

Huzaifa Patel

Deep Learning Project – 2025

Reinforcement Learning and Sequence Learning

Empirical evaluations of Reinforcement and Sequence
learning algorithms

Abstract

This report details the comparative study into the analysis of Reinforcement Learning and Sequence Learning neural networks. Section 1 introduces Deep Q -Learning and the motivation that led to Double Q- Learning, followed by an empirical evaluation of these neural networks. Section 2 introduces Recurrent Neural Network (RNN) and the motivation that led to Long Short-Term Memory (LSTM), followed by an empirical evaluation of these neural networks. This report details the model architecture, training processes, evaluation and results from this study.

Table of contents

Contents

Abstract.....	2
Section 1: Reinforcement Learning	3
1.1 Introduction.....	3
1.2 Deep Q-Learning and Double Q-Learning.....	3
1.3 Developing models to carry out an empirical comparison of Deep Q- Learning against Double Q-Learning.	4
1.3.1 Introduction	4
1.3.2 Methodology and Development.....	4
1.3.3 Empirical Results	9
Section 2: Sequence Learning	11
2.1 Introduction.....	11
2.2 The vanishing gradient problem.....	11
2.3 Developing neural network models to carry out an empirical comparison of RNNs and LSTMs	12
2.3.1 Introduction	12
2.3.2 Methodology and Development.....	13
References.....	18

Section 1: Reinforcement Learning

1.1 Introduction

The goal of this task is to understand the overestimation bias that occurs in Deep Q-Learning and how Double Q-Learning addresses these challenges. Firstly, the implications of overestimation in Deep Q-Learning and how Double Q-Learning solves this is discussed. This is followed by an empirical comparison using these neural networks to develop models for reinforcement learning.

Through developing sequence learning models to solve real-world tasks and comparing these models, I aim to understand their advantages and trade-offs for decision-making based off past experiences. This will give me a comprehensive understanding of the reinforcement learning concepts and models I have learned about at university.

1.2 Deep Q-Learning and Double Q-Learning

Deep Q-Learning uses a deep neural network (the Q-network) to approximate the Q-values for state-action pairs. The main advantage of this type of architecture is the ability to compute Q-values for all possible actions in a given state with only a single forward pass through the network. (Mnih et al., 2013, p. 28) A target network and experience relay are used to improve the algorithm's performance. Environments which have limited feedback or high variance in rewards can cause Deep Q-Learning to perform poorly. This algorithm uses the same network to select and evaluate actions during the Q-value update when computing target values. The max operation tends to overestimate the Q-values which leads to optimistic values and unstable learning.

Hado van Hasselt found that the overestimation of Q-values in Deep Q-Learning is because the same values are used both for selecting and

evaluating actions. He proposed Double Q-Learning to mitigate this, which separated these roles. The idea of Double Q-learning is to reduce overestimations by decomposing the max operation in the target into action. (Hasselt et al., 2016, p. 30) This algorithm uses the main Q-network to choose the action and the target network to evaluate its value. This separation reduces the overestimation and leads to less optimistic values and more stable learning.

1.3 Developing models to carry out an empirical comparison of Deep Q-Learning against Double Q-Learning.

1.3.1 Introduction

I have developed models to train on Taxi-v3 from OpenAI, a reinforcement learning environment where the taxi agent must pick up and drop off customers at the designated locations. The same model is used for both Deep Q-Learning and Double Q-Learning, the only change is to the 'enable_double_dqn= True' value to ensure the comparison is fair.

1.3.2 Methodology and Development

This section outlines the steps I have taken to plan, implement and evaluate the algorithms. The key stages of the methodology for this model include the environment, model logic, and training.

Environment

I have used OpenAI Gym's Taxi-v3 environment, and the state and action space dimensions were initialised.

The Taxi-v3 environment was tweaked by implementing reward shaping. I adjusted the default deterministic rewards, implementing custom slippery rewards. Testing revealed that this helped the model to train in less time,

and with better accuracy. Figure 1 below shows the adjusted rewards which are retained to be usable for further experimentation. I then implemented noise to promote experimentation instead of fixating on a proven strategy, which can help to reduce overfitting.

```
#deterministic rewards
# def process_reward(self, r):
#     if r == 20: # successful dropoff
#         return +20
#     elif r == -1:
#         return -0.1 # Smallre penalty
#     elif r == -10:
#         return -2.0 # Stronger penalty
#     return r

#slippery rewards
def process_reward(self, r):
    noise = np.random.normal(0, 0.01) #small standard deviation
    if r == 20:
        return 20 + noise
    elif r == -1:
        return -0.1 + noise # Small penalty with noise
    elif r == -10:
        return -2.0 + noise
    return r
```

Figure 1: Taxi-v3 custom rewards

Model Architecture

The architecture of the model begins with an embedding layer to convert the state indices into 10D. Then the embeddings are flattened into a 1D vector, prepared for the Dense layers. 2 Dense layers follow, with 32 neurons and uses the ReLU activation to allow the model to learn complex patterns from the inputs. The final Dense layer outputs the Q-values for each action, ready for the agent to select the best action during training.

```
def create_agent(states, actions): #WITHOUT EMBEDDINGS - Code labelled as such is retained to be reusable for further experimentation
    model = Sequential()
    model.add(Flatten(input_shape=(1, states)))
    model.add(Dense(32, activation='relu'))
    model.add(Dense(32, activation='relu'))
    model.add(Dense(actions, activation='linear'))
    return model

def create_agent_with_embedding(states, actions): #WITH EMBEDDINGS
    model = Sequential()
    model.add(Embedding(input_dim=states, output_dim=10, input_length=1))
    model.add(Flatten())
    model.add(Dense(32, activation='relu'))
    model.add(Dense(32, activation='relu'))
    model.add(Dense(actions, activation='linear'))
    return model
```

Figure 2: Model Architectures

Many changes were made to the agent model. It initially used a fully connected neural network to process flattened inputs. Experimenting with

reshape, BatchNormalisation, GlobalAveragePooling1D and other layers produced poor results, so I decided to keep a simple model architecture.

