KU LEUVEN

FACULTY OF
ENGINEERING SCIENCE

Sequence-to-Sequence with dedicated Memory Networks for Vision-and-Language Navigation

Peter De Wachter

Thesis voorgedragen tot het behalen
van de graad van Master of Science
in de ingenieurswetenschappen:
computerwetenschappen, hoofdoptie
Mens-machine communicatie

Promotor:
Prof. dr. M.-F. Moens

Evaluatoren:
Prof. dr. A. Simeone
Dr. E. Sansone

Begeleider:
V. Milewski

Academiejaar 2022 – 2023

Preface

I would like to thank my promoter professor Moens and my mentor Victor Milewski for being extremely helpful during our many meetings.

Peter De Wachter

Contents

Abstract

Vision-language navigation (VLN) is the task where a visual agent carries out natural language instructions in 3D environments. Most approaches use recurrent states to remember past navigation steps and actions. Often, the state vector of these recurrent units is small and the remembered information is heavily compressed.

We propose in this thesis a recurrent VLN agent with a separate, dedicated, memory component that allows the agent to encode all past states without compression loss. The memory-integrated VLN agent architecture is a sequence-to-sequence (Seq2Seq) network, where the decoder interacts with a memory component. We propose three such architectures: The first two agents, the *Combined Seq2Mem* and *Parallel Seq2Mem* agent, implement a first in, first out (FIFO) memory network as its memory component. The third agent uses a differentiable neural computer (DNC) to store its history and is called the *Seq2DNC* agent.

We test the capabilities of the memory-integrated agents under two operation modes: First, the agents are trained and evaluated navigating, where the memory component is used to remember past steps of its current task. For each new instruction, the memory is cleared again. This mode is called *short-term memory.* In the second mode, called *long-term memory*, the agents navigate using a persistent memory, where memories carry over between instructions. The goal is for the agent to use its knowledge about the environment in the form of its memories when navigating to its advantage. We train the agents to operate in this mode by fine-tuning the models trained under the first mode with a persistent memory.

Our Seq2DNC agent outperforms the baseline Seq2Seq VLN agent on all metrics on the Room-to-Room task in short-term memory mode, showing the viability of VLN memory agents. By comparing our different memory network architectures and their performances, we provide insight into what mechanisms in the memory network result in better performance for VLN memory agents. We did not produce a memory agent with persistent memory that outperforms the baseline Seq2Seq model performance.

Samenvatting

Visie-taal navigatie (VTN) is de taak waar een visuele agent gegeven instructie in natuurlijke taal uitvoert in 3D omgevingen. De meeste aanpakken gebruiken wederkerende toestanden om verleden navigatiestappen en ondernomen acties te onthouden. Vaak is de toestandsvector van deze wederkerende elementen te klein en wordt de onthouden informatie hevig gecompresseerd.

We stellen in deze thesis een wederkerende VTN agent met een aparte, speciaal toegewijde, geheugencomponent dat de agent toestaat alle verleden toestanden te encoderen zonder compressie verlies. De geheugen VTN agent architectuur is een sequentie-naar-sequentie (Seq2Seq) netwerk, waarbij de decoder interageert met een geheugencomponent. We stellen drie zulke architecturen voor: De eerste twee agenten, de *Gecombineerde Seq2Mem* en de *Parallelle Seq2Mem* agent, implementeren een eerst in, eerst uit geheugennetwerk als geheugencomponent. De derde agent gebruikt een differentieerbare neurale computer (DNC) om zijn verleden op te slaan en wordt de *Seq2DNC* agent genoemd.

We testen de mogelijkheden van de geheugen agent onder twee operatie modi: Eerst worden de agenten getraind en geevalueerd op navigeren waarbij het geheugen component gebruikt om stappen uit het verleden van de huidige navigatie instructie te onthouden. Voor iedere nieuwe instructie, wordt het geheugen opnieuw leeggemaakt. Deze modus noemen we *korte termijn geheugen*. In de tweede modus, genaamd *lange termijn geheugen*, navigeren de agenten met een persistent geheugen zodat de herinneringen worden overgedragen naar de volgende instructie. Het doel hierbij is dat de agent kennis uit zijn omgeving in de vorm van deze persistente herinneringen tot zijn voordeel kan gebruiken tijdens het navigeren. De agenten worden getraind om te opereren in deze modus door de modellen uit het eerste deel te fine-tunen met het persisterende geheugen.

Onze Seq2DNC agent presteert beter dan de standaard Seq2Seq VTN agent op alle metrieken op de Room-to-Room taak in korte termijn geheugen modus, wat het potentieel aantoont van de VLN geheugen agenten. Door onze verschillende geheugennetwerk architecturen en hun prestaties met elkaar te vergelijken, bieden we inzicht in welke mechanismes belangrijk zijn in het geheugennetwerk van een VTN geheugen agent. We zijn er niet in geslaagd een geheugen agent met een persistent geheugen te ontwikkelen dat beter presteert dan het basis Seq2Seq model.

List of Figures and Tables

List of Figures

List of Tables

List of Abbreviations and Symbols

Abbreviations

VLN	Vision-Language Navigation
R2R	Room-to-Room
NLP	Natural Language Processing
RNN	Recurrent Neural Network
LSTM	Long Short-Term Memory
Seq2Seq	Sequence-to-Sequence
DNC	Differentiable Neural Computer
FIFO	First In First Out
CNN	Convolutional Neural Network
RL	Reinforcement Learing
IL	Imitation Learning

Chapter 1

Introduction

A long-time goal of artificial intelligence is to be able to command a robot using your voice. This robot would understand the spoken instruction and be able to complete the task using cameras for visual perception. This task of navigating an agent with visual perception through 3D environments using natural language instructions is called Vision-and-Language Navigation (VLN).

Recently, [1] created an environment in which an agent as described above can be developed. Using the photorealistic imagery from the Matterport3D dataset [2], they created the Matterport3D simulator and the Room-to-Room (R2R) dataset. R2R is a dataset of human language navigation instructions and associated paths inside the Matterport3D simulator. Combined, we can train a VLN agent to follow the R2R instructions inside the Matterport3D simulator.

The VLN agent has to overcome several challenges. First, contrary to tasks like image captioning, where an AI program has to predict a single output from a single input, the VLN agent continuously receives new observation images and has to align them with its current given instruction. Most proposed agents encode the current navigation state using Recurrent Neural Networks (RNN) in a fixed-size state vector. Such a vector might be too small to remember extended trajectories and, as a result, lose important information [3] [4]. The navigation history is important to understanding instructions and the environment. For this reason, this thesis proposes a VLN model with a long-term memory component.

The second challenge for the VLN agent is the ability to generalize to new, unseen environments [5]. The generalizability problem is the fact that the agent performs significantly worse in environments it has not seen during training, as opposed to those it has already seen. Researchers have attempted to tackle this generalizability problem in different manners [6] [7] [8].

In psychology studies, it has been shown that long-term memory plays an important role in human navigation. The hippocampus is the area of the brain which creates episodic memories, memories of specific moments in time. [9] shows that during navigation this area of the brain is responsible for remembering landmarks and paths and organizing the memories in a spatial map.

Furthermore, as humans, we make use of a vast knowledge base to solve problems [10]. If asked to locate a toilet, most humans would not spend much time looking in a bedroom to find it. On an even more abstract level, as humans we understand that the layout of, for example, a house, is not random: Bedrooms are most likely not connected to a kitchen. Or if the house has a hallway, there is most likely a bathroom attached to it. All this is knowledge built from experience we as humans can leverage when solving navigational problems.

Inspired by the cognitive mechanisms humans use during navigation and past research that shows the value of long-term memory components in VLN and robotics [11] [12] [3] [4], the aims of this thesis are twofold: To design a VLN architecture with a long-term memory component that can **(1)** remember its observation history during a navigation task and **(2)** use a persistent memory between navigation tasks to gain familiarity with its environment.

1.1 Research Objective

The objective of this thesis is to build a VLN-agent that makes use of a long-term memory component to aid in navigation. The agent will be evaluated under numerous conditions and analyzed to shed insight into its workings. The goal of this thesis can be formulated as the following set of research questions:

Can integrating a memory network into a sequence-to-sequence network improve performance on the VLN task?

To answer the first research question, we take the following approach: We design three different sequence-to-sequence VLN agents with an integrated memory network. The purpose of the memory network is to remember past steps the agent has taken during navigation. The memory is also not persistent between navigation tasks: for each new instruction, the memory is cleared. Next, we train and evaluate the models on the R2R navigation task. Finally, we compare the three different models with each other, and, with the sequence-to-sequence VLN agent without memory component [1]. From this comparison, we will see if memory-integrated models perform better than the baseline model, and if so, what mechanisms are causing the improvement.

Is the agent able to use a persistent memory to leverage past experiences on new instructions?

To answer the second question, we will adapt the models designed in the first part to work with a persistent memory. This will be done by fine-tuning them after they are trained for the first part. Next, we test if the models are able to leverage past memories to better perform new tasks as follows: The agent performs each instruction of an evaluation set twice. The first time the agent executes the instruction it begins from an empty memory. Whilst it executes the instruction, it builds up memories in its memory component. Then, it performs the instruction a second time, but now with the memory filled with the memories from the first run. We will compare the performance of the first run with that of the second run. From the results, we can learn whether the agents can use their experience with the instruction and environment to perform better.

Chapter 2

Review of related literature

This section gives an overview of the research literature related to the topic of this thesis. In the first section, the research on VLN is covered in two parts: First, the different existing environments in which VLN tasks are performed are described. Next, an overview of the Room-to-Room VLN task and environment are given, together with an overview of the advances researchers have made on this task. The second section gives an overview of memory networks which are a class of neural networks. In the last section, we describe the literature on how memory networks have been used in VLN and other related tasks.

2.1 Vision-Language Navigation

Vision-Language Navigation (VLN) is a research area that aims to develop an agent that can navigate and execute tasks spoken by humans. The VLN task can be seen as a language grounding problem where language is learned from performing instructions using visual sensory data. This section will first give an overview of different VLN environments. Next, it will describe notable works on the Room-to-Room VLN task. A full taxonomy of research on VLN is given by [13].

2.1.1 Navigation Environments

Training an agent to navigate is done within an environment. Many of these environments have been developed by researchers for many different navigation tasks. Before the introduction of the mainstream Matterport3D simulator [1], navigational agents were trained within synthetic 3D environments. Vizdoom [14] is such an environment based on the classic first-person shooter game Doom. It allows bots to learn behavior using the screen buffer. This environment thus can also be used to train a VLN agent, as is done by [15]. A more realistic, but still synthetic, environment is House3D [16]. It is a collection of over 45,000 realistic 3D human-designed scenes of houses. Similar to House3D are THOR [17] and CHALET [18], both 3D environments of realistic looking indoor scenes. However realistic these environment scenes may look, they still are man-made with a limited number of assets and textures, lacking the richness and diversity of real-life environments.

2.1.2 Room-to-Room

The work in this thesis is based on the Room-to-Room dataset and task [1]. R2R is a route-oriented navigation task, requiring an agent to navigate from a starting pose, following a natural language instruction, toward a goal location within the Matterport3D simulator. The natural language instruction always describes a single route toward a goal location. The instruction is given as an input sequence of word tokens. The agent must predict actions that alter the agent's pose. When the agent predicts the 'stop' action, the current pose of the agent is considered the final destination. The task is accompanied by the R2R dataset which contains instructions describing over 7,000 paths within the Matterport3D simulator. A differentiating factor between the training environment of R2R and previous tasks is that the visual input to the agent is the photorealistic imagery from the Matterport3D dataset [2], which is much richer than any limited set of 3D assets and textures.

Since the introduction of the R2R benchmark, many approaches have been taken by different researchers to improve the performance on this task. This subsection aims to give an overview of these works by categorizing them by their different approaches to the task.

