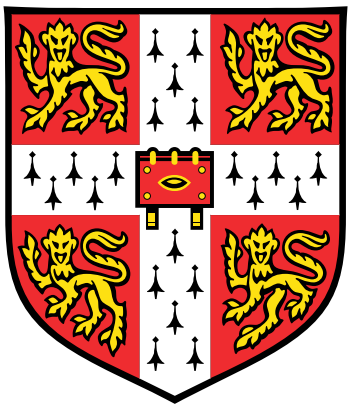
Natural Language to Neural Programs



Daniel Simig

Department of Engineering
University of Cambridge

This dissertation is submitted for the degree of
Master of Philosophy in Machine Learning, Speech and Language Technology

Homerton College

August 2017

Declaration

I, Daniel Simig of Homerton College, being a candidate for the M.Phil in Machine Learning, Speech and Language Technology, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed:

Date:

Word count: 11677

Daniel Simig
August 2017

Acknowledgements

I would like to thank my industrial supervisor, Nate Kushman from Microsoft Research Cambridge for his extremely valuable advice on this research. I would also like to thank Prof. Bill Byrne for his guidance throughout this project. I am grateful to Microsoft Research for providing me with computational resources which were essential to this work. Lastly, I am grateful to my parents for their unconditional support throughout my university years. Without them this work would not have been possible.

Abstract

Recent advances in neural program models allow us to build purely neural network based architectures that learn to execute complex, algorithmic tasks. However, these models require formal, structured specifications of the task. This thesis explores how well-known natural language processing techniques such as attention and word embeddings can be adopted in order to allow Neural Programmer-Interpreters (NPI-s) to take natural language instructions directly as their input. We propose two novel, attention-based architectures that extend the NPI model: The first model computes attention over the input word sequence and thus finds the key bits of informations required to make the right decisions when executing the task. With the second model we show the feasibility of using the attention mechanism over the world state in order to interpret references to the observed world in the instructions. Our models outperform our baseline, a trivial modification of original NPI model by up to 65% accuracy in particular experiments.

Table of contents

Chapter 1

Introduction

One of the most recognised machine learning (ML) experts of our time, Andrew Ng defines ML as "the science of getting computers to act without being explicitly programmed". [1] While this is clearly an overly condensed description of what machine learning truly is, it conveys well a general expectation we have towards ML. In this work we take this definition in the most literal sense and attempt to build models that understand instructions formed in natural language and produce a sequence of actions in order to execute them.

1.1 Existing work

Understanding and executing instructions given in natural language has been a major research area in the past 10 years. Particular applications include navigation [2], manipulation of spreadsheets [3], or instructing robots to execute everyday tasks [4].

Traditionally these problems were solved by first parsing the input sentences into some logical structures, then mapping those to action sequences ([5], [6], [7]). These forms typically need a set of hand-crafted terms, inherently limiting the generalisation ability of these techniques. Some attempts have been made to eliminate the need for prior linguistic knowledge ([8], [9]), but the ability of these models to produce a complex sequence of actions is still limited.

A more recent idea is to represent programs in a purely neural network based way, without incorporating expert knowledge about what the role of particular program is (starting with [10]) These models can learn to execute rather abstract and complex algorithms such as sorting numbers or carrying out arithmetic on a scratch-pad even from relatively little training data. Due to the fact that they were largely inspired by the way computers execute commands, they expect very precise and low level inputs just as a computer would do.

To summarise, there has been a large amount of work on understanding instructions in natural language and there are some exciting new models able to execute complex, algorithmic tasks. However, the author is not aware of any work that connects two classes of solutions in order to execute natural language instructions that require a complex sequence of actions.

1.2 Research aims and results

In this work we attempt to bridge the gap between the above mentioned two types of solutions. By using abstractions in the neural representation that handle the complexity of the execution, we are able to use a number of simple Natural Language Processing (NLP) techniques to extract the information from the natural language required to carry out a rather complex task. In our case this task is to construct different shapes on a 2D grid from simple building blocks, similar to the one described in [11].

The key idea in our approach was motivated by how humans would execute these tasks: When presented with the instruction, one would naturally look for key bits of information, such as what colour or material we have to use, or where do we need to put the objects. At the same time, we would likely not focus on parts of the instructions that do not carry useful information. The same idea can be applied to observing the world around us: at any moment we tend to focus on objects that are relevant to our current activity.

In the NLP literature this idea is called the "attention mechanism" [12]. Our work largely consisted of adapting this idea of attention to help neural program architectures understand natural language. In particular, we aimed to answer the following questions:

- Is the current NPI model able to deal with natural language input with minimal modifications?
- Can standard NLP techniques such as attention be used to improve on the performance of the original NPI model?
- Furthermore, can we introduce an attention mechanism over world observations in order to help understand references to the world in the instructions (often called language grounding)?

First we established that with only little modification the existing NPI model was able to significantly outperform a baseline sequence to sequence model [9]. We have then successfully introduced a number of changes (based on a modified version of the attention model) to the original architecture that in combination made our model perform even better,

increasing accuracy by up to 55% in certain settings. Finally, we have constructed a new task and further modified the model in order to show the feasibility of using world references in the input sentences. Adding this new type of attention resulted in up to 58% increase in accuracy in the new task.

1.3 Organization of the chapters

In Chapter 2 we describe a number of architectures and concepts our work builds on. This chapter introduces all the information and notation that is required to understand later parts of this work. Chapter 3 and Chapter 4 precisely describe the tasks we were trying to solve and the models we used to achieve this. Chapter 5 summarises our results, provides some analysis of how the models learned to execute the tasks and compares the findings to our expectations. Chapter 6 showcases works that succeeded in solving problems similar to ours. Finally, chapter 7 summarises the main achievements of this work and describes a number of possible extensions to it.

Chapter 2

Background

This chapter is structured into three main parts: In the first part we introduce some basic concepts that appear frequently in various sections of this report. In the second and the third part of this chapter we present existing work in the two areas we try to connect in our work: natural language processing and neural programs.

2.1 Sequence processing

Dealing well with sequences is a key component to solving the kind of tasks we are interested in, as either the input (natural language), the output (actions, program calls) or both are sequences. More importantly, these sequences can have variable length, thus traditional Neural Networks are not able to deal with them efficiently. Recurrent Neural Networks, or their more advanced variants, Long-Short Term Memory (LSTM) networks are a lot better in handling this kind of problem. In this section we formally describe these models, as majority of the works described in the following chapters make extensive use of these type of architectures.

2.1.1 Recurrent Neural Networks

In order to deal with sequential input, RNN-s process one input at a time, but maintain a so called hidden state that contains information about all the previously processed inputs. Optionally, at each timestep an output vector is also emitted. There are a number of variants of this model, but in this work we define and use the Elman-type [13] architecture for a sequence of inputs $\mathbf{x_t}$ over time $t = 1..T$:

$$\mathbf{h_t} = \sigma_h(W_{xh}\mathbf{x_t} + W_{hh}\mathbf{h_{t-1}}) \tag{2.1}$$

$$\mathbf{y_t} = \sigma_y(W_{hy}\mathbf{h_t}) \tag{2.2}$$

Note that $\mathbf{h_0}$ is not defined - this can be a constant initialisation, some random noise, but also some vector that carries information about the task to be solved - as we can see in later sections. $W_{..}$ are weight matrices of the appropriate dimensions [1] and $\sigma_.$ are some nonlinear activation functions similarly to traditional NN-s. These architectures are "rolled out" to fit the maximum length input.

Typically there are two outputs of this network that we are interested in. $\mathbf{y_t}$ over time $t = 1..T$ can represent some transformation of the time series. Additionally, one might only be interested in the last hidden state, $\mathbf{h_T}$, as this single, fixed length vector can encode all important information from the variable-length input.

2.1.2 Multi-layer RNN-s

Similarly to hidden layers of NN-s, RNN-s can be stacked on top of each other. Consider the n-th layer of a multi-layer RNN with layers $n \in 1..N$:

$$\mathbf{h_t^n} = \sigma_h(W_{xh}\mathbf{y_t^{n-1}} + W_{hh}\mathbf{h_{t-1}^n}) \tag{2.3}$$

$$\mathbf{y_t^n} = \sigma_y(W_{hy}\mathbf{h_t^n}) \tag{2.4}$$

The intuition is that the stacked layers can potentially learn more and more abstract features. The final output of the network is $\mathbf{y_t^N}$. The set of hidden states in the last timestep $\{\mathbf{h_T^n}\}_{n\in 1..N}$ is again a fixed size representation of the variable length input.

2.1.3 Bi-directional RNN-s

The RNN architectures are inherently asymmetrical, as inputs at the beginning of the sequence have to go through many more transformations as the one towards the end. To address this issue, one might build a "backwards" network with identical architecture but input $\mathbf{x'_t} = \mathbf{x_{T-t+1}}$. Hidden states and outputs of this network can be defined as the concatenations of hidden states and outputs of the forward and backward networks.

[1] In order to make equations more readable throughout this report, we do not write out the bias terms separately.

2.1.4 Encoder - decoder models

In cases where both the input and the output are sequences, a commonly used method is the following two-step process: First the input sentence is encoded into some hidden representation using an encoder RNN. In the second step, the hidden state of a decoder RNN is initialised with this hidden representation, and produces the output sequence. [14] introduced this type of models using it for translation tasks. This encoder-decoder model (often referred to as Neural Machine Translation, NMT in translation literature) was able to outperform previous Statistical Machine Translation (SMT) systems despite not making any assumptions about the data [14].

2.1.5 Long-Short Term Memory Networks

The update of the hidden state in a traditional RNN is a rather simple operation with limited expressive power. [15] define a number of gates to control the value of a so called LSTM-cell. This cell is persistent over time and carries information from all earlier inputs similarly to the hidden state of a traditional RNN. The gates at time t are defined as follows:

- Input gate: $\mathbf{i_t} = \sigma_g(W_{xi}\mathbf{x_t} + W_{hi}\mathbf{h_{t-1}})$
- Forget gate: $\mathbf{f_t} = \sigma_g(W_{xf}\mathbf{x_t} + W_{hf}\mathbf{h_{t-1}})$
- Output gate: $\mathbf{o_t} = \sigma_g(W_{xo}\mathbf{x_t} + W_{ho}\mathbf{h_{t-1}})$

Where the activation function σ_g is typically the Sigmoid function. The cell and the hidden states are updated based on the values of the gates above:

$$\mathbf{c_t} = \mathbf{f_t} \circ \mathbf{c_{t-1}} + \mathbf{i_t} \circ \sigma_c(W_{xc}\mathbf{x_t} + W_{hc}\mathbf{h_{t-1}}) \tag{2.5}$$

$$\mathbf{h_t} = \mathbf{o_t} \circ \sigma(\mathbf{c_t}) \tag{2.6}$$

An LSTM cell is illustrated on Figure 2.1. Note that LSTM-s can be stacked just like RNN-s. In order to do this, just use the hidden units in layer n at time t as the input to layer n+1 at time t: $x_t^{n+1} = h_t^n$. The bi-directional extension in Section 2.1.3 also applies to LSTM-s. In this document the notation for LSTM-s will be $\mathbf{h^t} = f_{lstm}(\mathbf{h^{t-1}}, \mathbf{x^t})$, where $\mathbf{h^t}$ encapsulates all hidden and cell states. LSTM-s are a special type of RNNs, and the terms LSTM and RNN are often interchangeably used as LSTM-s can be applied in virtually any scenario where traditional simple RNN-s are used.