I chose to comment out and not delete the model without embeddings to allow the code to be reusable for further experimentation. The TaxiProcessor class and model definitions compatible with this model were also commented out for this reason. The results for both agents are shown in the figures below.

```
Interval 1 (0 steps performed)
1000/1000 [=====] - 16s 15ms/step - reward: -0.7289
10 episodes - episode_reward: -72.890 [-82.200, -61.300] - loss: 0.048 - mae: 0.650 - mean_q: -0.180 - mean_eps: 0.973 - prob: 1.000

Interval 2 (1000 steps performed)
1000/1000 [=====] - 17s 17ms/step - reward: -0.6757
10 episodes - episode_reward: -67.570 [-78.400, -57.500] - loss: 0.003 - mae: 1.016 - mean_q: -0.565 - mean_eps: 0.933 - prob: 1.000

Interval 3 (2000 steps performed)
1000/1000 [=====] - 17s 17ms/step - reward: -0.6301
10 episodes - episode_reward: -63.010 [-86.000, -42.300] - loss: 0.004 - mae: 1.188 - mean_q: -0.753 - mean_eps: 0.888 - prob: 1.000

Interval 4 (3000 steps performed)
1000/1000 [=====] - 17s 17ms/step - reward: -0.5997
10 episodes - episode_reward: -59.970 [-78.400, -46.100] - loss: 0.003 - mae: 1.216 - mean_q: -0.767 - mean_eps: 0.843 - prob: 1.000
```

Figure 3: 4000 steps without embeddings

```
Interval 1 (0 steps performed)
1000/1000 [=====] - 2s 1ms/step - reward: -1.7682
10 episodes - episode_reward: -176.820 [-194.300, -91.700] - prob: 1.000

Interval 2 (1000 steps performed)
1000/1000 [=====] - 12s 12ms/step - reward: -0.1684
10 episodes - episode_reward: -16.840 [-23.300, -13.800] - loss: 0.083 - mae: 0.618 - mean_q: -0.270 - prob: 1.000

Interval 3 (2000 steps performed)
1000/1000 [=====] - 11s 11ms/step - reward: -0.1589
10 episodes - episode_reward: -15.890 [-19.500, -13.800] - loss: 0.003 - mae: 1.224 - mean_q: -0.824 - prob: 1.000

Interval 4 (3000 steps performed)
1000/1000 [=====] - 11s 11ms/step - reward: -0.1418
10 episodes - episode_reward: -14.180 [-25.200, -10.000] - loss: 0.006 - mae: 1.577 - mean_q: -1.207 - prob: 1.000
```

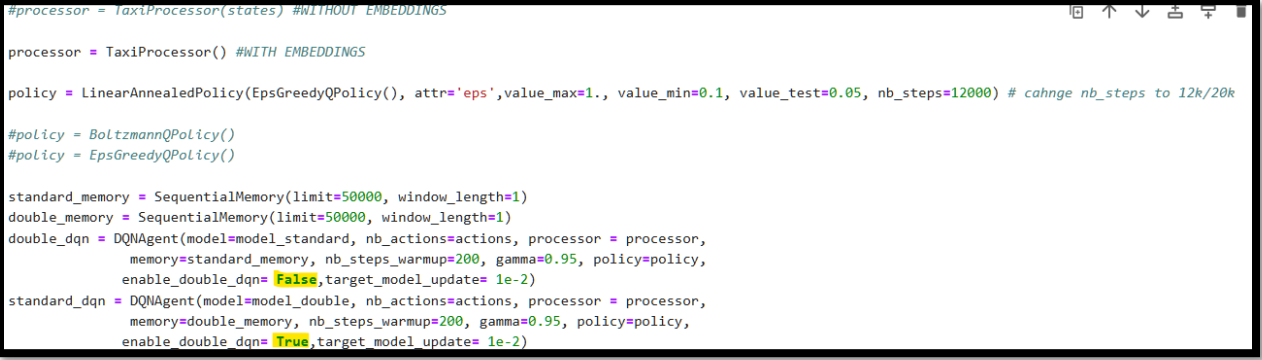
Figure 4: 4000 steps with embeddings

With embeddings the average reward increased much faster, and eventually fluctuated between -20 and 20, with an average of 0. More refinements were needed to average positive rewards, but the results show embeddings were an improvement.

Agent Setup

LinearAnnealedPolicy was used to gradually reduce the exploration rate. The policy is wrapped around EpsGreedyQPolicy() which chooses a random

action with probability ϵ or chooses the action with the highest predicted Q-value. Additional testing with BoltzmannQPolicy and EpsGreedyQPolicy revealed that my implementation of the LinearAnnealedPolicy produced the better results.



```
#processor = TaxiProcessor(states) #WITHOUT EMBEDDINGS
processor = TaxiProcessor() #WITH EMBEDDINGS

policy = LinearAnnealedPolicy(EpsGreedyQPolicy(), attr='eps', value_max=1., value_min=0.1, value_test=0.05, nb_steps=12000) # change nb_steps to 12k/20k

#policy = BoltzmannQPolicy()
#policy = EpsGreedyQPolicy()

standard_memory = SequentialMemory(limit=50000, window_length=1)
double_memory = SequentialMemory(limit=50000, window_length=1)
double_dqn = DQNAgent(model=model_standard, nb_actions=actions, processor=processor,
memory=standard_memory, nb_steps_warmup=200, gamma=0.95, policy=policy,
enable_double_dqn=False, target_model_update=1e-2)
standard_dqn = DQNAgent(model=model_double, nb_actions=actions, processor=processor,
memory=double_memory, nb_steps_warmup=200, gamma=0.95, policy=policy,
enable_double_dqn=True, target_model_update=1e-2)
```

Figure 5: Agent Setup

A processor was created as the Taxi environment involves handling a large discrete state space with multiple components.

The DQNAgent was configured with 200 initial steps before training took place. This random exploration allowed the agent to collect diverse data before updating the model. The other parameters in DQNAgent were fine-tuned during dozens of tests to balance the agent's exploration to provide stability during training. The highlighted text (enable_double_dqn) is the only difference between the 2 models for a fair comparison.

