Generating Predicate Logic Expressions From Natural Language

Oleksii Levkovskyi
College of Engineering and Computing
Nova Southeastern University
Fort Lauderdale, Florida USA
ol150@mynsu.nova.edu

Wei Li
College of Engineering and Computing
Nova Southeastern University
Fort Lauderdale, Florida USA
lwei@nova.edu

Abstract—**Formal logic expressions are commonly written in standardized mathematical notation. Learning this notation typically requires many years of experience and is not an explicit part of undergraduate academic curricula. Constructing and comprehending logical predicates can feel difficult and un-intuitive. We hypothesized that this process can be automated using neural machine translation.**

Most machine translation techniques involve word-based segmentation as a preprocessing step. Given the nature of our custom dataset, hosts first-order-logic (FOL) semantics primarily in unigram tokens, the word-based approach does not seem applicable. The proposed solution was to automate the translation of short English sentences into FOL expressions using character-level prediction in a recurrent neural network model.

We trained four encoder-decoder models (LSTM, Bi-directional GRU with Attention, and two variants of Bi-directional LSTM with Attention). Our experimental results showed that several established neural translation techniques can be implemented to produce highly accurate machine translators of English sentences to FOL formalisms, given only characters as markers of semantics. We also demonstrated that attention-based enhancement to the encoder-decoder architecture can vastly improve translation accuracy.

Keywords—machine learning, neural machine translation, NLP, predicate logic

I. Introduction

Semantic Parsing is a widely studied field in Neural Machine Translation, generally defined as the task of converting natural language to some logical representation. A commonly studied representation of natural language is first-order logic (FOL) representation. This form resolves ambiguity, as it represents entities and relationships between them as quantifiable variables and functions that take arguments. Referring to the examples in recent literature on semantic parsing [1] statements such as "Every prime greater than two is odd." can be expressed in first order logic (FOL) [2] as

$$\forall x.prime(x) \wedge greater(x, 2) \rightarrow odd(x) \qquad (1)$$

We present several systems that can automate the generation of such expressions. Logical formalisms can be written in a variety of representation systems and notation standards. Several such systems have been proposed. One example is Lambda Dependency-based Compositional Semantics [3], where lambda calculus notation serves as the backbone for the logic formalism's grammar and semantics. Another format [4] is graph-based – entities identified in a natural language expression, along with the relationships between them, are represented as a graph connecting nodes (entities) with labelled edges (relationships). This graph-based representation is known as an Abstract Meaning Representation (AMR) graph. Most commonly, especially in an academic setting, logical formalisms are represented in standard mathematical notation.

We trained several neural parser models, the architectures of which are inspired by state-of-the-art machine learning methodologies. The encoder-decoder framework has taken a lead amongst the various methodologies we reviewed during the architecting process [5][6]. Using this framework, we implemented a set of models with varying hyperparameters as performance-enhancing internal modules. Our contribution to this domain nests in discovering 1) the aptitude of recurrent encoder-decoder models to accurately extract, and learn to construct, semantics from character (unigram) representations of input and target sequences with non-similar grammars, 2) the differences in performance between Long-Short Term Memory (LSTM) [7] and Gated Recurrent Unit (GRU)-based [8] sequence-to-sequence parsers, and 3) the impact of integrating attention-based decoding on the performance of the baseline model. We also compiled a custom, publicly available dataset of English-FOL sequence pairs, on which each of our model was trained and tested.

This document is organized as follows. Section II provides a brief literature overview. Section III outlines the custom dataset design process. Section IV describes the methodology behind the presented model. Section V logs the training process. Section VI presents the experimental results, and Section VII offers a discussion of obtained results.

II. Literature Overview

One legacy approach to automated semantic parsing, is a rule-based system: a syntactic parser for constructing a parse tree, which is then mapped (according to a set of rules) to a domain-specific language [9]. More recently, semantic parsing is solved primarily using statistical methods [1][10][11][12]. A common theme amongst many recent statistical parsing methodologies, is the use of Combinatory Categorial Grammars (CCGs) [13] for mapping individual words in natural language to their respective syntactic representations in the logic form domain. CCG-based semantic parsers are ubiquitous in the problem domain of natural language formalization and are therefore worth exploring as a starting point for our experiment.

Most recent solutions to problems of automated natural language translation rely on deep neural networks. An increasingly popular application of deep neural networks for NLP is generating SQL queries from natural language expressions [15][16][17]. These methodologies are relevant to the topical problem because the structure of an SQL query involves elements of set theory and predicate logic. SQL and predicate logic are both formalisms that share some logical constituent parts. We expected performance overlap between a deep neural network-based SQL generator and a hypothetical predicate logic generator. The methodologies used in neural parsers of English expression to SQL queries, extensively employ the encoder-decoder framework, in particular, with LSTM cells used as encoder-decoder modules [17].

Dong and Lapata [5] proposed a sequence-to-sequence (seq2seq) neural parser for translating English queries (simple questions asking for highly specific data) into a specialized formal language that can be utilized in automated question answering. Their baseline model is defined as a fusion of single-layer LSTMs, and learns to map the semantics of word embeddings, derived from the input data, as opposed to isolated characters, n-grams of characters, or whole words. Other recurrent units such as GRUs and Standard RNNs, although explored only in a handful experimental settings [19][20], were worth exploring further in our own research – in particular, the implications of replacing the LSTM cell with a GRU cell.

