

# Extending Semantic Similarity-Based Logic Reasoners: Survey of Neuro-Symbolic Approaches

## Introduction

Semantic similarity methods have recently been applied in symbolic logic systems to classify axioms and estimate truth values by comparing formulas. The current system under consideration computes pairwise semantic similarity between formulas (e.g. via model sampling) and uses these similarities to predict whether a formula is true. While effective, this brute-force approach is computationally expensive (pairwise comparisons grow rapidly) and does not leverage learning from data. The user is exploring extensions that incorporate machine learning—such as training a neural network (NN) or graph neural network (GNN) to *learn* the semantic similarity function and use it for formula classification and truth prediction. This report surveys recent advances (primarily from the last 5 years) in neuro-symbolic reasoning and related areas to assess whether that approach is the most promising, or if alternative paradigms (e.g. transformer-based logic solvers, differentiable theorem provers, or symbolic–neural hybrids) offer greater potential. We will compare these methodologies in terms of generalization, interpretability, scalability, and precision in propositional and description logic settings. We then discuss how each aligns with the user’s bottlenecks (expensive similarity computations and truth value prediction) and recommend a direction that is theoretically sound and high-potential for top-tier research.

## Background: Semantic Similarity in Logic Reasoning

**Current Method:** In the existing system, formulas in propositional or description logic are compared by sampling models (interpretations) and measuring semantic similarity. Intuitively, two formulas are considered similar if they have *comparable truth behavior* across a range of models. Such a similarity metric can be used for tasks like clustering axioms, detecting redundancies, or predicting whether a new axiom is likely true (e.g. entailed by a theory) based on its similarity to known truths. However, computing all pairwise similarities via model evaluation is costly, especially as the number or complexity of formulas grows. Moreover, it treats each new query in isolation, without *amortizing* the cost – the system doesn’t “learn” from previous computations.

**Goal of Extension:** The proposed extension is to train a machine learning model to approximate this semantic similarity function. For example, a Siamese network could embed any formula into a vector space such that distances reflect semantic similarity. If successful, this would allow one to compute an embedding for each formula once, and

then compare embeddings (which is fast) instead of repeatedly sampling models for each pair. It could also enable generalization: the network might infer similarities for formula pairs it never explicitly tested, potentially recognizing semantic patterns. The key question is whether this approach is the most *theoretically optimal and publishable*, or whether a different neuro-symbolic strategy would yield more significant advances.

In the following sections, we survey a range of relevant approaches: **(1)** learning semantic similarity and formula embeddings, **(2)** logic-aware graph neural networks, **(3)** transformer-based logic reasoning models, **(4)** differentiable theorem provers and neural-symbolic reasoning frameworks, and **(5)** hybrid systems that combine neural and symbolic methods (e.g. neural-guided search). For each, we highlight recent work, discuss generalization, interpretability, scalability, and precision, and consider how well it addresses the current system's challenges.

## Learning Semantic Similarity and Formula Embeddings

One direct approach to extend the system is **learning continuous embeddings of logical formulas** that preserve their semantic properties. In a Siamese or metric-learning setup, a neural encoder (such as a feedforward network on a formula's syntax tree, or a GNN as discussed next) maps each formula to a vector. The encoder is trained such that semantically similar formulas map to nearby vectors, while dissimilar ones map far apart. The learned similarity (e.g. cosine similarity in embedding space) then approximates the expensive model-sampling metric.

- Recent Advances:** Several works have explored embedding logical formulas in low-dimensional spaces. For instance, Saveri and Bortolussi (2023) construct *semantic-preserving embeddings* for propositional formulas using graph-based encoders. Their aim is to ensure that formulas with similar truth conditions have embeddings that are close by, a property they term *semantic consistency*. Earlier, Wang *et al.* (2017) and others applied embeddings for **premise selection** in automated theorem proving, representing formulas as trees or graphs and learning vector encodings to predict relevance to a conjecture. Both LSTM-based encoders and GNN-based encoders have been used for such tasks. More recently, researchers have investigated transformers for encoding formal logic expressions, and even fine-tuned large language models to embed propositional formulas (e.g. by representing formulas in a textual form). These models are often trained in a Siamese manner: the network processes two formulas independently and then a comparison module (like a dot product or feed-forward layer on their concatenated embeddings) outputs a similarity or entailment prediction.
- Generalization:** Learning-based similarity models can generalize beyond the exact formula pairs seen during training, capturing underlying logical patterns. For example, a neural embedder can learn that  $\$A \wedge B\$$  is semantically closer to  $\$B \wedge A\$$  (which is equivalent) than to  $\$A \vee B\$$ , even if certain combinations were not in training data. However, a network's ability to generalize is limited by the expressiveness of its architecture and training distribution. A classical result is that a *perfect* embedding of formulas that preserves *all* logical entailments may require exponentially large vectors (since propositional truth assignments grow exponentially with atom count). In practice, learned embeddings are an approximation. They tend to work well within the scope of complexity observed in training, but can struggle with formulas much larger or more complex than those seen before. Empirically, Zhang *et al.* (2023) found that neural models trained to reason on certain logical tasks fail to *systematically generalize* if test examples differ in structure or size from training. Thus, while embedding approaches **improve generalization** compared to no learning at all, they might still exhibit brittleness for out-of-distribution logic problems (e.g. more atoms, deeper nesting).
- Interpretability:** Continuous embeddings of logic formulas are typically **not directly interpretable** – each dimension is a latent feature with no clear logical meaning. This is a downside compared to symbolic similarity measures, which at least can be traced to truth-value patterns in specific models. Recent research is starting to address this by extracting symbolic rules from embeddings (e.g. distilling a neural model's decisions into logical formulas), but such techniques are not yet standard. One exception is when the embedding method itself uses **geometric structures with clear semantics**. For example, some ontology embedding models represent logical concepts as geometric regions (balls or boxes) such that one region containing another signifies subsumption (logical implication). In such cases, the learned representation (a box or ball in  $\mathbb{R}^n$ ) has a loose interpretability as a "soft concept" that approximates a set of individuals. Overall, though, most learned similarity functions act as black boxes that trade some logical transparency for speed.
- Scalability:** Once trained, an embedding model is extremely **scalable at query time**. Computing the vector for a new formula and comparing it to others is fast (essentially a forward pass and a few dot products) – a huge improvement over enumerating models or using an exact theorem prover for each comparison. Training the model can be expensive if it requires a large dataset of formula pairs with known similarities or entailment relations. However, datasets can often be generated automatically: e.g. sample random formulas and compute their truth relationship with a logic solver to use as training labels. The approach amortizes the one-time training cost over many queries. Notably, embedding methods can handle large sets of formulas by indexing

