Mapping Text Instructions to Robot Actions for the RxR-Habitat Challenge

Sachin Arvind Gade

A Dissertation

Presented to the University of Dublin, Trinity College

in partial fulfilment of the requirements for the degree of

Master of Science in Computer Science (Intelligent Systems)

Supervisor: Gerard Lacey

September 2021

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

____
Sachin Arvind Gade

August 31, 2021

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Sachin Arvind Gade

August 31, 2021

Acknowledgments

I would like express my gratitude to my supervisor Dr. Gerard Lacey for guiding, encouraging and supporting me throughout this research. Even in this dire situation, he helped me to overcome the obstacles that were faced during this work.
I would also like to thank my family for supporting me in this academic year.

Sachin Arvind Gade

University of Dublin, Trinity College
September 2021

Abstract

A robot that can perform a task described using natural language instructions in a real-world environment has been a core robotics challenge. Vision-and-Language(VLN) is the task of guiding the robot across the real-world environment using natural language instructions. Navigational robots have a limited number of actions available such as Forward, Backward, Turn Right and Turn Left. The VLN agent has to interpret the natural language instructions and perform the actions while reasoning over the images gathered via the camera and avoid collisions while navigating. To follow the instructions, the robot needs to map the instructions to the relevant camera images using object detection. For example, to take right after the table, the robot has to find the table in the images using an object detection algorithm. The RxR Habitat Competition is a challenge that requires building the VLN agents using Habitat-Lab to follow natural language instructions to go across the rooms and reach the goal in a simulated environment using the Habitat Sim. Habitat-Sim is a simulator designed to simulate a 3D environment and can be accessed by Habitat-Lab API.

This thesis describes a baseline model and a state-of-the-art model for the development of a VLN agent which follows the instructions provided and navigates in the simulated scene. We also propose a VLN agent using a Transformer model which may work better than the state-of-the-art VLN agent. This thesis even explains the system constraints for the training environment of the VLN Task.

The VLN agents were implemented and tested using the Seq2Seq Model which is a baseline model containing a single recurrent network and the CMA Model which is more complex and includes two recurrent networks with attention layers in a 3D simulated environment. Usually, these kinds of agents are trained for a large number of episodes on the systems having 40+GB RAM and multiple 16GB GPU. For this dissertation, an average system having 25GB RAM and 16GB single GPU is used and agents were trained by reducing the number of episodes. The results show that even with reduced training, the agents can only carry out the VLN tasks which contain shorter instructions. The experiment also showed that to train the agent with the transformer model, a high-end system with more than 40GB RAM with multiple 16GB GPU is required.

Contents

Acknowledgments iii

List of Tables viii

List of Figures ix

Chapter 1 Introduction 1

1.1 Motivation 1
1.2 Objective 1
1.3 RxR-Habitat Competition 3
1.4 Layout of the Thesis 3

Chapter 2 Background 5

2.1 Chapter Overview 5
2.2 Recurrent Neural Network 6
2.2.1 LSTM 8
2.2.2 GRU 9
2.2.3 Teacher Forcing 10
2.3 Attention Model 11
2.3.1 Self-Attention 12
2.3.2 Multi-Head Attention 13
2.4 Transformer 14
2.5 SLAM 16
2.5.1 ORB-SLAM 17
2.6 Navmesh 18

2.7 Object Detection 18
2.7.1 Image Dataset for Object Detection 19
2.7.2 ResNet50 20

Chapter 3 Design **21**
3.1 Habitat Simulator 21
3.2 Habitat Lab 24
3.3 Dataset 25
3.4 VLN Agents 26
3.4.1 Seq2Seq Model 27
3.4.2 CMA Model 29
3.4.3 Episodic Transformer Model 30

Chapter 4 Implementation **32**
4.1 Habitat Simulator 32
4.2 Habitat Lab 33
4.3 Dataset 33
4.4 Training Environment 35

Chapter 5 Results and Discussion **36**
5.1 Evaluation 36
5.2 Results 37
5.3 Discussion 41

Chapter 6 Conclusion and Future Work **44**
6.1 Conclusion 44
6.2 Further Work 45

Bibliography **46**

Appendices **49**

Appendix A **50**
A.1 Steps to Run the code 50

Appendix B **51**
B.1 Agent Configuration File 51

List of Tables

3.1 Performance of the Habitat-Sim on an Intel Xeon E5-2690 v4 CPU and Nvidia Titan XP GPU(in FPS) 22
3.2 Comarision of Simulators[1] 23
3.3 Comarision of VLN Datasets[2] 25
3.4 Comarision of VLN Models on Alfred Dataset[3] 30

5.1 Execution of Seq2Seq Model on different machines 38
5.2 Execution of CMA Model on different machines 40

List of Figures

1.1 Demonstration of the VLN task. The instruction, the local visual scene, and the global trajectories in a top-down view is shown. The agent does not have access to the top-down view. Path A is the demonstration path following the instruction. Path B and C are two different paths executed by the agent.[4] . . . 2

2.1 VLN Agent Architecture . . . 6
2.2 RNN Architecture . . . 7
2.3 LSTM Architecture . . . 8
2.4 GRU Architecture . . . 10
2.5 Teacher Forcing . . . 11
2.6 One word “attends” to other words in the same sentence differently . . . 12
2.7 Self-Attention Model[5] . . . 12
2.8 Multi-Head Attention Model[5] . . . 14
2.9 Transformer Architecture . . . 15
2.10 Multi-session stereo-inertial ORB-SLAM3[6] . . . 17
2.11 Navigation Mesh(NavMesh) . . . 19
2.12 ImageNet Object Detection [7] . . . 19

3.1 Habitat-Sim Architecture [1] . . . 22
3.2 Habitat-Sim Sensor Outputs [1] . . . 23
3.3 Habitat Lab Architecture[1] . . . 24
3.4 RxR Dataset with Spatiotemporal Grounding[8] . . . 26
3.5 Basic structure of a VLN agent . . . 27
3.6 Seq2Seq Model[9] . . . 28

3.7 CMA Model[9] 29
3.8 Episodic Transformer Model 31

4.1 Habitat Output for Shortest Path(The left side of each image is the RGB input received by the agent and the right side is the top view of the path containing the path taken by the agent in red and ground truth in green.) 34

5.1 NDTW(Path taken by the agent 1(left) and agent 2(right)) 37
5.2 Seq2Seq Model output for Episode Number 436 39
5.3 CMA Model output for Episode Number 108 42
5.4 CMA Model output for Episode Number 631 43

Chapter 1

Introduction

1.1 Motivation

The coronavirus outbreak forced the whole world to face the pandemic situation in early 2020, countless lives were taken by this disease. This pandemic forced many businesses to shut down. But in all this chaos, the use of robots was increased rapidly. As the robots don't require masks, they are easy to disinfect and since they don't get sick, many industries started finding robotics as a solution. The army of robots was deployed all over the world to help with the crisis. The robots were doing various tasks such as monitoring patients, sanitizing areas, making deliveries, and helping frontline medical workers reduce their exposure to the virus. Not all robots operate autonomously. Most of them require humans to control them and supervise their actions and most are limited to simple tasks. The main challenge in this area is to understand the instructions given by humans in natural language and act on them. A large number of researchers are working in this area which is also the aim of this paper. This paper focuses on creating a learning model which will carry out tasks given by humans in simple natural language.

1.2 Objective

The main objective of this paper is to implement the baseline model and the state-of-the-art solution for the vision-and-language tasks [10]. This paper also

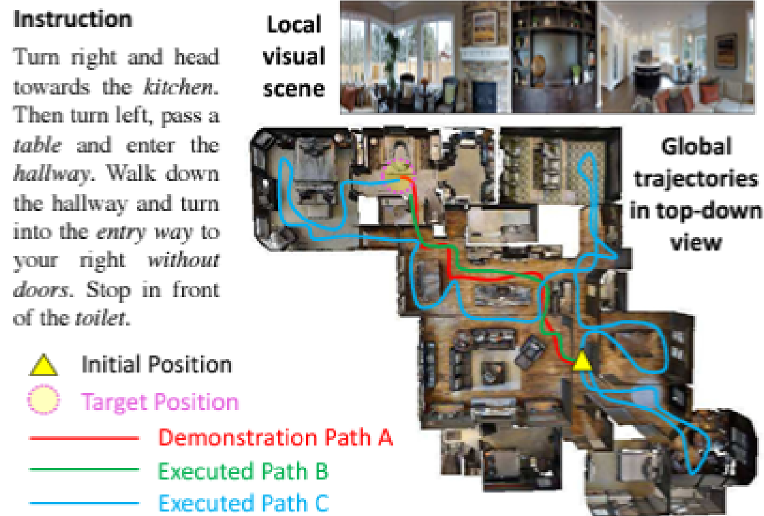

Figure 1.1: Demonstration of the VLN task. The instruction, the local visual scene, and the global trajectories in a top-down view is shown. The agent does not have access to the top-down view. Path A is the demonstration path following the instruction. Path B and C are two different paths executed by the agent.[4]

proposes a transformer-based model based on the latest research which may work better than the state-of-the-art solution. The baseline model is the simple sequence-to-sequence model[9] and the cross-modal attention[9] is the state-of-the-art model. Both the models aim to solve the VLN task which is understanding the instructions, observing the field of view, and then performing actions based on it while avoiding the obstacles in the path.

There are two main challenges in the VLN task. The first challenge is reasoning over images gathered and natural instruction provided. As demonstrated in figure 1.1, to reach the goal position, the agent needs to map the instructions, which are provided in the sequence of words, to the local visual space and match the instructions to the trajectory in the global space. The second main challenge is providing "Success" feedback. The agent successfully reached the goal should only be said when the agent reaches the goal and also closely follows the expert demonstration. For example, as shown in figure 1.1, path C can not be said as successful as even if it reached the goal position, it did not follow the expert path. In the case of path B, it follows the expert path but it did not reach the goal, therefore it is also not successful.