Evaluation Metrics Different evaluation metrics exist to evaluate a navigational agent's performance. The original R2R task measured the agent's performance with the *success rate* metric: the rate at which the agent stops within 3 meters of the goal location. This metric, however, does not take into account how closely the agent follows the intended path, only if it reaches its goal. More sophisticated metrics have been researched to improve performance.

[19] proposes the *Coverage weighted by Length Score* (CLS) metric. Which measures how closely the followed trajectory follows the ground-truth path. *normalized Dynamic Time Warping* (nDTW) introduced by [20], is another metric such that softly penalizes deviations from the reference path. This metric is naturally sensitive to the order of the nodes composing each path, is suited for both continuous and graph-based evaluations, and can be efficiently calculated. Another way of scoring the trajectory of the agent is proposed by [21], where a discriminative model predicts how well the trajectory matches the instruction.

Instruction Processing Another way in which researchers have improved performance is by pre-processing input instruction. The goal here is to adapt the instruction in a manner that is easier to understand by the agent. One such manner of pre-processing is splitting up the instruction into sub-instructions. Because one instruction can contain many steps for the agent to complete, it might be easier to predict the path when these steps are made distinct from one another. The splitting of instructions is either done by a separate ML module [22] or by human annotators [23].

Other than pre-processing instructions, researchers have designed models that generate new instructions [24]. The model generates new natural language instructions from the data. Allowing the agent to train on much more data than is available.

Pre-training A difficulty in completing this task is that data is relatively scarce and acquiring new data is expensive. To maximize the utility of the available data, researchers pre-train the machine on a related task for which data is more easily available, and in the next step fine-tune the weights of the machine on the data of the actual task. *Prevalent* performs [6] employs a BERT-like approach on text-image-action triplets by performing action prediction and Masked Language Modelling on the instruction. Another implementation of this idea is *VLN-BERT* [7], where visual grounding is learned on image-text pairs from the web.

Transformer architectures The model proposed initially with the R2R task is a form of sequence-to-sequence model. Transformer architectures [25] is another type of sequence transduction model, relying solely on attention mechanisms. Researchers have proposed several transformer architectures for the VLN task: [26] [27] and [3].

2.2 Memory Networks

Memory networks are a class of neural networks proposed by [28] where inference components interact with a long-term memory component. The most common application for these networks is in question answering (QA) tasks, where the memory component acts as a knowledge base, and the output is a textual response. The end-to-end memory network [29] proposes a memory network that performs similarly to the original memory network by [28] and requires less supervision during training. A more generalized memory network proposed by [30] is the key-value memory network that performs even better on many QA benchmarks than the previous two networks.

The above-described memory networks have a rigid knowledge base: A series of inputs are stored in the memory component, then the memory is queried to answer questions. A different class of neural networks called *neural computers* interact with a memory component differently. On each input, a controller network interacts with a memory component, both writing to it and reading from it, and returns an output. The neural Turing machine proposed by [31] is such a neural computer. It is analogous to a Turing Machine but is fully differentiable. It is capable of learning simple algorithms such as copying, sorting, and associative recall from input and output examples. An extension of the neural Turing machine is the differentiable neural computer [32] which has more extensive memory access methods.

2.3 Memory in related tasks

2.3.1 Memory in VLN

More closely related to the objective of this thesis, several studies have researched what integrated memory components might provide to a VLN agent in the R2R task. First, [3] have designed the History Aware Multimodal Transformer (HAMT), which incorporates a long-horizon history into multimodal decision making. HAMT explicitly remembers all past observations during a navigational instruction and encodes them in their hierarchical vision transformer. This work is similar to the proposed goal of this thesis, but differs in the following way: The navigation history in HAMT [3] is reset between each instruction: The agent only remembers its current task. In this thesis, our goal is to have the navigation history, or memories, be persistent between navigation instructions. The goal here is for the agent to build a familiarity with the environment it is navigating.

Next, [11] and [12] are both concurrent works focused on training the agent to memorize the layout of the environment it is navigating. During navigation in an environment, the agent constructs a scene map. The scene map is a graph representation of the environment layout, where nodes in the graph are navigable positions and edges represent positions that can be traveled to from one another. This graph acts as a map allowing the agent to perform long-range planning.

Vision-Dialog Navigation is a closely related task to VLN. In this task, the navigational agent is allowed to ask the instructor questions to clarify how it should navigate. The addition of a cross-modal dialog and vision memory component has shown to significantly improve performance for this task [33].

2.3.2 Memory in Robotics

In autonomous robot navigation, neural network memory architectures proved significant in long-term planning and overcoming cul-de-sacs [34]. This was achieved by having a differentiable neural computer as part of the navigation architecture, an idea that also is explored further in this thesis.

2.3.3 Memory in NLP

In NLP, memory networks have been used in a wide variety of tasks such as question answering [28, 30, 29, 32], language modeling [35], and chatbots [36]. The memory networks for these NLP tasks serve as inspiration for the memory components in the VLN agent which is described further in this thesis.

Chapter 3

Background

In this chapter, we describe concepts and mechanisms that are used throughout this thesis. First, a collection of machine learning architectures and methods are described. All these concepts are commonly used in the field of NLP and computer vision and are used in the VLN memory agent architectures designed by us in Section 4.3. Secondly, the Seq2Seq VLN agent architecture proposed by [1] is described. This architecture is key since the implementations proposed by this thesis are based on it. Lastly, an overview of relevant memory networks is given.

3.1 Machine Learning Architectures and Methods

3.1.1 Long short-term memory

Recurrent neural networks (RNN) are a class of networks where the output from the previous time step is fed as (part of) the input in the following step. This type of network is used in problems where inputs and outputs are dependent sequences, like translating a sentence [37].

Standard RNN architectures have difficulties learning information over long time intervals via backpropagation. During backpropagation, the error signal grows exponentially with the weights of the network. This causes the signal to either rapidly vanish or explode [38]. Long short-term memory (LSTM) units partially solve this problem because the gradient is allowed to flow unchanged through the unit.

The LSTM unit is comprised of a cell state, an input gate, an output gate and a forget gate. The cell remembers values over arbitrary intervals of time and the three gates regulate the information flow in and out of the cell [39]. The equations for an

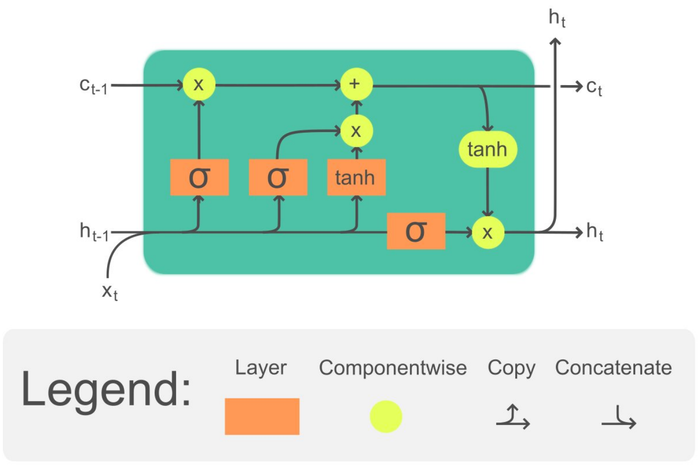


Figure 3.1: LSTM cell architecture. The inputs on the left side are the previous hidden state vector h_{t-1}, previous cell state c_{t-1}, and input x_t. The orange layers are the gates that determine the behavior of the cell. The leftmost gate is the forget gate: its effect is that the cell state is multiplied with a number between 0 and 1. The next two layers together are the input gate. They decide what of the input gets added to the cell state. The rightmost layer is the output gate: It decides what of the cell state gets put out. From [40].

LSTM unit are the following [40]:

$$\begin{aligned}
\boldsymbol{f}_t &= \sigma(\boldsymbol{W}_f \boldsymbol{x}_t + \boldsymbol{U}_f \boldsymbol{h}_{t-1} + \boldsymbol{b}_f) \\
\boldsymbol{i}_t &= \sigma(\boldsymbol{W}_i \boldsymbol{x}_t + \boldsymbol{U}_i \boldsymbol{h}_{t-1} + \boldsymbol{b}_i) \\
\boldsymbol{o}_t &= \sigma(\boldsymbol{W}_o \boldsymbol{x}_t + \boldsymbol{U}_o \boldsymbol{h}_{t-1} + \boldsymbol{b}_o) \\
\tilde{\boldsymbol{c}}_t &= \tanh(\boldsymbol{W}_c \boldsymbol{x}_t + \boldsymbol{U}_c \boldsymbol{h}_{t-1} + \boldsymbol{b}_c) \\
\boldsymbol{c}_t &= \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \tilde{\boldsymbol{c}}_t \\
\boldsymbol{h}_t &= \boldsymbol{o}_t \odot \tanh(\boldsymbol{c}_t)
\end{aligned}$$

Where the variables are the following: $\boldsymbol{x}_t \in \mathbb{R}^d$ is the input vector. $\boldsymbol{f}_t$, $\boldsymbol{i}_t$, and $\boldsymbol{o}_t$ are the forget, input, and output gate's activation vector respectively. $\boldsymbol{h}_t \in [-1,1]^h$ is the hidden state or output of LSTM unit. $\tilde{\boldsymbol{c}}_t \in [-1,1]^h$ is the cell input activation vector and $\boldsymbol{c}_t \in \mathbb{R}^h$ is the cell state vector. $\boldsymbol{W} \in \mathbb{R}^{h \times h}$ and $\boldsymbol{b} \in \mathbb{R}^h$ are learnable weight matrices and a bias vector. With d and h the dimension of the input vector and hidden units. Both $\boldsymbol{c}_c$ and $\boldsymbol{h}_0$ are zero vectors. The LSTM cell architecture is shown in figure 3.1.

3.1.2 Sequence to Sequence Model

The sequence-to-sequence (Seq2Seq) neural network proposed by [41] is a general end-to-end approach to sequence mapping problems. It is comprised of two (multilayered) LSTM. The first deep LSTM layer acts as the encoder. In the encoder, the input

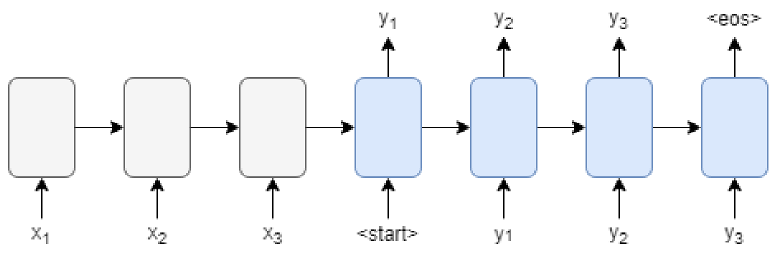


Figure 3.2: Graphical overview of the sequence-to-sequence architecture proposed by [41]. On the left side is the encoder. The grey cell is the encoder LSTM that encodes the input sequence. Its final hidden state is then passed on to the blue decoder LSTM on the right side. The initial input to the decoder LSTM is a special “start” token. The next input each step is an embedding of the previous output token. The decoder stops when it finally predicts a special “stop” or “end-of-sentence” token.

sequence is encoded into a hidden state vector of fixed dimensionality. The second LSTM layer is called the decoder. In this LSTM the hidden vector of the encoder is decoded into the target sequence. The initial state vector of the decoder LSTM is set the same as the final output of the encoder LSTM. At each step, the decoder LSTM receives as input the previously predicted token. Prediction is stopped once a special ’end-of-sentence’ token is predicted. The working of the sequence to sequence network with input $\boldsymbol{x}$ of length n is mathematically described as follows:

$$\begin{aligned}\boldsymbol{h}_t,\, \boldsymbol{c}_t &= LSTM_{enc}(\boldsymbol{x}_{t-1},\, \boldsymbol{h}_{t-1},\, \boldsymbol{c}_{t-1})\\ \boldsymbol{h}'_t,\, \boldsymbol{c}'_t &= LSTM_{dec}(Y_{t-1},\, \boldsymbol{h}'_{t-1},\, \boldsymbol{c}'_{t-1})\\ \boldsymbol{y}_t &= softmax(\boldsymbol{W}\boldsymbol{h}'_t)\end{aligned}$$

Here the $LSTM$ function has as the first argument its input, as the second argument its previous hidden state, and as the third argument its previous cell state. Y_t is the embedded previously predicted token from the output probability distribution $\boldsymbol{y}_t$. Also, $\boldsymbol{h}_0 = 0$, $\boldsymbol{h}'_0 = \boldsymbol{h}_n$. The output probability vector is computed by a successive linear matrix multiplication and softmax operation. Figure 3.2 shows a graphical overview of the sequence-to-sequence model.

