Learning to Align the Source Code to the Compiled Object Code

Dor Levy[1] Lior Wolf[1 2]

Abstract

We propose a new neural network architecture and use it for the task of statement-by-statement alignment of source code and its compiled object code. Our architecture learns the alignment between the two sequences – one being the translation of the other – by mapping each statement to a context-dependent representation vector and aligning such vectors using a grid of the two sequence domains. Our experiments include short C functions, both artificial and human-written, and show that our neural network architecture is able to predict the alignment with high accuracy, outperforming known baselines. We also demonstrate that our model is general and can learn to solve graph problems such as the Traveling Salesman Problem.

1. Introduction

The problem of aligning sequences is well studied in the literature across many domains, e.g., machine translation (Brown et al., 1993; Dyer et al., 2013; Bahdanau et al., 2014), speech recognition (Graves et al., 2006; 2013), handwriting recognition (Graves et al., 2009; Graves & Schmidhuber, 2009), alignment of books with movies made based on them (Zhu et al., 2015) and more. The alignment is often done sequentially, one step at a time. We propose a neural network architecture, capable of aligning two input sequences globally and at once.

We focus on the alignment of source code and its translation to the compiled object code. During compilation, source code typically written in a human-readable high level programming language, such as C, C++ and Java, is transformed by the compiler to object code. Every object code statement stems from a specific location in the source code, and, therefore, there is a statement-level alignment between source code and object code.

As far as we know, statement-by-statement alignment of source- and object-code is not treated in the literature. It is challenging, since the per-statement outcome of the compilation process also depends on other statements of the source code. In addition, this outcome is produced in increasing levels of sophistication that are determined by the compiler's optimization flags.

Our compound deep neural network combines one embedding and one RNN per input sequence, a CNN applied to a grid of sequence representation pairs and multiple softmax layers. Training is performed using both real-world and synthetic data that we created for this purpose. The real-world data consists of 53,000 short functions from 90 open-source projects of the GNU project. Three levels of compiler optimization are used and the ground truth alignment labels are extracted from the compiler's output.

Our experiments[1] show that the neural network presented is able to predict the alignment considerably more accurately than the literature baselines. Moreover, our method is general and transcends the problem of aligning sequences. We demonstrate it by using exactly the same architecture for learning the Traveling Salesman Problem.

1.1. Our Contributions

We propose a novel network architecture and challenge it with a difficult alignment problem, which has unique characteristics: the input sequences' representations are not per token, but per statement (a subsequence of tokens). The alignment is predicted by our architecture not sequentially (e.g., by employing attention), but by considering the entire grid of potential alignments at once. This is done using an architecture that combines a top-level CNN with LSTMs (Hochreiter & Schmidhuber, 1997).

While neural networks have been shown to be capable of aligning sequences in the domain of NLP, where a sentence in one natural (human) language is aligned with its translation (Bahdanau et al., 2014), the current domain has additional challenges. First, each source or object-code statement contains both an operation (a reserved C key-

[1]The School of Computer Science, Tel Aviv University [2]Facebook AI Research. Correspondence to: Dor Levy <dor.levy@cs.tau.ac.il>, Lior Wolf <wolf@cs.tau.ac.il>.

Proceedings of the 34th International Conference on Machine Learning, Sydney, Australia, PMLR 70, 2017. Copyright 2017 by the author(s).

[1]Our code and data are publicly available at: `https://github.com/DorLevyML/learn-align`

word or an opcode) and potentially multiple parameters, and are, therefore, typically more complex than natural language words. Second, highly optimized compilation means that the alignment is highly non-monotonous. Third, the alignment is very often partial, since not all source-code statements are aligned with the object-code statements. Finally, the meaning of each code statement is completely context dependent, since, for example, the variables are reused within multiple statements. In natural languages, the context helps to resolve ambiguities. However, a direct dictionary based alignment already provides a moderately accurate result. In the current domain, the alignment process has to depend entirely on the context.

1.2. Related Work

Extensive work was done on the problem of predicting the alignment and computing its probability given a pair consisting of a source sentence and a candidate target sentence (Brown et al., 1993; Dyer et al., 2013). The alignment probability is then used for re-ranking the translation candidates in the translation pipeline.

Bahdanau et al. (2014) propose an architecture for jointly aligning and translating between two languages. The encoder of the source language is based on a bidirectional RNN. During the decoding process, in which the new sentence in the target language is created, an RNN is used to predict one word at a time. This RNN pools as one of its inputs, a weighted combination of the representations of the various words in the source language. The weights of this combination are pseudo-probabilities that represent the similarity of the predicted word in the translated sentence to each of the words in the source sentence. In contrast to our work, the model described in (Bahdanau et al., 2014) *implicitly* aligns an input sequence to an output sequence, as part of the translation process, while our model *explicitly* aligns two input sequences. Note that most human languages are relatively similar and are constructed by similar rules. It is unlikely that the same translation architectures could successfully and accurately translate, for example, C code to object code.