1.3 RxR-Habitat Competition

The research and engineering of intelligent systems(robots and egocentric personal assistants) with a physical or virtual embodiment is known as embodied AI. Habitat Challenge is an annual autonomous navigation challenge which is hosted by the EvalAI platform that aims to accelerate the progress in embodied AI. The first Habitat Challenge was held in combination with Habitat: Embodied Agents Challenge and Workshop at CVPR 2019 and from 2020, Habitat Challenge is held in combination with Embodied AI workshop at CVPR.
Vision-and-Language navigation in Continuous Environment(VLN-CE) is an instruction-guided navigation task with crowdsourced instructions, realistic environments, and unconstrained agent navigation. The RxR-Habitat Competition challenges to build agents for VLN-CE tasks by using Matterport3D environment using the Habitat Simulator. The RxR-Habitat competition was first proposed as part of the CVPR 2021 Embodied AI Workshop but is now open on an ongoing basis. The core challenge of the RxR-Habitat challenge is to build an agent that can navigate into the previously unseen environment following the natural language commands. For this challenge, RxR Dataset is created which contains human-annotated navigation instructions in multiple languages(English, Hindi, and Telugu), matched to navigation paths through reconstructed buildings from the Matterport3D dataset. This dataset is the largest dataset present for the Vision-and-Language Navigation in Continuous Environment.

1.4 Layout of the Thesis

The structure of the paper is as follows: Chapter 2 will introduce the RNN models and Attention models. It will also provide the background knowledge for simultaneous localization and mapping, object detection.
Chapter 3 will provide the design of the environment including the simulator and will show the design of learning agents. Chapter 4 will describe how agents are trained in the simulated environment on the Matterport3D dataset. It will also provide information regarding the settings of the training environment.
Chapter 5 will show how agents are evaluated. It will also present the results from

the test and discuss them. Chapter 6 will conclude the paper by summarizing all the models, results obtained and will present the future work.

Chapter 2

Background

2.1 Chapter Overview

To carry out the VLN task, the agent needs to use different technologies and algorithms. This chapter will provide an overview of various technologies which are used in the implementation process. The basic architecture of the VLN agent contains 3 major components:

- **Learning model**: The learning model is used to train the model to generate accurate actions based on given input RGB, Depth, and instructions. Since input data in the VLN task is in sequence because the agent receives new RGB and Depth observation at every timestep, Recurrent Neural Network(RNN) models are used in most of the learning models.
- **Navigator**: The task of the navigator is to implement an obstacle avoidance system and to create the map of the environment simultaneously positioning itself in that map.
- **Object Detection**: To carry out navigation based on instructions, the agent needs to detect the object present in the RGB frame at a given timestep.

The architecture shown in figure 2.1 provides a highlevel overview of the VLN agent.

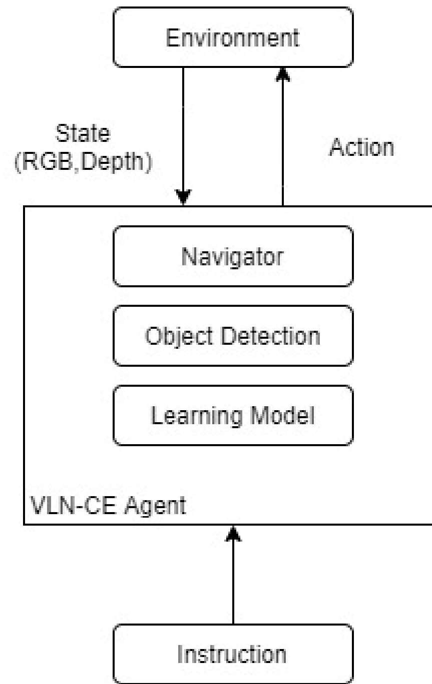

Figure 2.1: VLN Agent Architecture

2.2 Recurrent Neural Network

Recurrent Neural Networks(RNN) are state-of-the-art algorithms for sequential data. It is the first algorithm with an internal memory that remembers its input, making it ideal for machine learning problems involving sequential data.
In RNN, every node receives two inputs, one is input from the current state which is new input, and one from the previous node which is the hidden state. The internal working of RNN can be explained as:

1. Takes input as an input sequence
2. Applies the same set of operations to each input
3. Carry the internal state to remember the underlying pattern
4. Generates output as a sequence

The RNN model stores parameterized matrices corresponding to each connection. It stores the weight matrix for connections: input to hidden, hidden to hidden, and hidden

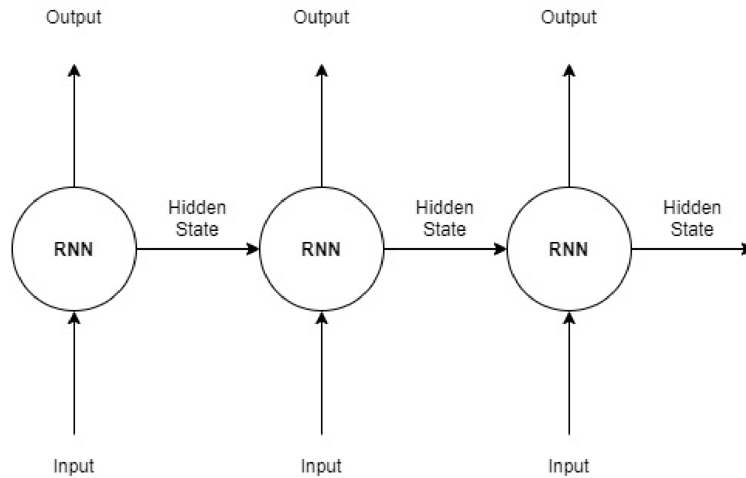

Figure 2.2: RNN Architecture

to output. All weights are shared across the time. Let's consider x(t) as an input taken at time t. For example, x1 could be input at time-step 1. The hidden state h(t) can be defined as follows:

$$h(t) = f(U * x(t) + W * h(t-1)) \tag{2.1}$$

Where U is the weight matrix for input to hidden connection and W is the weight matrix for hidden to hidden connection.
The goal of the RNN model is to use output and compare it with the test data which is usually a subset of original data. By using the difference between output and test data, the error rate is calculated. With the help of input, output, and error rate, the Back Propagation Through Time(BPTT) technique is applied. BPTT checks the data backward and based on the error rate, weights are adjusted. BPTT aims to reduce the error rate to make better predictions.
However, RNN has a major issue known as the vanishing gradient problem. This problem makes the learning of larger sequences difficult. Usually, gradients convey information that is used in RNN parameter updates, and when the gradient shrinks, the parameter updates become weaker, implying that no meaningful learning is taking place. This change is computed using multiplicative mathematics, which means that the gradient determined in very old step in the neural network is multiplied back via the weights earlier in the network. The gradient calculated in the past gets "diluted"

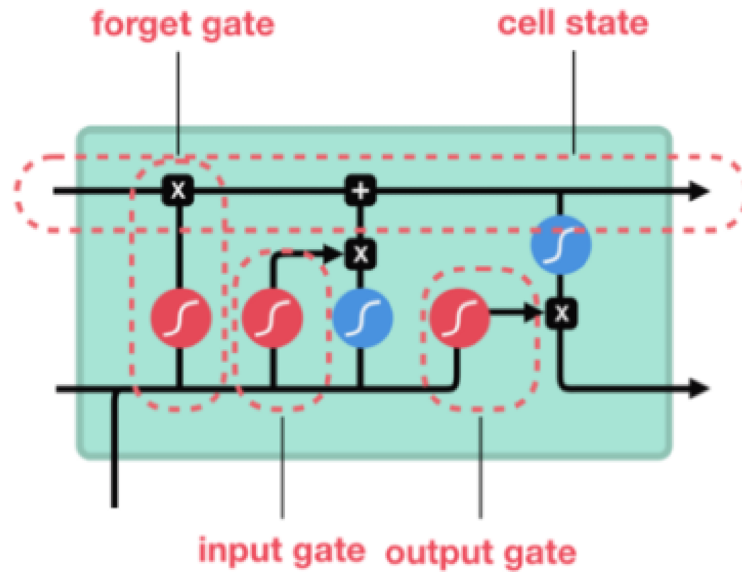

Figure 2.3: LSTM Architecture

as it travels back through the network via BPTT, which might cause the gradient to vanish, giving the issue its name.

2.2.1 LSTM

A Long-Short Term Memory (LSTM) [11][12] has a similar control as the RNN network. It processes the data at time t as it propagates through time. The difference lies inside the LSTM network. The figure 2.3 shows the inner architecture of the LSTM network. These operations allow LSTM to keep or forget the information.

The main component of the LSTM network is its cell state and gates. The job of the cell state is to carry the data from one node to another. The cell state also works as a memory. It keeps the data all along the sequence. As the cell state goes from node to node, either new data is added or old data is removed. This addition and removal of data take place via gates.

Forget Gate: The forget gate decides which data needs to be kept and which data should be removed. The data from the previous node's hidden state and the data from the current input are passed through the sigmoid function. The task of the sigmoid function is to flatten the values to always be between 0 and 1. If the value is closer to

zero means forget or if the value is closer to 1 means keep.

Input Gate: The job of the Input gate is to update the data in the cell state. The data from the previous node's hidden state and the data from the current input are passed through the sigmoid function which transforms the values between 0 and 1. If the value is closer to zero means not important or if the data is closer to 1 means important. The same data from the previous node's hidden state and the data from the current input are passed through the tanh function which flattens the data between -1 to 1 to regulate the data throughout the sequence. Then data from the tanh function output is multiplied with the output of the sigmoid function. The sigmoid outputs decide which data to be kept and which to be removed from the tanh output.

Cell State: The cell state from the previous state gets point-wise multiplied by the output of the forget gate. This has the possibility of removing the values based on the output of the forget gate if the values are closer to zero. Then, this data is point-wise added to the output of the input gate which gives the updated values of the Cell State.

Output Gate: The data of the next hidden state is decided by Output Gate. The hidden state contains data from the previous state and is also used for prediction. First, the data from the previous hidden state and current input is passed through the sigmoid function. Then, the updated cell state data is passed through the tanh function and the output is multiplied with sigmoid output to decide what data should be present in the hidden state.

To summarize the working of the LSTM, the Forget gate decides what data from the previous nodes to be kept and what data to be removed. The Input gate decides what data is relevant from the current input. The output gate determines what data is to be present in the next hidden state.

2.2.2 GRU

The Gated Recurrent Units(GRUs)[13][14] is the new generation of RNN and are similar to the LSTM. In GRU, the cell state is removed and the hidden state is used to transfer the data. GRU network contains only two gates, a reset gate and an update gate.

