Deep Almond: A Deep Learning-based Virtual Assistant

[Language-to-code synthesis of Trigger-Action programs using Seq2Seq Neural Networks]

Giovanni Campagna Rakesh Ramesh
Computer Science Department
Stanford University
Stanford, CA 94305
{gcampagn, rakeshr1}@stanford.edu

Abstract

Virtual assistants are the cutting edge of end user interaction, thanks to endless set of capabilities across multiple services. The natural language techniques thus need to be evolved to match the level of power and sophistication that users expect from virtual assistants. In this report we investigate an existing deep learning model for semantic parsing, and we apply it to the problem of converting natural language to trigger-action programs for the Almond virtual assistant. We implement a one layer seq2seq model with attention layer, and experiment with grammar constraints and different RNN cells. We take advantage of its existing dataset and we experiment with different ways to extend the training set. Our parser shows mixed results on the different Almond test sets, performing better than the state of the art on synthetic benchmarks by about 10% but poorer on realistic user data by about 15%. Furthermore, our parser is shown to be extensible to generalization, as well as or better than the current system employed by Almond.

1 Introduction

Today, we can ask virtual assistants like Amazon Alexa, Apple's Siri, Google Now to perform simple tasks like, "What's the weather", "Remind me to take pills in the morning", etc. in natural language. The next evolution of natural language interaction with virtual assistants is in the form of task automation such as "turn on the air conditioner whenever the temperature rises above 30 degrees Celsius", or "if there is motion on the security camera after 10pm, call Bob".

Almond [1] is an open, crowdsourced and programmable virtual assistant that was built as part of the Open Mobile Platform project [2] at Stanford. Central to Almond is *Thingpedia*, which is an open repository of different services, including Internet of Things (IoT) devices, open Web APIs and Social networks along their natural language interfaces. Thingpedia, which is an encyclopedia for the IoT, contains information about each *device* along with a set of *functions* that correspond to each device API. Each Thingpedia entry for a function also contains a natural language annotation that captures how humans refer and interact with the device. Through the efforts of crowdsourcing[1], Thingpedia contains a set of 50 devices and 187 functions. The 50 devices span a variety of domains from media (news papers, web comics), social networks (twitter, facebook), home automation (light bulb, thermostat), communication(email, calendar), etc.

Built on top of Thingpedia is the execution system, called *Thingsystem*, that takes user programs in the form of Trigger-Action programs (also known as If-This-Then-That programs) and maps them to the low-level device implementation in the repository. We express the intermediate Trigger-Action programs in a high-level domain specific language called *ThingTalk*. ThingTalk can connect devices

together by specifying the compositional logic while abstracting the device implementation and the communication. The following is an illustration of a ThingTalk program that posts Instagram pictures with hashtags containing "cat" as pictures on Facebook:

@**instagram**.new_picture(*picture_url*, *caption*, *hashtags*),
    Contains(*hashtags*, "cat")
⇒ @**facebook**.post_picture(*text*, *url*),
    text = *caption*, *url* = *picture_url*

Thus, any natural language command can thus be simply expressed as a program of ThingTalk, in the trigger-query-action world, by identifying the correct device, correct functions and the corresponding parameter values. There are 2 types of commands that are supported in Almond: *Primtive* and *Compound*. Any *Primitive* command is a simple invocations of an *action*, *query* or standing queries of triggers (*Monitors*) while *Compound* commands are constructed by composing two or more primitive operations. Fig.1 illustrates the scope of commands that are supported by Almond.

Class	Type	Examples
primitive	action	send email to bob
	query	get my latest email
	monitor	notify me when I receive an email
	filtered monitor/query	notify me when I receive an email if the subject contains deadline
compound	trigger+query	every day at 6am get latest weather
	trigger+action	every day at 6am send email to bob
	query+action	get latest weather and send it via email to bob
	trigger+query +action	every day at 6am get latest weather and send it via email to bob

Figure 1: Categories of commands accepted by Almond

The key step in the architecture of Almond is the parsing of the natural language command and synthesizing the correct ThingTalk program. We use logical forms as intermediate representation of a ThingTalk program because it captures the semantics while maintaining the structure of the program. An example parse of a natural language command is shown below:

play "presidential debate" from youtube on my tv
    ⇓
rule **tt:youtube.search_video** *query* is "presidential debate" **tt:tv.play_url** *video_url* is *video_url*
    ⇓
@**youtube**.search_video(*query*, _, _, _, _, _, _, _, *video_url*),
    query = "presidential debate"
⇒ @**tv**.play_url(*video_url*),
    video_url = *video_url*

Currently[1], Almond is built on top of the semantic parsing framework, SEMPRE [3], by specifying grammar rules to generate candidate programs and select the best program using a log-linear model with hand tuned paraphrasing and program synthesis features. We bootstrapped the parser by collecting natural language paraphrases for different ThingTalk programs using the generated canonical description and supplying them to Amazon Mechanical Turk, similar to the approach presented in Wang, Berant et al. [4]. Our parser achieves an accuracy of 51% on the collected dataset and a top-3 accuracy of 61% (right program is among the top 3 candidates) which are well-below the 90% threshold required for an usable virtual assistant. Furthermore, when we add more devices to our library and more training data, we found that the accuracy substantially decreases, which suggests that we have reached the limit of the classical model.

