Extending Logical Neural Networks using First-Order Theories

A CPSC/AMTH 552 Final Project; 8 pages

JORGE BLANCO, Yale University
AIDAN EVANS, Yale University

Logical Neural Networks (LNNs) are a type of architecture which combine a neural network's abilities to learn and systems of formal logic's abilities to perform symbolic reasoning. LLNs provide programmers the ability to implicitly modify the underlying structure of the neural network via logical formulae. In this paper, we take advantage of this abstraction to extend LNNs to support equality and function symbols. This extension improves the power of LNNs by significantly increasing the types, and thus the number, of problems it can tackle.

Additional Key Words and Phrases: logical neural networks, neural networks, symbolic reasoning, symbolic logic, explainable AI, first-order theories

1 INTRODUCTION

A current new and promising neural network architecture, Logical Neural Networks (LNNs) proposed by Riegel et al., has shown promising results as an architecture which combines neural network's abilities to learn and systems of formal logic's abilities to perform symbolic reasoning. [Riegel et al. 2020] LNNs work by associating with each neuron in the network associated with a subexpression of a real-valued logic formula, such as weighted Łukasiewicz logic. Because of the one-to-one association of neurons to logical formulae, LNNs have the ability to represent their decisions in terms of logic; therefore, unlike other neural architectures, we have the ability to easily interpret the decisions of LNNs while still retaining the robust learning ability of neural networks. In summary, LNNs allow for a neural architecture with explainable decisions.

Moreover, because of LNNs' inherit "two-sided" nature – i.e., their one-to-one correspondence of formulae to neurons – LNNs provide programmers the ability to modify the underlying structure of the neural network without needing to actually work with the neurons themselves. In other words, LNNs allow programmers to work with neural networks at an abstracted level via easy to understand and concise logical formulae. Thus, LNNs can be seen to have two *sides*: a "logic side" and "neuron side". This notion of abstraction is akin to that inherit in the design of the Internet because the Internet was made such that one could use and design Internet applications without needing to worry about the low-level details and protocols actually used to transmit data. In this paper, we argue that this abstraction allows for the robust design of more complex extensions of LNNs without needing to work with "low-level" neurons.

LNNs operate by taking as input a *knowledge base* where each entry is a logical formula. Currently, LNNs support formulae expressed in first-order logic (FOL) – a very powerful language with respect to what one can express in it. FOL languages *typically include* predicate, variable, constant, and function symbols – along with an equality operator represented as a special predicate or logical constant symbol. The current architectural design of LNNs, however, only supports the use of expressions in FOL *without equality and functions symbols*. While equality and functions are not needed to obtain the full expressive ability of FOL, they nevertheless provide one with the ability to write and reason with formulae in a much *easier and natural way*. Therefore, in this work, in order to demonstrate the robustness of LNNs, we extend LNNs to support equality and function symbols.

Authors' addresses: Jorge Blanco, Yale University, jorge.blanco@yale.edu; Aidan Evans, Yale University, aidan.evans@yale.edu.

Furthermore, in doing so, we restrict ourselves to working at the logic side of LNNs – demonstrating the power of abstraction inherit in LNNs. Specifically, we introduce both equality and functions as *first-order theories*, i.e., additional axioms expressed in terms of and reasoned with FOL. Therefore, we need only introduce these axioms into the network via logical formulae during the construction of the network. In this work, we introduce support for these theories into the IBM's LNN Python library[1], therefore, allowing IBM's system to now reason about equality and function symbols. We additionally provide a description of what the introduction of these theories corresponds to in terms of the low-level neuron side of LNNs.

In this paper, we first provide a background on LNNs in Section 2; we focus primarily on the basic structure of an LNN, i.e., how symbols and neurons interact and the connections between real-valued logic and activation functions. We also discuss how inference and learning work in LNNs. In Section 3, we introduce the theoretical framework used to add equality and function to LNNs and discuss how the addition of equality and functions using this framework affects the neural structure of LNNs. In Section 4 we discuss the details of our attempt to incorporate equality and functions by expanding IBM's LNN module, and discuss some problems we encountered. In Section 5 we give an overview of how we worked together to produce this paper. In Section 6 we conclude and discuss future work.

