Self-Aware LSTM-Based Agents

Introduction

Goal: A Simple Form of Self-Awareness

The goal of this work is the creation of neural-network-based agents that can, given a knowledge base of Boolean logic sentences and a question regarding the value of a Boolean variable, tell us whether the variable in question is True, False, or unknown. In addition to solving simple Boolean logic problems, these agents should exhibit self-awareness and be capable of exploiting self-awareness to solve a problem cooperatively. Since the term self-awareness can be defined differently, we provide our definition of self-awareness below.

1. An agent must know whether it poses adequate knowledge to solve a problem.
    a. If it possesses adequate knowledge, it should solve the problem.
    b. If it lacks adequate knowledge, it should request help solving the problem.
2. An agent must know whether its internal knowledge base is contradictory because a contradictory knowledge base in Boolean logic permits any conclusion to be drawn.
    a. An agent with a contradictory state of knowledge should state this instead of trying to provide an answer since delivering a solution is impossible in this case.
3. An agent should be capable of providing the contents of its knowledge base upon request.
    a. This implies an agent has explicit knowledge of its knowledge base contents, another form of self-knowledge.

For this work, agents capable of performing the three tasks above are considered "self-aware." Self-awareness allows agents with contradictory knowledge states to warn the user and avoid providing a wrong answer. It also will enable agents to cooperate. For example, suppose that an agent lacks adequate knowledge to solve a problem and requests help from a second agent. The second agent's knowledge may be sufficient to allow the first agent to solve the problem. Here, agent self-awareness enables cooperation since the first agent knows it must request help (by being aware of the inadequacy of its knowledge state). The second agent has direct, symbolic knowledge of the contents of its knowledge base and can provide those contents on demand.

In this work, we create neural network-based agents capable of solving simple problems in Boolean logic. Neural networks for solving Boolean logic problems and other problems in symbolic mathematics are not new. Using neural networks for logical entailment, which is very closely related to this problem, is discussed in (Evans, Saxton, Amos, Pushmeet, & Grefenstette, 2018). Moreover, the closely related issue of using neural networks for symbolic mathematics is covered in (Lample & Charton, 2019). The present work differs from these two because it emphasizes self-awareness and cooperation in neural-network-based agents, a topic not covered in the two works cited above. Self-aware neural-network systems are surveyed in (Du, et al., 2020), and the type of self-awareness covered in this survey focuses on implementation and operational of the neural network itself, with the network having awareness of its own execution environment with the ability to optimize its dataflow, use of resources, and execution within that environment for performance optimization. This is a very different type of self-awareness than what is described here, since this paper focuses on awareness of one's state of knowledge, not on awareness of neural network execution environment and performance.

The present work is organized as follows. We start by defining the propositional (Boolean) logic problems our agents are trained to solve and then explain the behavior of the logical agents we have created. Next, we provide an overview of agent architecture, consisting of a knowledge base (implemented as a Python list of strings) and an LSTM neural network. The basics of LSTM neural networks and the particular encoder-decoder architecture used in the present work are then described.

Propositional Logic Problems

Our treatment of propositional logic very closely follows (Norvig, Artificial Intelligence: A Modern Approach, 4th Edition, 2021) with our notation, along with many direct quotations, from (Norvig, aima-python, n.d.). In particular, our notation is described as follows:

<table>
<tr><th>Operation</th><th>Book</th><th>Python Infix Input</th><th>Python Output</th><th>Python Expr Input</th></tr>
<tr><td>Negation</td><td>¬ P</td><td>~P</td><td>~P</td><td>Expr('~', P)</td></tr>
<tr><td>And</td><td>P ∧ Q</td><td>P & Q</td><td>P & Q</td><td>Expr('&', P, Q)</td></tr>
<tr><td>Or</td><td>P ∨ Q</td><td>P | Q</td><td>P | Q</td><td>Expr('|', P, Q)</td></tr>
<tr><td>Inequality (Xor)</td><td>P ≠ Q</td><td>P ^ Q</td><td>P ^ Q</td><td>Expr('^', P, Q)</td></tr>
<tr><td>Implication</td><td>P → Q</td><td>P | '==>' | Q</td><td>P ==> Q</td><td>Expr('==>', P, Q)</td></tr>
<tr><td>Reverse Implication</td><td>Q ← P</td><td>Q | '<==' | P</td><td>Q <== P</td><td>Expr('<==', Q, P)</td></tr>
<tr><td>Equivalence</td><td>P ↔ Q</td><td>P | '<=>' | Q</td><td>P <=> Q</td><td>Expr('<=>', P, Q)</td></tr>
</table>

Table 1: Logical Notation

Throughout this paper, the notation used in the "Python Output" column of Table 1: Logical Notation will be used, as is done in our demonstration system. Propositional logic variables are capital letters 'A' through 'J' inclusive.

Logical Agents

Agents in our system are trained to perform propositional inference.   An agent is presented with a set of sentences and is then asked about the truth value of a propositional variable. For example, a simple case of modus ponens:

Input Sentences	A A ==> B

Question	What is B ?
Answer	B is true.

Agents are based on an LSTM sequence-to-sequence neural network described in the following two Sections. When presented with input sentences and a question, an agent can respond in one of five possible ways:

1. True
2. False
3. Input sentences are contradictory, making an answer impossible.
4. Input sentences lack adequate information to answer the question.
    a. In this case, the agent will output a text string requesting "HELP."
5. Respond to a request for help from another agent by outputting one's knowledge base.

For example, a contradiction:

Input Sentences	~A A & B C ==> A
Question	What is C ?
Answer	Contradictory

In the example above, the sentence 'A & B' implies A is true, but '~A' means A is false. Because there is a contradiction above, *any* conclusion can be drawn from a contradiction, and the knowledge base itself is invalid. In such a case, an agent needs to report a contradictory state of knowledge from which no conclusions may be drawn.

In another example, we may have insufficient knowledge:

Input Sentences	~A
Question	What is C ?
Answer	Unknown HELP!

Hence, if a propositional variable is true or false based on the knowledge base's sentences, an agent should be able to answer true or false. But in those cases where the agent's knowledge is either contradictory or insufficient, an agent needs to report this. In particular, when an agent lacks sufficient information, it should ask another agent (s) in the system for help.