Evaluating Performance

The optimiser was changed from 'RMSProp' to 'Adam' for its improvements in momentum and bias correction. Kingma and Ba, the developers of Adam describe it as a versatile algorithm that scales to large-scale high-dimensional machine learning problems. (Kingma and Ba, 2014)

During testing the models only averaged positive rewards around step 12,000. This means that the ϵ should finish its decay right when the agent has discovered its first good policies (at 12,000 steps). Because of this the linear annealing was set to decay ϵ from 1.0 \rightarrow 0.1 over 12 000 steps to maximise its chance to consolidate and improve that policy in the later phase. This significantly improved the performance, resulting in a much faster learning curve and higher mean Q-values (10.326 \rightarrow 5.797).

Performance Comparison

Parameter	nb_steps= 12000	nb_steps=16 000	nb_steps=20 000
Test Avg Reward	1.50	18.82	18.86
Test StdDev	14.09	0.19	0.21
Min Reward	-10.0	18.5	18.5
Training at 15k-16k interval	Mixed results	16.92 average reward	14.94 average reward
Overall trend	Good	Excellent	Excellent

Table 1: nb_steps performance comparison

Testing showed that the nb_steps=16000 configuration is clearly superior to 12000 steps and identical to 20000 steps in test performance, but with 20% less training time.

```
policy = LinearAnnealedPolicy(EpsGreedyQPolicy(), attr='eps', value_max=1., value_min=0.1, value_test=0.05, nb_steps=12000) # change nb_steps to 12k/20k
```

Figure 6: LinearAnnealedPolicy

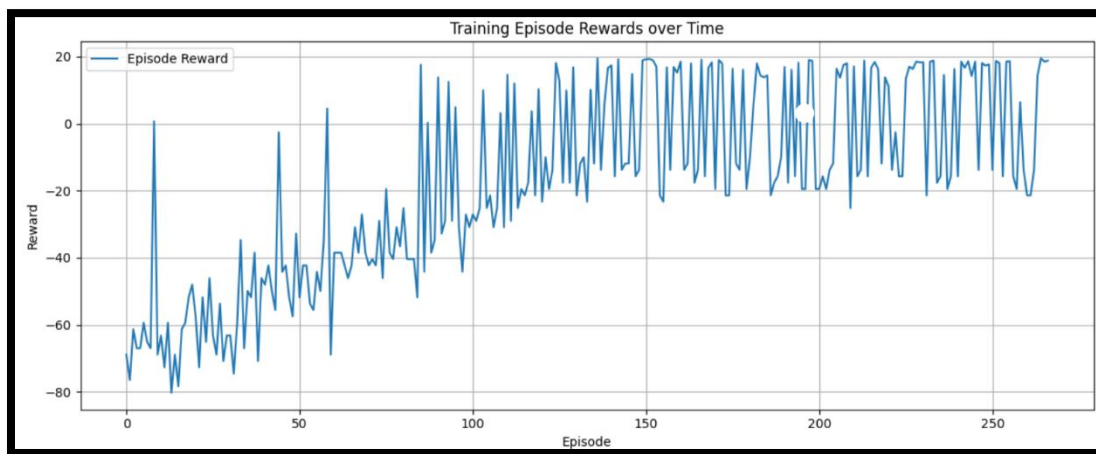


Figure 7: Results before fine tuning nb_steps

1.3.3 Empirical Results

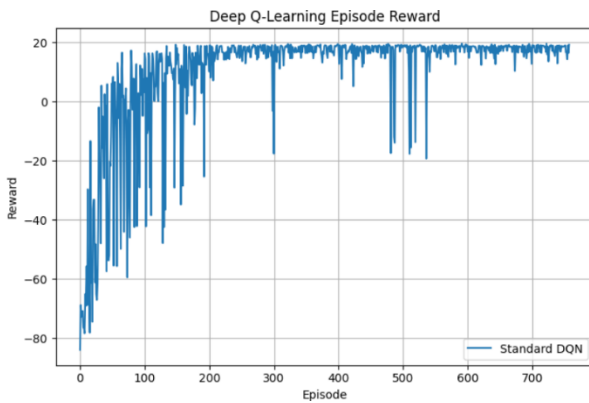


Figure 8: Deep Q-Learning rewards graph

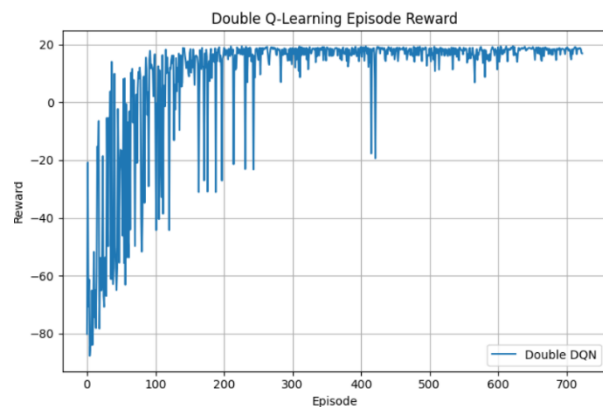


Figure 9: Double Q-Learning rewards graph

Overoptimism does not always adversely affect the quality of the learned policy. (Hasselt et al., 2016, p. 48) We can see that both algorithms eventually converged to the maximum reward. In early training, Double Q-learning shows slightly more stability.