Hence, the goal of this project is to investigate the state of the art deep learning approaches to improve the accuracy of Almond. In this report, we detail the performance of a simple sequence-to-sequence (seq2seq) neural network model with an attention layer, as detailed by Dong et.al. [] (henceforth referred to as Lang2Logic), on the Almond dataset. We evaluate our deep learning system on 3 high-level goals: Accuracy, Coverage, Extensibility. We compare our system to the baseline semantic parsing system and discuss the future directions of the Almond parsing system.

2 Related Work

2.1 Trigger-action programming

The first notable attempt to build a trigger-action programming system is CAMP [5]. They include a limited "natural language" system to describe the rules based on small sentence pieces that are glued together in a visual way like fridge magnets.

The state of the art for trigger-action programming is the If This Then That (IFTTT) [6] website. In this system, the users can use a graphical interface to select two functions for trigger and action, and connect the parameters. The expressive power of IFTTT is more limited than ThingTalk, as it lacks filters, compound types and queries. On the other hand, IFTTT has been shown to be effective in user testing [7], even when the expressive power is extended to include filters.

2.2 Semantic parsing

The body of previous work in semantic parsing is abundant, in domains such as question answering [8, 3, 4, 9], trigger action programming [10, 11, 12] and instructions to robotic agents [13, 14]. The techniques divide in 3 main group: structured classification, loose classification and machine translation.

The first technique uses a grammar of the input sentence, which can be specified manually [13, 15] or learned [16]. This technique has very high precision, at the expense of requiring a lot of knowledge of the input sentences. It also suffers from low recall, as sentences that do not follow the grammar are not parsed successfully.

The second technique is loose classification, which treats the input sentence as a bag of linguistic features to drive a logical form generator. This generator can build the program compositionally bottom-up, like in our cases and in the other works based on SEMPRE [8, 3, 4], or predict the logical form top-down, like in KRISP and the works based on it [17, 14, 10]. This technique has high recall and does not require large datasets, but it does so at the expense of precision, for the top-down predictors, or linguistic generalization, for the bottom-up generators. Finally, semantic parsing as machine translation [18] uses a sequence model of the input sentence, and predicts either the sequence of tokens forming the output, or the sequence of logical form AST nodes in some orders.

2.3 Deep learning

The state of the art approaches in deep learning models to generate trigger-action programs [11, 12, 19] use a generative approach to predict the output program. Beltagy et.al [11] use a feed-forward neural network model to predict the sequence of the grammar rules that are applied to generate the derivation tree of the program. However, in practice the number of grammar rules explode as more new items have to learned by the system because of the compositional nature of the programs.

The more recent models [12, 19] use sequence to sequence neural networks to predict the output logical form by decoding the output tokens one at a time with the encoded input sequence information. By embedding word vectors and adding attention mechanism, the sequence to sequence model captures the lexical variety and paraphrases of programs while decoding programs with large number of parameters while adding focus to parts of the input that contribute the most. The extensions to the sequence to sequence models propose a sequence to tree approach [12] to capture the program structure and using grammar constraints [9] to generate only valid programs.

3 Deep Learning Model

We implemented the sequence-to-sequence neural network model detailed in [12]. The aim of the model is to parse the natural language input x and predict the correct program y, represented in a logical form. The model takes in inputs x as a sequence of words $x = x_0 x_1 \cdots x_{|x|}$ and predicts outputs y by tokens left to right $y = y_0 y_1 \cdots y_{|y|}$. The models splits the learning into an **Encoder** which encodes x into a vector representation E_x by applying transformations to the input sequence word embeddings $we(x)$ and a **Decoder** which learns to generate the output tokens in the sequence y_j by using the encoded information E_x and the information $D_{y<j}$ from the previously decoded tokens $y_0 y_1 \cdots y_{j-1}$.