2 BACKGROUND

In this section, we provide an introduction to the LNN framework created by Riegel et al. [Riegel et al. 2020]

2.1 Logical Neural Networks

Historically, the field of artificial intelligence has focused on either *statistical AI* or *symbolic AI*. Statistical AI for example includes the study of neural networks, while symbolic AI has included the study of "good old-fasioned" AI, i.e., deductive systems. Statistical AI allows for inductive reasoning which enables the model to generalize inferences from a set of given data; symbolic AI allows for deductive reasoning which enables the model to draw for-sure conclusions using formal systems of logic. Moreover, symbolic AI allows for one to have a clear, explainable sequence of reasoning steps – statistical AI tends to just be a black-box with no way to understand *why* the model made the decision it did. LNNs aim to create a bridge between these two distinct approaches and, therefore, allows a model to perform both inductive and deductive reasoning – taking advantage of the benefits of both the statistical and symbolic AI approaches.

Basic Structure of LNNs. Currently, the framework of LNNs proposed by Riegel et al. combine the capabilities of neural networks and FOL by associating neurons with subformulae of real-valued logic, specifically, Łukasiewicz logic. Overall, the structure of the neural network associated with the formula is exactly the formula's syntax tree; an example is shown in Figure 1.

In Łukasiewicz logic, instead of having logical expressions evaluate to simply **true** or **false**, they now evaluate to some real-valued number between 0 and 1; we call this number the *truth value*. We define a *threshold of truth*, $\frac{1}{2} < \alpha \leq 1$, such that we consider a statement **true** if its truth value is above α and **false** if below $1 - \alpha$.

Each neuron returns a pair of numbers between 0 and 1 called the *lower* and *upper* bounds. These numbers determine which *primary state* the neuron is in; these states are displayed in Figure 2.

Essentially, each binary logical connectives of the input formula, such as conjunctions ($\wedge$ or $\otimes$) or disjunctions ($\vee$ or $\oplus$), is associated with a neuron; the neuron's activation function is the

[1] https://github.com/IBM/LNN

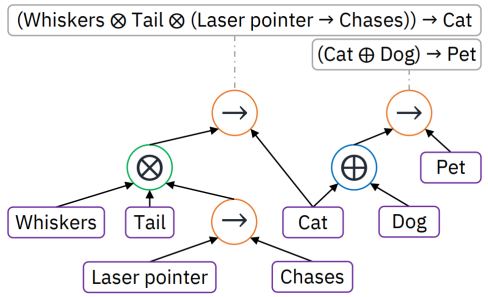


Fig. 1. Example Network [Riegel et al. 2020]

Bounds	Unknown	True	False	Contradiction
Upper	$[\alpha, 1]$	$[\alpha, 1]$	$[0, 1-\alpha]$	Lower > Upper
Lower	$[0, 1-\alpha]$	$[\alpha, 1]$	$[0, 1-\alpha]$	

Fig. 2. Primary Truth Value Bound States [Riegel et al. 2020]

real-valued logical operation, such as the following for Łukasiewicz logic, which computes the *truth-value* of the operation:

$$\begin{aligned}
\text{Conjunction:}\quad & p \otimes q = \max\{0, p+q-1\} \\
\text{Disjunction:}\quad & p \oplus q = 1 - ((1-p) \otimes (1-q)) = \min\{1, p+q\} \\
\text{Implication:}\quad & p \rightarrow q = (1-p) \otimes q = \min\{1, 1-p+q\}
\end{aligned}$$

These activation functions are then generalized to a "weighted real-valued logic" which allows one to express the importance of a subformula. The unary negation ($\neg$) connective and existential ($\exists$) and universal ($\forall$) quantifiers are represented as *pass-through nodes* which are neurons with basic unweighted activation functions. For example, negation is simply: $\neg p = 1 - p$.

FOL predicates are represented as input neurons where each predicate has an associated table of *groundings*, i.e., our known input data for what the predicate should evaluate to given various inputs. The input to the predicates of the table are the FOL constants.

Inference. To perform inference, LNNs perform an *upward* and *downward* pass through the network. The upward pass propagates the truth values from the predicates through the network, calculating the truth value of the entire formula itself. The downward pass works using the believed truth value of the entire formula to compute the truth values of its subformulae and predicates. Specifically, the upward pass will compute truth bounds for each subformula and, ultimately, the entire formula using the truth bounds of the predicates; the downward pass will then tighten these bounds for each subformula and, ultimately, each predicate until convergence. Convergence is guaranteed for propositional logic (i.e., no quantifiers and only zero-ary predicates) but is not guaranteed for FOL due to FOL's undecidability.