Reset Gate: The reset gate is used to decide how much past data is to be forgotten. It concatenates the current input and previous hidden state and it is passed through

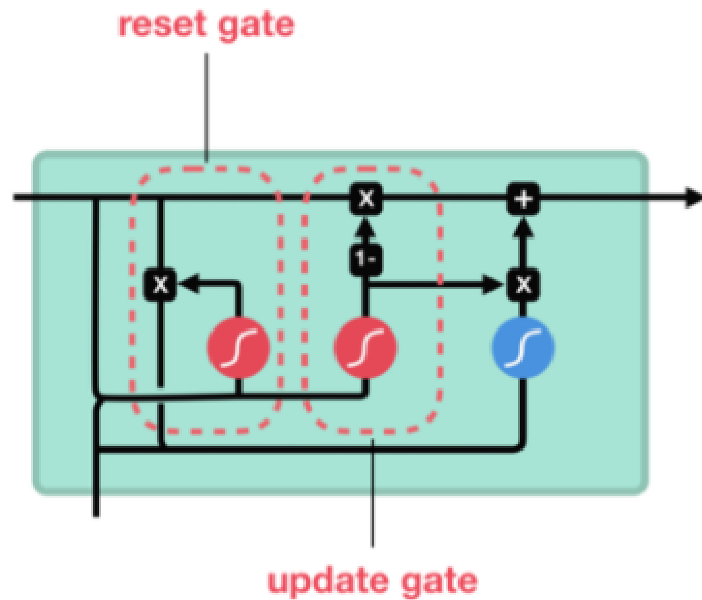

Figure 2.4: GRU Architecture

the sigmoid function. The output of the sigmoid functions is pointwise multiplied by the current input data.

Update Gate: The update gate acts similar to the forget gate and input gate of the LSTM. It decides what data to be removed and what new data to be added.

2.2.3 Teacher Forcing

Teacher forcing is a method used for training the RNN models that uses the ground truth as an input instead of the actual output of the previous step. While training, the traditional RNN model uses the output of the previous step as input for the current step. Teacher forcing works by using the actual output from the training dataset at the current time step as input in the next time step.

The figure 2.5 shows an example of how the teacher forcing technique is used. In the first step, the model predicted the action as "Forward" which is the correct action. It is passed as it is to the next step. In the second step, the model predicted the output action as "Left" which is incorrect. Now, instead of passing the predicted output to the next time step, the correct output is picked from the ground truth and the next

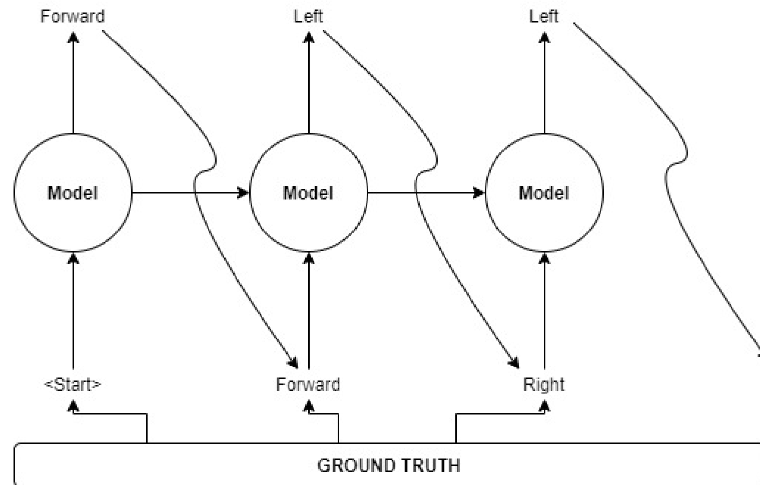

Figure 2.5: Teacher Forcing

action is predicted.
Using the Teacher Forcing technique reduces the time required for training the RNN model but it also raises an issue known as Exposure Bias or Train-Test Discrepancy. While training the model, ground truth is provided which increases the training speed but while testing, the ground truth is not present which may cause the model to perform poorly.

2.3 Attention Model

Attention[5] models are neural network input processing strategies that allow the network to focus on specific features of a complicated input one by one until the entire dataset has been processed. The goal of the attention model is to divide the entire problem into smaller tasks that are processed sequentially. This is exactly how the human mind works - by dividing the complicated problem into smaller simple tasks and solving them one by one. Attention models require Back Propagation Through Time(BPTT) to be effective. For example, as shown in the figure 2.6, one word attends to other words in the same sentence differently. When the input is received as "eating" then the next encounter with a food word is expected. The color of food, "green", describes the food but does not explain the "eating".

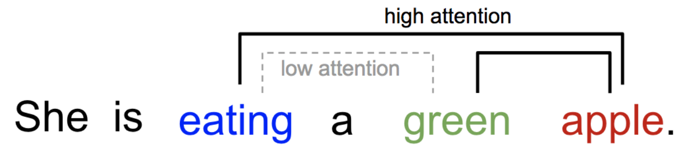

Figure 2.6: One word "attends" to other words in the same sentence differently

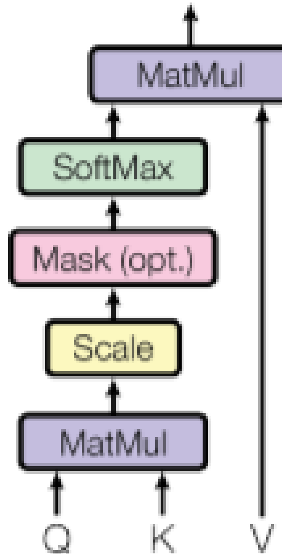

Figure 2.7: Self-Attention Model[5]

2.3.1 Self-Attention

Self-Attention[5], also known as intra-attention is an attention mechanism where attention is calculated for all input with respect to one input.
The figure 2.7 shows the basic architecture of the Self-Attention Model. The steps in the self-attention model are as follows:

- The first step is calculating the weight matrix(calculated while training phase) of each input to produce the input vectors: a query vector(Q), key vector(K), and value vector(V)

- Multiply query vector of current input with the key vector of other inputs
- Divide the result by the square root of the dimensions of the key vector. If the result is high then it causes the self-attention value to be small. To avoid this issue, the result is divided.
- The Softmax function is applied to the results with respect to the query input
- Multiply the result with the value vector
- Sum up the value vectors, this will give the self-attention value of the given input

The mathematical representation of the self-attention model is:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_K}})V \tag{2.2}$$

Where Q, K, and V are Query, Key, and Value Vectors. d_K is the dimension of K.

2.3.2 Multi-Head Attention

In multi-head attention, the scaled product of the attention model is run multiple times. The independent attention outputs are concatenated and transformed into the desired dimension. According to [5], multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions. The steps in the multi-head attention model are as follows:

- Pass each input through embedding
- Create various attention heads h with different weight matrices W(Q), W(K), W(V)
- Multiply input matrix with each weight matrices to produce key, query, and value matrices for each head
- Apply attention to all matrices to generate output matrix from each head
- Concatenate the output matrix from each head and apply dot product with weight W0 to generate the final output

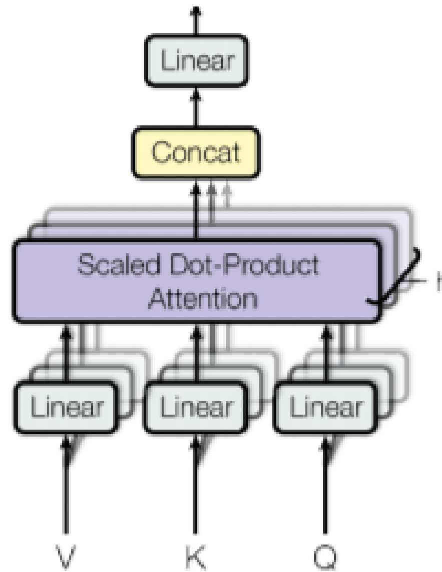

Figure 2.8: Multi-Head Attention Model[5]

Mathematically the multi-head attention model is represented as:

$$MultiHead(Q, K, V) = Concat(head_1, head_2, head_3.\dots.head_n) * W_0 \tag{2.3}$$

$$Where, head_i = Attention(QW_i^Q, KW_i^K, VW_i^V) \tag{2.4}$$

2.4 Transformer

In [5] paper, authors introduced an encoder-decoder model based on attention layers, termed as Transformer model. The main difference between the Transformer model and the RNN model is that in the Transformer model, the data can be passed parallelly to increase the effective use of GPU and reduce the training time whereas in RNN model, data is passed sequentially. The Transformer model also uses a multi-head attention layer which removes the vanishing gradient problem.
The figure 2.9 shows the architecture of the transformer model. As you can see, the model contains two major components, Encoder Block and Decoder Block.

Encoder Block

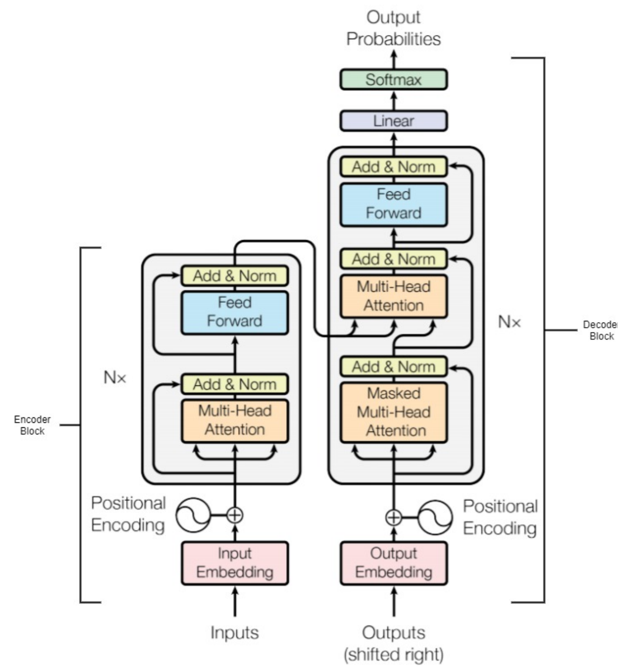

Figure 2.9: Transformer Architecture

Following are the steps which take place in this block:

- First, inputs are passed through Embedding Space. The Embedding Space is the dictionary where inputs with similar meanings are kept together. When the input is passed through the embedding space, it converts it into vectors based on the dictionary.
- Then the vectors are passed through Positional Embedding. The input usually has different meanings based on the position of the input in the sequence. The positional embedder converts the vectors based on their position.
- In the next step, the embedded vectors are passed through the Multi-Head Attention Block. It creates the attention vectors based on what part of the input should be focused on.
- The attention vectors are passed through the Feed-Forward layer which transforms the vectors in form required by the next layer. Unlike RNN,

attention vectors are independent of each other. So, data can be passed parallelly. The output of this layer is Encoded Vectors.