Our work is close in concept to Pointer Networks (Vinyals et al., 2015), where the proposed architecture outputs discrete tokens corresponding to positions in the input sequence. The input sequence is first encoded by an LSTM to a representation sequence. A second LSTM, at each time step, then points to a location in the input sequence through an attention mechanism and given the previously pointed value of the input sequence. Similarly, our architecture also points to locations in an input sequence. However, in contrast to Vinyals et al. (2015), our architecture receives two input sequences and points to locations on a grid formed by the two.

The approach that is most closely related to ours is Match-LSTM (Wang & Jiang, 2015). This architecture is used to determine, given a premise sentence and a hypothesis sentence, whether the hypothesis can be inferred from the premise. The Match-LSTM is designed to do so by matching of the hypothesis and premise word-by-word. First, the two sentences are processed using two LSTM networks. A third LSTM then processes sequentially the hypothesis representation sequence and for every word in the hypothesis sentence produces a match score for all words in the premise representation sequence using an attention mechanism. Finally, after the third LSTM is done processing the hypothesis sentence, its last hidden state is used to produce a single prediction for the relation between the hypothesis and the premise. Although our work is aimed to align two sequences, our proposed architecture is far from Match-LSTM. While Match-LSTM matches sequentially every word in the hypothesis sentence to all words in the premise, our architecture represents all the statement pairs as a grid and aligns all of them globally and combined, using a CNN. Another difference is that the alignment produced by Match-LSTM is only implicit, since the goal of the architecture is to predict the relation between the two sequences. In our architecture, the alignment is explicit and fully supervised during training.

Another aspect in which our architecture differs from the ones proposed by Bahdanau et al. (2014); Vinyals et al. (2015); Wang & Jiang (2015), is that in order to align statements, it does not learn representations that correspond to tokens in the input sequences, but learns representations that correspond to segments in the input sequence – each segment being a mini-sequence of tokens that corresponds to one statement.

Sequence processing with CNNs The use of CNNs for sequence processing tasks has been expanding recently. Such tasks include sequence encoding (Zhang et al., 2015; Lee et al., 2016), sentiment prediction (Blunsom et al., 2014), document summarization (Denil et al., 2014) and translation (Gehring et al., 2017). One reason is the computational efficiency of CNNs compared to RNNs, which leads to faster computations both on GPU and CPU. Another reason is their ability to capture translation invariant features in text, as shown by, e.g., Allamanis et al. (2016), who use a convolutional attention mechanism in order to generate extreme summarization of source code functions.

Neural networks and source code tasks Neural networks have been shown to be useful in tasks involving source code. For example, In (Zaremba & Sutskever, 2014) a sequence-to-sequence LSTM learns to execute simple class of python programs only from seeing input-output pairs. In Allamanis et al. (2016), a model learns to generate meaningful summaries to short functions written in Java.

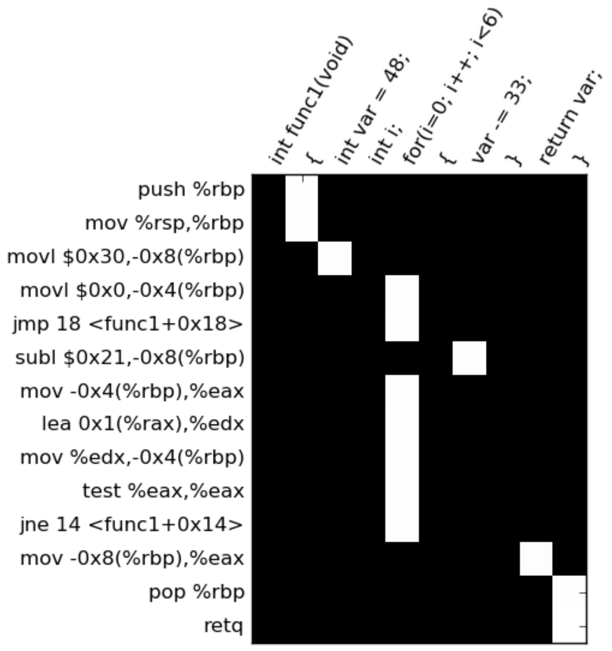

Figure 1. The alignment matrix, presenting statement-by-statement alignment between a sample C function (columns) and the object code that results from compiling the C code (rows). The white cells indicate correspondence.

2. The Code Alignment Problem

We consider source code written in the C programming language, in which statements are generally separated by a semicolon (;). The compiler translates the source code to object code. For example, the GCC compiler (Stallman et al., 2009) is used. We view the object code as assembly, where each statement contains an opcode and its operands. Since the source code is translated to object code during compilation, there is a well-defined alignment between them, which is known to the compiler. GCC outputs this information when it runs with a debug configuration.

2.1. Problem Formalization

In the GCC alignment output, the statement level alignment between source- and object-code is a many-to-one map from object code statements to source code statements: while every object-code statement is aligned to some source-code statement, not all source-code statements are covered. This is due to optimization performed by the compiler. Our definition of a statement is slightly modified, due to the convention used in the GCC alignment output. A C statement can be one of the following: (i) a simple statement in C containing one command ending with a semicolon; (ii) curly parentheses ({,}); (iii) the signature of a function; (iv) one of `if(EXP1)`, `for(EXP1;EXP2;EXP3)`, or `while(EXP1)`, including the corresponding expressions; (v) `else` or `do`. Note that the following code

```
1  do
2  {
3      a += 4;
4  }
5  while(i < 500);
```

contains five statements (as numbered) since the `do`, the {, the }, and the `while` are all separate statements.

