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 this feature to solve a problem cooperatively. Because the term "self-awareness" can be defined in different ways, we provide our definition below:

1. An agent must know whether it possesses 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 report this instead of trying to provide an answer because 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 that the agent has explicit knowledge of its knowledge base contents, which is another form of self-knowledge.

For the purposes of this work, agents fulfilling these three criteria are considered "self-aware". Self-awareness allows agents with contradictory knowledge states to warn the user and avoid providing a wrong answer. It also enables 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

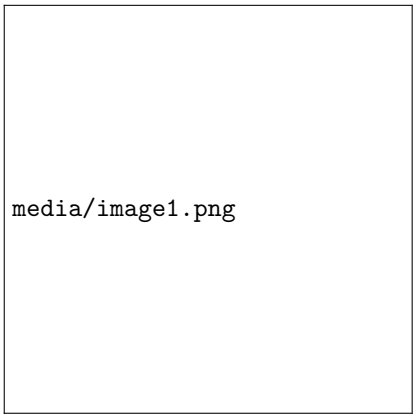
knowledge may be sufficient to allow the first agent to solve the problem. Here, agent self-awareness permits cooperation because the first agent knows that it must request help (by being aware of the inadequacy of its knowledge state). The second agent has direct, symbolic knowledge of the contents 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 connected to such problems, was discussed in (Evans, Saxton, Amos, Pushmeet, & Grefenstette, 2018). Moreover, the related issue of using neural networks for symbolic mathematics was covered in (Lample & Charton, 2019). The present work differs from these two reports because it emphasizes self-awareness and cooperation in neural-network-based agents, a topic not covered in the cited works. Self-aware neural network systems were surveyed in (Du, et al., 2020), where the type of self-awareness focused on the implementation and operation of the neural network itself, with the network having awareness of its own execution environment and the ability to optimize its dataflow, resource use, and execution within that environment for performance optimization. This is a very different type of self-awareness than what is described here, because 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 that our agents are trained to solve and then explain the behavior of the logical agents that 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 basic aspects 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 that used in (Norvig, Artificial Intelligence: A Modern Approach, 4th Edition, 2021), with our notation, along with many direct quotations, from (Norvig, aima-python, n.d.). Specifically, our notation is summarized in Table 1.



The notation given in the "Python Output" column of Table 1 will be used throughout this paper, as is done in our demonstration system. The propositional logic variables are denoted by 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 then asked about the truth value of a propositional variable. For example, a simple case of modus ponens would be as follows:

Table 1: Table : Logical Notation

Input sentences	A A ==> B
Question	What is B ?
Answer	B is true.

Agents are based on an LSTM sequence-to-sequence neural network as 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 its knowledge base.

For example, a contradiction would be as follows:

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

In this example, the sentence "A & B" implies that A is true, whereas "~A" means that A is false. Owing to this contradiction, *any* conclusion may be drawn and the knowledge base itself is invalid. In such a case, the agent needs to report a contradictory state of knowledge from which no conclusion can 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 sentences of the knowledge base, an agent should be able to answer true or false. However, in those cases where the agent's knowledge is either contradictory or insufficient, the agent needs to report this. In particular, when an agent lacks sufficient information, it should ask another agent or agents in the system for help.

The basic flow of agent operation is depicted in Figure 1.

Figure : Agent Operation

An agent contains a knowledge base, which is a Python list of strings containing sentences of propositional logic (e.g., “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, which is encoded using a one-hot encoding scheme to create an array of inputs. This input array is then presented to the LSTM neural network (described in the next section). The outputs are one-hot decoded, and the resulting string is returned. These stages are explained 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 as follows:

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

With only three letters to encode, a one-hot vector set with three elements,

precisely one of which is one 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 30×1 column vector with 29 zeros and a single one whose location denotes the character being encoded. Similarly, each output string character is represented by a 42×1 vector containing 41 zeros and a single one, the position of which indicates the character of the output alphabet being encoded. Thus, a string of 11 input characters would be represented by a 30×11 array, and a string of 17 output characters would be represented by a 42×17 output array. Let M denote the maximum allowed length of an input sequence, such that we encode the input as a $30 \times M$ array. If the input is of length L where $L \quad M$, then the remaining $M \quad L$ 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 of $30 \times M$, and we denote the input array as $\mathbf{X}$, where the t^{th} column of the input array is denoted $\mathbf{x}(t)$. The $30 \times M$ array is used to create inputs for the LSTM encoder–decoder neural network and presented to the LSTM encoder stage. At each time step t, where t ranges from zero through $(M \quad 1)$ inclusive, we give $\mathbf{x}(t)$ to the recurrent LSTM sequence-to-sequence network.

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

Basic Neural Networks

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

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

where the 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 $\mathbf{x}$ is denoted $\mathbf{x}_0$. The function f is a non-linear function, common choices for which include but are not limited to the following:

1. $\text{relu}(x) = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases}$
2. $tanh(x)$
3. $\text{sigmoid}(x) = \frac{1}{1+\exp(-x)}$
4. $\sigma_i\,(\text{x}) = \frac{\exp(x_i)}{\sum_{j=1}^{K} exp(x_j)}$

The first three equations are applied on an elementwise basis to obtain the output vector. The fourth equation defines the softmax function. If $\mathbf{x}$ is a K-dimensional vector then the softmax function is also a K-dimensional vector with the i^{th} element defined as in the fourth equation. The softmax vector elements are strictly positive and always sum to exactly one. In some cases, the identity function $f(\mathbf{x})$=$\mathbf{x}$ 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 $\mathbf{y} = \mathbf{x}_N$, which is the output of the final layer. A different function f is sometimes used for the final layer compared with the previous layers.

Neural networks based on one or more dense layers have numerous applications, and we use a dense layer as part of our larger neural network in a manner that will be described later. However, such neural networks lack any memory; their current output is strictly dependent on the current input. In the following two sections, we will discuss a type of recurrent neural network whose output depends on both current and previous inputs, i.e., an LSTM network.

LSTM Neural Networks

LSTM neural networks are among the most successful recurrent neural network architectures for processing series data. Here we provide a brief description of these networks, closely following the work of Christopher Olah (Olah, 2015), from which we borrow our explanatory figures in this section.

Traditional neural networks are designed to have a fixed input vector $\mathbf{x}$ and a fixed output vector $\mathbf{h}$ that depends entirely on the input $\mathbf{x}$, and such networks are defined by a function $\mathbf{h} = f(\mathbf{x})$. These neural networks have no memory, and $\mathbf{h}$ does not depend on previous inputs, only the current input $\mathbf{x}$.

A recurrent neural network, in contrast, has an output that depends on not only the current input but also all previous inputs. Let ($\mathbf{x}_0$, $\mathbf{x}_1$, …, $\mathbf{x}_t$) denote a time series of inputs and ($\mathbf{h}_0$, $\mathbf{h}_1$, …, $\mathbf{h}_t$) denote the corresponding time series of outputs. The resulting recurrent neural network is depicted in Figure 2.

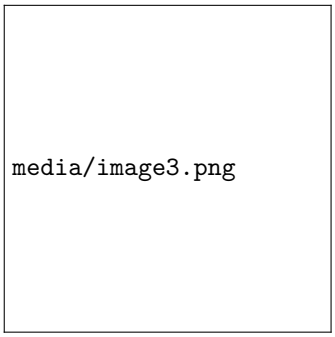


Figure : A Recurrent Neural Network

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 possess memory, unlike traditional neural networks.

One particularly successful type of recurrent neural network is the LSTM network, the basic architecture of which is shown in Figure 3.