The training data shows more nuanced differences between the algorithms. 300-400 episodes was the point that training should have ended.

```
enable_double_dqn=True
Training for 50000 steps ...
Interval 1 (0 steps performed)
1000/1000 [=====] - 19s 18ms/step - reward: -0.7213
10 episodes - episode_reward: -72.130 [-87.900, -48.000] - loss: 0.047 - mae: 0.660 - mean_q: -0.187 - mean_eps: 0.973 - prob: 1.000
Interval 10 (9000 steps performed)
1000/1000 [=====] - 26s 26ms/step - reward: -0.4333
11 episodes - episode_reward: -42.000 [-55.600, 6.600] - loss: 0.197 - mae: 3.364 - mean_q: 4.973 - mean_eps: 0.573 - prob: 1.000
Interval 20 (19000 steps performed)
1000/1000 [=====] - 25s 25ms/step - reward: 0.2920
28 episodes - episode_reward: 10.932 [-19.500, 19.300] - loss: 0.383 - mae: 6.310 - mean_q: 8.868 - mean_eps: 0.123 - prob: 1.000
Interval 30 (29000 steps performed)
1000/1000 [=====] - 27s 27ms/step - reward: 0.5905
41 episodes - episode_reward: 14.037 [-19.500, 19.400] - loss: 0.434 - mae: 8.254 - mean_q: 11.281 - mean_eps: 0.100 - prob: 1.000
Interval 40 (39000 steps performed)
1000/1000 [=====] - 29s 29ms/step - reward: 1.2109
69 episodes - episode_reward: 17.549 [12.300, 19.400] - loss: 0.265 - mae: 8.639 - mean_q: 11.883 - mean_eps: 0.100 - prob: 1.000
Interval 50 (49000 steps performed)
1000/1000 [=====] - 24s 24ms/step - reward: 1.1878
done, took 1318.263 seconds
-----
Testing for 5 episodes ... Episode 1: reward: 18.700, steps: 14 Episode 2: reward: 19.000, steps: 11 Episode 3: reward: 19.000, steps: 11 Episode 4: reward:
18.900, steps: 12 Episode 5: reward: 19.000, steps: 11
```

Figure 10: Double Q-Learning training results

```

enable_double_dqn=False
Training for 50000 steps ...
Interval 1 (0 steps performed)
1000/1000 [=====] - 17s 16ms/step - reward: -0.7403
10 episodes - episode_reward: -74.030 [-84.100, -61.300] - loss: 0.461 - mae: 7.516 - mean_q: 10.287 - mean_eps: 0.973 - prob: 1.000
Interval 10 (9000 steps performed)
1000/1000 [=====] - 19s 19ms/step - reward: -0.1329
17 episodes - episode_reward: -7.712 [-53.700, 12.900] - loss: 0.414 - mae: 6.045 - mean_q: 8.524 - mean_eps: 0.573 - prob: 1.000
Interval 20 (19000 steps performed)
1000/1000 [=====] - 17s 17ms/step - reward: 1.0050
58 episodes - episode_reward: 17.326 [8.200, 19.500] - loss: 0.384 - mae: 8.727 - mean_q: 11.901 - mean_eps: 0.123 - prob: 1.000
Interval 30 (29000 steps performed)
1000/1000 [=====] - 18s 18ms/step - reward: 1.1859
67 episodes - episode_reward: 17.703 [12.700, 19.400] - loss: 0.249 - mae: 9.405 - mean_q: 12.864 - mean_eps: 0.100 - prob: 1.000
Interval 40 (39000 steps performed)
1000/1000 [=====] - 18s 18ms/step - reward: 1.1571
65 episodes - episode_reward: 17.823 [7.700, 19.400] - loss: 0.166 - mae: 9.716 - mean_q: 13.276 - mean_eps: 0.100 - prob: 1.000
Interval 50 (49000 steps performed)
1000/1000 [=====] - 18s 18ms/step - reward: 1.1688
done, took 909.330 seconds
-----
Testing for 5 episodes ...Episode 1: reward: 18.600, steps: 15 Episode 2: reward: 18.500, steps: 16 Episode 3: reward: 18.700, steps: 14 Episode 4: reward:
18.700, steps: 14 Episode 5: reward: 18.400, steps: 17

```

Figure 11: Deep Q-Learning training results

The training results are obtained by training both models over 50,000 steps, with identical policy, processor, model and rewards. The only difference was in the `enable_double_dqn=True` value for a fair comparison. Deep Q-Learning had more fluctuations in later intervals while Double Q-Learning had more consistent performance. The training and testing results show that although this Deep Q-Learning model was quicker to converge, the mechanism to reduce overestimation in Double Q-Learning resulted in more reliable and slightly better results.

```

=== Test Performance Summary ===
Total Test Episodes: 5
Average Reward      : 18.58
Standard Deviation  : 0.12
Min Reward          : 18.4
Max Reward          : 18.7
All Rewards         : [18.6, 18.5, 18.7, 18.7, 18.4]

```

Figure 12: Deep Q-Learning testing data

```

=== Test Performance Summary ===
Total Test Episodes: 5
Average Reward      : 18.92
Standard Deviation  : 0.12
Min Reward          : 18.7
Max Reward          : 19.0
All Rewards         : [18.7, 19.0, 19.0, 18.9, 19.0]

```

Figure 13: Double Q-Learning Testing data

Testing data shows Double Q-Learning had a higher average reward as well as lower minimum and higher maximum rewards. This aligns with Double DQN's theoretical purpose of reducing overestimation bias. My results show that my both of my models for Double Q-Learning and Deep Q-Learning were effective, and the improvement in average rewards show the benefits of Double Q-Learning

Section 2: Sequence Learning

2.1 Introduction

The goal of this task is to explore how neural networks process sequential data and solve the vanishing gradient problem which occurs over long sequences. Firstly, the architectures of RNNs, LSTMs and Transformers are compared. This is followed by an empirical comparison of RNNs and LSTMs by using these neural networks to develop models for sequence learning.

Through developing sequence learning models to solve real-world tasks and the comparative analysis of these models, I aim to understand their advantages and trade-offs for memory retention. This will give me a comprehensive understanding of the sequence learning concepts and models I have learned about at university.

2.2 The vanishing gradient problem

Recurrent Neural Networks (RNNs) handle sequential data by processing inputs one at a time and retain information from all previous inputs through hidden states. (Vaswani et al., 2017) This allows the model to understand the relationships between the different data and make predictions. These models have a short memory which greatly affects their ability to learn long sequences. As the RNN trains on a dataset, the gradients from the previous inputs continuously decline until they are extremely small or 'vanish'. This causes the network to learn slowly or not at all and is known as the vanishing gradient problem.