The object code statements follow the conventional definition, as shown, for example, in assembly code listings. Each statement contains a single opcode such as `mov`, `jne`, or `pop`, and its operands.

An example is shown in Fig. 1, which depicts both the source code of a single C language function, which contains $M = 10$ statements, and the compiled object code of this function, which contains $N = 14$ statements. The alignment between the two is shown graphically by using a matrix of size $N \times M$. Each column (row) of this matrix corresponds to one source (object) code statement. The matrix (i, j) element encodes the probability of aligning object-code statement $i \in 1, \ldots, N$ with the source-code statement $j \in 1, \ldots, M$. Since the matrix is the ground truth label, all probabilities are either 0 (black) or 1 (white). In other words, each row i is a one-hot vector showing the alignment of one object-code statement i.

As can be seen in the figure, the first opcode `push` corresponds to the function's statement {, that opens the function's block. As expected, there are also many opcodes that implement the `for` statement, which comprises comparing, incrementing and jumping.

The matrix representation is the target value of the neural alignment network. The network will output the rows of the alignment matrix as vectors of pseudo probabilities. We can view the resulting prediction matrix as a soft-alignment. In order to obtain hard alignments, we simply take the index of the maximal element in each row.

Another dimension in which we challenge our alignment network, is compilation optimization, which drastically changes the object code based on the level of optimization used (see Fig. 1 of supplementary material). This optimization makes the object code more efficient and can render it shorter (more common) or longer than the code without optimization, see supplementary material.

3. The Neural Alignment Network

Each statement is treated as a sequence of tokens, where the last token of each such sequence is always the end-of-statement (EOS) token. A function is given by concatenating all such sequences to one sequence.

We employ a compound deep neural network for predicting the alignment, as explained in Sec. 3.2. It consists of four parts: the first part is used for representing each source code statement j as a vector v_j. The second part does the same for the object code, resulting in a representation vector u_i. The third part processes using a convolutional neural network pairs of vector representations, one of each type, as a multi-channel grid, and produces an alignment score $s(i,j)$. It is not a probability. However, the higher the alignment value, the more likely the two statements are to be aligned. The alignment scores are fed to the top-most part of the network, which computes the pseudo probabilities p_{ij} of aligning object code statement i to the source code statement j. Specifically, the fourth part considers for an object-code statement i, all source-code statements $j = 1, 2, \dots, M$ the alignment score, and employs the softmax function: $p_{ij} = \frac{exp(s(u_i,v_j))}{\sum_{k=1}^{M} exp(s(u_i,v_k))}$.

3.1. Encoding the Input Statements

Our model incorporates two LSTM networks to encode the sequences, one for each sequence domain: source code and object code. Therefore, we first embed each token in the input sequences in a high-dimensional space. We use different embeddings for source code and for object code, since each is composed of a different vocabulary. The vocabularies are hybrid, in the sense that they consist of both words and characters.

The source code vocabulary is a hybrid of characters and the C language reserved words. A C reserved word is embedded to a single vector, while variable names, arguments and numeric values are decomposed to character by character sequences. The vocabulary contains the C language reserved words as atomic units, EOS, and the following single character elements: (i) alphanumeric characters including all letters and digits; (ii) the operators `+`, `-`, `/`, `*`, `&`, `|`, `^`, $\sim$, `?`; and (iii) the following punctuation marks: `(`, `)`, `[`, `]`, {, }, <, >, =, `!`, `,`, `'`, `"`, `;`, `#`, \. Let $\varepsilon(\alpha)$ denote the embedding of a C token α to a vector. Then the C code string `if (a5<42)`, for example, is decomposed to the following sequence: $\varepsilon(if)$, $\varepsilon(()$, $\varepsilon(a)$, $\varepsilon(5)$, $\varepsilon(<)$, $\varepsilon(4)$, $\varepsilon(2)$, $\varepsilon())$, $\varepsilon(EOS)$.

Similarly, the object code vocabulary is also a hybrid, and contains opcodes, registers and characters of numeric values and is based on the assembly representation of the object code. The opcode of each statement is one out of dozens of possible values. The operands are either registers or numeric values. The vocabulary also includes the punctuation marks of the assembly language and, therefore, contains the following types of elements: (i) the various opcodes; (ii) the various registers; (iii) hexadecimal digits; (iv) the symbols `(,),x,-,:`; and (v) EOS, which ends every statement. Let $\varepsilon'(\beta)$ denote the embedding of an object code token β to a vector. Then the following assembly string `mov %eax,-0x8(%rbp)`, for example, is decomposed to the following sequence: ε'(`mov`), ε'(`%eax`), ε'(`-`), ε'(`0`), ε'(`x`), ε'(`8`), ε'(`(`), ε'(`%rbp`), ε'(`)`), ε'(`EOS`).

