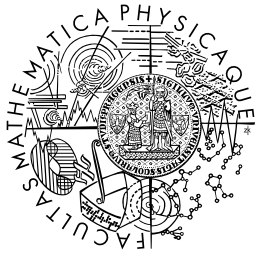

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

**MASTER THESIS**

Bedřich Pišl

# Natural language communication with Robots

Institute of Formal and Applied Linguistics

| | |
|---|---|
| Supervisor of the master thesis: | RNDr. David Mareček, Ph.D. |
| Study programme: | Informatics |
| Study branch: | Artificial Intelligence |

Prague 2017

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date .....  signature of the author

Title: Natural language communication with Robots

Author: Bedřich Pišl

Institute: Institute of Formal and Applied Linguistics

Supervisor: RNDr. David Mareček, Ph.D., Institute of Formal and Applied Linguistics

Abstract: Interpreting natural language actions in a simulated world is the first step towards robots controlled by natural language commands. In this work we present several models for interpreting unrestricted natural language commands in a simple block world. We present and compare rule-based models and recurrent neural network models of various architectures. We also discuss strategies to deal with errors in natural language data and compare them. On the Language Grounding dataset, our models outperform the previous state-of-the-art results in both source and location prediction reaching source accuracy 98.8% and average distance 0.71 between the correct and predicted location.

Keywords: neural networks natural language processing recurrent neural networks language grounding

I would like to thank to my supervisor, David Mareček, for the guidance and advice he gave me. Also I would like to thank to the Institute of Formal and Applied Linguistics for letting me use their cluster.

# Contents

# Introduction

The field of robotics is evolving. With more and more advanced robots, the question how to effectively control them and communicate with them is getting more important. For humans the most natural way of controlling robots is using the natural language. This opens up an important and interesting problem: creating robots who understand and follow natural language commands.

The first step to solve this task is being able to control a robot using natural language in a simple simulated environment.

This thesis describes a neural network approach for interpreting natural language commands in such a simple simulated environment. Our environment consists of a board with square blocks, with commands describing how these blocks should be moved. The set of commands is static and also contains a formal description of the correct action. Thus the problem can be viewed as a supervised machine learning problem with a static dataset.

There was already an attempt to solve this problem by Bisk et al. [2016b]. The main goal of this work is getting a better results than the ones described in Bisk et al. [2016b]. To do this we are going to use neural networks.

The second goal of this work is to create a rule-based model using language parsing and compare its results with the neural networks solution. This can open a ways for novel solutions.

Our third goal is to create an interactive demo for visualizing the environment and the actions.

This thesis is divided into 7 chapters. In the first chapter "Related work" we describe the work of Bisk et al. [2016b] and shortly discuss other similar datasets and corresponding solutions.

The second chapter "Theoretical background" introduces the reader to neural networks with focus on recurrent networks and methods used in our models.

The third chapter "Data and evaluation" describes the dataset used in this work and how we are measuring performance of predictive models in our environment.

The fourth chapter "Tokenization" describes the process of identifying words in the commands, which is a necessary preprocessing step before using the models.

The fifth chapter "Models development" discusses rule-based models, neural network models and hyperparameter values of our models.

The sixth chapter "Advanced data preprocessing" describes data augmentation, correcting errors in the commands and other approaches of improving the data or getting additional data.

The last seventh chapter "Evaluation and discussion" discusses our results and analyzes our best models.

# 1. Related work

Bisk et al. [2016a] created a dataset for translating commands in natural language to actions in a simulated world with square blocks. The block have logos or digits on them for an easy identification. The world is represented by a grid, and the actions are movements of blocks to some location.

In another article Bisk et al. [2016b] described a few neural models for understanding the commands in the dataset and predicting the actions. They found out that recurrent neural networks work better than feed forward neural networks for this task. They also tried three different architectures of the networks. The first one consisted of three classification networks, which predicted which block should be moved, a reference (the block closest to the target destination) and a direction to the target destination from the reference. Based on this triple they then computed the final predicted action. The second model learned to compute the final location based on the triple predicted by the first model. The last model was a regression neural model, which directly computed the coordinates of the block which should be moved and where it should be moved. With the first model they reached their best results - 98% accuracy when predicting the source block and 0.98 distance between the gold and the predicted location.

Robot Commands Treebank is another dataset with similar commands, which was created by Dukes [2013]. It also contains instructions for a virtual robot in a simulated world, which here consists of a grid and blocks of different shapes and colors. But in this dataset the target value is not a single action but a command or a sequence of commands in a formal Robot Control Language. Dukes [2014] compared six systems designed for solving this task. The systems are mainly using semantic parsers and hand designed rules, none of the systems uses neural networks. The best solution presented by Packard [2014] is based on the English Resource Grammar (Flickinger [2000]), whose results are than converted to the Robot Control Language by a manually created rule based system.

A similar dataset was created by MacMahon et al. [2006]. The task here is to predict a sequence of actions which will navigate a virtual robot to the correct location in a simulated world. As far as we know, nobody used neural networks for this dataset, the solutions are mainly based on semantic parsers, for example Artzi and Zettlemoyer [2013]. Another similar dataset and solution based on bigrams was described by Han and Schlangen [2017].

A similar system used in the real world was described by Tellex et al. [2011] and Walter et al. [2015]. They created a robotic forklift which should be able to understand simple natural language commands. For creating the system for understanding the commands, they first created a small dataset by using Amazons Mechanical Turk and manually annotated the data. Then they used a model invented by them specifically for this task, which is based on probabilistic graphical models.

To our knowledge, the only application of neural networks in similar problems was the one described by Bisk et al. [2016b]. We find this surprising, because neural networks are the most promising approach in many NLP tasks. One of the reasons likely lies in the sizes of the datasets - the one created by Bisk is much bigger then the other ones.

# 2. Theoretical background

This chapter gives a general introduction to artificial neural networks with focus on recurrent neural networks and techniques used in following chapters.

## 2.1 Neuron

Artificial neural networks are a widely used family of machine learning models inspired by the human brain. The basic building blocks of neural networks are neurons. Each neuron computes a simple function of its inputs based on an activation function, threshold and parameters which are called weights. The number of weights is the same as the number of inputs of the neuron. The output is then the result of the activation function applied on a dot product of the weights and the inputs.

More formally let us denote the inputs of a neuron $x_1, x_2, ..., x_n = \vec{x}$, the weights $w_1, w_2, ..., w_n = \vec{w}$, the threshold $\theta$ and the activation function $f$. The output of the neuron $y$ is then computed as:

$$y = f(\vec{x} \cdot \vec{w} - \theta) \tag{2.1}$$

The function $f$ is typically the sigmoid function:

$$f(z) = \frac{1}{1 + e^{-\lambda z}} \tag{2.2}$$

or the ReLU function:

$$f(z) = \begin{cases} z, & \text{if } z \geq 0 \\ 0, & \text{otherwise} \end{cases} \tag{2.3}$$

## 2.2 Perceptron

The simplest neural model, which has only a single neuron, is called perceptron. It was first described by Rosenblatt [1957]. It classifies inputs into two classes with an activation function:

$$f(z) = \begin{cases} 1, & \text{if } z \geq 0 \\ 0, & \text{otherwise} \end{cases} \tag{2.4}$$

The learning process is described in Algorithm 1.

The perceptron is seldom used in practice. One of the reasons is that it is a linear classifier and it searches for a hyperplane dividing the training instances into two classes. However, in many cases, the classes are not linearly separable, the most trivial example is when the features are inputs of a XOR function and the classes are its outputs.

**Algorithm 1** Perceptron learning algorithm. $D$ is a set of training examples, which are feature vectors of length $l$ together with labels

```
1: procedure learn(Training set D, Iterations n, Number of features l)
2:     Randomly initialize weights w_1, ..., w_l = w and threshold θ
3:     for k in [1, ..., n] do                          ▷ For each interation
4:         for (x, y) in D do                ▷ For each feature vector X and label y
5:             o ← f(x · w − θ)                                  ▷ Compute output
6:             for i in [1, ..., l] do
7:                 w_i ← w_i + (y − o) · x_i                     ▷ Update all weights
8:             end for
9:             θ ← θ + (y − o)                                   ▷ Update threshold
10:        end for
11:    end for
12: end procedure
```

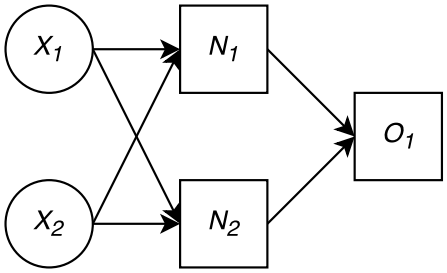


Figure 2.1: A simple multilayer perceptron with two inputs $X_1, X_2$, two hidden neurons $H_1, H_2$ and a single output neuron $O_1$.

## 2.3 Multilayer perceptron

To overcome the limitation of a single perceptron we can create a network consisting of layers of neurons. Such a network works in following way. The inputs of the network are first transformed by the first layer and the results of this transformation are used as inputs of the second layer. This happens until the last layer is reached, which produces the prediction.

The predictions of a multilayer perceptron are not limited to 0 or 1 as it was for the perceptron. Instead they are vectors of real numbers with same dimension as the number of neurons in the last (output) layer. Predictions of the network can and often are further transformed. For example for a classification to $n$ classes, we predict $n$ real numbers, where $i$-th is interpreted as the probability that the instance belongs to the $i$-th class. Then we choose the class with the highest probability.

The major learning method of a multilayer perceptron is the backpropagation algorithm together with some optimization algorithm and a loss function. Sometimes these three entities are described together as the backpropagation algorithm, but we will discuss them separately.

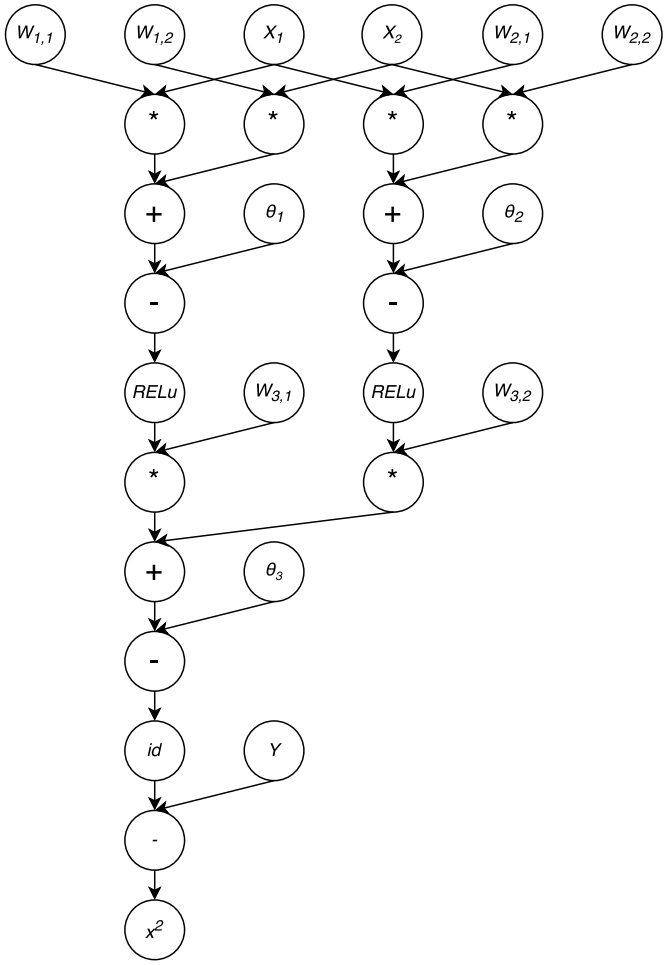


Figure 2.2: A computational graph of the multilayer perceptron showed in Figure 2.2. The network has two inputs, two neurons in the hidden layer with a RELu activation function, one output neuron with a linear activation function and a mean square error loss. The input of the network has features $X_1, X_2$ and a target value $Y$, the $j$-th weight of the $i$-th neuron is $W_{i,j}$, the threshold of the $i$-th neuron is $\theta_i$, $id$ denotes an identity function and $x^2$ denotes a square function.

### 2.3.1 Loss function

In supervised training, we have a correct target for each instance. As was described in the Section 2.3, the neural network produces a vector of real numbers. A loss function takes the vector of real numbers and the correct target and outputs a single real number - the loss. As in the other areas, the loss is a measure of how bad the prediction is and during training we are trying to minimize it.

Often used loss function is the mean square error function (MSE) and the cross entropy loss (Golik et al. [2013]).

### 2.3.2 Backpropagation algorithm

The backpropagation algorithm is the major learning algorithm used not only in multilayer perceptron networks, but almost in every kind of neural network. Therefore we describe it using not a multilayer perceptron, but a more general structure - a computational graph.

A computational graph is a directed acyclic graph, where the nodes with $n \geq 1$ parents contain an $n$-ary function and nodes with 0 parents contain a constant (which can be viewed as a constant function). For simplicity, we assume all the functions use real numbers: $R^n \mapsto R$. In most implementations there are tensors of real numbers instead. A computational graph for the network depicted in Figure 2.2 is in Figure 2.2.

In a computational graph, an input nodes are the nodes without any parents and the loss node is the node without any children. We assume that the graph has only one loss node, which computes the value of the loss function.

The backpropagation algorithm computes the derivatives of the loss function with respect to the network parameters. To be able to do this, the functions in all the nodes must be differentiable.

Since the computational graph is directed and acyclic there is a topological ordering of its nodes. Therefore all the nodes can be numbered $1, ..., n$ such that there are no edges $uv$ with $u > v$. We use this numbering of nodes in the following text. Note that the last $n$-th node is the loss node.

**Algorithm 2** Forward phase

1: **procedure** $forward$(Computational graph $G$)
2: **for** $i = 1, ..., n$ **do** ▷ In topological order
3: $p_1, ..., p_j \leftarrow$ Parents of $i$
4: $o_i = f_i(o_{p_1}, ..., o_{p_j})$ ▷ Compute output value of $i$-th node
5: **end for**
6: **return** $(o_1, ..., o_n)$
7: **end procedure**

**Algorithm 3** Backward phase

1: **procedure** *backward*(Computational graph $G$, Node outputs $(o_1, ..., o_n)$)
2: $g_n \leftarrow 1$
3: **for** $i = n - 1, ..., 1$ **do** ▷ In reversed topological order
4: $g_i \leftarrow \sum_{j \in Child(i)} g_j \cdot \frac{\partial o_j}{\partial o_i}$ ▷ $g_i$ is equal $\frac{\partial o_n}{\partial o_i} = \frac{\partial loss}{\partial o_i}$
5: **end for**
6: **return** $(g_{j_1}, ..., g_{j_k})$ where $j_1, ..., j_k$ are nodes corresponding to network parameters
7: **end procedure**

The algorithm works in two phases: the forward and the backward phase, described as Algorithms 2 and 3. In the forward phase the outputs of all the nodes are numerically evaluated, starting from the input nodes. Then in the backward phase, the derivatives with respect to the network parameters are computed. This is done in the other direction, starting from the last node back to the input nodes.

The most important part of both algorithms is on the line 4 of Algorithm 3. The analytical formula for the expression $\frac{\partial o_j}{\partial o_i}$ is found by some automatic differentiation system or by hand, before the the backpropagation algorithm starts. It is a function of $o_i$. Then during the run of the backpropagation algorithm, this expression is numerically evaluated by the assignment of $o_i$ to this function.

It holds that $g_i = \frac{\partial loss}{\partial o_i}$. For $i = n$ it holds because for each $x$ $\frac{\partial x}{\partial x} = 1$. For $i < n$ it can be derived by using the chain rule of calculus.

### 2.3.3 Optimizer

The optimizer in a neural network is trying to find the minimum of the loss function by changing the parameters of the network. Basically it uses the steepest descent method - it is decreasing the weights by their gradient and therefore moving in the direction of the steepest descent of the loss function.

Most often used optimizers use only the derivatives computed by the back-propagation algorithm, but some optimizers use also additional information for updating the weights such as the second derivatives.

The basic algorithm is called stochastic gradient descent, which is described as Algorithm 4.

**Algorithm 4** One epoch of the stochastic gradient descent. For simplicity we assume that the number of training examples is divisible by the batch size.

**procedure** $SGD$(Computational graph $G$, Training data $T$, Batch size $b$, Learning rate $\alpha$)
  $i \leftarrow 0$ ▷ Training data index
  **while** $i < length(T)$ **do**
    $\vec{g} \leftarrow (0, ..., 0)$
    **for** $j = i, ..., i + b$ **do** ▷ For instances in the batch
      $(x_j, y_j) \leftarrow T_j$ ▷ Update feature and target nodes with next instance
      $o_1, ..., o_n \leftarrow forward(G)$
      $\vec{g} \leftarrow \vec{g} + backward(G, o_1, ..., o_n)/b$ ▷ Computing average gradient
    **end for**
    **for each** network parameter $w_i$ **do**
      $w_i \leftarrow w_i - \alpha \cdot g_i$ ▷ Update input node representing the weight $w_i$
    **end for**
    $i \leftarrow i + b$
  **end while**
**end procedure**

It works with batches of training examples. If the batch size is as large as the whole training dataset, the algorithm is changing the weights in the exact direction of the steepest descent of the loss function. But such a large batch is in most cases not possible because of the large training times. With smaller batches the algorithm only estimates the steepest direction based on the average gradient over training examples in the batch. It can be proved that the estimate is unbiased (Goodfellow et al. [2016]).