their embeddings and using fast nearest-neighbor search to find similar formulas. The primary scalability limit is the size of formulas the encoder can handle (for instance, a transformer has a token limit, a GNN might need more memory for very large graphs). In practice, GNN and transformer encoders have been applied to formulas with hundreds of symbols, and graph-based embeddings inherently handle variable-sized inputs.

- **Precision:** A learned similarity model is **approximate**. It might mis-rank some pairs or give false-positive entailment predictions if two formulas are usually but *not always* aligned. High precision can be achieved on the distribution of problems it was trained on – e.g., GNN encoders have matched or exceeded traditional methods on benchmarks for propositional entailment. For instance, a GNN-based formula embedding by Glorot *et al.* (2019) significantly outperformed LSTMs and CNNs on a propositional entailment task, coming close to perfect classification of entailment vs non-entailment pairs. Nonetheless, because the model's reasoning is not exact, there is always a possibility of error. This contrasts with symbolic methods that, while slower, are **sound and complete** by design. One way to balance precision and speed is to use the neural model as a *filter* or heuristic: e.g. use it to quickly rule out obviously irrelevant comparisons and only defer to a full logic solver for borderline cases. This hybrid use can maintain overall precision while still cutting down the workload.

In summary, learning a neural similarity function (via Siamese networks or embeddings) is a **natural extension** to the current system. It directly addresses the expensive pairwise comparisons by *amortizing* them: once the network is trained, computing similarity is trivial. This approach is grounded in a growing body of work on neural representations of logic formulas and has shown strong results on tasks like formula classification and entailment prediction. The main caution is ensuring that the model truly captures logical meaning and generalizes well, rather than overfitting to superficial patterns. Techniques like incorporating known logical invariances (symmetries) into the model architecture can help – which leads us to logic-aware GNNs.

## Graph Neural Networks for Logical Reasoning

Graph Neural Networks have emerged as a powerful architecture for reasoning over structured data, including logical formulas. A **logic-structured GNN** treats a formula or a knowledge base as a graph: for example, the abstract syntax tree of a formula or a graph where nodes represent subformulas, predicates, or constants and edges represent logical relations. The GNN then performs message-passing on this graph, producing embeddings for nodes or entire formulas that capture the formula's semantic properties. This approach has a strong synergy with logic because many logical properties are *structural* (e.g. variable occurrences, subformula sharing, graph isomorphism under renaming, etc.).

- **Capturing Logical Invariances:** Unlike sequential encoders, GNNs naturally respect certain invariances in logic. For instance, propositional formulas are invariant to commutativity of  $\wedge$  and  $\vee$  and to syntactic variations like variable renaming. A well-designed graph representation can factor out ordering or naming – e.g. treating conjunction as an unordered set of children, and treating each variable symbol as a node so that renaming corresponds to relabeling nodes. Glorot *et al.* (2019) explicitly constructed a graph encoding of formulas to enforce *order invariance* and *variable rename invariance*, which traditional sequence models struggle with. By leveraging these symmetries, GNN models avoid “shortcut learning” based on arbitrary symbol identities or token positions, and instead learn more *semantically relevant* features. This partly explains why GNN encoders have outperformed LSTM or CNN encoders on logical tasks.
- **Applications and Performance:** GNN-based methods have seen success in several reasoning tasks:
  - **Propositional Entailment:** A GNN encoder can be trained to decide if a set of propositional premises entails a conclusion. On a benchmark by Evans *et al.* (2018), a GNN achieved higher accuracy than tree-based LSTMs by better generalizing the logical rules.
  - **Satisfiability and SAT Solving:** Selsam *et al.* (2019) introduced *NeuroSAT*, a message-passing GNN that learns to predict if a CNF formula is satisfiable. Remarkably, NeuroSAT not only predicted SAT/UNSAT with high accuracy on small formulas, but it also sometimes produced satisfying assignments implicitly by clustering its internal node embeddings. This demonstrated that a GNN can learn a **search procedure** (like DPLL) in vector form. It also hinted at generalization: the network trained on small SAT problems was able to solve some larger instances by iteratively refining an embedding until it “converged” to a solution representation.
  - **First-Order and Theorem Proving:** Graph representations have been used in **premise selection** for theorem provers, where the goal is to pick relevant axioms from a large knowledge base for proving a conjecture. For example, Paliwal *et al.* (2020) used graph embeddings of formulas (in higher-order logic) to improve premise ranking, and GNNs have been used to guide theorem provers by representing the partial proof state as a graph. In another work, a GNN trained on the structure of proofs in the Mizar corpus was able to predict the *difficulty* or *length* of a proof for a given statement, which can help in

deciding which tactics to apply.