3.1 Sequence-to-Sequence Model

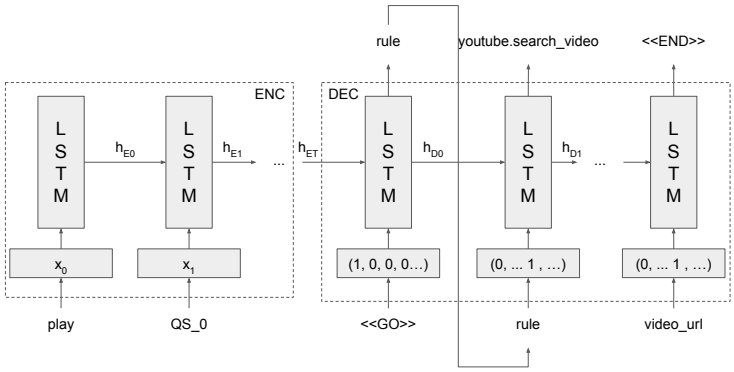


Figure 2: Sequence-to-Sequence (SEQ2SEQ) model detailed in this report

The model treats each token in the input x_i and the output y_j as a stage and learns $E_{x>i}$ and $D_{y>j}$ using $E_{x>(i-1)}$ and $D_{y>(j-1)}$ and the current token respectively. In this report, we focus on the 1-layer Recurrent Neural Network (RNN) model with each stage as a Long Short Term Memory (LSTM) unit as shown in Fig.2. Let $h_{Ei} \in \mathbb{R}^n$ and $h_{Dj} \in \mathbb{R}^n$ represent the hidden vectors at encoder time step i and decoder time step j respectively. Let $we_E(x_i) : V_x \to \mathbb{R}^w$ be the embedding function that generates the word vector given an input token x_i in the vocabulary V_x and $we_D(y_j) : V_y \to 0, 1^w$ be the output embedding function that generates a one-hot word vector for an output token y_j in the vocabulary V_y. We can recursively compute them by:

$$h_{E0} = \text{LSTM}(we_E(x_0)) \tag{1}$$

$$h_{Ei} = \text{LSTM}(h_{E(i-1)}, we_E(x_i)) \tag{2}$$

$$h_{D0} = \text{LSTM}(h_{E|x|}, we_D(\text{`«}\textbf{GO}\text{»'})) \tag{3}$$

$$h_{Dj} = \text{LSTM}(h_{D(j-1)}, we_D(y_{j-1})) \tag{4}$$

where the LSTM$(.)$ denotes the standard LSTM operations. We predict the output tokens y_j from the hidden state h_{Dj} by computing:

$$p(y_j|y_{<j}, x) = \text{softmax}(U \cdot h_{Dj} + b_y)^T we_D(y_j) \tag{5}$$

where $U \in \mathbb{R}^{|V_y| \times n}$ is the output parameter matrix and $b_y \in \mathbb{R}^{|V_y|}$ is the bias term. Then the output token y_j is generated greedily by computing:

$$\hat{y}_j = \underset{y \in V_y}{\text{argmax}}\, p(y|y_{<j}, x) \tag{6}$$

The sentences are appended with the start and end of string tokens and a special decoder token '«**GO**»' is used to signal the start of the decoder.

3.2 Attention Mechanism

Although, the hidden vectors of the input are not used directly in the computation of the output, it makes sense to weight parts of the input more when generating different parts of the output. To capture this effect, an attention layer is added (shown in Fig.3), which captures the encoder-side context $c_{Dj} \in \mathbb{R}^n$ before decoding the output symbol.

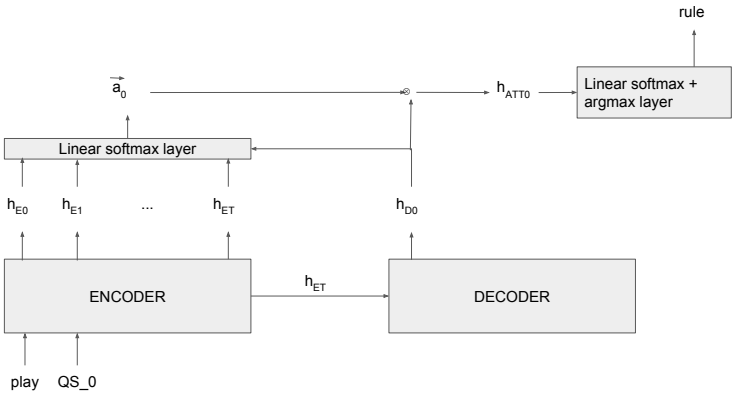


Figure 3: Attention Layer to SEQ2SEQ

Let $a_{Ei,Dj}$ ($a \in \mathbb{R}^{|x| \times |y|}$) be the attention score of the encoder stage Ei at decode time Dj. It is computed from the hidden state vectors as follows:

$$a_{Ei,Dj} = \frac{\exp(h_{Ei}^T \cdot h_{Dj})}{\sum_{k=0}^{k=|x|} \exp(h_{Ek}^T \cdot h_{Dj})} \tag{7}$$

Then the context vector c_{Dj} is computed as follows:

$$c_{Dj} = \sum_{i=0}^{i=|x|} a_{Ei,Dj} \cdot h_{Ei} \tag{8}$$

Using the context vector and the hidden state vector of the decoder, a new hidden state with attention $h_{Dj}^{att} \in \mathbb{R}^n$ is generated:

$$h_{Dj}^{att} = \tanh(V_1 \cdot h_{Dj} + V_2 \cdot c_{Dj}) \tag{9}$$

The generated h_{Dj}^{att} is used in Eq.5 instead of the original h_{Dj}.