The basic flow of agent operation is shown in.

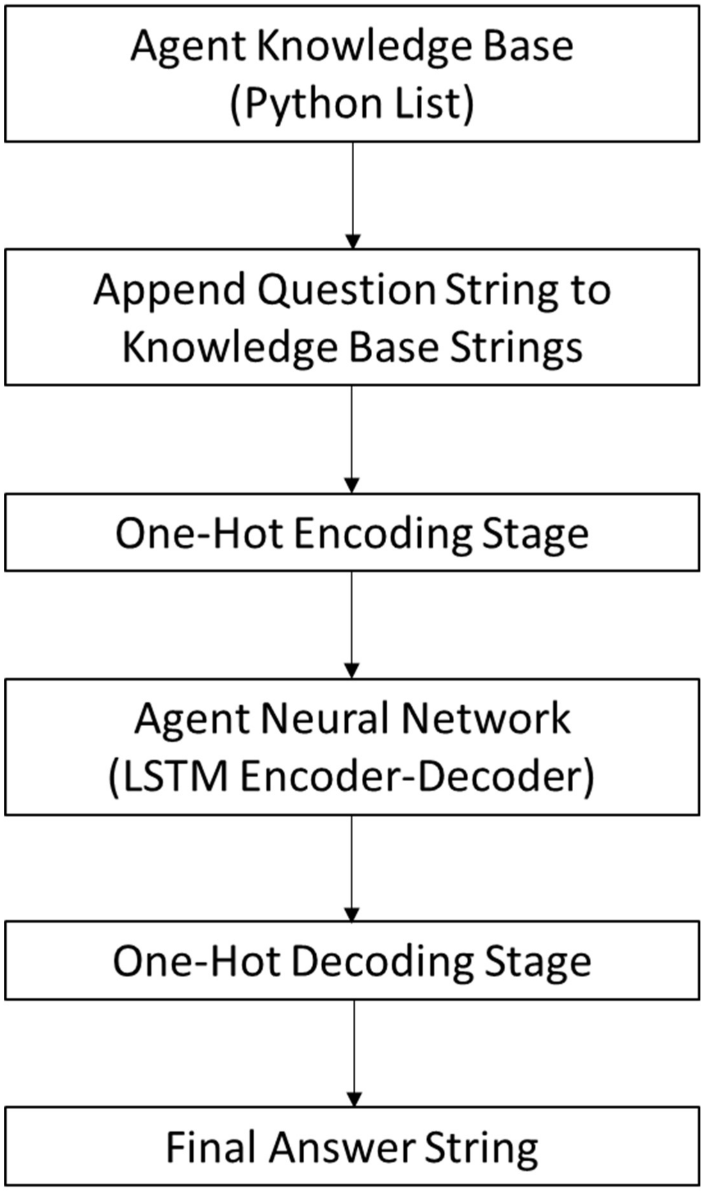


Figure 1: Agent Operation

An agent contains a knowledge base, which is a Python list of strings containing sentences of propositional logic (i.e. 'A', 'A ==> B'). When an agent is presented with a question, the strings from the knowledge base are concatenated with the question string to form the string input. The string input is encoded using a one-hot encoding scheme to create an array of inputs. The input array is presented to the LSTM neural network (described in the next section). The outputs are one-hot decoded, and the resulting string is returned. We will explain the stages in greater detail below.

One-Hot Encoding

Each character in a string, such as a space, a period, a variable name, or an operator or part thereof, is uniquely represented as a column vector with zeros in all positions except one. For example, a straightforward one-hot encoding scheme for the set of letters {a, b, c} could be described by:

$$a \rightarrow \begin{bmatrix}1\\0\\0\end{bmatrix} \quad b \rightarrow \begin{bmatrix}0\\1\\0\end{bmatrix} \quad c \rightarrow \begin{bmatrix}0\\0\\1\end{bmatrix}$$

With only three letters to encode, a one-hot vector set with three elements, precisely one of which is 1 with all the rest being zero, can be used to encode the letters 'a', 'b', and 'c'. For the encoding of logical sentences, however, the alphabet has 30 possible characters while the output, which can include extra characters, has 42 possible characters. Hence, each character in an input string is a 30 x 1 column vector with 29 zeros and with the location of a single 1 used to denote which character is being encoded. Likewise, each output string character is represented by a 42 x 1 vector. There are 41 zeros with a single 1 entry, the position of which indicates which character of the output alphabet is encoded. Thus, a string of 11 input characters would be represented by a 30 x 11 array, and a string of 17 output characters would be represented by a 42 x 17 output array. Let M denote the maximum allowed length of an input sequence. We encode the input as a 30 x M array. If the input was of length L where L <= M, then the remaining M-L characters will be blank spaces, represented as one of the 30 possible input characters. If L > M, then we truncate at M characters. Hence, all inputs are set to a uniform size 30 x M, and we denote the input array as **X.** The t^{th} column of the input array is denoted as **x(t).** The 30 x M array is used to create inputs for the LSTM encoder-decoder neural network and is presented to the LSTM encoder stage. At each time step *t,* where t ranges from 0 through (M-1) inclusive, we give **x(t)** to the recurrent LSTM sequence-to-sequence network.

The network will generate a set of output vectors **y(t)**, each of which is a 42 x 1 column. These are concatenated together to create final output array **Y**. Because neural network outputs from the softmax operation described in the following two sections are not exactly 1 or 0 (i.e., you may see 0.995 or 0.005 instead of 1 or 0), we use the index of the largest element of each output column vector **y(t)** to obtain the output character at step *t*. The process of generating **y(t)** given **x(t)** lies at the heart of the agent, and an LSTM sequence-to-sequence neural network performs this. The basics of this network are described in the following three sections.