3.2. Neural Network Architecture

The network architecture is depicted in Fig. 2(a). The input sequences introduce many complex and long-range dependencies. Therefore, the network employs two LSTM encoders: one for creating a representation of the source code statements and one is used for representing the object code statements. In all of our experiments, the LSTMs have one layer and 128 cells.

Recall that each statement in the input sequences is a sequence of tokens. However, for alignment, only a single vector representation is required per statement. In order to obtain a single vector representation per statement, we sample the representation sequences output by the encoders only at time steps corresponding to EOS's. It should be noted that information from other statements is not lost, since each RNN activation is affected by other activations in the sequence. Moreover, since EOS is ubiquitous, its representation must be based on its context. Otherwise, it is meaningless. During training, the network learns to create meaningful representations at the location of the EOS inputs.

The result of the LSTM encoders are M representation vectors output by the source-code encoding LSTM, denoted by $\{v_j\}_{j\in(1,\dots,M)}$, and N representation vectors output by the object-code encoding LSTM, denoted by $\{u_i\}_{i\in(1,\dots,N)}$. The statement representation vectors are then assembled in an $N\times M$ grid, such that the (i,j) element is $[u_i; v_j]$, where ; denotes vector concatenation. Since each encoder LSTM has 128 cells, the vector $[u_i; v_j]$ has 256 channels.

In order to transform the statement representation pairs to alignment scores, we employ a decoding Convolutional Neural Network (CNN) over the 256-channel grid. The decoding CNN has five convolutional layers, each with 32 5×5 filters followed by ReLU non-linearities, except for the last layer which consists of one 5×5 filter and no non-linearities. The CNN output is, therefore, a single channel $N \times M$ grid, $s(i,j)$, representing the alignment score of object code statement i and source code statement j.

In the many-to-one alignment problem, the network's output for each row should contain pseudo probabilities.