LSTM-s have been shown to perform better than traditional RNN-s in various tasks. In the original work [15] it's power was demonstrated by learning the well-defined grammar in

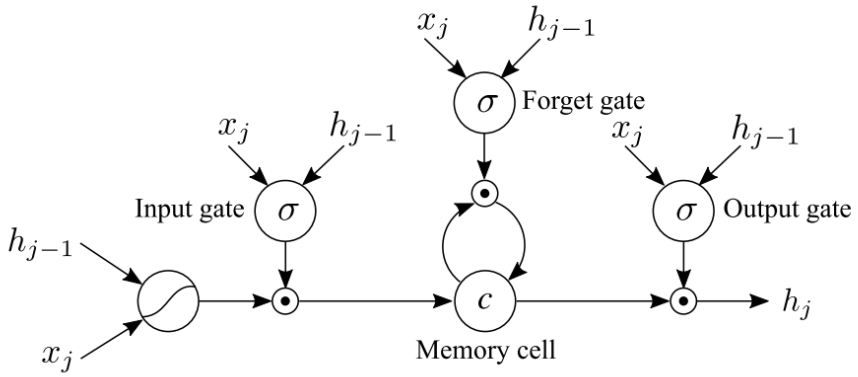


Fig. 2.1 An LSTM cell. Source: [9]

text inputs. LSTM-s are particularly important for our work as these networks provided the basis for most of the neural program architectures.

2.2 Natural Language Processing

2.2.1 Attention

Although applying the encoder-decoder model for machine translation was a tremendous success (see [14]), the architecture has a major limitation. The model is forced to encode the whole (variable-length) input sequence into a relatively small, fixed size vector, at the risk of losing critical information about the input data. In order to overcome this issue, [12] uses all the hidden states $\{\mathbf{h_{in}^t}\}_{t\in 1..T}$ of the encoder RNN in order to help the decoding process in a very specific way mimicking the way humans can focus on parts of a sentence.

At each time step j of the decoding, an alignment vector $\boldsymbol{\alpha_j}$ is computed that describes a distribution over the elements of the input sequence. The intuition behind this is that a given element in the output sequence might only be related to a well-defined region of the input sequence - the alignment vector will try to capture this. Using this distribution a weighted average is taken over the hidden states of the encoder LSTM (called the context, $\mathbf{c_j}$) which is then used as an input to the decoding RNN. As the concept of attention is central to our work, we now formally define it:

Consider time step j in the decoding process. Let the hidden state at this timestep be $\mathbf{h_{out}^j}$. The relevance e_{ij} of position i in the input sequence to position j in the output sequence is defined as

$$e_{ij} = a(\mathbf{h_{in}^i}, \mathbf{h_{out}^j})$$

where $a(\dot{)}$ is the aligner model. In the original paper this is a feedforward neural network that is trained simultaneously with other parts of the model. Another popular (and simpler) option is to use the cosine distance of the two vectors.

The alignment vector is created by taking a softmax over the appropriate relevances, and then used to create the context vector:

$$\alpha_{ij} = \frac{exp(e_{ij})}{\sum_{k=1}^{T} exp(e_{kj})} \tag{2.7}$$

$$\mathbf{c_j} = \sum_{i=1}^{T} \alpha_{ij} \mathbf{h_{in}^i} \tag{2.8}$$

Note that the hidden state $\mathbf{h_{in}^t}$ of an RNN can be interpreted in various ways. [12] uses a bi-directional network and a concatenation of forward and backward hidden states to compute the alignment. A diagram illustrating this architecture is shown on Figure 2.2 (in the notation of the diagram $\mathbf{s_j} = \mathbf{h_{out}^j}$). [9] additionally concatenates the current input at time t, $\mathbf{x_{in}^t}$ to the hidden state $\mathbf{h_{in}^t}$ and uses the resulting vector as the basis for computing the alignment. This allows to directly choose words in the input that we want to pay attention to. This model is called a multi-level aligner.

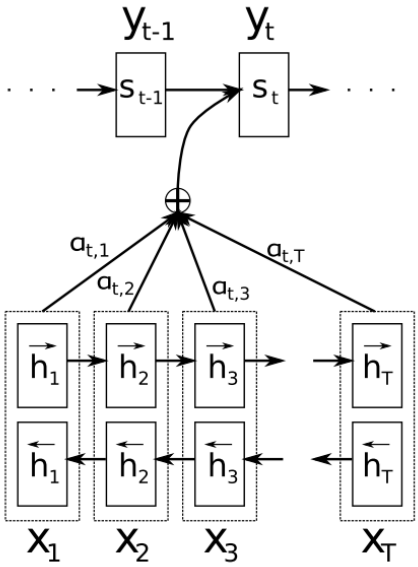


Fig. 2.2 A simplified representation of the attention architecture from [12]

2.2.2 Word embeddings

The simplest way to mathematically represent a word is to use a one-hot representation. For a vocabulary size of N, we need a vector of size N consisting of 0-s except for one location set to 1. The limitations of this representation have been known for a long time. Firstly, it is hard to scale this model as the size of the vector and hence the number of parameters required to process this vector grows linearly with the vocabulary size, making training harder and more vulnerable to overfitting.

Secondly, this primitive approach does not allow for representing similarities and relations between different words, that is, does not convey the meaning of the word. By only looking at these vectors there is no way of telling whether "apple" and "pear" are more similar to each other than let's say "apple" and "guitar".

These problems were encountered in the case of Neural Network Language Models (NNLM-s) [16] and solved by a better approach, word embeddings (called Distributed Representations at the time). The idea behind word embeddings is to represent each word as a point in a relatively low (typically not larger than a few hundred) dimensional, continuous space. In this space similar words are forced to be close to each other, hence the "meaning of the word" becomes present in some sense in this representation. An additional benefit is of course the significantly reduced number of parameters. Dimensionality reduction techniques such as PCA [17] or t-SNE [18] are often used to map embeddings into 2 or 3 dimensional spaces allowing them to be visualised. An example of this is shown later in this report on Figure 5.5

These vector representations are learned jointly with the model they are part of. In the case of [16] the model is a neural network that represents a language model. A crucial property of this concept, however, is is that after learning these representations for a particular task the vector representations can potentially be reused in a different task without further modifications. Many alternative approaches have been proposed to train these embeddings, for example Word2Vec [19], GloVe [20] or fastText [21]. We briefly describe the first one as it is used in this work.

Word2Vec

The model called Word2Vec was invented by researchers at Google [19]. It simultaneously uses word embeddings to complete two tasks. The first is called Continuous Bag of Words (CBOW) - this is an N-gram model where a NN-based classifier is trained to predict the next word based on the average of the embeddings of n consequent words. The second task is the so-called skip-gram model where the embedding the i^{th} single word is used to predict the (i +

d)th word, where d is a randomly chosen number between 1 and some maximum distance. This model was used to train 300-dimensional word embeddings based on 100B words [22].

2.3 Neural Programs

2.3.1 Neural Turing Machines

RNN-s (and thus LSTM-s) have been shown to be Turing-complete for a long time [23], however methods for teaching such networks to carry out algorithmic tasks have only recently been discovered. Neural Turing Machines (NTM) are one of the first architectures to achieve this goal [10], and as such they introduce a number of key concepts also used in later works. Since it is not directly related to our work we only briefly describe it's architecture, focusing on the ideas that are relevant to later sections.

Outline of the architecture

NTM-s are an extension of recurrent neural networks with a memory unit and the ability to write to and read from locations in the memory using heads (following the nomenclature of regular Turing Machines). This memory-handling mechanism is facilitated by the so called controller network, which can be built using any of the recurrent architectures (RNN, LSTM) described in Section 2.1.

First the hidden state of the controller network is initialised with some (learned) hidden state bias vector. Then in each time step, this network takes some external input (world observations typically), and produces an output vector. From this output vector, a number of mappings with specific purposes create a number of output variables that control memory I/O. These variables are responsible for:

- Addressing the memory by defining a continuous distribution over locations to be written to / read from. This is done by both content-based methods (finding matching patterns) and location-based methods (defining shifts).
- Determining the content to be written. This is done by erase and add vectors, similarly to the gates of LSTM networks
- Determining the "sharpness" of the continuous distributions over states for reads / writes.

Upon execution of memory operations the state of the memory is updated and some final output (required by the task) is produced.

Results and main achievements

NTM-s have been tested on a number of algorithmic tasks, including copying long sequences, repeated copies (involving loops), associative recall and sorting. All of the listed tasks were completed nearly perfectly. The key advantage of NTM-s seemed to be that they were able to generalise to much larger datasets than the ones seen at training times. This was achieved by learning algorithms that behave very similarly to human-constructed algorithms despite being encoded as a set of continuous weights in a fully differentiable network (thanks to the soft addressing mechanisms).

2.3.2 Neural Programmer-Interpreters

Neural Programmer-Interpreter (NPI) [24] is an architecture that learns to represent and execute programs. Similarly to the previously described architecture, it has an LSTM-based controller mechanism that in each time step decides about some operation to take. In contrast to the previous approaches, instead of memory read/write operations the LSTM outputs determine program calls and arguments to those calls, or the opposite, they instruct to return from a subprogram. The model is trained from full execution traces (calls to subprograms and elementary operations with arguments) of solving a task. Our work is based on this architecture, more precisely on an implementation of this model by the authors of [11], described in Section 6.2. For this reason, we present NPI in a mindset similar to how [11] describes and uses NPI.

Stack-based program execution

Analogously to traditional program executions, NPI-s operate a stack of LSTM instances (and parallel to this a stack of the program calls with corresponding arguments). In the first timestep the LSTM for the top-level program is initialised and pushed on top of the stack. From this point, at every timestep an input-output pair is added to the sequence of inputs / outputs for the LSTM instance on the top of the stack. Here the input is an ensemble of a number of different things: (1) a learnable embedding of the program currently executed, (2) arguments for this program call, and (3) an embedding of the current world state. The output is the entry in the execution trace for this timestep. The precise format of the output is defined later in this section.

After the input and output is added to the current LSTM instance, one of the following three events happen based on the current operation in the execution trace:

- If a subprogram is called (PUSH), a new instance of the interpreter LSTM is initialised with new hidden and cell states. This is pushed onto the top of the stack, moving the old LSTM down the stack.
- On a return (POP) event the exact opposite happens: the current LSTM instance is discarded and popped from the stack, bringing the previous LSTM (the caller) to the top of the stack.
- On a call to an elementary program (OP), the stack of LSTM-s remains unchanged.

It has to be emphasized that throughout this process only a single set of weights is used for all LSMT-s. Thus, a single execution trace can be considered as a set of training examples (one for each (sub)program call) for training a generic interpreter LSTM. Since the currently called (sub)program and the arguments are included in the input at a given time step, this generic interpreter LSTM learns to understand what program is currently called and act correspondingly.

The main contribution of this thesis is extending the architecture of this interpreter LSTM in a way that it can handle Natural Language inputs instead of a precisely defined top-level program call with a set arguments. In the following we formally define the architecture used for this interpreter LSTM and introduce the notation that will also be used to describe our contribution.