Basic Neural Networks

A basic neural network accepts a vector **x** of inputs and returns a vector **y** of outputs. It normally consists of one or more "dense" layers (Chollet, keras.io, n.d.) each of which can be described by the following equation:

$$\mathbf{x}_{i+1} = f(\mathbf{A}_i\mathbf{x}_i + \mathbf{b}_i)$$

Here, subscript *i* denotes the layer, of which there must be at least one; $\mathbf{x}_i$ is the input vector to the layer; $\mathbf{A}_i$ is the weight matrix of the layer; and $\mathbf{b}_i$ is the bias vector. The original input vector **x** is denoted by $\mathbf{x}_0$. The function *f* is a non-linear function. Common choices for *f* include, but are not limited to:

1. $\mathrm{relu}(x) = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases}$
2. $\tanh(x)$

3. $\text{sigmoid}(x) = \frac{1}{1+exp(-x)}$
4. $\sigma_i(\text{x}) = \frac{exp(x_i)}{\sum_{j=1}^{K} \exp{(x_j)}}$

Equations (1) through (3) would be applied on an elementwise basis to get the output vector. Equation (4) defines the softmax function. If **x** is a K-dimensional vector then the softmax function is also a K-dimensional vector with i^{th} element defined as in equation (4). The softmax vector elements are strictly positive and always sum to exactly 1. In some cases, the identity function *f*(**x**)=**x** can also be used, particularly for the final dense layer of a multilayer network. If a network has N layers, then the final output will be **y** = $\mathbf{x}_N$, which is the output of the final layer. Sometimes the final layer uses a different function *f* than the previous layers.

Neural networks based on one or more dense layers have many applications, and we use a dense layer as part of our larger neural network in a way to be described later. However, such neural networks lack any memory. Their current output depends strictly on the current input. In the next two sections we will discuss a type of recurrent neural network, a network whose output depends on both current and previous inputs, called a long short-term memory (LSTM) network.

Long Short Term Memory (LSTM) Neural Networks

LSTM neural networks are among the most successful recurrent neural network architectures for processing series data. We give a brief description of them here, closely following (Olah, 2015) from which we borrow our explanatory figures in this section and briefly summarize here.

Traditional neural networks are designed to have a fixed input vector **x** and a fixed output vector **h** which depends entirely on the input **x**, and such networks define a function **h**=*f(***x***)*. These neural networks have no memory, and **h** does not depend on previous inputs, only on the current input **x**.

A recurrent neural network, by contrast, has an output that depends not only on the current input but also on all previous inputs. Let ($\mathbf{x_0,x_1,\ldots,x_t}$) denote a time series of inputs, and let ($\mathbf{h_0,h_1,\ldots,h_t}$) denote the corresponding time series of outputs. In a recurrent neural network, we have:

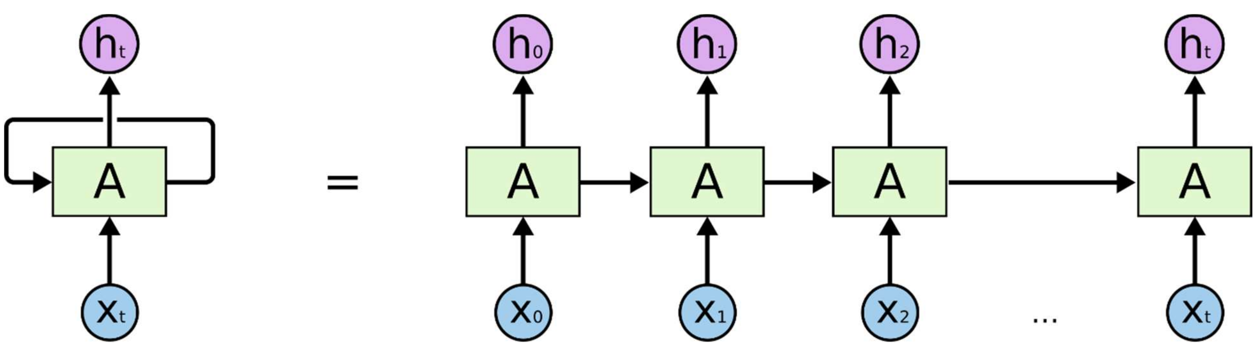


Figure 2: A recurrent neural network

Here, we observe that $\mathbf{h_t}$ is a function not only of $\mathbf{x_t}$ but of all previous inputs as well. Hence, recurrent neural networks have memory, unlike traditional neural networks.

One particularly successful type of recurrent neural network is the LSTM. The basic architecture of the LSTM is shown here.

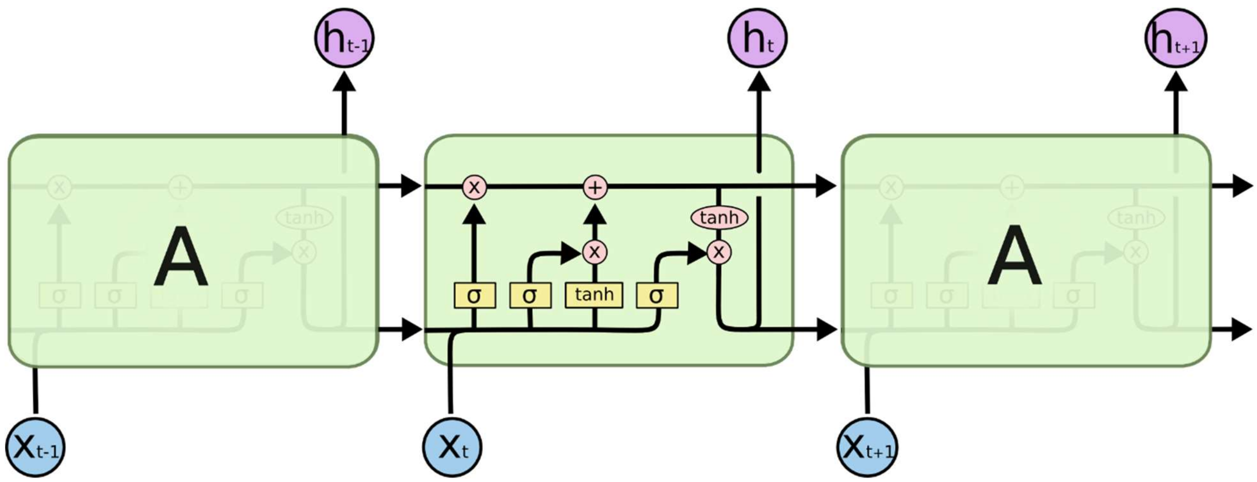


Figure 3: LSTM Illustration

We will now explain the internal operation of the LSTM cell in the middle of Figure 3: LSTM Illustration.