Learning. Because we are working with real-valued formulae, the equations used are differentiable and, therefore, we are able to use backpropagation to update the parameters, such as the weights of each connective or the truth value bounds, of the formulae. The loss function for the optimization via backpropagation aims to minimize the amount of contradiction present in the model; in other

words, it aims to remove as many contradictory conclusions as possible. This allows LNNs to learn from noisy and conflicting data sets.

3 MODEL AND THEORETICAL RESULTS: EXTENDING LNNS IN THEORY

In this section, we describe how to formalize equality and functions as first-order theories and how adding these theories into an LNN will affect its underlying structure.

Informally, a first-order theory is a set of symbols which we we may include in our FOL formulae where these symbols also have some sort of *additional meaning*. A set of axioms also included with the theory specify the additional meaning placed on these symbols. Essentially, first-order theories allow us to formalize more complex structures and concepts in which we would like to discuss: e.g., lists, arrays, trees, etc. In our case, we wish to formalize equality and functions.

3.1 Equality

3.1.1 Equality in FOL. We first wish to include a theory of equality to formalize the notion of the equality operation. The theory of equality is a common first-order theory and is a popular way to introduce an equality symbol to FOL. [Bradley and Manna 2007] To do so, we begin by introducing the equality symbol itself as a binary predicate: "=". Specifically, we will add a sort of "universal predicate" to the LNN which is defined from the start of the LNN model and, therefore, always accessible for use in any formula; we discuss the detailed implementation of this in Section 4. Meanwhile, this predicate is supposed to capture our notion of what it means for two things to be equal. Specifically, in FOL, the equality operator should be comparing two *terms*. A term in FOL is either a constant, variable, or the application of a function to another term. Because we currently do not have functions, we restrict our discussion in this subsection to terms which are simply constants or variables.

In FOL, formulae are evaluated under a specific *interpretation* or *model*. Terms are then said to *refer* to some *object* in the domain of the interpretation; under different interpretations, different terms may refer to different objects. Within an interpretation, we use the equality symbol to say that two terms refer to the same object. For example, if we had two constants, a and b, and we then said that $a = b$, this means that both a and b refer to the same object within our domain.

Using this knowledge, we then add axioms which formalize the intended meaning we wish the equality symbol to have. Because we wish the equality symbol to say that two terms are the same if the refer to the same object, we are essentially claiming that the equality symbol is an equivalence relation: it's partitioning our terms into classes which all refer to the same object. Thus, the predicate we use to represent equality should be reflexive, symmetric, and transitive. We, thus, have our first three axioms:

$$\forall x(x = x) \qquad \text{(Reflexivity)}$$

$$\forall x \forall y(x = y \rightarrow y = x) \qquad \text{(Symmetry)}$$

$$\forall x \forall y \forall z((x = y \otimes y = z) \rightarrow x = z) \qquad \text{(Transitivity)}$$

We also desire that two equal terms can be replaced by one another; i.e., we can substitute any term for one to which its equal. For example, say we had some unary predicate, P, and terms x and y such that $x = y$. Then, we'd wish to say that if Px then Py too since $x = y$. Here, because $x = y$, we should be able to replace any occurrences of x with y.

In the case of our unary predicate, P, we may state this with the following axiom:

$$\forall x \forall y(x = y \rightarrow (Px \leftrightarrow Py))$$

This now ensures that we can replace equal terms with each other in our expressions for P.

More generally, say that P is an n-ary predicate. We would then include the following axiom:

$$\forall x_1 \dots \forall x_n \forall y_1 \dots \forall y_n \left(\left(\bigotimes_{1 \le i \le n} x_i = y_i \right) \rightarrow (Px_1 \dots x_n \leftrightarrow Py_1 \dots y_n) \right) \qquad \text{(Congruence)}$$

3.1.2 Effect on Neural Structure. In order to add these axioms to our LNN model, we may add our first three axioms (reflexivity, symmetry, and transitivity) during the instantiation of our model since they only make a claim about our equality predicate, which is also introduced from the beginning; i.e., we add the three axioms as formulae of our knowledge base from the very start. This will lead our neural network to contain neurons corresponding to each axiom's formula.