- **Generalization and Scalability:** GNNs offer good generalization to formulas of different sizes, because they are fundamentally *inductive* over graph structures. A GNN does not have a fixed input length – it can process an arbitrary-size formula by iterating messages until convergence or for a fixed number of rounds. This makes it feasible to apply a GNN trained on small formulas to moderately larger formulas. However, generalization is not limitless: extremely deeper or larger structures might require more message-passing iterations than the network was trained for, or could introduce new patterns (e.g. a logical tautology of a form not seen before) that the GNN might not recognize. Some benchmarks have shown GNNs performing surprisingly well on slightly out-of-distribution cases (like NeuroSAT solving SAT problems larger than training instances through iterative refinement), but performance can degrade beyond a point. Scalability-wise, GNN inference scales roughly linearly with the size of the formula graph (number of nodes + edges), which is quite favorable compared to exponential blow-ups in symbolic search. This means GNNs can handle reasonably large knowledge bases or formulas, though extremely large graphs (e.g. thousands of nodes) may hit memory or time limits for the message passing.
- **Interpretability:** The internal reasoning of a GNN is generally not transparent – it spreads numeric messages over the graph, which is hard to interpret as logical steps. That said, there have been attempts to extract human-understandable explanations from trained GNNs. For example, researchers have distilled GNN decisions into logical rules by analyzing node activations, and in the case of NeuroSAT, the authors observed that one neuron’s activation could be correlated with finding a satisfying assignment, providing a hint of *emergent* symbolic behavior. In general, one can inspect which parts of the graph were most influential (via attention mechanisms or gradients) to get insights. Compared to large fully connected NNs, GNNs at least ground their processing in the structure of the formula, which can make it easier to *localize* why a certain formula was classified a certain way (e.g. the network might focus on a particular subformula or a particular interaction between two predicates). Still, achieving *fully explainable* GNN reasoning remains an open challenge.
- **Precision:** With sufficient training data, logic-aware GNNs have achieved very high accuracy on various reasoning tasks. They can learn to *mimic* formal inference patterns. For instance, the DeepMind GNN model for propositional entailment could perfectly classify 2-step entailments and significantly improved on 3-step entailments, whereas prior models struggled. However, as with any learned model, there is a risk of mistakes, especially on edge cases or adversarial inputs. GNNs do not guarantee soundness – they might declare a formula satisfiable when it is actually unsatisfiable, etc., if they haven’t encountered a similar pattern before. In safety-critical domains, one might use a GNN as a fast heuristic and then double-check the answer with a traditional solver if absolute certainty is required.

In the context of the user’s system, a **GNN-based approach** could mean designing a graph representation for propositional or description logic formulas (or sets of formulas) and training a GNN to either produce an embedding (as discussed earlier) or directly output truth predictions (e.g. a binary classifier for “tautology vs not” or “entailed vs independent”). GNNs excel at **amortizing reasoning** over many similar problems: once trained, the GNN can quickly estimate the result for new formulas, thereby alleviating the need for repeated model sampling. Given their strong performance and structural alignment with logic, logic-structured GNNs are certainly among the *top contenders* for a publishable, high-impact solution.

## Transformer-Based Symbolic Manipulation and Logic Solvers

Transformer models and large language models (LLMs) have become extremely powerful at sequence tasks – and recently, researchers have probed their ability on *symbolic manipulation and formal reasoning*. The question is whether a model like a transformer, either trained from scratch or fine-tuned from a pre-trained LLM, can learn the semantics of logical formulas and perform reasoning such as satisfiability, entailment, or even proof generation. This category of approach leverages the enormous capacity and generality of sequence models, treating logic inference somewhat analogously to translation or question-answering tasks.

- **Direct Classification and Regression:** One line of work simply encodes logical formulas (or problems) as sequences – for example, as a string in some logical syntax or even a natural language description – and trains a transformer-based classifier. Richardson and Sabharwal (2022) studied the performance of transformers (like BERT or GPT) on *propositional satisfiability* instances posed in a natural language form. They found that large models could be fine-tuned to decide SAT on formulas of a fixed size with strong accuracy, essentially learning to simulate a SAT solver for that distribution. However, they also noted the models **struggled to generalize to larger instances** than those seen in training. Similarly, transformers have been fine-tuned on logical entailment datasets (with formulas in a linear notation) to predict entailment. Bhattamishra *et al.* (2020) tested

transformers on tasks defined by formal grammar rules (including some logic primitives) and found that with sufficient training data, transformers can learn these tasks, but they often rely on statistical patterns rather than systematic logical rules.

- **Language Models for Proofs:** Another approach is to use sequence models in a *generative* manner – for example, generating a proof or sequence of inference steps. Recent work like *ProofWriter* (2021) trained a T5 transformer model on synthetic logical reasoning data (rules expressed in English, questions about conclusions) to produce step-by-step proofs and final answers. The model learned to some extent to simulate *forward chaining*, outputting intermediate conclusions that lead to the final answer. The results were impressive on the provided tasks (the model answered and explained entailment queries with high accuracy), showing that transformers can internalize non-trivial logical reasoning patterns when *explicitly trained on multi-step reasoning data*. Another example is work by Saha *et al.* (2020) and Tafjord *et al.* (2021), who used language models to do *one-step inference at a time*, iteratively building a proof: the model is given the current facts and asked to propose the next fact or rule application. By chaining these inferences, they simulate a proof search guided by the LM (this can be seen as a neural-symbolic hybrid where the language model is the “brain” suggesting steps, and an optional verifier checks each step).
- **Chain-of-Thought and Prompting:** With the advent of very large models (GPT-3, GPT-4, etc.), it’s been observed that they can solve certain logical or arithmetic problems better if asked to generate a *chain-of-thought* explanation. Essentially, prompting the model to reason step-by-step (and possibly using self-consistency checks) can improve its accuracy on logical questions. This suggests that transformers do have some latent capacity for *algorithmic* or stepwise reasoning, but it often needs to be elicited. In a formal logic setting, one could prompt an LLM with a series of deduction steps (like a mini-proof) and have it continue – effectively using it as a theorem prover working in natural language. Some experiments (e.g. using GPT-4 to solve puzzle-like logical deductions or help with formal proofs in interactive theorem proving) have shown promise, though this area is very new.
- **Generalization and Scalability:** Transformers are data-hungry. They excel when provided with massive training corpora or extremely many examples of a specific task. For formal logic, such data can be generated (e.g. random formula pairs with known relationships) but ensuring the model *truly learns logic* and not just statistical quirks of the sample is challenging. As noted, transformers often **lack systematic generalization**: an LM might learn to handle formulas up to a certain size or a certain pattern seen in training, but fail catastrophically when a slightly different pattern appears. This is because they don’t inherently know the underlying rules of logic; they are approximating them from data. There has been progress in improving this (for instance, training on curricula of gradually increasing formula complexity, or adding position-aware encodings for structures), but generalization remains a concern. In terms of scalability, applying a transformer to very large formulas or huge knowledge bases might be infeasible simply due to input length limits (although one can consider iterative approaches or Longformer variants). On the plus side, a transformer can process **multiple formulas at once** if formatted properly, potentially evaluating a whole batch of queries in parallel or reading an entire theory and a query together.
- **Interpretability:** Standard transformers offer little interpretability – their multi-head attention patterns can sometimes be visualized, but drawing a clear correspondence to logical reasoning steps is difficult. However, if the model is trained to produce *explanations or proofs*, as in the chain-of-thought paradigm, the output itself can serve as a form of interpretable artifact. For example, if a model outputs a sequence of logical implications as a proof, one can verify those steps. This doesn’t mean the model is guaranteed to always produce valid proofs (it might output plausible-sounding but flawed reasoning if not rigorously constrained), but it is a step toward transparency. Some researchers enforce constraints during decoding to ensure each step is logically valid (using external checkers in a loop), which boosts interpretability and correctness at the cost of speed.
- **Precision:** A fine-tuned transformer can achieve very high accuracy on a narrow distribution of logical problems – sometimes near 100% on tasks like Horn-rule entailment or small SAT instances. But precision can drop when confronted with cases that deviate from what it “expects.” There’s also a risk of *false confidence*: unlike a symbolic solver that would simply not return a result if it cannot find a proof, a neural model will always output some answer, which might be wrong. That said, in contexts where a little error is tolerable or can be mitigated by verification, transformers offer a powerful and flexible approach.