The cell state $\mathbf{C_t}$ is propagated in each computational step as shown in Figure 4: Propagation of Cell State.

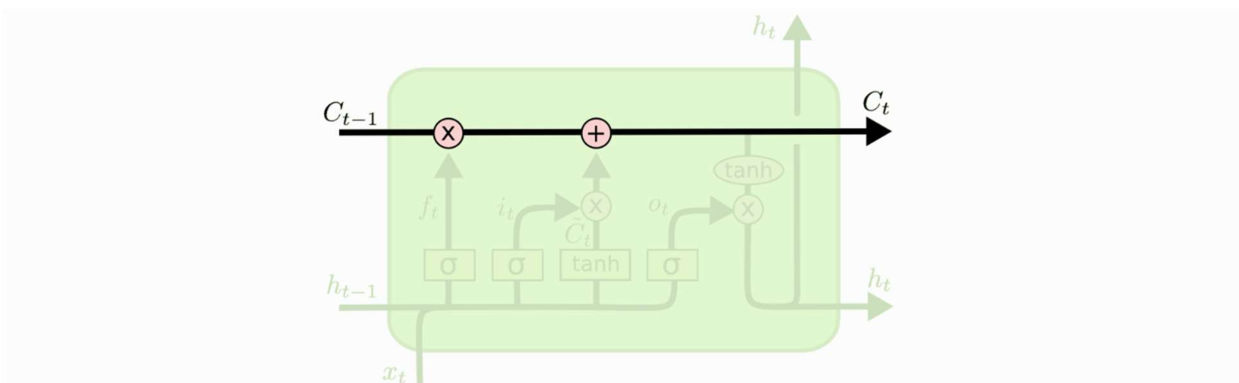
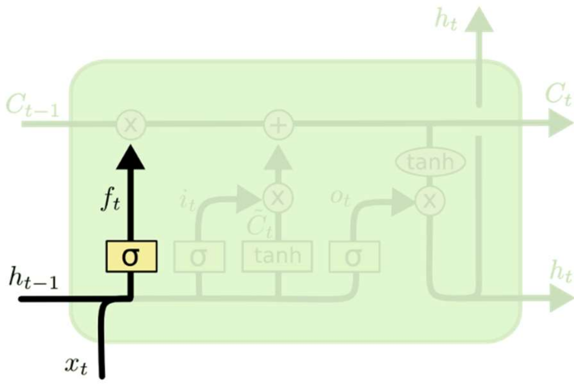


Figure 4: Propagation of Cell State

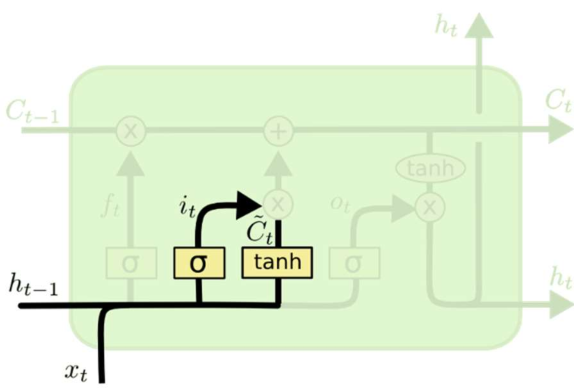
Here, the state must pass through two key stages: a forgetting stage followed by an updating stage. The forgetting stage multiplies elements of the previous cell state $\mathbf{C_{t-1}}$ by numbers between 0 (completely forget that vector element) and 1 (remember that vector element perfectly). The updating stage adds new information to the cell state. The computation of the forgetting stage is shown in Figure 5: Forgetting Stage Computation



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$$

Figure 5: Forgetting Stage Computation

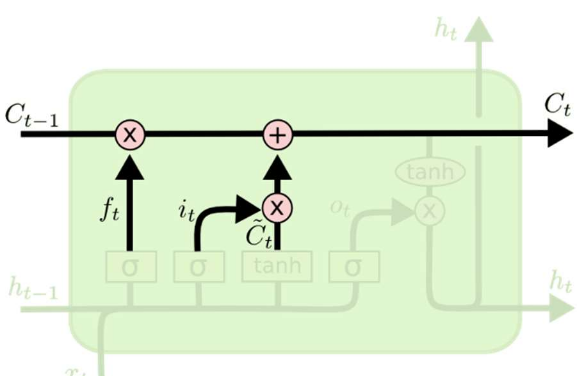
Once this stage is passed, we must update the cell state. The computation for the update is shown in Figure 6: Update Computation.



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

Figure 6: Update Computation

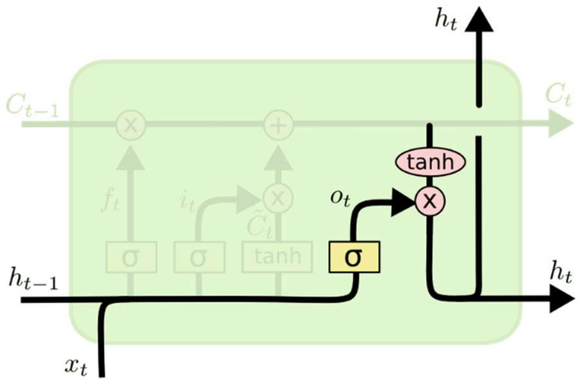
Combining the above, we see that the new cell state is computed as follows:



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Figure 7: Computing New Cell State

Once the new cell state has been computed, it is necessary to compute the new output $\mathbf{h_t}$.



$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Figure 8: Computing the Output of the LSTM Cell

Sequence-to-Sequence LSTMs

Sequence-to-Sequence LSTMs have been successfully used for many complex tasks, including language translation (Bengio, 2014). A description of Sequence-to-sequence LSTMs can be found in (Le, 2014), (Chollet, 2017), and (Bengio, 2014), and our implementation code is a modified version of code from (Chollet, 2017). The description of sequence-to-sequence LSTMs given here summarizes (Chollet, 2017).