In the case of our fourth axiom (congruence), however, we must handle this for every predicate; therefore, whenever a new predicate is introduced into our knowledge base, we also add the appropriate congruence axiom for this predicate, which also leads to the introduction of neurons for each of the congruence axioms.

We include a model schematic displaying the effect on the network from adding these axioms in Section A.1 of the Appendix.

3.2 Functions

For the incorporation of functions as a first-order theory, we had much less to go off of. The introduction of equality as something added "later on" to FOL is a much more common practice than for functions. For example, in the proof of Gödel's completeness theorem, we first prove the claim for FOL without equality and then prove that introducing equality does not change anything. Normally, however, functions are assumed to be included in FOL from the start – unlike equality. Therefore, for the formalization of functions, we did not have prior resources to work with. We describe our results below.

3.2.1 Functions in FOL.

Functions as "Functional Relations". To formalize functions as a first-order theory, we must understand what functions are fundamentally. Now, an n-ary function is used to map n inputs to an output such that no set of inputs can map to two different outputs. Moreover, both functions and predicates are fundamentally relations; the only difference being that the relation for a function is constrained to capture how each input may only map to one output. Formally, we say that an $(n+1)$-ary relation, R, is *functional* if and only if

$$\forall w_1 \dots \forall w_n \forall x \forall y ((Rw_1 \dots w_n x \otimes Rw_1 \dots w_n y) \rightarrow x = y) \qquad \text{(Functional)}$$

This condition ensures that if the first n inputs to our $(n+1)$-ary relation are the same, then so will the $(n+1)^{th}$ input. In other words, we may say that this ensures that the first n inputs map to the value of the $(n+1)^{th}$ input as we desire for functions. Thus, we may say that for any n-ary function, f, we can construct an $(n+1)$-ary functional relation, R_f, such that for every term x and terms w_1 through w_n,

$$R_f w_1 \dots w_n x \text{ if and only if } f(w_1, \dots, w_n) = x \qquad (\star)$$

Since predicates are simply relations, we can then rewrite each function as a predicate.

Rewriting Formulae to use Functional Relations instead of Functions. Furthermore, when it comes to how functions may be used in FOL formulae, functions are another type of term – like constants and variables. Notice that for any predicate P and term t, Pt is logically equivalent to $\exists x(t = x \otimes Px)$; see Section B.1 of the Appendix for a proof. In other words, we may extract the term out of our predicate and introduce an existentially quantified variable which must be equal to our term.

Therefore, by ($\star$) and in the case of t being a function $f(t_1 \dots t_n)$, we may say that Pt is logically equivalent to $\exists x(R_f t_1 \dots t_n x \otimes Px)$. Since the relation R_f may be represented simply as a predicate, we have now rewritten an FOL formula which contained functions in terms of an FOL formula without any functions. Since functions may themselves have functions as arguments, this procedure repeats until all the functions are removed. An example further explaining this rewriting process is given in Section B.2.

Therefore, in order to add functions to LNNs, we may simply rewrite the formulae of an LNN's knowledge base to remove functions via the rewriting procedure outlined above and then add axioms ensuring that the predicates we introduced for each function during the rewriting steps are functional.

3.2.2 Effect on Neural Structure. Similar to equality, we will add axioms for each of the predicates associated with a function; specifically, we will do so for those predicates we wish to be defined as functional. This will involve introducing several new neurons to the network to handle the axiom for each function. We again provide a model schematic displaying the effect on the network from adding these axioms in Section A.2.

We will also modify the neuronal structure corresponding to our input formula as well. Specifically, for each predicate which contains a function as an argument, we will end up adding additional neurons for each of the functions and neurons for existential quantification and conjunction since those operators are introduced from the rewriting process. Further detail on the rewriting process and can again be found in Section B.2.

4 IMPLEMENTATION

In this section, we include details of our implementations of extending IBM's LNN module to include equality and functions. Equality was fairly straight forward to implement. Functions, however, were much more difficult to implement. Overall, we were not able to do as significant testing as we wished, especially in the case of functions, because of some deep bugs in IBM's LNN implementation. The module does not contain any helpful utility functions to help debug or even to print the model's formulae in a transparent way. Moreover, the entire module is very sensitive to the form the input is represented in; for example, in some cases, it would work for a formula φ but would throw errors for a formula logically equivalent to φ; we found that their module is the most happy with formulae in prenex form using only universal quantifiers, negation, and conjunction. Nevertheless, we were still able to at least implement the extensions for equality and functions and, thus, we describe the basics of the implementation here.