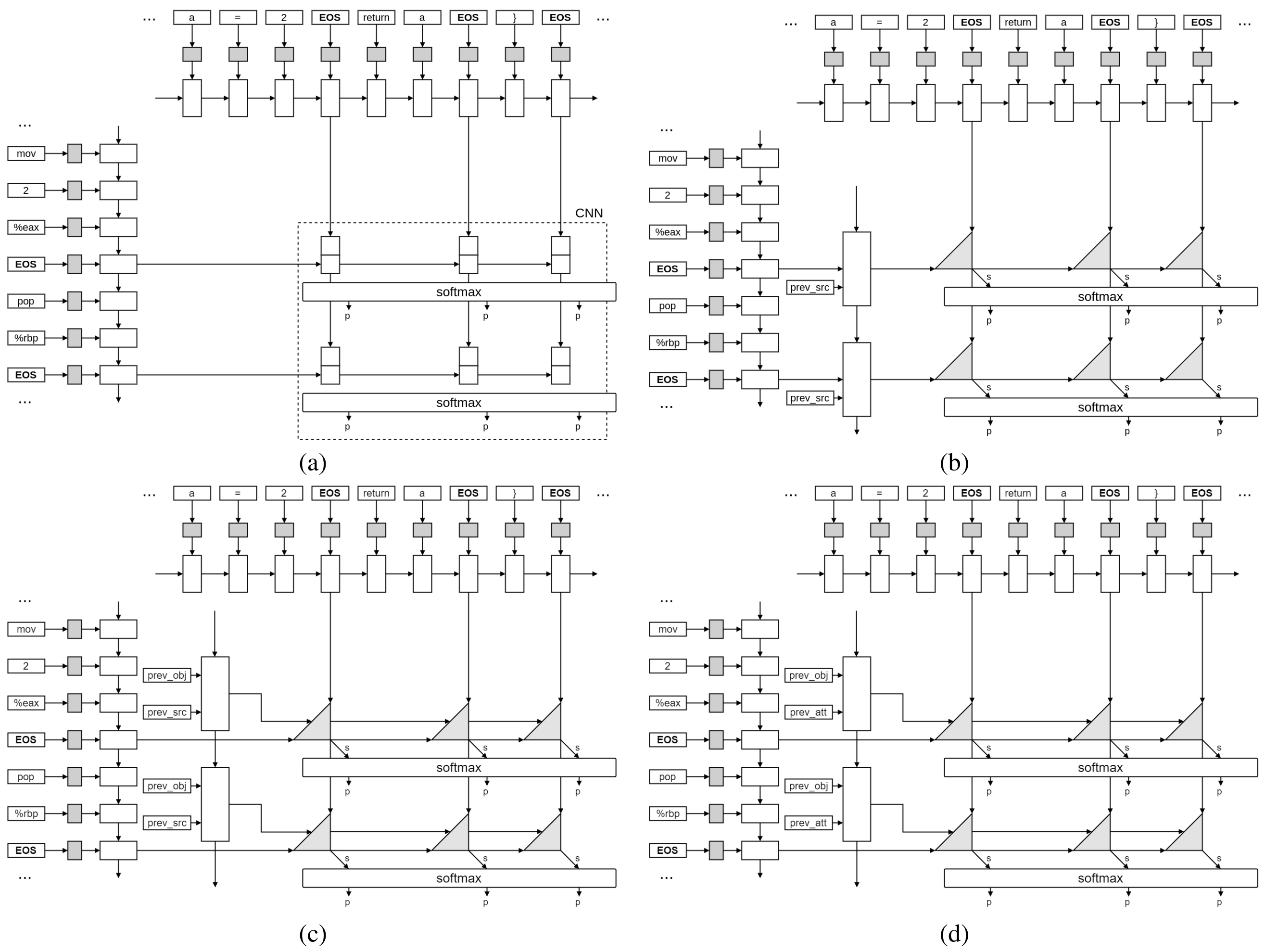

Figure 2. Various alignment networks, showing three source code statements and two object code statements. The sequences' tokens are first embedded (gray rectangles). The embedded sequences are then encoded by LSTMs (elongated white rectangles). The statement representations are fed to a decoder (different in every figure) and then the alignment scores (s) output by the decoder are fed into one softmax layer per each object code statement (rounded rectangles), which generates pseudo probabilities (p). (a) Our proposed Grid Decoder, in which the grid of encoded statements is processed by a CNN. (b),(c) The Ptr1 and Ptr2 baselines, respectively, in which a Ptr-Net decoder (Vinyals et al., 2015) processes sequentially the previously pointed source code statement (”prev src”) and either the current (Ptr1) or previous (Ptr2, ”prev obj”) object code statement. (d) The Match-LSTM baseline (Wang & Jiang, 2015), in which an LSTM decoder processes sequentially the current object code statement and the current attention-weighted sum of source code statements. The attention model receives the LSTM output of the previous time step. The Match-LSTM is similar to Ptr2, except that instead of the pointed source code statement, it receives the attention-weighted sum of source code statements (”prev att”).

Therefore, we add a softmax layer on top of the list of alignment scores computed for each object-code statement i: $s(u_i, v_1)$, $s(u_i, v_2)$, $\ldots$, $s(u_i, v_M)$, i.e., there are N softmax layers, each converting M alignment scores to a vector of probabilities $\{p_{ij}\}_{j\in(1,\ldots,M)}$ for each row $i \in (1,\ldots,N)$.

During training, the Negative Log Likelihood (NLL) loss is used. Let $\mathcal{A}$ be the set of N object-code to source-code alignments (i, j). The training loss for a single training sample is given by $\frac{1}{N}\sum_{(i,j)\in\mathcal{A}} -log(p_{ij})$, i.e., the loss is the mean of NLL values of all N rows.

3.3. Local Grid Decoder

For comparison, we also consider a model that performs decoding directly over the statements grid. In this model, the decoder consists only of a single layer network s attached to each one of the NM pairs of object code and source code statement representations (u_i and v_j). The same network weights are shared between all NM pairs and are trained jointly. This network is given by:

$$s(u_i, v_j) = v^T tanh(W_o u_i + W_s v_j)$$

where v, W_o and W_s are the network's weights. We consider another, simpler version of the Local Grid Decoder, where $s(u_i, v_j) = u_i^T \cdot v_j$, i.e., an inner product operation is employed, instead of the single layer network.

4. Literature Baseline Methods

In this section, we describe the baseline methods that we compare to our architecture. Our architecture and all baselines use LSTM encoders to encode the input sequences, and softmax layers on top of the decoder output in order to produce an alignment probability, as explained in Sec. 3.2. The architectures differ only in the decoder part that produces alignment scores, i.e., in the model $s(i, j)$. A profound difference between our architecture and the baselines, is that while our architecture predicts the alignment over the whole statements grid at once, the baselines predict the alignment sequentially.

4.1. Pointer Network

This baseline adapts the Pointer Network (Ptr-Net) architecture proposed by (Vinyals et al., 2015) in two ways. Ptr-Net is designed to solve the task of producing a sequence of pointers to an input sequence. The Ptr-Net architecture employs an encoder LSTM to represent the input sequence as a sequence of hidden states e_j. A second decoder LSTM then produces hidden states that are used to point to locations in the input sequence via an attention mechanism. Denote the hidden states of the decoder as d_i. The attention mechanism is then given by:

$$u_j^i = v^T tanh(W_1 e_j + W_2 d_i) \quad j \in (1, \ldots, n)$$
$$p_i = softmax(u^i)$$

where n is the input sequence length and p_i is the soft prediction at time step i of the decoder LSTM. The input to the decoder LSTM at time step i is $\arg\max_j(u_j^{i-1})$, i.e., the input token "pointed" by the attention mechanism at the previous step. Thus, the output of the decoder LSTM can be considered as a sequence of pointers to locations at the input sequence.

Since in the alignment problem we need to align each object code statement to one of the source code statements, we adapt Ptr-Net to produce "pointers" to the source code statements sequence for every object code statement. The adaptation is not trivial: our problem presents two input sequences, while Ptr-Net is originally designed to handle one. We create two such adaptations, Ptr1 and Ptr2, which are depicted in Fig. 2(b) and (c), respectively.