III. The Dataset

A. Ground Truth

We have analyzed the utility several public-domain datasets containing logical formulae for the purpose of our experiment. Our goal was to utilize a robust collection of pre-derived English-FOL pairs. Several benchmark datasets have been considered, including the Geo880 and Free917 query data sets. These include elements of lambda calculus, introducing unneeded complexity. The SNLI corpus, another strong candidate, features FOL expressions with embedded POS tags. We believed it was possible to bypass the use of POS tags in the final dataset in order to train a performant model.

Roughly 100,000 simple English sentences were scraped from the English Web 2015 Corpus (enTenTen15), using a specialized querying engine, and a set of queries representing parts of speech (POS) combinations. The querying engine, hosted as a web platform known as Sketch Engine (https://www.sketchengine.eu/), provided us with the ability to do structured query-based scraping. Based on the queries used, a total of 6 distinct sentence categories (termed Parts-of-Speech Combination Classes, or POSCC's) were derived. We hypothesized that a variety of sentence structures would make the translation model more apt at mapping natural language semantics to the semantic of FOL. Hence, the dataset's English sentence classification is defined by the combination of POS:

1) [all | every | some | no] N V [N|J] [or | and] [N | J]
   a) *"All women wore dresses."*
2) some N V [N|J] and some N V [N|J]
   a) *"Some theories are false and some facts are irrelevant."*
3) all N [and | or] N V
   a) *"All professors and students agree."*
4) all N that [are | were] N | J [are | were] N | J
   a) *"All men that were present were prepared."*
5) P V N [and | or] N
   a) *"He drinks coffee or tea."*
6) P [do | does] not V N
   a) *"She does not ride bicycles."*

The notation used for category abstraction loosely represents the query language defined by Sketch Engine (dubbed "CQL" by the platform creators). The italicized capital letters represent parts of speech without explicitly stating their tense, plurality or singularity: "N" marks nouns, "V" marks verbs, "J" marks adjectives and "P" marks pronouns. Entities in square brackets represent interchangeability (i.e. A or (|) B). Entities in parentheses represent optional groups. The set of quantifier words is limited to "all", "some", "every", "no", "nothing".

The queried set of English sentences comprises half of the required dataset. Each sentence was paired with a ground-truth logical expression that is, to a sufficient extent, semantically equivalent. A valid candidate for providing ground truth labels, is a CCG-based formal semantic parser (ccg2lambda) that generates logical formalisms from English sentences and outperforms state-of-the-art first-order systems [21]. Given the following English sentences taken from the training dataset:

1) *All women ordered coffee or tea.*
2) *Some actions are not specific.*

The following formalisms are derived:

1) *all x.(_woman(x) -> exists y.((_tea(y) | _coffee(y)) & _order(x,y)))*
2) *exists x1.(_action(x1) & -_specific(x1))*

In the given notation (formally known as MathML), a relationship between the quantifier ("all") and the quantified ("women") is represented as some lambda term x that is subject to predicates derived from verbs and nouns ("tea", "coffee", "order"). The relationship between the lambda term x and the predicates, is rendered as a set of functions and their logical operators.

The ccg2lambda parser was used for labelling English sentences with logical expressions. Careful measures were taken to ensure that the dataset meets an acceptable standard of quality. Sentences which the parser failed to translate properly, were not included in the final dataset, and only valid sentence-expression pairs were kept.

B. Dataset

The entire corpus of labelled sentences is sorted into a group of 42 text files (21 for English sentences [from here on, source sequences], and 21 for derived logical expressions [target sequences]). A file contains a list of character strings, separated by newlines (full-stop-terminated, in the case of source sequences). Each source or target sequence-containing file is homogenous in its POSCC (as per the 7 Classes outlined prior). Namely, each file pair contains expressions belonging to one, and only one specific POSCC. Given that there are 7 total Classes, and 21 files encompassing the sentences belonging to each POSCC, some subsets of the total file collection represent the same POSCC's.

Given the composition of the POSCC's, each source sequence is relatively short and simple in grammatical structure. The derived target sequence is often considerably longer than the source. There is a 50% margin between the length (in characters) of the longest source sequence and the length of the longest target sequence (81 and 162 characters, respectively). A source sequence file may contain the following sequences, sampled from the ordered list at random indices. The corresponding target sequence file places the target sequences at the same indices. The pairing, therefore, is defined strictly by the position of corresponding sequences in a given file.[1]

IV. The Methodology

A. Data Preprocessing

The unfiltered dataset contains 103,847 sentence pairs. The final dataset, with "faulty" sentences taken out contains 93,248 (from here onwards, N) valid sentence pairs, resulting in an 89.8% trainable yield. For the training phase, 80% of the data are separated from N, and the remaining 20% was reserved for the test phase (see Results section). In addition, the training set was further split according to the 80:20 ratio, where 20% of the training set was used for validation during training. The rationale behind this split comes from the widely known Pareto principle, namely that 80% of observed phenomena are the result of 20% of causes. Applying this principle in relation to our training dataset, we hypothesized that 80% of the data should contain enough entropy in its distribution, for the trained model to generalize over the remaining 20%.