3.1.3 Attention

Attention in neural networks is a technique that mimics cognitive attention. Attending an input to context data has the effect of highlighting aspects of the input which are important in the context of the data, and diminishing the unimportant aspects. If the context data is the input itself, the attention mechanism is called self-attention. Otherwise, if the context data is different from the input data, the mechanism is called cross-attention.

The Attention mechanism was first introduced by [42], in the context of sequence-to-sequence models for translation problems. It works as follows: As standard in a sequence-to-sequence model, the encoder RNN, from an input sequence $\mathbb{X}$ of length

n, computes a sequence of hidden states $(\boldsymbol{h}_1, \boldsymbol{h}_2, ..., \boldsymbol{h}_n)$. The decoder RNN, then, at each time step computes a hidden state vector $\boldsymbol{h}'_t$. The decoder hidden state vector is attended over the encoder state vectors as follows: At each time step t during decoding an alignment score between the decoder state $\boldsymbol{h}'_{t-1}$ and each encoder state vector $\boldsymbol{h}_j$ is computed.

$$e_{ij} = score(\boldsymbol{h}'_{i-1}, \boldsymbol{h}_j)$$

The alignment score can be any metric that resembles how well two inputs match. The *cosine similarity* or the *scaled dot product* are often good candidates [43] [25]. Next, the alignment scores are transformed to a probability or weighting by a *softmax* operation.

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{n} \exp(e_{ik})}$$

Finally, the context vector is computed as a sum of the encoder states, weighted by the similarity weight distribution. The context vector is thus a combination of the encoder states wherein parts that *align* well with the decoder state are highlighted.

$$\boldsymbol{c}_i = \sum_{j=1}^{n} \alpha_{ij} \boldsymbol{h}_j$$

By having the attention mechanism at the decoder, the encoder is relieved of having to encode all input information in the final hidden state vector. This comes from the fact that during decoding, the attention mechanism allows the decoder to look at each encoder state separately. This results in allowing the encoder to encode information in all its hidden states, making information flow easily over arbitrary time distances.

The attention mechanism can also be used outside of the context of sequence-to-sequence architectures. In general, the attention mechanism can be seen as a mapping of a *query*, to a set of *key-value* pairs. Suppose a set of input vectors $(\boldsymbol{x}_1, ..., \boldsymbol{x}_{L_1})$ is attended to a set of vectors $(\boldsymbol{y}_1, ..., \boldsymbol{y}_{L_2})$.

$$\boldsymbol{q}_i = \boldsymbol{x}_i \boldsymbol{W}^Q$$
$$\boldsymbol{k}_j = \boldsymbol{y}_j \boldsymbol{W}^K$$
$$\boldsymbol{v}_j = \boldsymbol{y}_j \boldsymbol{W}^V$$

Where $\boldsymbol{W}^Q \in \mathbb{R}^{d_{model} \times d_k}$, $\boldsymbol{W}^K \in \mathbb{R}^{d_{model} \times d_k}$ and $\boldsymbol{W}^V \in \mathbb{R}^{d_{model} \times d_v}$ are learned weight matrices. Same as before, an alignment score is calculated between the *queries* and each *key*, which is then used as a weighting for a weighted sum of the *values*. If, for the score, the *scaled dot product* is used, the whole attention operation can be efficiently computed with the following matrix multiplication:

$$Attention(\boldsymbol{Q}, \boldsymbol{L}, \boldsymbol{V}) = softmax\left(\frac{\boldsymbol{Q}\boldsymbol{K}^\top}{\sqrt{d_k}}\right)\boldsymbol{V}$$

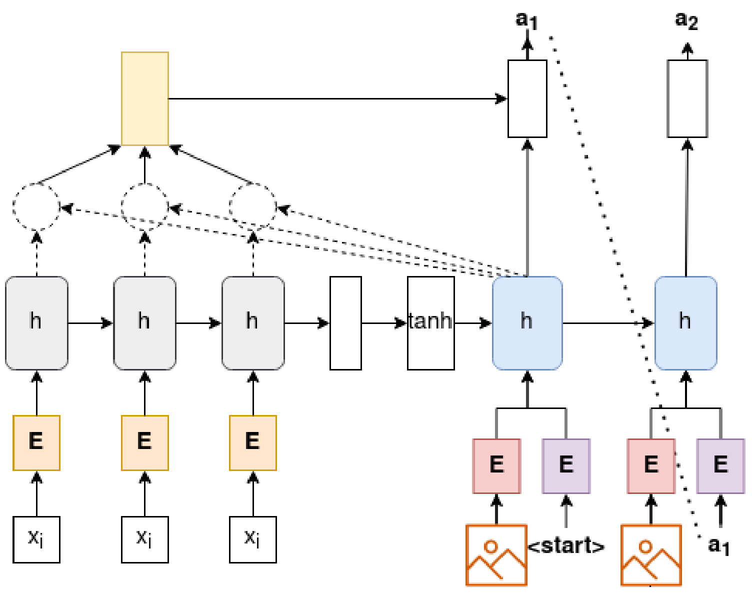


Figure 3.3: Graphical overview of the Seq2Seq VLN agent by [1]. On the left side is the encoder. The encoder LSTM receives as input the embedded natural language instruction. The final hidden encoder state vector is put through a linear layer and activation layer before being passed on to the decoder LSTM. The decoder LSTM receives at each time step the concatenated embedding of the current agent view and previously predicted action. The output of the LSTM is then attended over the hidden states before finally being transformed to a probability distribution of the next action.

3.2 Seq2Seq VLN architecture

In this thesis, we base our own implementation on the architecture of the VLN agent proposed by [1]. The architecture of this model is an adaptation of the sequence-to-sequence architecture explained in section 3.1.2. The following section describes the architecture of this Seq2Seq VLN agent. Figure 3.3 shows a graphical overview of the architecture.

Encoding The given input to the encoder is a natural language instruction $\mathbb{X} = (x_1, x_2, ..., x_l)$ with l the length of the instruction and x_i a single word token. During encoding of the input, each word token is sequentially passed through an embedding layer and an LSTM cell. As a result, a sequence of hidden state vectors, $\bar{\boldsymbol{h}} = [\boldsymbol{h}_1, \boldsymbol{h}_2, ..., \boldsymbol{h}_l]$, is built up that is relevant later during decoding. Before passing the final state vector onto the decoder, it is passed through a linear, and hyperbolic tangent activation layer. The final output vector of the encoder, and also the initial decoder hidden state vector, is thus:

$$\boldsymbol{h}'_0 = \tanh(\boldsymbol{A}\,\boldsymbol{h}_l)$$

Where $\boldsymbol{A}\ in\mathbb{R}^{d\times d}$ is a learned linear layer with d the size of the encoder and decoder LSTM.

Decoding From the agent's current pose at each time step, the Matterport3D simulator constructs an image view. A ResNet-152 CNN [44], pre-trained on ImageNet [45], constructs a feature vector of the current view. The decoder input at each time step is then the concatenation of the current view feature vector $\boldsymbol{v}_t$ with the embedding of the previous action token $\boldsymbol{a}_{t-1}$. Call this vector $\boldsymbol{i}_t = [\boldsymbol{v}_t : \boldsymbol{a}_{t-1}]$. The output of the decoder LSTM is then:

$$\boldsymbol{h}'_t,\, \boldsymbol{c}'_t = LSTM_{dec}(\boldsymbol{i}_t,\, \boldsymbol{h}'_{t-1},\, \boldsymbol{c}'_{t-1}$$

Attention and action prediction Before predicting the next action from the decoder LSTM hidden state vector, it is attended to the encoder hidden states to highlight the most important parts of the instruction. First, the general alignment function [46] is used to compute an instruction context:

$$\boldsymbol{c}_t = f(\bar{\boldsymbol{h}}, \boldsymbol{h}'_{t-1})$$

Next, the context is concatenated with the decoder hidden state and passed through both a linear and activation layer.

$$\tilde{\boldsymbol{h}}_t = \tanh(\boldsymbol{W}[\boldsymbol{c}_t : \boldsymbol{h}'_t])$$

The predictive distribution of the next action is then the following:

$$\boldsymbol{p}_t = softmax(\boldsymbol{L}\tilde{\boldsymbol{h}}_t)$$

Same as with a general sequence-to-sequence model, the decoding stops once the agent predicts a *stop* action.

3.3 Memory Networks

Memory Networks are a class of neural networks, introduced by [28], combining inference components and a long-term (dynamic) memory component to solve problems. The memory can be read from and written to, with the goal of using it for prediction.

3.3.1 End-to-end Memory Network

The end-to-end memory network [29] is such a memory network. It takes in a discrete set of inputs $(\boldsymbol{x}_1, \boldsymbol{x}_2, \dots, \boldsymbol{x}_n)$ and a query $\boldsymbol{q}$, and outputs an answer $\boldsymbol{a}$. From the set of inputs two memory representations are computed, both by an embedding matrix transformation. The *input memory* representation is computed by transforming each input $\boldsymbol{x}_i$ by embedding matrix $\boldsymbol{A}$, and the *output memory* representation is computed with embedding matrix $\boldsymbol{C}$. The query $\boldsymbol{q}$ subsequently also is embedded by an embedding matrix $\boldsymbol{B}$ with the same dimension as $\boldsymbol{A}$ and $\boldsymbol{C}$ to obtain internal state $\boldsymbol{u}$. The output is then computed by a weighted sum of the output memories, where the weighting is found by computing the dot product between the internal state and the input memory entries, followed by a *softmax* operation.

$$
\begin{aligned}
\boldsymbol{m}_i &= \boldsymbol{A}\boldsymbol{x}_i \\
\boldsymbol{c}_i &= \boldsymbol{C}\boldsymbol{x}_i \\
\boldsymbol{u} &= \boldsymbol{B}\boldsymbol{q} \\
p_i &= softmax(\boldsymbol{u}^\top \boldsymbol{m}_i) \\
\boldsymbol{o} &= \sum_{i=1}^{n} p_i \boldsymbol{c}_i
\end{aligned}
$$

Where $\boldsymbol{A}$ and $\boldsymbol{C} \in \mathbb{R}^{d \times d_x}$ and $\boldsymbol{B} \in \mathbb{R}^{d \times d_q}$. Here, d_x is the size of the input set vectors, d_q is the query vector size, and d is the size of the memory representations.