Many improvements of the stochastic gradient descent were suggested such as the use of momentum (Qian [1999]). In this approach the weights are changed not only by the current gradient, but also by an exponentially weighted average of all the previous gradients. Based on the improvements of the stochastic gradient descent, new algorithms were proposed. The most widely used one is the Adam optimizer, first described by Kingma and Ba [2014]. It rescales the sizes of updates based on the sizes accumulated over past updates and also uses the momentum.

## 2.4 Recurrent neural networks

Feed forward neural networks were successfully applied to tasks with a fixed length of an input and an output. But in some tasks the length is variable. For example in machine translation, we want to map a sentence of a variable length in one language to a sentence of a possible different length in another language. For such tasks feed forward networks do not work well. For this reason a new type of neural networks was invented - recurrent neural networks (Medsker and Jain [1999]).

Recurrent neural networks are similar to the feed forward ones with one important difference. Some neurons in recurrent networks contain a state (memory) which changes over time. These neurons are called recurrent units. When the network processes an input, it reads it one part (e.g. one word in machine translation) at a time. It processes the part of the input and produces an output the same way as feed forward network does, with the only difference that the state of recurrent units changes. Then it processes next part of the input, produces an output and the state of recurrent units changes again. This is happening until all the parts of the input are processed. Because of this, the network can return for the same input different outputs based on the state of its recurrent units.

If the network should produce an output with a fixed length but has a variable length input, then all the outputs over time are somehow added together (e.g. averaged) and used as an output. A second option is that the state of the recurrent units after whole input was processed is used as an output.

If the network should produce an output with a variable length different than the length of the input, an encoder-decoder architecture is used. This means that the variable size input is encoded into a fixed sized vector, which is then used for generating the the final variable size output. A precise description of this architecture can be found in an article by Cho et al. [2014].

The main advantage of recurrent networks over the feed forward ones in sequence processing is a better generalization in case the same information is appearing in a different parts of the sequence. For example if we want to extract the age of John from a sentences such as: "John is 9 years old and likes football", "John likes football and is 9 years old", the recurrent network can learn to find the sequence "$x$ years old" and extract the age from this. A feed forward network on the other hand cannot learn a simple rule like this, because the words in different parts of the sequence are using different parameters of the network.

### 2.4.1 Recurrent units

The simplest recurrent unit uses its hidden state and an input to compute a new hidden state and from the new hidden state the output of the unit is computed (Pascanu et al. [2012]). Formally in time $t$ with an input vector $x_t$ and a previous hidden state $h_{t-1}$ the unit updates its hidden state:

$$h_t = f(Wx_t + Vh_{t-1} + b) \tag{2.5}$$

and computes the output vector:

$$o_t = Uh_t + c \tag{2.6}$$

where $U, V$ and $W$ are matrices with learned parameters, $b$ and $c$ are learned thresholds and $f$ is the activation function.

This unit is not widely used, because it does not work well with long range dependencies in the sequence and suffers from the exploding/vanishing gradient problem. But there are two other widely used recurrent units, which do not suffer from these problems as much: the long short-term memory (LSTM) (Hochreiter and Schmidhuber [1997]) and the gated recurrent unit (GRU) (Cho et al. [2014]).

LSTM contains a hidden state and three gates: a forget, an input and an output gate. These gates use the previous output $y_{t-1}$ and the current input $x_t$ and their output vectors $f_t$, $i_t$ and $o_t$ in time $t$ are computed as follows:

$$f_t = \sigma(W_f y_{t-1} + V_f x_t + b_f) \tag{2.7}$$

$$i_t = \sigma(W_i y_{t-1} + V_i x_t + b_i) \tag{2.8}$$

$$o_t = \sigma(W_o y_{t-1} + V_o x_t + b_o) \tag{2.9}$$

where $\sigma$ is the sigmoid function $W_f, V_f, W_i, V_i, W_o, V_o$ are the learned parameter matrices and $b_f, b_i, b_o$ are the learned thresholds. Note that the sigmoid function is applied separately on each vector component.

The input is multiplied by the input gate and added to the previous hidden state multiplied by the forget gate, which gives us the new hidden state:

$$h_t = f_t h_{t-1} + i_t \sigma(W_h y_{t-1} + V_h x_t + b_h) \tag{2.10}$$

Finally the output is computed as the hidden state transformed by a hyperbolic tangent and multiplied by the output gate:

$$y_t = tanh(h_t) o_t \tag{2.11}$$

GRU is similar to LSTM but it contains only two gates - a reset and an update gate. Their values $r_t, u_t$ are computed in a similar way as in LSTM. The hidden state $h_t$ update is following:

$$h_t = h_{t-1} u_t + (\vec{1} - u_t) \sigma(W_h(r_t h_{t-1}) + V_h x_t + b_h) \tag{2.12}$$

Note that in the equations 2.7 through 2.12 all the multiplications and applications of $tanh$ and $\sigma$ are done by components. The lower letters denote vectors, $\vec{1}$ is a vector of ones and the capital letters denote matrices.

It is not clear why these recurrent units should look exactly the way they are and whether there are any other units with a similar or better performance. This is true especially for LSTM, which is more complicated and where the motivation behind some elements (e.g. input gate) is not clear.

Jozefowicz et al. [2015] tried to answer these questions. After an evolutionary exploration and testing of many different architectures of recurrent units, they found other architectures with a similar or maybe even better performance than LSTM and GRU. They also found that a simple addition of bias 1 to the forget gate makes LSTM better than both the original LSTM and GRU and that LSTM without output gate works almost as good as the original one.

### 2.4.2 Backpropagation through time

Recurrent neural networks are trained similarly as the feed forward ones - using the backpropagation algorithm. The only difference is in the creation of the computational graph.

In case of the recurrent networks the computational graph is unfolded. Suppose the input sequence has length $t$, which means that for processing the input once, the original computational graph $G$ with a single hidden unit (for simplicity) is used $t$ times with hidden states $h_0, ..., h_t$ and inputs $i_1, ...i_t$. The unfolded computational graph then consists of $n$ graphs $G_1, ..., G_n$ same as $G$ but with inputs $i_1, ..., i_t$ and with edges between the hidden states:

$$\forall i = 1, ..., n : (h_{i-1}, h_i) \in E \tag{2.13}$$

On this unfolded graph the backpropagation algorithm is used. When updating parameters, which appear multiple times in the unfolded graph, but only once in the original one, the updates from all the times are summed together and used as the final update to a given parameter.

### 2.4.3 Bidirectional networks

The recurrent networks as described above can easily deal with forward dependencies but not with the backward ones. Given a problem where we are generating a sequence from another sequence of the same length and the $i$-th desired output $y_i$ is a function of the next part of the input $x_{i+1}$: $y_i = f(x_{t+1})$, the recurrent network cannot learn this mapping at all.

To solve this problem, bidirectional networks were introduced (Schuster and Paliwal [1997]). They have one layer of recurrent units which works exactly the same way as for the regular recurrent networks. But they also have an additional layer of recurrent units which processes the same input in reversed order. The bidirectional network produces two outputs for each of the inputs, these outputs are then added together, concatenated or processed by another layers, possibly feed forward ones.

The bidirectional recurrent networks are often better than the unidirectional and are widely used in practice (Lipton et al. [2015]).

## 2.5 Dropout

Dropout is a regularization technique whose efficiency was shown for example by Srivastava et al. [2014]. In each training step each neuron is ignored with some probability $p$. This can be viewed as implicitly training not only a single neural network, but an ensemble of networks. Each of the networks of the ensemble is a subnetwork of the original one. They share their weights, but each one has only a subset of the neurons of the original network. Thus we have one set of weights which represents multiple subnetworks.

During testing, we want to average the outputs of all the subnetworks. This is done by using all the neurons, but their outputs are multiplied by $1 - p$, since they are not in one $p$-th of the subnetworks.

## 2.6 Embeddings

Word embeddings is a method of encoding words as a vectors of real numbers, which is widely used when processing text by machine learning algorithms. There are many possibilities how the vectors will look like.

The simplest variant is the one-hot encoding. Let $V$ be the vocabulary. In the one-hot encoding the word $w_i \in V$ is encoded as a vector $e_i \in R^{|V|}$ with $e_{i,i} = 1$ and $\forall j \neq i : e_{i,j} = 0$. In other words the $i$-th word is encoded as a vector containing zeros except for the $i$-th place, which contains one. The advantage of this approach is its simplicity, but the disadvantage is that with a big vocabulary the embeddings become too large.

Second variant are randomly initialized word embeddings. Here for each $w_i \in V$ random vector of some dimension (typically 50 - 500) is generated, where each element is from a continuous uniform distribution with bounds -1 and 1. Formally $e_i \in R^{50}, \forall j = 0, ..., |e| - 1 : e_{i,j} \sim U([-1, 1])$. It is possible to change these embeddings during the training of the network by the backpropagation algorithm, in which case we call them trainable word embeddings.

Another approach is the usage of pretrained embeddings. This means that the embeddings are trained by unsupervised learning on some large general natural language dataset, for example on Wikipedia articles. Two common methods of computing the pretrained embeddings are the Continuous Bag-of-Words Model (CBOW) and the Skip-gram Model described by Mikolov et al. [2013].

The idea of the CBOW is that similar words which should have similar embeddings often appears in similar contexts. The model for training the embeddings is predicting $i$-th word of the data from its context - words $i-k, i-k+1, ..., i-1, i+1, i+2, ..., i+k$. The order of the words does not matter, which is why this method is called Bag-of-Words. The model used for generating the embeddings consists of two weight matrices $W \in R^{d \times |V|}$ and $U \in R^{|V| \times d}$, where $V$ is the vocabulary and $d$ is the embedding dimension. The output of the model is given by:

$$\vec{o_i} = U(\sum_{j \in i-k,...,i+k, j \neq i} W \cdot \vec{e_{w_j}}), \tag{2.14}$$

where $\vec{e_l}$ is the one-hot encoding of the $l$-th word from the vocabulary and $w_j$ is the $j$-th word in the training data. The model is trained to predict $\vec{o_i} = \vec{e_i}$. $k$ has typically value between two and four. The final embeddings of the $i$-th word is the $i$-th column of the matrix $W$.

The skip-gram model is similar to CBOW but it predicts a context of a word from itself.

Another variant are the character embeddings (Ling et al. [2015]), in which each character has it's own randomly initialized trainable embedding. The final word embeddings are then computed from them using a recurrent layer.

# 3. Data and evaluation

We use the dataset created by Bisk et al. [2016a]. Our world consists of 20 square blocks distributed on a board. Their positions are described by 2-dimensional coordinates. The dataset contains two types of information: the coordinates of square blocks and commands (sentences in English) describing how these blocks are moved.

We call the positions of all blocks currently on the board a world state. The world states in the dataset form 100 sequences of length 9-21 (with an average length 19.63) and therefore there are 1,963 world states. A part of one sequence is shown in Figure 3.

The first world state in each sequence was generated randomly and the last one forms a digit taken from the MNIST dataset (LeCun [1998]). Each two consecutive world states differs from each other in just one block, which is moved to its final position in the MNIST digit. There are 8-20 blocks (with average 18.63) in each world state. To be easily distinguished among each other, the blocks are not blank. They have digits or logos of companies (see Figure 3) on them, where one half of the sequences contains blocks with digits and the other half with logos.

The block coordinates in the first state in each sequence are randomly generated real numbers between -6.6 and 6.6. In the last state the block coordinates are always divisible by 1.0936. Apparently 1.0 is the block size and 0.0936 is the distance between blocks. It would be possible to scale these numbers so that they are divisible by 1.0, but that would make it harder to compare our results with the results of Bisk et al. [2016b], which is the reason why we have not done it.

Each change in the world states (block move) is described by nine commands, all of them describing the same action. This gives us 16,767 commands in the whole dataset. The 100 sequences are divided into 70 train sequences, 10 development sequences and 20 test sequences, with the same amount of sequences with logos and digits in them. Because the sequences does not have the same length, there are 11,871 commands in the train data, 1,719 commands in the development data and 3,177 in the test data (see Table 3).

The commands were written by people from the Amazon Mechanical Turk, who saw two consecutive world states and were asked to write instructions for another human how to get the latter state from the former one. The nine commands describing each change in the world were always written by three different people, where each wrote three commands. In total, 126 people wrote some commands, but some contributed much more than others. The most productive author contributed with 4,269 commands - approximately a quarter of the whole dataset - and the four most productive authors together contributed with 8,787 com-

| | Sequences | Commands | Tokens | Average command length |
|---|---|---|---|---|
| Train | 70 | 11871 | 176401 | 14.86 |
| Development | 10 | 1719 | 30781 | 17.91 |
| Test | 20 | 3177 | 47877 | 15.07 |

Table 3.1: Division of data to the training, development and test dataset

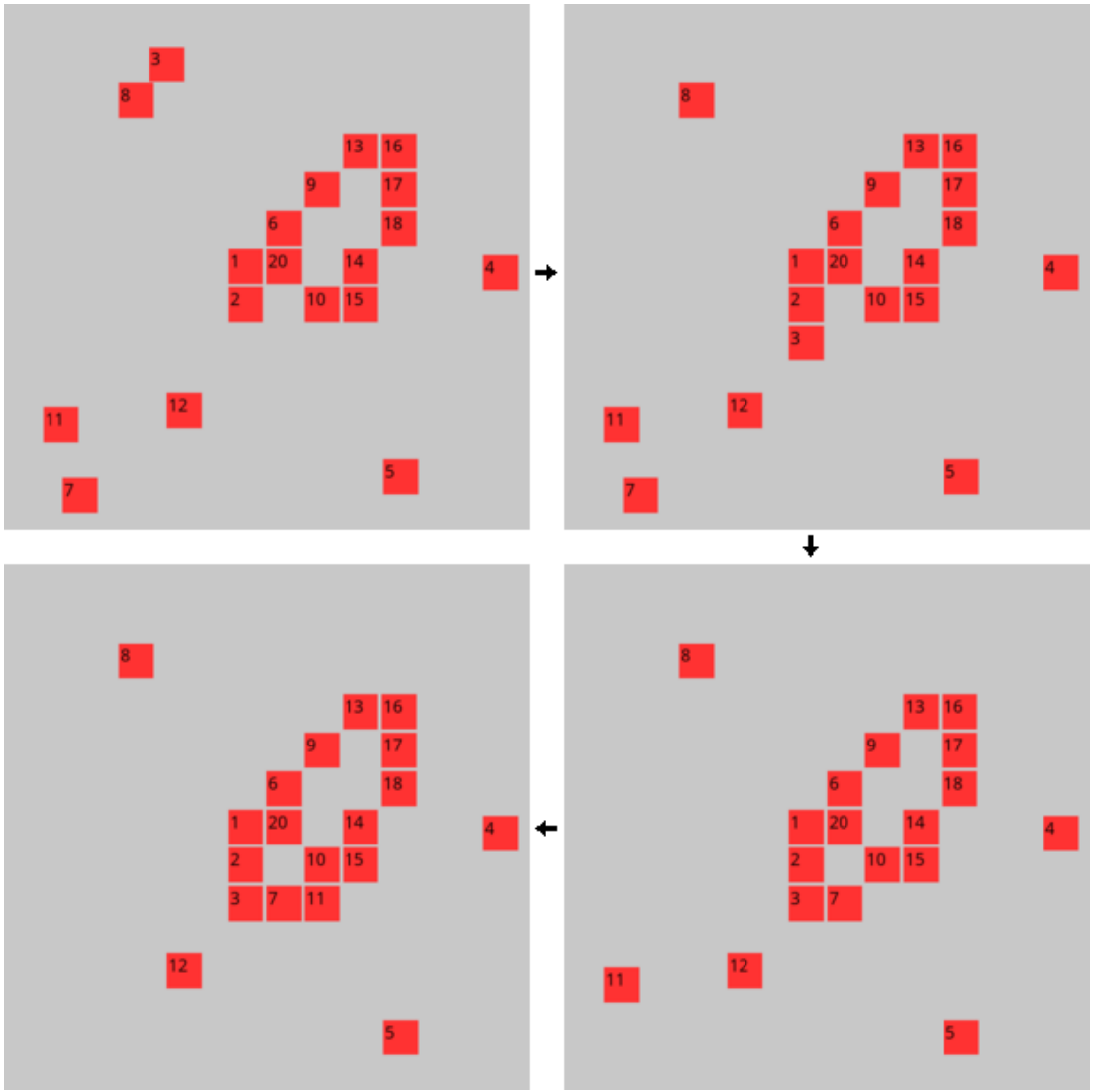

Figure 3.1: Sequence of world states

mands, which is a little bit over one half. For some authors, all their commands are in only one subset (for example the development data), for others the commands are in all the subsets. This is a little bit problematic, because the style of the commands differs between authors, which together with how the blocks were generated and the data divided between the train, test and development subsets leads to inequalities in these subsets. It would be possible to divide the data in a different way, but then we would have difficulties with comparing our results with Bisk et al. [2016b].