Every source sequence sample x in the source set $S = (x_1, x_2, \ldots, x_N)$ and for every target sequence sample y in the target set $T = (y_1, y_2, \ldots, y_N)$ the following transformation is applied. Let C be the set of unique characters in the sequence sets S and T, respectively C_S and C_T. We build vocabulary mappings of character indices for each unique characters in S and T, resulting in V_S: $C_S \rightarrow \{0,1, \ldots 39\}$ and $V_T : C_T \rightarrow \{0,1, \ldots 50\}$. Since the fixed vocabularies are relatively small, we did not use any reserved symbols for out-of-vocabulary characters. Using the mappings, we replace the sample x and y sequences in S and T, into matrices $x^T_{m \times n}$ and $y^T_{p \times q}$ where:

$$m = |C_S|,\ p = |C_T|,\ n = max(S),\ q = max(T) \qquad (2)$$

Here, *max* computes the length of the longest sequence in the source and target sets. Each column in $x^T_{m \times n}$ and $y^T_{p \times q}$ is the one-hot-encoded representation of the character indices mapped by V_S and V_T. Because not all sequence in S and T are of the same length, most samples are padded with a $1 \times | C |$ zero matrix, to ensure uniformity of length. Note that, prior to the transformation above, each sequence in set T was prefixed with the start character '\t' and suffixed with the end character '\n'. Thus, we have converted each sample in S and T to a character-wise, one-hot-encoded representation.

B. Model

A LSTM-based encoder-decoder model outperforms other legacy models [22], such as SMT phrased-based systems [23] scoring higher on the BLEU [24] scale (34.8 versus 33.3 on the same dataset) and is able to process long sequences well. A standard RNN [25] could also be trained to make sequence-to-sequence mappings, but only if the input and output sequences have a fix length which is known in advance.

1) Model A

This is the baseline model from our model set. Formally speaking, the model computes a vector of character predictions, given the state of a Long Short-Term Memory (LSTM) cell. Just as in [22], the LSTM cell is used to estimate the probability $p(y_1, \ldots, y_{L'} \mid x_1, \ldots, x_L)$ for each x in S and each y in T (note that sequences lengths L and L' can be different). This conditional probability is computed as:

$$(y_1, \ldots, y_{L'} \mid x_1, \ldots, x_L) = \prod_{t=1}^{L'} p(y_t \mid \upsilon, y_1, \ldots, y_{t-1}) \qquad (3)$$

Here, υ is the representation of the sequence $(x_1, \ldots, x_L)$ in fixed dimensionality, obtained from the last hidden state of the LSTM, and the initial hidden state is a representation of the sequence itself. The probability distribution $p(y_t \mid \upsilon, y_1, \ldots, y_{t-1})$ over all characters in the target vocabulary, is given by the softmax function output. The purpose of placing a start character prefix ('\t') and an end character suffix ('\n') in each target sequence during the preprocessing stage, is to "allow the model to define a distribution over sequences of all possible lengths" [22]. As is the case in the plain LSTM model [22], the definition given above is not entirely consistent with our actual architecture of the baseline Model A in one significant way, namely that it uses two different LSTMs. The **Encoder LSTM** processes the input sequence and computes internal states (h,c) at each time step (or, after processing each character representation), which are used by the decoder to predict the next character. The **Decoder LSTM** is trained on the set of input sequences to predict each successor character in the target

[1] The dataset, along with the source code, can be downloaded at: https://github.com/alevkov/text2log

sequence, until the end character '\n' is reached. The decoder derives the "context" from the encoder (in the form of hidden states *h* and *c*) to make a prediction for the next character at a given time step. Using the teacher forcing technique [26], whereby the target (output) token is injected into the decoder as the next input token. The decoder can therefore make a more accurate prediction for the current token based on the prediction score of the previous token. In practice, the decoder is trained to generate output sequences 1 time step into the future, given the target sequence without the by-1 offset, and the context obtained from the encoder. We have replicated this technique in our model because teacher forcing is known to produce faster convergence during training.

Unlike Sutskever's architecture [22], Model A does not reverse the order of input sequences as a purported technique for improving performance. In our experimental set up, doing this actually reduced performance, to an insignificant degree.