Decoder Block

For the training purpose, the actual results are required to be passed for the model to learn. Following are the steps which take place in Decoder Block:

- The outputs are passed through Embedding space and Positional Embedding to generate embedded vectors that can be easily understood by the computers.
- The embedded vectors are passed through the Multi-Head Attention layer and the output of this layer is masked. The reason behind the masking of the output is to avoid the model knowing the actual output in advance.
- The masked attention vectors and the Encoded vectors from Encoder Block are passed through the Multi-Head Attention Layer together. The output of this block is the attention vector for each actual input and actual output.
- The attention vectors are passed through the Feed-Forward layer which transforms the vectors in form required by the next Linear Layer which expands the dimension of the vector.
- Now, the results are passed through the softmax layer which transforms the input into a probability distribution, which is human interpretable. And the resulting output is produced with the highest probability

Transformer Models are faster than the RNN model which makes them better than the RNN. The transformer utilizes more GPU and memory as it passes the data parallelly.

2.5 SLAM

Simultaneous Localization and Mapping(SLAM) is a problem of generating and updating a map of an unknown 3D environment while simultaneously keeping a track of the agent's location in the generated map. There are many algorithms present to solve the SLAM problem. ORB-SLAM and LSD-SLAM are popular open-source SLAM algorithms.

Figure 2.10: Multi-session stereo-inertial ORB-SLAM3[6]

2.5.1 ORB-SLAM

ORB-SLAM[15] is a SLAM solution for Monocular, Stereo, and RGB-D cameras. It is one of the most adaptive and accurate solutions for SLAM. ORB-SLAM is currently a state-of-the-art solution to the SLAM problem. It can compute the camera trajectory and a sparse 3D reconstruction of the scene in real-time for a wide range of situations, from modest hand-held desk sequences to a car driven over multiple city blocks. It can complete enormous loops and execute global relocalization from a wide range of baselines in real-time. It comes with a robust and automatic initialization from both planar and non-planar scenarios. ORB-SLAM3[6] is the third version of ORB-SLAM released in 2021. ORB-SLAM3 is the first real-time SLAM framework that can do Visual, Visual-Inertial, and Multi-Map SLAM with monocular, stereo, RGB-D, and pin-hole, and fisheye lens models with monocular, stereo, and RGB-D cameras. ORB-SLAM3 is as robust and more accurate than the best systems available in the literature for all sensor configurations.

The figure 2.10 shows the example of trajectory generated using ORB-SLAM3 with Multi-Session Stereo-Internal. In red, the trajectory is estimated after single-session processing outdoors. In blue, multi-session processing of university is done first, and then outdoors.

2.6 Navmesh

A navigation mesh, often known as a navmesh, is an abstract data structure used in artificial intelligence applications to help agents navigate across complex areas. This method has been used in robotics since the mid-1980s, where it is known as a meadow map, and was popularized in video game AI in 2000. A navigation mesh is a two-dimensional convex polygon collection that defines on which sections of an environment agents can navigate. An agent can navigate freely within these areas unobstructed by obstacles. Because the polygon is convex and traversable, pathfinding within one of these polygons is simple and in a straight line. Any graph searching methods, such as A*, can be used to locate paths between polygons in the mesh. As a result, agents on a navmesh can skip computationally costly collision detection checks with barriers in the surroundings.
Navigation meshes can be created manually, automatically, or by some combination of the two. The figure 2.11 shows the example of the navmesh. The green-colored cylinder is an agent, the red-colored area is a non-walkable area and the remaining blue-colored area is walkable. The agent can walk on the white dotted path to reach the goal without any collision or obstacle.

2.7 Object Detection

Object Detection is the task of identifying the objects of a certain class from the image. This task requires the algorithm to find from where the instance of an object starts and where it ends. It also requires drawing a bounding box around the object and classify it within the known classification labels. Before using a neural network for object detection, feature extraction algorithms like SIFT were used. The models trained using neural network gives better result than the traditional SIFT feature extraction.

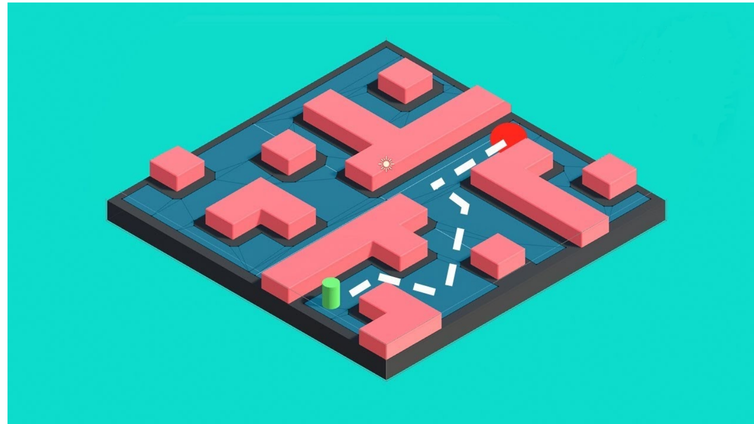

Figure 2.11: Navigation Mesh(NavMesh)

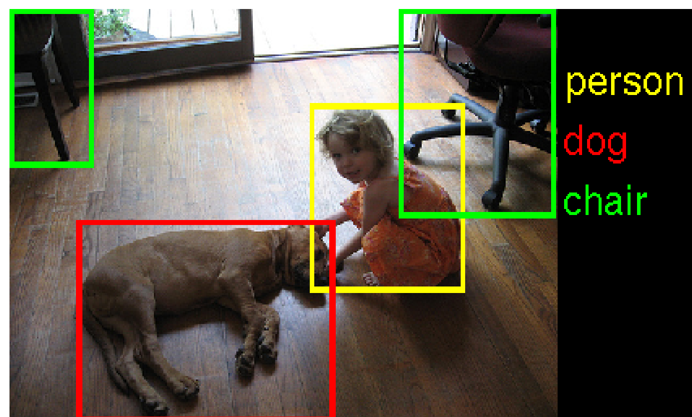

Figure 2.12: ImageNet Object Detection [7]

To train the model, a large image dataset is required. There are various pre-trained models present which are trained on many open-source image datasets.

2.7.1 Image Dataset for Object Detection

To train the object detection models, a large number of images of respective classes are required. The availability of such kinds of image datasets plays a crucial role in training the model. Access to these large datasets allows the deep learning models to capture the diversity of objects in different environments. There are many open-source image detection datasets present such as PASCAL VOC[16], MS COCO[17], ImageNet[7],

and Open Images Dataset[18].
Frames per second (fps), precision, and recall also known as true positive rate are the evaluation criteria for object detection. The average precision (AP) is the most commonly used assessment metric for object detection of a single object class.The mean AP (mAP) is used to compare performance across all object categories. The "intersection over union" (IoU) method is used to assess object localization accuracy. This is the percentage of the anticipated bounding box that overlaps with the ground truth box. Typically, the ratio exceeds a predetermined threshold of 0.5.

$$IOU(b, b_g) = \frac{area(b \cap b_g)}{area(b \cup b_g)} \tag{2.5}$$

where b and b_g is the predicted bounding box and the ground truth bounding box respectively.

2.7.2 ResNet50

The Convolutional Neural Network (CNN, or ConvNet) is a class of deep neural networks that is most typically used to analyze visual data. ResNet-50[19] is a type of CNN that is a pre-trained Deep Learning model for image categorization. ResNet-50 is a deep neural network with 50 layers that was trained on a million photos from the ImageNet database for 1000 categories. In addition, the model comprises approximately 23 million trainable parameters, indicating a dense architecture that improves image identification. When compared to building a model from scratch, where you must collect large amounts of data and train it yourself, using a pre-trained model is a highly effective option.

Chapter 3

Design

This chapter will present how the system is designed. It will also provide information about how the agents are designed including the model design. It will even provide how a train-test environment is created.
The agents are implemented in Habitat Lab with the help of different models. These models contain various Recurrent Neural Networks, Encoders, and Object Detection Algorithms. For Object Detection, the ResNet50 model is used which is trained on ImageNet Dataset. The implemented model is trained and tested in Habitat Sim. For training and testing of the model, the RxR Dataset is used which is generated on the Matterport3D scene dataset.

3.1 Habitat Simulator

Habitat-Sim is a high-performance physics-enabled 3D simulator designed by the Facebook AI Research team. It supports 3D scans of indoor/outdoor spaces like MatterPort3D [20] or Gibson [21]. It also supports CAD models generated via ReplicaCAD [22] and objects generated by Google Scanned Objects [23].
The figure 3.1 shows the architecture of the Habitat-Sim main classes. The simulator transfers all 3D environment-related tasks to the ResourceManager. The ResourceManager is responsible for generating the 3D objects and materializing them by using Textures and Shaders and then loading them to create the 3D environment for the simulator. These resources are utilized at the level of individual SceneNodes

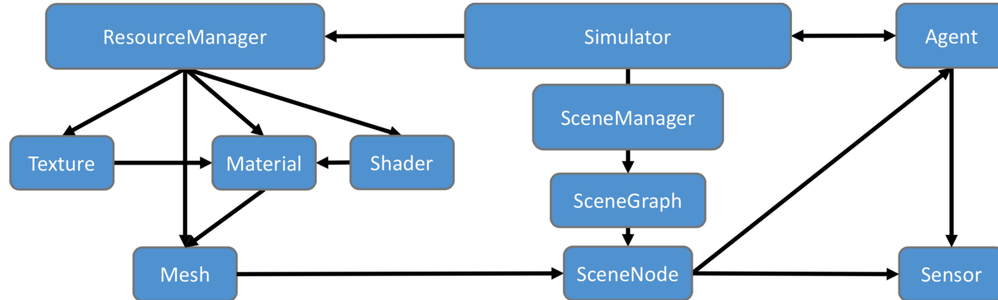

Figure 3.1: Habitat-Sim Architecture [1]

in SceneGraphs to represent unique items or regions in a Scene. Agents and their Sensors are created by attaching them to SceneNodes within a SceneGraph.
Habitat-Sim uses a hierarchical scene graph to generate a simulation of the 3D environmental datasets. The use of scene graphs allows it to abstract the features from the given datasets and makes it easy to generate a 3D simulation of the scene, scene editing, and scene manipulation. The Habitat-Sim supports cross-platform deployment by using the Magnum graphics middleware library. It generates all sensor outputs, RGB camera sensors, depth sensors, and semantic mask sensors, in a single pass which removes any additional overhead. The figure 3.2 shows sensor outputs of the Habitat-Sim.