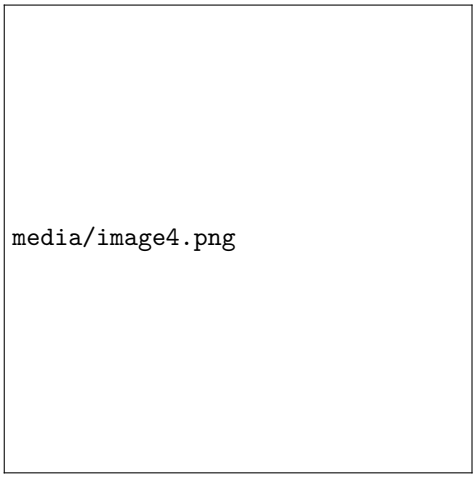


Figure : An LSTM Network

We will now explain the internal operation of the LSTM cell in the center of Figure 3. In each computational step, the cell state $\mathbf{C}_t$ is propagated as depicted in Figure 4.

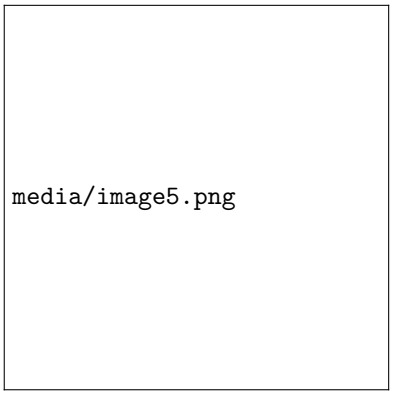


Figure : Propagation of Cell State

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 $\mathbf{C}_{t1}$ by numbers between zero (completely forget that vector element) and one (remember that vector element perfectly). The subsequent updating stage adds new information to the cell state. The computation of the forgetting stage is shown in Figure 5.

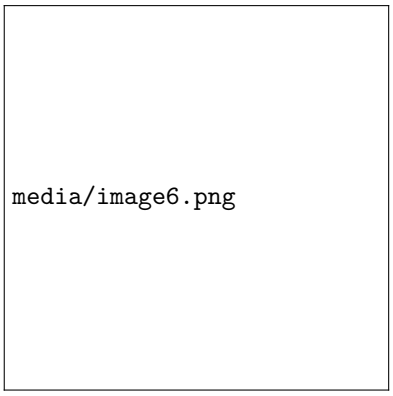


Figure : Forgetting Stage Computation

Once this stage is passed, we must update the cell state. The computation of the updating stage is presented in Figure 6.

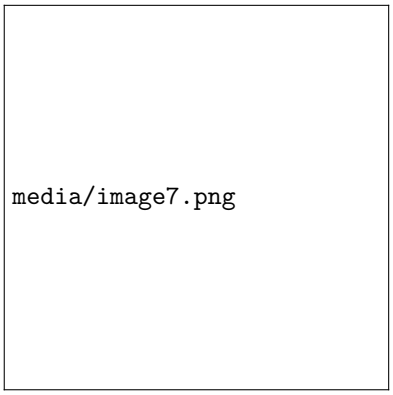


Figure : Updating Stage Computation

Combining the above, we see that the new cell state is computed as depicted in Figure 7.

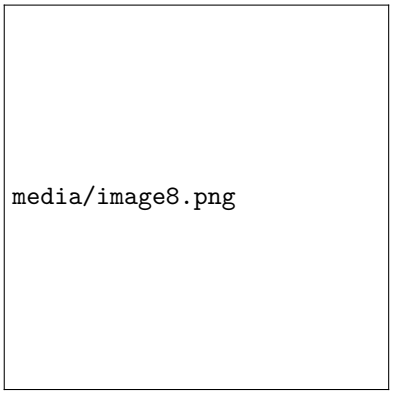


Figure : Computing New Cell State

Once the new cell state has been calculated, it is necessary to compute the new output $\mathbf{h}_t$, as shown in Figure 8.