The interpreter LSTM

Consider a (potentially multi-layer) LSTM with input $\mathbf{u^t}$ and hidden states in its final layer $\mathbf{h^t_{out}}$ as described in Section 2.1.:

$$\mathbf{h^t_{out}} = f_{lstm}(\mathbf{h^{t-1}_{out}}, \mathbf{u^t}) \tag{2.9}$$

$$\mathbf{u^t} = f_{enc}(\mathbf{w^t}, \mathbf{g^t_{in}}) \tag{2.10}$$

Here $\mathbf{w^t}$ is a suitable representation of the world and $\mathbf{g^t_{in}}$ is a task-specific joint encoding of the currently executed program and arguements. f_{enc} is a task-specific encoder of all this information. Mover, from the final hidden state 4 different output variables are mapped:

- $\mathbf{p^t_a} = W_a\mathbf{h^t_{out}}$ determines the action to be taken (PUSH, POP, or OP).
- $\mathbf{p^t_g} = W_g\mathbf{h^t_{out}}$ determines the subprogram to call (if $\mathbf{p^t_a}$ = PUSH)
- $\mathbf{p^t_o} = W_o\mathbf{h^t_{out}}$ determines the elementary operation to perform (if $\mathbf{p^t_a}$ = OP)

- $\mathbf{p^t_{r,i}} = W_{r,i}\mathbf{h^t_{out}}$ determines the i$^{\text{th}}$ argument to the called subprogram or elementary operation.

All of the above variables are represented as one-hot values over possible actions, programs, elementary operations and possible (discrete) values of arguments respectively. $\mathbf{g^{t+1}_{in}}$ is constructed based on these output variables. This model together is the core interpreter LSTM of NPI-s, shown on 2.3.

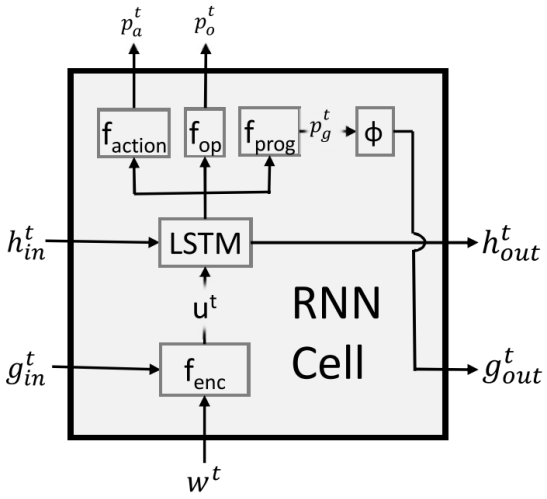


Fig. 2.3 A simplified representation of the NPI interpreter from [11]. Note that the argument outputs $\mathbf{p^t_{r,i}}$ are omitted for clarity.

Training and evaluation

Training data consists of pairs of top-level program arguments (defining the task to be executed) and fully decorated execution traces providing ground truth values for all the output variables described above. The network is trained to simply maximalise the joint log-probability [2] of the execution trace given the top-level program arguments $\mathbf{g^1_{in}}$.

Note that two different ways of training this model correspond to two different aims of NPI-s:

- Training a network with fixed weights for the interpreter LSTM can be considered as programming: only the observation encoders and program embeddings are trained, while the interpreter is unchanged.

[2] One can take the softmax of any of those output variables. As a result of this, the output variable can be considered as a probability distribution, assigning a probability to the true value.

- Training the network without fixing any weights is equivalent to building an interpreter.

At evaluation time, in each timestep, the argmax of the one-hot output vectors is used to determine the next step to take. The interpreter LSTM is run until a POP action is taken in the top-level program call, producing a full execution trace.

Results

The above described model was tested in a range of applications: number addition and bubble-sorting on a scratch pad, rotating 3D models. For all of these tasks the model was able to learn already from an extremely small amount of data (8 examples for sorting). Furthermore, the model achieved previously unseen generalisation abilities by being able to sort sequences 3 times as long as any provided training example.

On top of this, the proposed model generalises not only quantitatively but also qualitatively. In the 3D rotation experiment the input was an image represented as an array of pixels (as opposed to the scratchpad in other experiments) and the actuator was moving the camera around the model (as opposed to writing digits). The same core interpreter module was successfully used to solve this task, demonstrating the versatility of NPI-s.

Finally, [11] builds on top of this work and evaluates it as a baseline for the Nanocraft task as described in Section 6.2. The NPI model is able to perfectly learn to construct a "house" from only 128 examples. We will use variants of this task in order to evaluate our work.

Chapter 3

Task Specification

The task we are trying to solve in this work is based on the task called Nanocraft, which appeared originally in [11]. The authors of this paper have created an implementation of the NPI architecture suitable for this task, our work builds on top of that. In this chapter we describe this dataset, explain how the NPI architecture in [11] was suited to this task, finally we describe the modifications we made to this dataset.

3.1 The Nanocraft Task

The Nanocraft world consists of a d by d grid (16 X 16 in our experiments) and a robot. This robot can freely move around the grid and place blocks at different positions of the grid. Blocks have two attributes, color (reg, green, blue, orange or yellow) and material (iron, steel, glass, wood or plastic). If these blocks are arranged into a rectangle shape, we call those shapes houses. Houses can have different positions, widths and heights (each attribute can take 5 different values). The robot is tasked to build a house by executing a sequence of elementary operations. There are two elementary operations, ACT_Move and ACT_PlaceBlock. On top of the elementary operations, there are higher level functions (often referred to as subprograms) that the robot is using while executing the task, such as MoveMany or BuildWall. Building a house consists of 2 MoveMany and 4 BuildWall calls, as illustrated on Figure 3.1. Table 3.1 gives an overview of the functions used in Nanocraft. Note that for easier implementation we use the maximum 4 arguments all the time, with unused arguments are set to 0.

In order to force models to condition their decisions on the world observations, some blocks are pre-built and already present on the grid - no PlaceBlock action is needed at those positions even if building a house would involve a block at that position.

name	arguments
MoveMany	distance, direction
BuildWall	distance, direction, color, material
ACT_Placeblock	color, material
ACT_Move	direction

Table 3.1 Subprograms and their required arguments for the Nanocraft task

The training data consists of tuples of house specifications, fully decorated execution traces and world states at every step of execution: $\{(\mathscr{H}_i, \mathscr{E}_i, \mathscr{W}_i)\}_{i=1}^{N}$ where N is the number of samples. A house specification is simply a tuple of six integers representing where the house should be built, what dimensions it should have, what colour and material should be used: $\mathscr{H}_i = (PosX, PosY, SizeX, SizeY, Col, Mat)$.

Execution traces contain the function calls at each time step (including higher-level subprograms and elementary actions) along with the supplied arguments to those calls. Formally we can define an execution trace analogously to the output of the NPI (refer to Section 2.3.2):

An execution trace $\mathscr{E}_i$ is defined as

$$\mathscr{E}_i = \{A^t, G^t, O^t, R^t\}_{t=1}^{T_i}$$

A^t and G^t are integers representing the elementary operation or subprogram calls respectively at time step t. $O^t \in \{0,1,2\}$ represents whether at a given timestep t a PUSH, POP, or OP happened. Finally $R^t = \{r_j^t\}_{j=1}^{4}$ is the set of 4 arguments passed to the subprogram / elementary operation. Note that depending on the value of O^t some of these variables are not defined - we use a special integer for this that "pad" the sequence.

An execution trace is illustrated on Figure 3.1, although note that the used set of functions on the illustration is slightly different from the one described above.

The world is represented as a set of feature maps :

$$\mathscr{W}_i = \{W_j\}_{j=1}^{n}, W_j \in \{1..d_j\}^{DXD}$$

Each map has size D by D and has d_j possible values in each position, we call this the depth of the map. In the original implementation there are $n = 3$ maps: a one-hot map for showing the position of the robot and two maps that represent the colors and materials of pre-built blocks (locations without a block are represented as 0-s, built blocks are represented with some non-zero code for their color and material).

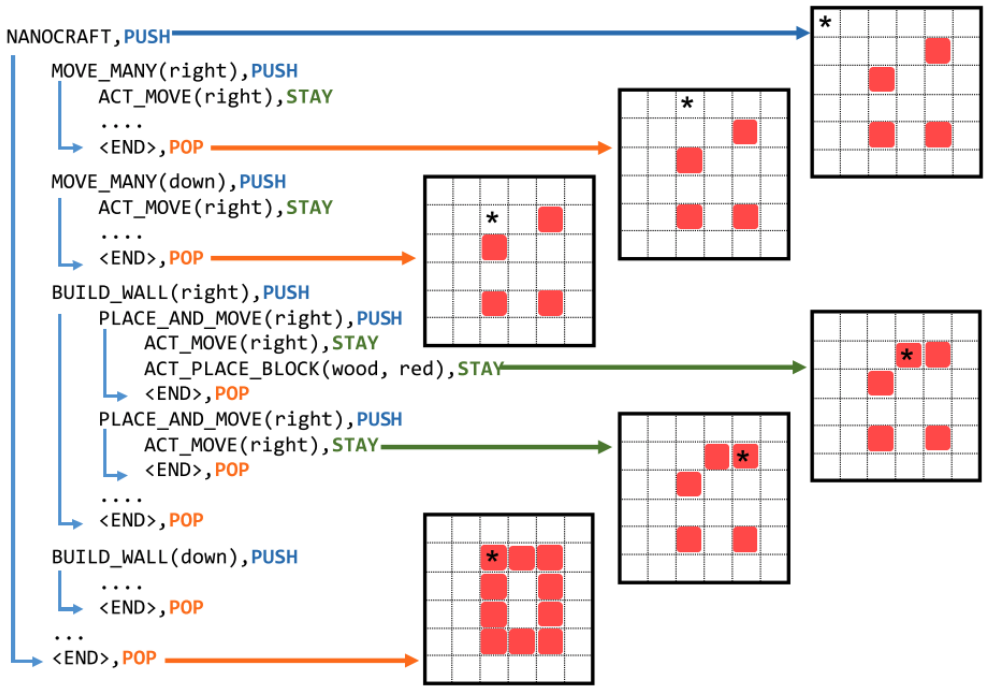


Fig. 3.1 An illustration of an execution trace from [11]. Every line represents 1 timestep.

The task is to exactly reproduce the execution trace provided in the data. The evaluation metric is computed in the following way: execution traces are generated and scored for 500 test samples: 1 point if the produced sequence of function calls and arguments exactly match the true execution trace ($\mathbf{p}_\mathbf{0}^\mathbf{t} = O^t$, and so on for actions, subprograms and arguments), and 0 otherwise. The scores are then averaged, producing the final metric called zero-one accuracy.

3.2 NPI Implementation details

Section 2.3.2 describes a the core module of the NPI architecture but does not go into detail the task-specific parts of the model, the world and argument encoders. Here we briefly explain the architecture that was used to encode the Nanocraft world observations in [11], as our work is based on the very same architecture.

World encoder: Consider a single world observation with n D by D feature maps. For a given position in the grid we have n features, each is embedded into a 32-dimensional (learnable) embedding. We then concatenate these embeddings achieving $32n$ feature maps, each being a D by D grid. This representation is then fed through 2 convolutional layer and 2 fully connected MLP layers with ReLU activation.

Program and elementary operation representation: Program ID-s are embedded into a 32-dimensional vector.

Argument representation: There are a small amount of possible discrete arguments, and as such the 4 arguments are represented as 4 one-hot vectors for the 4 arguments, then concatenated together.

To jointly encode world observations , current program and arguments, the outputs of the above three encoders are concatenated and fed through a 2 layer fully connected network with ReLU units. For further parameters of the architecture, see the original paper ([11])

3.3 Natural Language Instructions

In order to study how the NPI model can be extended to natural language input, the original dataset was modified by replacing the parameters of the houses to be built by a number of english sentences specifying the properties of the house to be built. Due to the complexity of collecting input from humans and the lack of time to do this, we decided to programatically generate natural language like sentences based on the 6 parameters (positions, sizes, color, material) of the houses.