On the other hand the unequal division of the commands between the authors better corresponds to a potential real world application, where the system will have to understand people who did not participate in creating the data.

Because the authors were ordinary people who were payed for writing the commands, they contain a lot of mistakes and typos, for example:

> *place the Mecedes box so that a vertical line read from top to bottom reads Mconalds, blank space, Mercedes Benz, Nvidia*

Note wrong spelling of *Mercedes* and *McDonald's*, an unusual capitalization of

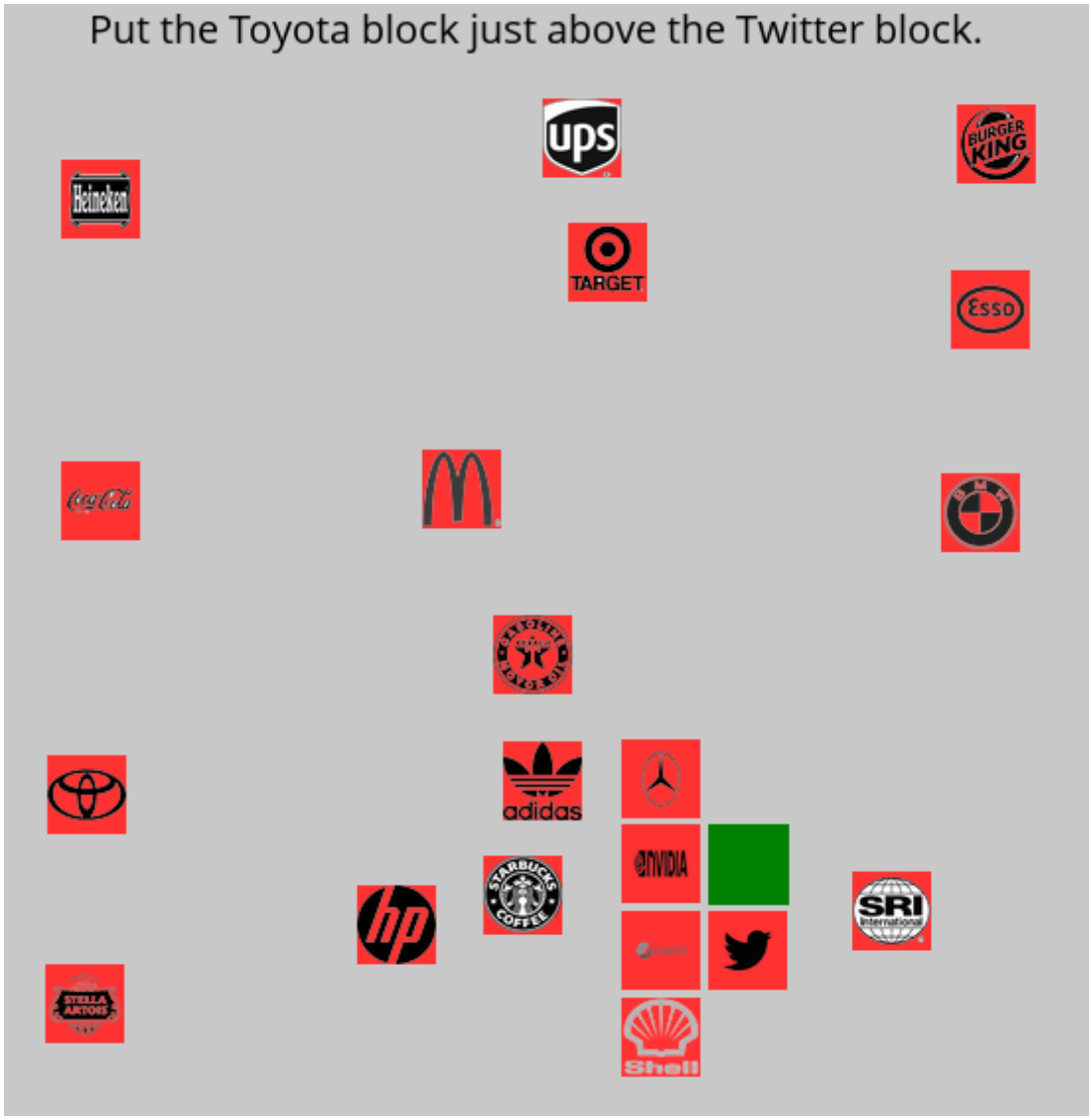


Figure 3.2: Command and world example - the model should predict that the Toyota block should move to the green location

*NVIDIA*, no capitalization of the first letter and a missing full stop in the end of the sentence.

The longest command with 83 tokens is:

> *The 9 box is in the top right corner. Slide it left at least two block widths. Then slide the 20 box between the 9 and 16 boxes. Then slide the 20 box between the 15 and 18 boxes. Line the 20 box so that that top edge of the box is halfway down the 13 box to the right. The 20 box will also be about a half of a box length away from box 13.*

It does not make much sense, because two blocks (9 and 20) should be moved according to it. The second longest with 79 tokens is correct:

> *Push the 3 block over until it is just above the 2 block, then slide it all the way down, then slide it all the way right, then slide it all the way down until it touches the 2 block, then move it left until it touches the*

| Token | the | block | . | of | to | left | and | right | place |
|---|---|---|---|---|---|---|---|---|---|
| Count | 18597 | 14823 | 8702 | 7887 | 5024 | 4356 | 4234 | 4003 | 3781 |

Table 3.2: Most frequent tokens

*four block, then move it up until it clears the 4 block, then move it left until it is directly above the 4 block.*

The shortest commands have 4 tokens (for example "*put adidas above bmw*"). The average number of tokens in a command is 15.2.

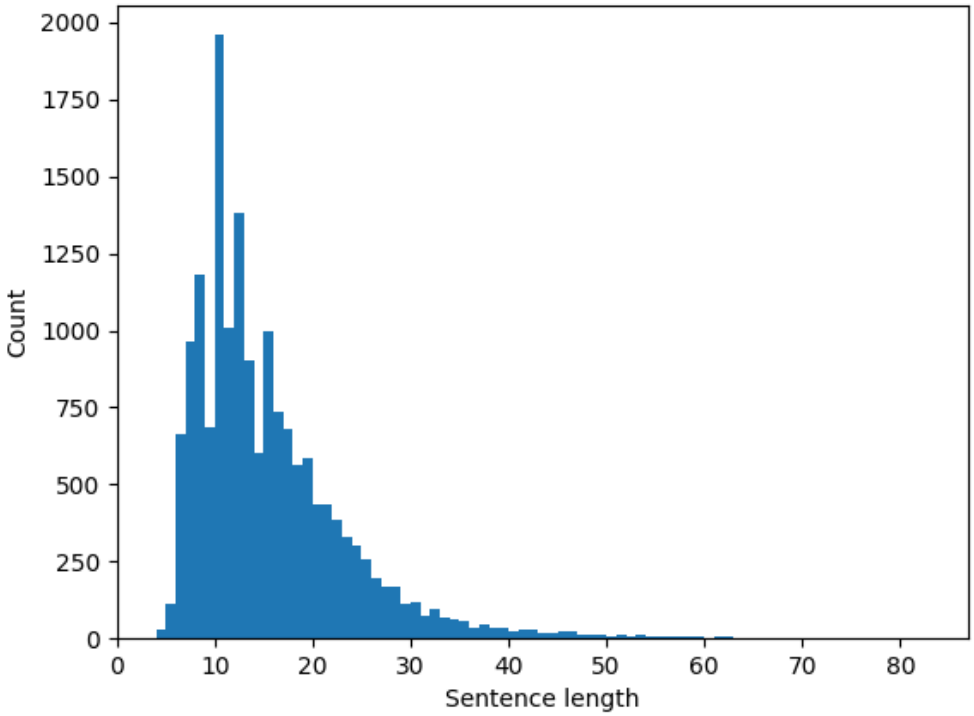


Figure 3.3: Histogram of command lengths

Without any error correction, the training commands contain 1,016 distinct tokens, from which 377 are there only once (ignoring casing of letters).

## 3.1 Types of commands

We tried to analyze the data by finding words denoting blocks (i.e. *1,2, ..., one, two, ..., adidas, bmw, ...*), words denoting directions (i.e. *left, west, southwest, bottom, top, under, above, ...*) and words denoting distances (i.e. *Move BMW **five** spaces left*) in the commands. Unfortunately we were not able to find all the words which denote blocks or directions, because there are many different possibilities how to describe them. For example in a sentence:

*Put the block that looks like a taurus symbol just above the bird.*

| Number of blocks | 0 | 1 | 2 | 3 | 4 | 5 | 6 | $> 6$ |
|---|---|---|---|---|---|---|---|---|
| Number of commands | 23 | 394 | 13746 | 2208 | 292 | 66 | 20 | 18 |

Table 3.3: Approximate number of blocks mentioned in commands

where the first mentioned block is Toyota and the second is Twitter we do not recognize neither of them. Also sometimes the context is necessary to determine the meaning of a word, for example numerals can describe blocks as well as distances. Thus the numbers in this section are only approximate.

Based on the words we find in them, we divide the commands in the following categories (first number in the parentheses means the total number of commands in the category, the second number is the percentage):

- Source, reference, direction (7799, 47%)

  These commands contain a noun denoting which block should be moved (which we call a source), then a word describing a direction (*west, above, left, ...*) and finally a reference block. These commands also contain other words, but these three words are the distinctive features. This structure of commands was previously described by Bisk et al. [2016b].

  Examples:

  > *11 should be east of 6*
  >
  > *move toyota just to the right of stella*
  >
  > *Move the Toyota block around the pile and place it just to the right of the SRI block.*

- Source, reference, 2 directions (2611, 16%)

  These commands are similar to the previous category except they have two direction words instead of one. Often these two direction words are used to describe a diagonal position from the reference block:

  > *Move block 19 diagonally below and to the right of block 14.*

  or the first direction word describes direction from the source and the other from the reference:

  > *Move Nvidia to the southeast until it's on top of Pepsi.*

- Source, reference, more than 2 directions (706, 4%)

  > *Slide 16 down and to the left until it is slightly above and left of 13.*

- Source, reference, no direction (296, 2%)

  Without any information about the world state these commands are often ambiguous.

  > *Place 7 next to 9*

- Source, reference, distance (1400, 8%)

  The commands of this category have a source, a reference and a word determining distance in them (e.g. ***five*** *spaces left*). Almost all of them contains some direction words.

  > *move 1 three spaces to the left of 19.*

- Block mentioned multiple times (934, 6%)

  These commands also have two distinct blocks in them, but at least one of them is mentioned multiple times. Their structure is complicated. More than one third of these commands is composed of two or more sentences. Commands with a single sentence almost always use a compound sentence. For comparison only 4% of the commands in the other categories are composed of multiple sentences, therefore multiple sentence commands are about ten times overrepresented in this category.

  > *Take the 5 block and place it so that the top left of the 5 block is touching the top right of the 2 block and the bottom left of the 5 block is touching the bottom right of the 2 block.*

- More references (2604, 16%)

  The commands in this category contain more than two block words, where one is a source and the others are references. Their structure is complicated and more diverse than the structure of commands in the other categories. They often use two references to describe the location between them:

  > *Pick up box 5 and place is directly between and lined up with boxes 4 and 6.*

  or use one reference for the horizontal and one reference for the vertical positioning:

  > *Move block 10 to the same horizontal line as block 8 and same vertical line as block 5*

  Multiple references are also used when the location is described with geometric figures:

  > *move block 13 such that it continues the diagonal created by 20, 19, 15*

  Also commands with multiple sentences and compound sentences often appear in this category:

  > *Move Adidas above Coca-Cola, slide right until hitting SRI, slide up until almost touching Mercedes, then slide left until just clearing Burger King, so the King's left lower corner is almost kissing Adidas' upper right corner. Ta-da!*

- No reference (417, 2%)

  The commands without reference describe the location in two ways. They use either an absolute position:

*Move the Stella block down to the very bottom of the square.*

or a position relative to the source:

*move the pepsi bolck three block widths to the left*

Some of the commands in this category actually contain two blocks, but we are not able to automatically detect one of them, because they are described in an unusual way. For example:

*Put the pepsi block directly below the one with the spiral.*

We estimate that approximately 1 out of 5 commands of this category has this problem.

Although high number of the commands are simple and straighforward, some of them are very creative and need advanced reasoning to understand them:

*Put block 16 in the empty space at the top of the column of two blocks that have numbers on them that when added to 16 equal 51.*

*Consider a grid with 3 rows and columns, numbered 1,2,3 then 4,5,6 then 7,8,9. If SRI is at 3, Shell is in 5, then place HP in 7.*

## 3.2 Two tasks

The problem we are trying to solve is given an initial world state and a command we should generate a world state after the execution of the action described in the command. Each action consists of moving one of the blocks on the board from one 2-dimensional position to another. The naive approach for solving this task is to create a model, which gets an initial world state and a command as input and which produces the target world state. Because there are at most 20 blocks in each world state and each has two dimensional coordinates, such a model has 40 real valued outputs. It is possible to try solving the problem this way, but we decided to use the fact that in our data only a single block changes its location between an initial and a target world state. So we can divide the problem into two subproblems:

- Source prediction: Predict which block should be moved.
- Location prediction: Predict the target location of the source.

For example given the world in Figure 3 the source block is the Toyota block and the location is the green square. Predicting the source block is a classification problem in which one of the 20 classes is outputed - one for each block. For location prediction we use regression model, which outputs 2 numbers - $x$ and $y$ coordinates of the location, where source should be moved.

From the initial world, the source and the location we can compute the target world by simply changing the location of the predicted source block in the initial world to the predicted location. So regarding our data this representation is as expressive as the naive one and we think it is easier to train models for this one.

Note that models using this representation can be trained separately, because for each instance we can compute which block was moved and where, because we have both the world state before the command was executed and after.

The reasoning behind this division is that in the naive approach the model which would be able to solve the task, would have to internally decide which block should be moved and where and also copy all other blocks from input to output. This is harder than just deciding which block should be moved or deciding where it should be moved. Also the later approach prevents mistakes such as moving many blocks at a time, which is something that the model using the naive approach would have to learn. Based on these arguments we used the approach with two tasks.

Bisk et al. [2016b] go even further in the division of the task. They found out that the most common structure of sentences contains three distinctive features: source, reference and direction. Based on this, they divide the task into three subtasks predicting these three features and computing the target world from them. Source and reference are blocks and direction represents shift to one of the eight neighbouring tiles. However this representation has two problems. First, we do not know what is the correct reference and direction. This can be solved by some rules, which in most cases find the correct reference and direction, but which are sometimes wrong. This is what Bisk et al. [2016b] use. Second, this representation cannot describe some of the commands, for example the command

*Move Stella Artois down til it is at the bottom edge.*

does not contain reference.

Fortunately for evaluating their results Bisk et al. [2016b] use the two tasks approach, so we can easily compare our results with theirs.

## 3.3 Evaluating model performance

As was written in Section 3.2, we divide our task in problems of predicting source and location. And we also test these two components independently.

For the source prediction, we use standard accuracy (ratio of correctly predicted instances to all instances) as the metric for comparing performance. For evaluating the location performance, we use the average Euclidean distance between the predicted and correct location over all instances. The unit of measurement is the size of the block edge.

All the models are trained on the train set and does not see any instances from development and test set. Development set is used for tuning hyperparameters, comparing different models, analysis of models and for tuning components programmed for this task (such as rule-based tokenization or benchmark model). Test set is used only for final evaluation of some of the models.

The results of training neural network depends on the random initialization of weights. To compare different settings, architectures and hyperparameters of different networks we want to minimize the impact of randomness. The easiest way to do this is to train same network multiple times with different random seed.

Based on results of ten network trainings each with same network but different seed, we estimated the standard deviation of the source prediction accuracy

as 0.17%. Similarly we estimated standard deviation of the location prediction distance as 0.017.

To achieve standard deviation of approximately 0.1% for the source prediction and 0.01 for the location prediction we decided that the best way of comparing different networks is to train each of the networks in comparison three times with different seed and use the average result of these three trainings. Then the probability of difference of two results (averages of three) being 0.2% for source or 0.02 for location just by chance is less than 95%. Unless written otherwise, this method of computing results is used in whole thesis.

The results also depend on number of epochs we train the network. In all experiments we use the same setting: if the development error has not decreased in 20 epochs we end the training and use weights which had the best development result. This technique called early stopping is useful in preventing overfitting (Prechelt [1998]).

# 4. Tokenization

The very first step of data preprocessing is tokenization - dividing the input sentences into words. This process is often considered to be a trivial (Straka et al. [2016]). However for our task the tokenization is more difficult because of errors (e.g. missing space between words) in the sentences and because the logo names are unusual words in general text.

For example in the command:

> *Slide the Adidas block upwards, until it is directly beneath the Burger King block. Slide it to the left, so it's right edge is lined up with the BurgerKing blocks left edge. Slide it up so the bottom and top edges of the BurgerKing block and the Adidas block are parallel, and the two blocks are directly next to each other.*

the logo *Burger King* is spelled two times without space and one time with space. Other difficult example is the command:

> *Move Burger King so it is belowBMW*

where the space is missing between the last two words.