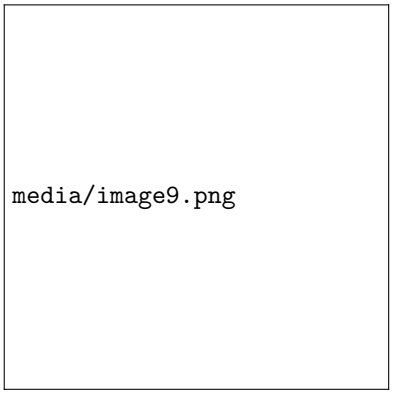


Figure : Computing the Output of the LSTM Cell

Sequence-to-Sequence LSTMs

Sequence-to-sequence LSTMs have been successfully applied to numerous 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 that from (Chollet, 2017). The description of sequence-to-sequence LSTMs given here summarizes (Chollet, 2017).

As exemplified in Figure 9, sequence-to-sequence LSTMs involve two LSTM networks. 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 the input 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 then converted into an output sequence in French (for example) by the LSTM decoder. The decoder LSTM cell is presented, initially, with the input (**h**, **c**) capturing the 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 output from the decoder is fed back into the input, generating the French phrase one character at a time until the network generates a [STOP] token.

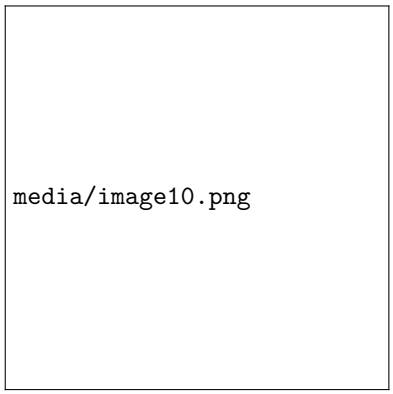


Figure : Sequence-to-Sequence LSTM

The process of training such a network is known as teacher forcing. In teacher forcing, which is illustrated in Figure 10, 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.**

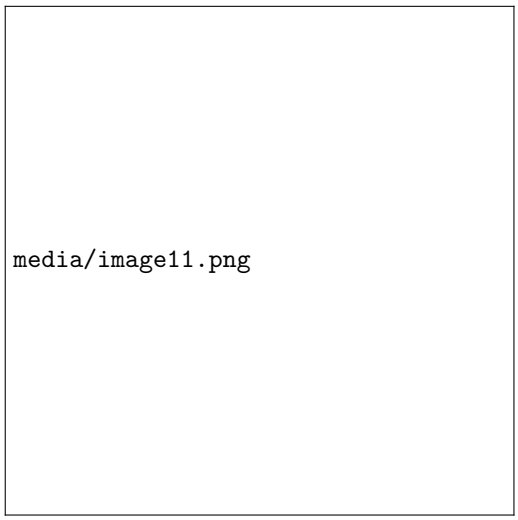


Figure : Training a Sequence-to-Sequence LSTM

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. In a bidirectional LSTM, there are two LSTM layers: one layer processes the sequence from left to right, while a second parallel layer processes the sequence in the opposite direction, as illustrated in Figure 11. We note that bidirectionality can be applied to any type of recurrent neural network, not just LSTMs.

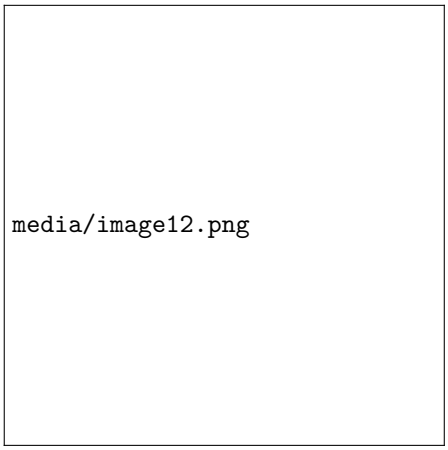


Figure : A Bidirectional Recurrent Neural Network

A bidirectional network allows us to generate output pairs $(\mathbf{x}_i, \mathbf{y}_i)$, where $\mathbf{x}_i$ denotes the outputs of the left-to-right network and $\mathbf{y}_i$ denotes the outputs of the reverse (right-to-left) network. The concatenation $\mathbf{z}_i = (\mathbf{x}_i, \mathbf{y}_i)$ is a vector whose contents depend on both the previous and future characters in the sequence. In cases where the full sequence is available, the resulting concatenated vector encodes information about the complete sequence, not just the preceding (left) portion of the sequence. Hence, the $\mathbf{z}_i$ vectors outputted by the bidirectional LSTM yield a more comprehensive picture of the entire input sequence than that afforded by the original $\mathbf{x}_i$ output alone.