Our implementation is in a forked GitHub repository[2]. The equality implementation is on the main branch; the implementation of functions is on the functions branch.

4.1 Equality

First, to use equality within a model, we specify that an LNN model instance should have support for equality during its instantiation:

```
from lnn import Model

model = Model(theories=['equality'])
```

This ensures that equality's required axioms are added from the start.

[2] https://github.com/nsnave/LNN

In IBM's LNN module, predicates are instances of a `Predicate` class. Since we are treating equality as a predicate, to add an equality operator to IBM's LNN module, we define a new variable[3] `Equals` which is an instance of the `Predicate` class; this variable is then added as an additional import like the other operators. Therefore, users have access to the variable globally.

This also allows for the `Model` class to have access to the equality variable so we can add the axioms we need to for equality within the `Model` class. During the instantiation step above, this introduces the axioms for reflexivity, symmetry, and transitivity. We add the congruence axioms for each predicate when each predicate is added with the `add_predicates` function.

Example. Say we wish to introduce a unary predicate `Dog` which tells us whether the input argument refers to a dog. We would then write the following:

```
1   from lnn import (Model, Equals)
2
3   model = Model(theories=['equality'])
4   Dog = model.add_predicates(1, "dog")
```

At this point, the model now has axioms for reflexivity, symmetry, and transitivity (via line 3) along with congruence for the `Dog` predicate (via line 4).

If we then wish to say that there is a dog named "Aggie" and that Aggie's nickname is "Fruton", we'd introduce the following facts to the model:

```
    model.add_facts({
        Dog.name: {
            'Aggie': Fact.TRUE
        },
        'equals': {
            ('Aggie', 'Fruton'): Fact.TRUE
        }
    })
```

This estabilishes that `Dog('Aggie')` and `'Aggie' = 'Fruton'` are true. We can then demonstrate our model's ability to reason about equality by having it prove that `Dog('Fruton')` is true. To do so, we introduce as an axiom that `Not(Dog('Fruton'))` is true; if this results in our model reporting that we have a contradiction, then we know that `Not(Dog('Fruton'))` must in reality be false and, therefore, `Dog('Fruton')` is true.

We first add `Not(Dog('Fruton'))` as a formula to our model as an axiom:

```
    model["query"] = Not(Dog(`Fruton'), world=World.AXIOM)
```

Then, we'll have the model deduce what it can about its knowledge base via the `infer` function and print the resulting state of our query, which ultimately reports that we have a contradiction:

```
    model.infer()
    print(model['query'].state())
```

Currently, LLNs necessarily make the *unique-names assumption*; this assumes that every constant refers to a unique object in the domain. Now that we've introduced equality, we need not make this assumption as the above example proves. Because of our introduction of equality, we were able to state that both the constants 'Aggie' and 'Fruton' refer to the same object. Thus, we were able to deduce that whatever 'Fruton' referenced was a dog because we knew that what 'Aggie' referenced was.

[3] "variable" here is being used to describe a variable in Python – not to be confused with our earlier discussion of variables within FOL.

4.2 Functions

The implementation of functions was significantly more complicated since it involved the actual rewriting of the input functions. The easy part was adding the functional axiom for each function's associated relation as this could be done akin to how the congruence axioms were added in the case of equality; namely, we introduced a new `add_functions` function to the `Model` class which allows us to declare what are functions and add their associated axioms. During the declaration of a function, we create a new instance of the `Function` class, described below, and creates an associated predicate and axiom declaring that the predicate is functional.

The complicated part involved rewriting each predicate to remove any function it may have as arguments. This involved the creating of a new `Predicate` class inheriting the original, newly-renamed `_Predicate` class along with a new `Function` class. These classes cause the input arguments to instances of the classes to recursively extract the functions and rewrite them in terms of their associated functional predicate. This rewriting procedure implements the rewriting rules discussed in Section 3.2.