As shown in Figure 9: Sequence to Sequence LSTM, two LSTM networks are involved. The LSTM encoder accepts an input sequence and generates a final internal state (**h**, **c**) as described in Section 0. The LSTM decoder takes (**h**, **c**) as inputs to its cells along with the [START] token and generates an output sequence of characters in the target language until the [STOP] token is reached. Hence, the meaning of the original English sentence is captured in (**h**, **c**) by the LSTM encoder, and this meaning is then converted into an output sequence in the French language by the LSTM decoder. The decoder LSTM cell is presented, initially, with inputs (**h**, **c**) capturing cell state and linguistic meaning from the encoder, and this is the initial state passed to the cell. The initial input is the [START] token. The decoder's output is fed back into the input, generating the French phrase one character at a time until the network generates a [STOP] token.

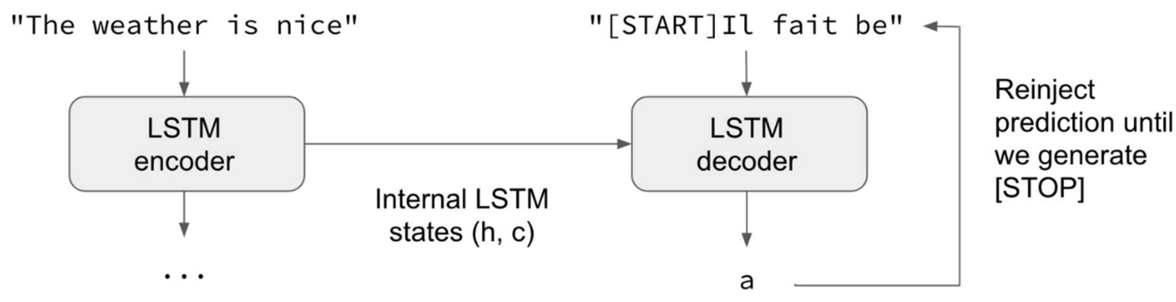


Figure 9: Sequence to Sequence LSTM

The process of training such a network is known as teacher-forcing. In teacher forcing, which is illustrated in Figure 10: Training a Sequence-to-Sequence LSTM, the input sequence begins with the [START] token and continues one character at a time with the desired French phrase. The desired output sequence is the selected target phrase. **Add links for teacher forcing.**

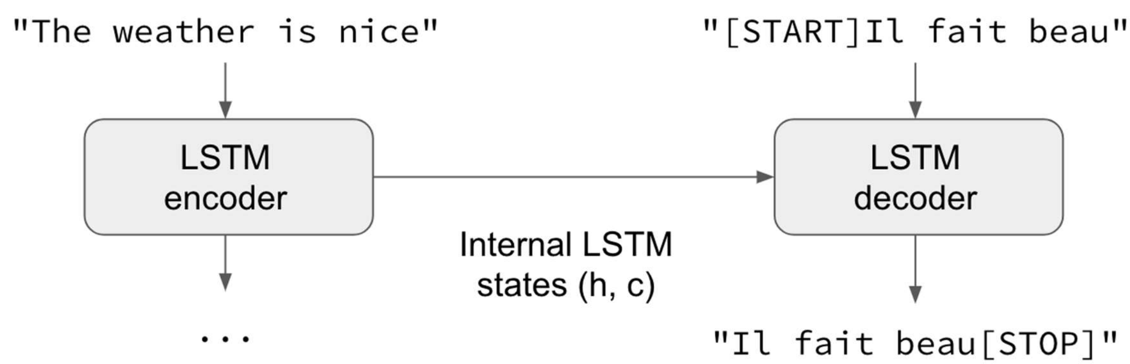


Figure 10: Training a Sequence-to-Sequence LSTM

The teacher forcing method has been successfully used to train language translation LSTM encoder-decoder networks, and this is the training method used in our work.

Bidirectional LSTMs and Sequence-to-Sequence Models

The sequence-to-sequence LSTM described above is unidirectional, with information flowing from left-to-right in Figure 3: LSTM Illustration. In a bidirectional LSTM, there are two LSTM layers: one layer processes the sequence left-to-right, and a second parallel layer processes from right-to-left in the opposite direction. This is illustrated in Figure 11: Bidirectional RNN below, and we note that bidirectionality can be applied to any type of RNN, not just LSTMs. We observe one LSTM processes information left-to-right as described earlier, but the second parallel layer processes in the opposite direction.

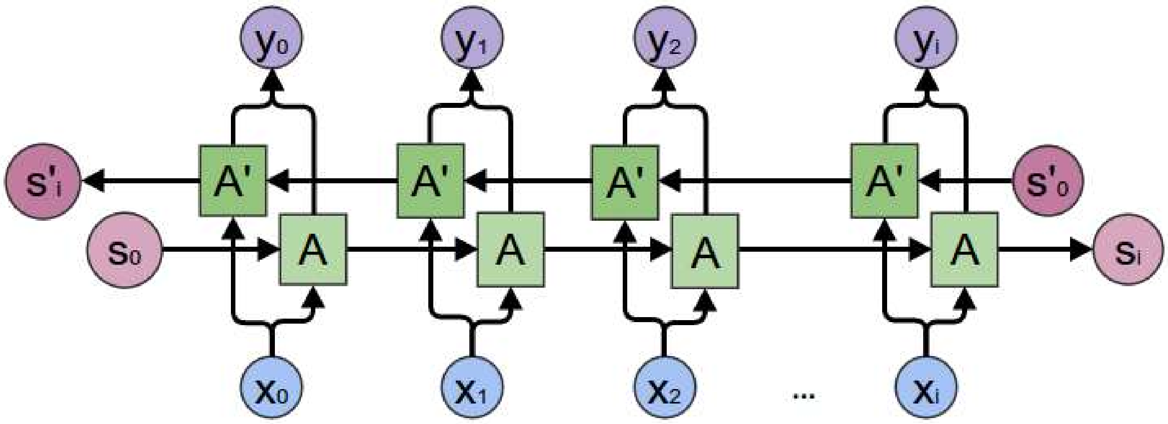


Figure 11: Bidirectional RNN

A bidirectional network allows us to generate output pairs (x_i,y_i) where the x's are the outputs of the left-to-right network and the y's are the outputs of the reverse right-to-left network. The concatenation z_i=(x_i,y_i) is a vector whose contents depend on *both* the previous *and* the future characters in a sequence. In cases where the full sequence is available, the resulting concatenated vector encodes information about the complete sequence, not just information from the previous left portion of the sequence. Hence, the z's that constitute the output of the bidirectional LSTM yield a more comprehensive picture of the entire input sequence than the original x's would have alone.