The final prediction is then made by computing the sum between $\boldsymbol{o}$ and $\boldsymbol{u}$ and passing this sum through a final weight matrix $\boldsymbol{W}$ (of size $V \times d$, with V the size of the output vocabulary) and performing the *softmax* operation.

$$\boldsymbol{a} = softmax(\boldsymbol{W}(\boldsymbol{o} + \boldsymbol{u}))$$

The above-described architecture can be extended into multiple layers. Each layer has its own embedding matrices $\boldsymbol{A}^k$, $\boldsymbol{B}^k$, and $\boldsymbol{C}^k$. At the end of each layer except the final one, instead of predicting the answer, a new internal state $\boldsymbol{u}^k$ is computed as the sum of the current layer's output and internal state. In the k'th layer this would look like the following:

$$\boldsymbol{u}^{k+1} = \boldsymbol{u}^k + \boldsymbol{o}^k$$

Different types of *weight tying* are used to reduce the number of learned parameters in the model. In one setting, each layer has identical memory input and output embedding weights: $\boldsymbol{A}^1 = \boldsymbol{A}^2 = ... = \boldsymbol{A}^k$ and $\boldsymbol{C}^1 = \boldsymbol{C}^2 = ... = \boldsymbol{C}^k$. In another, the output embedding is the same as the input embedding for the next layer: $\boldsymbol{A}^k = \boldsymbol{C}^{k-1}$.

3.3.2 Differentiable Neural Computer

Neural computers are memory-augmented neural networks that combine the capabilities of neural networks and programmable computers to learn algorithms from examples. The differentiable neural computer (DNC) [32] is a model consisting of a neural network that can read and write from a memory matrix. The neural network is called the *controller* and can be either a feed-forward network or (multilayered) RNN. When forwarding an input through the controller network it emits an *interface vector* that defines the interaction with the memory matrix at the current time-step. The interface vector contains all the parameters for the read and write operation on the memory.

The DNC uses differentiable attention mechanisms to calculate a distribution over the memory slots. The is a weighting and represents how much of each slot is used in the memory operation. A read weighting $\boldsymbol{w}^r$ returns a read vector $\boldsymbol{r}$ from the memory matrix $\boldsymbol{M}^{N\times L}$, with N the number of memory slots and L the size of those slots, as follows:

$$\boldsymbol{r} = \sum_{i=1}^{N} \boldsymbol{M}_{i,:}\, w_i^r$$

Similarly, the DNC uses a write weighting $\boldsymbol{w}^w$ to first erase old memories with an erase vector $\boldsymbol{e}$ and then write with an update vector $\boldsymbol{v}$:

$$M_{i,j} \leftarrow M_{i,j}(1 - w_i^w e_j + w_i^w v_j)$$

The DNC uses three different differentiable addressing mechanisms to calculate the read and write weightings. The first is *content-based addressing*, in which a key vector that is emitted by the controller is compared to the memory slots by some similarity score. These scores then determine a weighting between the slots. A second mechanism is *temporal linkage*. A temporal link matrix $\boldsymbol{L}^{N\times N}$ allows the controller to track the sequences in which the memory was written to. $L_{i,j}$ is close to 1 if i was the next location written to after j, and is close to 0 otherwise. The final addressing method is a *usage vector* that tracks rarely used locations in memory. This allows the memory to overwrite memory slots that are no longer useful. Figure 3.4 shows an overview of the DNC architecture.

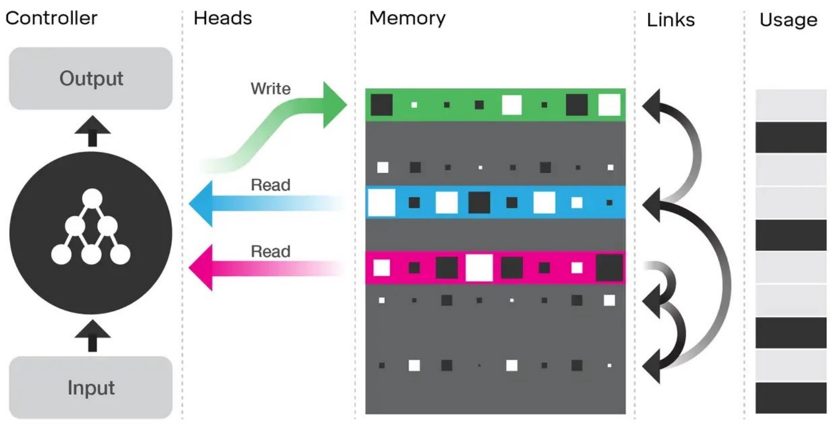


Figure 3.4: Graphical overview of the DNC architecture from [32]. The components from left to right can be described as follows: The controller is a recurrent neural network that receives an input and produces an output. It also predicts the parameters which are used by the write head (green) and multiple read heads (blue and pink). The write head defines a write and an erase vector which are used to update the memory matrix right of the heads. The read heads define a read operation each with different parameters, to read from the memory matrix. The read operation returns a read vector which is returned to the controller. Right of the memory matrix is shown a representation of the *Link matrix* and the *Usage vector*. The link matrix keeps track of which memory entries are written in sequence to each other. The usage vector is a vector that keeps track of which memory entries are being read from and written to, so irrelevant memories can be erased. The final output by the controller is a combination of the read vectors and output vector predicted by the controller itself.

Chapter 4

Method

In this chapter we describe in the first section the navigation environment wherein we perform the VLN task. Next the different evaluation metrics used to measure the performance on the task are explained. Lastly, we describe in full detail three different VLN agent architectures with long-term memory components. These agents will be trained and evaluated in the next chapter.

4.1 Environment

The dataset used for this task is the R2R dataset [1] for visually-grounded natural language navigation within the Matterport3D simulator. The Matterport3D simulator enables the development of AI agents that interact with real 3D environments using visual information (RGB-D images) from the Matterport3D dataset [2]. The primary intention lies within research in the intersection of the fields of computer vision, NLP, and robotics. The images for the simulator is sampled from the Matterport3D dataset containing comprehensive panoramic imagery of 90 indoor buildings. In the Matterport3D simulator, the 3D scans of buildings are discretized into views which are computed from the navigating agent's current position, angle, and camera elevation. The discretized space is accompanied by a navigation graph that defines the connectivity between views. This graph is shown in the left figure in Figure 4.1.

The R2R dataset contains three human language instructions for a total of 7,189 sampled trajectories through the Matterport3D buildings, totaling 21,567 instructions. To perform the VLN task, every image observation is encoded by a ResNet-152 [44] CNN pre-trained on ImageNet [45]. The agent is thus required to choose the next viewpoint given the instruction, the current view, and the connectivity between the views. Figure 4.1 shows such a connectivity graph in an environment.

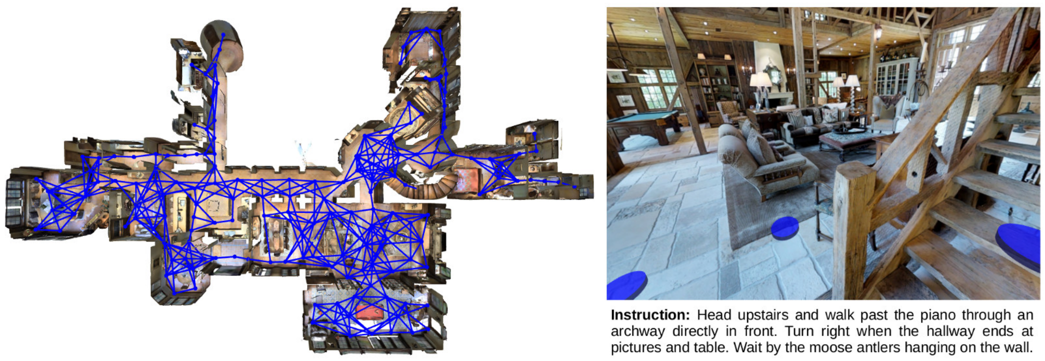

Instruction: Head upstairs and walk past the piano through an archway directly in front. Turn right when the hallway ends at pictures and table. Wait by the moose antlers hanging on the wall.

Figure 4.1: The R2R navigation task. The agent is required to follow a natural language through a 3D environment. The blue discs on the right image represent positions the agent can move to. The left image shows the connectivity graph, defining which locations are accessible from which point.

4.2 Evaluation Protocol

Evaluating performance on the R2R task is done in the same manner as in [1]. To evaluate the navigational agents the **navigation error** metric is used. Navigation error is defined as the shortest graph distance between the agent's final node and the goal node. Furthermore, the navigation is considered successful if the navigation error is less than 3 meters. This distance is significantly below the average starting position navigation error of 5 meters. The traversed path is not taken into consideration.

An important aspect of this task is that the agent must decide to stop at the right place. There are related vision-only tasks where the agent is not required to decide to stop. To separate the problems of navigation and recognizing the goal location a second metric, **oracle success**, is introduced. Under the oracle rule, the navigation error of the agent is the shortest graph distance between any of the agent's visited nodes and the goal node.

The agents are evaluated under two different circumstances. In the first setting, the agent has to complete instructions from the validation set, which are performed within environments the agent has previously seen during training. In the second setting, the agent performs navigation tasks in new, unseen environments. These settings are called **seen** and **unseen** respectively.

4.3 Memory-integrated VLN architectures

In this section, we describe three memory-integrated VLN architectures. Of the three architectures, the first two are variations of each other, they implement the exact same memory network, but encode its output in a different manner. The third architecture uses the DNC described in Section 3.3.2 as its memory component. An overview of the agent architectures is given by Table 4.1.

The general structure of the memory-integrated VLN architectures is as follows: Just as the original Seq2Seq VLN agent described in 3.3, the instruction is first encoded by an encoder, which passes the encoded instruction on to the decoder. The decoder receives at each time step as input an embedding of the agent's current view and an embedding of the previously predicted action. Different from the originally proposed Seq2Seq VLN architecture, the decoder interacts with a neural memory component at each time step, both reading from and writing to it. Next, both the decoder and memory outputs are encoded further to a final probability distribution. The memory component at the decoder allows the agent to make decisions having full knowledge of its past steps, without having to compress the memories.

Table 4.1: Overview of the three memory-integrated VLN agents described in Section 4.3. Both the *Combined Seq2Mem* and *Parallel Seq2Mem* use the FIFO memory network described in section 4.3.1. The Combined Seq2Mem agent's implementation combines the memory output immediately with the current navigation state and encodes these further together. The Parallel Seq2Mem agent, on the contrary, processes the current state and the memory output separately. The final agent, Seq2DNC, uses a DNC as the memory component, which produces a single, combined, output.

Agent	Memory Network	Implementation
Combined Seq2Mem	FIFO	Combined immediately
Parallel Seq2Mem	FIFO	Encoded separately
Seq2DNC	DNC	-

4.3.1 FIFO memory agent

First, in this section, we describe the memory network used by the first two VLN architectures by mathematically defining its *read* and *write* operations. It's design is mainly adapted from the *end-to-end memory network* [29] and is shown in Figure 4.2. Next, the implementation of the memory network into the complete VLN architecture is described.

Memory Network

Write Writing to the memory is done in a first in, first out (FIFO) manner. Meaning that every new input will be appended to the end of the memory until it is full. When there is no more free space, the oldest entry in the memory is removed to make space for the new input.