	1 proc			**3 proc**		
Sensors / Resolution	**128**	**256**	**512**	**128**	**256**	**512**
RGB	4093	1987	848	10638	3428	2068
RGB + Depth	2050	1042	423	5024	1715	1042
RGB + Depth + Semantics	709	596	394	1312	1219	979

Table 3.1: Performance of the Habitat-Sim on an Intel Xeon E5-2690 v4 CPU and Nvidia Titan XP GPU(in FPS)

Habitat-Sim prioritizes the simulation speed to achieve greater frames per second (FPS) while rendering a scene. It achieves 10,000 FPS on Matterport3D scene dataset with multi-process on a single GPU. The table 3.1 shows the performance of the Habitat-Sim on an Intel Xeon E5-2690 v4 CPU and Nvidia Titan XP GPU. The table 3.2 shows the comparison between various simulators based on speed. Based on TensorFlow

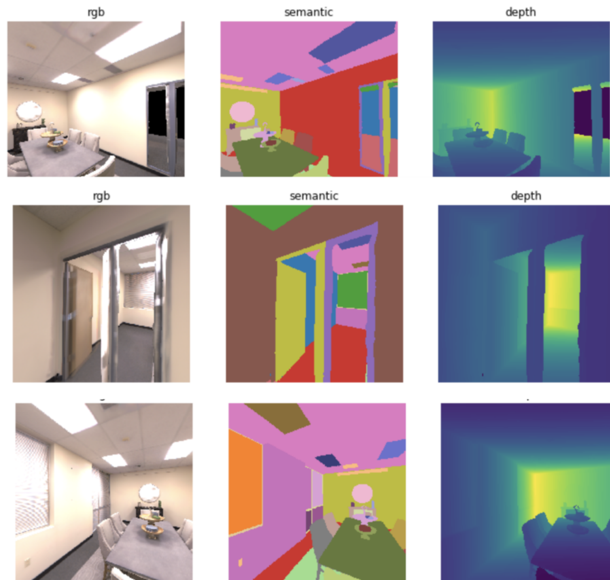

Figure 3.2: Habitat-Sim Sensor Outputs [1]

benchmarks, Habitat-Sim has the highest FPS rate compared to other simulators on a single GPU.

Simulator	**Frames Per Second(FPS)**
AI2-THOR	10
CHALET	10
MINOS	100
Gibson	100
House3D	300
Habitat-Sim	10000

Table 3.2: Comarision of Simulators[1]

To summarize, Habitat-Sim is a high-performance 3D simulator that can load 3D scenes into a standardized scene-graph format, configure agents with numerous sensors, simulate agent mobility, and return sensory input from the sensor suite of an agent.

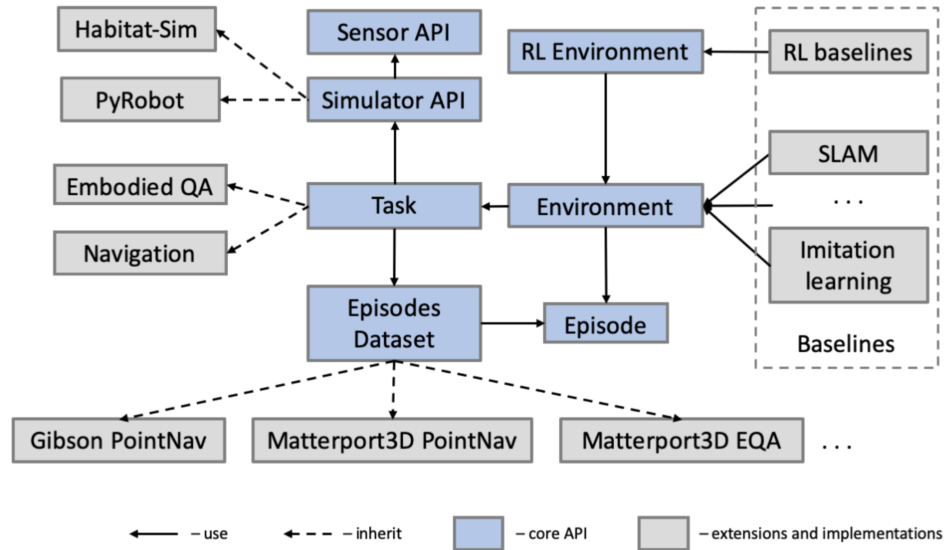

Figure 3.3: Habitat Lab Architecture[1]

3.2 Habitat Lab

Habitat Lab is developed by the Facebook AI Research team to use Habitat-Sim. It is a high-level library designed to build embodied AI agents. It is used to define tasks for the agent, configure the agent's sensors, train these agents and monitor their performance. The main benefit of Habitat Lab is the ease of use. It is very easy to create a new agent or add/remove sensors for the existing agents. The figure 3.3 shows the architecture of the Habitat Lab.
The following are the key concepts of the Habitat Lab:

- **Env**: This acts as a base class for other classes. It connects 3 major components: a Simulator, Datasets, and Task. All the information required to execute a task on a simulator is present inside the environment.

- **Dataset**: It includes a list of task-specific episodes from a given data split, as well as extra information. It also handles loading and storing a dataset to disk, as well as retrieving a list of scenes and episodes for a certain scenario.

- **Episode**: An episode contains all the information about the task. It includes the agent's initial position, its orientation, the goal position, and a scene ID.

- **Task**: This class defines the condition for an episode to terminate. For example, the episode can be terminated if the agent is stuck or it reached the goal position.
- **Sensor**: It provides all the data from the sensors in the required format.
- **Observation**: This class is used to show the data from the sensors. It includes data from various sensors like RGB Camera Sensor, Depth Sensor, collision sensor, etc.

Habitat Lab presently uses Habitat-Sim as the core simulator, although the simulator backend is designed with a modular abstraction to ensure interoperability across other simulators.

3.3 Dataset

The Room-Across-Room(RxR) Dataset [24] is a new vision-and-language(VLN) dataset with multilingual(English, Hindi, and Telugu) support and is larger than any other VLN dataset. The RxR Dataset contains 126K human-annotated instructions for 16.5K paths. It includes 14K paths for each language(English, Hindi, and Telugu) making it 42K paths and each path has 3 Instructions making the dataset total of 126K instructions. The table 3.3 shows the comparison between different VLN datasets.

Dataset	**Languages**	**Instructions**	**Paths**
CVDN	1	2K	7K
R2R	1	22K	7K
Touchdown	1	9K	9K
REVERIE	1	22K	7K
RxR	3	126K	16.5K

Table 3.3: Comarision of VLN Datasets[2]

The RxR instructions are collected by using spatiotemporal grounding. While generating the instructions, the guides are directed to move while speaking the instructions and their position is also recorded with the time alignment. For example, in figure 3.4, the path is multicolored and instructions for that path are also

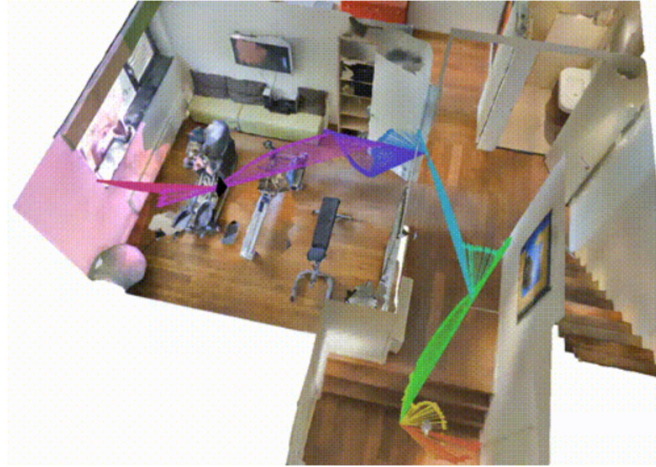

Figure 3.4: RxR Dataset with Spatiotemporal Grounding[8]

multicolored. For example, while giving the instruction the guide was present at the respective color of the instruction and the path. While recording the instruction "This is gym room, move forward" which is in blue, the guide was present at the blue colored path. The time was also recorded providing the spatiotemporal grounding of the instructions.

3.4 VLN Agents

The models for VLN agents are built by using a combination of various neural networks and attention layers. The base code provided by [9] consists of two models: Seq2Seq Model and CMA Model. The [3] proposes a new model by using a Transformer tested on Alfred Dataset. The models are trained using the Teacher Forcing technique explained in chapter 2.

The basic structure of a VLN agent is shown in figure 3.5. The agent receives three inputs - 2 inputs(RGB Camera and Depth) at every timestep and one input(Instructions) at the start. These inputs are passed through the model and

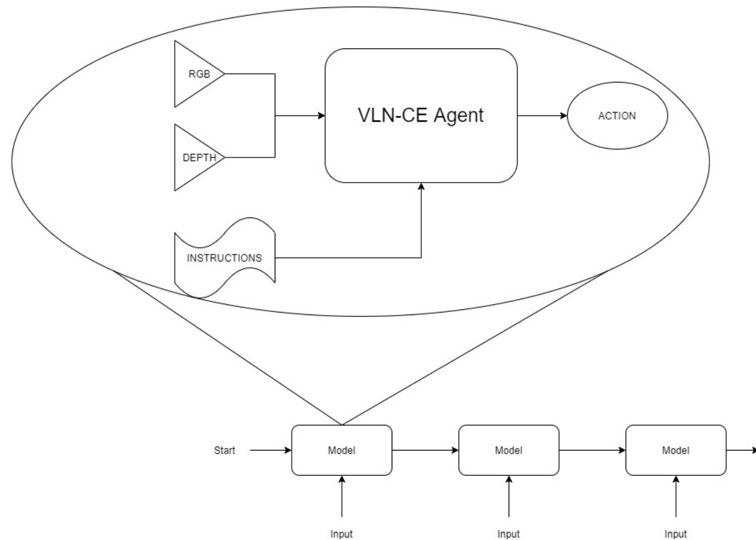

Figure 3.5: Basic structure of a VLN agent

action is generated. The generated action is then executed in the simulated environment to receive the next set of inputs. This process is repeated until either the agent reaches the goal or the episode gets terminated due to the agent being stuck or the minimum number of steps is crossed.