In the sequence-to-sequence model defined previously, we didn't use the sequence of outputs as input to the decoder segment of the encoder-decoder architecture. Rather, we simply used the final encoder state (**h,c**) as the initial state of the decoder network, and we used a start token as the input to go with the initial state. In order to implement a sequence-to-sequence model that takes advantage of bidirectionality, we do the following:

1. Make the encoder stage a bidirectional network. This results in two final states, which are ($\mathbf{h_f}$,$\mathbf{c_f}$) for the forward network and ($\mathbf{h_r}$,$\mathbf{c_r}$) for the reverse network.
2. Concatenate the states. Here, **h** = ($\mathbf{h_f}$,$\mathbf{h_r}$) and **c** = ($\mathbf{c_f}$,$\mathbf{c_r}$).
3. Present this concatenated state (**h,c**) to the decoder LSTM as its initial state, exactly as would be done in the previous sequence-to-sequence model.
4. Run the decoder network in a forward-only direction, just as in the previously presented sequence-to-sequence model.
5. Note that the decoder network state vector dimensionality is now twice that of each of the two encoder networks. Previously, the encoder and decoder networks would have had the same state vector dimensionality.

The neural network used here uses 256-dimensional vectors for $\mathbf{h_f},\mathbf{c_f},\mathbf{h_r},\mathbf{c_r}$. Hence, we have two parallel 256-dimensional LSTMs running in the forward and reverse directions in the encoder stage. This means that the final encoder output state (**h**,**c**) has 512-dimensional **h** and **c**. Since this is the initial state for the unidirectional decoder LSTM, the decoder LSTM is based on 512-dimensional vectors.

In order to reduce from a 512-dimensional vector to a one-hot encoded vector, the well-known dense neural network layer described previously, with a final softmax non-linearity, is used. The index of the maximum element of the output vector tells us which character is to be decoded.

Data Sets for Training and Testing

Data set files are tab separated files with the filename extension "tsv" and use tabs to divide their data into four columns:

1. Column 0: Original problem and question. This consists of a Boolean sentence, a period, and a question about a Boolean variable. For example, "(~J & B) & F & (B & F) & (B & F) & (B & F) . What is B ?" consists of the Boolean sentence "(~J & B) & F & (B & F) & (B & F) & (B & F)" and the question "What is B ?". Here, Boolean clauses or variables can be randomly repeated one or more times, and in this example "(B & F)" is repeated three times.
2. Column 1: Simplified problem and question. Here, all repetitions of Boolean clauses, including one variable clauses, are removed. The sentence is still separated from the question by a period. For example, "(~J & B) & F & (B & F) . What is B ?" is the entry corresponding to the sentence above, but with the repetitions removed. Hence, the Boolean sentence that comes before the question has the exact same meaning, but with the repetitions removed it is a concise version of the original sentence. The question remains unchanged from the first column.
3. Column 2: Simplified Boolean sentence. In this example, it would be "(~J & B) & F & (B & F)", which is the simplified sentence from column 1 but without the question.
4. Column 3: The desired answer. The four possible desired answers are:
    a. "TRUE"
    b. "FALSE"
    c. "Contradictory"
    d. "Unknown HELP!"
    e. In this example, the desired answer is "TRUE" because the variable B is TRUE given the sentence "(~J & B) & F & (B & F)".

The example above would appear in a spreadsheet like this:

(~J & B) & F & (B & F) & (B & F) & (B & F) . What is B ?	(~J & B) & F & (B & F) . What is B ?	(~J & B) & F & (B & F)	TRUE

All of the training data exist in this four-column tab separated format. These data need to be one-hot encoded in order to create inputs and targets for the neural network. For each line in the TSV training set, two string pairs are created.

1. Pair 1: This consists of the original string in column 0 and the target string in column 3.
    a. In the example above, the pair would be ("(~J & B) & F & (B & F) & (B & F) & (B & F) . What is B ?", "TRUE")

2. Pair 2: This consists of the Boolean sentence in column 0, with the question replaced by "HELP", followed by the simplified sentence of column 2.
    a. In the example above, the pair would be ("("(~J & B) & F & (B & F) & (B & F) & (B & F) . HELP","(~J & B) & F & (B & F)")

The purpose behind the two training pairs is as follows:

1. Purpose of Pair 1: Teach the neural network how to answer a question about a Boolean variable when given a Boolean sentence.
    a. If the sentence is contradictory, then no answer is possible, and the network must indicate "Contradictory".
    b. If there is insufficient information, the network should respond with "Unknown HELP!".
    c. If the sentence implies the variable is true, then the network should respond with "TRUE".
    d. If the sentence implies the variable is false, then the network should respond with "FALSE".
2. Purpose of Pair 2: Teach the neural network to consolidate a knowledge base, which may contain repetitions, and to dump a concise version of the knowledge base when given a requestion for help.
    a. Here, the question is replaced with the word "HELP".
    b. Upon seeing the keyword "HELP" instead of a question, the network should create a concise version of the knowledge base without any repetition and dump that concise knowledge base out.
    c. The network must dump out the concise version faithfully without error and without repetition.

Hence, the goal of training is to enable a neural network to perform two basic functions:

1. Perform Boolean reasoning and answer a question about a Boolean variable given a possibly repetitive or contradictory or incomplete knowledge base.
2. Perform simplification and return a concise version of a knowledge base given a requestion for help.

The two functions described above are central to the operation of agents described in Figure 1: Agent Operation. The agent's internal Python list stores Boolean sentences and is the knowledge base of the agent. If an agent is asked a question about the value of a Boolean variable, then the following operations take place:

1. The sentences of the knowledge base are concatenated using logical-AND, which is the ampersand symbol "&".
2. The resulting large Boolean sentence is concatenated with the question using a period in order to create the input string.
3. The input string is one-hot encoded and fed to the neural network.
4. The neural network output is one-hot decoded to create an output string, which should be one of the four possible responses to a question about a Boolean variable.

