(1), pp.253–258. doi:https://doi.org/10.1007/bf02980577.

Shadrina, N.K., Sutoyo, E. and Widartha, V.P. (2021). Sentiment Analysis in Reviews About Beaches in Bali on Tripadvisor Using Recurrent Neural Network (RNN). *2021 IEEE 7th Information Technology International Seminar (ITIS)*. doi:https://doi.org/10.1109/itis53497.2021.9791501.

Hochreiter, S. and Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, 9(8), pp.1735–1780.