In Ptr1, we employ a Ptr-Net decoder at each time step i over the sequence of object code statement representations u_i. The decoder is an LSTM network, whose hidden state h_i is fed to an attention model employed over the whole sequence of source code statement representations v_j: $s(i, j) = v^T tanh(W_s v_j + W_h h_i)$.

The outputs $s(i, j)$ of the attention model are used as the alignment scores that will be fed later to the softmax layers. The Ptr-Net decoder receives at each time step i, the source code statement representation that the attention model "pointed" to at the previous step $i-1$, i.e., $v_{p_{i-1}}$ where $p_i = \arg\max_j(s(i, j))$.

Finally, in order to condition the output of the pointer decoder at the current object code statement representation u_i, the input of the pointer decoder LSTM is the concatenation of u_i and $v_{p_{i-1}}$:

$$h_i = LSTM([u_i; v_{p_{i-1}}], h_{i-1}, c_i, c_{i-1}),$$

where c_i is the contents of the LSTM memory cells at time step i. At the first time step $i = 1$, the value of v_{p_0} is the all-0 vector, and h_0 is initialized with the last hidden state of the source-code statements encoding LSTM.

It should be noted, that at each step, the Ptr-Net decoder sees the current object code statement and the previous "pointed" source code statement. It means that the LSTM sees the source code statement that is aligned to the previous object code statement. A wiser adaptation would present the Ptr-Net decoder LSTM with the explicit alignment decision, i.e., the previous "pointed" source code statement and the previous object code statement, such that the input is a pair of two statements that were predicted to align. Thus, in the second adaptation of Ptr-Net to our problem, which we call Ptr2, the input to the Ptr-Net decoder LSTM is the concatenation of u_{i-1} and $v_{p_{i-1}}$:

$$h_i = LSTM([u_{i-1}; v_{p_{i-1}}], h_{i-1}, c_i, c_{i-1}).$$

The current object code statement representation u_i is then fed directly to the attention model, in addition to the Ptr-Net decoder output and the source code statement representation: $s(i, j) = v^T tanh(W_o u_i + W_s v_j + W_h h_i)$.

4.2. Match-LSTM

This baseline uses the matching scores of the Match-LSTM architecture (Wang & Jiang, 2015). The architecture receives as inputs two sentences, a premise and a hypothesis. First, the two sentences are processed using two LSTM networks, to produce the hidden representation sequences v_j and u_i for the premise and hypothesis, respectively. Next, attention a_i vectors are computed over the premise representation sequence as follows: $a_i = \sum_{k=1}^{M} \alpha_{ij} v_j$, where

Table 1. Mean±SD for the two code alignment datasets.

DATASET	#FUNCTIONS	#STATEMENTS PER FUNCTION	#TOKENS PER STMT
SYNTHETIC C	150,000	21.8±8.0	8.2±9.0
SYNTH. OBJ.	150,000	17.1±8.3	3.6±2.0
GNU C	53,118	10.5±6.7	16.8±20.7
GNU OBJ.	53,118	21.2±18.1	1.2±1.1

α_{ij} are the attention weights and are given by

$$\alpha_{ij} = \frac{exp(s(u_i, v_j))}{\sum_{k=1}^{M} exp(s(u_i, v_k))}$$
$$s(i, j) = v^T tanh(W_o u_i + W_s v_j + W_h h_{i-1}),$$

where h_i is the hidden state of the third LSTM that processes the hypothesis representation sequence together with the attention vector computed over the whole premise sequence: $h_i = LSTM([u_i; a_i], h_{i-1}, c_i, c_{i-1})$.

For further details about the Match-LSTM architecture, see (Wang & Jiang, 2015). In order to adapt Match-LSTM to our problem, we simply substitute the premise (hypothesis) representation sequence with the source (object) code statements representation sequence, and use the matching scores $s(i, j)$ as the alignment scores.

5. Evaluation

Data collection We employ both synthetic C functions generated randomly and human-written C functions from real-world projects. In order to generate random C functions, we used pyfuzz, an open-source random program generator for python (Myint, 2013), and modified it so it will output short functions written in C rather than python. For the real-world human-written data set, we used over 53,000 short functions from 90 open-source projects, that are part of the GNU project and are written in C. Among them are `grep`, `nano`, etc. Before compilation, we ran only the preprocessor of GCC, in order to clean the sources of non-code text, such as comments, macros, `#ifdef` commands and more.

In order to compile the source code with optimizations, we use the GCC compiler (Stallman et al., 2009) with the optimization levels `-O1`, `-O2` or `-O3`. Each level turns on additional optimization flags. Each of the datasets of generated and human-written C functions has three parts, each compiled using one of the three mentioned optimization levels. After compilation of the human-written projects, some functions contained object code from other, inline functions. These functions were excluded from the dataset in order to introduce the network with pure translation pairs, i.e., source code and object code that has originated entirely from it. In addition, we tell GCC to output debugging information that includes the statement-level alignment between each C function and the object code compiled from it. Therefore, each sample in the resulting dataset consists of source code, object code compiled at some optimization level and the statement-by-statement alignment between them. Tab. 1 reports the statistics of the code alignment datasets.