In the sequence-to-sequence model defined previously, we did not use the sequence of outputs as an input to the decoder segment of the encoder–decoder architecture. Rather, we simply adopted the final encoder state $(\mathbf{h}, \mathbf{c})$ as the initial state of the decoder network, and we used a start token as the input to go with the initial state. Thus, to implement a sequence-to-sequence model that takes advantage of bidirectionality, we perform the following steps:

1. Make the encoder stage a bidirectional network. This results in two final states, which are $(\mathbf{h}_\mathrm{f}, \mathbf{c}_\mathrm{f})$ for the forward network and $(\mathbf{h}_\mathrm{r}, \mathbf{c}_\mathrm{r})$ for the reverse network.
2. Concatenate the states. Here, $\mathbf{h} = (\mathbf{h}_\mathrm{f}, \mathbf{h}_\mathrm{r})$ and $\mathbf{c} = (\mathbf{c}_\mathrm{f}, \mathbf{c}_\mathrm{r})$.
3. Present this concatenated state $(\mathbf{h}, \mathbf{c})$ to the decoder LSTM as its initial state, in exactly the same manner as the previous sequence-to-sequence model.
4. Run the decoder network in the forward-only direction, just as in the previous 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 adopted here uses 256-dimensional vectors for $\mathbf{h}_\mathrm{f}$,$\mathbf{c}_\mathrm{f}$,$\mathbf{h}_\mathrm{r}$,$\mathbf{c}_\mathrm{r}$. 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 ($\mathbf{h}$, $\mathbf{c}$) has 512-dimensional $\mathbf{h}$ and $\mathbf{c}$. Because this is the initial state for the unidirectional decoder LSTM, the decoder LSTM is based on 512-dimensional vectors.

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

Data Sets for Training and Testing

The data sets consist of tab-separated files with the filename extension "tsv", with tabs used to divide the data into four columns:

1. Column 0: Original problem and question. This consists of a Boolean sentence, a period, and a question regarding a Boolean variable. For example, "(~J & B) & F & (B & F) & (B & F) & (B & F) . What is B ?" comprises 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 single-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 above example would appear in a spreadsheet as follows:

All of the training data exist in this four-column tab-separated format. These data need to be one-hot encoded to create inputs and targets for the neural network. For each line in the 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 purposes of the two training pairs are as follows:

1. 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 that the variable is true, then the network should respond with “TRUE”.
    d. If the sentence implies that the variable is false, then the network should respond with “FALSE”.
2. 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 upon receiving a request for help.
    a. Here, the question is replaced by 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 the neural network to perform two basic functions:

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

The two functions described above are central to agent operation as depicted in Figure 1. 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 should take place:

1. The sentences of the knowledge base are concatenated using the logical AND operator, which is the ampersand symbol "&".
2. The resulting large Boolean sentence is concatenated with the question following a period 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, the aforementioned pair 1 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 the logical AND operator (the "&" symbol) as above.
2. However, 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, the aforementioned pair 2 is used to teach the neural network how to respond to a request for help by providing a correct and concise dump of the agent's knowledge base.

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, which is compared with the target string. In the case of pair 1, 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 considered correct. For example, if the correct 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”, and “fALse” would all be scored as incorrect. Similarly, in the case of pair 2, where the neural network has to perform a knowledge base dump in response to a request for help, the output must exactly match the target string, which contains 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, 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, and even logically equivalent outputs that are not an exact match will be scored as wrong. This is a different metric to the character-by-character accuracy metric, in which a string with a single character error is marked as wrong, even when all of the characters but one are correct.