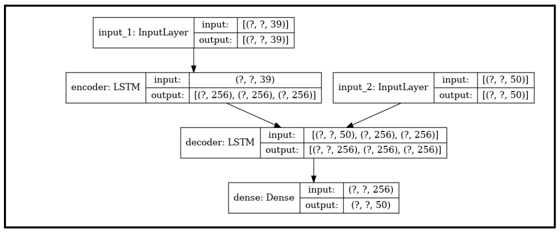

input_1: InputLayer | input: [(?, ?, 39)] | output: [(?, ?, 39)]
encoder: LSTM | input: (?, ?, 39) | output: [(?, 256), (?, 256), (?, 256)]
input_2: InputLayer | input: [(?, ?, 50)] | output: [(?, ?, 50)]
decoder: LSTM | input: [(?, ?, 50), (?, 256), (?, 256)] | output: [(?, ?, 256), (?, 256), (?, 256)]
dense: Dense | input: (?, ?, 256) | output: (?, ?, 50)

Fig. 1. Model A architecture diagram.

a) Number of hidden layers

There is no rule of thumb for choosing the appropriate number of layers for an LSTM network, and no sources, to our knowledge, define a quantitative "starting point". As a substitute for k-fold cross-validation (which is time-consuming), we use a specialized **formula** [27] that takes into account the number of degrees of freedom (input/output neuron count and total dataset size) present in the model:

$$N_{\text{hidden}} = \frac{N_{\text{samples}}}{\alpha * (N_i + N_o)} \qquad (4)$$

Here, N_i and N_o are the numbers of input and output neurons, respectively; α is a free parameter that takes values 2-20, and N_{samples} is the number of training samples. $N_i = 39$ and $N_o = 50$, corresponding to the number of unique characters in the source and target sequences (remembering that the final softmax layer produces a 1×50 probability distribution vector over each target character).

The value α is chosen to be quite low, specifically $\alpha = 3.2733$ resulting in $N_{\text{hidden}} \approx 256$ We want to keep α quite low for this model, because the architecture is quite simple, and the learning power of the neural network must be maximized. A higher α which results in $N_{\text{hidden}} \approx 128$, for example, produces a worse performing model, which is an indication of underfitting.

b) Number of samples in minibatch

We attempted to find a "sweet spot" that would result in a reasonably fast convergence and reduced training time overall, while preserving the ability to minimize generalization error. It is known that larger batch sizes result in a larger generalization error due to the reduced amount of noise introduced to the weights when their values are updated using calculated gradients [28]. A batch size of B = 32 is a good starting value, although in our experience, values below B = 64 resulted in very slow training times. Therefore, it is assumed that B = 64 will introduce a sufficient amount of noise to the gradient estimator, without sacrificing training time.

c) Recurrent dropout rate

Dropout is the de-facto standard technique for preventing overfitting [29] although it is important not to set the recurrent Dropout rate to a very high value, because dropping too many connections between the recurrent units of the network will result in underfitting (a rate higher than 0.1 incurred worse performance of the model overall).

d) Number of training epochs

A "vanilla" LSTM-based model learns relatively slowly, given the chosen batch size and the number of latent recurrent units. We found 300 be the minimum required.

e) Gradient optimizer

The LSTM cell may suffer from the problem of exploding gradients [38] – an unintended drastic increase in the magnitude of weight updates due to accumulated error during gradient calculation. This can be mitigated by an adaptive gradient descent optimizer, such as the one proposed by Geoffrey Hinton – RmsProp [30], which produced a faster convergence than the Adam [31] optimizer, which is usually a more popular choice.

2) Model B

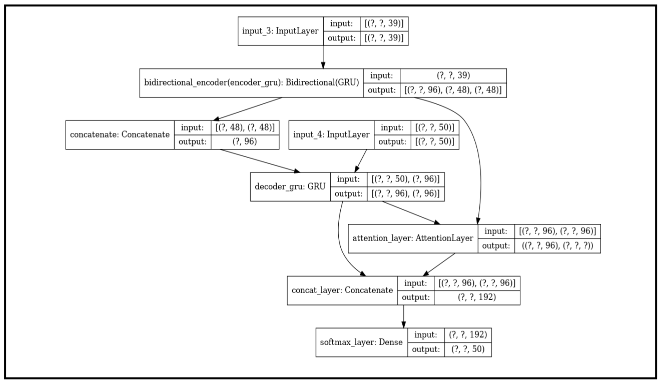

input_3: InputLayer | input: [(?, ?, 39)] | output: [(?, ?, 39)]
bidirectional_encoder(encoder_gru): Bidirectional(GRU) | input: (?, ?, 39) | output: [(?, ?, 96), (?, 48), (?, 48)]
concatenate: Concatenate | input: [(?, 48), (?, 48)] | output: (?, 96)
input_4: InputLayer | input: [(?, ?, 50)] | output: [(?, ?, 50)]
decoder_gru: GRU | input: [(?, ?, 50), (?, 96)] | output: [(?, ?, 96), (?, 96)]
attention_layer: AttentionLayer | input: [(?, ?, 96), (?, ?, 96)] | output: ((?, ?, 96), (?, ?, ?))
concat_layer: Concatenate | input: [(?, ?, 96), (?, ?, 96)] | output: (?, ?, 192)
softmax_layer: Dense | input: (?, ?, 192) | output: (?, ?, 50)

Fig. 2. Model B architecture diagram.