Hence, "pair 1" as described above, is used to teach the neural network how to perform the Boolean reasoning operation.

On the other hand, if an agent is asked for help instead of being asked a question about a Boolean variable, then the workflow is as follows:

1. The sentences of the knowledge base are concatenated using logical-AND (the "&" symbol) as above.
2. But the large Boolean sentence is now concatenated with the word "HELP", which is separated by a period from the large sentence from step 1.
3. The input string is one-hot encoded and sent to the network.
4. The network output is one-hot decoded and should, ideally, contain a concise and correct Boolean sentence describing the agent's knowledge base, without any repetitions.

Hence, "pair 2", as described above, is used to teach the neural network how to respond for a request for help by providing a correct and concise dump of the agent's knowledge base.

The Accuracy Metric

Agent performance is measured by simple string comparison. The output of the agent's neural network is one-hot decoded to create a string, and the string is compared to the target string. For the "pair 1" case where the agent must answer a question about a Boolean variable, the output string possibilities are "TRUE", "FALSE", "Contradictory", and "Unknown HELP!". An exact string match is required for the agent's response to be correct. For example, if the right answer is "FALSE", then the agent's response is scored as correct only if its output string exactly matches "FALSE". Responses such as "F", "false", "False", or "fALse" would all be scored as incorrect. In like manner, for the "pair 2" case where the neural network has to perform a knowledge base dump in response to a help request, the output must exactly match the target string. The target string has all of the Boolean clauses of the input string, in exactly the same order, but without any repetition. If the correct answer is "(~J & B) & F & (B & F)", then answers such as "(~J & B) & (B & F) & F", while clearly logically equivalent, will still be marked as wrong. Hence, in order to achieve a score of 98%, for example, an agent's outputs must be an exact string match to the target 98% of the time. Even logically equivalent outputs that don't provide this exact match will be scored as wrong. This is a different metric than the character-by-character accuracy metric in that a string with a single character error is marked as wrong, even when most of the characters but one are correct.

The Data Sets

Each tab-separated file contains 25,000 lines, each of which is used to create both a logic problem, or "pair 1", and a problem of restating knowledge concisely, or "pair 2". Hence, each file will yield a total of 50,000 training problems split evenly between Boolean reasoning and concise knowledge recitation.

The first 100 files, with filenames “logic_data_extended_00.tsv” through “logic_data_extended_99.tsv”, are the training files. The remaining files, which are “logic_data_extended_100.tsv” through “logic_data_extended_233.tsv”, are for accuracy testing.

1. Future neural networks may train on nearly all of these files. The neural network as of repository commit fe2b92d6c7c25b2752650b0358a3df885b794558 from 29 December 2021 was trained on the first 100 files as stated above. It remains to be seen if accuracy can be improved by training on a larger number of files.
2. The author intends to generate a total of 500 data sets and to use the majority of these to train a new version of the network to determine whether we can achieve improved accuracy.
3. The author has obtained accuracy exceeding 98% with the existing network using data set logic_data_extended_200.tsv, a data set not used in training.

It is now important to point out some of the limitations of the data sets in the present work. Problems are randomly generated, but a large percentage of problems, unfortunately, tend to allow a network to deduce a value as true or false simply based on a single Boolean term. This is certainly a weakness of the present randomized clause generator, and it will result in some weaknesses when the network is presented with highly complex Boolean sentences. The goal of this work is to focus on the concept of self-awareness, not to create a general-purpose Boolean reasoning system. Even so, the biases in data set generation are a deficiency that must be addressed in future work. However, the present agents are able to demonstrate the basics of self-awareness that are the goal of this work, but readers are cautioned that particularly for moderately to highly complex sentences, the agents may make errors in Boolean reasoning. This requires improvements to the data set generation procedures. Data set generation is a random process described as follows.

1. Boolean variables are randomly selected to populate a sentence.
    a. Some of these are randomly negated to insure we have both positive and negative atomic Boolean terms.
2. Binary logic operators, such as logical AND, OR, and implication, are randomly selected in order to create binary clauses.
3. Randomly generated terms and clauses are joined using the logical AND “&” operator to create sentences.
4. A variable is randomly chosen for the question, and each question is always about the value of a single variable, which may be negated.
5. The resulting sentences are processed using automatic reasoning code, taken from (Norvig, aima-python, n.d.), in order to determine whether the variable is True or False, or whether there is insufficient information or whether the input knowledge is contradictory.
6. The clauses of the sentence can be repeated one or more times in order to generate the “long” version of the sentences that may contain repeated clauses.
7. The “long” sentence, with repetition, is concatenated with the question to create the column 0 entry.
8. The “short” sentence, without repetition, is concatenated with the question to create the column 1 entry.
9. The “short” sentence, without repetition, becomes the column 2 entry.

10. The answer to the question (i.e. “TRUE”, “FALSE”, etc.) becomes the column 3 entry.

The choice to have repetition in data set design requires some explanation.

1. In earlier attempts, Boolean reasoning performance without the repeated clauses proved to be poor. For example, if an agent saw a clause repeated twice, it could make errors in reasoning!
    a. In multi-agent scenarios, if a given sentence was known to two agents, the repetition of the sentence would sometimes result in reasoning errors! This was the original motivation for our repetitive clause training.
2. Forcing agents to learn how to create a concise sentence with each clause repeated only once is believed to teach their internal neural network to treat clauses as conceptual units, resulting in improvements to reasoning performance.
    a. This is a hypothesis, however, that requires further testing, and the author does not claim sufficient evidence to claim that this type of training actually does teach a neural network to treat terms and clauses as conceptual units.

All TSV data sets are available at (Mukai, S3 Source Code and Data Sets, n.d.) in order to facilitate peer review and to make both the strengths and the weaknesses of the training and test data easily accessible and understood.

Key Software Used and Neural Network Specifications

The source code is publicly available at (Mukai, LSTM Source Code, n.d.) and is based on source code from (Chollet, keras.io, n.d.) and from (Norvig, aima-python, n.d.), with the LSTM neural network code taken from the former and with the Boolean logic code taken from the later. The training and testing process works as follows.