3.4.1 Seq2Seq Model

The Seq2Seq Model[9] is a simple sequence-to-sequence model which is built as shown in figure 3.6.

The model consists of the recurrent network which takes encoded RGB, Depth, and instruction inputs, and generated action as an output at every timestep t. For encoding the input, the following encoding standards are used:

Instruction Encoding: The input instructions are first passed through the tokenizer to create tokens which are then passed through GLoVE [25] to generate respective embedded tokens for the instructions. Then these tokens are passed to any required RNN(LSTM or BiLSTM or GRU).

RGB Encoder: The RGB inputs are passed through the Object Detection algorithm to collect visual features. In this model, to generate visual features, the RGB inputs are processed through the ResNet50 [19] model which is pre-trained on ImageNet Dataset.

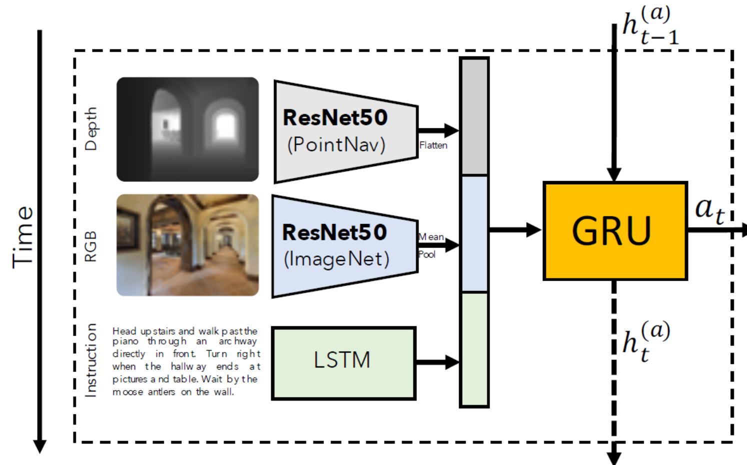

Figure 3.6: Seq2Seq Model[9]

Depth Encoder: For Depth encoding, a modified ResNet50 was used that was trained to perform navigation to a location given in relative coordinates. These weights were trained using Decentralized Distributed Proximal Policy Optimization(DD-PPO) method.
The embedded instruction tokens are denoted as $w_1, w_2 \ldots, w_T$ for time length T. The RGB visual features are denoted as V=vi and depth features are denoted as $D = d_i$ where i is the index. The Seq2Seq model can be represented as:

$$\bar{v_t} = mean - pool(V_t), \bar{d_t} = mean - pool(D_t), s = LSTM(w_1, w_2 \ldots, w_T) \tag{3.1}$$

Where s is the final hidden state of the LSTM encoder. The final hidden state and the action can me computed as:

$$h_t^{(a)} = GRU([\bar{v_t}, \bar{d_t}, s], h_{t-1}^{(a)}) \tag{3.2}$$

$$a_t = argmax_a \quad softmax(W_a h_t^{(a)} + b_a) \tag{3.3}$$

This simple model establishes the baseline model for the VLN-CE task.

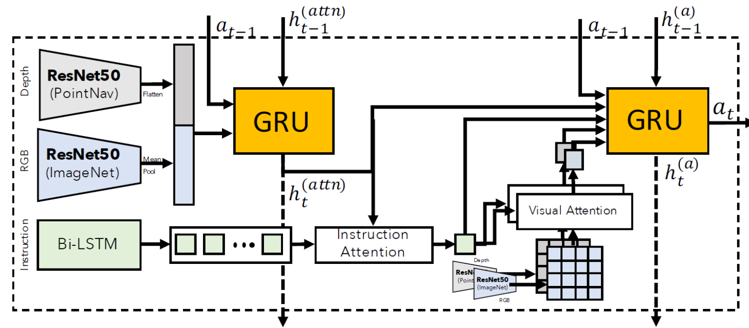

Figure 3.7: CMA Model[9]

3.4.2 CMA Model

The Cross-Modal Attention(CMA) Model[9] is more complicated than the baseline Seq2Seq Model and generates better results than that. The human-annotated instruction includes references to the other objects, for example "opposite to the mirror" which is difficult to follow by using the mean-pool feature. In addition to this issue, the baseline model does not take already completed part of the instructions into consideration while predicting the next action. To overcome these issues, CMA Model is generated which is shown in figure 3.7.

In CMA Model, the RGB and Depth encoders are kept the same as the baseline model. The instruction encoder is changed from LSTM to Bi-LSTM encoder which is a bidirectional LSTM where one takes the instructions from start to end and the other takes from end to start providing both future and past instruction information.

Unlike the Seq2Seq model, the CMA model uses two recurrent networks, one for visual observations same as the Seq2Seq model and the other for the attended instructions and visual observations. The first recurrent network can be written as:

$$h_t^{(attn)} = GRU([\bar{v}_t, \bar{d}_t, a_{t-1}], h_{t-1}^{attn}) \tag{3.4}$$

Where a_{t-1} is learned embedding of the previous action.

The instruction encoder with BiLSTM network can be shown as:

$$S = \{s_1, s_2, ..., s_T\} = BiLSTM(w_1, w_2, \ldots, w_T) \tag{3.5}$$

The attended instruction feature st is then computed over these representations and

utilized to attend to visual ($\hat{v}_t$) and depth ($\hat{d}_t$) features. This process is done by applying the attention layer.

$$\hat{s}_t = Attn(S, h_t^{(attn)}), \hat{v}_t = Attn(V_t, \hat{s}_t), \hat{d}_t = Attn(D_t, \hat{s}_t) \tag{3.6}$$

The next recurrent network takes these features as an input and predicts the action.

$$h_t^{(a)} = GRU([\hat{s}_t, \hat{v}_t, \hat{d}_t, a_{t-1}, h_t^{(attn)}], h_{t-1}^{(a)}) \tag{3.7}$$

$$a_t = argmax_a \quad softmax(W_a h_t^{(a)} + b_a) \tag{3.8}$$

3.4.3 Episodic Transformer Model

Episodic Transformer(E.T.)[3] is an attention-based model which uses a multimodel transformer to solve Vision-and-Language(VLN) tasks. The E.T. Model uses a multimodel transformer to encode the instructions and all previous observations, and previous actions to predict the next action. The architecture of E.T. is represented in figure 3.8.

Unlike previous models where action from the previous state is used to determine the current action, E.T. uses full action history to compute the current action. The generalized E.T. can be represented as:

$$\hat{a}_t = f(s, v_{1:t}, \hat{a}_{1:t-1}) \tag{3.9}$$

Where the s is encoder instructions, $v_{1:t}$ is all the observations(RGB and Depth) from start till the current time t, and $\hat{a}_{1:t-1}$ is the history of actions taken till current time t.

Models	**Task**	
	Seen	**Unseen**
LSTM	23.2	2.4
LSTM + E.T. Language Encoder	27.8	3.3
E.T.	33.8	3.2

Table 3.4: Comarision of VLN Models on Alfred Dataset[3]

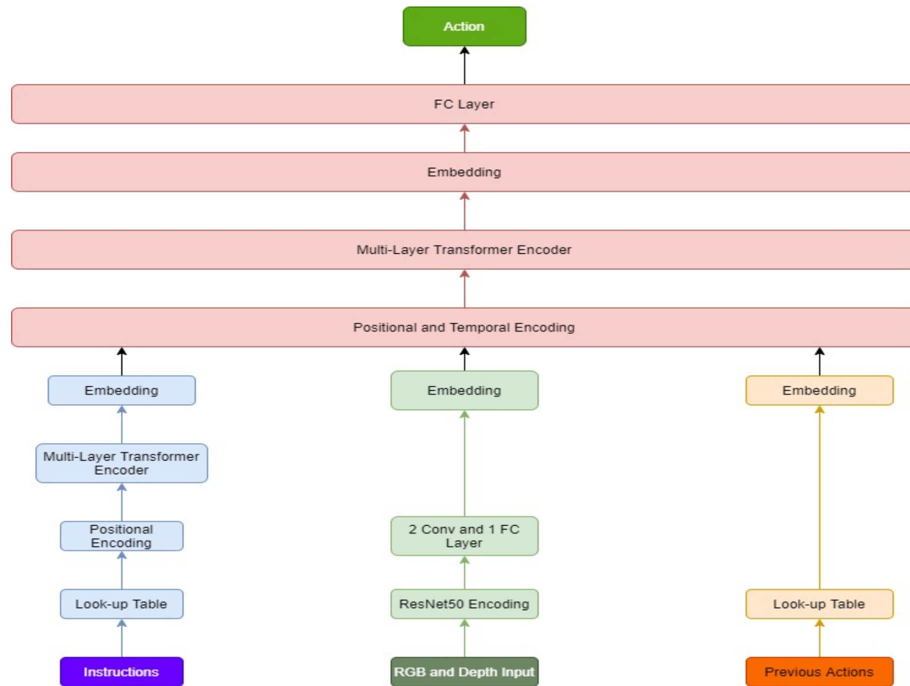

Figure 3.8: Episodic Transformer Model

In [3], the authors ran the E.T. model on the Alfred[26] dataset and compared it with other models. The results generated from the comparison are shown in the table 3.4 which shows that E.T. outperforms other models.
The E.T. model works better than the basic LSTM model and the combination of LSTM and E.T. model where the instruction encoder of E.T. is used with the other LSMT features.

Chapter 4

Implementation

This chapter provides the low-level implementation process of all the components and how the environment is created for the agents to train and test.
Currently, Habitat only supports Nvidia-Cuda-enabled systems because it uses GPU to render a scene. We have used Google Colab Pro to develop the system. The Google Colab Pro provides approximately 25 GB of RAM and NVIDIA TESLA P100 GPU with 16GB memory allocation.