Long short-term memory (LSTM) is a type of RNN which aimed to solve the vanishing gradient problem by introducing a cell state during training. The cell state has 3 gates, allowing the model to decide which of the previous information is to be forgotten, added/updated, and passed onto the next cell as an output. This allows the cells to disregard unimportant data whilst selecting relevant information to be stored and later retrieved over long sequences. LSTMs have great improvements over RNNs in regard to capturing both short-term and long-term dependencies, but their use of sequential processing means they can still have problems with the vanishing gradient problem over extremely long sequences. (Ghojogh et al., 2023)

19 years after LSTM was first introduced in 1997, the transformer model was proposed and has become the core architecture in Natural language processing technology (NLPs) such as BERT and ChatGPT. Unlike LSTMs which use the previous data to predict what follows, transformers handle sequential data by tracking the relationships between words, allowing the model to account for the full context. Transformers do not process information word by word or sequentially and instead use the Self Attention technique which considers all the words in a sentence and then weighs the importance of each word given the context. (Vaswani et al., 2017) The transformer model's architecture consists of encoder and decoder layers that involve multi-head self-attention mechanisms. In addition to these layers, feed-forward neural networks are also used.

2.3 Developing neural network models to carry out an empirical comparison of RNNs and LSTMs

2.3.1 Introduction

I have developed RNN and LSTM models to train on a dataset of Tweets from social media, a sequence learning task where the model must accurately predict whether a tweet is labelled in one of the following 3 categories: sadness, neutral, happiness. The same model is used for both RNN and LSTM, the only change is to the individual networks where SimpleRNN and LSTM are used ensure the comparison is fair.

2.3.2 Methodology and Development

This section outlines the steps I have taken to plan, implement and evaluate the RNN and LSTM algorithms. The key stages of the methodology for these models includes the environment, model logic, and training.

Environment

To ensure reproducibility I have fixed the random seed (`np.random.seed(9)`) and configured `os.environ['KMP_DUPLICATE_LIB_OK']='True'` to resolve potential library conflicts.

The model also trains on the original dataset with 40,000 tweets to ensure reproducibility.

Data pre-processing

The dataset has 40,000 tweets labelled to 13 different emotions. These were first displayed to understand the dataset.

```
Unique sentiments: ['empty' 'sadness' 'enthusiasm' 'neutral' 'worry' 'surprise' 'love' 'fun'
'hate' 'happiness' 'boredom' 'relief' 'anger']

Sentiment counts:
sentiment
neutral      8638
worry        8459
happiness    5209
sadness      5165
love         3842
surprise     2187
fun          1776
relief       1526
hate         1323
empty        827
enthusiasm   759
boredom      179
anger        110
Name: count, dtype: int64
```

Figure 14: Displaying the dataset

The dataset is then filtered down to the 3 target classes and mapped to integer labels: sadness:0, neutral:1, happiness:2. The highlighted text in figure 14 above shows the 3 sentiments, with a total sum of 19012 relevant tweets. This shows the database is imbalanced:

sadness: 5165 tweets (27.2%)
neutral: 8638 tweets (45.4%)
happiness: 5209 tweets (27.4%)

A tokenizer was implemented with a vocabulary size of 5,000 most frequently used words. The tweets were then converted to integer sequences and padded/truncated to the 'max_tweets_length' set to 50 tokens.

The number of samples per class was increased using random oversampling to use 10,000 samples per class, significantly increasing the number of tweets. For reproducibility the random oversampling can be set to False, and the models will be trained using a maximum of 5,165 tweets from each sentiment, as shown in figure 14, the minimum tweets per sample is this figure. This ensures that each sentiment is not overrepresented, and the same number of tweets is taken from each.

Data Augmentation was considered, but the dataset had enough data from each sentiment. Instead, random oversampling was used to help balance the sentiment classes without modifying data.

The database was then split into training set: 65%, validation set: 30% and testing set: 5%.

Glove embeddings

I have used the 200-dimensional GloVe embeddings to initialise the word representations. As described by Pennington Et al., GloVE efficiently leverages statistical information by training only on the nonzero elements in a word-word cooccurrence matrix, rather than on the entire sparse matrix or on individual context windows in a large corpus. (Pennington Et al., 2014, p. 1) The embedding layer for both models is set to non-trainable to preserve the pre-trained embeddings.

Model Architectures

```

RNN_network = Sequential()
RNN_network.add(Embedding(max_words, embedding_dim, input_length=max_tweets_length, weights=[embedding_matrix], trainable=False))
RNN_network.add(SimpleRNN(32))
RNN_network.add(Dropout(0.5))
RNN_network.add(Dense(3, activation='softmax'))
RNN_network.summary()

STM_network = Sequential()
STM_network.add(Embedding(max_words, embedding_dim, input_length=max_tweets_length, weights=[embedding_matrix], trainable=False))
STM_network.add(LSTM(32))
STM_network.add(Dropout(0.5))
STM_network.add(Dense(3, activation='softmax'))
STM_network.summary()

```

Figure 15: RNN and LSTM architectures

The architecture was kept the same for a fair comparison. The embedding layer has 5000 words and 200 dimensions, followed by the RNN/LSTM layer, then dropout to help reduce overfitting, and finally the Dense layer with SoftMax as the output layer.

The training protocol uses sparse-categorical-cross-entropy and the Adam optimiser with batch size 32. The epochs are up to 24, and I have introduced early stopping protocols as well as model checkpointing and ReduceLROnPlateau.