The key difference between Model A and Model B are: **1)** The LSTM cell has been replaced with a GRU cell. We set out to test the hypothesis proposed in a paper from Google [32] namely that GRU layers can have comparable performance with LSTM layers, while being composed of fewer parameters (GRU only has 1 hidden state), therefore reducing the memory footprint of the model. **2)** An improvement to the baseline, we introduce a second layer to the encoder cell, which processes the input sequence in reverse. The rationale behind this is that the "richer" context (unconstrained by time) will result in faster convergence [33]. **3)** We have integrated a context-based Attention module proposed by Bahdanau and Hinton. This addresses a limitation in the original encoder-decoder architectures [20][22]. The source sentence is compressed into a single-length vector, which is used by the decoder to compute the next target character. Because the encoder processes the input sequence in two temporal directions, the decoder makes predictions based on the future state, as well as the current state, reducing instances of "forgetting" words at the tail end of the sequence.

a) Number of hidden layers

Using the same formula for calculating latent dimensionality as for Model A. Given that the number of internal parameters in the GRU layer is doubled as a function of bi-directionality, and with the addition of the Attention layer, the latent dimensionality was reduced in order to prevent overfitting (otherwise, the total number of parameters would be very large), approximately equating to the number of trainable parameters in Model A. The value of α is chosen to be 17.45, yielding 48 hidden layers.

b) Recurrent dropout rate

Slightly higher than Model A, set at 0.2, to reduce the probability of overfitting, given a more complex architecture.

c) Number of training epochs

The "enhanced" Model B architecture converges (training loss equals validation loss) 10x faster than Model A (compared to 300 epochs with Model A), withing only 30 epochs. This value may even be somewhat excessive.

d) Gradient optimizer

Same as Model A. Learning rate is set to 0.0015, as opposed to the standard 0.001, for faster learning (without degradation in performance). Standard stochastic gradient descent was tested for posterity in the initial run of Model B's training; however, the training time with non-adaptive SGD proved to be unfeasibly slow, compared to Adam and RmsProp.

3) Models C (I) and (II)

This is a modified version of Model B, where the GRU layer has been replaced with the LSTM layer. As stated in the discussion of Model A, LSTM-based models give promising results (based on some metrics, such as the BLEU scale) in sequence-to-sequence translation tasks. Therefore, it was necessary for us to observe the combined performance of this particular recurrent module with the performance-boosting techniques discussed in the Model B section, in order to draw a direct performance comparison between LSTMs and GRUs. Iteration (I) of Model C differs only from Iteration (II) in the number of hidden layers in the LSTM cell. The Attention mechanism used in Model C is the same as the one used in Model B, for consistency. Relevant architectural details are presented below, although there aren't many deviations from Model B.

a) Number of hidden layers

For purely experimental reasons, Model C (I) was given a significantly larger parameter space (64) than Model C (II) (44) in order to demonstrate the potential for overfitting. Indeed, as stated in the Metrics & Results section, C (II) outperforms C (I), having fewer parameters, suggesting that the complexity of Model C demands fewer parameters in order to better generalize over the test set.

b) Number of samples in minibatch

Originally, this value was supposed to be borrowed from Model B's hyperparameter set. Due to memory limitations, the batch size was reduced to 48.

V. Training

The models were implemented using the Keras machine learning library in the Python programming language. At training time, Keras fitted the product of concatenating the input and target sequence tensors, to the sequence of target sequence tensors misplaced by 1 timestep forward in time (as explained in Section IV). The weight updating, as a function of gradient descent, is performed automatically by the Keras training script at the end of each epoch. A 20% cut of the training data was used for incremental validation of the running loss and accuracy measures, i.e. at the end of each epoch, this data passes through the network in "inference" mode in order to compute loss and accuracy averages after each epoch. The validation data does not "leak" into the training data, since a fixed split is maintained, and therefore does not affect the neuron weight distribution.

To avoid the overfitting problem, we configured the training script to halt prematurely when both the training and validation loss scores remain unchanged for 2 consecutive epochs. This is a slightly more restrictive variation on a widely adopted, effective strategy for reducing overfitting, known as Validation-based Early Stopping [35]. The original paper recommends monitoring the validation loss only. We have observed that our models (in particular, models B and C) do not suffer from a notable degree of overfitting, evident by the remarkably stable flat lining of the validation loss curve, even given an "excessive" epoch count. By adding the additional constraint of monitoring the training loss, we forced more epochs out of the training script, in an attempt to "squeeze out" maximum model performance, without the sacrifice of allowing the model to overfit.

The Keras training script maintained a log of loss and accuracy scores for the training and validation data sets. The training loss and accuracy scores are computed at the end of each batch's passing through the network, and the averages are recorded before each next epoch.

TABLE I. Validation Dataset Metrics

	Model A	**Model B**	**Model C(I)**	**Model C (II)**
Accuracy	99.953	99.998	99.997	99.998
Loss	0.047	0.002	0.003	0.002

Various optimization algorithms were tested during Model A's hyperparameter fine-tuning. Specifically, we tested plain Stochastic Gradient Descent [36] (without adaptive gradients and without momentum), Adam and RmsProp, which is a standard optimizer triad testing in instances of hyperparameter tuning. We evaluated the optimizer functions' performance based on the following metrics: 1) N_e , or the number of epochs it takes to stabilize the training and validation losses and 2) L_v, the validation loss at the stabilization point. We define "stabilization point" as the ordered epoch number E_t where, at the end of E_{t-1} and E_t, the values L_v and L_t (training loss) did not change.