4.1 Habitat Simulator

Habitat-Sim can be installed in 3 ways- By using Docker, By using Conda, or Directly from the Source. We installed the Habitat-Sim using miniconda3-4.5.4 and python version 3.7. The Habitat-Sim comes in 3 types: with attached display, headless, and with bullet physics. For the VLN task, we used a headless version of Habitat-Sim v0.1.7. After installing the Habitat-Sim, performance can be tested by executing the example script provided in the base code of Habitat-Sim[1][22]. After testing the Habitat-Sim on one of the Matterport3D scene datasets, we received the following performance:

- Resolution: 640 x 480
- total time 1.58 s,
- frame time 1.579 ms
- FPS: 633.1

4.2 Habitat Lab

To build a VLN agent and use it on Habitat-Sim, Habitat-Lab is used. The code and installation steps are available on GIT provided by [1]. Once the code is downloaded, all the required python packages and dependent packages are installed as explained by [1]. Once the Habitat-Lab is installed, it can be tested by using the example file which is provided in the code.
Agents are configured in Habitat-Lab using YAML config files. The config file defines:

1. Maximum number of steps
2. Sensors attached to the Agent with Sensor configuration
3. Step size and turning angle
4. Type of task
5. Success Criteria for the task that is the criteria for episode termination on success
6. Possible actions agent can take
7. Measurements for the evaluation
8. Path of the datasets to be used

After providing the correct config file to the agent, it can produce the results. For example, a config file is created for an agent to go to a particular location in the scene by using the shortest path. When the agent is executed, then the results generated are shown in figure 4.1

4.3 Dataset

To implement the system, we have used the following datasets:
Matterport3D: We have used 60 Matterport3D scene datasets for which VLN instructions were generated. These scene datasets are present under the tag of

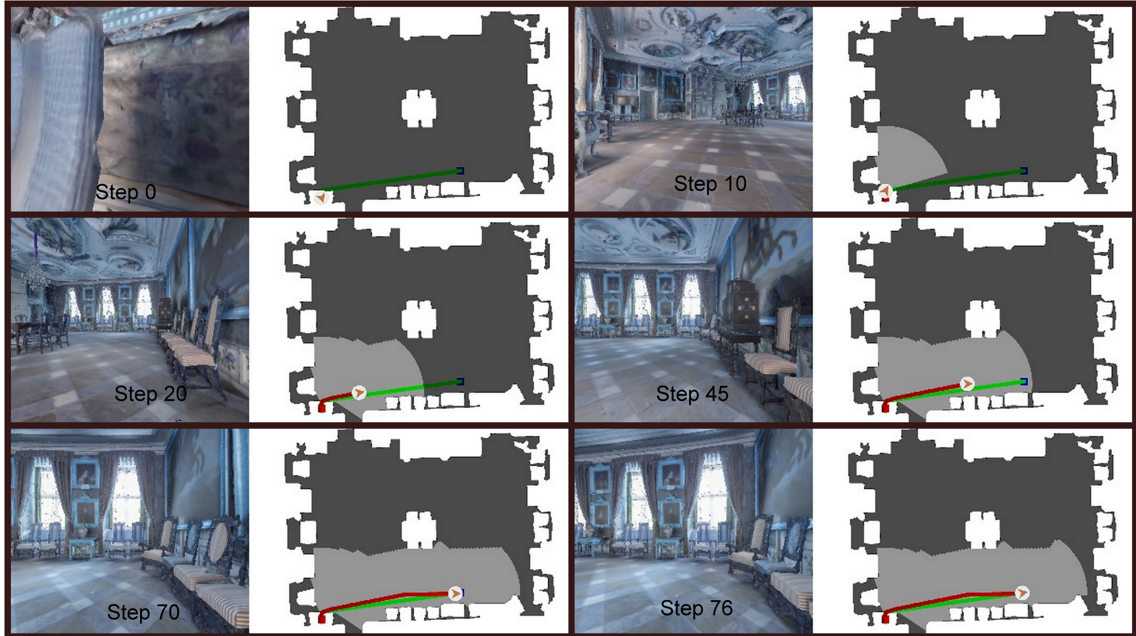

Figure 4.1: Habitat Output for Shortest Path(The left side of each image is the RGB input received by the agent and the right side is the top view of the path containing the path taken by the agent in red and ground truth in green.)

"habitat" in Matterport3D Dataset.

Path and Instructions: The instructions for the downloaded scenes are available on Google Cloud Storage. These instructions are divided into train, val_seen, val_unseen, and test_challenge. The respective path for each instruction in JSON format is also required. For the instruction encoder, pre-computed BERT text features are used.

Observation Encoder Weights: To encode RGB Camera Observations, ResNet50 model weights pre-trained on ImageNet dataset is used from TensorFlow. As explained in chapter 3, for depth encoder, weights from the modified ResNet50 model pre-trained for DD-PPO are used which are publicly available.

4.4 Training Environment

For training, the recollection_trainer class is created which performs the teacher forcing technique using the ground truth. The recollection trainer does not save the path on the disk.

While training or testing, for each episode, the scene is initialized in the Habitat-Sim for the respective instruction. After scene initialization, the NavMesh is generated. Since the scene is static, there are no other movements than the agent, NavMesh can be used for navigation, mapping, localization, and obstacle avoidance. For every scene, NavMesh is constructed and saved temporarily. The agent is positioned at the start location and instructions are passed to the agent.

Chapter 5

Results and Discussion

5.1 Evaluation

As explained in chapter 1, Evaluating the VLN agent is the most challenging issue. To overcome this issue, multiple criteria are used to define the success of the agent. The evaluation matrix consists of two evaluation criteria:
Primary – Normalized Dynamic Time Warping(NDTW)
Secondary – Success Rate(SR) and Success Weighted by Path Length(SPL)

Normalized Dynamic Time Warping(NDTW)

The primary evaluation criteria is to calculate the NDTW value of the agent. The NDTW value for the agent is calculated by measuring the distance between the agent's position at time t and the actual position at the same time from the ground truth. Larger the NDTW value of the agent, the better the agent performance of the agent. The figure 5.1 shows the example of how NDTW is calculated.
Let's consider there are 2 agents, agent 1(left) and agent 2(right) with the path q and ground truth r as shown in figure 5.1. In the case of agent 2, the distance between the agent's path and ground truth is increases at the end. Therefore, the NDTW value of agent 2 will be very small. Whereas, for agent 1, the distance between the agent's path and the ground truth is less throughout the time resulting in a larger NDTW value.

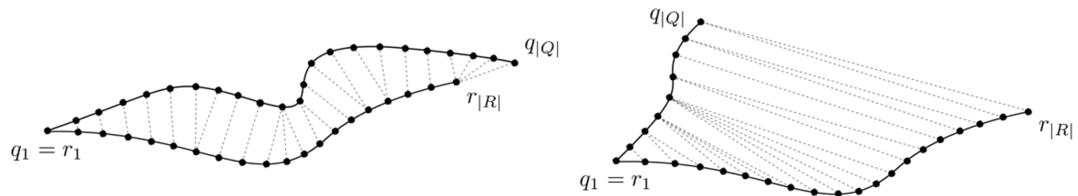

Figure 5.1: NDTW(Path taken by the agent 1(left) and agent 2(right))

Success Rate and Success Weighted by Path Length

As the name suggests, the success rate is calculated by dividing the number of episodes in which the agent reached the goal successfully by the total number of episodes.
Let l_i be the shortest path distance from the agent's starting position to the goal in episode i, and let p_i be the length of the path taken by the agent in this episode. Let S_i be a binary indicator of success in episode i. Then SPL is calculated as:

$$\frac{1}{N}\sum_{i=1}^{N} S_i \frac{l_i}{max(p_i, l_i)} \tag{5.1}$$

To understand it more clearly, let's consider there are a total of 100 episodes.

- If 50% of episodes are successful with the optimal path taken by the agent, then SPL will be 0.5
- If 100% of episodes are successful with twice the path taken by the agent, then SPL will be 0.5
- If 50% of episodes are successful with twice the path taken by the agent, then SPL will be 0.25

5.2 Results

Each model is trained with a different number of episodes depending on the memory limit of the system. After training the model, weights are generated which are then tested for 100 episodes. For this experiment, we only used English instructions.
Seq2Seq Model

Episodes for Iterations	Nvidia GTX 1650 4GB with 8GB RAM	Nvidia GTX 1080 8GB with 12GB RAM	Nvidia GTX 1650 4GB with 32GB RAM	Nvidia TESLA P100 16GB GPU with 25GB RAM
5000 episodes for 10 iterations	Out of Memory Error	Out of Memory Error	Out of Memory Error	Out of Memory Error
5000 episodes for 1 iteration	Out of Memory Error	Out of Memory Error	Out of Memory Error	Time Out
500 episodes for 1 iteration	Out of Memory Error	Out of Memory Error	Out of Memory Error	6 hrs
300 episodes for 1 iteration	Out of Memory Error	Out of Memory Error	Out of Memory Error	2 hrs
100 episodes for 2 iteration	Out of Memory Error	Out of Memory Error	12 hrs	1.5 hrs
100 episodes for 1 iteration	Out of Memory Error	Out of Memory Error	7 hrs	1 hr

Table 5.1: Execution of Seq2Seq Model on different machines

We tried to run Seq2Seq Model on various systems. Whether or not the model ran on the system with the time taken is provided in the table 5.1.
We took the weights generated by training the Nvidia TESLA P100 16GB GPU with a 25GB RAM system for 500 episodes for 1 iteration and tested it for 100 episodes. The results generated are as follows:

- NDTW – 0.273
- SR – 0.0
- SPL – 0.0

The RGB and depth output of one of the tested episodes with the instruction is shown in figure 5.2.

Figure 5.2: Seq2Seq Model output for Episode Number 436

Instructions for Iterations	Nvidia GTX 1650 4GB with 8GB RAM	Nvidia GTX 1080 8GB with 12GB RAM	Nvidia GTX 1650 4GB with 32GB RAM	Nvidia TESLA P100 16GB GPU with 25GB RAM
19K Instructions for 5 iterations	Out of Memory Error	Out of Memory Error	Out of Memory Error	Out of Memory Error
10K Instructions for 1 iteration	Out of Memory Error	Out of Memory Error	Out of Memory Error	Time Out
5K Instructions for 2 iteration	Out of Memory Error	Out of Memory Error	Out of Memory Error	Time Out
5K Instructions for 1 iteration	Out of Memory Error	Out of Memory Error	Out of Memory Error	8 hrs
500 Instructions for 2 iteration	Out of Memory Error	Out of Memory Error	Out of Memory Error	4 hrs
100 Instructions for 1 iteration	Out of Memory Error	Out of Memory Error	Out of Memory Error	1 hr

Table 5.2: Execution of CMA Model on different machines

After few steps, the agent got stuck in the last position shown in figure 5.2.