Overall, IBM's current implementation of LNNs is not friendly to rewriting formulae. A better approach for introducing theories to LNNs which involve rewriting the input formulae would be to modularize the rewriting process by introducing a separate parser. This parser would take as input an abstract syntax tree for each input formulae, apply the rewriting rules, and then pass these rewritten formulae to the LNN module as usual. Currently, the rewriting is done in a much less straight-forward way in order to mesh with the current LNN module.

5 INDIVIDUAL FOCI

Overall, the project was a team effort. Because of Jorge's applied mathematics and statistics background, he focused on understanding how LNNs represent formulae and work when it comes to reasoning and learning. Because of Aidan's background in formal logic and computer science, he focused on the generation and implementation of the first-order theories. We both worked closely together to figure out how the introduction of the first-order theories affect the neural structure of the LNNs.

6 CONCLUSION

In this project, we focused on developing the theory needed to incorporate equality and functions to the LNN Model. We proposed a way to incorporate these elements by implicitly changing the LNN architecture using first-order theories. This allows us to then use equality and functions in the LNNs model proposed by Riegel et al. The addition of equality and functions allows us to use LNNs to reason about statements in FOL that are much more interesting and natural because the introduction of functions and equality make for a more natural and common FOL language. For instance, we can now express things such as $\exists x(2 + x = 4)$. Additionally, we explain how adding such elements affects the underlying structure of LLNs. With this, we have, therefore, demonstrated that the reasoning ability of LNNs can easily be extended to other domains via first-order theories.

For future work, we plan to look into the implementation of additional first-order theories such as theories of arrays, binary search tress, multisets, etc. This would then allow one to represent and reason about these structures using a neural network-based architecture. We additionally would like to provide more empirical support for the reasoning and learning ability of LLNs by testing their ability to prove theorems on the TPTP benchmark set[4]. We would also like to look into LNNs ability to recognize logical entailment as described by Evans et al. [Evans et al. 2018].

[4] https://www.tptp.org/

REFERENCES

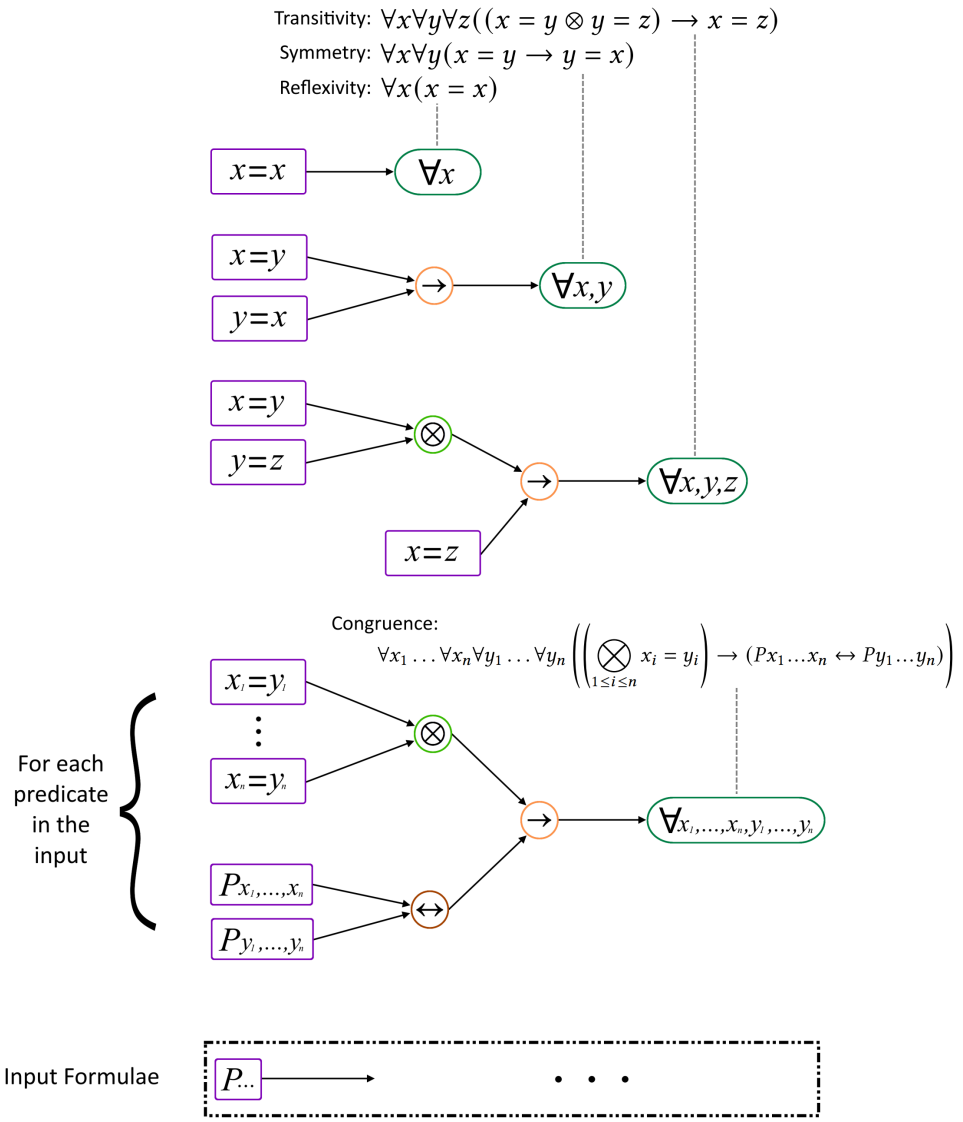
Aaron R. Bradley and Zohar Manna. 2007. *The Calculus of Computation: Decision Procedures with Applications to Verification.* Springer Berlin Heidelberg, Berlin, Heidelberg, Chapter First-Order Theories, 69–94. https://doi.org/10.1007/978-3-540-74113-8_3