In relation to the user’s needs, a transformer-based model could be used to **predict truth values of formulas directly**. For example, train a model on random propositional formulas labeled by their ground-truth (tautology or satisfiable or unsatisfiable etc.), so it learns to approximate a solver. Or in description logic, train on entailment queries using an ontology (perhaps turned into textual axioms) so the model answers if a candidate axiom holds. This could bypass explicit similarity computations altogether – the transformer might implicitly learn which features of formulas make them true or similar to known facts. While this approach could be *publishable*, especially given the current

interest in applying LLMs to reasoning, it has the caveat of being less *theoretically grounded*. It's essentially a statistical learner of logic, which might not satisfy those looking for provable guarantees or deep logical insight. Nevertheless, given the right benchmarks and some innovation (like combining this with symbolic verification), a transformer-based solution could be a high-impact path, leveraging cutting-edge NLP techniques for logical reasoning.

## Differentiable Theorem Provers and Neuro-Symbolic Reasoning

Differentiable theorem provers and related **neuro-symbolic frameworks** aim to blend symbolic logic with neural networks in a *continuous, end-to-end trainable* manner. The core idea is to encode the discrete structures of logic (like truth values, proof rules, logical connectives) into a differentiable computational graph so that gradient-based optimization can be used to learn or guide reasoning. This approach is quite theoretical and has been explored in both propositional logic and first-order logic settings.

- **Neural Theorem Provers (NTP):** Rocktäschel and Riedel's *Differentiable Theorem Prover* (2017) is a seminal example. It unrolls backward-chaining logic inference as a neural network – effectively creating a differentiable proof tree. In an NTP, facts and rules are embedded as vectors; unification (matching of symbols) is done by measuring vector similarity; and logical AND/OR operations are replaced with continuous approximations (e.g. using differentiable t-norms for conjunction). The network can be trained with examples (like positive and negative query answers) to adjust the embeddings such that desired proofs have high “activation”. In essence, the model learns latent representations of predicates that make certain entailments possible. NTPs have been applied to learn simple logic programs (e.g. family relations) from examples. They offer the appealing feature that the learned embeddings can sometimes be interpreted as encoding a symbolic rule (for instance, the model might learn an embedding for a relation that ends up representing a composition of other relations, implying a logical rule).
- **Differentiable Logic Programming:** Related to NTPs, there are systems like TensorLog (Cohen *et al.*, 2017) which compile a logic program into a sequence of matrix operations – these matrices (representing relations) can then be partially learned or multiplied with real-valued vectors to do fuzzy inference. Also, *Neural LP* (Yang *et al.*, 2017) uses an RNN to learn relational rules (as sequences of relations) in a differentiable fashion. These approaches typically focus on learning symbolic knowledge (like new axioms or rules) from data, rather than classifying arbitrary formulas. But they demonstrate how logical deduction can be integrated with gradient descent.
- **Logical Tensor Networks and Soft Logic:** Another thread is **fuzzy logic networks** that allow *soft truth values*. For example, *Logic Tensor Networks* (LTN) introduced by Serafini *et al.* use real numbers in  $[0,1]$  to represent truth of formulas and define differentiable operators for conjunction, disjunction, etc., based on t-norms. One can input a set of logical formulas (with some perceived truth degrees or weights) and then train embeddings for constants or unary/binary predicates such that the formulas are as true as possible. Essentially, LTNs treat knowledge base completion as a constraint satisfaction problem in a continuous space, solved via gradient descent. Similarly, *Semantic Loss* (Xu *et al.*, 2018) integrated logical constraints into the loss functions of neural networks to penalize violations of known logical relationships. These methods are particularly useful when you have a mixture of hard logic and noisy data – they enforce consistency softly and can learn to satisfy logical constraints over a dataset.
- **Differentiable SAT and ILP Solvers:** At the propositional level, researchers have even created differentiable versions of SAT solvers. *SATNet* (Wang *et al.*, 2019) is a network layer that can take in a SAT problem (in an encoded form) and output a solution, with the internal satisfaction constraints enforced via a differentiable projection operation. By inserting such a layer into a bigger network, one can train a system that includes a logical solving component inside – the gradients flow through the SAT solver approximation to influence upstream layers. This is useful for tasks where a neural net should output something that meets logical constraints; the SATNet ensures solutions are feasible during training. Another example is *differentiable ILP* (*Integer Linear Programming*) layers used in constrained learning problems. These techniques are a bit tangential to our main focus (they're often used in vision or scheduling tasks to enforce logic), but they illustrate the trend of making solvers differentiable.
- **Generalization:** Differentiable provers typically aim at *inductive generalization* – they learn underlying logic rules from data that can apply to new instances. For example, an NTP trained on a small knowledge graph might learn a rule like “if X is the parent of Y and Y is parent of Z, then X is grandparent of Z”. Once learned, this rule can apply to any new trio of people, even if they were not in the training set, showing a form of logical generalization. However, there is usually a predefined limit on the depth or form of rules (NTP might only consider proofs up to a certain depth, etc.). They may also struggle with higher combinatorial complexity or large domains, as the differentiable proof search can become intractable if too many possibilities exist. In

practice, these models work on relatively small toy problems or well-structured domains. Generalizing to full-scale theorem proving or large ontologies through pure differentiation remains an open research challenge.