Read Reading is done similarly to the end-to-end memory network proposed by [29], described in section 3.3.1. The read operation has two arguments: a query vector $\boldsymbol{q}$ and a memory matrix $\boldsymbol{M}$. First, two representations of the memory are computed: an *input memory* representation $\boldsymbol{K}$ and an *output memory* representation $\boldsymbol{V}$. These embeddings are computed with learned linear weight matrices $\boldsymbol{A}$ and $\boldsymbol{C}$. Furthermore, since the read operation contains no recurrence or convolution, some

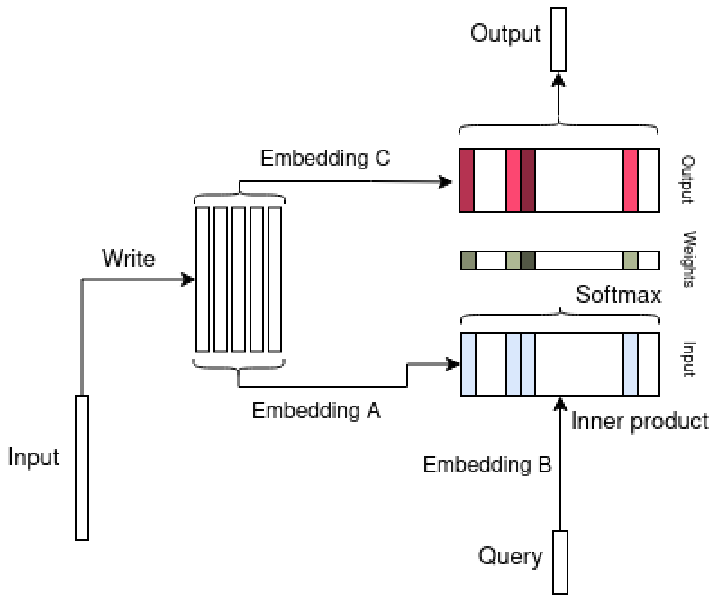


Figure 4.2: Close-up view of the FIFO memory network architecture. Left, the actual memory matrix is shown. Inputs are written to it in a first in, first out manner. On its right, the two memory representations are shown. The memory is read by computing the inner product between an embedded given query and the *input memory* representation of the memory entries. Next, these inner products are transformed into weights using the softmax operation. Finally, the output is the weighted sum of the *output memory* representation entries.

information about the positioning of the memories must be encoded. This way, the agent knows the order of the memories. The position encoding chosen here is the encoding proposed by [25]. Sine and cosine functions of different frequencies are added to the memory representations dependent on the position it has:

$$\begin{aligned} P_{pos,\,2i} &= \sin(pos/10000^{\frac{2i}{d}}) \\ P_{pos,\,2i+1} &= \cos(pos/10000^{\frac{2i}{d}}) \\ \boldsymbol{K} &= \boldsymbol{AM} + \boldsymbol{P} \\ \boldsymbol{V} &= \boldsymbol{CM} + \boldsymbol{P} \end{aligned}$$

Where pos is the position, i is the dimension and d is the size of the memory representations. Both weight matrices and the positional encoding matrix are $\in \mathbb{R}^{d \times d_x}$, where d_x is the size of the memories and d, again, is the size of the embedded memory representations. The query $\boldsymbol{q}$ is embedded with matrix $\boldsymbol{B} \in \mathbb{R}^{d \times d_q}$, with d_q the query vector size:

$$\boldsymbol{u} = \boldsymbol{Bq}$$

It is then compared with each row of the *input memory* and a similarity score is calculated between each. Here, the metric used to compute the similarity score is the *inner product*. These scores are then transformed into a weight vector using the softmax operation.

$$\boldsymbol{w} = softmax(\boldsymbol{u}\boldsymbol{K}^\top)$$

The weighting is then used to compute the final output $\boldsymbol{o}$, which is the weighted sum of the *output memory* entries.

$$\boldsymbol{o} = \sum_i w_i \boldsymbol{V}_{i,:}$$

Implementations

The above-described memory network is implemented in two different manners in a sequence-to-sequence VLN agent architecture. We call the two implementations the *Combined Seq2Mem* and the *Parallel Seq2Mem* network. They are shown in Figure 4.3. In the Combined Seq2Mem implementation, the decoder output and memory output are immediately combined and further processed together. The second implementation does the opposite: the two outputs are processed separately and the output is predicted from the two separate vectors. We will now describe the two implementations in more detail, starting with the former.

Combined Seq2Mem The architecture is adapted from the Seq2Seq architecture from [1], described in Section 3.3. The encoder encodes the natural language instruction identically to the original architecture. Also identical to the original architecture, is the input to the decoder LSTM: At each time step, the input for the decoder LSTM, $LSTM_{dec}$, is the concatenation of the embedded current view representation $\boldsymbol{v}_t$ and the embedded previously predicted action $\boldsymbol{a}_{t-1}$. The output of the decoder LSTM is the hidden vector $\boldsymbol{h}'_t$.

$$\boldsymbol{i}_t = [\boldsymbol{v}_t : \boldsymbol{a}_{t-1}] \tag{4.1}$$

$$\boldsymbol{h}'_t,\, \boldsymbol{c}'_t = LSTM_{dec}(\boldsymbol{i}_t,\, \boldsymbol{h}'_{t-1},\, \boldsymbol{c}'_{t-1}) \tag{4.2}$$

Here, $\boldsymbol{c}'_t$ is the cell state vector of the decoder LSTM. The decoder LSTM output vector is then used as a query to read the memory to produce output

$$\boldsymbol{o}_t = read(\boldsymbol{h}'_t,\, \boldsymbol{M}_t) \tag{4.3}$$

With $\boldsymbol{M}_t$ the memory matrix at the current time step.

Next, the LSTM and memory outputs are combined by summing them together. This sum is then attended over the encoder hidden states. The attention mechanism used is the same as the original VLN agent [1] described in Section 3.3. From the resulting attended vector, a probability distribution is computed over the possible outputs by the succession of a linear layer and softmax operation. Finally, the input of this time step is written to the memory so it can be used in the following steps.

$$\begin{aligned}
\boldsymbol{u}_t &= \boldsymbol{o}_t + \boldsymbol{h}'_t \\
\tilde{\boldsymbol{h}}_t &= Attn(\boldsymbol{u}_t,\, \bar{\boldsymbol{h}}) \\
\boldsymbol{p}_t &= softmax(\boldsymbol{W}\,\tilde{\boldsymbol{h}}_t) \\
\boldsymbol{M}_{t+1} &= write(\boldsymbol{i}_t,\, \boldsymbol{M}_t)
\end{aligned}$$

Parallel Seq2Mem In the second implementation, the decoder LSTM and memory outputs, $\boldsymbol{h}'_t$ and $\boldsymbol{o}_t$, are computed identically to the first with equations 4.1, 4.2, and 4.3. Different from before, these outputs are attended separately over the encoder context. This way, the agent can highlight separately what information is important from the current view, and what is important from the past views. Next, the attended vectors are concatenated and transformed into a probability distribution over the possible outputs. Finally, the current input is written to the memory.

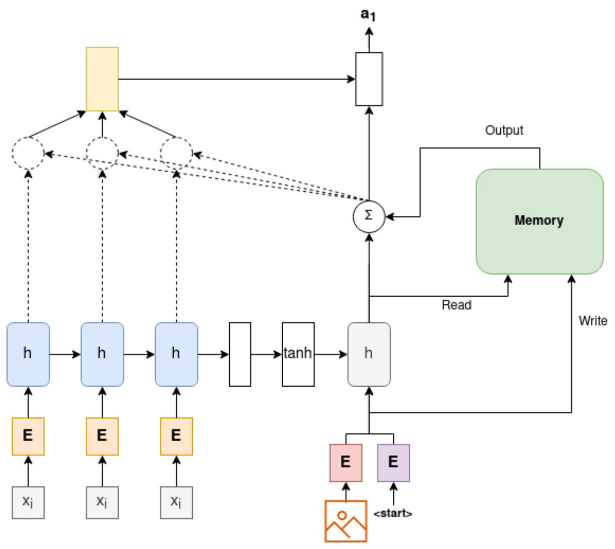
$$\begin{aligned}
\tilde{\boldsymbol{h}}_t &= Attn(\boldsymbol{h}'_t,\, \bar{\boldsymbol{h}}) \\
\tilde{\boldsymbol{o}}_t &= Attn(\boldsymbol{o}_t,\, \bar{\boldsymbol{h}}) \\
\boldsymbol{p}_t &= softmax(\boldsymbol{W}[\tilde{\boldsymbol{h}}_t : \tilde{\boldsymbol{o}}_t]) \\
\boldsymbol{M}_{t+1} &= write(\boldsymbol{i}_t,\, \boldsymbol{M}_t)
\end{aligned}$$

4.3.2 DNC decoder agent

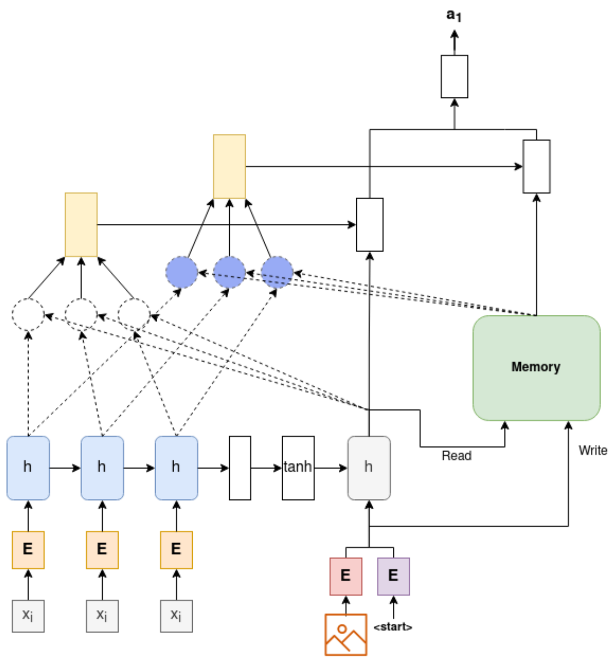
The DNC is a neural computer that can learn algorithms from data using a controller and memory matrix [32]. The memory interactions of the DNC are described in 3.3.2. Most importantly of these memory interactions, the DNC has two capabilities that the FIFO memory network described in Section 4.3.1 lacks: First, the DNC is capable of selectively writing to and erasing from the memory matrix, taking into account which memories are used more than others. Second, the DNC explicitly takes into account which memories are written in succession of each other when writing and reading the memory.

The agent described in this section is called the *Seq2DNC* agent and uses the DNC as memory component during navigation. The DNC controller can be any (recurrent) neural network. In this implementation, the role of the controller is taken on by the decoder LSTM. The full implementation is described further below.

Seq2DNC The encoder encodes the instruction and passes the result on to the decoder. The decoder is a DNC, consisting of an LSTM controller and memory matrix. At the start of decoding, the decoder LSTM's state is initialized to the final state passed on by the encoder. The input to the decoder at each time step is the concatenation of an embedding of the current view and the previously predicted action, denoted by equation 4.1. From this input, the controller interacts with the memory matrix by writing to and reading from it in the manner described in section 3.3.2. The output of the DNC, $\boldsymbol{y}_t$, is a combination of a vector predicted by the DNC controller, $\boldsymbol{u}_t$, and the read vectors from the memory interaction, $\boldsymbol{r}_t = [\boldsymbol{r}_t^1 : \boldsymbol{r}_t^2 : \ldots : \boldsymbol{r}_t^R]$. The output combination is computed by the DNC internally as follows: $\boldsymbol{y}_t = \boldsymbol{u}_t + \boldsymbol{W}\boldsymbol{r}_t$. The next action is predicted similarly to the basic Seq2Seq VLN agent, by attending the output vector over the encoder's hidden states and transforming the resulting vector into a probability distribution. The entire



(a) The Combined Seq2Mem VLN agent architecture.



(b) The Parallel Seq2Mem VLN agent architecture

Figure 4.3: Overview of the two Seq2Mem VLN agent architectures. **(a)** shows the *Combined Seq2Mem* agent, **(b)** shows the *Parallel Seq2Mem* agent. For both agents, the encoder is shown on the left side. The encoder LSTM encodes a given natural language instruction and passes its result on to the decoder LSTM, which sets it as its initial state. This decoder LSTM receives at each time step as input the embedded current view and previously predicted action. The output of the LSTM is used as a query to read the memory. In **(a)** the memory output is added to the LSTM output, it then is attended over the encoder states before being transformed to a probability distribution over the output classes. In **(b)** the memory and LSTM outputs are attended separately over the encoder states. The attended vectors are then concatenated before being transformed to a probability distribution.