3.3 Grammar Constraints

To generate only valid programs at inference time, we add a set of grammar constraints on the output logical forms, inspired by Xiao, et.al. [9]. We use a deterministic finite state automaton (DFA) to infer the constraints since the ThingTalk grammar is regular. Thus, at every stage of the Seq2Seq decoder, the DFA state s captures the set of allowable tokens $\{y\}$ that can be emitted in the form of the transition state matrix G_{sy}. Using these constraints, we transform the output logits as follows:

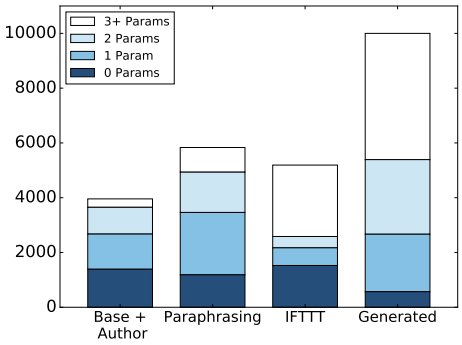
$$l_{Dj} = U \cdot h_{Dj} + b_y \tag{10}$$

$$\bar{l}_{Dj} = l_{Dj} - \infty \cdot \mathbb{1}[G_{s_j y} = \perp] \tag{11}$$

$$p(y_j | y_{<j}, x) = \text{softmax}(\bar{l}_{Dj})^T we_D(y_j) \tag{12}$$

$$\hat{y}_j = \underset{y \in V_y}{\text{argmax}}\, p(y | y_{<j}, x) \tag{13}$$

$$s_{j+1} = G_{s_j \hat{y}_j} \tag{14}$$



(a) Training sets

Dataset	# Sentences		# Programs	
	Prim	Comp	Prim	Comp
Paraphrasing	294	497	144	91
Scenarios	35	36	12	22
Composition	40	51	25	49

(b) Test set

Figure 4: Breakdown of the dataset

3.4 Model Training

The loss function is the cross-entropy between the predicted output $\hat{y}$ and the gold output over the training examples $(x^{(i)}, y^{(i)})$:

$$L^{(i)} = -\sum_{t=1}^{T} y_t^{(i)} \cdot \log(\hat{y}_t^{(i)}) \tag{15}$$

At training time the decoder layer is fed the whole gold sequence, while at inference time time $t+1$ in the decoder is fed the previously inferred token (after applying grammar constraints). At training time, we apply dropout between the sequence layers and the attention, and RMSProp is used to optimize the loss function.

4 Data Acquisition

The main challenge in building the Almond virtual assistant is the lack of real user data to train on. In this section we detail how we overcome this challenge. We build both a training set that is effective for the neural network algorithm, and a test set that is representative of real user data.[1]

Our dataset consists of pairs of sentence with their annotated ThingTalk program. We pre-process the sentences to identify argument values (quoted strings, numbers, hashtags, ...) and replace them with a special token that represents the type and the order (eg. QUOTED_STRING_0) in the sentence and in the program.[2]

4.1 Training Data Acquisition

Because a neural network requires a large corpus to train on, we construct our training portion of the dataset by combining different datasets that the Almond project has acquired. We show the relative sizes of our datasets in Figure 4a, and we show the number of parameters in the corresponding program as a indication of the program complexity.

The first set, which we define as the *Base* set, is built from Thingpedia. For each function, Thingpedia contains a list of *example sentences*. These sentences map to the primitive command corresponding to the function (e.g., for @gmail.receive_email, the sentence would be "notify me when I receive an email") and provided by the Thingpedia contributors. Because contributors are required to provide some examples when submitting a new entry to Thingpedia, we assume that these sen-

[1] This dataset was not acquired as part of the class, and was not acquired entirely by the authors. We describe it here for completeness only.

[2] In this section, the number of programs refers to programs that are distinct after preprocessing.

tences are always available, to give the neural network a baseline of lexical knowledge about each function. The base set contains 3028 sentences.

Thingpedia also contains a *confirmation string* for each function. In ThingTalk, it is possible to combine confirmation strings for individual functions to form a full description of the program. For example, the confirmation of @gmail.receive_email is "I receive an email on GMail" and the confirmation of @gmail.send_email" is "send an email to $to on GMail", where $to is a placeholder. Given that, from the program @gmail.receive_email $\Rightarrow$ @gmail.receive_email, $to = sender$ it is possible to generate "when I receive an email on GMail send an email to the sender on GMail". By sampling random ThingTalk programs and then constructing their corresponding confirmation sentence this way, we can mechanically generate a large set of data. This data has low linguistic variety but very high program variety, which ensures the neural network has good coverage of the ThingTalk program space. We generate 10000 sentences this way, and we call this portion of the dataset the *Generated* set.

Thingpedia contributors also have the option of providing full sentence, program pairs for the devices they submit. These sentences are highly representative of useful programs, because they come from the domain experts and developers who are themselves users of Almond. However, since they are written by only a few people, they show little linguistic variety. Additionally, because these sentences have to be submitted by developers and experts, they are comparatively more expensive than the other training sets. We combine the 929 sentences submitted by the Thingpedia contributors, the Almond developers and the report authors into the *Author* set.