- **Interpretability:** A big motivation for neuro-symbolic systems is *interpretability*. Since they incorporate symbolic concepts, one can sometimes extract human-readable knowledge from them. For instance, after training, one might examine the learned embeddings in an NTP to see if they correspond to known predicates or compose into logical rules. Some neuro-symbolic models are even constrained to produce outputs that are symbols or structures (e.g. Neural LP outputs an explicit logical rule). This is more interpretable than a black-box network, though not as straightforward as reading a manually written logic rule. There is a spectrum: approaches like LTNs and soft logic give numeric truth values which might be hard to directly interpret, whereas approaches like Neural LP yield actual rule strings. In general, differentiable provers are more interpretable than plain neural nets but less so than traditional provers (which have fully transparent proofs). They inhabit a middle ground where part of the reasoning is transparent and part is hidden in vectors.
- **Scalability:** Scalability is one of the **weak points** of these methods right now. Making logic differentiable often incurs a heavy computational cost. The proof unrolling in NTP, for example, grows with the number of potential proof paths. Training such models is slow and memory-intensive, which limits the size of problems they can handle. There have been improvements (like pruning irrelevant parts of the search or using attention to focus on promising inference paths), but pure differentiable reasoning doesn't yet scale to, say, full first-order logic with hundreds of axioms. One promising avenue is combining these with neural guidance: e.g. use a neural model to suggest which proof paths to pursue (reducing branching). But at that point, the system becomes a hybrid of differentiable and discrete components.
- **Precision:** Differentiable provers are usually *approximate but improve with data*. They might initially give low confidence to the correct inference, but as they train on more examples, they adjust to prove those correctly. They don't guarantee 100% precision because they trade some logical rigor for learnability (e.g. using a continuous relaxation might allow "almost proofs" that wouldn't count in real logic). However, if well-trained and if the domain has some tolerance (like predicting facts that are probabilistically true), they can achieve very high accuracy. In tasks like knowledge base completion, neural ILP methods have matched or exceeded purely symbolic ILP in predictive accuracy, precisely because they can handle noise and exceptions gracefully while symbolic methods either find a rule or fail. The flip side is that if a strict guarantee is needed (e.g. in verification), these methods alone are insufficient – they might miss a corner case or assign a 0.99 truth value to a statement that should be either 0 or 1.

In the context of extending the similarity-based system, differentiable logic approaches could be used to **learn the logic itself** rather than just similarity. For example, instead of learning "similarity = same truth in many models", a differentiable model could try to directly learn which combinations of premises lead to a conclusion being true. This is more ambitious: it could result in the system discovering new axioms or rules that explain when a formula is true. For description logics, one might integrate a differentiable reasoner that approximates the tableau algorithm (some initial work has been done on using neural networks to guide tableau expansion). While very attractive academically, these approaches are complex to implement and may be considered *risky* if the primary goal is a robust improvement of the current system's performance. They are likely to be publishable in top venues if one can demonstrate a significant advance (e.g. learning to reason in a fragment of logic more efficiently than known methods, or scaling differentiable proving to a new domain).

## Symbolic–Neural Hybrid Approaches

Between pure neural methods and pure symbolic logic, there is a rich middle ground of **hybrid systems** that use neural networks to assist or guide symbolic reasoning without entirely replacing it. These approaches strive to get the "best of both worlds": the learning and pattern-recognition ability of neural nets, and the exactness and interpretability of symbolic logic. We highlight a few prominent hybrid strategies:

- **Neural Guidance for Theorem Proving:** One successful paradigm is using a neural model to guide the search in an automated theorem prover. Traditional provers (like resolution-based or tableau-based provers) face huge search spaces. A learned heuristic can prioritize the most promising inference steps. For example, Loos *et al.* (2017) trained a CNN to predict which clauses were likely useful for a proof in E-prover, improving proof search efficiency. More recently, GNNs have been used to represent the state of a proof (as a graph of clauses or a proof tree) and a neural policy selects the next inference rule or premise to use. The **3SIL** system by Piepenbrock *et al.* (2021) is a clear demonstration: they used reinforcement learning to train a network that proposes *rewriting steps* for conjectures in a prover. The network suggests how to restate the goal in ways that are easier to prove (for instance, by algebraic simplifications or generalizations), effectively *steering* the prover. This neural rewriting approach improved the success rate of the prover significantly, solving more theorems

within a given time than the prover alone. Importantly, the final result is still a **formal proof**, so there's no risk of a hallucinated answer – the neural net only helps find the proof faster.