## 4.1 Rule-based tokenization

First we created rule-based tokenization system which is tuned manually. Since the amount of data is not very big, we want the system to produce low amount of distinct tokens, because otherwise many tokens would be in the data just once and learning would be difficult. To achieve this, the system first changes all upper-case letters to lower-case. Then, in this order:

- It separates commas by spaces.
- It removes posessive *'s*, quotation marks and parentheses.
- It substitutes semicolons with spaces.
- It adds space before each full stop followed by space. If there is not space after full stop, it does nothing, because we want to prevent changes in floating point numbers (e.g. *1.5* should stay as it is).
- It removes apostrophes.
- It substitutes dashes with spaces.
- It adds a space between letter and digit and between digit and letter if there is not space already between them. This is useful for separating block numbers from other words e.g. *block6*.

Then as a final step, the sentences are split to tokens according to spaces.

In majority of cases it works well, but in some specific cases it is wrong. For example, the system ignores slash, because it is in rational numbers (e.g. *1/2*) which should stay as they are, but in command:

*Slide the Shell block left until its left edge is in a vertical line with the right edge of the Heineken (and HP/McD/Mercedes) block, then slide it downward until its lower left corner is touching the Heineken's upper right corner.*

it would be better to remove the slash or treat it as standalone token.

We tried to make the system better by adding more rules but it was becoming more and more complex and the tuning was getting harder. Also because this system was tuned manually based on training dataset, there was a danger of creating tokenizer which is good for the training data but which is much worse on the testing data.

## 4.2 UDPipe tokenization

To overcome the problems with rule-based tokenization we decided to use a software for general tokenization. We decided to use UDPipe[1] (Straka et al. [2016]), which is a easy-to-use tool for tokenization, lemmatization, tagging and parsing. It is composed of several neural models which perform these individual tasks. Models for many languages including English are available for UDPipe. They were trained on Universal Dependencies data (Nivre et al. [2016]), but it is possible to train models with UDPipe for different datasets. On the English Universal Dependency dataset the model has F1-score 98.7% for tokenization.

It works relatively well, but it has some problems, especially with the logo names, which are the most important words. For example word *"BMW's"* is tokenized as *"B", "MW", "'s"* or *"20,19,18,17"* is tokenized as single word, because the spaces between numbers are missing. Also unlike the rule based system UDPipe does not remove special characters and makes mistakes because of it. For example while most fractions are denoted as single token, sometimes UDPipe divides them to two tokens (e.g. *1/2* becomes *1* and */2*). Similarly for parentheses in most cases they are a single separated token, but sometimes they are joined together with other word.

Since we have no gold data for tokenization, we cannot automatically evaluate which approach is better. But we can at least test our prediction algorithms, which are described in following chapters, on data tokenized by both approaches. The results suggest that the rule-based approach is better, but the difference is very small.

[1] Online demo: http://lindat.mff.cuni.cz/services/udpipe/run.php

# 5. Models development

In this chapter we present the models for solving the task, from the simplest one to the more complicated. Also we discuss hyperparameter optimization. Note that all results in this chapter are measured on development set, test set results are in Chapter 7.

## 5.1 Baseline

We created two baseline models - random one and deterministic one. The random baseline predicts a random block as the source and a random location within the board as the target location. On development set it has source prediction accuracy 5.8% and average distance between correct and predicted location of 5.96.

The deterministic baseline predicts the block 1 as the source, which (together with other blocks) is the source most often. For the location prediction this baseline always predicts the middle of the board.

Source accuracy of this baseline is 5.2% and the average distance between predicted and correct location is 3.55.

Note that results of random baseline depends on random seed. It happened only by chance that the random baseline has better result then the deterministic one for predicting source.

Similar baselines were measured by Bisk et al. [2016b].

## 5.2 Benchmark model

Before trying models based on neural networks we created a benchmark model based on finding words in the command. Specifically it searches for words which would denote

- blocks
  - digits *1, 2, ..., 20*
  - numerals *one, two, ..., twenty*
  - logo names *adidas, bmw, burger, king, ..., ups*
- directions *west, north, east, south, left, above, right, below*

If the logo name contains more words (for example *burger king*), this model searches for each part of the word and for concatenation of both words (for example for logo name *burger king* it would search for three words *burger, king, burgerking*). If the blocks have digits on them, then the model searches only for digits and numerals when looking for words denoting block. Similarly if the blocks have logos on them, the model ignores digits and numerals and looks for logo names. The model receives information whether logos or digits are used in the command as part of its input.

| | Source | Location |
|---|---|---|
| Random baseline | 5.8% | 5.96 |
| Deterministic baseline | 5.2% | 3.55 |
| Basic benchmark | 98.2% | 1.45 |
| Improved benchmark | 98.2% | 1.10 |

Table 5.1: Results of benchmark on development set

For predicting the source (which block should be moved), the model predicts the block corresponding to the first word in the sentence which could denote a block. For predicting the location (where the source should be moved), the model predicts position of the last word describing block, if there are at least 2 blocks in the sentence. Otherwise it predicts the middle of the board. If there were some words describing directions, the last one is chosen and the position is changed by one in the direction corresponding to the word.

For example given blocks with logos and sentence:

> *Put the **UPS** block in the same column as the **Texaco** block, and one row **below** the **Twitter** block.*

The model would "see" the bold words, from which three can be describing blocks and one - *below* - is describing a direction. The predicted source would be the first block word - *UPS*. The predicted location would be the current location of *Twitter* block (last block word) moved one tile down, because of the *below* word. If the blocks had digits on them, then only one word - *one* - could be denoting block and would be predicted as source.

We tested this model on development set. It has source accuracy 98.2% and average distance between predicted and correct location 1.45.

After analyzing the locations predictions of this model, we found out it often makes following three mistakes:

- It uses source as reference if source is mentioned in the end of sentence again.
- It mistakes a number describing distance between blocks with reference (e.g. In *Slide block 13 to sit directly under block 15. Move over one block space to the left.* the model thinks that *one* is reference instead of 15.).
- It ignores all but the last reference.

Based on these problems we improved the model. To overcome the first problem, we changed the model so that it ignores the blocks predicted as the source (first block in the sentence). To solve the second problem, we do not recognize as block names those words which are followed by *space, row* or *column*, because these three words indicate that the previous word is describing distance (e.g. *"one row", "two spaces"*). Also if there are digits in the command, we do not recognize as block names the numerals, because in most commands digits are used to denote blocks and numerals to denote distance. And for the third problem the model predicts the center of gravity of all recognized block names except the first one, which is assumed to be the source.

We also started to recognize diagonal directions (e.g. *southeast*) and more direction words such as *underneath* and *downwards*.

These changes improve the performance of the location prediction from 1.45 to 1.10 on development set. In 44.3% of predictions the error is less than 0.5. The source prediction accuracy remains the same after these changes. Comparison of benchmark results with baselines is in Table 5.2.

The improved benchmark model makes mistakes mostly in sentences without reference block:

*Move Stella Artois down til it's at the bottom edge.*

and in sentences where first block is not source:

*To the left of Target block, put the Shell block*

Solving these problems needs at least understanding of many more words and their order in the sentence.

## 5.3 Neural models with the world on input

The first models we tried are relatively straightforward. They use both world state and command as input and predict source or location, as was described in Section 3.2. To be comparable with other architectures we tried, we will describe them using the best hyperparameters, which are discussed in Section 5.6.

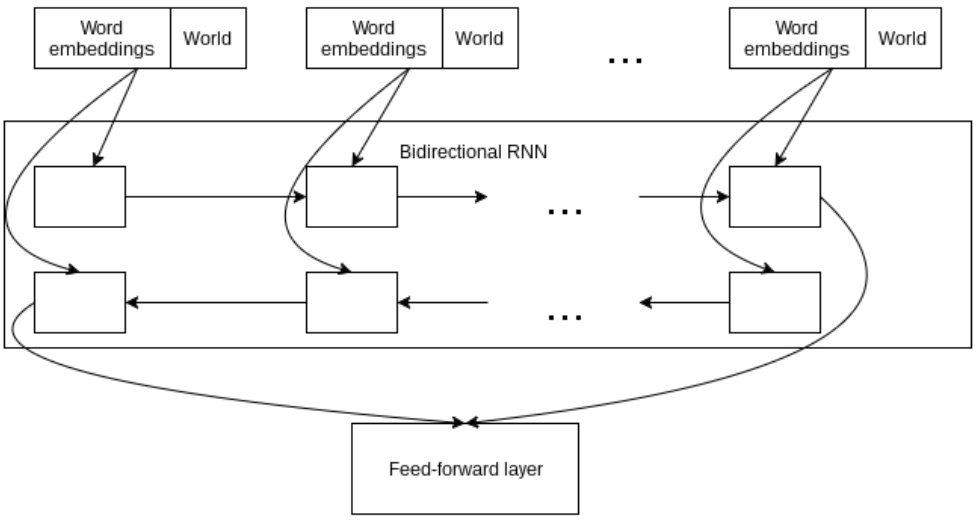


Figure 5.1: Multiple world architecture

The model for predicting source block uses randomly initialized trainable word embeddings to encode the command. Each embedded word is concatenated with the world state and used as an input into the bidirectional recurrent layer. We use only the two last states of this layer, the outputs are ignored. The last states are concatenated and fed into single feed forward layer with linear activation function. This layer has dimension 20 and its outputs are then used as logits to predict the source block. The architecture is illustrated in Figure 5.3.

| | Source | Location |
|---|---|---|
| Random baseline | 5.8% | 5.96 |
| Deterministic baseline | 5.2% | 3.55 |
| Basic benchmark | 98.2% | 1.45 |
| Improved benchmark | 98.2% | 1.10 |
| Multiple world architecture | 97.8% | 3.96 |
| Single world architecture | 98.5% | 2.87 |

Table 5.2: Results of architectures with world on input

For predicting the location the network is similar except for the feed forward output layer. This layer in location predicting model has dimension 2 instead of 20 and its outputs are directly interpreted as the predicted location.

The main problem of this architecture, which we call "Multiple world architecture", is overfitting. We think, that the reason why it is happening is, that the model receives the world state many times, but each word in the sentence appears only once. And the words are more important for the predictions than the world state itself.

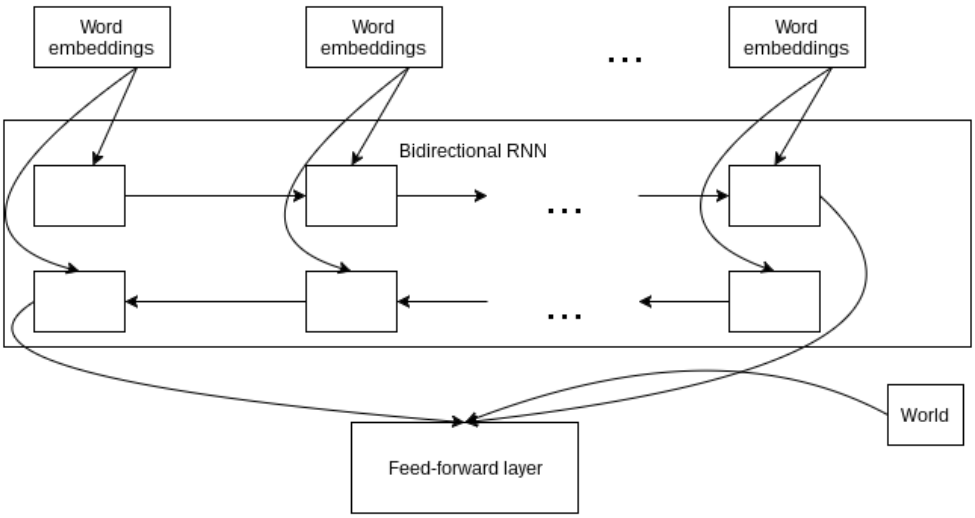


Figure 5.2: Single world architecture

Based on this, we tried another architecture for predicting both source and location, which is similar to the previous one but with a single difference. The world state is not concatenated to each word, but it is instead concatenated to the outputs of the recurrent layer, which is illustrated in Figure 5.3. We call this architecture "Single world architecture".

The results of these two architectures are in shown Table 5.3. Except for predicting source with the "Single world architecture", the models failed to surpass the benchmark. And for predicting location the "Multiple world architecture" was also worse than middle of the board baseline.

The main reason behind the failure of the "Multiple world architecture" for predicting the location is overfitting. After epoch 2 this model reaches best result on development dataset. At this time, the difference between development and training error is big - the development error is 3.83 and training is about 2.2.

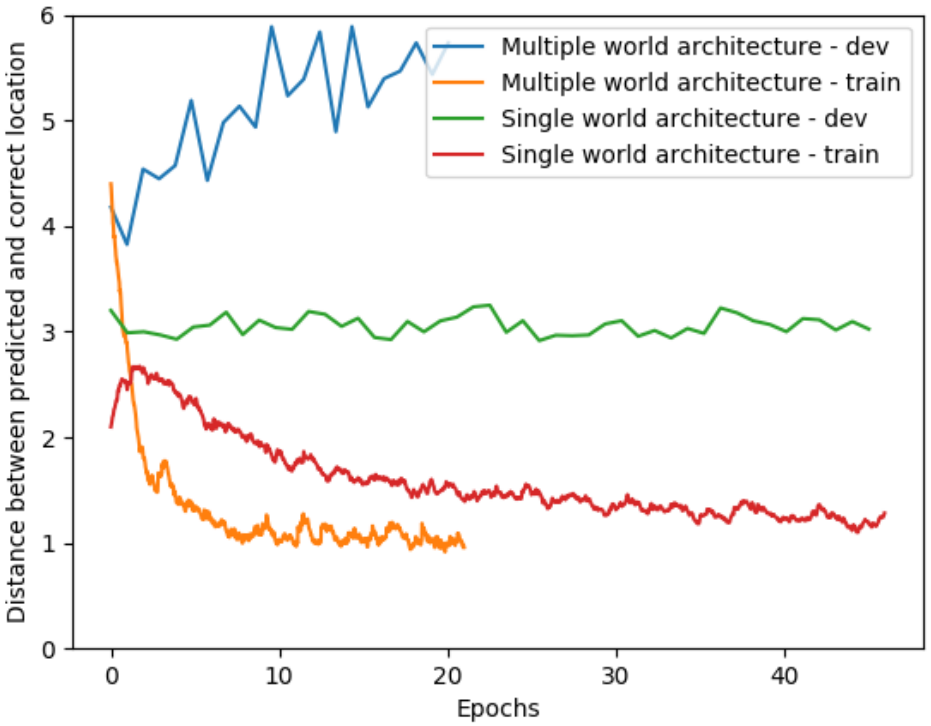


Figure 5.3: Both architectures with world on input have problems with overfitting when predicting location

And from then the development error is increasing while the training error is decreasing. The reason why this happens is probably in the dataset. As it is written in Chapter 3, each change of a world state is described by 9 commands, which means that during the training the network receives same combination of world and target location 9 times, each time with different command. Therefore, the network probably learns to predict the location based on the current world and ignores the command. And this is probably also the reason why the network for predicting source does not work well.

For predicting location with the "Single world architecture" the overfitting is also a problem, but not as much as for the "Multiple world architecture". After few initial epochs, the training error is steadily decreasing, but the development error is staying about the same.

The reason why the "Single world architecture" does not predict location well might be that the task we want the network to learn is too hard. To work at least as good as the benchmark, the network has to understand the sentence, identify reference block and direction words, identify the location of reference block and change this location based on the direction words. For learning this many tasks in one network we would probably need bigger network and more data to prevent overfitting. We can also reduce the number of tasks the network has to do.

## 5.4 Predicting block weights and direction

The network for predicting the source block does not need the world state at all, because all the information needed for identifying source are in the command. For predicting the location it is more complicated.

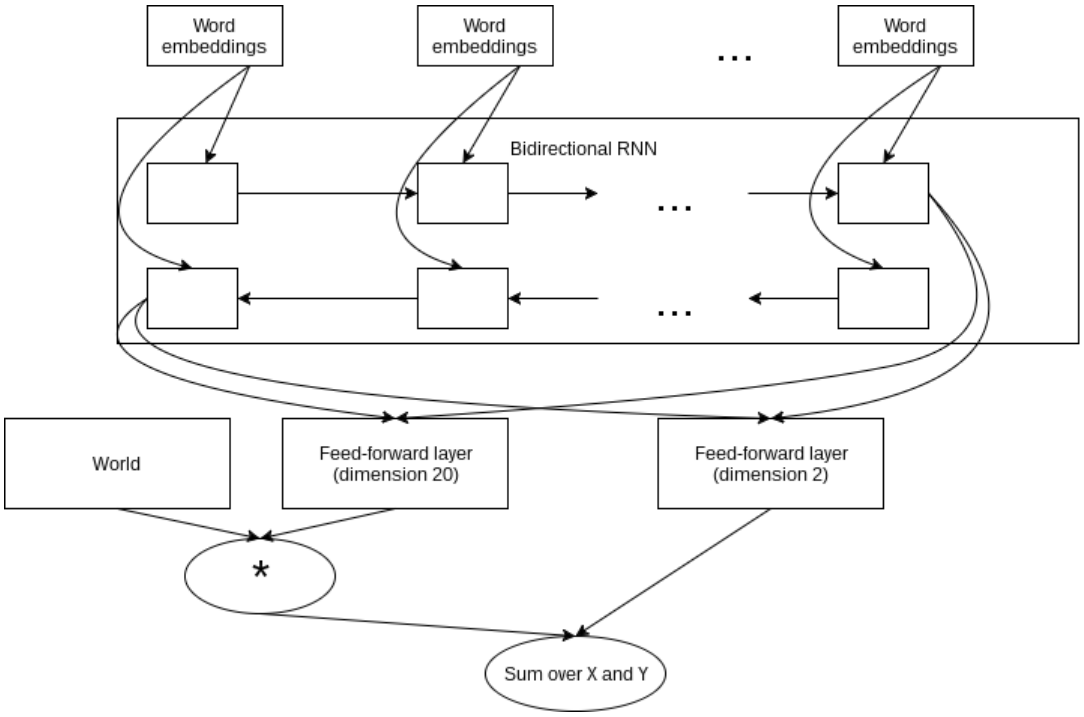


Figure 5.4: Predicting weights and direction

In most cases, for understanding the command, world state is not needed. But it is needed after the command was understood, to interpret it and to compute the final location. For example, given the command "*Move block 3 above block 8*", people can understand it without knowing anything about the world, but to predict the correct location, they need to know where the block "*8*" is. Our architecture is similar, it does not predict directly the location, but some representation of the meaning of the command, which is then interpreted based on the world state to get the final predicted location.