TABLE II. OPTIMIZER FUNCTION SCORES

	SGD	Adam	RMSProp
N_e	179	300	297
L_v	0.221	0.049	0033

SGD was the worst-performing algorithm, though it produced the shortest-running training run. It was surprising to see RmsProp outperform Adam in the validation loss category, since it is known, from the inventors of the Adam optimizer, that it is able to rapidly locate local minima owing to the acceleration from momentum. The training run of Model A with Adam as the optimizer did not undergo early stopping, as the 300th epoch was reached without validation loss stabilizing.

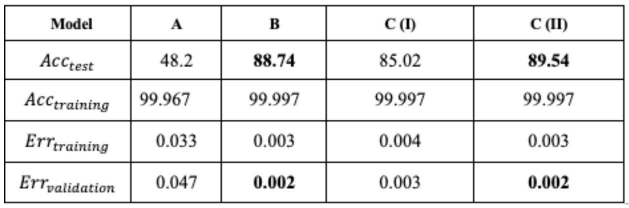
By the 10th epoch of Model B's first training run, the supposed optimum point had already been reached. We allowed the second training run of Model B to run for a total of 37 epochs. The final training and validation loss values were 0.003 and 0.002, respectively. The same values were seen in Model C's training.

VI. RESULTS

The test data set S_{TS} containing 20% of all source and target sentences was used to compute the accuracy score for each model. In the test script, each source sequence x_i (in English) from S_{TS} was presented to the inference algorithm outlined above, and the decoded sequence d_i was recorded. The decoded sentence was compared to the target sequence y_i from S_{TS}, and a "correctness" counter c was incremented by 1 when $d_i = y_i$.

Accuracy of the models were calculated by analyzing the correctness of the models' predictions. "Correctness" implies correct representation of an English sentence with the resulting logical predicate, identical to the target predicate from the test data set. The comparison for correctness is done character-by-character. This implies that the predictor must correctly predict the lemmatization of every natural-language subsequence (for example, it must lemmatize the word "brought" to the present tense "bring", and any other set of predicted characters would be penalized).

TABLE III. ACCURACY & ERROR SCORES FOR MODELS A, B AND C



Model	A	B	C (I)	C (II)
Acc_{test}	48.2	**88.74**	85.02	**89.54**
$Acc_{training}$	99.967	99.997	99.997	99.997
$Err_{training}$	0.033	0.003	0.004	0.003
$Err_{validation}$	0.047	**0.002**	0.003	**0.002**

To provide an interpretation of the scores, we must refer to the architectural changes introduced at each model iteration. Model A served as a proof of concept to demonstrate the efficacy of sequence-to-sequence translators with an LSTM-based encoder and decoder, as per [22]. Model A's drawbacks were immediately obvious: its parameter density resulted in long training times, while reducing the number of parameters resulted in comparably poor performance (34.1% accuracy score with only about 96,000 trainable parameters), as did *increasing* the number (42.4% with 800,000 parameters – an obviously excessive amount, given the size of the dataset and the input dimensions). Our resolution tuning of Model A landed us in the theoretical sweet spot. The final accuracy score, however, is not necessarily indicative of "poor" performance. In a binary classifier, which Models A, B and C are not, a score hovering near the 50% suggests that the classifier is "random", which is translatable to total failure. For a sequence-to-sequence translator, however, to accurately (down to the character) translate a completely new English sentence into a logical expression 48.2% of the time, is impressive on its own. As stated in the previous section, the output of the inference stage was tested against the expected sequence with 100% fidelity, all characters and their positions aligned. Even the smallest misspellings of a word were harshly penalized. Therefore, as a baseline, Model A is "good enough" by the toughest of standards.

Models B and C, however, take a far leap of improvement from the baseline. Interestingly, we did not observe a significant difference in performance mediated by the choice between LSTM and GRU. This is likely due to the nature of the data set. The sequences on both the input and target sides are quite short, and while LSTMs have the advantage of better recall of longer sequences the dataset's sequences are not long enough to observe this theoretical advantage in practice. On the contrary, the LSTM ground-truthiness of Model A's performance reveals that an accuracy slump with longer target sequences, with most errors occurring towards the end of the decoded sequence. Interestingly, the errors that Model A was most challenged with, occurred specifically with word spelling, while the semantics of the translated sentence remained intact. This is illustrated by the following translation, produced by Model A.

Source sentence: *"Some engineers are women and some women are teachers."*

Target sequence: *"exists x1.(_woman(x1) & exists x2.(_slower(x2) & (x1 = x2))) & exists x3.(_state(x3) & exists x4.(_work(x4) & (x3 = x4)))*

Model A correctly predicted the grammatical structure of the logical formalism but drastically missed the mark with the spelling of each subsequent word after "woman". It is unusual to see this type of selective recall with LSTM-based machine translators, although there is an intuitive hypothesis for this error. The LSTM cell is trained to preserve tightly coupled relationships between consecutive subsequences, or whole words. In the whole word case, the neuronal weight assignment at timestep t_e (end of word) is predicated on the assignment of all the weights at timesteps t_{e-1} t_{e-2} , … t_{e-e+1} (this is commonly referred to as the LSTM chain's "recall"). This predicated weight assignment eventually leads to the correct mapping of the “women” to “woman”. In the target sequence, whole words are embedded in a loosely coupled sequence of logical syntax, the decoding of which is not the result one consecutive sequence’s encoding. The network derives the keyword “exists” not from character relationship information (the word does not appear in the source sentence in any form), but instead from positional information, which is not as significant of a challenge as learning consecutive character relationships