CMA Model

Same as Seq2Seq Model, We ran the CMA model on multiple systems with different numbers of instructions. Whether or not the model ran on the system with the time taken is provided in the table 5.2.

We took the weights generated by training the Nvidia TESLA P100 16GB GPU with a 25GB RAM system for 5K Instructions for 1 iteration and tested it for 100 instructions. The results generated are as follows:

- NDTW – 0.21
- SR – 0.03

- SPL – 0.03

The RGB and depth test output of one successful instruction is shown in figure 5.3. The RGB and depth test output of one failed instruction is shown in figure 5.4.

5.3 Discussion

The results gathered from the experiments are analyzed and the output of the models compared with each other.

Seq2Seq Model

From the results of the Seq2Seq Model, we can see that model was able to achieve an NDTW value of 0.273 for 100 episodes but in none of the episodes, the agent reached the goal position. Since the Seq2Seq model is the basic model with less complexity, it was not able to follow any instructions till the end. One of the reasons for the low success rate can be training the model for fewer episodes and fewer iterations.

CMA Model

The NDTW score of the CMA model is 0.21 which is lesser than the Seq2Seq Model but in CMA Model, the agent managed to reach the goal position in few episodes. For 100 test episodes, the success rate was 0.03. The CMA Model is more complex than the Seq2Seq Model and more efficient than that. Therefore, results achieved from the CMA model are better than Seq2Seq Model. The agent was able to reach the goal position where the length of instruction was less. But when the instruction length is increased, the agent got stuck halfway. This may be the result of less training period.

Figure 5.3: CMA Model output for Episode Number 108

Figure 5.4: CMA Model output for Episode Number 631

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This thesis aimed to develop and evaluate an agent to perform Vision-and-Language Task on the simulated environment for RxR Habitat Challenge by using Habitat-Sim and Habitat Lab. Two models built by using recurrent networks and attention layers, which were provided in the base code were implemented and tested. Results gathered from the experiments show that CMA Model performs better than the basic Seq2Seq Model. Although the results of the CMA Model are better than the Seq2Seq model, in general, the results are low which shows that new research and development is required in this area.

This thesis also implemented models on different machines with different configurations to find the ideal system configuration for the VLN task. It shows that to train the agent on full training data, a high-end system is required. Even with having 25GB RAM and 16GB GPU, the system was not able to train for full data. It can be said that the ideal system configuration for training the VLN task consists of more than 40GB RAM and multiple 16GB GPU. Training the model with fewer episodes and iterations due to system constraints produces poor results.

The instruction encoder from the Episodic Transformer converts the instruction into "long" form to perform embedding and positional encoding. The "long" datatype requires a larger space to store the data. The experiments from this thesis show that the E.T. model or instruction encoder of E.T. or the simple transformer model can

not be built on the system with 25GB RAM and 16GB GPU. It requires a higher configuration system for storing the data and training the model.

6.2 Further Work

There are several changes present that can be done to remove the issues with the current models and to improve the score of the VLN Task.
To train the model, a recollection trainer class is used which uses the data from ground truth while training the model. While testing, ground truth is not present. Due to this, the model may perform poorly in the testing environment. This issue is known as the Exposure Bias Issue. Some changes can be done to avoid the Exposure Bias issue. One possible solution for the exposure bias issue is to change the trainer class from recollection trainer to the DAgger(Database Aggregator).
Due to system constraints, this thesis did not manage to implement E.T. Model for RxR Challenge. According to the [3], the E.T. model works better than the LSTM. Implementing the E.T. model may generate better results than the current state-of-the-art CMA Model.
Currently, the instructions present in the RxR dataset are simple navigational instructions that require the agent to perform basic commands like move forward or backward or turning left and right. Creating complex instructions that require the agent to perform complicated commands can be considered as the next step for the VLN Task. The instructions containing moving objects from one place to another or opening and closing the door.

Bibliography

[1] Manolis Savva*, Abhishek Kadian*, Oleksandr Maksymets*, Y. Zhao, E. Wijmans, B. Jain, J. Straub, J. Liu, V. Koltun, J. Malik, D. Parikh, and D. Batra, "Habitat: A Platform for Embodied AI Research," in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019.

[2] A. Ku, P. Anderson, R. Patel, E. Ie, and J. Baldridge, "Room-across-room: Multilingual vision-and-language navigation with dense spatiotemporal grounding," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 4392–4412, 2020.

[3] A. Pashevich, C. Schmid, and C. Sun, "Episodic transformer for vision-and-language navigation," *arXiv preprint arXiv:2105.06453*, 2021.

[4] X. Wang, Q. Huang, A. Celikyilmaz, J. Gao, D. Shen, Y.-F. Wang, W. Y. Wang, and L. Zhang, "Reinforced cross-modal matching and self-supervised imitation learning for vision-language navigation," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 6629–6638, 2019.

[5] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, pp. 5998–6008, 2017.

[6] C. Campos, R. Elvira, J. J. Gómez, J. M. M. Montiel, and J. D. Tardós, "ORB-SLAM3: An accurate open-source library for visual, visual-inertial and multi-map SLAM," *arXiv preprint arXiv:2007.11898*, 2020.

[7] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet

Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[8] Google Research Team, "Rxr habitat challenge," 2021. [Online].

[9] J. Krantz, E. Wijmans, A. Majundar, D. Batra, and S. Lee, "Beyond the nav-graph: Vision and language navigation in continuous environments," in *European Conference on Computer Vision (ECCV)*, 2020.

[10] P. Anderson, Q. Wu, D. Teney, J. Bruce, M. Johnson, N. Sünderhauf, I. Reid, S. Gould, and A. Van Den Hengel, "Vision-and-language navigation: Interpreting visually-grounded navigation instructions in real environments," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3674–3683, 2018.

[11] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[12] J. Cheng, L. Dong, and M. Lapata, "Long short-term memory-networks for machine reading," *arXiv preprint arXiv:1601.06733*, 2016.

[13] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv preprint arXiv:1412.3555*, 2014.

[14] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[15] R. Mur-Artal, Montiel, J. M. M., Tardós, and J. D., "ORB-SLAM: a versatile and accurate monocular SLAM system," *IEEE Transactions on Robotics*, vol. 31, no. 5, pp. 1147–1163, 2015.

[16] M. Everingham, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes (voc) challenge," *International journal of computer vision*, vol. 88, no. 2, pp. 303–338, 2010.

[17] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," in *European conference on computer vision*, pp. 740–755, Springer, 2014.

[18] A. Kuznetsova, H. Rom, N. Alldrin, J. Uijlings, I. Krasin, J. Pont-Tuset, S. Kamali, S. Popov, M. Malloci, A. Kolesnikov, *et al.*, "The open images dataset v4," *International Journal of Computer Vision*, vol. 128, no. 7, pp. 1956–1981, 2020.

[19] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[20] A. Chang, A. Dai, T. Funkhouser, M. Halber, M. Niessner, M. Savva, S. Song, A. Zeng, and Y. Zhang, "Matterport3d: Learning from rgb-d data in indoor environments," *International Conference on 3D Vision (3DV)*, 2017.

[21] F. Xia, A. R. Zamir, Z.-Y. He, A. Sax, J. Malik, and S. Savarese, "Gibson env: real-world perception for embodied agents," in *Computer Vision and Pattern Recognition (CVPR), 2018 IEEE Conference on*, IEEE, 2018.

[22] A. Szot, A. Clegg, E. Undersander, E. Wijmans, Y. Zhao, J. Turner, N. Maestre, M. Mukadam, D. Chaplot, O. Maksymets, A. Gokaslan, V. Vondrus, S. Dharur, F. Meier, W. Galuba, A. Chang, Z. Kira, V. Koltun, J. Malik, M. Savva, and D. Batra, "Habitat 2.0: Training home assistants to rearrange their habitat," *arXiv preprint arXiv:2106.14405*, 2021.

[23] GoogleResearch, "Google scanned objects," August.

[24] A. Ku, P. Anderson, R. Patel, E. Ie, and J. Baldridge, "Room-Across-Room: Multilingual vision-and-language navigation with dense spatiotemporal grounding," in *Conference on Empirical Methods for Natural Language Processing (EMNLP)*, 2020.

[25] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pp. 1532–1543, 2014.

[26] M. Shridhar, J. Thomason, D. Gordon, Y. Bisk, W. Han, R. Mottaghi, L. Zettlemoyer, and D. Fox, “ALFRED: A Benchmark for Interpreting Grounded Instructions for Everyday Tasks,” in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

Appendix A

A.1 Steps to Run the code

The source code is available at: `https://github.com/Saching35/RxR-Habitat-Challenge.git`
It includes a Google Colab file containing steps for:

1. Setting up Conda environment
2. Installing Habitat-Sim
3. Installing Habitat-Lab
4. Cloning VLN-CE Repo for building and executing VLN agents
5. Downloading all required datasets
6. Training and Evaluation the agent

Following are the prerequisite to run the file on Google Colab:

- Subscription of Google Colab Pro or Higher
- 250GB+ free space on Google Drive

Following are the prerequisite to run the file on Local Machine:

- RAM: 25GB+
- GPU: 16GB+
- 250GB+ free space

Appendix B

B.1 Agent Configuration File

An example of an agent configuration file in YAML format is given below:

```
ENVIRONMENT:
  MAX_EPISODE_STEPS: 500 #Criteria for agent termination
SIMULATOR:
  AGENT_0:
    SENSORS: ['RGB_SENSOR'] #List of Sensors to be attached to the agent
  HABITAT_SIM_V0:
    GPU_DEVICE_ID: 0
  RGB_SENSOR: #Sensor details
    WIDTH: 256
    HEIGHT: 256
  DEPTH_SENSOR:
    WIDTH: 256
    HEIGHT: 256
TASK:
  TYPE: Nav-v0
  SUCCESS_DISTANCE: 0.2 #min distance between agent location and goal location

  SENSORS: ['POINTGOAL_WITH_GPS_COMPASS_SENSOR'] #Input Sensors
  POINTGOAL_WITH_GPS_COMPASS_SENSOR: #Sensor details
    GOAL_FORMAT: "POLAR"
```

```
  DIMENSIONALITY: 2
GOAL_SENSOR_UUID: pointgoal_with_gps_compass

MEASUREMENTS: ['DISTANCE_TO_GOAL', 'SUCCESS', 'SPL'] #Evaluation matrix
SUCCESS:
  SUCCESS_DISTANCE: 0.2 #Success Criteria
```