- **Neural Premise Selection (Hammers):** In interactive theorem proving and large knowledge bases, a key challenge is selecting relevant facts (premises) to use. Machine learning has been deployed to address this: given a conjecture, predict which axioms are most likely to be useful in its proof. This can be seen as a ranking/similarity problem (which ties back to the user's scenario of axiom similarity). Systems like DeepMath (2016) and subsequent works used neural networks (LSTMs, GNNs) to embed both the conjecture and candidate axioms and estimate relevance. The top-ranked axioms are then given to a prover. This hybrid can dramatically increase the proof success rate on large libraries (like the HOL Light corpus or Isabelle's libraries). The neural part quickly narrows down the search space, and the symbolic part rigorously proves the result if possible. Such a combination can scale to very large theory spaces, because the network learns from previous proofs which statements tend to co-occur, capturing *semantic relationships* between axioms.
- **Neuro-Symbolic Knowledge Graph Reasoners:** In the realm of description logics and ontologies, hybrids exist where an ontology's logical reasoning is augmented by neural components. For instance, one approach is to use a reasoner to infer some consequences and then use a neural network to fill in likely missing links (analogous to knowledge graph embedding models), or vice versa. Kulmanov *et al.* (2019) and others have embedded *OWL ontologies* into vector spaces for tasks like link prediction, but they rely on a reasoner to preprocess the ontology (e.g. compute the closure of simple implications) and then train an embedding model on that. Another hybrid strategy is using reinforcement learning on *top* of a symbolic reasoner: as mentioned earlier, Singh *et al.* (2021) proposed an RL agent that learns to apply tableau expansion rules more efficiently. The agent doesn't change the correctness of tableau (which is symbolic), but learns a policy for which branch to explore first or which rule to apply, minimizing wasted work. This can greatly speed up reasoning on hard ontologies.
- **Symbolic Constraints on Neural Nets:** A different angle is to encode symbolic knowledge *into* a neural network's architecture or training. IBM's *Logical Neural Networks* (LNN, 2020) is a good example – they design a neural network whose structure mirrors a set of logic formulas (each neuron corresponds to a clause, etc.), and truth values are soft. By training this network, they could refine uncertain rules or handle noise, all while maintaining a partially interpretable structure (you can inspect neuron weights to see learned rule confidences). Another example is *DeepProbLog* (Manhaeve *et al.*, 2018), where a Prolog program is augmented with neural predicates: the Prolog handles the discrete chaining, and whenever a subgoal requires perceptual interpretation (like classifying an image), a neural network is called. The whole system can be trained end-to-end, which is a hybrid between logic programming and neural nets. While these might be outside the direct scope of "formula similarity", they represent ways to tightly integrate logic with neural methods.
- **Generalization, Interpretability, Precision:** Hybrid systems often inherit the best traits of each side. Because a symbolic engine is in the loop, they tend to maintain **high precision and soundness** – e.g. a neural-guided prover only outputs a proof if one exists, and if it exists the proof is verifiable. They also can generalize well *in principle*, since the symbolic component can handle arbitrary new problems (given the right guidance). The neural part learns heuristics that, once trained, can apply to new problems even if they are larger, as long as they are similar enough in nature. (For instance, a premise selection network trained on many small proofs can often generalize to a new, bigger theorem by picking relevant axioms, because it has learned semantic embeddings of the math statements that are scale-independent up to a point.) Interpretability is also improved: one can often explain a result in terms of symbolic reasoning (the proof, or the selected premises), with the neural contributions being secondary. You might not fully understand *why* the network ranked a certain premise high, but since that premise appears in a valid proof, you have a clear justification for the end result. In essence, hybrids allow a **human-in-the-loop or logic-in-the-loop** that keeps the reasoning chain transparent and correct.
- **Scalability:** Many hybrid approaches are designed specifically for scalability. For example, using learned premise selection has allowed ATPs to tackle millions of axioms, something infeasible with brute-force search. Neural guidance often has negligible overhead during search, yet prunes huge swathes of the search tree. In description logics, if an ontology is extremely large, a neural pre-processing might identify a subset of axioms most relevant to a query, reducing the load on the exact reasoner. This addresses the user's concern about expensive computations: rather than comparing a query to every axiom via brute-force semantics, a learned model can shortlist likely candidates *in constant or sublinear time*. We do note that training the neural components may require a lot of problem instances (e.g. many solved proofs as training data), which can be a bottleneck if such data is scarce. But one can often generate synthetic training data or use self-play (as in 3SIL, which used the prover to generate training examples for the RL agent).

In summary, symbolic-neural hybrids offer a compelling direction: they aim to *amortize and accelerate* reasoning

without sacrificing the guarantees of symbolic logic. For the user's scenario, one could imagine a system where a GNN or transformer first generates an embedding for each formula (capturing semantic similarity), then a secondary stage uses those embeddings to do tasks like clustering or entailment *suggestions*, and finally a logic module verifies which suggested entailments are real. Such a system could handle thousands of axioms, quickly flag likely relationships using the neural part, and then confirm them with a semantic check or theorem prover. This way, the costly semantic similarity computation is learned and amortized, but final truth predictions remain trustworthy. Hybrid approaches are quite publishable – they are at the core of many recent breakthroughs in automated reasoning and often considered state-of-the-art in their respective niches (e.g. neural premise selection is used in the best theorem provers, and neurosymbolic ontology reasoning is a hot research topic). The complexity is higher, but so is the potential payoff.

## Comparative Analysis of Approaches

To clarify the trade-offs, we compare the main approaches on key criteria, and how they specifically address the **bottlenecks of pairwise similarity cost and truth prediction**:

- **Siamese/Embedding Models for Similarity:**

- *Pros:* Greatly amortize the cost of similarity computations – once each formula is embedded, similarity queries are fast. Can leverage continuous optimization to **learn semantic patterns** beyond simple string matching. Have been shown to improve classification tasks like entailment detection. Conceptually straightforward and leverages abundant research in metric learning.
- *Cons:* Require a large and representative training set of formulas/pairs to generalize well. Risk of learning superficial features if not carefully designed (e.g. might latch onto syntax patterns correlating with truth on training data). No guarantees of logical soundness – an embedding might accidentally place two logically inequivalent formulas close together. Interpretation of results is difficult (embeddings don't explain *why* two formulas are similar). Also, extreme cases (tautologies vs contradictions) must be distinctly handled or they could confuse the embedding space (since they have very different semantics but could appear similar to a network without explicit guidance).