Our representation is composed of 20 weights, which represent how much each block is used as reference, and 2-dimensional vector representing how far and in which direction from the reference the predicted location should be. The neural network outputs these 22 numbers. Let $w = (w_1, w_2, ...w_{20})^T$ represent the 20 weights, $d = (d_1, d_2)^T$ represent direction and

$$S = \begin{pmatrix} s_{1,1} & s_{1,2} & \dots & s_{1,20} \\ s_{2,1} & s_{2,2} & \dots & s_{2,20} \end{pmatrix}$$

be the state of world, where $s_{1,1}$ and $s_{2,1}$ are $x$ and $y$ coordinates of first block (Adidas or block with digit 1), $s_{1,2}$ and $s_{2,2}$ are coordinates of second block and so on. Then the final location $l \in R^2$ is computed as $l = Sw + d$.

In most commands location is described in one of the following ways:

- by reference and direction: *Move BMW above Adidas*

- by reference, distance and direction: *Move BMW 3 spaces above Adidas*
- by absolute location: *Move BMW to the middle of bottom edge of the table*
- by direction relative to source: *Move BMW 3 spaces down*
- by two references: *Move BMW between Adidas and UPS*

Our representation can express the meaning of all of these. Assume that the weight $w_1$ is associated with the Adidas block and the weight $w_{20}$ with the UPS block. Then for example the first command can be represented as $w = (1, 0, 0, ..., 0)^T, d = (0, 1)^T$, and last one as $w = (0.5, 0, 0, ..., 0, 0, 0.5)^T, d = (0, 0)^T$.

The architecture of the network for location prediction is in Figure 5.4. For predicting the source, the network is much simpler. It has only one feed-forward layer with dimension 20, whose outputs are used directly as logits for identifying the right block. It is similar to the "Single world architecture" network in Figure 5.3, but it does not uses world in any way.

### 5.4.1 Location prediction error analysis

The model for location prediction achieves average distance of 1.26 on development set. This is better than our previous models and also better than our basic benchmark, but worse than our improved benchmark which has average distance 1.10.

Also the best model of Bisk et al. [2016b] is better[1]. They reported distance of 1.05 on with his End-to-End model, which is not described in their article precisely, but is probably similar to our model. Also they achieved distance of 0.98 with their discrete model, which contains two neural networks, one for predicting reference block and the other for predicting one of eight possible directions.

Given these results, it would be possible to try to improve our location prediction performance by using architecture similar to the discrete model of Bisk et al. [2016b]. But their proposed architecture has significant disadvantage in that it cannot work well for complicated sentences, as we noted in Section 3.2. And because we want our model to work with these sentences, we decided to not use their architecture and rather analyse why our model does not work better.

ca

We let the model predict locations for all instances in the development data set. Then we selected 100 instances, where the model made the biggest mistakes and manually classified them according to the subjective cause of the bad prediction. The major causes of problems in predictions are in Table 5.4.1.

The most common problem are relatively simple sentences without any obvious features which makes the correct prediction hard. The hyperparameters of our network were already optimized (hyperparameter optimization will be described in Section 5.6) so we came to the conclusion that the problem have to lie in the architecture.

[1]Note that we are here presenting our results on development set and comparing them to the results Bisk et al. [2016b] reported on test set. But even after testing our model on test set, our model is still worse, so the conclusions presented here are still correct. Our test set results will be discussed in Chapter 7.

| Mistake type | Count | Description & Example |
|---|---|---|
| Learning mistake | 22 | Relatively simple example, yet still bad prediction<br>*Block 4 should be moved almost straight down until it is resting on block 5.* |
| More references | 15 | 2 or more references in the command<br>*Place block 20 parallel with 8 block slightly to the right of the 6 block.* |
| Grounding error | 13 | Unusual description of blocks<br>*Put the block that looks like a taurus symbol just above the bird.* |
| Missing reference | 11 | No reference block in the command<br>*Move the Stella block down to the very bottom of the square.* |
| Source same as reference | 11 | Network predicts the source as reference. Typically the last block mentioned in the sentence is source.<br>*Move block 10 above block 11, evenly aligned with 10 and slightly separated from the top edge of 10.* |
| Large direction | 10 | Distance between reference and gold location is more than 1 block.<br>*move the texaco block 5 block lengths above the BMW block* |
| Exchanged source and reference | 6 | First block in the sentence is reference followed by source.<br>*THE MCDONALD'S TILE SHOULD BE TO THE RIGHT OF THE BMW TILE. MOVE BMW* |
| Others | 12 | |

Table 5.3: Worst predictions analysis: Probable reasons behind bad predictions in 100 worst instances of development set.

Other problems such as "More references", "Missing reference", "Source same as reference" and "Large direction" have in common that the structure of the command is complicated, but the network can theoretically learn it. But in our opinion there are not enough examples of such commands in the data, which causes the high errors.

Some of the instances in the "Grounding error" category are in our opinion unsolvable without very specific rules, because the way in which the blocks are described is very unusual. Examples of such commands are:

> *Put the block with the spiral directly below the one with the circle and three lines.*
>
> *Move the Esso block left until it's in line with the delivery company block.*

However, some of the problems are caused by bad sentence preprocessing, which can be improved.

All the sentences with "Exchanged source and reference" problem are probably written by a single author, who has a specific way of giving instructions. Because of this and because of higher amount of wrong sentences caused by other problems, we decided to ignore this problem.

The "Other" category contains problems in which the command do not describe the action, some very long commands and problems where it is not clear what is the cause of bad prediction.

### 5.4.2 Source prediction error analysis

For the source prediction the results are better after removing world as input of the network. The accuracy on development set improved from 98.5% to 98.8%. This improvement just by removing part of input confirms our hypothesis, that the main problem of the previous architecture is overfitting.

As was written in Section 3.3 we test each setting three times with different random seeds. To better understand the source prediction, we analyzed the best of these three models on develoment set. It makes only 18 mistakes there. The sentence structure which causes most errors - seven - is the "Exchanged source and reference" problem which is described in Table 5.4.1 and which also causes problems to the location predictions. Other sentence structures where the network makes errors are:

- Three sentences with "switching":

  *The 16 and 17 block moved down a little but switched places.*

- Three sentences with typos (Note that the third word here should be *three*.):

  *slide block the to the space above block 4*

- Two sentences with sequence:

  *Continue 13, 14, 15...*

- Two sentences with Grounding error (see Table 5.4.1)
- One sentence which denoted a wrong block as a source.

The major improvement of the source accuracy could be achieved by solving the "Exchanged source and reference problem". However we cannot find any sentences with similar structure in the train dataset and since the structure is unusual, it is hard to come with solution. It is possible to generate commands with similar structure, but it would likely cause overfitting and not generalize at all. The other solution is having a model with deeper understanding of the commands, but it is unlikely to come with such a model with our dataset.

Similar case are the sentences with word *switch*, because this word appears only once in the train dataset.

Methods for typos corrections are already used when preprocessing the data for these models. The exact methods are described in next Chapter 6, but we do not think they can be significantly improved.

Overall, we think that for the source prediction we reached the limitations given by the dataset we are using and without usage of another data it is very hard to get significant improvements.

## 5.5 Removing feed-forward conversion layer

Based on the analysis in the previous Section 5.4.1 we decided to revise our architecture. We identified the usage of feed-forward layer to get the right dimension of the output as a possible cause of the inefficiency in the network. Thus we replaced the feed-forward layers by another recurrent layers. The architecture for location prediction is in Figure 5.5.

For the source prediction, the network is similar but it does not have the bidirectional layer with dimension 2 and the last states of the second bidirectional layer with dimension 20 are the final output - no multiplication with world is used.

### 5.5.1 Distinct predictions of block coordinates

We also try to work with the output layer. There are some command structures, which cannot be accurately represented by the representation described in Section 5.4. One example are sentences with distinct vertical and horizontal reference:

> *Move BMW so that it is vertically in line with Adidas, and horizontally in line with UPS.*

Here the best the model can do is to predict the location between the Adidas and the UPS blocks, which is not the correct solution.

We tried to solve this problem by using two weights instead of one for each block, one is multiplied by the block $x$ coordinate and the other by $y$ coordinate. Therefore the dimension of first output recurrent layer is 40, where the second output recurrent layer remains the same with dimension 2.

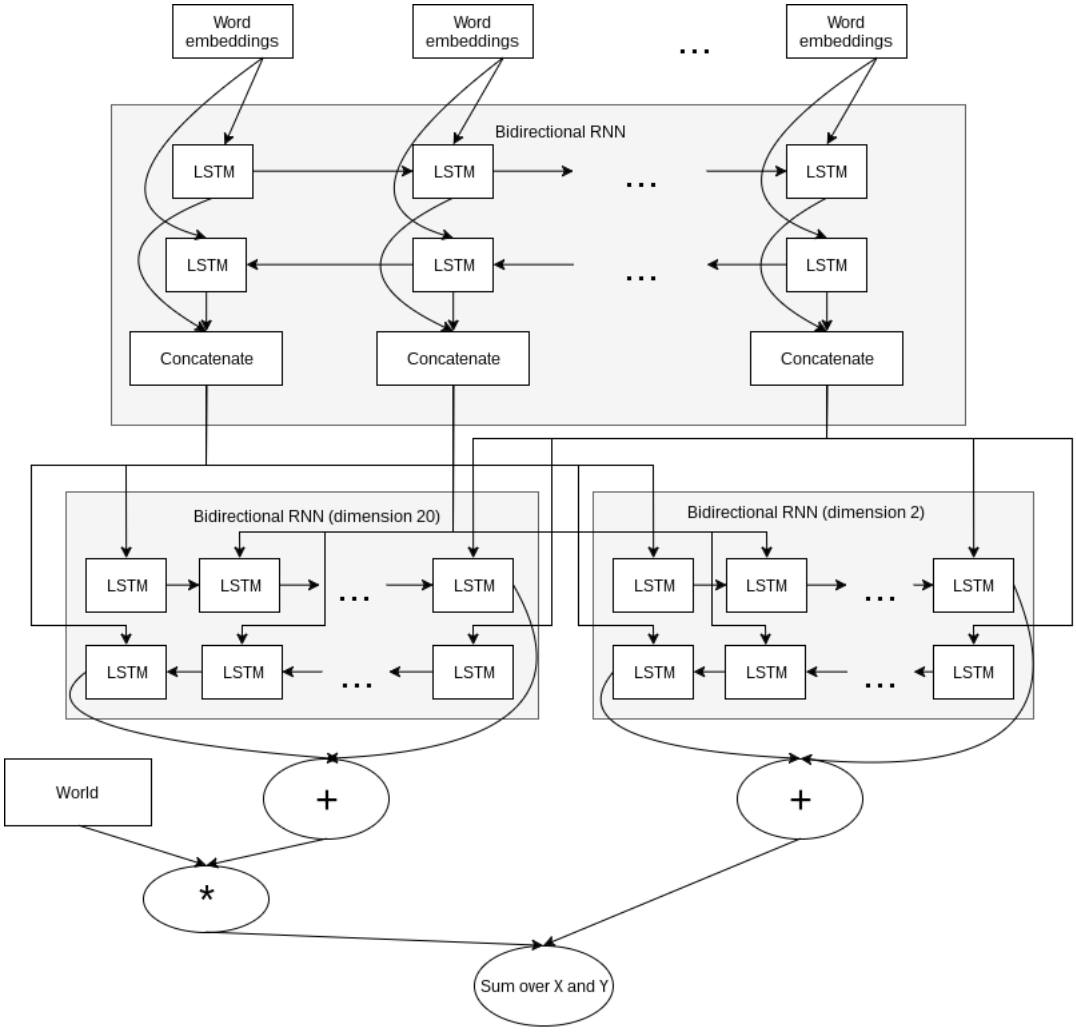


Figure 5.5: Using recurrent layer instead of feed-forward one. Arrows from bottom side of LSTM denote usage of LSTM output, arrows from left and right side denote usage of LSTM state.

### 5.5.2 Results

For predicting location the architecture with recurrent output layer is much better than the previous one. On the development set it has average distance between predicted and gold location 0.94. This is much better than the rule-based benchmark. The major difference between predictions of this architecture and the previous one with feed-forward layer (see 5.4) is that the previous architecture makes mistakes in relatively simple commands, which does not happen with the new one.

The distinct prediction of $x$ and $y$ coordinates works better then the simple prediction of one weight per block for commands with multiple references. But for simple commands it is worse. This leads to overall worse performance because there are more simple commands than commands with multiple references.

For source prediction the previous architecture with feed-forward layer is better, it has accuracy 98.8% and the one with recurrent layer only 98.4%.

All the results are in Table 5.5.2.

| | Source | Location |
|---|---|---|
| Random baseline | 5.8% | 5.96 |
| Deterministic baseline | 5.2% | 3.55 |
| Basic benchmark | 98.2% | 1.45 |
| Improved benchmark | 98.2% | 1.10 |
| Multiple world architecture | 97.8% | 3.96 |
| Single world architecture | 98.5% | 2.87 |
| Feed-forward output layer | 98.8% | 1.26 |
| Recurrent output layer | 98.4% | 0.94 |
| Distinct predictions | - | 0.98 |

Table 5.4: Results of models without any feedforward layers and their comparison with the previous models: "Recurrent output layer" refers to architecture on 5.5, "Distinct predictions" refers to the same architecture with the distinct prediction of $x$ and $y$ coordinates described in Section 5.5.1 and "FF output layer" is the architecture described in the Section 5.4.

## 5.6 Hyperparameters optimization

All the experiments in this section were conducted using our best architectures - for the source prediction "Feed-forward output layer" architecture was used and for the location prediction "Recurrent output layer" architecture was used. We did not use a grid search, when tuning one hyperparameter all the others were fixed to their best founded values. The best founded values are in Section 5.6.6.

### 5.6.1 Reccurent units

There are three widely used types of recurrent units: traditional non-gated recurrent units, Long short-term memory units (LSTM) and Gated recurrent units (GRU). It is unclear whether LSTM or GRU are better, but non-gated units generally perform worse (Chung et al. [2014]). Thus we tested both LSTM and GRU in our task.

We find out that for source prediction GRU units work better than LSTM. They reached accuracy 98.8% on development set when LSTM units have only 98.4% accuracy.

For location prediction it is the opposite. LSTM units are better with 0.94 average distance between correct and predicted location, compared to 1.23 average distance for GRU units.

### 5.6.2 Embeddings

We tried many variants of encoding the words in the command. We used one-hot encoding, randomly initialized trainable embeddings, two variants of character embeddings and pretrained embeddings. The formal description of these methods can be found in Section 2.6.

As for the pretrained embeddings we used Glove embeddings (Pennington et al. [2014]), because they are easily accessible[2]. These are general embeddings

[2]They can be downloaded from https://nlp.stanford.edu/projects/glove/

| | Source | Location |
|---|---|---|
| One-hot encoding | 98.0% | 0.94 |
| Random trainable embeddings (dimension 50) | 98.8% | 1.42 |
| Random trainable embeddings (dimension 500) | - | 2.71 |
| Trainable Glove embeddings | 98.7% | 1.76 |
| Static Glove embeddings | 97.6% | 2.38 |
| Character embeddings (GRU, single direction) | 98.4% | 1.32 |
| Character embeddings (LSTM, bidirectional) | 98.5% | 1.23 |

Table 5.5: Token encoding comparison on development set

trained on 6 billions of tokens from Wikipedia and news. They show state-of-the-art or near state-of-the-art performance on embeddings benchmarks. These are tasks such as word analogies, which is answering questions "$a$ is to $b$ as $c$ is to _?" (e.g. "Athens is to Greece as Berlin to _?") or word similarity (e.g. words "cash" and "money" should have similar embeddings).

The available embeddings have dimensions 50, 100, 200 and 300. Since our dataset is small, we used the dimension 50. We tried to use these embeddings in two different ways - with and without updating them during training our model.