To address the linguistic variety problem, we take advantage of the IFTTT website. In IFTTT, users are able to construct Trigger-Action programs that are similar in scope to ThingTalk, and then provide a natural language description of the recipe. These natural language sentences are often ambiguous, not intellegible or not in English, but for a subset of them, they provide a very high variance dataset of ways to refer to a Trigger-Action program. In the past, multiple groups [10, 11, 12, 20] have tried to use a set of recipes scraped from IFTTT as their only data source, but they found that the data was too noisy to be useful. This is especially true when attempting to recover recipe parameters from the description, as they are very often implied subtly, ambiguous or not specified at all. For this reason, we do not use their approach of directly converting the IFTTT representation to ThingTalk code. Instead, we take the subset of 62 recipes that can be mapped to equivalent ThingTalk programs, and then we write those programs ignoring the parameters, unless they are explicitly specified in the description. This set has a very large linguistic variety, because we obtain 5191 sentences corresponding to 64 programs. We call this set the *IFTTT* set.

Finally, to address at once the linguistic variety and the program variety, we turn to the idea by Wang, Berant et al. [4]. Given the confirmation sentences we generate from randomly sampled ThingTalk programs, we ask Mechanical Turk workers to write 5 different paraphrases in their own words. This set provides linguistic variety, and provides a source of non-compositional linguistic constructs, such as "auto reply to my emails" as a paraphrase for @gmail.receive_email $\Rightarrow$ @gmail.receive_email, $to = sender$. On the other hand, to keep the confirmation sentence understandable for the Mechanical Turk worker, we need to reduce the program complexity and thus, the program variety. Additionally, this set is the most expensive to acquire (with the exception of the Author set) and might not always be available. We call this the *Paraphrasing Train* set, and it consists of 4833 sentences.

4.2 Testing Data Acquisition

Neither the IFTTT nor the Generated data sets contain virtual assistant commands that would be written by real users. The Base and the Author contains some useful programs, but they are only written by a few people with intricate knowledge of the system. Therefore we are left with the problem of finding a test set that would be representative of realistic users. We use different approaches to obtain test sets that are increasingly more realistic, starting from data that is very similar to the training data and moving to data that is as close as possible to real user data.

As a baseline, we construct a test set using the same mechanism as the Paraphrasing Train set, which contains data written by humans. If the algorithm is able to understand the sentences in this paraphrasing test set, then at least it has acquired a basic level of understanding of the natural language. For this reason, we also construct a Paraphrasing Dev set. To ensure that the neural

network is robust and does not overfit on programs, we construct these datasets in a way that the same program does not appear both in the train and in the dev set.[3]. We preserve this constraint for the compound part of the test set, but not for the primitive test, because otherwise the test set would have too few programs and not be statistically significant.

The paraphrase sentences are still not a very realistic test of Almond as a virtual assistant, because the sentences are very verbose and explicit. To acquire more realistic test data, we conducted two experiments. In the first experiment, we create a few scenarios that describe a day in life of a person, and we ask Mechanical Turk workers to come up with a command that would make their life easier. We collected 327 sentences this way, of which 71 were both meaningful and in scope, which we annotated with the correct program manually. In the second experiment, we present a Mechanical Turk worker with a page showing all the supported Thingpedia functions, and a few examples of composition, and we ask them to choose to two functions to combine into a compound command. We collect 200 sentences, of which 91 are in scope and we annotated with the correct program. Both these methods present a fair illustration of an user who uses the virtual assistant for its purpose and understands the capabilities before supplying real commands. We show the breakdown of our test sets in Figure 4b.

5 Evaluation

In this section we evaluate our parser first on our validation set, and then on test sets of increasing complexity.

We implemented our parser in Python using Tensorflow 1.0 [21]. Preprocessing was implemented in Java using CoreNLP [22]. We use the pretrained GloVe [23] vectors of size 300 trained on Common Crawl as our word vectors, and we do not train the word vectors.

5.1 Model Validation & Tuning

We train our model by minimizing the cross-entropy loss of the predicted sequence against the gold sequence in our data set. We then tune the hyperparameters (hidden size, cell type, number of seq2seq layers, regularization) by choosing the model with the highest development accuracy.

We experiment both with and without the attention layer, and with and without the grammar constraints. We show the results of our model tuning in Figure 5a. We observe that in fact the grammar constraints do not have a measurable impact on accuracy. Qualitative investigation of the two models showed that while the predictor with grammar constraints generate valid programs, they do not approach correct programs because the error had been made in the earlier stages of the decode process. We believe that doing a non-greedy decode process like Beam search can help improve such cases. Further, the attention layer improves the accuracy by 16% and the recall by 10%, which is in line with the findings of Lang2Logic [12].