VII. Discussion

Due to limitations of time, a word-based sequencing approach was not explored. The “character-by-character” approach makes more intuitive sense for the given data set. This is due to the difficulty of defining what exactly constitutes a “word” in the target sequence, since the bulk of the semantics are encoded in the character-level symbols. This was not found to be a detriment to the performance. We made a strong case for the ability of neural parsers to infer complex semantics at the character level. While significant, this is not entirely surprising, since character-level models **1)** are not challenged by the presence of out-of-vocabulary (OOV) tokens [37], **2)** can learn to model unexpected word morphologies and **3)** trivialize the problem of text segmentation, which is hyperparameter of infinite variations in itself (and can harm model performance). Character models are advantageous for multilingual machine translators, allowing for a shared parameters space between character sets of different languages [34]. This wouldn’t be possible with a word-based approach, where two separate word dictionaries are required for encoding and decoding.

The Attention mechanism’s role in character-based semantics can be visualized by weight “activation”. The *x* and *y* axes are labelled with the output and input sequence characters, respectively. The color intensity is a visual representation of the magnitude of weights assigned by the Attention mechanism to a particular character.

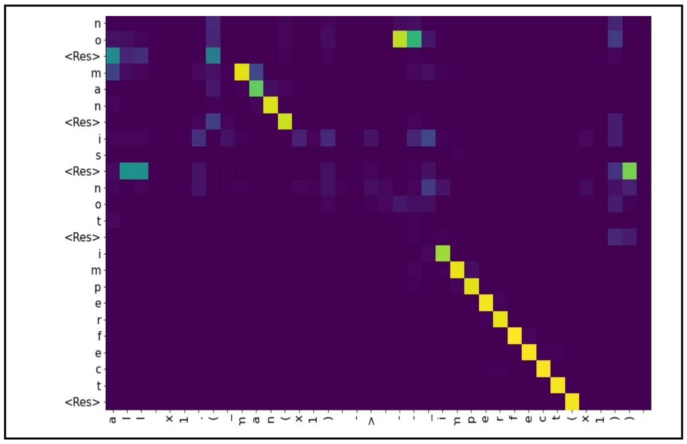


Fig. 3. Attention heatmap for the decoded sentence “No man is not imperfect.” produced by Model C(II). Note the co-activation between the word “no” and the double negative marker “--”. The two conjoined negation operators do not have equal activation values at the position of the word “no”.

VIII. Conclusion

We propose several variations of a character-based model for translating sentences in natural language to logical expressions. Our empirical results carry a notable implication that bi-directional encoder-decoder networks are able to infer and correctly reconstruct semantics from characters alone.

The models tested and the metrics obtained reaffirm the conclusions reached by past investigations that involve a similar experimental set up with our architectures being comparatively even less complex. We set up a baseline encoder-decoder model and demonstrated that a recurrent neural network with encoder and decoder LSTMs is decently performant at the task of translating English sentences into formal language. We demonstrated that the introduction of an Attention module to bi-directional LSTM and GRU-based decoders, can vastly improve their ability to make semantic mappings between English and FOL expressions, given a character-based representation of said sequences. The accuracy of the latter models appeared to be quite remarkable, given the semantic-less nature of the preprocessed data. We believe this is a significant contribution to the domain of neural machine translation using recurrent neural networks.

In addition to the technical contribution, we have designed an entirely novel, substantially large dataset containing English-FOL sentence pairs. The data format derivation process was outlined, citing specific frameworks and algorithms, and we have released the dataset into the public domain.

The next step would be to compare the performance of the existing character-based model variations, and their word-level counterparts, all other key parameters left unchanged.

References

[1] P. Liang, “Learning executable semantic parsers for natural language understanding,” *Commun. ACM*, vol. 59, no. 9, pp. 68–76, 2016.

[2] R. Smullyan, *First-Order Logic*. 1968.

[3] P. Liang, “Lambda dependency-based compositional semantics,” *arXiv [cs.AI]*, 2013.

[4] L. Banarescu, C. Bonial, S. Cai, M. Georgescu, K. Griffitt, U. Hermjakob, K. Knight, P. Koehn, M. Palmer, and N. Schneider, "Abstract meaning representation for sembanking." In *Proceedings of the 7th linguistic annotation workshop and interoperability with discourse*, pp. 178-186. 2013.

[5] L. Dong and M. Lapata, "Language to logical form with neural attention." *arXiv preprint arXiv:1601.01280*. 2016.

[6] H. Singh, M. Aggrawal, and B. Krishnamurthy, “Exploring neural models for parsing natural language into First-Order Logic,” *arXiv [cs.CL]*, 2020.

[7] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, “LSTM: A search space odyssey,” arXiv [cs.NE], 2015.