We also use two variants of character embeddings. One bidirectional with LSTM units, because this is the model that Ling et al. [2015] used to achieve state-of-the-art results in part-of-speech tagging (basically predicting whether word is noun, verb, ...) on English. Second setting uses only single forward direction and GRU units.

It should be noted that all these settings except of the character embeddings are tested with data preprocessed by spellchecker. We do not use it for character embeddings, because we think that the model with character embeddings can learn to deal with spelling errors. Error in a single letter changes just one of the inputs of the embedding layer whose result shouldn't be very different.

The results of experiments using different embeddings are in Table 5.6.2. The most interesting findings are the differences between what works for the source prediction and what works for the location prediction. The one-hot encoding has the best results for the location prediction but for the source prediction it has the second worst results. On the other hand the random trainable embeddings with dimension 50 has the best results for the source prediction but don't work for the location prediction.

We thought that the reason why one-hot encoding works much better than other embeddings for location prediction is the different dimension of vectors encoding single token. With one-hot encoding each token is encoded as vector of size $|V|$ which is between 500 and 1000 depending on the spell checker used, but with the other encodings each word is represented as vector of dimension 50. Because of this we tried random embeddings with dimension 500, but they do not work at all. The optimizer is unable to lower the error on training set under the distance 2.5.

After seeing this result, we also inspected the errors on training sets for other embedding variants. It is the case that for one-hot encoding the optimizer is able to lower error on training set under 0.3, but it is unable to do so for other embedding variants. For character embeddings with LSTM the training error

| | Source | Location |
|---|---|---|
| No dropout | 97.8% | 0.94 |
| Dropout input 0.5 | 98.6% | 3.11 |
| Dropout hidden 0.1 | 97.5% | 0.95 |
| Dropout hidden 0.2 | 97.7% | 0.96 |
| Dropout hidden 0.5 | 98.0% | 1.00 |
| Dropout input 0.5 + hidden 0.5 | 98.8% | 3.10 |

Table 5.6: Dropout comparison

goes as low as 0.9. For other embeddings the training error is even worse, with the training error being approximately proportional to the validation error in Table 5.6.2.

It is also obvious from the results that the pretrained embeddings do not work. The trainable variant, which worked, is essentially equivalent to trainable random embeddings, thus its success for the source prediction cannot be interpreted as success of the pretrained embeddings. The reason why they do not work is probably that the embeddings are not designed for our task. Words which have very different meaning in our task often have similar embeddings, because they are in the same category of words. For example embeddings of numerals are very similar, but we have to distinguish them efficiently in our task. Also embeddings of *west* and *east* are much closer to each other than *west* and *left*.

### 5.6.3 Dropout

We applied dropout to both the input layer (embeddings) and the hidden layer. For the location prediction we used the dropout as described in Section 2.5. For the source prediction we used slightly modified version of dropout where during testing we use all the neurons but do not multiply their outputs by any $1 - p$, where $p$ is the probability of not using the neuron. We use this version, because it has slightly better results on our dataset than the regular dropout for the source prediction.

The results of models with different dropout setting are in Table 5.6.3. The models for source and location prediction reacts to dropout differently. The source prediction model achieves best results with dropout 0.5 applied to both input and hidden layer.

The location prediction model on the other hand performs best without any dropout. The reason why dropout in input layer decreases performance of location prediction model is that for this model we are using one-hot encoding. With this encoding and 0.5 dropout the model in 50% cases receives vector of only zeros instead of the one-hot encoded word.

Hidden layer dropout might decrease the performance of location prediction model because it makes optimizing the weights more difficult. But it improves the performance of source prediction model, because optimization in this case is likely not a problem.

| Unit dimension | Source | Location |
|---|---|---|
| 32 | 98.7% | 0.99 |
| 64 | 98.8% | 0.95 |
| 128 | 98.5% | 0.94 |
| 256 | 98.6% | 0.95 |

Table 5.7: Comparison of recurrent units dimensions

| Hidden layers | Source | Location |
|---|---|---|
| 1 hidden layer | 98.8% | 0.94 |
| 2 hidden layers | 98.6% | 0.93 |
| 3 hidden layers | 98.6 % | 1.53 |

Table 5.8: Results of additional layers

### 5.6.4 Recurrent unit dimension

We tested how the dimension of recurrent units influences the model performance. We tried four different settings: recurrent units with dimension 32, 64, 128 and 256. We found out that the model for source prediction works better with lower dimension 32 and 64, when the model for location prediction needs higher dimension. Based on the results in table 5.6.4 we are using dimension 64 for source prediction and 128 for location prediction.

### 5.6.5 Adding layers

In many tasks deeper networks offer better performance (He et al. [2016], Wu et al. [2016]).

All the networks presented until now had only a single hidden layer. Here we try networks with more hidden layers. The new layers are exactly the same as the original hidden layer, therefore for example dropout is used on the additional layers in the source model. They have recurrent architecture and use all the inputs from the previous layer, which is also recurrent. All the layers are bidirectional, but the two outputs from forward and backward run are concatenated and both given to the forward and backward cell of the next layer, the forward and the backward directions are not isolated. Precisely given a sequence of length $l$, the outputs of the first forward layer $f_{1,1}, f_{1,2}, ..., f_{1,l}$ and the outputs of the first backward layer $b_{1,1}, b_{1,2}, ...b_{1,l}$ are first concatenated: $c_{1,1} = f_{1,1}b_{1,1}, c_{1,2} = f_{1,2}b_{1,2}..., c_{1,l} = f_{1,l}b_{1,l}$. Then both the first LSTM cell of forward layer and the last LSTM cell of backward layer are given $c_{1,1}$ as their input. More generally, the $i$-th forward cell and the $l - i$ backward cell of $j$-th layer are given $c_{j-1,i}$ as input and produce $f_{j,i}$ and $b_{j,i}$ respectively as output. For better understanding see Figure 5.5, where additional layers can be added by simply stacking the "Bidirectional RNN" box multiple times under the original one.

The results on development set of models with different number of layers is in 5.6.5. As in all other results the numbers in the table are averages over three runs. The three different runs of the model with 3 hidden layers have very different results. Two runs have the average distance between the correct and the

predicted location 0.98 and 1.08, while last run had distance 2.52. Because of the learning curves (see Figure 5.6.5), we thought that the bad result can be caused by exploding gradient.

To prevent it, we introduced model with gradient clipping. We changed each element of gradient vector lower than -1 to -1 and similarly each element greater than 1 to 1. But the result of deep networks with gradient clipping are similar to the ones without it - for example the 3 hidden layer network with gradient clipping has average location distance 1.72.

This suggest that there might be other optimization problems or that the clipping thresholds -1 and 1 are not chosen well.

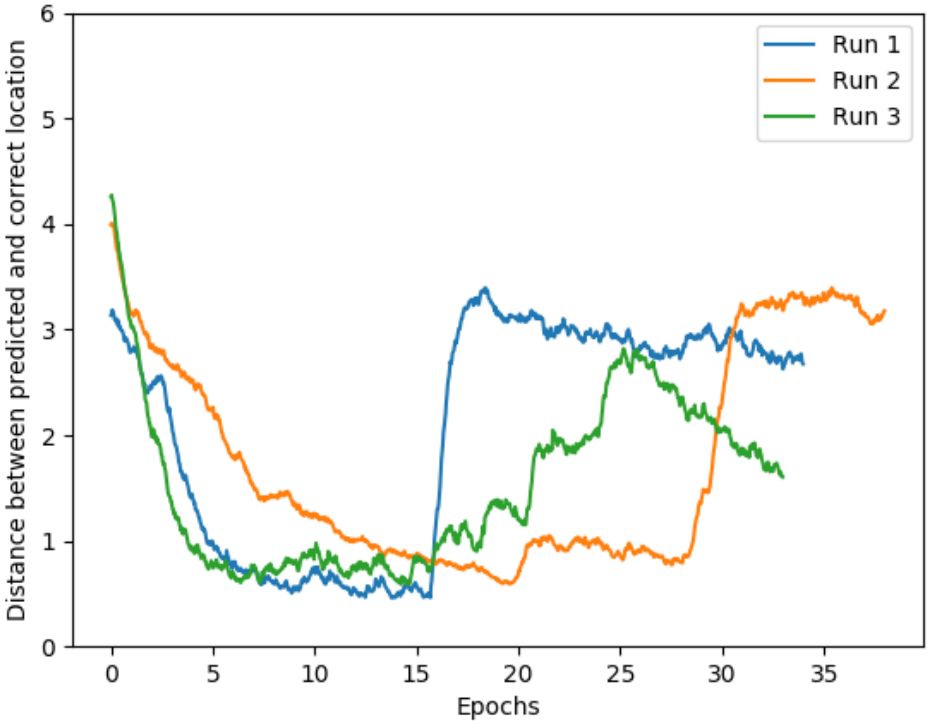


Figure 5.6: Exploding gradient problem - performance on training data of three different runs of model with 4 layers.

### 5.6.6 Used hyperparameters

Unless stated otherwise, we used for all the source prediction experiments following hyperparameters: GRU with dimension 64, random trainable embeddings, dropout 0.5 on both input and a single hidden layer. As for the location prediction we used: LSTM with dimension 128, one-hot encoding, no dropout and also a single hidden layer.

In all the experiments, we used batch size 8 and Adam optimizer. We did not try tuning these.

As for the learning rate we used 0.001 for the source prediction, while for the location prediction we used learning rate 0.01. These values were found by

a manual hyperparameter search. But we did this search in the very beginning of our experiments with a different architecture than the one used when searching for other hyperparameters and also with different method of conducting the experiments (e.g. using single run instead of average of three runs).

# 6. Advanced data preprocessing

## 6.1 Correcting errors and word normalization

At first we simply tried to create vocabulary directly from tokens without any error correction and feed the encoded sentences to the model, which worked without problems. But we saw that there was a space for improvement because of high amount of typos in the tokens.

This leads to having more tokens in vocabulary which all represent single word misspelled in many different ways. For example the word *"Mercedes"* was spelled correctly only in 94.35% cases (Table 6.1) and even if we allow the *"Mercedez"* variant then there is 1.75% of mistakes. And each of these mistakes very likely causes error in predicting source block. Given that Bisk et al. [2016b] reached accuracy of 98% in predicting the source block without any error correction we can conclude that correcting typos is important part of the task, because it probably causes large amount of mistakes.

### 6.1.1 Levenshtein distance

Levenshtein distance between two strings is the minimal number of operations "remove character", "add character" and "change character" needed to get one string from the other. Typos can be also viewed as these operations, which makes Levenshtein distance usable for error correction.

If we know that some word is spelled wrong, we could correct it by changing it to the another correct word, such that their Levenshtein distance is minimal.

The problem with this approach is that we often do not know if a word is correct or not. So as a first attempt of error correction we tried to correct only words whose Levenshtein distance to logo names were 1, because logo names are correct and important words and because we think that it is unlikely that another correct word has Levenshtein distance only 1 to logo name. This does not work very well, because it corrects only small fraction of words and because we do not know whether the word which we are changing is really wrong.

| | | |
|---|---|---|
| mercedes | 647 | 94.35% |
| mercedez | 26 | 3.80% |
| merceds | 3 | 0.44% |
| mecedes | 3 | 0.44% |
| merecedes | 2 | 0.29% |
| mecerdes | 1 | 0.15% |
| mnercedes | 1 | 0.15% |
| merces | 1 | 0.15% |
| nercedes | 1 | 0.15% |

Table 6.1: Variants of 'Mercedes' token

### 6.1.2 Hunspell

Hunspell[1] is a widely used spellchecking software, which comes with already created dictionary for many languages including English. Given a word it can return whether the word is in its dictionary and also a list of suggestions how to correct spelling of a given word. The suggestions are computed based on difference between the original word and the suggestion and on the number of occurrences of the suggestion in the text from which the dictionary was created. We used it to correct spelling of all words which were not correct according to its dictionary. The problem is, that the dictionary was created on general language data which does not contain words often used in our text, especially the logo names. Fortunately, it is possible to manually add words to the created dictionary. We add all the logo names, since they are important for our task.

This works, however there is a theoretical problem with this approach. The distribution of words in our text is very different then the distribution of words on texts on which Hunspell is trained. For example our data contains only approximately 1000 tokens but in general English there are orders of magnitude more of them. This lead to situations where Hunspell suggests a correction which never appeared anywhere in our text and thus it is often incorrect.

It is not possible to train Hunspell on our data since we have no golden data without any mistakes, so we try the following approach.

### 6.1.3 Using task-specific data

We can use our task-specific data with mistakes to select the most probable correction:

**Algorithm 5** Error correction using task-specific data

1: **procedure** *correct*(Word $w$, Text $t$)
2: $A \leftarrow \{a|\text{levenshtein\_distance}(w, a) \leq 1\}$
3: **return** $a \in A$ such that number of occurrences of $a$ in $t$ is maximal
4: **end procedure**

Note that we correct only words with less than 30 occurrences in our text, because it is unlikely that a single word is spelled in the same incorrect way more times.

The reasoning behind this algorithm is that given a correct word, it is unlikely that there is another more frequent word with Levenshtein distance = 1, so the correct word likely stay the same. If the word is incorrect, then probably the correct variant appeared in text more times then the incorrect one, so the word is changed. And it is likely changed the right way, because it is more probable that it is a variant of a more frequent word than the less frequent word.

We tried this algorithm, but we saw that it can be improved, because it changes too many already correct words. We decided to change how $A$ is generated. It is possible to use Hunspell which returns a set of alternative spellings instead of Levenshtein distance. Based on our subjective analysis, this variant works well. It is able to correct even words when the Levenshtein distance from

[1] http://hunspell.github.io/

the correct word equals two (for example: *merces* is corrected to *mercedes*). And it changes word incorrectly only in the hard cases, where the context is needed for the right decision. For example in the sentence:

*Move SRI so it si to the right of Esso*

our algorithm changes *si* to *so*, while the right correction is probably *is*.

The only problem with this algorithm is that some words are not corrected, for example *witter* is not corrected to *Twitter* and *tow* to *two*.

### 6.1.4 Synonyms and lemmatization

Besides the spelling errors in the sentences, another big problem are out-of-vocabulary words and words with a very low number of occurrences. It is almost impossible for model to learn their meaning, because they appeared in the data only few times or not at all. Thus if they appear both in the testing and training data, they will be ignored during testing or worse they will cause overfitting and decrease the model performance.

To solve these problems we tried to encode the less frequent words in the same way as some other words if possible. To do this we used two approaches: lemmatization and dictionary of synonyms.

Lemmatization is a process where different forms of some word are all changed to the same basic form called lemma. For example the word forms *write, writes, wrote, written* would all be changed to lemma *write*. Given the lemma of each token, we can than encode in the same way all the tokens with the same lemma. With this encoding, rare words which do not have rare lemma are encoded as the lemma and thus are not rare anymore. The problem with this encoding is, that we are losing the information about the exact form of the word. Because of this, we decided to encode only groups of forms where at most one is not rare. We set the rare threshold to 30 occurrences.

For example, suppose that the word "*write*" has 500 occurrences, the word "*wrote*" 300 and the word "*writes*" 20 occurrences. Then "*writes*" is rare and is encoded the same way as "*write*", but "*wrote*" is not rare and is encoded in a different way.

With this rule, we are losing some information only about rare words, which is not a problem, because without this encoding we would probably lose all the information of such words.

The lemmas, which we use here, are generated by UDPipe (Straka et al. [2016]).

We also use dictionary with synonyms from Wordnet (Oram [2001]). We use it in similar way as the lemmatization - words which are synonyms according to the dictionary are encoded in the same way. The reasoning behind it is that rare words which are synonyms to some unrare words are no longer rare with this encoding.

To compute encodings with synonyms and lemmas we change the "Error correction algorithm 6.1.3" in such way that the set of alternatives $A$ contains also synonyms and lemma of the given token. By this approach if there are more ways how to encode some word, for example if word might be misspelled or it might be unusual synonym to some other word, the most frequent variant in our data is chosen, which should be the most probable option.

## 6.2 Other preprocessing

By analysing the results of data preprocessing we found out that the previously described architecture have problem with one specific error - missing space between words. Hunspell can often recognize this type of error and suggest dividing the token into two words. But because the Error correction algorithm 6.1.3 needs already tokenized text, it is to late to further divide tokens when Hunspell is used. Because of this we decided to use Hunspell twice: first time during the tokenization when the words are corrected only if the suggestion is to divide them into two tokens and second time to correct all the other mistakes.

Some tokens were in the data just once or few times, even after using spellchecking, synonyms and lemmas. We thought that the network cannot understand the word from a single appearance and would overfit because of these words. Therefore we decided to replace all these words by a single token "*<unk>*" which would represent unknown words. The out-of-vocabulary words are represented as this token "*<unk>*" also during the testing.

We also think that some tokens are useless in the prediction and remove them. Specifically, we ignore tokens *"the", "to", "so", "s", "a", "e", ".", ","*. The *"s", "e"* tokens appear mainly as a result of other preprocessing methods.