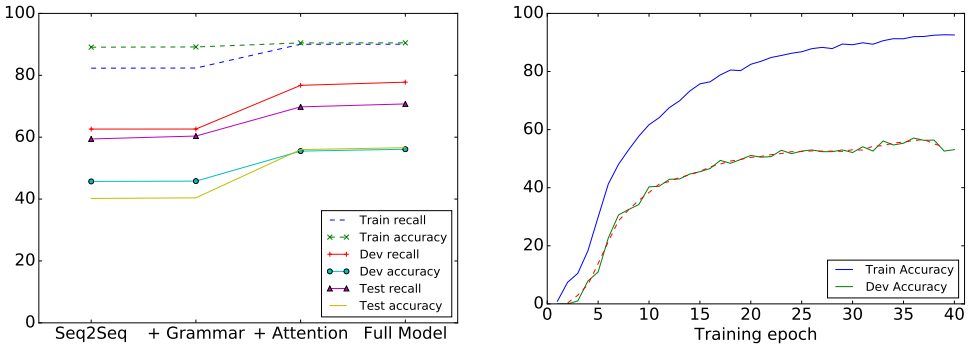
We train for a fixed number of epochs, and then choose the model with the highest development accuracy. We choose to train for 40 epochs, based on the curve of development accuracy (Figure 5b). Empirically, we found the best model is a 1-layer LSTM with a hidden size of 175, with a dropout probability of 0.5 and no L2 regularization.

5.2 Comparison Metrics

In the rest of the section, we compare the performance to the existing semantic parser for the Almond virtual assistant, which is based on the SEMPRE framework. Our high level goals are to improve on correctness, coverage and extensibility. Specifically, we define the following three metrics:

- **Accuracy**: what percent of sentences can be parsed correctly? This is a measure of correctness on user input.
- **Recall**: what percent of programs can be parsed correctly, for at least one sentence? This is a measure of coverage of the program space.

[3]The program might still be in the Generated train set, because the Generated set includes a large portion of the set of valid ThingTalk programs.



(a) Comparison of different models

(b) The learning rate

Figure 5: Model tuning

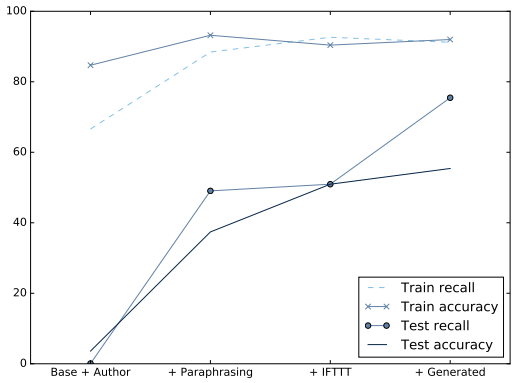


Figure 6: The contribution of the different training sets to the accuracy on the paraphrasing set.

- **Extensibility**: what accuracy can the parser achieve, on devices and domains never seen before?

We compare against the Almond parser as described in [1] and as currently deployed. This parser is trained on the Base, Author and Paraphrasing Train set, but not the IFTTT or Generated data, because it's not robust enough to those high variance datasets.

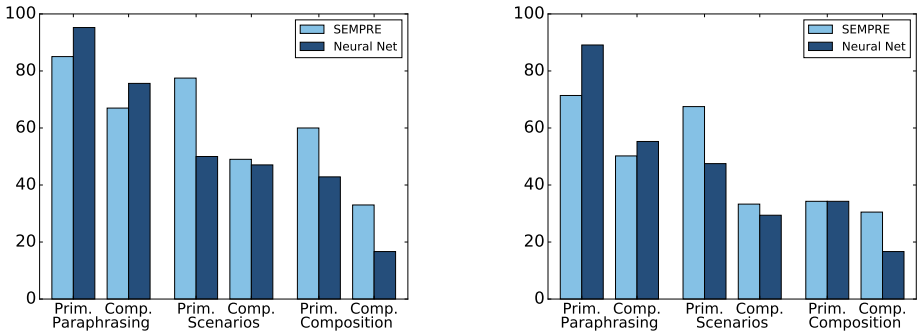
5.3 Sensitivity to Training

We first evaluate the relative importance of the different components of our training set. We train our parser on increasingly richer set of sentences, and we evaluate on the paraphrasing test set.

The results are shown in Figure 6. We first observe that the Base and Author set are too small and have too little linguistic variety, so they result in a large amount of overfitting. Adding the Paraphrasing Set increases both the linguistic variety and the program variety, and thus increases both accuracy (up to 37%) and recall (to 49%). The addition of IFTTT, which has high linguistic noisy variety but few programs, acts a strong regularizer and brings the accuracy up to the recall at 51%. Finally, the addition of the Generated set, which have a high variety of programs, reduces overfitting of programs and increases the recall again, up 75%, while increasing the accuracy only slightly to 55%.

5.4 Accuracy

In this section, we compare our parser to SEMPRE on the ability to interpret user input, by evaluating their accuracy on our different test sets.



(a) Correct function

(b) Correct program

Figure 7: The accuracy of our parser in our different test sets.

Previous work applying deep learning to synthesis of trigger-action programs [11, 12] have not considered the problem of identifying parameters correctly, so we measure both the ability to predict the program without parameters (**correct function**) and the full program, including any parameter (**correct program**).

We show the result for predicting the correct function in Figure 7a. We observe that our system is significantly better than the SEMPRE based system for the paraphrasing test, with an accuracy of 95% for primitives and 75%, compared to 85% and 67% for SEMPRE. We have to attribute this to overfitting (combined with the small amount of possible primitives, ignoring parameters), because our system shows no improvement or a significant decrease in the scenario test set (50% and 47% for primitive and compound resp., compared to 77% and 49% for SEMPRE) and in the composition set (43% for primitive and 16% for compound, compared to 60% and 33%). We also believe that SEMPRE does markedly better for primitive commands because its test set does not include Generated or IFTTT, and therefore the Base set is weighted comparatively more.