Evaluating Performance

```

Epoch 1/24
1219/1219 - 39s - loss: 0.9366 - accuracy: 0.5550 - val_loss: 0.8379 - val_accuracy: 0.6263 - lr: 0.0010 - 39s/epoch - 32ms/step
Epoch 2/24
1219/1219 - 36s - loss: 0.8303 - accuracy: 0.6321 - val_loss: 0.8128 - val_accuracy: 0.6447 - lr: 0.0010 - 36s/epoch - 30ms/step
Epoch 3/24
1219/1219 - 36s - loss: 0.7909 - accuracy: 0.6571 - val_loss: 0.7576 - val_accuracy: 0.6771 - lr: 0.0010 - 36s/epoch - 30ms/step
Epoch 4/24
1219/1219 - 35s - loss: 0.7509 - accuracy: 0.6791 - val_loss: 0.7314 - val_accuracy: 0.6883 - lr: 0.0010 - 35s/epoch - 29ms/step
Epoch 5/24
1219/1219 - 36s - loss: 0.7149 - accuracy: 0.6954 - val_loss: 0.6940 - val_accuracy: 0.7119 - lr: 0.0010 - 36s/epoch - 29ms/step
Epoch 6/24
1219/1219 - 36s - loss: 0.6809 - accuracy: 0.7161 - val_loss: 0.6812 - val_accuracy: 0.7184 - lr: 0.0010 - 36s/epoch - 29ms/step
Epoch 7/24
1219/1219 - 37s - loss: 0.6544 - accuracy: 0.7306 - val_loss: 0.6548 - val_accuracy: 0.7369 - lr: 0.0010 - 37s/epoch - 30ms/step
Epoch 8/24
1219/1219 - 36s - loss: 0.6289 - accuracy: 0.7438 - val_loss: 0.6253 - val_accuracy: 0.7518 - lr: 0.0010 - 36s/epoch - 30ms/step
Epoch 9/24
1219/1219 - 36s - loss: 0.6017 - accuracy: 0.7574 - val_loss: 0.6108 - val_accuracy: 0.7570 - lr: 0.0010 - 36s/epoch - 30ms/step
Epoch 10/24
1219/1219 - 37s - loss: 0.5799 - accuracy: 0.7658 - val_loss: 0.5919 - val_accuracy: 0.7666 - lr: 0.0010 - 37s/epoch - 30ms/step
Epoch 11/24
1219/1219 - 37s - loss: 0.5608 - accuracy: 0.7757 - val_loss: 0.5828 - val_accuracy: 0.7734 - lr: 0.0010 - 37s/epoch - 30ms/step
Epoch 12/24
1219/1219 - 36s - loss: 0.5419 - accuracy: 0.7843 - val_loss: 0.5729 - val_accuracy: 0.7762 - lr: 0.0010 - 36s/epoch - 30ms/step
Epoch 13/24
1219/1219 - 37s - loss: 0.5230 - accuracy: 0.7946 - val_loss: 0.5484 - val_accuracy: 0.7943 - lr: 0.0010 - 37s/epoch - 30ms/step
Epoch 14/24
1219/1219 - 37s - loss: 0.4970 - accuracy: 0.8066 - val_loss: 0.5330 - val_accuracy: 0.8008 - lr: 0.0010 - 37s/epoch - 30ms/step
Epoch 15/24
1219/1219 - 37s - loss: 0.4836 - accuracy: 0.8153 - val_loss: 0.5370 - val_accuracy: 0.7992 - lr: 0.0010 - 37s/epoch - 30ms/step
Epoch 16/24
1219/1219 - 37s - loss: 0.4711 - accuracy: 0.8168 - val_loss: 0.6118 - val_accuracy: 0.7651 - lr: 0.0010 - 37s/epoch - 30ms/step
Epoch 17/24
1219/1219 - 37s - loss: 0.4498 - accuracy: 0.8270 - val_loss: 0.5039 - val_accuracy: 0.8164 - lr: 0.0010 - 37s/epoch - 30ms/step
Epoch 18/24
1219/1219 - 37s - loss: 0.4413 - accuracy: 0.8290 - val_loss: 0.4860 - val_accuracy: 0.8275 - lr: 0.0010 - 37s/epoch - 30ms/step
Epoch 19/24
1219/1219 - 37s - loss: 0.4279 - accuracy: 0.8368 - val_loss: 0.4940 - val_accuracy: 0.8201 - lr: 0.0010 - 37s/epoch - 30ms/step
Epoch 20/24
1219/1219 - 37s - loss: 0.4187 - accuracy: 0.8421 - val_loss: 0.4649 - val_accuracy: 0.8369 - lr: 0.0010 - 37s/epoch - 30ms/step
Epoch 21/24
1219/1219 - 37s - loss: 0.4103 - accuracy: 0.8451 - val_loss: 0.4663 - val_accuracy: 0.8346 - lr: 0.0010 - 37s/epoch - 30ms/step
Epoch 22/24
1219/1219 - 37s - loss: 0.3896 - accuracy: 0.8538 - val_loss: 0.4987 - val_accuracy: 0.8221 - lr: 0.0010 - 37s/epoch - 30ms/step
Epoch 23/24
1219/1219 - 37s - loss: 0.3828 - accuracy: 0.8559 - val_loss: 0.4456 - val_accuracy: 0.8449 - lr: 0.0010 - 37s/epoch - 30ms/step
Epoch 24/24
1219/1219 - 37s - loss: 0.3817 - accuracy: 0.8564 - val_loss: 0.4286 - val_accuracy: 0.8547 - lr: 0.0010 - 37s/epoch - 30ms/step