- **Logic-Aware Graph Neural Networks:**

- *Pros:* **Naturally exploit formula structure and invariances** (like commutativity, symmetry), leading to better generalization on logical tasks. Scale to variable-sized inputs; adding more clauses or atoms mostly just enlarges the graph, which GNNs can handle with more message passing. Achieve high accuracy on complex reasoning benchmarks by effectively learning algorithmic reasoning (e.g. message passing can emulate unit propagation or resolution in a fuzzy way). GNN embeddings of formulas can replace pairwise model checks by encoding semantic info in the node messages.
- *Cons:* Still a learned model with no absolute correctness guarantee. Could require many iterations to capture long proofs or deep logical implications (if a proof requires combining 10 facts, the GNN might need 10+ rounds of message passing, which must be set or learned). Some logic constructs (e.g. negation or quantifiers) need careful graph representation, otherwise the GNN might struggle (for instance, representing negation as a node might require adding special message-passing rules). Interpretability is limited – it's hard to extract a human-readable reason for a GNN's truth prediction aside from observing which part of the graph was activated. Training can be expensive if graphs are large or if many message-passing steps are needed.

- **Transformer-Based Solvers and LLMs:**

- *Pros:* Extremely flexible and powerful function approximators. With enough data, they can learn complex mappings (from formulas to truth values or proofs). **Pre-trained LLMs** come with a lot of background knowledge and linguistic intelligence, which can be repurposed for logic tasks (e.g. encoding description logic axioms in English might allow an LLM to use its language understanding to draw inferences). Transformers can output actual *sequences* (proofs or explanations), not just yes/no, which is useful for validation and interpretability if done right. They also integrate well with modern ML ecosystems – using few-shot or fine-tuning techniques – making them convenient to experiment with.
- *Cons:* Tend to **lack systematic generalization** in formal domains. They might solve problems within a certain range but then fail unpredictably outside that range, which is risky for a logic system. They treat logic as just another sequence pattern problem, which may lead to brittle shortcut strategies (e.g. memorizing the truth tables of small combinations, rather than truly learning logical laws). Data efficiency is an issue: one may need a very large corpus of training examples to get strong performance, whereas other methods might incorporate prior knowledge (like graph structure) to do more with less

data. Also, long input formulas can exceed token limits or make training slow. From a research standpoint, a straightforward transformer application might be seen as less novel unless it involves some innovation like a specialized encoding or a new way to enforce logical consistency (since the field has already seen several attempts with mixed success).

- **Differentiable Theorem Provers and Soft Logic:**

- *Pros:* Conceptually elegant fusion of symbolic and neural – **theoretically rich** and often publishable purely on novelty. They can *learn to infer*, discovering latent logical rules or features that explain the data. Once trained, can answer queries very fast by evaluating the neural network (much like embeddings). They handle uncertainty naturally; e.g. they can say “formula is true with confidence 0.9”, which can be useful if the domain has noise or exceptions. They maintain a degree of **interpretability**: e.g. weights in a differentiable logic network might correlate with the importance of certain rules. This approach directly targets the truth-prediction problem by incorporating the logic into the model – it’s not just learning similarity as a proxy, it’s learning a *differentiable model of truth*.
- *Cons:* Implementation complexity is high – requires crafting a continuous approximation for each logical operation and often customizing solvers. Scalability is limited; these models often work in toy domains or require restricting the logic fragment (like Horn clauses, limited depth) to keep things tractable. Training can get stuck if the search space is huge (the loss surface of a logic solver is very non-convex and riddled with local minima corresponding to incorrect “proofs”). Also, the results are approximate: even if a differentiable prover says a formula is true with 0.99 confidence, that’s not as reassuring as an actual proof. In many cases, after training such a model, one would still use a symbolic prover to verify the most critical queries – which raises the question of whether the differentiable part was needed or if a simpler learning heuristic could have sufficed.

- **Symbolic–Neural Hybrids (Neural-Symbolic Systems):**

- *Pros:* Often **best of both worlds**: neural networks take on the heavy lifting of pattern recognition and heuristic search, while the symbolic side ensures exact correctness and interpretable outcomes (e.g. proofs or logically consistent models). They directly address the user’s bottlenecks by inserting ML where it can speed things up most – e.g. replacing expensive similarity checks with a neural estimator, or using a network to prune irrelevant model evaluations. Hybrids have demonstrated scaling to very large problems (millions of axioms) by cutting down the search space dramatically while still solving problems exactly. From a research perspective, a novel hybrid architecture or application (especially in description logic reasoning, which is relatively under-explored compared to theorem proving) could be highly publishable. The approach aligns with calls for AI systems that are both data-driven and logically grounded.
- *Cons:* Designing a hybrid can be complex – one has to decide how the neural and symbolic parts communicate (what data is exchanged? when is the neural component invoked?), and this often requires deep knowledge of both ML and the logic domain. There’s also a risk that the neural component doesn’t integrate well, for example giving suggestions that the symbolic part cannot exploit, or vice versa. Tuning such systems may involve more trial and error since you’re effectively tuning two systems (the neural model’s hyperparameters and the symbolic solver’s parameters). In some cases, the speed-up from the neural part may not justify the added complexity if the domain is small (but in the user’s case, we assume we are dealing with enough formulas that it *is* worth it). Finally, ensuring the neural suggestions don’t introduce bias or blind spots is important – if the network systematically misses a type of useful axiom because it wasn’t in training, the prover might start failing on those edge cases.

## Recommendation: High-Potential Direction

Considering the survey above, the **most promising direction** for a theoretically sound and impactful extension is to pursue a **neuro-symbolic hybrid approach centered around learned representations (embeddings) and symbolic verification**. In particular, a combination of a **graph-based neural encoder for formulas** and a **logic reasoning module** could offer the best trade-off:

- **Rationale:** A GNN-based or Siamese network encoder can *learn the semantic similarity function*, addressing the immediate bottleneck by amortizing expensive computations. This component alone is an advancement: it leverages recent progress in representation learning for logic and can be evaluated in its own right (for instance, by how well its similarity scores predict actual entailment or classification of axioms). To make the work top-tier, one can innovate in this encoder – e.g. designing a novel graph representation for description logic axioms, or introducing an *invertible embedding* that can be decoded back into a formula. Such contributions are theoretically interesting (they touch on the invertibility and completeness of the logic-to-vector

mapping) and would be of interest to venues focusing on neural-symbolic learning.