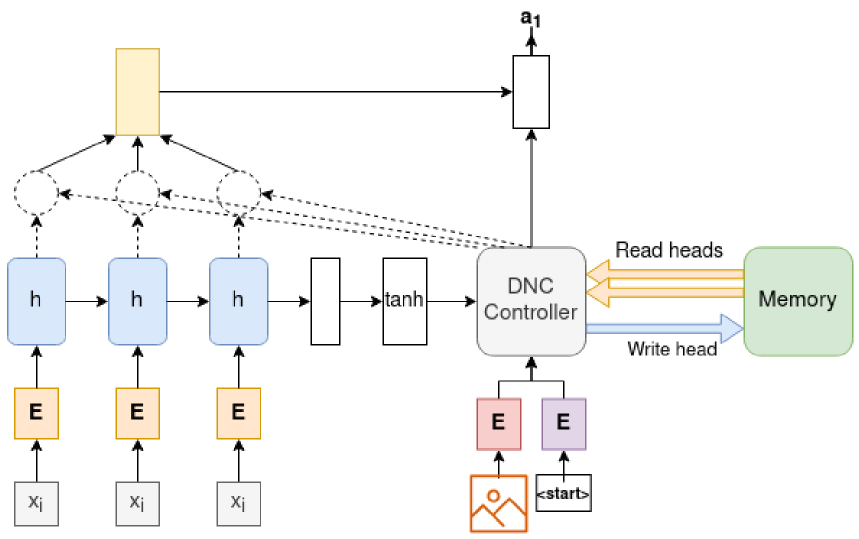


Figure 4.4: Overview of the Seq2DNC VLN agent architecture. On the left side is the encoder which consists of a single LSTM cell that encodes the given input instruction. On the right side is the decoder, which is a differentiable neural computer. Its initial hidden state is set to the final hidden state passed on from the encoder. At each time step, it receives as input the embedded view image and the previous predicted action. It interacts with the memory matrix through a *write head* and a number of *read heads*. Its output is then attended over the encoder's hidden states before finally being transformed into a probability distribution over the possible actions.

decoding is formally defined as follows:

$$\begin{aligned} \boldsymbol{y}_t, \boldsymbol{h}'_t, \boldsymbol{c}'_t, \boldsymbol{L}_t, \boldsymbol{u}_t, \boldsymbol{M}_t &= DNC(\boldsymbol{i}_t, \boldsymbol{h}'_{t-1}, \boldsymbol{c}'_{t-1}, \boldsymbol{L}_{t-1}, \boldsymbol{u}_{t-1}, \boldsymbol{M}_{t-1}) \\ \tilde{\boldsymbol{y}}_t &= Attn(\boldsymbol{y}_t, \bar{\boldsymbol{h}}_{enc}) \\ \boldsymbol{p}_t &= softmax(\boldsymbol{W}\,\tilde{\boldsymbol{y}}_t) \end{aligned}$$

Here, $\boldsymbol{h}'_t$ and $\boldsymbol{c}'_t$ are the DNC controller LSTM current hidden and cell states respectively. $\boldsymbol{L}_t$ is the DNC Link matrix, $\boldsymbol{u}_t$ is the DNC usage vector, and $\boldsymbol{M}_t$ is the DNC memory matrix. How these values interact with the controller and memory matrix is described in Section 3.3.2, and are defined in full detail in [32]. An overview of the architecture is given in Figure 4.4.

There are several parameters to choose for the DNC:

- *number of memory slots*: The size of the memory matrix.
- *memory slot size*: The DNC stores memories in an embedded form. Thus, the size of this embedding is a parameter.
- *hidden size*: The size of the controller hidden state vector.

- *number of layers*: The DNC controller can be a multilayered RNN. The number of layers.
- *number of read heads*: The number of read heads that read from the memory and each return a read vector every memory interaction.

Chapter 5

Experiments

In this chapter, the agents described in Section 4.3 are trained to navigate under two different modi operandi and evaluated. Under the first mode of operating the agent learns to navigate using the memory component to remember its past steps during single a navigation instruction. The memory is only used during a single task and is refreshed for each new task. This mode will be called *short-term memory.* The second mode is *long-term memory*: The agent's memory is persistent between instructions. The agent will use past experiences to better navigate future instructions.

5.1 Short-term memory

Under *short-term memory* the agent has a term memory component that is not persistent. During navigation, the agent stores its observations and actions in the memory component and clears it for every new instruction.

5.1.1 Training

In each model, the encoder remains the same as the Seq2Seq implementation from [1]. For this reason, we load the pre-trained weights from the Seq2Seq model for the encoder and freeze them while training the models. The original paper [1] trained their proposed Seq2Seq model under two regimes: *teacher forcing* [47] and *student forcing*, where during training either (1) at each step either always the ground-truth action is always used, or (2) the action is sampled from the predicted probability distribution. Since results show that the *student forcing* method proves to be more effective [1], this is the regime under which the models are trained.

Implementation Details Text pre-processing is kept identical to [1]: All sentences are converted to lowercase, tokenized on white space, and words that occur less than five times are filtered out. For all models, the parameter values are kept the same as [1] where possible: The number of hidden units in every LSTM is 512, the size of the input word embedding is 256 and the size of the input action embedding is 32. A dropout probability of 0.5 is used on embeddings, CNN features, and within the attention model. For the view images, we use the pre-computed feature vectors given by [1]. We set the number of memory slots for all memory networks to 20, which is higher than the maximum number of steps it takes to complete any instruction.

The Seq2DNC VLN agent, described in Section 4.3.2, has a set of parameters to configure. The number of memory slots is set to 20. The controller hidden size and number of layers are set to be identical to the LSTM implementation of [1]: The LSTM has a hidden size of 512 and consists of a single layer. The two remaining parameters are the memory slot size and the number of read heads. These are decided using a grid search, training the model for 10 000 iterations with different parameter configurations. The results of the grid search are shown in Table 5.1. The results show that the best performing configuration is a single read head with a memory slot size of 1 500.

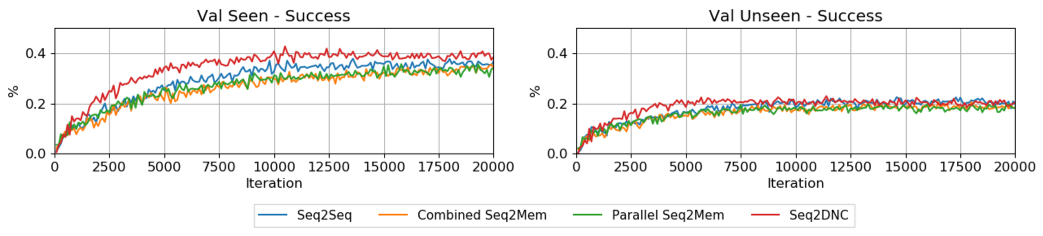
Table 5.1: Results of a grid search for a variable number of read heads (left) and memory slot size (top) for the Seq2DNC VLN agent. The score shown is the average success rate between the seen and unseen validation sets.

	Memory slot size		
read heads	500	1000	1500
1	0.2956	0.2924	**0.3068**
2	0.3004	0.2926	0.2967
3	0.3026	0.2973	0.3027

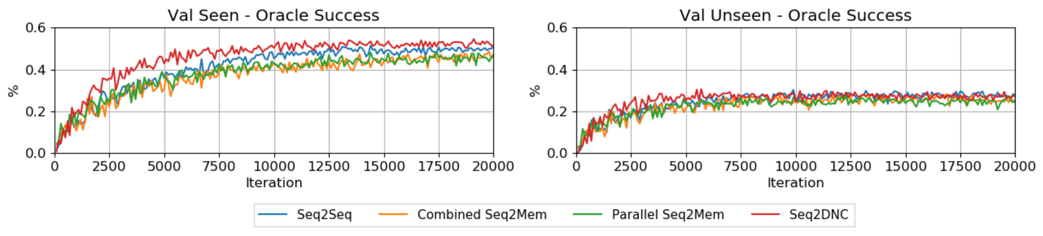
We train the models in PyTorch using the Adam optimizer with weight decay, using a batch size of 100. All training is done on a laptop with an NVIDIA GeForce RTX 2060 Mobile graphics card. Training times were between 120 and 300 minutes for 20 000 iterations. The exact times are shown in Table 5.2. The best model is selected according to the best average success rate (not oracle success rate) between the seen and unseen validation sets.

Table 5.2: Training times in short-term memory mode for the Seq2Seq model [1] and the memory-integrated models for 20 000 iterations. The times range between 2 hours for the Seq2Seq model [1] and 6 hours for the Seq2DNC model.

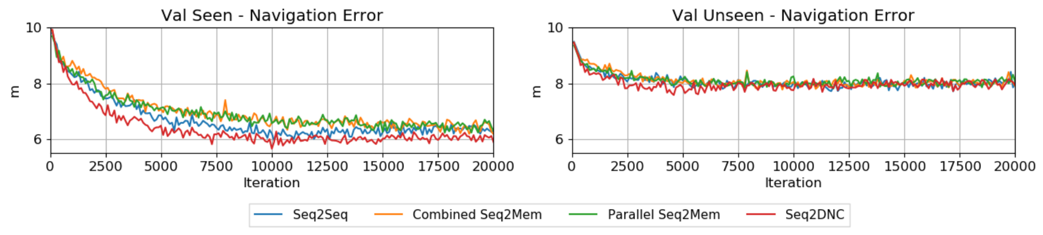
Agent	Training Time [minutes]
Seq2Seq [1]	123
Combined Seq2Mem	152
Parallel Seq2Mem	160
Seq2DNC	297



(a) Success rate



(b) Oracle success error



(c) Navigation error

Figure 5.1: Success rate, oracle success rate, and navigation error of the Combined and Parallel Seq2Mem, the Seq2DNC and Seq2Seq VLN agents on both the seen and unseen validation sets.

5.1.2 Evaluation

Figure 5.1 shows the training results of the models described in Section 4.3 under *short-term memory* mode. Their performances at the selected iterations are shown in Table 5.3. The evaluation metrics are defined in Section 4.2.

The results show that the Combined and Parallel Seq2Mem agents perform slightly worse than the baseline Seq2Seq model in both the seen and unseen validation sets. Their regular success rates on both validation sets are nearly identical: The Combined Seq2Mem agent shows a 5.4% and 9.4% relative decrease on the seen and unseen validation sets respectively. For the Parallel Seq2Mem agent these values are 6.2% and 9.4%. The oracle success rates on both sets are higher for the Combined Seq2Mem model than for the Parallel Seq2Mem model.

The Seq2DNC model performs better than the baseline in all categories. The

Table 5.3: R2R Navigation results of the models using evaluation metrics defined in Section 4.2 under the short-term memory operation mode. The Seq2Seq model, constructed by [1], is the baseline model we seek to improve. Both the combined and parallel Seq2Mem models perform slightly worse than the baseline, while the Seq2DNC model shows an improvement in every category.

	Validation Seen		Validation Unseen	
Agent	Success (%)	Oracle Success (%)	Success (%)	Oracle Success (%)
Seq2Seq [1]	37.7	49.2	21.2	27.9
Combined Seq2Mem	35.0	48.6	19.2	27.1
Parallel Seq2Mem	35.3	47.7	19.2	25.2
Seq2DNC	**41.3**	**53.6**	**22.0**	**28.7**

regular success rates show a 9.5% and 3.8% relative improvement on the seen and unseen validation sets respectively. For the oracle success rate, these values are 8.9% and 2.9%.

5.2 Long-term memory

The agent under *long-term memory* operation mode has a persistent memory. It remembers observations and actions during navigation and uses those when asked to execute new instructions. In this section, we describe how the models are trained to operate in this mode, and evaluate them. The goal of this experiment is to learn if the agents can perform better on a task if they have access to memories of a previous attempt on the same task.