Table 2. Alignment accuracy results for synthetic code.

METHOD	-O1	-O2	-O3	ALL
PTR1	99.27%	98.37%	98.49%	98.70%
PTR2	99.48%	98.71%	98.76%	98.98%
MATCH-LSTM	99.21%	97.98%	98.25%	98.46%
INNPROD GRID	99.42%	98.71%	98.81%	98.97%
LOCAL GRID	99.47%	98.75%	98.83%	99.02%
CONV. GRID	99.62%	98.77%	98.86%	99.08%

Table 3. Alignment accuracy results for GNU code.

METHOD	-O1	-O2	-O3	ALL
PTR1	86.90%	83.45%	83.77%	84.91%
PTR2	86.21%	85.48%	86.35%	85.95%
MATCH-LSTM	87.02%	84.03%	84.69%	85.36%
INNPROD GRID	88.34%	88.90%	90.90%	89.09%
LOCAL GRID	88.73%	88.09%	89.70%	88.64%
CONV. GRID	91.19%	90.10%	91.54%	90.86%

Training procedure For each data set, we train one network for all optimization levels. The length of all functions has been limited to 450 tokens. The training set of synthetic functions contains 120,000 samples. The validation and the test sets contain 15,000 samples each. The training, validation and test sets of human-written functions contain 42,391, 5,474 and 5,253 samples, respectively. During training, we use batches of 32 samples each.

The weights of the LSTM and attention networks are initialized uniformly in $[-1.0, 1.0]$. The CNN filter weights are initialized using truncated normal distribution with a standard deviation of 0.1. The biases of the LSTM and CNN networks are initialized to 0.0, except for the biases of the LSTM forget gates, which are initialized to 1.0 in order to encourage memorization at the beginning of training (Józefowicz et al., 2015). The Adam learning rate scheme (Kingma & Ba, 2015) is used, with a learning rate of 0.001, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 1e - 08$.

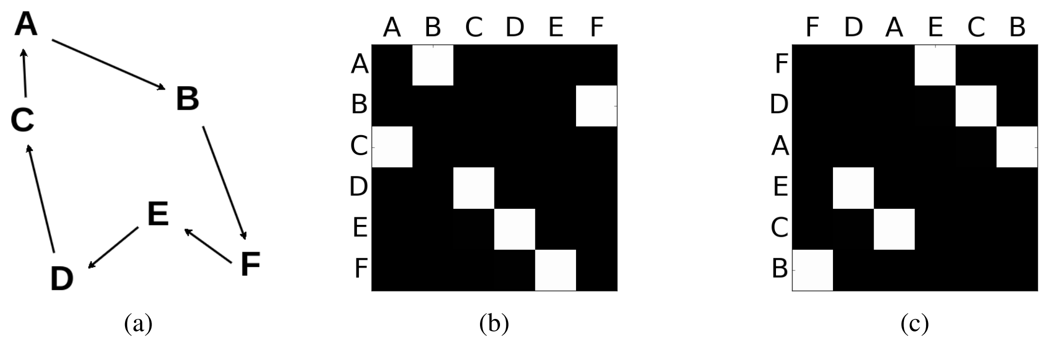

Figure 3. A sample TSP route (a), and its connectivity matrix, before (b) and after (c) permuting the node IDs.

Table 4. Average length of predicted TSP route.

n	OPT.	A1	A2	A3	PTR-NET	GRID D.
5	2.12	2.18	2.12	2.12	2.12	2.12
10	2.87	3.07	2.87	2.87	2.88	2.88

5.1. Alignment Results

Our proposed network and the baseline methods are trained and evaluated over the datasets of synthetic and human-written code. Tab. 2 and Tab. 3 present the resulting accuracy, which is computed per object-code statement as follows. First, the network predicts pseudo-probabilities of aligning source code statements to each object code statement. Second, in order to obtain hard alignments, we take the index of the maximal element in each row of the predicted soft alignment matrix. Third, for every object code statement, we count a true alignment only if the aligned source code statement is the ground truth alignment. The accuracy is reported separately for the three optimization levels and for all of them combined. As can be seen in Tab. 2, all models excel over synthetic code, reaching almost perfect alignment accuracy with a slight advantage to our Convolutional and Local Grid Decoders. Tab. 3 shows that the GNU code is more challenging to all methods. Our proposed Grid Decoder models outperform all baseline methods, and the Convolutional Grid Decoder is superior by a substantial margin over the local and inner product alternatives. The Ptr1, Ptr2 and Match-LSTM baselines reach about the same performance. It is an expected result, since these models are very similar: they all employ a decoding LSTM and an attention mechanism, with only small differences in performing the sequential processing of the encoded representation sequences.