Richard Evans, David Saxton, David Amos, Pushmeet Kohli, and Edward Grefenstette. 2018. Can Neural Networks Understand Logical Entailment? https://doi.org/10.48550/ARXIV.1802.08535

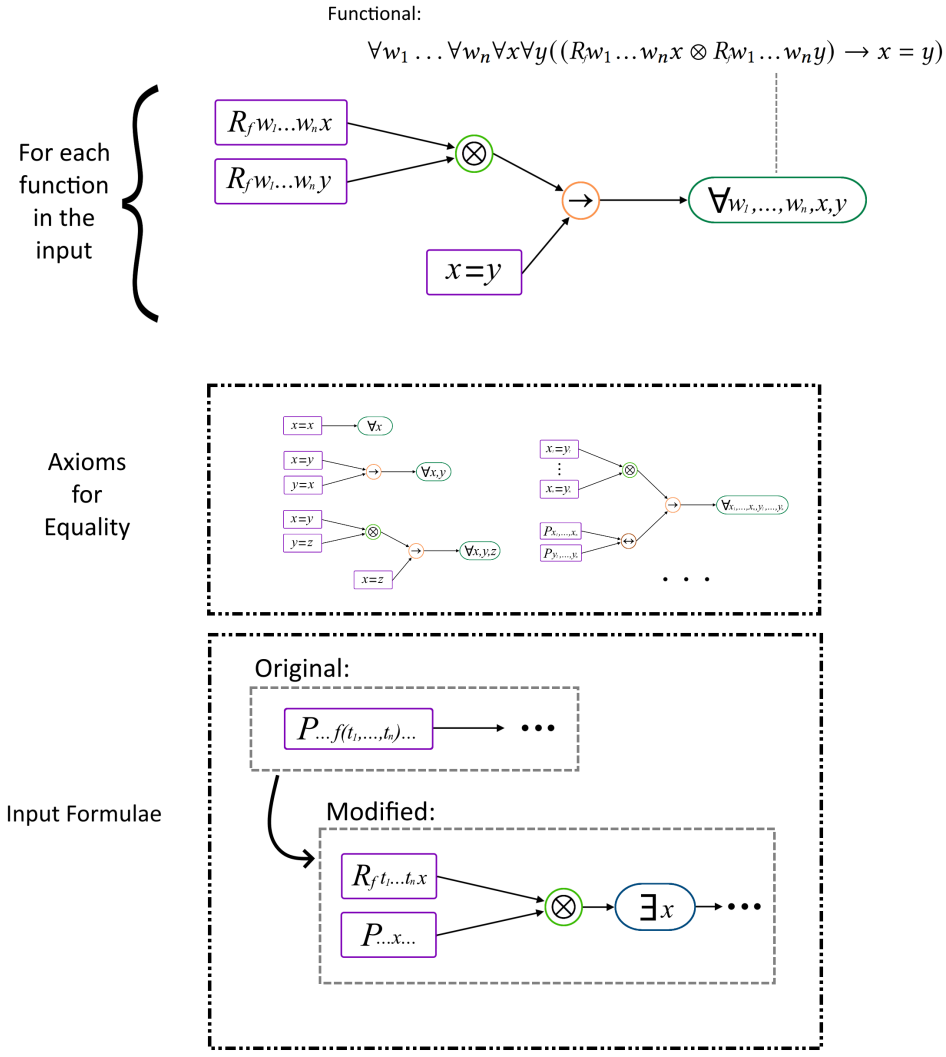
Ryan Riegel, Alexander Gray, Francois Luus, Naweed Khan, Ndivhuwo Makondo, Ismail Yunus Akhalwaya, Haifeng Qian, Ronald Fagin, Francisco Barahona, Udit Sharma, Shajith Ikbal, Hima Karanam, Sumit Neelam, Ankita Likhyani, and Santosh Srivastava. 2020. Logical Neural Networks. https://doi.org/10.48550/ARXIV.2006.13155