[8] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, “Empirical evaluation of gated recurrent neural networks on sequence modeling,” arXiv [cs.NE], 2014.

[9] W. A. Woods, “Progress in natural language understanding: An application to lunar geology,” in Proceedings of the June 4-8, 1973, national computer conference and exposition on - AFIPS ’73, 1973.

[10] R. J. Kate, Y. W. Wong , and R. J. Mooney, "Learning to transform natural to formal languages." In *AAAI*, vol. 5, pp. 1062-1068. 2005.

[11] R. J. Kate and R. 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 ACL - ACL ’06, 2006.

[12] L. Zettlemoyer and M. Collins, "Online learning of relaxed CCG grammars for parsing to logical form." In Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL), pp. 678-687. 2007.

[13] M. Steedman, “A very short introduction to CCG.” *Unpublished paper*. http://www.coqsci. ed.ac.uk/steedman/paper.Html. 1996.

[14] T. Kwiatkowksi, L. Zettlemoyer, S. Goldwater and M. Steedman, “Inducing probabilistic CCG grammars from logical form with higher-order unification.” in *Proceedings of the 2010 conference on empirical methods in natural language processing* (pp. 1223-1233). 2010.

[15] X. Xu, C. Liu, and D. Song, “SQLNet: Generating structured queries from natural language without reinforcement learning,” *arXiv [cs.CL]*, 2017.

[16] J. Guo *et al.*, “Towards complex Text-to-SQL in cross-domain database with intermediate representation,” *arXiv [cs.CL]*, 2019.

[17] V. Zhong, C. Xiong, and R. Socher, “Seq2SQL: Generating structured queries from natural language using reinforcement learning,” arXiv [cs.CL], 2017.

[18] D. Bahdanau, K. Cho, and Y. Bengio, “Neural machine translation by jointly learning to align and translate,” *arXiv [cs.CL]*, 2014.

[19] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, “Empirical evaluation of gated recurrent neural networks on sequence modeling,” *arXiv [cs.NE]*, 2014.

[20] K. Cho, B. van Merrienboer, D. Bahdanau, and Y. Bengio, “On the properties of neural machine translation: Encoder-decoder approaches,” arXiv [cs.CL], 2014.

[21] K. Mineshima, P. Martínez-Gómez, Y. Miyao, and D. Bekki, “Higher-order logical inference with compositional semantics,” in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, 2015.

[22] I. Sutskever, O. Vinyals and Q.V. Le, “Sequence to sequence learning with neural networks.” in *Advances in neural information processing systems* (pp. 3104-3112). 2014

[23] P. Koehn, O. J. Franz and M. Daniel, “Statistical phrase-based translation.” UNIVERSITY OF SOUTHERN CALIFORNIA MARINA DEL REY INFORMATION SCIENCES INST, 2003.

[24] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, “BLEU: A method for automatic evaluation of machine translation,” in Proceedings of the 40th Annual Meeting on Association for Computational Linguistics - ACL ’02, 2001.

[25] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[26] N. Toomarian and J. Barhen, “Fast temporal neural learning using teacher forcing,” in IJCNN-91-Seattle International Joint Conference on Neural Networks, 2002.

[27] “User hobs,” Stackexchange.com. [Online]. Available: https://stats.stackexchange.com/users/15974/hobs. [Accessed: 26-Feb-2021].

[28] Y. Bengio *et al*., “Rmsprop and equilibrated adaptive learning rates for nonconvex optimization.” corr *abs/1502.043.* 2015

[29] S. Nitish, G. Hinton, A. Krizhevsky, I. Sutskever and Ruslan Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting." in *The journal of machine learning research 15, no. 1*: *1929-1958*. 2015.

[30] T. Tieleman and G. Hinton, “Lecture 6.5–-RmsProp: Divide the gradient by a running average of its recent magnitude.”

[31] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” arXiv [cs.LG], 2014.

[32] R. Jozefowicz, W. Zaremba and I. Sutskever, “An empirical exploration of recurrent network architectures.” in *International conference on machine learning*, pp. 2342-2350. 2015

[33] Q. Yu, H. Zhao, and Z. Wang, “Attention-based bidirectional gated recurrent unit neural networks for sentiment analysis,” in Proceedings of the 2nd International Conference on Artificial Intelligence and Pattern Recognition - AIPR ’19, 2019.

[34] J. Lee, K. Cho, and T. Hofmann, “Fully character-level neural machine translation without explicit segmentation,” Trans. Assoc. Comput. Linguist., vol. 5, pp. 365–378, 2017.

[35] G. Montavon, G. Orr, and K.-R. Müller, Eds., *Neural networks: Tricks of the trade*, 2nd ed. Berlin, Germany: Springer Berlin, 2012.

[36] H. Robbins, “A Stochastic Approximation Method.” in *Annals of Mathematical Statistics*, 22, pp. 400-407. 1951.

[37] J. Chung, K. Cho, and Y. Bengio, “A character-level decoder without explicit segmentation for neural machine translation,” arXiv [cs.CL], 2016.

[38] R. Pascanu, T. Mikolov, and Y. Bengio, “On the difficulty of training Recurrent Neural Networks,” arXiv [cs.LG], 2012.