3.3.1 Sentence Generation

Given a house specification $\mathscr{H}_i$, we generate a sequence of integers representing words of the instructions $\{x^t_{nl,i}\}^T_{t=1}$ in the following way:

1. Each property in the specification is mapped to a string. Colors and Materials are mapped to the appropriate single words. Positions are expressed as *((lower | upper) (left | right)) | middle*. The width of the house can be expressed using one of the following terms: *"very wide", "wide", "average width", "narrow", "very narrow"*. Height is expressed in a similar manner.

2. For each property we randomly choose one from two substantially different ways of expressing the given property. For example, we can add "red" in front of the word "house" or "building", but alternatively we could also specify the color with an additional sentence: "Paint it red".

3. For some words appearing in the sentence we randomly choose from synonyms - we can use "construct" or "build" for example.

4. A template sentence is created based on the result of point 2. and is filled with the appropriate words based on points 1. and 3. This results in a natural-language like sequence of sentences.

5. Some sentences and adjectives are randomly shuffled so that the order in which different types of information appear is random.

6. The resulting text is split by spaces and resulting list of words is mapped to a corresponding sequence of integer codes. Shorter sequences are padded from the right such that all sentences will have the same length T.

To create the new training data we simply replace each of the house specifications $\mathscr{H}_i$ with its randomly generated natural language instruction set, $\{x^t_{nl,i}\}^T_{t=1}$. A random sample of sequences generated by this procedure is found below:

"Build a blue building from plastic. Make it very wide and short. It has to be in the upper left."

"Construct a plastic very wide and short house in the upper left. Paint it red."

"Build a glass blue building in the middle. Make it narrow and very tall."

The way we represent world states also had to be changed. We found that position, material and color of pre-built blocks carry significant information about the house to be built, and hence prevented us from accurately measuring the ability to understand natural language. In the final version of our work, the world state has only 2 feature maps. The one-hot player position map was unchanged, but instead of precisely representing the color and material of built blocks, we only used a binary map expressing whether there is a block at a given position or not. This map was initialised with equal amount of empty and non-empty blocks randomly, thus avoiding the leakage of information about the position argument. To summarise, the new world representation (as opposed to previous works) guarantees that the only flow of information is via the natural language instructions.

3.3.2 Some analysis

Although the programmatically generated sentences are considerably more predictable and less noisy than real life instructions, we believe they provide enough variety to test the performance of our models. A number of properties make them similar to human-produced instructions:

- Information can appear in different order and in different context.
- There are parts of the input where interpreting words in context is important, such as *"very wide"*. To get these cases right, the model has to learn the effect of "very" on other words.
- There are distractive words that have no impact on the outcome of the task and the model has to learn to ignore: *"build"*, *"a"*, *"the"*, "." etc.
- There are non-trivial connections between words and meanings: different words can refer to the same concept but in different context. For example *"wide"* and *"tall"* should both eventually be used to infer that a particular wall should be long, but it also should be learned that the two words refer to different walls.

We carried out a short experiment that counted how many ways a particular house configuration can be expressed: There are 148 possible different instructions for a specific set of parameters.

3.4 Different shapes

Note that in the above introduced dataset the parameters of the buildings only affected the arguments passed to higher level functions. The sequence of high-level function calls were identical for all samples: 2 calls to MoveMany, followed by 4 calls to BuildWall. A secondary aim of this work was to investigate whether deciding between different sequences of higher level function calls is possible using natural language instructions. In order to find this out, we added an additional parameter to the house specification: the shape of the house, which can be be one of *rectangle*, *trapezoid* or *triangle*. Parallel to this we changed the data generation pipeline by adding an additional function called BuildDiagonalWall, which takes the same parameters as BuildWall but builds walls diagonally. Using this function we modified the top-level sequence of function calls: building the rectangle is exactly the same process as previously but building a trapezoid involves alternating calls to BuildWall and BuildDiagonalWall and building a triangle requires two diagonal walls followed by two normal BuildWall-s. The grammar for natural language instructions was extended accordingly by adding an extra adjective, the shape. We refer to this version of the Nanocraft task as the multi-shape task.

3.5 World references

The third aim of this thesis was to explore how instructions containing references to the world state can be handled. Due to time constraints this idea was only briefly explored, using a rather primitive task. This new task had only a simple template sentence, "Move to the <color> <material> block". The initial world state included 5 pre-built blocks at different locations, each with a different pair of color and material properties, one of which was the one specified in the instruction. [1] The task was to move to the appropriate block by two calls to the MoveMany function. We refer to this task as the move task.

[1] Since pre-built blocks have great importance here, the old type of world representation from Section 3.1 is used again.

Chapter 4

Models

In this chapter we present and formalize the sequence of changes we made to the existing model in order to solve the tasks described in the previous chapter.

4.1 Simple encoder

As mentioned in Chapter 2, the most obvious way to encode a variable-length sequence of words (the instructions in natural language) is using some variant of recurrent neural networks. In our case this an implementation of LSTM-s was already present in the codebase, hance we started by testing a simple LSTM-based architecture. The main idea of this solution is to encode the input with an LSTM into a fixed-size vector, then use this vector to initialise the decoder LSTM instances of the NPI architecture - in a very similar mindset to what was described in Section 2.1.4, but utilising the hierarchy provided by the full execution traces.

Let x_{nl}^t for $t \in 1..T$ be the input sequence of words (that is, the instructions) represented as integers. We first construct a learnable embedding, $f_{nl\_emb}(\dot{})$ that assigns a 32 dimensional vector to each word. These embeddings are then fed into the encoder LSTM:

$$\mathbf{h_{in}^t} = f_{nl\_lstm}(\mathbf{h_{in}^{t-1}}, f_{nl\_emb}(x_{nl}^t))$$

This encoder LSTM is identical in architecture to the decoder LSTM which was already implemented: it has 2 layers and 128 units in both the cell and hidden states. Because of this, we can directly use the hidden and cell states in all layers of the encoder to initialise their counterparts in the decoder LSTM:

$$\mathbf{h_{out}^0} = \mathbf{h_{in}^T}$$

4.2 Attention

Section 2.2.1 describes a traditional attention mechanism that is widely used in recent literature. We adopted this approach for the NPI model in order to better predict which function or elementary operation to call and the 4 corresponding arguments. Figure 4.1 shows a diagram summarising how we extended the NPI model with a sequence of operations in order to incorporate attention. In this section we provide the details for each of those steps shown on the diagram and give a precise mathematical definition of our new model.

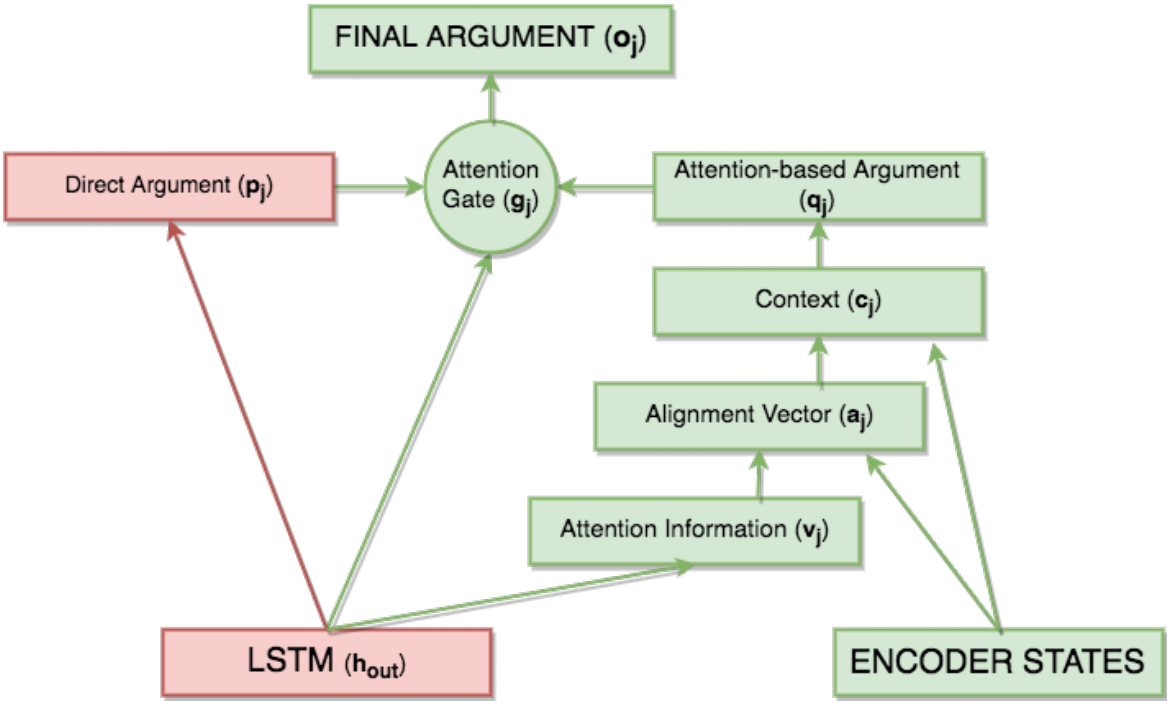


Fig. 4.1 An overview of the operations required to compute an argument (or program) output $\mathbf{o_j}$ at decoding time-step j. Green color shows pieces of the new architecture, red is the orignal NPI model

4.2.1 Computing the context vector

In our previous notation, $\mathbf{h_{in}^t}$ represented the ensemble of hidden states and cell states of the encoder LSTM in all 2 layers at time step t. To clarify notation, we will call this $\overrightarrow{\mathbf{h_{in}^t}}$, and $\overrightarrow{\mathbf{h_{in}^{t,2}}}$ represent the ensemble of the hidden state and the cell state of the final layer. In addition, we construct a backward encoder as described in Section 2.1.3:

$$\overleftarrow{\mathbf{h_{in}^t}} = f_{nl\_lstm\_bw}(\overleftarrow{\mathbf{h_{in}^{t+1}}}, f_{nl\_emb}(x_{nl}^t))$$

Inspired by [9], the input to the alignment algorithm is the concatenation of the top layer hidden and cell states of the forward and the backward encoder, moreover the embedding of the natural language input. Lets denote this input at time t by $\mathbf{u^t}$:

$$\mathbf{u^t} = \begin{bmatrix} f_{nl\_emb}(x^t_{nl}) \\ \overrightarrow{\mathbf{h^{t,2}_{in}}} \\ \overleftarrow{\mathbf{h^{t,2}_{in}}} \end{bmatrix}$$

For the purposes of this section assume that we want to compute some attention-based context vector that is later going to be used to produce a particular argument to use with the next function call or elementary operation at time j. For the sake of simplicity denote this by $\mathbf{c_j}$. An identical attention mechanism is used to compute other outputs: other arguments and the function/elementary operation to call.