A MODEL SCHEMATICS

A.1 Equality Schematic



A.2 Functions Schematic



B ADDITIONAL DETAILS ON FUNCTIONS

B.1 Proof of Rewriting Rule

Theorem: *For a given n-ary predicate, P, and a chosen term t_i, $Pt_1 \dots t_i \dots t_n$ is logically equivalent to $\exists x(t_i = x \wedge Pt_1 \dots x \dots t_n)$.*

Proof:

Let P and t_i be arbitrary.

1	$Pt_1 \dots t_i \dots t_n$	
2	$t_i = t_i$	=-Introduction
3	$t_i = t_i \wedge Pt_1 \dots t_i \dots t_n$	$\wedge$-Introduction via 1, 2
4	$\exists x(t_i = x \wedge Pt_1 \dots t_i \dots t_n)$	$\exists$-Introduction via 3
5	$\exists x(t_i = x \wedge Pt_1 \dots x \dots t_n)$	=-Elimination via 4
6		
7	$\exists x(t_i = x \wedge Pt_1 \dots x \dots t_n)$	
8	$\exists x(t_i = x \wedge Pt_1 \dots t_i \dots t_n)$	=-Elimination via 7
9	$\exists x(Pt_1 \dots t_i \dots t_n)$	$\wedge$-Elimination via 8
10	$Pt_1 \dots t_i \dots t_n$	$\exists$-Elimination via 9
11	$Pt_1 \dots t_i \dots t_n \leftrightarrow \exists x(t_i = x \wedge Pt_1 \dots x \dots t_n)$	$\leftrightarrow$-Introduction via 1-10

Therefore, we have deduced that $Pt_1 \dots t_i \dots t_n \leftrightarrow \exists x(t_i = x_1 \dots x \dots t_n)$. From the Soundess Theorem, it follows that $Pt_1 \dots t_i \dots t_n$ is logically equivalent to $\exists x(t_i = x \wedge Pt_1 \dots x \dots t_n)$. □

Corollary: If t_i is a k-ary function, $f(r_1, \dots, r_k)$, then $Pt_1 \dots t_i \dots t_n$ is logically equivalent to $\exists x(R_f r_1 \dots r_k x \wedge Pt_1 \dots x \dots t_n)$ where R_f is the $(k+1)$-ary functional predicate associated with f.

B.2 Recursively Extracting Functions

To explain the recursive extraction of functions, we do so by providing a basic example of the phenomena. Say we have a unary predicate P, two unary functions f and g, and a constant c. Let R_f and R_g be the functional relations associated with f and g, respectively.

Now say we wish to remove the use of all functions from the formula $Pf(g(c))$. Since $f(g(c))$ is a term of P, we first extract this out so that P does not contain any terms which are functions:

$$Pf(g(c)) \Rightarrow \exists x(R_f g(c) x \wedge Px)$$

Note that after rewriting, however, we are still left with a predicate containing a term which is a function. While P may be function free, R_f is not. Therefore, we again apply the rewriting rules to our new formula in order to again extract any functions from the terms of our newly introduced predicate R_f:

$$\exists x(R_f g(c) x \wedge Px) \Rightarrow \exists x(\exists y(R_g c y \wedge R_f y x) \wedge Px)$$

Thus, we now have an equivalent function-free formula for $Pf(g(c))$.