```

Figure 16: LSTM Training data

```

Number of training tweets: 39000
Epoch 1/24
1219/1219 - 19s - loss: 1.0474 - accuracy: 0.4810 - val_loss: 0.9368 - val_accuracy: 0.5664 - lr: 0.0010 - 19s/epoch - 15ms/step
Epoch 2/24
1219/1219 - 16s - loss: 0.9307 - accuracy: 0.5788 - val_loss: 0.9011 - val_accuracy: 0.5921 - lr: 0.0010 - 16s/epoch - 13ms/step
Epoch 3/24
1219/1219 - 16s - loss: 0.9079 - accuracy: 0.5942 - val_loss: 0.8773 - val_accuracy: 0.6090 - lr: 0.0010 - 16s/epoch - 13ms/step
Epoch 4/24
1219/1219 - 16s - loss: 0.9177 - accuracy: 0.5834 - val_loss: 0.9029 - val_accuracy: 0.5930 - lr: 0.0010 - 16s/epoch - 13ms/step
Epoch 5/24
1219/1219 - 16s - loss: 0.8752 - accuracy: 0.6156 - val_loss: 0.8478 - val_accuracy: 0.6316 - lr: 0.0010 - 16s/epoch - 13ms/step
Epoch 6/24
1219/1219 - 16s - loss: 0.8616 - accuracy: 0.6241 - val_loss: 0.8565 - val_accuracy: 0.6243 - lr: 0.0010 - 16s/epoch - 13ms/step
Epoch 7/24
1219/1219 - 16s - loss: 0.8499 - accuracy: 0.6290 - val_loss: 0.8319 - val_accuracy: 0.6437 - lr: 0.0010 - 16s/epoch - 13ms/step
Epoch 8/24
1219/1219 - 16s - loss: 0.8362 - accuracy: 0.6380 - val_loss: 0.8297 - val_accuracy: 0.6423 - lr: 0.0010 - 16s/epoch - 13ms/step
Epoch 9/24
1219/1219 - 16s - loss: 0.8160 - accuracy: 0.6507 - val_loss: 0.8046 - val_accuracy: 0.6536 - lr: 0.0010 - 16s/epoch - 13ms/step
Epoch 10/24
1219/1219 - 16s - loss: 0.8105 - accuracy: 0.6536 - val_loss: 0.8373 - val_accuracy: 0.6304 - lr: 0.0010 - 16s/epoch - 13ms/step
Epoch 11/24
1219/1219 - 16s - loss: 0.7948 - accuracy: 0.6615 - val_loss: 0.7948 - val_accuracy: 0.6616 - lr: 0.0010 - 16s/epoch - 13ms/step
Epoch 12/24
1219/1219 - 16s - loss: 0.7858 - accuracy: 0.6657 - val_loss: 0.7828 - val_accuracy: 0.6712 - lr: 0.0010 - 16s/epoch - 13ms/step
Epoch 13/24
1219/1219 - 16s - loss: 0.7760 - accuracy: 0.6730 - val_loss: 0.7709 - val_accuracy: 0.6727 - lr: 0.0010 - 16s/epoch - 13ms/step
Epoch 14/24
1219/1219 - 16s - loss: 0.7704 - accuracy: 0.6761 - val_loss: 0.7758 - val_accuracy: 0.6751 - lr: 0.0010 - 16s/epoch - 13ms/step
Epoch 15/24
1219/1219 - 16s - loss: 0.7610 - accuracy: 0.6804 - val_loss: 0.8283 - val_accuracy: 0.6522 - lr: 0.0010 - 16s/epoch - 13ms/step
Epoch 16/24
1219/1219 - 17s - loss: 0.8612 - accuracy: 0.6023 - val_loss: 0.9066 - val_accuracy: 0.5713 - lr: 0.0010 - 17s/epoch - 14ms/step
Epoch 17/24
1219/1219 - 17s - loss: 0.8465 - accuracy: 0.6231 - val_loss: 0.8238 - val_accuracy: 0.6442 - lr: 0.0010 - 17s/epoch - 14ms/step
Epoch 18/24
1219/1219 - 17s - loss: 0.8083 - accuracy: 0.6564 - val_loss: 0.7578 - val_accuracy: 0.6853 - lr: 0.0010 - 17s/epoch - 14ms/step
Epoch 19/24
1219/1219 - 17s - loss: 0.7505 - accuracy: 0.6876 - val_loss: 0.7567 - val_accuracy: 0.6799 - lr: 0.0010 - 17s/epoch - 14ms/step
Epoch 20/24
1219/1219 - 17s - loss: 0.7614 - accuracy: 0.6811 - val_loss: 0.7463 - val_accuracy: 0.6922 - lr: 0.0010 - 17s/epoch - 14ms/step
Epoch 21/24
1219/1219 - 17s - loss: 0.9032 - accuracy: 0.5807 - val_loss: 0.9434 - val_accuracy: 0.5402 - lr: 0.0010 - 17s/epoch - 14ms/step
Epoch 22/24
1219/1219 - 17s - loss: 0.8999 - accuracy: 0.5812 - val_loss: 0.8543 - val_accuracy: 0.6164 - lr: 0.0010 - 17s/epoch - 14ms/step
Epoch 23/24
1219/1219 - 17s - loss: 0.8367 - accuracy: 0.6354 - val_loss: 0.7965 - val_accuracy: 0.6659 - lr: 0.0010 - 17s/epoch - 14ms/step
Epoch 24/24
1219/1219 - 17s - loss: 0.7648 - accuracy: 0.6807 - val_loss: 0.7474 - val_accuracy: 0.6869 - lr: 0.0010 - 17s/epoch - 14ms/step

```

Figure 17: RNN Training data

The LSTM outperformed the RNN by 17% with a substantially lower cross-entropy loss. This is because of LSTM's gating mechanisms which mitigate the vanishing-gradient issues in RNNs.