At decoding time, in each time step t, a 32-dimensional "attention information" vector $\mathbf{v^t}$ is computed based on the current state of the decoder LSTM. The purpose of this vector is to encode what kind of information we are looking for in the input text. It is computed very similarly to other outputs:

$$\mathbf{v^k} = W^{info}_{r,1}\mathbf{h^j_{out}}$$

The alignment and the context vector is then computed analogously to what was described in Section 2.2.1:

$$\mathbf{c_j} = \sum_{i=1}^{T} \alpha_{ij}\mathbf{u^i} \tag{4.1}$$

$$\alpha_{ij} = \frac{exp(e_{ij})}{\sum_{k=1}^{T} exp(e_{kj})} \tag{4.2}$$

$$e_{ij} = a(\mathbf{u^j}, \mathbf{v^j}) \tag{4.3}$$

As opposed to the simple cosine-distance based aligner presented earlier, the aligner function a concatenates its two inputs and uses a two-layer fully connected neural network to compute the relevance number. This network uses tanh activation function, has a single hidden layer with 160 units (half as many as in the input layer), then a single output unit, e_{ij} The reason for choosing this type of aligner instead of the one using cosine-distance is simply that it performs better.

Recall that this mechanism is used 5 times in parallel in order to compute the 5 outputs: the function and the 4 argumens to use. For each of these 5 instances different weights

W^{info} are used to compute the "attention information", but the weights used in the aligner network are shared. The intuition behind this is that we force the aligner network to gain a general understanding of all the of information found in the instructions, which results in more general, rich representations.

4.2.2 Integrating the context vector

The computed context vector $\mathbf{c_j}$ is intended to carry information about the argument we are about to produce. Hence the first step of integrating this context vector into the NPI architecture is mapping it to an appropriate one-hot vector. This is done in the same fashion as the outputs are computed in the orignial architecture (Section 2.3.2), that is, using a fully connected layer with a softmax at the output. Naturally this function will be different for each of the the 5 kinds of outputs (function and the 4 arguments), but let's assume we want to compute what the attention-based method tells us about the frist argument at time j, call this $\mathbf{q^j_{r,1}}$:

$$\mathbf{q^j_{r,1}} = SoftMax(W^{attn}_{r,1}\mathbf{c_j})$$

A last, crucial observation is that attention can not always be used to calculate the outputs. As an example, the directions in which we move are not dependent on the instructions but the internal state of the interpreter LSTM. Motivated by this, we added a gating mechanism to combine outputs computed in the original NPI architecture and by the new attention-based mechanism. The gate is dentoed as $g^j_{r,1}$ and is a scalar number taking values between 0 and 1. It is computed using a fully connected layer with a sigmoid activation σ_{gate} from the hidden state of the decoder. Let's denote the final output as $\mathbf{o^t_{r,i}}$. The gating mechanism is defined as follows:

$$g^j_{r,1} = \sigma_{gate}(W^{gate}_{r,1}\mathbf{h^j_{out}}) \tag{4.4}$$

$$\mathbf{o^t_{r,i}} = g^j_{r,1} * \mathbf{q^j_{r,1}} + (1 - g^j_{r,1})\mathbf{p^t_{r,i}} \tag{4.5}$$

The changes described above maintain the differentiability of the network, that is, gradients can be propagated through all the bits of architecture we added to the original model.

4.3 Word Embeddings

In our experiments we use a very small training set, typically between 64 and 512 samples. This is a reasonable requirement for our model, as detailed data about execution of tasks

is very hard to find. However, when it comes to language modeling, there is an enormous corpus of text available. Justified by this, we decided to experiment with using the pre-trained word embeddings mentioned in Section 2.2.2 to initialise the word embeddings in the NPI model, $f_{nl\_emb}(\dot{})$. In this section we provide the details of how we implemented this.

A possible option would have been to directly use the pre-trained embeddings from [22] in place of $f_{nl\_emb}(\dot{})$. However, experimental results showed that this significantly reduce performance. Our hypothesis was that this was due to the largely increased number of parameters - the original embeddings are 300 dimensional as opposed to our earlier 32 dimensional embeddings.

To overcome this potential issue, we applied the following method. We first collect the set of words appearing in our training data, then map this set of words to its embeddings, resulting in a small set of 300-dimensional vectors. We train a version of Principal Component Analysis (PCA) model [17] on this set of vectors and use it to extract 32-dimensional vectors for each word. The embeddings in this 32 dimensional space will capture most of the variation in the higher dimensional space but allow us to keep the original number of parameters.

In our final model, $f_{nl\_emb}(\dot{})$ is initialised to the 32-dimensional embeddings. As opposed to the original model, we do not change these embeddings at training time, they remain at their original values throughout. We also experimented with allowing these embeddings to be finetuned, but this resulted in less stable models.

4.4 World references

The purpose of this last extension was to help the NPI model to understand references in the instructions to objects in the world. The architecture we used for this can be considered a special kind of attention mechanism that compares information from the encoded instructions with each position in the world to compute an "attention map". This map can be embedded and used as an input to the NPI decoding stage. In the following we formalise our final model.

Recall that the final hidden state of the top layer of our input encoder LSTM is denoted as $\mathbf{h}_{\mathbf{in}}^{\mathbf{T},\mathbf{2}}$. This can be considered as a fixed-length encoding of the instructions. Following the idea of "attention information" from the previous section, the encoded instruction is mapped to a vector $\mathbf{v}$ that is meant to represent information about references to the world in the instructions:

$$\mathbf{v} = W^{w.attn}\mathbf{h}_{\mathbf{in}}^{\mathbf{T},\mathbf{2}}$$

Recall that the world is represented as n pieces of d by d feature maps. Hence at any time t, a particular location with coordinates i, j has n features associated with it. We represent

these features as one-hot vectors and concatenate them, let's denote the resulting vector by $\mathbf{w}^{\mathbf{t}}_{\mathbf{i,j}}$. Computing the alignment matrix is analogous to the computation of alignment vectors.

$$\alpha^t_{ij} = \frac{exp(e^t_{ij})}{\sum_{k=1}^{d}\sum_{l=1}^{d} exp(e^t_{kl})} \tag{4.6}$$

$$e^t_{ij} = a(\mathbf{w}^{\mathbf{t}}_{\mathbf{i,j}}, \mathbf{v}) \tag{4.7}$$

We used the cosine distance for the aligner function $a()$. [1]. The final context vector $\mathbf{c^t}$ is computed as follows:

$$\mathbf{c^t} = \sum_{i=1}^{d}\sum_{j=1}^{d} C_{i,j}\alpha^t_{ij} \tag{4.8}$$

where C is a learnable 3D weight tensor over the positions of the world grid, with shape d by d by 32. [2] Finally, the context vector is appended to the encoded world state $\mathbf{u^t}$ (see Section 2.3.2) at each time step and fed into the decoder LSTM.

[1] Note that this requires the length of $\mathbf{v}$ to be the sum of the depths of the world feature maps

[2] Since the last dimension is 32, all $C_{i,j}$ and the final context vector $\mathbf{c^t}$ are 32-dimensional - this optimal value was empirically determined.

Chapter 5

Results

5.1 Attention over the instructions

5.1.1 Performance

This section describes our main results on the basic Nanocraft task with natural language (described in Section 3.3). As a baseline, we used an implementation of the Sequence-to-Sequence model [9] to NPI (Section 6.1.2) that uses an encoder-decoder model (Section 2.1.4) to translate between the input sequence of words and the output sequence of program calls. As this model does not make use of the information about higher-level subprograms, we expected it to do poorly. We then tested and compared 3 different types of models: We first took the Encoder-only model described in Section 4.1, then added the attention mechanism on top of this (Section 4.2). Finally, we tested the same setup but with the word embeddings fixed to the pre-trained ones (Section 4.3). This last model is our full, final model. Figure 5.1 summarises the performance of the above listed 4 models on the basic Nanocraft task [1]

The baseline Seq2Seq model does extremely poorly, it learns close to nothing on this range of sample sizes. We note that the same model with a simple template instruction "Build a <sizeX> by <sizeY> <color> house from <material> in the <position>" learns perfectly for 512 samples - this demonstrates the challenge presented by the programmatically generated NL instructions.

The basic encoder-only model does reasonably well and learns perfectly given enough data. As the encoder of this architecture is identical to the encoder of the Seq2Seq model, the difference in performance proves that the abstractions provided by the high-level functions in the execution trace make learning the language a lot easier.

[1] Note that the performance of some models is extremely sensitive to the random seed used for training. Appendix A describes the method we use to deal with this problem.

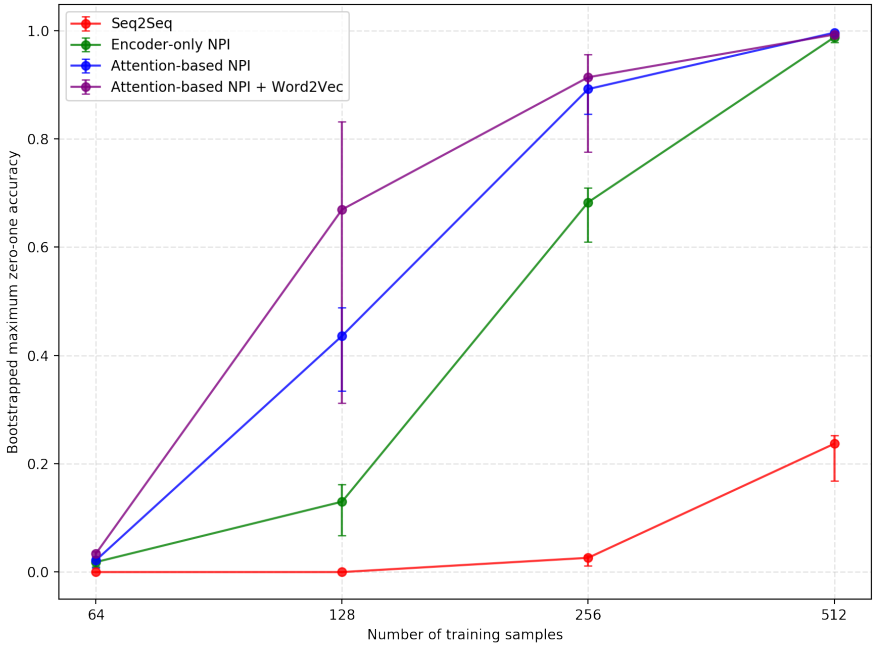


Fig. 5.1 Zero-one accuracies of the 4 main types of models on varying data sizes

Using attention brings further clear improvements in a specific range of sample sizes. At 128 samples, attention improves on the bootstrapped zero-one loss by 29%, and the non-overlapping confidence intervals provide clear statistical evidence that attention makes the model better. Further analysis on why this is the case is found in the next section.

Measuring the effect of the pre-trained Word2Vec embeddings is less obvious. While it is true that the mean bootstrapped zero-one loss is higher for the W2V version for all sample sizes, the error bars significantly overlap. This means that there is a weak indication that the pre-trained embeddings might potentially be useful, but the combination of extremely unstable training the limited amount of times we can train a model prevents us from drawing clear conclusions. We analyse the embeddings in more detail in the next section.

5.1.2 Some analysis

The architectures described in Section 4 were designed with particular intuitions in mind: With attention we hope that the model can locate relevant information in the instructions or in the world. Word embeddings were added with the expectation that they would inform our model about the "meaning" of particular words. In the previous section we gave quantitative

evidence for the usefulness of these extended models, this section is concerned with giving more of a qualitative analysis. By interpreting what is happening inside those models, we want to check whether our intuition about the roles of the added pieces of architectures was correct or not.

Alignment

To observe how the attention mechanism works, we inspected values of the alignment vector α for particular arguments of particular kind of function calls. This attention vector could be compared to the ground truth, that is, what a human would pay attention to in order to determine the argument needed for this particular argument.