## 6.3 Analysis

For comparing different preprocessing strategies we use two models: rule-based benchmark model and our best architecture neural model with tuned hyperparameters. In short the rule-based benchmark model identifies words denoting block names (*BMW*) and directions (*left*). It predicts the first block name in the command as the source and the location of the last block name in the command as the location possibly moved by one if a direction word is present. Detailed description is in Section 5.2.

Results of the benchmark model are useful because they are relatively easy to interpret and because the benchmark model is deterministic. On the other hand, results of the neural model are stochastic - they depend on random initialization of the weights - and therefore it is possible that worse preprocessing looks better just by chance. But in the end we want to know which preprocessing works for neural networks, which is why we used this model for comparing preprocessing too.

Also note, that results in this section are measured on the development dataset not on the test set. Because of differences between these two sets, the results of neural model when predicting location are much better on the test set (approximately by 0.15) and similar when predicting the source, while the benchmark is worse on the test set (approximately 2% worse for source and 0.08 worse for location).

All the preprocessing strategies are listed in Table 6.3.

The most obvious observation is that using Levenshtein distance for the spellchecking does not work. By looking at the results of benchmark model, we can also see that Hunspell works only if it is used inside the proposed Algorithm 6.1.3 for error correction, at least when correcting block names. Then it is quite effective - improvement from 97% to 98.2% is by more than one third.

| Error correction | Source | | Location | |
|---|---|---|---|---|
| | Neural | Benchmark | Neural | Benchmark |
| Only tokenization | 98.3% | 97.0% | 0.97 | 1.19 |
| Hunspell | 98.5% | 98.2% | 0.95 | 1.10 |
| Hunspell, lemmatization | 98.5% | 96.9% | 0.94 | 1.16 |
| Hunspell first choice | 98.2% | 97.0% | 0.96 | 1.19 |
| Hunspell, synonyms | 98.5% | 98.1% | 0.95 | 1.15 |
| Hunspell, ignore tokens | 98.4% | 98.2% | 0.95 | 1.10 |
| Hunspell, levenshtein | 71.6% | 48.2% | 1.76 | 3.27 |
| Hunspell, unknown 4 | 98.8% | 98.2% | 0.94 | 1.10 |
| Hunspell, unknown 6 | 98.6% | 98.2% | 0.95 | 1.10 |
| Hunspell, unknown 8 | 98.5% | 98.2% | 0.91 | 1.10 |
| Hunspell, unknown 10 | 98.5% | 98.2% | 0.94 | 1.10 |
| Hunspell, unknown 12 | 98.5% | 98.1% | 0.94 | 1.10 |
| Hunspell, unknown 20 | 98.4% | 98.1% | 0.93 | 1.10 |
| Combination 1 | 98.6% | 96.9% | 0.96 | 1.16 |
| Combination 2 | 98.6% | 96.8% | 0.94 | 1.21 |

Table 6.2: Preprocessing method comparison: Usage of synonyms and lemmatization was described in Section 6.1.4, ignoring tokens was described in Section 6.2, Hunspell first choice refers to Hunspell without the usage of our task-specific data, unknown $N$ means that all tokens with less than $N$ occurrences are replaced with $<unk>$ token. Note that default $N$ was 2. Combination 1 uses Hunspell, token ignoring, lemmatization and unknown 4. Combination 2 is the same but it also uses synonyms. All results use rule-based tokenization and are measured on development data.

Lemmatization is decreasing the performance of the benchmark model. The reason is that the UDPipe in some cases lemmatizes *UPS* as *up*, which the benchmark of course does not recognize as a word describing block. Ignoring rare tokens have almost no effect on benchmark, probably because it uses almost only the frequent tokens.

Based on these results on the development set, we tested the benchmark model on the test set without error correction and with Hunspell. Without error correction the model has accuracy 94.8% for predicting source and distance of 1.27 for predicting location. With Hunspell and our error correction algorithm which uses the task specific data the accuracy is 96.3% and the distance is 1.20.

The results of neural model are not affected by preprocessing as much as the results of the benchmark model. The neural model can understand the sentence even if there are some mistakes in it.

For example, unlike the benchmark, the neural model performance is not decreased by the lemmatization. It is able to learn that the same token *up* sometimes represents *UPS* and sometimes the direction. On the 1719 instances of development set, out of which 27 have the correct source *UPS*, the model for source predictions makes only 2 mistakes related to this block.

Still, better error correction improves the performance of the neural model. Except of Levenshtein distance, worst results of neural model appears without any error correction or if the task-specific data are not used. Also relatively bad result of distance 0.96 appears for Combination 1, but this might have happened just by chance, because the accuracy is very good for the source prediction.

The accuracy of the source prediction model on preprocessing with replacing rare words by single *<unk>* token (rows with "unknown N") decreases when more words are considered rare. Based on this, we might conclude that it can learn some useful information even from words with as few as 4 appearances in the data. On the other hand the location prediction model is not worsened, so it is probably not able to use the rare words.

For the rare word limit 8 the location model has very good result of 0.91. But we think this happened just by chance, since results for limit 6 and 10 are not very good.

Therefore we decided to use the Hunspell with our task specific data and rare word limit 4 as the default preprocessing. Unless written otherwise this preprocessing is used for all our experiments, including the one previously described in Chapter 5.

## 6.4 Additional information

### 6.4.1 Tags and logos

Using the UDpipe (Straka et al. [2016]) analysis, we have also other information available besides the word forms and lemmas. One of them are part-of-speech tags, which can be generated by UDPipe. We can also work with information whether the particular sentence contains blocks with logos or with digits.

Universal part-of-speech tags, which we are using, are categories of words which are similar to word classes (i.e. noun, adjective, verb, ...). There are 17

| | Source | Location |
|---|---|---|
| No additional information | 98.8% | 0.94 |
| Tags | 98.7% | 0.93 |
| Tags one-hot | - | 0.95 |
| Logos | 98.6% | 0.94 |

Table 6.3: Results of models with additional information on development set

classes. Besides the standard ones, there are also classes for example for symbols and punctuation.[2] Therefore we can have one tag for each token.

We tried to append both the tags and logos/digits information to embeddings of each token. Tags are represented by randomly initialized trainable embeddings of dimension 10 and information whether logos are used is represented as single boolean value. Because one hot encoding is better than embeddings for location model (see 5.6.2), we also tested tags encoded by one hot encoding. But their results are slightly worse (0.95).

Results of models with and without this information are in Table 6.4.1. The differences are small and might have happened just by chance.

### 6.4.2 Language parser

We wanted to use language parser for improving the benchmark model. Since we were already using UDPipe, which also contains language parser, we decided to test it. Unfortunately the parser do not work very well for our sentences. The reasons were probably following:

- Logos are unusual words.
- There are imperatives, whose the parser cannot handle.

Because of such a bad performance on our data, we did not implement any model which would use the parser outputs.

### 6.4.3 Predicted source as input of location model

When analysing the results of location predictions (see Section 5.4.1 notably Table 5.4.1), we find out that in some cases the network incorrectly uses source block as the reference. Since the model for predicting source is very accurate we speculated that this mistake can be avoided by using the source block predicted by the source model as another input for the location model.

Our best source model with test accuracy 98.8% trained on the training data is used to generate predictions for all the data - training, development and test set. we have many possibilities how to encode this information. We can use simple one-hot encoding for numbers 1-20 denoting which block is the source in the command. Given a command with logos we use the same encoding, but the first element of the one-hot vector indicates the *Adidas* block, the second element the *BMW* block and so on. But since we use recurrent networks it would be

[2]See http://universaldependencies.org/u/pos/index.html for details.

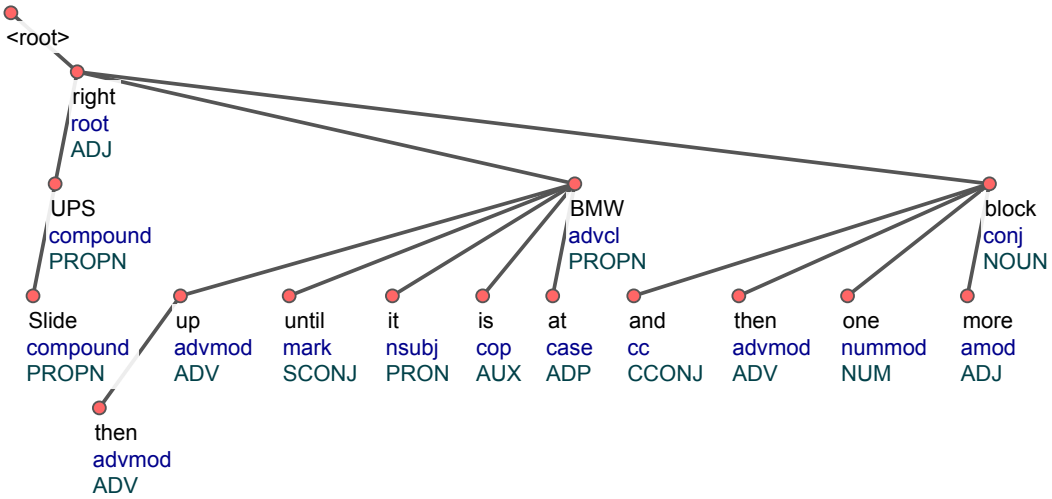


Figure 6.1: Parse tree of command *Slide UPS right then up until it is at BMW and then one more block* generated by UDPipe.

necessary to append this information behind every word, or somehow encoded it in the initial state of the recurrent units. This is not an elegant approach.

Better approach to append single bit of information to each word embedding signifying, whether this word is the source in the current command. With this approach the network can very easily learn to ignore source blocks when computing the location. However, we have to somehow identify words denoting blocks. For this we use the benchmark model.

This information improved the average distance between correct and predicted location on development set from 0.94 to 0.89. This improvement happens almost only on commands where at least one block is mentioned multiple times (see Section 3.1 for example and description of this type of commands).

## 6.5 Data augmentation

### 6.5.1 Generating new commands

When analyzing the predictions of our models, we find out that they perform badly on some types of commands, such as commands with more than one reference (see Table 5.4.1 for details). We also find out that there are not many training commands of these types in the training data, which may be causing the bad performance. Therefore we decided to generate new training commands.

The generator use sentence skeletons, which are sentences where some words are replaced by another words from a specified set. Then, during sentence generation, one of the words from the set is chosen randomly to form the final sentence. For example we have following sentence skeletons:

> ***move block Adidas 1 spaces up*** *and* ***2 spaces left***

> ***move block Adidas so*** *it is horizontally* ***aligned*** *with* ***block BMW*** *and vertically* ***aligned*** *with* ***block Coca-Cola***

The bold words can be replaced by similar words, phrases or in some cases removed from the sentence. For example following sentences can be generated from them:

*Put cube Target 3 block lenghts up and 4 spaces right*

*Position block Toyota such that it is horizontally aligned with block Mercedez Benz and vertically in line with block Adidas*

*Place block thirteen so it is horizontally in line with cube 15 and vertically in line with 3*

The generated sentences are not always grammatically correct but in our opinion there are even worse sentences in the training data.

We also randomly generate world state and correct prediction of both location and source for each generated sentences. Given that we know the meaning of the skeleton and which words are used there, it is simple to generate these. To prevent invalid instances, the generator also enforces some simple rules:

- Each block appears only once in single sentence.
- Positions of the blocks do not collide with each other.
- The correct location do not collide with blocks positions.
- Blocks in one sentence are never denoted with both logos and numbers.
- The predicted location (and the blocks obviously) are within bounds of the board.

The generated sentences focused mainly on commands, where first block is not source and where location is described using 2 references, relative distance from source and absolute position on board.

We than tested location prediction model trained on the original 11,871 training commands and 5,000 generated commands. This model is with 1.01 development error worse than the original one, which had 0.94 development error.

The reason for this might be that the generated commands are too hard for the model, because the training error on the set of generated commands is 0.97 which is much more than 0.25 on the original training set.

We tried to make the generated commands simpler by using digits (i.e. *1, 2, 3*) to denote blocks and not distances and numerals (i.e. *one, two*) for distances but not for blocks. After this change the model has 0.97 development error, which is a small improvement.

### 6.5.2 Block swapping

It is possible to generate new commands similar to the existing ones by permuting block coordinates and blocks names in the command the same way. Given a permutation $p_1, ..., p_20$ of numbers $1, ..., 20$, we can substitute words denoting $i$-th block by the word denoting $p_i$-th block and coordinates of $i$-th block by coordinates of $p_i$-th block. For example if $p_1 = 10$, $p_2 = 11$ and we have command

"*Move block 1 to the left of block 2*", this method gives us sentence "*Move block 10 to the left of block 11*". In our world the 10th block will be in the original place of 1st block, 11th in the place of 2nd block and similarly for the other blocks.

Since the source prediction accuracy is good even without block swapping, we tested it only for location prediction. However, the model performance remains similar, the development error is 0.95 using the swapping and 0.94 without it. But on the training set the models with block swapping have higher training error (approximately 0.4 vs 0.25).

This can be caused by the worse optimization or by some errors in our block swapping method. The method relies on detecting all words denoting blocks, which is done by simply searching the logo worlds in the sentence. But this method does not find some unusual words describing blocks such as *coke* instead of *Coca-Cola* or *circle with three lines* which should mean *Mercedez Benz*. If these words appear in the command, the corresponding block is swapped with another block, but the word remains the same, which causes generation of erroneous commands.

We think there are approximately 50-100 commands with these words, which is 0.5%-1% of the training dataset.

# 7. Evaluation and discussion

## 7.1 Objective evaluation

Here we present results of some of our models on the test set. In contrast with the previous chapters, where we averaged performance of three runs to evaluate architectures, hyperparameters or other settings, in this chapter we are selecting only the single run with best performance on development set.

The results of source prediction models are in Table 7.1, models for predicting location are in Table 7.1.

There are large differences between development and test results in both these tables. This is caused by differences in the two datasets. It was already discussed in Chapter 3 and notably in Table 3 from which we can see that the average length of command in the train set is 14.86, in the test set it is 15.07 but in the development set it is 17.91. Also 26.6% of the development dataset are the commands of the most complicated category "More references" (see Section 3.1 for more information about this category), when only 15.5% of the test set and 13.9% of the train set fall into this category. Thus we can conclude that the test set is simpler than the development set, especially for the location prediction.

However we can see that the benchmark model performed better on the development set, both for the source and the location prediction. This happened because we tuned the benchmark model on the development set and therefore it overfits this dataset.

For the source prediction our benchmark achieves accuracy 96.3% which is better than the feed-forward network implemented by Bisk et al. [2016b], but worse than his recurrent network.

As for our neural networks, it is unclear whether our "Single world architecture" network is better then the recurrent model of Bisk et al. [2016b]. They reported accuracy of 98% for their best recurrent network. Unfortunately it is unclear how this number was rounded, therefore it is possible that their network is better than our "Single world architecture" network with 98.2% accuracy. But our best source prediction network is clearly better with prediction accuracy of 98.8%.

When analyzing our location prediction results we find out, that the average

| | Development | Test |
|---|---|---|
| Random baseline | 5.8% | 5.1% |
| Deterministic baseline | 5.2% | 5.7% |
| Basic benchmark | 98.2% | 96.3% |
| Improved benchmark | 98.2% | 96.3% |
| Bisk et al. [2016b] FFN | - | 93% |
| Bisk et al. [2016b] RNN | - | 98% |
| Single world architecture | 98.5% | 98.2% |
| Feed-forward output layer | 99.0% | 98.8% |
| Recurrent output layer | 98.6% | 98.6% |

Table 7.1: Final source prediction results

| | Development | Test | Correct |
|---|---|---|---|
| Random baseline | 5.96 | 5.92 | 0.9% |
| Deterministic baseline | 3.55 | 3.44 | 1.4% |
| Basic benchmark | 1.45 | 1.54 | 35.0% |
| Improved benchmark | 1.10 | 1.20 | 46.2% |
| Bisk et al. [2016b] FFN | - | 1.81 | - |
| Bisk et al. [2016b] RNN | - | 0.98 | - |
| Multiple world architecture | 3.77 | 3.79 | 2.5% |
| Single world architecture | 2.85 | 2.84 | 5.9% |
| Feed-forward output layer | 1.24 | 1.08 | 49.0% |
| Recurrent output layer | 0.93 | 0.73 | 66.6% |
| 2 hidden layers, source flags | 0.87 | 0.71 | 68.9% |
| Human | - | 0.53 | - |

Table 7.2: Final location prediction results. The "2 hidden layers, source flags" row refers to model similar to Recurrent output layer model (see Figure 5.5) with one more hidden layer and with usage of source predicted by source prediction model. The row "Bisk et al. [2016b] FFN and RNN" refers to the best feed-forward and recurrent model in the article by Bisk et al. [2016b]. Human performance was measured by Bisk et al. [2016b]. All other models were described in Chapter 5. Column "Development" and "Test" contain average Euclidean distance between predicted and correct location on these datasets, "Correct" column contains percentage of predictions with Chebyshev distance less than 0.5468 from correct location.

Euclidean distance between predicted and correct location is not very accurate metric, because very small number of badly predicted instances has very big influence on the average distance. Thus we decided to measure another performance metric for our location prediction models - percentage of instances, where the Chebyshev distance[1] between correct and predicted location is less than $0.5 * (block\_edge + distance\_between\_blocks)$. The $block\_edge = 1$ is the length of block side and $distance\_between\_blocks = 0.0936$ is the distance between two adjacent blocks in the final world state. Thus our proposed metric measures the percentage of instances with Chebyshev distance less than 0.5 * 1.0936 = 0.5468.