- **Augmenting with Symbolic Reasoning:** Instead of relying solely on the neural model's output, integrate a step where **predictions are verified or refined by a logic solver**. For example, the neural model might rapidly predict "Axiom X is likely true given the theory" or cluster axioms by similarity. Then, a symbolic reasoner (an automated theorem prover or a model checker) can be used to *confirm* the highest-confidence predictions or to derive the actual proof if it exists. This hybrid ensures that the final outcomes remain sound and interpretable (you can provide an explanation, like "X is true because it was similar to known axiom Y and indeed Y entails X under the logic's rules"). From a research standpoint, showing that your system can learn to propose the right conjectures and then formally verify them will demonstrate both efficiency and rigor – a combination valued in top-tier AI conferences. Piepenbrock *et al.*'s work provides a precedent of how effective this approach can be: their neural network proposed rewrites that a prover then used to finalize proofs, yielding state-of-the-art results. A similar methodology can be applied here (proposing likely entailments or consistency checks, then verifying).
- **Exploiting Transformers as a Component:** If one aims to be at the cutting edge, one could incorporate transformer-based techniques in the hybrid model – for instance, use a transformer to handle parts of the task that involve sequential reasoning (like generating a candidate proof outline in natural language or some linear format), which the symbolic module then attempts to formalize. However, given the specific strengths needed (structural understanding of formulas), a GNN is likely more suitable for the core embedding. Transformers could be auxiliary, e.g. for processing any *non-logical input* or for proof synthesis once relevant axioms are identified. Including an LLM-based aspect could increase the paper's appeal (since LLMs are a hot topic), but it should be done in a way that genuinely improves performance (perhaps in handling the description logic's terminology or bridging symbolic formulas with textual explanations).
- **Focus on Description Logic Use-Case:** Many existing works target propositional or first-order logic in mathematics; relatively fewer target description logics (which underlie ontologies like OWL). A high-impact research path could be demonstrating your approach on **description logic reasoning tasks** – for example, predicting subsumptions or inconsistency in large ontologies using neural guidance. You could build on ideas like Box embeddings for EL++ or the RL-guided tableau, but go further by combining them. A potential system could embed concepts and axioms as vectors (so that subsumption hierarchies are reflected by vector inclusion, similar to box/n-ball embeddings) and simultaneously train a neural policy that, given a complex query, suggests a sequence of reasoning steps (like which part of the ontology to focus on). The final verification can be done by a DL reasoner, ensuring no false positives. This line of work would be novel and align with the needs of semantic web applications that require scalable yet reliable reasoning.
- **Generalization and Learning:** Emphasize in the research the aspects of *learning to reason*. For a top-tier contribution, it's not enough to apply a known technique – you should show new insights. For instance, perhaps your experiments reveal that the neural similarity model uncovers an interpretable structure of the theory's latent concept space (e.g. clusters of axioms that correspond to intuitive groupings, which you could visualize). Or you might prove that under certain conditions (like a limit on formula size), the neural model *provably* preserves truth for a fragment of logic (a theoretical guarantee). Combining theoretical analysis with empirical success would make the work stand out. Another angle is to show **improved generalization**: e.g. training on smaller problems and succeeding on larger ones by using the neural-symbolic approach, thus overcoming the typical neural network limitation through the aid of symbolic reasoning.

In conclusion, while simply training a GNN to mimic the semantic similarity function is a solid step (and likely to yield a publication in a specialized conference), the **most powerful and publishable approach** is one that **integrates neural similarity learning with symbolic reasoning**. This could involve a Siamese GNN that embeds formulas, combined with a logic module that uses those embeddings for fast indexing of potential inferences and then verifies them. Such an approach directly tackles the expensive computations by **learning an internal model of the logic**, offers scalability (via fast embedding lookups), maintains precision (via final symbolic checks), and provides interpretability (via extracted proofs or analogies from similar known axioms). It aligns well with recent trends in neuro-symbolic AI that seek robust reasoning by marrying neural networks with symbolic structure.

By surveying the landscape, we see that no single method is a silver bullet: each has pros and cons. However, a carefully designed hybrid can harness the strengths of each. The recommended path – a neural-symbolic hybrid with learned semantic similarity – stands out as theoretically sound (grounded in logical semantics), novel (few have applied it to description logics with the latest deep learning techniques), and practically impactful (promising significant speed-ups and new capabilities). Pursuing this path could lead to *top-tier research* results, potentially opening new avenues in how AI systems learn and reason with formal knowledge.

## References (Key Sources)

- Gaia Saveri, Luca Bortolussi. *Towards Invertible Semantic-Preserving Embeddings of Logical Formulae*. NeSy 2023. (Introduces a GNN-based autoencoder for logic formulas; discusses semantic consistency in embeddings).
- Xavier Glorot et al. *Learning Representations of Logical Formulae using GNNs*. NeurIPS Workshop on Graph Representation Learning, 2019. (Demonstrates GNNs capturing invariances like variable renaming and outperforming LSTMs on entailment tasks).
- Steven Schockaert et al. *Can Language Models Learn Embeddings of Propositional Logic?* LREC 2024. (Surveys neural approaches to formal logic reasoning, including premise selection with GNNs and transformers for logic).
- Jelle Piepenbrock et al. *Guiding an Automated Theorem Prover with Neural Rewriting*. IJCAR 2022. (Uses reinforcement learning to guide prover search via neural network proposals, significantly improving success rates).
- Gunjan Singh et al. *Neuro-Symbolic Techniques for Description Logic Reasoning*. AAAI 2021 (Student Abstract). (Proposes ontology embeddings in vector space and RL-based tableau reasoning for OWL DL, highlighting neuro-symbolic reasoning challenges in description logics).
- Zhaoyu Li et al. *A Survey on Deep Learning for Theorem Proving*. arXiv 2024. (Comprehensive overview of neural approaches in formal and informal theorem proving, including transformer-based proof generation and neural theorem provers)..
- Luís C. Lamb et al. *Graph Neural Networks Meet Neural-Symbolic Computing: A Survey and Perspective*. IJCAI 2020. (Background on how GNNs can be applied in symbolic domains and the importance of combining learning with reasoning).