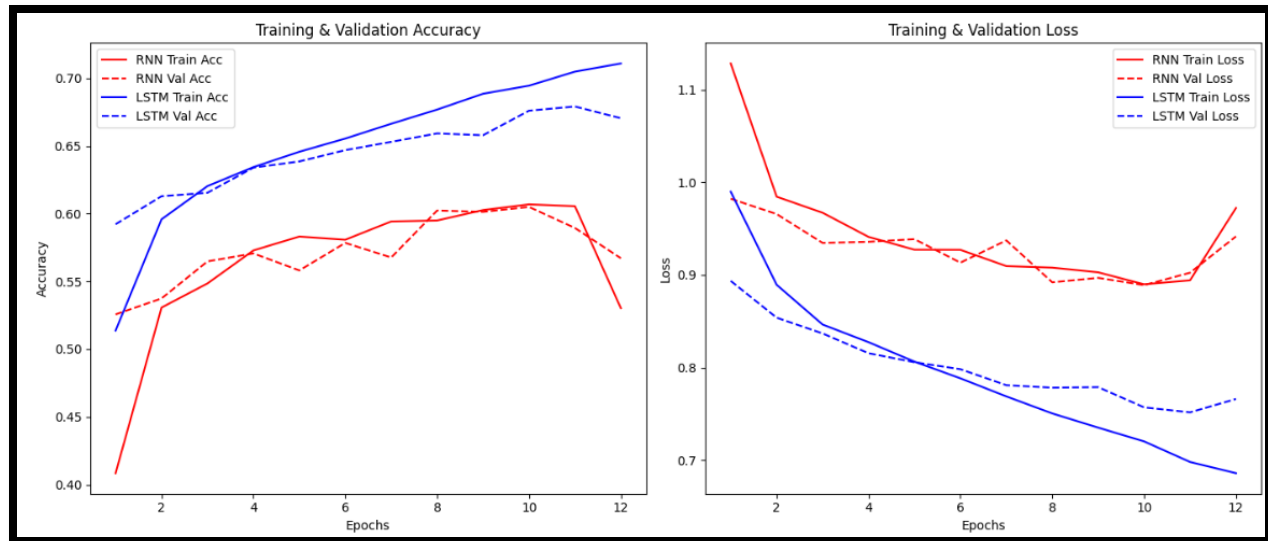


Figure 18: RNN & LSTM Comparison over 12 epochs

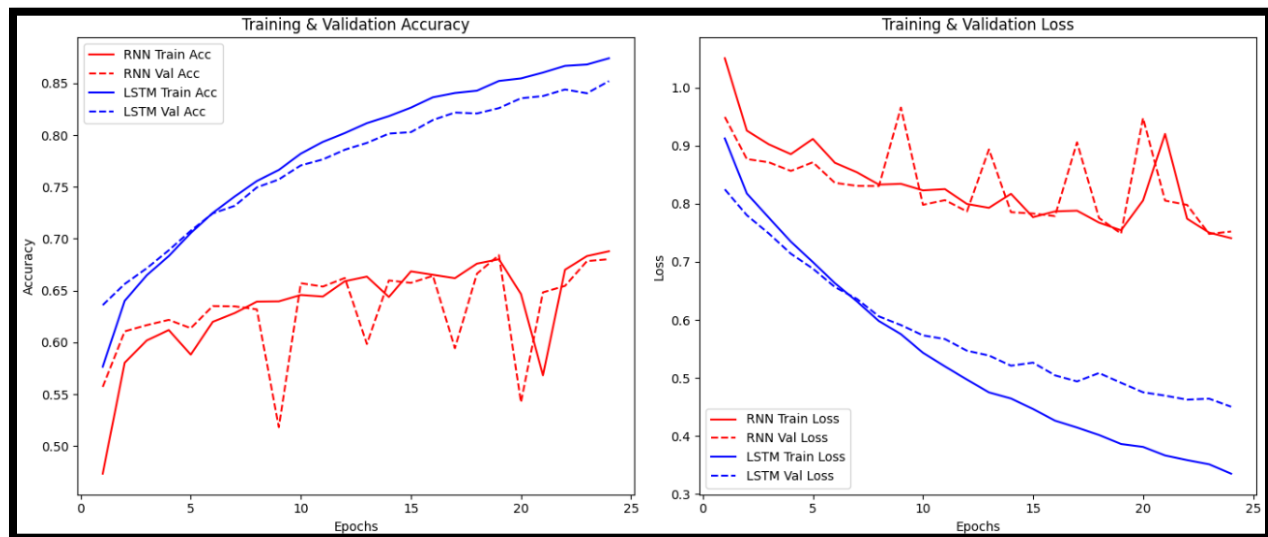


Figure 19: RNN & LSTM Comparison over 24 epochs

The graphs show that the LSTM outperforms the RNN with faster convergence and higher plateaued accuracy.

```

--- Test Results ---
RNN Accuracy: 0.6903 | Loss: 0.7430
LSTM Accuracy: 0.8617 | Loss: 0.4132

/// Sample Predictions \
Tweet 1001: RNN Predicted = 0, LSTM Predicted = 0, Actual = 0
Tweet 3000: RNN Predicted = 1, LSTM Predicted = 1, Actual = 1
Tweet 136: RNN Predicted = 2, LSTM Predicted = 2, Actual = 2
Tweet 2746: RNN Predicted = 2, LSTM Predicted = 2, Actual = 2
Tweet 2868: RNN Predicted = 1, LSTM Predicted = 1, Actual = 1
Tweet 1231: RNN Predicted = 0, LSTM Predicted = 2, Actual = 1
Tweet 1590: RNN Predicted = 1, LSTM Predicted = 0, Actual = 0
Tweet 678: RNN Predicted = 1, LSTM Predicted = 2, Actual = 2
Tweet 2771: RNN Predicted = 1, LSTM Predicted = 2, Actual = 2
Tweet 1391: RNN Predicted = 2, LSTM Predicted = 2, Actual = 2

```

Figure 20: Testing results

The testing results also show the greater results from LSTM.

The simple RNN struggles with long-range dependencies which leads to underfitting, and this is clear from the results. The advantages of LSTM with its input, output, and forget gates help it to regulate the information and effectively categorise the tweets into the give sentiments.

The empirical results favour the LSTM architecture in this tweet categorisation task. The LSTMs ability to manage long-term dependencies and mitigate the vanishing gradient problem have resulted in a higher test accuracy (84.6% vs. 68.6%). This proves its superiority over RNN in sequence learning.

References

Ghojogh, Ghodsi, and Ca (April 2023). Recurrent Neural Networks and Long Short-Term Memory Networks: Tutorial and Survey.
<https://doi.org/10.48550/arXiv.2304.11461>

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A., Kaiser, Ł. and Polosukhin, I. (2017). Attention Is All You Need.
<https://doi.org/10.48550/arXiv.1706.03762>

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M. (2013). Playing Atari with Deep Reinforcement Learning.
<https://doi.org/10.48550/arXiv.1312.5602>

van Hasselt, H., Guez, A., & Silver, D. (2016). Deep Reinforcement Learning with Double Q-Learning. Proceedings of the AAAI Conference on Artificial Intelligence, 30(1).
<https://doi.org/10.1609/aaai.v30i1.10295>

Kingma, D.P. and Ba, J. (2014). Adam: A Method for Stochastic Optimization.
<https://doi.org/10.48550/arXiv.1412.6980>

Pennington, J., Socher, R. and Manning, C. (2014). GloVe: Global Vectors for Word Representation. [online] Association for Computational Linguistics, pp.1532–1543.
<https://aclanthology.org/D14-1162.pdf>.