5.2.1 Training Regime

Training the agent to navigate under long-term memory operation starts from a model trained, in the previous section, under short-term memory operation. The model is then fine-tuned for the long-term memory mode as follows: During training, the agent no longer performs each instruction starting with an empty memory. Instead, each training instruction is performed twice: The first time it is executed, the agent starts with an empty memory. The second time, the agent begins with the memory the agent accumulated whilst running the instruction the first time. The training loss is calculated only for the second time navigating. Table 5.4 shows the time needed to fine-tune the models for 20 000 iterations. The best model is selected according to the best average success rate (not oracle success rate) between the seen and unseen validation sets on the second attempt.

Table 5.4: Training times in long-term memory mode of the memory-integrated models for 20 000 iterations. Training times are longer in long-term memory mode than in short-term memory mode since each instruction is performed twice.

Agent	Training Time [minutes]
Combined Seq2Mem	231
Parallel Seq2Mem	254
Seq2DNC	568

Table 5.5: R2R Navigation results of the models using evaluation metrics defined in Section 4.2 under the long-term memory operation mode, before and after fine-tuning. The models execute each instruction twice consecutively with the memory persisting between runs. In run 1 the agents start the instruction with an empty memory. In run 2, the agent performs the instruction again, but with the memory persisting after the first run.

		Validation Seen		Validation Unseen	
Agent	runs	Success (%)	Oracle Success (%)	Success (%)	Oracle Success (%)
Not Fine-tuned:					
C. Seq2Mem	1	35.0	48.6	19.2	27.1
	2	36.0	49.7	19.8	28.0
P. Seq2Mem	1	35.3	47.7	19.0	25.0
	2	36.5	48.5	**22.7**	**28.8**
Seq2DNC	1	**41.3**	**53.6**	22.0	28.7
	2	23.0	32.0	10.7	16.1
Fine-tuned:					
C. Seq2Mem	1	39.1	51.5	21.2	28.0
	2	39.0	**52.2**	21.6	**28.1**
P. Seq2Mem	1	37.9	50.9	21.5	27.8
	2	39.7	51.7	**22.1**	28.0
Seq2DNC	1	**39.8**	50.8	19.8	26.1
	2	38.0	49.9	19.6	26.8

5.2.2 Evaluation

Table 5.5 shows the training results of the models described in Section 4.3 under the long-term memory mode. The evaluation metrics are defined in Section 4.2.

On their first run, the not fine-tuned models perform identically to before, under short-term memory, which is expected since the task and the models are the same. In the first run, the memory starts empty. On the second run, however, we can see that the Parallel and Combined Seq2Mem models improve marginally. The Seq2DNC performs drastically worse on the second run. Its success rates show a relative decrease of 44.3% and 51.4% on the seen and unseen validation sets respectively. Those values are 40.3% and 43.9% for the oracle success rates.

The results show that both the Combined and Parallel Seq2Mem agents perform better after fine-tuning than before. Even when performing each instruction once, with an empty memory, these models have a higher success rate after fine-tuning than before. The combined Seq2Mem shows a relative improvement in success rate of 10.5% and 10.4% on the seen and unseen validation sets respectively. The Parallel Seq2Mem changes similarly with relative improvements in success rates being 7.4% and 13.2% on the seen and unseen validation sets respectively. The improvements when running an instruction a second time compared to the first time are again only marginal for these models. The Seq2DNC model, which performed best under short-term memory operation, performs worse after fine-tuning on its first run. Its performance still drops on the second run but not as drastically as before fine-tuning.

Chapter 6

Discussion

In this chapter, we will discuss the results of the two experiments performed in chapter 5. We first look at the results of the models under short-term memory operation. Next, we discuss the results of the models with persistent memory, before and after fine-tuning.

6.1 Short-term memory

In this section, we discuss the results of the VLN memory agents without memory persistence, described in Section 5.1. The model performances are shown in Table 5.3. First, we discuss the overall performance of the memory agents compared to the baseline Seq2Seq model [1]. Next, we explain the performance gap between the Combined and Parallel Seq2Mem models on the oracle success rates.

6.1.1 Memory Network performance

The first observation that can be made is that both the Parallel and Combined Seq2Mem models perform worse than the baseline Seq2Seq model [1], while the Seq2DNC performs better than baseline. This can be explained if we look at the differences between the memory components of these networks. Both the Combined and Parallel Seq2Mem use the FIFO memory network described in Section 4.3.1, while the Seq2DNC model uses a DNC [32]. We look at the differences in the **(1)** writing, **(2)** forgetting, and **(3)** reading mechanisms of these networks.

(1) The DNC has a learned writing mechanism. It writes inputs via an attention mechanism with learned weights. The FIFO memory network appends an exact copy of the input to the memory matrix.

(2) The DNC forgets old or unused memories in a learned manner, the FIFO network does not. The DNC erases old memories based on a combination of values predicted by the controller, which is done by learned weights, and by keeping track of which memory slots have been written to and read from in the previous steps. The FIFO memory forgets by removing its oldest memory once the memory is full.

(3) Both networks use several different mechanisms when reading. First, both the DNC and FIFO networks use a learned attention mechanism between a query and the memory slots. Second, they both use information about memory order but do so in different ways. The DNC tracks which memories are written in sequence with a dedicated Link Matrix. The FIFO network does this instead by adding positional embeddings to its memory matrix. Lastly, like with the forget mechanism, the DNC's reading mechanism uses information about the past usage of the memory slots, while the FIFO memory network does not take this into account.

The difference in performance is caused by one or more of these differences in the mechanisms. It is unlikely the write or forget mechanisms are responsible for the discrepancy. When writing, the FIFO network stores the entire input in its memory matrix which results in no information loss. The FIFO network also never loses information by forgetting, since it has more memory slots than navigation steps. Given this reasoning, the expected cause lies with the sequence tracking, usage tracking, and controller parameter prediction mechanisms of the DNC. Since the aim of this thesis is to show the viability of general memory networks in the VLN task, we leave an in-depth analysis into which of these mechanisms are conducive to a high VLN performance for future research.

6.1.2 Seq2Mem oracle success rate

The second observation we make from the results in Table 5.3 is that there exists a discrepancy between the oracle success rates of the Combined and Parallel Seq2Mem models on both the seen and unseen validation sets.

In Figure 5.1b the oracle success rates for each iteration during training are plotted. Here we see that the two curves of the Seq2Mem models lie very close to each other. The final selected model is the one with the highest average non-oracle success rates between the seen and unseen validation sets. Since the oracle success rates of the two models in general over the different iterations align with each other, the difference between the models on the selected iteration is most likely a random outlier and not representative of a significant performance difference.

6.2 Long-term memory

In this section, we discuss the results of the models under long-term memory mode, described in Section 5.2. In Table 5.5 the performance of the memory models is shown where instructions are performed twice with persistent memory. First, we discuss the effects of the persistent memory on the models that are not fine-tuned. Next, we discuss how the models perform after fine-tuning.

6.2.1 Effects of memory persistence

The not fine-tuned Combined and Parallel Seq2Mem both perform better on their second run on all metrics than their first run. This suggests that initiating the navigation task with the memories from the first attempt has a positive effect, but, the increase in performance is too small to be conclusive. The Seq2DNC model, however, performs drastically worse on its second runs. This can be the result of several possible causes: First, the DNC may have erased its early memories while navigating the instruction the first time, leaving only the memories of the end of the trajectory, which may confuse the model on the second run. The second possible cause is the DNC's reliance on its past usage: Restarting navigation a second time may confuse the DNC since now different memories than those most recently used are suddenly relevant.

After fine-tuning, the effect of repeating the instruction with persistent memory is similar to before for the Combined and Parallel Seq2Mem models. We see a slight improvement in the second run, but still too small to be certain of the merits of persistent memory. The Seq2DNC has adapted to the persistent memory and no longer shows a drastic performance drop. However, performance is still lower on its second run than its first. This seems to suggest that, because of the specific mechanisms used by the DNC, the used fine-tuning protocol is not sufficient for the Seq2DNC model. More experimentation is required to learn how to train the models for the long-term memory mode.

6.2.2 Effects of fine-tuning

In Table 5.5 we see the performance of the models before and after fine-tuning with memory persistence. If we compare the performance of the Combined and Parallel Seq2Mem agents, we see that, generally, the fine-tuning had a positive effect on these models. Surprisingly, the fine-tuned models perform better even on the first run, where memory persistence has no effect yet. This suggests that the models can somehow learn from being able to see ahead in the environment during navigation. This training method might be seen as a sort of intermediate method between reinforcement learning, where the agent is rewarded for completing goals, and imitation learning, where the agent is pointed in the right direction each step. The agent has a suggestive guide contained in its memory.

The effect of fine-tuning on the Seq2DNC model is different than the Seq2Mem models. The Seq2DNC performs slightly worse after fine-tuning on its first run than before. However, the drastic performance drop no longer happens. These results are most likely a result of the close interconnectedness between the DNC controller and memory matrix. When the DNC is fine-tuned for a persistent memory, some forgetting occurs, causing a lower performance when navigating from an empty memory. To further investigate this phenomenon in the future, we might retry fine-tuning the Seq2DNC model, carefully selecting which weights to freeze within the DNC.

Chapter 7

Conclusion & Further Work

In the last chapter we conclude by summarizing the experiments we performed in this thesis, and what we have learned from them. Next, we formulate an answer to the research questions posed in the introduction of this thesis. We finish the thesis by mentioning what further research can be done to further explore the ideas of this thesis.

7.1 Conclusion

In this thesis, we have explored the concept of sequence-to-sequence models with an integrated memory network for VLN. We designed a total of three such architectures, using two different memory network architectures. The first two architectures are called the Combined and Parallel Seq2Mem and both use the FIFO memory network architecture. The third agent is called the Seq2DNC and implements a DNC as memory component. Next, to investigate the capabilities of these architectures we perform two experiments. In the first experiment, called *short-term memory*, we train and evaluate the agent with a memory that resets between every instruction. In the second experiment, we explore the possible benefits of the agent memory being persistent between instructions. We do this by first fine-tuning the agents with a persistent memory and then having them perform instructions twice, with the memory from the first run persisting into the second run. We then compare the performance of the first and second run, before and after fine-tuning. This experiment is called *long-term memory.*

Experiment 1: The Seq2DNC model outperformed the baseline Seq2Seq model on all metrics, but both Combined and Parallel Seq2Mem models performed worse than the baseline. This result suggests that a dedicated memory component can improve performance for Seq2Seq VLN agents, under the condition that the memory network has the right read, write, and forget mechanisms. By comparing the different memory networks implemented by the agents, we also gain insight into what mechanisms are beneficial for the memory network. We can suspect that explicit sequence tracking of memories and usage tracking of the memory matrix are important for VLN. We also learn there is no significant difference between the Combined and Parallel Seq2Mem models.

Experiment 2: In second the second experiment, we were not able to show a significant improvement for any of the models when running navigation instructions twice with a persistent memory. The Combined and Parallel Seq2Mem models performed only marginally better on their second runs, while the Seq2DNC model performed worse.

Fine-tuning the models did have a significant positive effect on the Combined and Parallel Seq2Mem models on the performance on both their first and second runs of an instruction. The Seq2DNC model also performed much better on its second runs after fine-tuning. The results may suggest that having access to observations from the environment, in the form of the persisting memories, during training can lead to better performance, but, this is not guaranteed, as seen with the Seq2DNC model. More investigation is required to make a definitive conclusion.

Research questions After performing our experiments and discussing the results, we are now ready to formulate our answers to the research questions posed in the introduction of this thesis.

Can integrating a memory network with a sequence-to-sequence network improve performance on the VLN task?