To illustrate this, we evaluated a well-performing attention-based model on the instructions "Construct a green steel building in the lower right. make it narrow and very tall". The execution trace was correct, and contained 4 calls to the BuildWall function as expected - corresponding to the 4 walls of the building. Each time this function was called, we recorded the alignment vector used for the third argument and then plotted the values of attention over the input sequence on Figure 5.2.

Note that the third argument is for the length of the wall. Thus two times we expect the attention to look at the part of instructions specifying the height of the building, and two times to look at the width. We can confirm that this is indeed the case.

We chose this particular argument as it involves learning a more complex relationship than simply mapping single words to arguments such as in the case of colour or material. Note how the attention only points at a single position within "very tall". A likely explanation is that the backwards encoder LSTM remembers the necessary information from the word "tall", and hence its hidden state at word "very" has all the necessary information to infer the appropriate argument.

Attention gates

A core idea in the adaptation of attention to NPI-s is the gating mechanism that allows the model to learn distinguishing between outputs that benefit from attention and ones that do not. Staying with the example using BuildWall, we evaluated the same model on a large number of instructions, and noted down the values of the attention gates for the 4 arguments. The averages of these values for each of the arguments are shown in Table 5.1

The color, material and length arguments can all be directly inferred from text, but the last argument, direction depends on which stage of the execution we are at - hence this argument should intuitively be inferred from the state of the LSTM in the original way, and not using

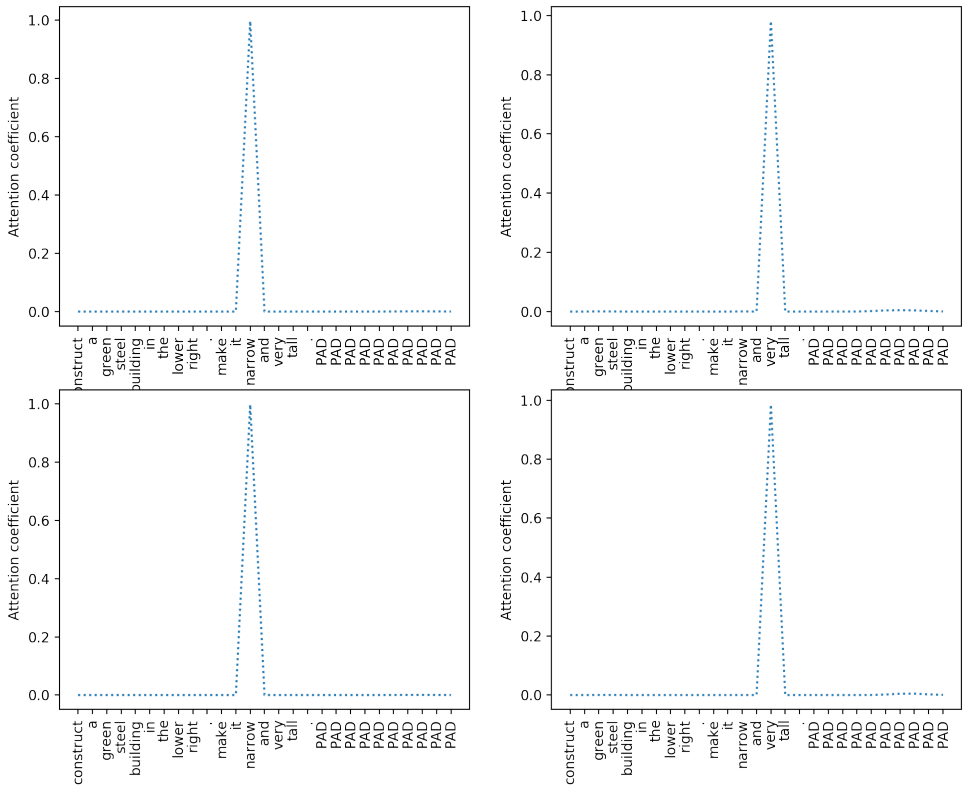


Fig. 5.2 Alignment vectors for the third argument of BuildWall. Each subplot represents one time the function was called)

attention. A clear contrast can be seen between the first 3 and the last argument, but it is not perfectly what we expect. The average for the third argument is slightly different from what we expect, but this is reasonable as this argument is particularly hard to learn (as described earlier). The last argument is, however, very far from the true value. To understand why this is the case, we plotted alignment vectors for this argument as well (using the same method as previously) for a single execution trace, this is shown on figure 5.3. In this plot the alpha (transparency) values of the lines represent the value of the attention-gate. These 4 values are shown in the legend.

We first note that in 2 cases the lines are absolutely not shown, the attention gates judged well that this argument does not need attention mechanism. In the other two cases, however, the value of the gate is close to 1 and the lines show reveal a clear pattern that explains this unexpected behaviour: They tend to point to locations in the text that are independent of the parameters of the house: dots, padding, words like "to". We observed similar behaviour

Position	Role	Expected	**Measured**
1	colour	1	1.000
2	material	1	1.000
3	length	1	0.885
4	direction	0	0.482

Table 5.1 Averaged activations of the attention gate for the four arguments of BuildWall

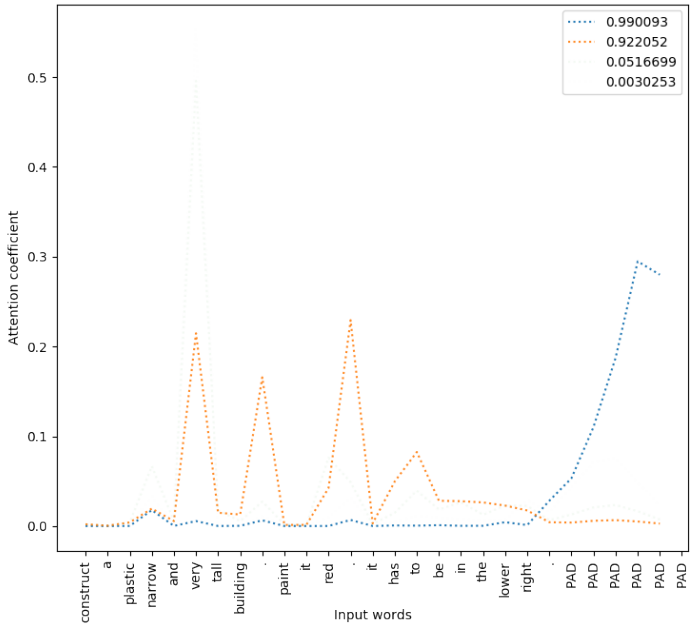


Fig. 5.3 Alignment vectors for the fourth argument of BuildWall

in many different cases: the blue line pointed at paddings and the yellow on dots and some other words. An interpretation of this phenomenon is that the model might be using these states as a constant value, and learns to pick the correct value for the argument based on these different constants.

We can conclude that our model learns to reproduce data perfectly, and most often it does it in a way we would expect it to do. However, in some cases the model learns an overly complicated, less intuitive, but correct way of deciding about the output.

Word embeddings

We experimented with 2 kinds of embeddings. The first approach was to let them be trained from a random initialisation along with the rest of the network, the second to use a fixed,

pre-trained Word2Vec embedding. Using the latter approach showed seemingly better but not statistically significant improvements. In order to better understand the differences between the two approach, we used PCA to map the embeddings into 3 dimensional spaces, then plotted the low-dimensional embeddings. Figure 5.4 visualises the result of training embeddings from scratch as part of our model, and Figure 5.5 shows a visualisation of the pre-trained Word2Vec model. Note that the most important 2 dimensions are represented by the x and y axis, and the 3rd dimension is represented by colouring the points based on some color-map.

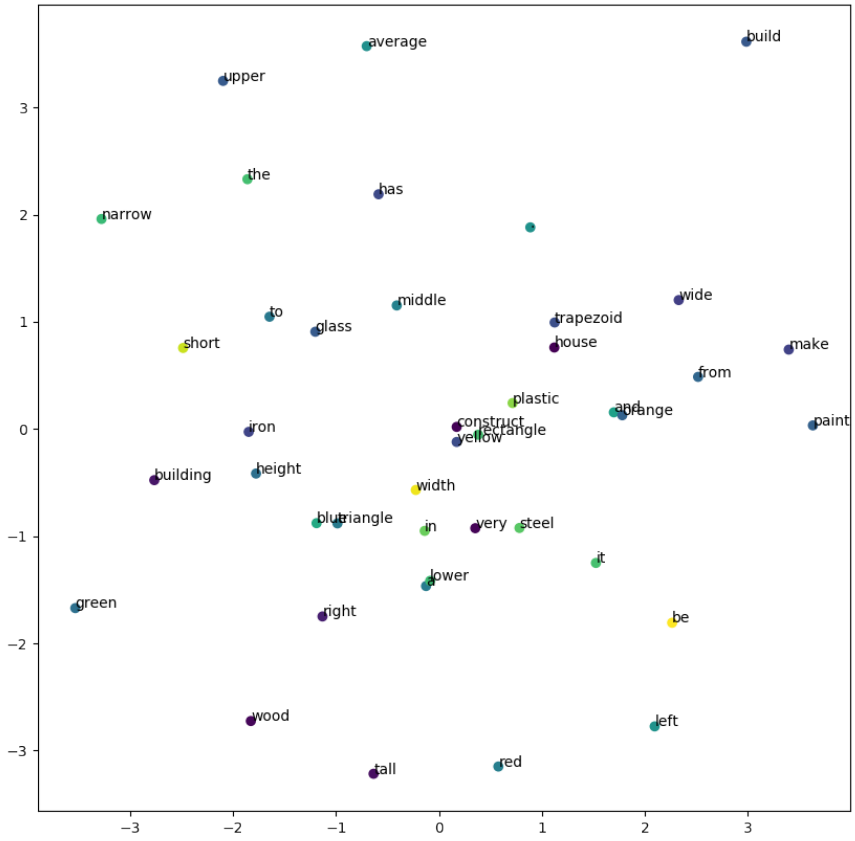


Fig. 5.4 Low-dimensional visualisation of the embeddings learned by an NPI model with attention on 512 samples

The difference between the two embeddings is remarkable. While the first approach results in a seemingly random cloud of words, the World2Vec embeddings clearly help to

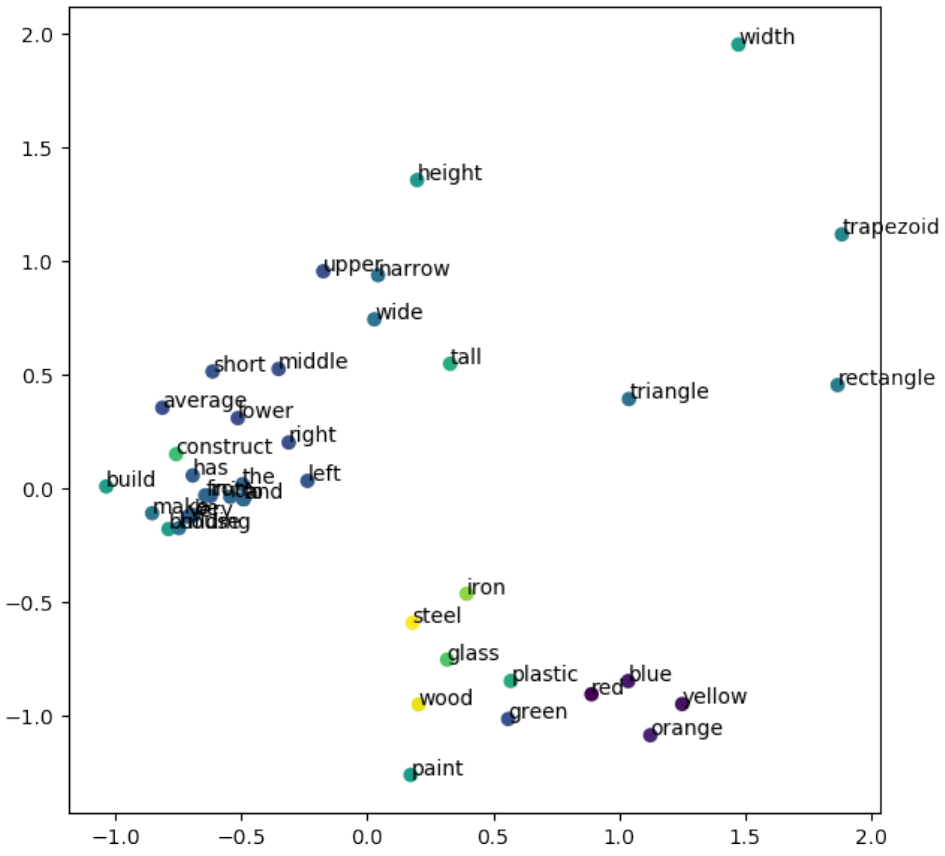


Fig. 5.5 Low-dimensional visualisation of the pre-trained Word2Vec embeddings

cluster and identify different kind of parameters such as colour, material, or shape. While it is not impossible that the embeddings from the first approach also have some hidden structure that this simple visualisation can not show, it is reasonable to say that the pre-trained Word2Vec model indeed conveys additional, useful information about the meaning of words, and has the potential to improve the overall performance.