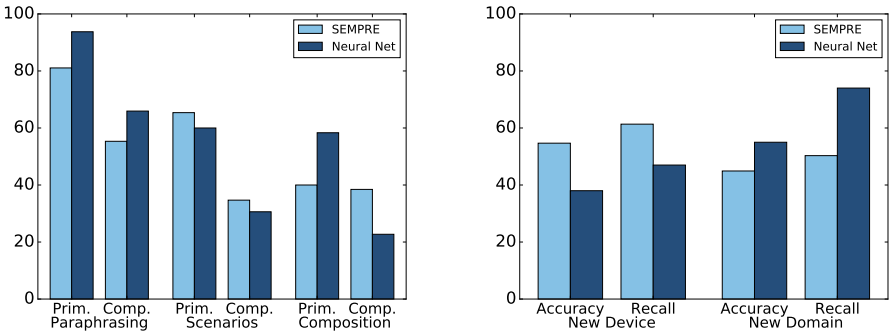
For predicting correct programs including parameters (Figure 7b), we observe similar results. On the paraphrasing test, our parser achieves an accuracy of 89% for primitives and 55% for compound, which is higher than SEMPRE at 71% and 50% resp. For the scenario test, our parser has an accuracy of 47% for primitives and 29% for compound, which is lower or same as SEMPRE at 67% and 33%. For the composition test, we obtain an accuracy of 34% for primitives and 16% for compound, compared to SEMPRE obtaining 34% and 30% resp. This results show significant overfitting in both systems, primarily caused by the use of paraphrasing as a source of training data. They also show that, for scenario and composition test cases, both parsers can either guess the correct program fully or they cannot find the correct function at all, owing to training on synthetic sentences with low linguistic variance.

We observe also that, in the real system, the user is given the ability to confirm that the assistant interpreted the program correctly by choosing from a list of 3 programs, thus as long as the correct program is among the top 3 choices returned by the SEMPRE algorithm (which is beam search based) the assistant is usable, even if the exact accuracy is low. For the neural network based system, this is not possible, because the greedy search decoder only outputs one program, which makes it significantly less usable in practice, at the same level of accuracy.

5.5 Coverage

In the next experiment, we evaluate the ability of parser to cover the ThingTalk program space. We test our parser on our three test sets, and we measure the recall.

The results are shown in Figure 8a. We observe that for the paraphrasing test our parser performs significantly better than the SEMPRE based system on the paraphrasing test set, achieving a recall of 94% for primitives and 66% for compounds, compared to 81% and 55% for SEMPRE. We attribute this fact to training the neural network with the Generated set and to some overfitting of the programs. This increase does not translate to the scenario test set (the difference is 1 or 2 programs and



(a) Recall of our parser on different test sets. (b) Extensibility to new devices and new domains.

Figure 8

is not statistically significant). For the composition experiment, the neural network does much better for primitives (up to 58% compared to SEMPRE at 40%) and much worse for compound (22% compared to 38% for SEMPRE), which suggests there is overfitting in both systems in different directions.

5.6 Extensibility

One of the goals of the Thingpedia project is to be able to collect all the world knowledge about devices and web services. To do that, the parser must be able to generalize to devices and domains that have not been seen before – otherwise the amount of training data to acquire would be too large. In this section, we evaluate our parser on test sentences of an unseen device and an unseen domain that had no manual training, compared to the existing SEMPRE-based Almond parser. We show the result of this evaluation in Figure 8b.

For the first extensibility experiment, we remove one device from our paraphrasing set, and we train our model on the remaining data. Because the Base data is always available, and the Generated data is easy to acquire, we don't remove the device from those sets. We choose to remove Slack because it has good mix of triggers and actions to receive and send data. On a testing set composed only of paraphrased sentences that include Slack, our parser obtains an accuracy of 38% and a recall of 47%, whereas the previous SEMPRE system has an accuracy of 55% and a recall of 61%. We attribute this fact to the ability of SEMPRE to generalize from other devices in the same domain, by reusing the knowledge about similar words such as 'send' or 'receive', whereas the neural network would predict a different device such as GMail or Phone when seeing those words.

For the second extensibility experiment, we remove all the Thingpedia communication devices (Slack, GMail, Phone and Twilio) from the training set, and we test on data includes only programs that mention one of those 4 devices. On this set, our system obtains an accuracy of 55% and a recall of 74%, which is significantly higher than the SEMPRE result of 45% accuracy and 50% recall. We attribute the higher accuracy to the higher recall, which in turn is caused by the presence of the Generated set (which lets the neural network learn about composition) and the fact that GMail in fact has a lot of composition cases. We also explain this result with the use of a dense word representation, through which the neural network can learn about 'send' or 'receive' without ever seeing those words in any input.