From the results of our first experiment, described in Section 5.1, we see that, yes, under certain conditions, a Seq2Seq with integrated memory network can outperform a base Seq2Seq model. The condition is that the memory network needs to have the right write, erase, and read mechanisms. We saw that the mechanisms existing in the DNC lead to better results than the FIFO memory network, suggesting that sequence tracking and usage tracking are important in the memory network for VLN agents. In our experiment, the Seq2DNC model outperformed the baseline Seq2Seq model [1], in every category. The two Seq2Mem models performed worse than the baseline, regardless of implementation.

Is the agent able to use a persistent memory to leverage past experiences on new instructions?

We cannot give a decided yes or no answer to this question. In our experiment from Section 5.2, we let agents perform the same instructions twice with a persistent memory. The goal was for the agent to leverage what it had remembered from the first run and, as a result, perform better on the second run. No models performed significantly better on their second run than the first. Since, presumably, using your past experience when re-attempting the same instruction a second time is easier than using it for a new instruction, we can definitely not say we have a positive answer for this research question. We did, however, learn that certain models can be positively influenced by having a persistent memory during training. However, more experimentation is required to make a conclusive statement about the benefits of training with a persistent memory.

7.2 Further Work

In the first experiment in Section 5.1 we showed a a memory agent that outperforms the baseline Seq2Seq VLN agent from 4.1. Current state-of-the-art VLN models perform much better than the models used in this thesis. The best-performing models implement a number of techniques that are not feasible for us to implement due to limited time and computing power. To learn what the full capabilities are of the models designed in this thesis, a future investigation should implement these common methods in our models. Here, we list two of those common methods. First, in this thesis, a simple Reinforcement Learning (RL) training policy was implemented to train the models. Most current research [48, 49, 8] adopt a multi-phase training protocol that combines RL with Imitation Learning (IL). In IL the agent is forced to imitate the behavior of the teacher, showing a quicker convergence. During RL the agent is not forced, allowing it to have more exploration. Combining these policies leads to higher performance. Another often used [11, 8, 24] technique is called back-translation. Back-translation is a form of data augmentation that constructs new natural-language instructions from the predicted routes. The agent is then also trained on these synthetic instructions.

In the discussion of the results of the memory models under short-term memory mode, in Section 6.1.1, we discuss what mechanisms may cause the Seq2DNC model to outperform the Seq2Mem models. Further research should perform an analysis into which mechanisms allow the agents to navigate better.

Lastly, it is still unclear how we can train a memory agent to use a persistent memory to better navigate its environment. further investigation is needed to learn how we can train a VLN agent to this goal. In our experiment from Section 5.2, agents with a persistent memory attempted instructions twice, to see if they could use the experience from the first attempt to navigate better on the second attempt. The next challenge would be to leverage these memories when performing a different instruction, in the same environment. The final challenge could be to bring those past experiences of some environment to a different one, resulting in an agent that can learn and adapt to its surroundings.

Bibliography

[1] Peter Anderson, Qi Wu, Damien Teney, Jake Bruce, Mark Johnson, Niko Sünderhauf, Ian Reid, Stephen Gould, and Anton 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 (CVPR)*, 2018.

[2] Angel Chang, Angela Dai, Thomas Funkhouser, Maciej Halber, Matthias Niessner, Manolis Savva, Shuran Song, Andy Zeng, and Yinda Zhang. Matterport3d: Learning from RGB-D data in indoor environments. *International Conference on 3D Vision (3DV)*, 2017.

[3] Shizhe Chen, Pierre-Louis Guhur, Cordelia Schmid, and Ivan Laptev. History aware multimodal transformer for vision-and-language navigation. *CoRR*, abs/2110.13309, 2021.

[4] Kuan Fang, Alexander Toshev, Li Fei-Fei, and Silvio Savarese. Scene memory transformer for embodied agents in long-horizon tasks. *CoRR*, abs/1903.03878, 2019.

[5] Yubo Zhang, Hao Tan, and Mohit Bansal. Diagnosing the environment bias in vision-and-language navigation. *CoRR*, abs/2005.03086, 2020.

[6] Weituo Hao, Chunyuan Li, Xiujun Li, Lawrence Carin, and Jianfeng Gao. Towards learning a generic agent for vision-and-language navigation via pre-training. *CoRR*, abs/2002.10638, 2020.

[7] Arjun Majumdar, Ayush Shrivastava, Stefan Lee, Peter Anderson, Devi Parikh, and Dhruv Batra. Improving vision-and-language navigation with image-text pairs from the web. *CoRR*, abs/2004.14973, 2020.

[8] Hao Tan, Licheng Yu, and Mohit Bansal. Learning to navigate unseen environments: Back translation with environmental dropout. *CoRR*, abs/1904.04195, 2019.

[9] Howard Eichenbaum. The role of the hippocampus in navigation is memory. *J Neurophysiol*, 117(4):1785–1796, February 2017.

[10] Melissa V. The meaning and structure of scenes. *Vision Research*, 181:10–20, 04 2021.

[11] Hanqing Wang, Wenguan Wang, Wei Liang, Caiming Xiong, and Jianbing Shen. Structured scene memory for vision-language navigation. *CoRR*, abs/2103.03454, 2021.

[12] Zhiwei Deng, Karthik Narasimhan, and Olga Russakovsky. Evolving graphical planner: Contextual global planning for vision-and-language navigation. *CoRR*, abs/2007.05655, 2020.

[13] Wansen Wu, Tao Chang, and Xinmeng Li. Visual-and-language navigation: A survey and taxonomy. *CoRR*, abs/2108.11544, 2021.

[14] Michal Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek, and Wojciech Jaskowski. Vizdoom: A doom-based AI research platform for visual reinforcement learning. *CoRR*, abs/1605.02097, 2016.

[15] Devendra Singh Chaplot, Kanthashree Mysore Sathyendra, Rama Kumar Pasumarthi, Dheeraj Rajagopal, and Ruslan Salakhutdinov. Gated-attention architectures for task-oriented language grounding. *CoRR*, abs/1706.07230, 2017.

[16] Yi Wu, Yuxin Wu, Georgia Gkioxari, and Yuandong Tian. Building generalizable agents with a realistic and rich 3d environment. *CoRR*, abs/1801.02209, 2018.

[17] Eric Kolve, Roozbeh Mottaghi, Daniel Gordon, Yuke Zhu, Abhinav Gupta, and Ali Farhadi. AI2-THOR: an interactive 3d environment for visual AI. *CoRR*, abs/1712.05474, 2017.

[18] Claudia Yan, Dipendra Kumar Misra, Andrew Bennett, Aaron Walsman, Yonatan Bisk, and Yoav Artzi. CHALET: cornell house agent learning environment. *CoRR*, abs/1801.07357, 2018.

[19] Vihan Jain, Gabriel Magalhães, Alexander Ku, Ashish Vaswani, Eugene Ie, and Jason Baldridge. Stay on the path: Instruction fidelity in vision-and-language navigation. *CoRR*, abs/1905.12255, 2019.

[20] Gabriel Ilharco, Vihan Jain, Alexander Ku, Eugene Ie, and Jason Baldridge. General evaluation for instruction conditioned navigation using dynamic time warping. *CoRR*, abs/1907.05446, 2019.

[21] Haoshuo Huang, Vihan Jain, Harsh Mehta, Jason Baldridge, and Eugene Ie. Multi-modal discriminative model for vision-and-language navigation. *CoRR*, abs/1905.13358, 2019.

[22] Wang Zhu, Hexiang Hu, Jiacheng Chen, Zhiwei Deng, Vihan Jain, Eugene Ie, and Fei Sha. Babywalk: Going farther in vision-and-language navigation by taking baby steps. *CoRR*, abs/2005.04625, 2020.

[23] Yicong Hong, Cristian Rodriguez Opazo, Qi Wu, and Stephen Gould. Sub-instruction aware vision-and-language navigation. *CoRR*, abs/2004.02707, 2020.

[24] Daniel Fried, Ronghang Hu, Volkan Cirik, Anna Rohrbach, Jacob Andreas, Louis-Philippe Morency, Taylor Berg-Kirkpatrick, Kate Saenko, Dan Klein, and Trevor Darrell. Speaker-follower models for vision-and-language navigation. *CoRR*, abs/1806.02724, 2018.

[25] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[26] Federico Landi, Lorenzo Baraldi, Marcella Cornia, Massimiliano Corsini, and Rita Cucchiara. Perceive, transform, and act: Multi-modal attention networks for vision-and-language navigation. *CoRR*, abs/1911.12377, 2019.

[27] Zongkai Wu, Zihan Liu, Ting Wang, and Donglin Wang. Improved speaker and navigator for vision-and-language navigation. *IEEE MultiMedia*, 28(4):55–63, 2021.

[28] Jason Weston, Sumit Chopra, and Antoine Bordes. Memory networks, 2014.

[29] Sainbayar Sukhbaatar, Arthur Szlam, Jason Weston, and Rob Fergus. Weakly supervised memory networks. *CoRR*, abs/1503.08895, 2015.

[30] Alexander H. Miller, Adam Fisch, Jesse Dodge, Amir-Hossein Karimi, Antoine Bordes, and Jason Weston. Key-value memory networks for directly reading documents. *CoRR*, abs/1606.03126, 2016.

[31] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *CoRR*, abs/1410.5401, 2014.

[32] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwiska, Sergio Gmez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, Adri Puigdomnech Badia, Karl Moritz Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen King, Christopher Summerfield, Phil Blunsom, Koray Kavukcuoglu, and Demis Hassabis. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471476, 2016.

[33] Yi Zhu, Fengda Zhu, Zhaohuan Zhan, Bingqian Lin, Jianbin Jiao, Xiaojun Chang, and Xiaodan Liang. Vision-dialog navigation by exploring cross-modal memory. *CoRR*, abs/2003.06745, 2020.

[34] Steven W. Chen, Nikolay Atanasov, Arbaaz Khan, Konstantinos Karydis, Daniel D. Lee, and Vijay Kumar. Neural network memory architectures for autonomous robot navigation. *CoRR*, abs/1705.08049, 2017.

[35] Ankit Kumar, Ozan Irsoy, Jonathan Su, James Bradbury, Robert English, Brian Pierce, Peter Ondruska, Ishaan Gulrajani, and Richard Socher. Ask me anything: Dynamic memory networks for natural language processing. *CoRR*, abs/1506.07285, 2015.

[36] Donghyun Kang and Minho Lee. Seq-dnc-seq: Context aware dialog generation system through external memory. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2019.

[37] Alex Sherstinsky. Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network. *CoRR*, abs/1808.03314, 2018.

[38] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6:107–116, 04 1998.

[39] Sepp Hochreiter and Jrgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997.

[40] Wikipedia contributors. Long short-term memory — Wikipedia, the free encyclopedia, 2010. [Online; accessed 22-June-2022].

[41] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks, 2014.

[42] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2016.

[43] Chunjie Luo, Jianfeng Zhan, Lei Wang, and Qiang Yang. Cosine normalization: Using cosine similarity instead of dot product in neural networks, 2017.

[44] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[45] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael S. Bernstein, Alexander C. Berg, and Li Fei-Fei. Imagenet large scale visual recognition challenge. *CoRR*, abs/1409.0575, 2014.

[46] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. *CoRR*, abs/1508.04025, 2015.

[47] Alex Lamb, Anirudh Goyal, Ying Zhang, Saizheng Zhang, Aaron Courville, and Yoshua Bengio. Professor forcing: A new algorithm for training recurrent networks, 2016.

[48] Xin Wang, Qiuyuan Huang, Asli Celikyilmaz, Jianfeng Gao, Dinghan Shen, Yuan-Fang Wang, William Yang Wang, and Lei Zhang. Reinforced cross-modal matching and self-supervised imitation learning for vision-language navigation. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6622–6631, 2019.

[49] Yicong Hong, Cristian Rodriguez Opazo, Yuankai Qi, Qi Wu, and Stephen Gould. Language and visual entity relationship graph for agent navigation. *CoRR*, abs/2010.09304, 2020.