1. The program train_seq2seq_help.py is used to:
    a. Create the neural network itself.
    b. Perform a training epoch on logic_data_extended_00.tsv.
    c. Save the resulting neural network.
2. The neural network created is a bi-directional LSTM with
3. The program retrain_seq2seq_help.py is used to:
    a. Load the neural network created in step 1 above.
    b. Run for a specified number of epochs, presently 512.
    c. On each epoch:
        i. Randomly select a training set from logic_data_extended_00.tsv through logic_data_extended_99.tsv.
        ii. Perform a training epoch using the randomly selected data set.
4. The program run_seq2seq_help.py is used to:
    a. Load the neural network.
    b. Load the file logic_data_extended_200.tsv for use as a test set (this was not used in training above).
    c. Compute neural network accuracy over data set logic_data_extended_200.tsv.

5. The program run_seq2seq_demo.py is not a standalone program.
    a. It is simply imported in our web demo at (Mukai, Dual Agent Demo, 2022) in order to allow a demonstration of a two-agent system with basic self-awareness.

Web Demo

The goal of the Google Colab web demo is to provide a simple demonstration of agent self-awareness which, again, is defined as an agent's ability to be aware of its own knowledge state, in this case being aware of not knowing the right answer, and acting according, in this case by requesting aid from another agent. In the web demo:

Agent 1 begins with two facts:

~A

C ==> B

Agent 2 begins with just one fact:

B ==> A

Agent 1 is presented with this question:

What is C ?

However, agent 1 does not have enough information to know the value of C. At this stage, agent 1 asks for help, causing agent 2 to dump its knowledge base. Armed now with all three facts:

~A

C ==> B

B ==> A

Agent 1 can reason that since A is false and B ==> A, then B must be false. Since B is false, then C ==> B implies that C is also false. The demo ends with agent 1 indicating that C is false.

Agent 1 demonstrates a very simple form of self-awareness in terms of being aware of its own lack of knowledge. When agent 1 realizes it lacks the knowledge to answer the question regarding Boolean variable C, it will ask for help. Likewise, agent 2 also demonstrates a simple form of self-awareness in that it knows what its knowledge base contains and will dump those contents in response to a request for help. Please note that self-awareness is a property of the *agent*, and the *agent* is a composite of a neural network plus a Python list knowledge base and appropriate code that performs one-hot encoding and decoding and that can generate or receive a request for help. Hence, the neural network, while certainly the most important part of the agent, is not the entire agent. *Self-awareness is a property of the complete agent, not of the neural network as a standalone entity.*

Summary and Conclusions

This work presents agents that exhibit a simple form of self-awareness defined by:

1. Knowledge of one's own knowledge base.
    a. Exemplified by the ability to provide a concise version of the knowledge base on request for help from another agent.
2. Knowledge of one's own knowledge state.
    a. The ability to know whether one's knowledge is contradictory.
    b. The ability to know whether one's knowledge is insufficient.
    c. The ability to answer a question when one has sufficient and non-contradictory knowledge.

There are multiple ways that self-awareness could be defined, and the definition here is very simple. It falls significantly short of the much broader reasoning-about-knowledge capability characteristic of epistemic modal logic, which is a possible future direction. The fact that neural networks can be used to process complex mathematical formulas (Evans, Saxton, Amos, Pushmeet, & Grefenstette, 2018) (Lample & Charton, 2019) including those requiring logical entailment may suggest that this is a reasonable future direction. Agents using epistemic modal logic have a significantly deeper ability to reason about their own knowledge and about the knowledge of other agents, which may make this a promising direction.

A possible variation of this work may use a neural network to guide a formal reasoning engine. Since Boolean reasoning already entails substantial computational complexity, and since well-known neural networks have been able to excel in handling problems with exponential complexity, most notably the game of Go (DeepMind, n.d.) a promising direction may involve using neural networks to guide a formal reasoning engine. This will help to prevent outright errors in reasoning, a weakness of the present work, while simultaneously helping to overcome the exponential complexity of reasoning, which can be a significant problem with epistemic modal logic.

Hence, this work is a starting point. The form of self-awareness presented here is extremely basic, and the work presented is a proof-of-concept to demonstrate the potential of neural-network based systems to exhibit a basic form of self-awareness.

References

Bengio, K. C. (2014, September 3). *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation.* Retrieved from arXiv: https://arxiv.org/abs/1406.1078

Chollet, F. (2017, September 29). *A ten-minute introduction to sequence-to-sequence learning in Keras.* Retrieved from The Keras Blog: https://blog.keras.io/a-ten-minute-introduction-to-sequence-to-sequence-learning-in-keras.html

Du, B., Guo, Q., Zhao, Y., Zhi, T., Chen, Y., & Xu, Z. (2020). Self-Aware Neural Network Systems: A Survey and New Perspective. *Proceedings of the IEEE*, 1047--1067.

Evans, R., Saxton, D., Amos, D., Pushmeet, K., & Grefenstette, E. (2018). *Can Neural Networks Understand Logical Entailment?* Retrieved from arXiv: http://arxiv.org/abs/1802.08535

Lample, G., & Charton, F. (2019). *Deep Learning for Symbolic Mathematics.* Retrieved from arXiv: https://arxiv.org/pdf/1912.01412.pdf

Le, I. S. (2014, June 3). *Sequence to Sequence Learning with Neural Networks.* Retrieved from arXiv: https://arxiv.org/abs/1406.1078

Mukai, R. (2022, February 11). *Dual Agent Demo*. Retrieved from Dual Agent Demo: https://colab.research.google.com/drive/1Tr2AQKTGtG3VnNgHhRFLt2BXiXqSb3No?usp=sharing

Norvig, S. J. (2021). *Artificial Intelligence: A Modern Approach, 4th Edition.* Hoboken, NJ: Pearson Education, Inc.

Norvig, S. J. (n.d.). *aima-python*. Retrieved from aima-python: https://github.com/aimacode/aima-python

Olah, C. (2015, August 27). *Understanding LSTM Networks*. Retrieved from colah's blog: https://colah.github.io/posts/2015-08-Understanding-LSTMs/