6 Conclusion & Future Work

In this report, we show that the deep learning model has the potential to outperform the current SEMPRE-based system to identify a larger set of trigger-action programs. In particular, we show that it increases the percentage of parsed programs to 94% for primitive commands and 66% for compound commands, on the paraphrasing test set. On the other hand, it needs a larger set of training data to learn a more robust linguistic model, and currently underperforms on realistic user inputs.

We believe that, once the system is performing on par with SEMPRE on our benchmarks, we will be able to deploy it in production. To do so, we must address the issue of usability, as, unlike SEMPRE, our parser does not give the user a choice of correcting the assistant. We must also address the issue of online extensibility, as in the current system, even though the parser is able to cope with the lack of data for new devices and new domains, it cannot do so in an online fashion and must be trained from scratch on a larger output space.

We hope that, as we collect better training data by crowdsourcing and experiment with compositional deep learning models, we will be able to surpass the limitations of the current SEMPRE-based systems on all benchmarks.

Acknowledgments

We would like to thank the rest of the Mobile and Social Research Group for the help and the funding to collect the data for this project.

References

[1] Giovanni Campagna, Rakesh Ramesh, Silei Xu, Michael Fischer, and Monica S. Lam. Almond: The architecture of an open, crowdsourced, privacy-preserving, programmable virtual assistant. In *Proceedings of the 26th International World Wide Web Conference (WWW-2017)*, 2017.

[2] Monica S. Lam, Giovanni Campagna, Jiwon Seo, and Michael Fischer. A distributed open social platform for mobile devices. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*, MOBILESoft '16, pages 173–174, New York, NY, USA, 2016. ACM.

[3] Panupong Pasupat and Percy Liang. Compositional semantic parsing on semi-structured tables. In *Proceedings of the 53nd Annual Meeting of the Association for Computational Linguistics (ACL-15)*, pages 1470–1480, 2015.

[4] Yushi Wang, Jonathan Berant, and Percy Liang. Building a semantic parser overnight. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics (ACL-15)*, pages 1332–1342, 2015.

[5] Khai N Truong, Elaine M Huang, and Gregory D Abowd. Camp: A magnetic poetry interface for end-user programming of capture applications for the home. In *International Conference on Ubiquitous Computing*, pages 143–160. Springer, 2004.

[6] If This Then That. `http://ifttt.com`.

[7] Blase Ur, Elyse McManus, Melwyn Pak Yong Ho, and Michael L Littman. Practical trigger-action programming in the smart home. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 803–812. ACM, 2014.

[8] Jonathan Berant and Percy Liang. Semantic parsing via paraphrasing. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (ACL-14)*, pages 1415–1425, 2014.

[9] Chunyang Xiao, Marc Dymetman, and Claire Gardent. Sequence-based structured prediction for semantic parsing. *Proceedings Association For Computational Linguistics, Berlin*, pages 1341–1350, 2016.

[10] Chris Quirk, Raymond Mooney, and Michel Galley. Language to code: Learning semantic parsers for if-this-then-that recipes. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics (ACL-15)*, pages 878–888, 2015.

[11] I Beltagy and Chris Quirk. Improved semantic parsers for if-then statements. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL-16)*, pages 726–736, 2016.

[12] Li Dong and Mirella Lapata. Language to logical form with neural attention. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL-16)*, pages 33–34, 2016.

[13] Rohit J Kate, Yuk Wah Wong, and Raymond J Mooney. Learning to transform natural to formal languages. In *Proceedings of the National Conference on Artificial Intelligence*, volume 20, page 1062, 2005.

[14] David L Chen and Raymond J Mooney. Learning to interpret natural language navigation instructions from observations. In *AAAI*, volume 2, pages 1–2, 2011.

[15] Luke S Zettlemoyer and Michael Collins. Learning to map sentences to logical form: structured classification with probabilistic categorial grammars. In *Proceedings of the Twenty-First Conference on Uncertainty in Artificial Intelligence*, pages 658–666. AUAI Press, 2005.

[16] Yuk Wah Wong and Raymond J Mooney. Learning synchronous grammars for semantic parsing with lambda calculus. In *Annual Meeting-Association for computational Linguistics*, volume 45, page 960, 2007.

[17] Rohit J Kate and Raymond J Mooney. Using string-kernels for learning semantic parsers. In *Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics*, pages 913–920. Association for Computational Linguistics, 2006.

[18] Jacob Andreas, Andreas Vlachos, and Stephen Clark. Semantic parsing as machine translation. In *ACL (2)*, pages 47–52, 2013.

[19] Xinyun Chen, Chang Liu Richard Shin Dawn Song, and Mingcheng Chen. Latent attention for if-then program synthesis. *arXiv preprint arXiv:1611.01867*, 2016.

[20] Chang Liu, Xinyun Chen, Eui Chul Shin, Mingcheng Chen, and Dawn Song. Latent attention for if-then program synthesis. In *Advances in Neural Information Processing Systems*, pages 4574–4582, 2016.

[21] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Savannah, Georgia, USA*, 2016.

[22] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. The Stanford CoreNLP natural language processing toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*, pages 55–60, 2014.

[23] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.