5.2. Traveling Salesman Problem (TSP)

We perform an additional experiment based on the TSP benchmark presented in (Vinyals et al., 2015) in order to directly compare with the Pointer Network architecture (Ptr-Net), where it was already tested. The input of the TSP problem is a randomly ordered sequence of 2D points. The output is a sequence of all the points reordered, such that the route length (sum of distance between adjacent points) is minimal. For our method, we consider the connectivity matrix of the cycle graph in lieu of the alignment matrix. As reported in (Vinyals et al., 2015), overfitting was observed here. Therefore, we performed the following data augmentation process. For each sample in the training set, the IDs of the 2D points are permuted randomly and independently of the other samples. It is equivalent to randomly shuffling the order of the points in the sample sequence. The IDs in the label are then permuted accordingly, to represent the same target route. During training, the process was repeated at the beginning of every epoch, and independently of past epochs. Fig. 5 depicts an example route and its connectivity matrix before and after permutation of the node IDs. The results are presented in Tab. 4, along with the optimal and approximated results (see (Vinyals et al., 2015) for further details). As can be seen, our method is comparable to the original Ptr-Net model for both $n = 5$ and $n = 10$.

6. Summary

We present a neural network architecture for aligning two sequences. We challenge our network with aligning source code to its compiled object code, sequences that in some aspects are more complex than human language sentences. Our experiments demonstrate that the proposed architecture is successful in predicting the alignment. On this task, the network outperforms multiple literature baselines such as Pointer Networks and Match-LSTM, suggesting that a global, CNN-based approach to alignment is better than the sequential, RNN-based approach.

Our model can be used for alignment of any two sequences with a many-to-one map between them, and extended to other graph problems, as demonstrated for TSP.

Acknowledgments

The authors would like to thank Ofir Press, Dotan Kaufman and Shimi Salant for useful advice and insightful discussions. This work was supported by a 2016 ICRC grant.

References

Allamanis, M., Peng, H., and Sutton, C. A convolutional attention network for extreme summarization of source code. *arXiv preprint arXiv:1602.03001*, 2016.

Bahdanau, D., Cho, K., and Bengio, Y. Neural machine translation by jointly learning to align and translate. *arXiv preprint*, 1409.0473, 2014.

Blunsom, P., Grefenstette, E., and Kalchbrenner, N. A convolutional neural network for modelling sentences. In *Proc. of the 52nd Annual Meeting of the Association for Computational Linguistics*, 2014.

Brown, P. F., Della-Pietra, S. A., Della-Pietra, V. J., and Mercer, R. L. The mathematics of statistical machine translation. *Computational Linguistics*, 19(2):263–313, 1993.

Denil, M., Demiraj, A., Kalchbrenner, N., Blunsom, P., and de Freitas, N. Modelling, visualising and summarising documents with a single convolutional neural network. *arXiv preprint arXiv:1406.3830*, 2014.

Dyer, C., Chahuneau, V., and Smith, N. A. A simple, fast, and effective reparameterization of IBM model 2. In *HLT-NAACL*, pp. 644–648. The Association for Computational Linguistics, 2013.

Gehring, J., Auli, M., Grangier, D., Yarats, D., and Dauphin, Y. N. Convolutional sequence to sequence learning. *arXiv preprint arXiv:1705.03122*, 2017.

Graves, A. and Schmidhuber, J. Offline handwriting recognition with multidimensional recurrent neural networks. In *NIPS*, pp. 545–552, 2009.

Graves, A., Fernández, S., Gomez, F., and Schmidhuber, J. Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In *ICML*, 2006.

Graves, A., Liwicki, M., Fernndez, S., Bertolami, R., Bunke, H., and Schmidhuber, J. A novel connectionist system for unconstrained handwriting recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(5):855–868, May 2009.

Graves, A., r. Mohamed, A., and Hinton, G. Speech recognition with deep recurrent neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 6645–6649, May 2013.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, 1997.

Józefowicz, R., Zaremba, W., and Sutskever, I. An empirical exploration of recurrent network architectures. In *ICML*, 2015.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. In *ICLR*, 2015.

Lee, J., Cho, K., and Hofmann, T. Fully character-level neural machine translation without explicit segmentation. *arXiv preprint arXiv:1610.03017*, 2016.

Myint, S. Pyfuzz: Random program generator for python. `https://github.com/myint/pyfuzz`, 2013.

Stallman, R. M. et al. *Using The GNU Compiler Collection: A GNU Manual For GCC Version 4.3.3*. CreateSpace, Paramount, CA, 2009.

Vinyals, O., Fortunato, M., and Jaitly, N. Pointer networks. In *NIPS*, pp. 2692–2700, 2015.

Wang, S. and Jiang, J. Learning natural language inference with LSTM. *arXiv preprint arXiv:1512.08849*, 2015.

Zaremba, W. and Sutskever, I. Learning to execute. *arXiv preprint arXiv:1410.4615*, 2014.

Zhang, X., Zhao, J., and LeCun, Y. Character-level convolutional networks for text classification. In *NIPS*, pp. 649–657, 2015.

Zhu, Y., Kiros, R., Zemel, R., Salakhutdinov, R., Urtasun, R., Torralba, A., and Fidler, S. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *ICCV*, 2015.