Data Sets

Each tab-separated file contains 25,000 lines, each of which is used to create both a logic problem (pair 1) and a problem of restating knowledge concisely (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 December 29, 2021, was trained on the first 100 files as stated above. It remains to be seen whether the 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 use the majority of these to train a new version of the network to determine whether improved accuracy can be attained.

3. The author has obtained accuracy exceeding 98% with the existing network for 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. Although the problems are randomly generated, a large percentage of the problems, unfortunately, tend to allow a network to deduce a value as true or false simply based on a single Boolean term. This is undoubtedly a weakness of the present randomized clause generator, and it will result in some shortcomings

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 with respect to data set generation are a deficiency that must be addressed in future work. The present agents are able to demonstrate the basic criteria of self-awareness that are the goal of this work, but readers are cautioned that they may make errors in Boolean reasoning, particularly for moderately to highly complex sentences. This will require improvements to the data set generation procedures. Data set generation is a random process that can be summarized as follows:

1. Boolean variables are randomly selected to populate a sentence.
    a. Some of these are randomly negated to ensure that we have both positive and negative atomic Boolean terms.
2. Binary logic operators, such as logical AND, OR, and implication, are randomly selected 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.), to determine whether the variable is true or false, or whether the input knowledge is insufficient or contradictory.
6. The clauses of the sentences may be repeated one or more times to generate the "long" versions of the sentences.
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 (e.g., "TRUE", "FALSE") becomes the column 3 entry.

The choice to include repetition in the 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 present 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 assert that this type of training actually does teach a neural network to treat terms and clauses as conceptual units.

All of the data set files are available at (Mukai, S3 Source Code and Data Sets, n.d.) to facilitate peer review and make both the strengths and 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 previously developed code for LSTM neural networks (Chollet, keras.io, n.d.) and Boolean logic (Norvig, aima-python, n.d.). 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 bidirectional 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 the training above).
    c. Compute the neural network accuracy over this data set.

5. The program run_seq2seq_demo.py is not a standalone program. It is simply imported in our web demo at (Mukai, Dual Agent Demo, 2022) to illustrate 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 the following question:

What is C ?

However, agent 1 does not possess sufficient information to ascertain the value of C. At this stage, agent 1 asks for help, causing agent 2 to dump its knowledge base. Agent 1 is now armed with all three facts:

~A

C ==> B

B ==> A

Thus, agent 1 can reason that because A is false and B ==> A, then B must be false. Because 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 that it lacks the knowledge to answer the question regarding Boolean variable C, it asks 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. It should be noted that this self-awareness is a property of the agent, which is a composite of a neural network plus a Python list knowledge base and appropriate code that performs one-hot encoding and decoding and 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. Thus, self-awareness is a property of the complete agent, not a property 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 the following:

1. Knowledge of one's own knowledge base.
    a. This is exemplified by the ability to provide a concise version of its knowledge base upon a 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 can be defined, and the definition used here is very simple. It falls significantly short of the much broader reasoning-about-knowledge capability that is 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, suggest that this is a reasonable avenue for future research. Agents using epistemic modal logic have a considerably deeper ability to reason about their own knowledge and the knowledge of other agents, which may make this a promising direction.

Because Boolean reasoning already entails substantial computational complexity, and considering that 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 variation of this work 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 represents a starting point. The form of self-awareness described here is extremely basic, but the presented work should serve as 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/

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

Chollet, F. (2017). *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*, 108, 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). *Sequence to Sequence Learning with Neural Networks.* Retrieved from arXiv: https://arxiv.org/abs/1406.1078

Mukai, R. (2022). *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). *Understanding LSTM Networks.* Retrieved from Colah's blog: https://colah.github.io/posts/2015-08-Understanding-LSTMs/