In our opinion this metric is more stable and is easily interpretable. After rounding the predicted location to multiples of 1.0936, the instances with Chebyshev distance less than 0.5468 are in the correct position, while those with higher distances are not. Also with this metric the problem can be easily reframed as classification problem, when each square on the board will be represented by one class. The results of our models measured by this metric and by the standard average distance are in Table 7.1.

From the table we can see that the location prediction models with world on input - "Multiple world architecture" and "Single world architecture" - are about as good as the baselines, which is very bad result. Our rule-based benchmark performs much better. On the test set both the "Improved benchmark" and "Basic benchmark" models with 1.20 and 1.54 average distance between correct

[1]Chebyshev distance is defined as $D_{Chebyshev}(p, q) = max(|p_i - q_i|)$.

and predicted location outperforms the best feed-forward network of Bisk et al. [2016b].

Our neural architectures without world on input are better than the rule-based models and also better than the ones which uses the world state. The "Feed-forward output layer" architecture reaches average distance of 1.08 on the test set.

For "Recurrent output layer" architecture, which is our best location prediction architecture , we tested two different settings. The first one has a single hidden layer and no input other than the command and can be seen in Figure 5.5. This one has the average distance between correct and predicted location 0.73. The second one is similar but has one more hidden layer and also uses source predicted by the source prediction model (see Section 6.4.3). This network achieves average distance of 0.71 on test set, which is much better than the best model of Bisk et al. [2016b] with 0.98. Considering that human performance as measured by Bisk et al. [2016b] is 0.53, we more than halved the previous difference between the previous state-of-the-art model and the human performance.

## 7.2 Source prediction analysis

We manually analyzed bad predictions of our best source model. There were only 18 mistakes made on the development set:

- Two-sentence commands (7 mistakes). In the first sentence, it looks like the first mentioned block is the source, but the second sentence states otherwise.[2] *"The McDonald's tile should be to the right of the BMW tile. Move BMW."*
- Block switching (3 mistakes): *"The 16 and 17 block moved down a little but switched places."*
- Commands with typos (3 mistakes): *"Slide block the to the space above block 4"* (Note that the third word here should be *three*.)
- Commands including a sequence (2 mistakes): *"Continue 13, 14, 15..."*
- Grounding error (2 mistakes), see Table 5.4.1.
- Annotation error (once, not a mistake).

Major improvement of the source accuracy may be achieved by solving the problem where second sentence changes the meaning of the first one. However, there are no similar commands in the training data, so it is hard to come up with a solution. Similarly, the word *switch* appears only once in the training set.

Methods for typos corrections are already used when preprocessing the data for these models.

Overall we think that for the source prediction we reached the limitations given by the dataset we are using and without usage of another data it is very hard to get significant improvements.

[2] All these seven sentences were likely written by single author.

| Category | Commands | Neural | | Benchmark | |
|---|---|---|---|---|---|
| | | Distance | Correct | Distance | Correct |
| Source, reference, direction | 1312 | 0.16 | 91.0% | 0.48 | 84.5% |
| Source, reference, two directions | 483 | 0.36 | 82.0% | 0.90 | 36.6% |
| Source, reference, more directions | 127 | 0.64 | 77.2% | 1.32 | 11.8% |
| Source, reference, no direction | 64 | 1.06 | 42.2% | 1.55 | 0.0% |
| Source, reference, distance | 418 | 1.29 | 43.1% | 2.17 | 6.2% |
| Block mentioned multiple times | 207 | 0.68 | 71.0% | 1.31 | 29.5% |
| More references | 492 | 1.77 | 26.4% | 2.04 | 15.6% |
| No reference | 74 | 2.41 | 20.3% | 4.10 | 1.3% |
| **Overall** | 3177 | 0.71 | 68.9% | 1.20 | 46.2% |

Table 7.3: Location prediction on individual command categories. In the "Neural" columns there are the results of our best neural network model, in the "Benchmark" columns there are the results of "Improved benchmark" model (see Section 5.2 for details). "Distance" refers to average Euclidean distance, "Correct" refers to predictions with Chebyshev distance less 0.5468.

## 7.3 Location prediction analysis

### 7.3.1 Neural model

As is written in Table 7.1, our best location prediction model has average distance between correct and predicted location 0.87 on the development set and 0.71 on the test set. On the test set the predictions of this model has Chebyshev distance of less than 0.5468 in 67.9% cases, which means that with rounding the model predicts the correct location in this percentage of instances.

To better understand the behaviour of this model, we compute the average distance and percentage of the correctly predicted instances (less than 0.5468 Chebyshev distance) on the command categories described in Section 3.1. Note that the division of commands into categories is not perfect, as was already discussed. The numbers can be found in Table 7.3.1. There are also results of Improved benchmark for comparison in the table.

We can see that the quality of the predictions widely differs across the command categories. As for the simplest category "Source, reference, direction" our model succeeds in 91.0% cases and the average distance is 0.16. Out of five worst predictions, which in our analysis falls into this category, two of them contain wrong command (error of the annotator), two of them are in a wrong command category and only one is a real mistake of the network. This suggest that the performance of our best neural model on "Source, reference, direction" commands is even better than the numbers in Table 7.3.1.

On the other hand, for the most difficult command categories "More refer-

| Mistake type | Count | Description & Example |
|---|---|---|
| More references | 31 | Two or more references in the command<br>*Line up the Texaco cube with the Target cube but one row lower than the Starbuck's cube.* |
| Large distance from reference | 17 | Distance between reference and gold location is more than 1 block.<br>*McDonalds moves right until it is on top of Esso, then down to the bottom of the screen.* |
| Annotator error | 17 | The command does not describe the action correctly<br>*Move the Starbucks cube so that it's upper left corner touches the lower left corner of the right cube.* |
| World dependence | 13 | The world state is needed to interpret the sentence, our representation with weights is insufficient for these commands<br>*place block 1 right above all of the blocks for a full sequence* |
| Other | 22 | |

Table 7.4: Worst predictions analysis: Probable reasons behind bad predictions in 100 worst instances of test set.

ences" and "No reference" the model succeeds only in 26.4% and 20.3% cases respectively with average distance of 1.77 and 2.41. The model fails on the commands of these categories mainly because of two reasons:

- low amount of commands in these categories,
- more diverse sentence structure.

For example only 2% of the dataset (417 commands) falls into the "No reference" category compared to 47% in the "Source, reference, direction" category.

The analysis of 100 worst predictions (Table 7.3.1) shows similar results with 31 out of 100 worst predictions likely caused by multiple references in the command. The model is not able to understand such commands. We looked at the predictions of commands of this category and find out, that even if the prediction is very good the model uses only one reference. For example consider commands:

*In the space to the right of block 7 and below block 9, place block 10.*

*place box 10 so that box 6 is between it and box 4*

In both of them, the model uses as a single block as a reference - 9 in the first command, 6 in the second one. In other words, if you consider the representation described in 5.4, for the first sentence $w_9 = 0.88$ and all other weights are 0. For the second sentence $w_6 = 1.00$ and all other weights are also 0. Only in few cases the model sets uses weights of more blocks mentioned in the command. Even then one block is clearly preferred and its weight is much bigger than the other weights.

Besides the "More references" problem, other large source of errors are commands with distances such as:

*Take block 1 and place it so it is 6 spaces above and 1 to the left of block 8.*

To better understand this problem we try to change the numerical distance (i.e. word "*four*") in the sentence:

*place box 16 four places below box 15*

We find out, that the model understands distances up to approximately 4 blocks. For larger distances the predictions are not accurate. This is likely caused by the number of occurrences of the numerals in the training data. The token"*four*" has 99 occurrences, when "*five*" and "*six*" have 33 and 32 occurrences. For higher numerals the number of occurrences is even lower ("*ten*" has 6 occurrences) and lower numerals have more occurrences ("*two*" has 423 occurrences).

It should be noted, that not all these occurrences are describing distances, some of them are used to describe blocks. Therefore conclusion that the model cannot learn a word with 33 or fewer occurrences is not true. For example, the model understands word "*coke*" which appears in the training data only 16 times. Also it understands word "*nine*" (13 occurrences) when it is used to describe block, but "*seven* (11 occurrences) in similar context is not understood.

The third big source of errors in location prediction of our best neural model are annotator errors. Most often the badly annotated command describes action which is needed to get $i$-th world state of the sequence from $i+1$-th, but it should be the other way around. Some of the badly annotated commands also miss very important information such as command:

*Place the Shell box so there are two empty spaces between it and the SRI box*

does not specify the direction, or some are plain wrong. Obviously the model is not able to predict them. But this is not the only problem they are causing. Even if there were no other badly annotated instances then the 17 we found out, which is unlikely, it is 0.5% of the test dataset. If there is similar percentage of such commands in the training data, the training process is likely negatively influenced by them.

Last large source of location prediction errors are commands where the location depends on the world state. An example of such a command is in Table 7.3.1. These commands cannot be represented by our chosen representation, because they often refer to some geometric figures the blocks form or work with the world state in another way which do not fit the "block weights and direction" representation.

### 7.3.2 Rule-based model

For the rule-based model the differences between command categories are even bigger than for the neural model. The rule-based model is relatively successfull only on the simple "Source, reference, direction" commands, while showing bad performance on the other categories. This is not very surprising, because the model was designed mainly with the "Source, reference, direction" commands

in mind. On the "More references" category, the performance of the rule-based model is not very far behind the neural model. This shows, that for such commands, predicting the center of gravity of blocks mentioned in the command is a good heuristic.

# Conclusion

In this work we presented various neural network architectures for interpreting natural language commands in a simulated world. We also discussed rule-based models and compared them to the neural ones. We find out that the neural models with wisely created architecture work better, but the performance gap is not very wide.

We developed and compared various preprocessing strategies which improved our predictive performance. We tested two forms of data augmentation - generating new commands and switching blocks in the existing commands - but did not observed significant improvement of prediction accuracy with them.

Lastly we discussed the performance of our models, analyzed their behaviour and proposed another metric for measuring location prediction performance.

This work had three main goal:

1. To create models with better performance than the ones described by Bisk et al. [2016b].

2. To create rule-based model which uses language analysis and compare its performance to the neural models.

3. To create an interactive visualization of our problem.

The best models of Bisk et al. [2016b] have source accuracy 98% and average distance between the correct and the predicted location 0.98. Our best models have 98.8% accuracy and the average distance 0.71. Thus we created better models for both the source and the location prediction and thus we accomplished our first goal.

We created a rule-based model, but did not use a language parser in it. The reason why we did not use the language parser is that the performance of UDPipe parser (Straka et al. [2016]), which we tested, was in our opinion bad on our dataset and using it would not improve our rule-based benchmark. Since Straka et al. [2016] reports competitive parsing performance we did not tested other parsers. We compared our rule-based benchmark to neural models, thus the second goal was partially accomplished.

The visualization tool was created and is described in the Attachment.

## Future work

In our opinion the source prediction performance is satisfying and it is hard to improve. But for the location prediction we think there is a space for improvement, namely in the following areas:

1. Optimization - many approaches, such as embeddings, deeper networks and new command generation failed mainly because we were not able to find good enough network parameters. However we know that such parameters exists. This is the case, because we were able to optimize networks with one-hot encoding, which is a subset of random trainable embeddings with big enough dimension, but we could not optimize networks predicting location

with embeddings. We were also able to optimize shallower networks, which are a subsets of deeper networks, which we were unable to successfully optimize.

2. Data augmentation - we think that there are many other possibilities for data augmentation than the ones we tried. Because the dataset size is relatively small, some form of data augmentation is very likely to help.

3. Novel neural architectures

4. Model ensembles

Since there are many errors in the dataset, it might be more useful to improve the dataset itself or create a new and better dataset than trying to create better performing models for the current data. Trying to create such a system in the real world would be even more useful, because even if it does not work it will improve our understanding of the requirements for a module for understanding commands.

# Bibliography


Yoav Artzi and Luke Zettlemoyer. Weakly supervised learning of semantic parsers for mapping instructions to actions. *Transactions of the Association for Computational Linguistics*, 1:49–62, 2013.

Yonatan Bisk, Daniel Marcu, and William Wong. Towards a dataset for human computer communication via grounded language acquisition. In *Workshops at the Thirtieth AAAI Conference on Artificial Intelligence*, 2016a.

Yonatan Bisk, Deniz Yuret, and Daniel Marcu. Natural language communication with robots. In *Proceedings of NAACL-HLT*, pages 751–761, 2016b.

Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

Kais Dukes. Semantic annotation of robotic spatial commands. In *Language and Technology Conference (LTC)*, 2013.

Kais Dukes. Semeval-2014 task 6: Supervised semantic parsing of robotic spatial commands. In *Proceedings of the 8th International Workshop on Semantic Evaluation (SemEval 2014)*, pages 45–53, 2014.

Dan Flickinger. On building a more effcient grammar by exploiting types. *Natural Language Engineering*, 6(01):15–28, 2000.

Pavel Golik, Patrick Doetsch, and Hermann Ney. Cross-entropy vs. squared error training: a theoretical and experimental comparison. In *Interspeech*, pages 1756–1760, 2013.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

Ting Han and David Schlangen. Grounding language by continuous observation of instruction following. *EACL 2017*, page 491, 2017.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. An empirical exploration of recurrent network architectures. 2015.

Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Yann LeCun. The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*, 1998.

Wang Ling, Tiago Luís, Luís Marujo, Ramón Fernandez Astudillo, Silvio Amir, Chris Dyer, Alan W Black, and Isabel Trancoso. Finding function in form: Compositional character models for open vocabulary word representation. *arXiv preprint arXiv:1508.02096*, 2015.

Zachary C Lipton, John Berkowitz, and Charles Elkan. A critical review of recurrent neural networks for sequence learning. *arXiv preprint arXiv:1506.00019*, 2015.

Matt MacMahon, Brian Stankiewicz, and Benjamin Kuipers. Walk the talk: connecting language, knowledge, and action in route instructions, proceedings of the 21st national conference on artificial intelligence, 2006.

Larry Medsker and Lakhmi C Jain. *Recurrent neural networks: design and applications*. CRC press, 1999.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

Joakim Nivre, Marie-Catherine de Marneffe, Filip Ginter, Yoav Goldberg, Jan Hajic, Christopher D Manning Ryan McDonald, Slav Petrov, Sampo Pyysalo, Natalia Silveira Reut Tsarfaty, and Daniel Zeman. Universal dependencies v1: A multilingual treebank collection. 2016.

Peter Oram. Wordnet: An electronic lexical database. christiane fellbaum (ed.). cambridge, ma: Mit press, 1998. pp. 423. -. *Applied Psycholinguistics*, 22(1): 131–134, 2001.

Woodley Packard. Uw-mrs: leveraging a deep grammar for robotic spatial commands. *SemEval 2014*, page 812, 2014.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. Understanding the exploding gradient problem. *CoRR, abs/1211.5063*, 2012.

Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *EMNLP*, volume 14, pages 1532–1543, 2014.

Lutz Prechelt. Automatic early stopping using cross validation: quantifying the criteria. *Neural Networks*, 11(4):761–767, 1998.

Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151, 1999.

Frank Rosenblatt. *The perceptron, a perceiving and recognizing automaton Project Para*. Cornell Aeronautical Laboratory, 1957.

Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

Milan Straka, Jan Hajič, and Straková. UDPipe: trainable pipeline for processing CoNLL-U files performing tokenization, morphological analysis, pos tagging and parsing. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*, Paris, France, May 2016. European Language Resources Association (ELRA). ISBN 978-2-9517408-9-1.

Stefanie A Tellex, Thomas Fleming Kollar, Steven R Dickerson, Matthew R Walter, Ashis Banerjee, Seth Teller, and Nicholas Roy. Understanding natural language commands for robotic navigation and mobile manipulation. 2011.

Matthew R Walter, Matthew Antone, Ekapol Chuangsuwanich, Andrew Correa, Randall Davis, Luke Fletcher, Emilio Frazzoli, Yuli Friedman, James Glass, Jonathan P How, et al. A situationally aware voice-commandable robotic forklift working alongside people in unstructured outdoor environments. *Journal of Field Robotics*, 32(4):590–628, 2015.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

# List of Figures

# List of Tables

# List of Abbreviations

**CBOW** Continuous bag-of-words
**FFN** Feed-forward network
**GRU** Gated recurrent unit
**LSTM** Long short-term memory
**MSE** Mean squared error
**NLP** Natural language processing
**RNN** Recurrent neural network

# Attachments

The DVD attached to this thesis contains this document in file *thesis.pdf* and folder *blockworld_commands* with source code used for conducting experiments described here. Copy of the *blockworld_commands* folder can also be obtained from *https://github.com/bedapisl/blockworld_commands.git*.

The folder contains:

- A readme file *README.md* with installation instructions and description of the application.
- A folder *src* with the source code and the best models.
- A folder *data*, which contains, among other things, file *basic_database.db* with data preprocessed by our best preprocessing method.
- A file *requirements.txt* which lists needed Python libraries.