5.2 Extension to multiple shapes

For the multi-shape task we evaluated two NPI models: the basic encoder-only and the attention-based one (without Word2Vec). The results are shown on Figure 5.6

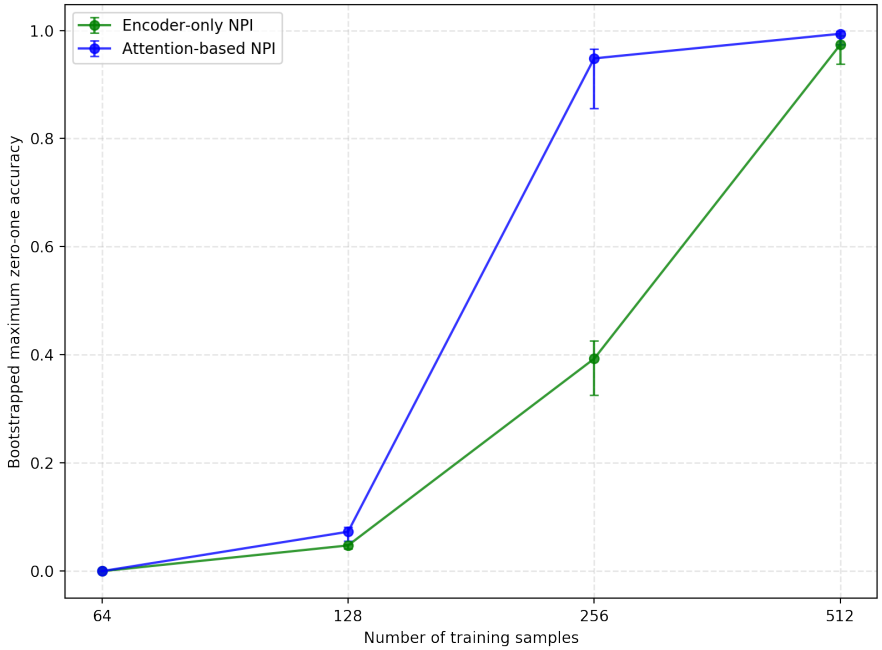


Fig. 5.6 Zero-one accuracies on the multi-shape task for the two major models

For smaller training sets (64, 128 samples) both models struggle to learn anything at all. This is not surprising as there are 3 different function sequences that the NPI model needs to learn instead of one. The real power of attention is demonstrated at 256 samples: The performance of the attention-extended NPI does not drop at all compared to the single-shape task (see Figure 5.1), however, the performance of the encoder-only NPI drops by almost 30% compared to the single-shape task. The gap between the two models is 56% for this sample size.

5.3 World references

In this experiment, we compared two models. The first model - used as a baseline - is the NPI model extended with the attention mechanism over the instructions (see Section 4.2). Note that we did not use the Word2Vec embeddings as we wanted to use models with less variation in the performance, such that the effect of world attention could be measured. The second model is the same architecture but further extended with the attention mechanism over the world grid (see Section 4.4). The task was the move task defined in Section 3.5. The results are shown on Figure 5.7.

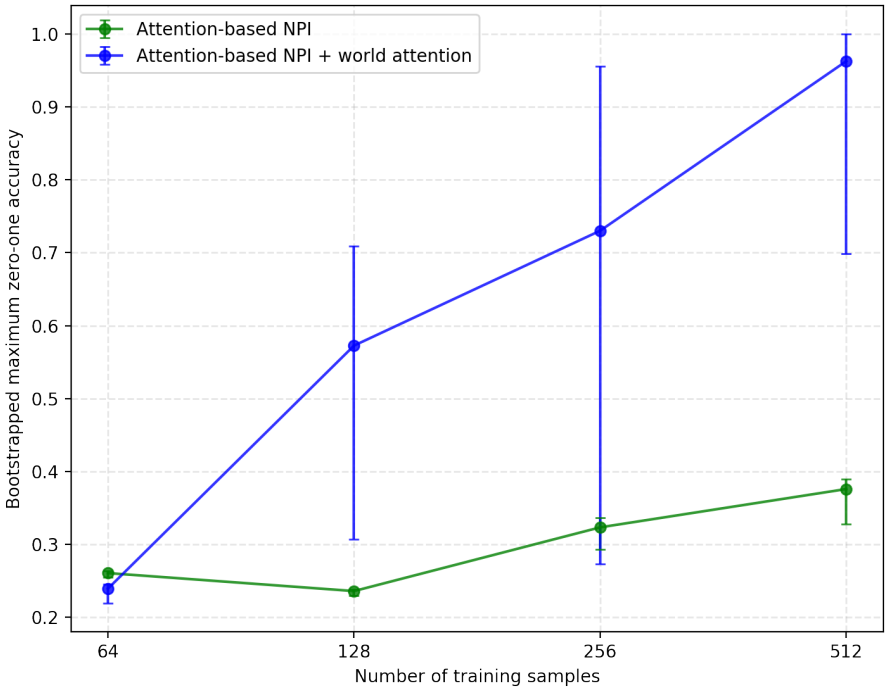


Fig. 5.7 Zero-one accuracies on the move task for the two main models

We conclude that adding attention over the world state allows the model to learn from the data a lot more efficiently - the slope of the blue line on the diagram is significantly higher. With 512 samples the difference significant, the bootstrapped accuracy of the world-attention model is 59% higher. A downside of the additional attention mechanism is that the final performance of a model becomes a lot more dependent on the random seed, and thus potentially many seeds have to be used to produce the best results.

Chapter 6

Related Work

6.1 Following Natural Language Instructions

6.1.1 Some history

[25] describes a navigational instructions dataset containing 768 pairs of free-form natural language instructions and corresponding sequences of actions. Both the instructions and executions were created by human participants of a navigational study. The environment is a simulated network of intersecting corridors where each corridor has a particular flooring and images of objects are hung on the walls. The instructions describe how to get from point A to point B and contain references to the environment. The task is to generate a sequence of actions given the natural language instructions.

The original work solves this problem using a complicated system that consists of six different modules out of which three are responsible for parsing the instructions linguistically and the other three interpret references to the spatial environment. There have been a large number of attempts to solve this navigational task, gradually improving the state-of-the-art performance on this task.

Earlier works relied explicitly on parsers that extracted the some sort of intermediate form from the instructions, and then learned to execute those forms. For example [26] infers a so-called navigation plan from the action sequence, including checks based on landmarks. Then it uses pairs of NL instructions and navigation plans to learn a semantic parser for the instruction sentences. [27] also considers this task as a parsing problem: the instructions are parsed into lambda calculus expressions which are then deterministically executed.

6.1.2 Moving away from parsers: Seq2Seq

[28] create a framework in which (for the first time) sequence-to-sequence models play a crucial role. Their model assumes structure in both the input sentences and the output action sequences, as well as access to different parsers that map the input and output into graphs that express their semantics. Their model learns to align and translate the structures in the input and output and hence is able to translate between natural language and action sequences. This model outperformed all the previously mentioned models.

The currently state-of-the art model ([9]) does not use any linguistic expert knowledge nor trains an explicit parser. Instead it uses a plain encoder-decoder model which encodes the input sentences into a fixed-length with a simple encoder LSTM. Then with the help of a multi-lever aligner (see Section 2.2.1) it uses a decoder LSTM that takes the world state and produces an action at each time step. This model, called the sequence-to-sequence, or shortly Seq2Seq model outperforms all previous models despite its relative simplicity compared to the previous works. This model was used as a baseline in our work.

6.1.3 More realistic environments

The environment in the corridor navigation problem is a rather simple model of real world scenarios. There has also been work ([29], [30]) on trying to interpret natural language instructions given some vision-like observations of geometrically complex 2D maps.

[8] uses a similar task where the world observations are 3D rendered images of objects, provided to the model in raw pixel form. Their proposed solution - a purely neural architecture - is notable because it is not making use of any prior linguistic or perceptual knowledge. This makes is similar to the methods described in the previous section and also our current work. The input sentences are encoded with a Gated Recurrent Unit (GRU) network [31], a gated RRN somewhat similar to LSTM-s. The world observations are encoded using Convolutional Neural Nets (CNN-s) [32].

The two kinds of inputs are fused into a single state which is then used by a reinforcement learning algorithm. In order to get this single state, first a d-long attention vector is formed from the final hidden state of the input language encoder, where d is the depth of the output of the world-encoder CNN. Using this attention vector over different feature maps of the encoded world as described in 2.2.1, they essentially gate different features of the world based on the instructions - hence the name Gated-Attention.

An exciting and relatively new task is proposed and solved in [4]. The dataset is called verb-environment-instruction-library (VEIL) and consists of pairs of natural language instructions for tasks in a kitchen and corresponding action sequences that could be performed by a

robot in order to execute the instructions. Instructions can come with various level of detail - there might be an instruction just saying "heat water in the pot", or an other one detailing the steps of this :"take the pot, fill it with water, ...". Furthermore, the action sequence is largely dependent on the observed world state. Along with the position of an object, the state of an object also plays a crucial role: is the stove on? Is there already water in the pot?. While somewhat more complex, this task is definitely the most similar one to the task we are trying to solve in this work.

In order to solve this problem they parse the input sentences into so-called verb-clauses: tuples of verbs, objects on which they act and relationship matrices. Similarly the world is represented in a semantically rich form, graphs. They use Conditional Random Fields in order to find the most probable action in each time step of the execution.

6.2 Neural Program Lattices

While the NPI model has performed well in a number of different algorithmic tasks, it was based on a kind of training data which might be very hard to get hold of. Having full execution traces is not very realistic in real-life scenarios as often only the sequence of elementary operations can be observed, while the structure is hidden. As an example, we can observe the movement of a human body (elementary operations), but might not be able to tell what the person was thinking (hidden structure) at the same time.

This problem was addressed by Neural Program Lattices or NPL-s [11]. By incorporating some novel ideas into the NPI model, NPL-s are able to consider a variety of possible latent call-structures corresponding to the observed sequence of elementary operations, and compute the marginal probability of the elementary operation sequence in the training data. The training of this model consists of two stages. In the first stage a small number of full execution traces are provided and the training procedure is effectively the same as for NPI-s. However, in the second stage data is provided without the call-structure - the NPL model is able to significantly improve using only this semi-supervised data.

While our work was based on the NPI architecture, it can be trivially extended to NPL-s as well. Doing so would take the current work a step closer to a model that translates natural language instructions into elementary sequence operations. This ability would be desirable as it would allow us to directly compete with the Seq2Seq model without the addition of the program call hierarchy.

Chapter 7

Conclusion

This work has explored the idea of extending the Neural Programmer-Interpreter model with modified versions of well-known NLP techniques. The resulting, novel class of models was evaluated on tasks where instructions given in natural language had to be carried out. Our contributions can be summarised in the following points:

- We created a new version of the Nanocraft task where instructions consist of programmatically generated English sentences. We extended this task by adding different possible shapes and finally created a new task to examine instructions with references to the world state.
- We proposed two novel attention-based architectures for these tasks that outperformed the best two currently available baseline models:
  - Attention over the instructions
  - Attention over the world observations
- Finally we carried out some experiments that provided insight into the inner workings of the models.

While this work has demonstrated the feasibility of combining the NLP methods with the NPI architecture, the journey is far from over. We propose extensions to our work in two categories. The first type of future work involves evaluating and possibly adopting our models to more realistic and thus harder tasks. This would be particularly useful as our current work only confirmed the ability of our models to learn particular tasks, but have not explored the limits of our models. In particular, we suggest evaluating our models in the following ways:

- The programmatically generated natural language should be replaced with instructions collected from real human users. As a first step it would be interesting to see how the currently trained models generalise to real NL, but ultimately the aim should be to train and evaluate the current models on real, human generated data .
- Inspired by real-world applications, noise (incorrect execution steps) could also be added to the output in the training data. It would be interesting to see if the model could still learn to carry out instructions correctly.
- The original Nanocraft task could be combined with world references, resulting in complex instructions like "Build a red house below the plastic block". A purely neural network based model able to understand these complex sentences would be an exciting competitor to the parser-based models

Finally, we suggest a number of ways in which the models developed in this work could be further improved:

- The current architecture could be easily adopted to NPL instead of NPI, thus allowing the model to learn from pure action sequences, making it a closer competitor to the Seq2Seq model.
- Different word embeddings could be used and evaluated instead of the current pre-trained Word2Vec. This could potentially seriously effect the performance on noisy, human-generated instructions.
- The attention mechanism used for world-references is rather just a prototype than a well thought out model - we believe that further improvements to this architecture can be made that would improve its ability to handle complex instructions.

References

[1] Andrew Ng, "Machine Learning | Coursera." [Online]. Available: https://www.coursera.org/learn/machine-learning

[2] M. Macmahon, M. Macmahon, B. Stankiewicz, and B. Kuipers, "Walk the Talk: Connecting Language, Knowledge, Action in Route Instructions," *IN PROC. OF THE NAT. CONF. ON ARTIFICIAL INTELLIGENCE (AAAI*, pp. 1475–1482, 2006. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.60.9798

[3] S. Gulwani and M. Marron, "NLyze: Interactive Programming by Natural Language for SpreadSheet Data Analysis and Manipulation," 2014. [Online]. Available: http://dx.doi.org/10.1145/2588555.2612177.

[4] D. K. Misra, J. Sung, K. Lee, and A. Saxena, "Tell me Dave: Context-sensitive grounding of natural language to manipulation instructions," *The International Journal of Robotics Research*, vol. 35, no. 1-3, pp. 281–300, 1 2016. [Online]. Available: http://journals.sagepub.com/doi/10.1177/0278364915602060

[5] L. Dong and M. Lapata, "Language to Logical Form with Neural Attention," 1 2016. [Online]. Available: http://arxiv.org/abs/1601.01280

[6] Y. Artzi and L. Zettlemoyer, "Weakly Supervised Learning of Semantic Parsers for Mapping Instructions to Actions," *Transactions of the Association for Computational Linguistics*, vol. 1, no. 0, pp. 49–62, 3 2013. [Online]. Available: https://transacl.org/ojs/index.php/tacl/article/view/27

[7] C. Quirk, R. Mooney, and M. Galley, "Language to Code: Learning Semantic Parsers for If-This-Then-That Recipes," in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Stroudsburg, PA, USA: Association for Computational Linguistics, 2015, pp. 878–888. [Online]. Available: http://aclweb.org/anthology/P15-1085

[8] D. S. Chaplot, K. M. Sathyendra, R. K. Pasumarthi, D. Rajagopal, and R. Salakhutdinov, "Gated-Attention Architectures for Task-Oriented Language Grounding," 6 2017. [Online]. Available: http://arxiv.org/abs/1706.07230

[9] H. Mei, M. Bansal, and M. R. Walter, "Listen, Attend, and Walk: Neural Mapping of Navigational Instructions to Action Sequences," 6 2015. [Online]. Available: http://arxiv.org/abs/1506.04089

[10] A. Graves, "Neural Turing Machines arXiv : 1410 . 5401v2 [cs . NE] 10 Dec 2014," pp. 1–26.

[11] D. Tarlow, A. L. Gaunt, M. Brockschmidt, and N. Kushman, "Nerural Program Lattices," pp. 1–17, 2017.

[12] D. Bahdanau, K. Cho, and Y. Bengio, "Neural Machine Translation by Jointly Learning to Align and Translate," 9 2014. [Online]. Available: http://arxiv.org/abs/1409.0473

[13] J. Elman, "Finding structure in time," *Cognitive Science*, vol. 14, no. 2, pp. 179–211, 6 1990. [Online]. Available: http://doi.wiley.com/10.1016/0364-0213(90)90002-E

[14] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to Sequence Learning with Neural Networks," 9 2014. [Online]. Available: http://arxiv.org/abs/1409.3215

[15] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 11 1997. [Online]. Available: http://www.mitpressjournals.org/doi/10.1162/neco.1997.9.8.1735

[16] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin, "A Neural Probabilistic Language Model," *Journal of Machine Learning Research*, vol. 3, no. Feb, pp. 1137–1155, 2003. [Online]. Available: http://www.jmlr.org/papers/v3/bengio03a.html

[17] M. E. Tipping and C. M. Bishop, "Mixtures of Probabilistic Principal Component Analysers," *Neural Computation*, vol. 11, no. 2, pp. 443–482. [Online]. Available: http://www.miketipping.com/papers.htm

[18] L. v. d. Maaten and G. Hinton, "Visualizing Data using t-SNE," *Journal of Machine Learning Research*, vol. 9, no. Nov, pp. 2579–2605, 2008. [Online]. Available: http://www.jmlr.org/papers/v9/vandermaaten08a.html

[19] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient Estimation of Word Representations in Vector Space," 1 2013. [Online]. Available: http://arxiv.org/abs/1301.3781

[20] J. Pennington, R. Socher, and C. Manning, "Glove: Global vectors for word representation." *EMNLP*, 2014. [Online]. Available: http://llcao.net/cu-deeplearning15/presentation/nn-pres.pdf

[21] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching Word Vectors with Subword Information," 7 2016. [Online]. Available: http://arxiv.org/abs/1607.04606

[22] "Google Code Archive - Long-term storage for Google Code Project Hosting." [Online]. Available: https://code.google.com/archive/p/word2vec/

[23] H. Siegelmann and E. Sontag, "On the Computational Power of Neural Nets," *Journal of Computer and System Sciences*, vol. 50, no. 1, pp. 132–150, 2 1995. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0022000085710136

[24] S. Reed and N. de Freitas, "Neural Programmer-Interpreters," 11 2015. [Online]. Available: http://arxiv.org/abs/1511.06279

[25] M. Macmahon, M. Macmahon, B. Stankiewicz, and B. Kuipers, "Walk the Talk: Connecting Language, Knowledge, Action in Route Instructions," *IN PROC. OF THE NAT. CONF. ON ARTIFICIAL INTELLIGENCE (AAAI*, pp. 1475–1482, 2006. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.60.9798

[26] D. L. Chen and R. J. Mooney, "Learning to interpret natural language navigation instructions from observations," pp. 859–865, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=2900560

[27] Y. Artzi, D. Das, and S. Petrov, "Learning Compact Lexicons for CCG Semantic Parsing." *EMNLP*, 2014. [Online]. Available: https://pdfs.semanticscholar.org/5521/093f13fd041277705f52b34434237a1e7263.pdf

[28] J. Andreas and D. Klein, "Alignment-based compositional semantics for instruction following," 8 2015. [Online]. Available: http://arxiv.org/abs/1508.06491

[29] C. Matuszek, D. Fox, and K. Koscher, "Following directions using statistical machine translation," in *2010 5th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*. IEEE, 3 2010, pp. 251–258. [Online]. Available: http://ieeexplore.ieee.org/document/5453189/

[30] T. Kollar, S. Tellex, D. Roy, and N. Roy, "Toward understanding natural language directions," in *2010 5th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*. IEEE, 3 2010, pp. 259–266. [Online]. Available: http://ieeexplore.ieee.org/document/5453186/

[31] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling," 12 2014. [Online]. Available: http://arxiv.org/abs/1412.3555

[32] Y. LeCun and Y. Bengio, "Convolutional networks for images, speech, and time series," *The handbook of brain theory and neural*, 1995. [Online]. Available: https://www.researchgate.net/profile/Yann_Lecun/publication/2453996_Convolutional_Networks_for_Images_Speech_and_Time-Series/links/0deec519dfa2325502000000.pdf

[33] B. Efron and R. Tibshirani, *An introduction to the bootstrap*, 1994. [Online]. Available: https://books.google.co.uk/books?hl=en&lr=&id=gLlpIUxRntoC&oi=fnd&pg=PR14&dq=bootstrap&ots=A9utS6NcG8&sig=C51qeLkePbs9yS7Nv0EQt89Xe4g

Appendix A

Evaluation metric

Recall that the metric we use to measure performance of our models is the zero-one loss (see Section 3.1). Due to the rather complicated architecture of our model, the zero-one loss of a particular model depends largely on the random seed used for training. A sensible way of evaluating these unstable models is to train many models with different seeds, pick the one with the minimum validation error and report the error of this model on the test set.

A second problem is that these models take a relatively long time (approx. 20 hours) to train, even when using 5 CPU-s of a machine. Given our limited resources this means that we can not train models with different seeds more than a few times - to generate our results, we used 12 different seeds for each data point. Due to the some models being rather unstable, choosing the minimum over 12 attempts still can't be considered to be robust metric. To have a better understanding of the relative performance of our models, we used a method called bootstrapping [33], which we briefly describe in the following.

Suppose we have n samples from some distribution and a test statistic (test error on min. validation error model in our case) on these samples. We randomly sample n times from the n samples **with** replacement, and compute the given statistic on that sample. We repeat this experiment k times, and note down the result every time. As a result of this we get k numbers: we can take the mean of this to get a relatively robust estimate of our statistic (we call this the bootstrapped zero-one loss), moreover we can take percentiles of this data to get confidence intervals. For our plots we used $n = 12$, $k = 10000$ and took the 5% and 95% percentiles of the k numbers, giving us 90% confidence intervals (Intuitively, 90 out of 100 times our results were between those two numbers)