Differentiable Functions for Combining First-order Constraints with Deep Learning via Weighted Proof Tracing

Naveen Sundar Govindarajulu  
RealityEngines.AI  
Rensselaer AI & Reasoning Laboratory (RPI)  
San Francisco  
CA 94103  
naveen@realityengines.ai

Colin White  
RealityEngines.AI  
Carnegie Mellon University  
San Francisco  
CA 94103  
colin@realityengines.ai

Abstract

Effectively training deep learning models often requires massive amounts of labeled data. Recent work has shown that accuracy on small or noisy datasets can be improved by incorporating domain knowledge into a neural network in the form of logic constraints. However, this line of work does not yet scale well to more complex forms of constraints as these approaches use logics such as propositional logic or fragments of first-order logic. One source of scalability issues is the expressivity bottleneck of such systems. Information such as the number of relevant objects, mathematical information or information about deep nested hierarchies can be cast in first-order logic with a linear number of formulae, but propositional logic and fragments of first-order logic may require in the worst case an exponential number of formulae to represent the same information.

In this work, we design a new framework for incorporating knowledge in arbitrarily complex logics (such as full first-order logic) into neural networks. Our framework converts arbitrary logic constraints into a differentiable term that can fit into the loss function of a neural network and can be used with gradient descent. We use weighted proof tracing to get an initial differentiable function, and then we use a logic minimization algorithm to finally compress and simplify the function. To the best of our knowledge, this is the first technique that can incorporate general first-order logic statements into a neural network. To demonstrate our technique, we run several object detection experiments on sets of MNIST and Fashion-MNIST images, and we show that our method significantly improves the performance of neural networks, especially in the case of noisy or missing labeled data. We also run debiasing experiments with the UCI income dataset and show that our approach can produce improved bias scores without a significant loss in accuracy.

1 Introduction

Since the deep learning revolution in 2012, neural networks have been the leading method for a wide variety of tasks including image classification and object detection. The dominant approach is to train a neural network on huge labeled datasets by minimizing task loss. A natural question is whether accuracy or efficiency can be improved by incorporating domain knowledge. For example in object detection, a bicycle is more likely to appear in the same image as a car than a dining room table. Many object detection datasets have a hierarchical set of labels, therefore, there is a huge set of constraints governing the likelihood of objects appearing in the same image as other objects. For example, many types of food are likely to appear in images together, and with kitchen appliances,
but not with sea creatures. However, most work on combining logic rules with deep learning are not scalable enough to incorporate large sets of constraints, or cannot encode complex logic constraints.

In this work, we design a new framework for incorporating arbitrary logics with neural networks. Our approach is based on representing knowledge and constraints in any logic with an inference system. We assume that a neural network outputs probabilities for a set of formulae. Our goal is to ensure that the probabilities output by the network adhere to the information present in the constraints. Inference in classical logics is not differentiable. Therefore, we present an approach that converts inference into a differentiable format by tracing all possible inference paths that lead to a violation. Specifically for first-order logic (FOL), we use a resolution theorem prover to perform this proof tracing. We finally simplify and compress the derived function by using the Quine-McCluskey algorithm for minimization of logic formulae.

We demonstrate this approach with standard first-order logic. In our instantiation of the framework, we use first-order resolution theorem proving for inference to turn arbitrary sets of constraints into a differentiable term that can fit into the loss function of a neural network. To the best of our knowledge, this is the first technique that can incorporate full first-order logic statements into a neural network. To demonstrate our technique, we run several object detection experiments on sets of MNIST and Fashion-MNIST images, and we show that our method significantly improves the performance of neural networks, especially in the case of noisy or missing labeled data. We also run debiasing experiments with the UCI income dataset and show that our approach can produce improved bias scores without a significant loss in accuracy.

**Need for First-order Logic** Why do we need the full power of first-order logic? Many important problems and domains have usable first-order axiomatizations. For example, consider fundamental theories of physics. There are robust first-order logic axiomatizations of special and general relativity presented by Andréka et al. [1]. All of their axioms are cast in and require standard first-order logic. As far as we are aware of, all automated reasoning and inference projects in this space have been done using full first-order logic [27] [12]. More closer to standard AI, we have extensive axiomatizations of common sense knowledge in full first-order calculi such as the event calculus [23]. Planning formalisms such as PDDL [11] are based on variants of first-order logic. In the industry, business rules and knowledge are cast and formulated in different fragments of first-order logic, such as SQL [19], RDF [6], OWL [34] etc.

2 Related Work

There are some notable prior approaches that try to achieve some combination of neural networks and knowledge based systems. We go through some of them that are relevant to our work. Xu et al. present a system for integrating knowledge into a neural network through the network’s loss function [33]. They represent constraints as a formula $\phi$ in propositional logic. The network’s outputs assign probabilities or weights to the atoms in the formula. Weighted model counting of $\phi$ is then used as a component in the loss function. Weighted model counting should be done in a manner that is differentiable so that it can work with gradient descent training methods. While this approach is elegant, it suffers from two scalability issues. The first issue is that propositional logic is not scalable enough to represent knowledge in many real-world applications. Second, Xu et al. use model counting in their loss function. Model counting does not scale up for first-order logics, as quantified logics can have unbounded models even for small sets of constraints.

Hu et al. [14] build neural networks by distilling a set of rules using soft logic. In their approach, a given set of logic statements are essentially considered to be another parallel model. In [16], Krishna et al. follow up with more extensive experiments that cast doubts on the efficacy of Hu et al. when random effects are taken into account.

Manhaeve et al. [20] consider a differentiable form of ProbLog, a probabilistic version of Prolog. [2] where the weights for the clauses come from the outputs of a neural network. They present a form of training which can adjust both the weights of the network as well as the weights of clauses in a ProbLog program. DeepProbLog aims to address a different problem than what we address here. In DeepProbLog, a neural network provides inputs to a logic program. The parameters of the logic program can be jointly trained with the neural network.
Marra et al. [21] present a system in which arbitrary first-order formulae can be converted into optimization problems by defining first-order connectives and quantifiers as differentiable operations over ground atoms. While general, this approach requires considering $n^k$ ground atoms where $n$ is the size of the domain and $k$ is the number of variables in a formula. Cohen et al. [5] present a differentiable subset of first-order logic, TensorLog, that addresses this complexity issue. The subset considered is made up of Datalog clauses [13]. While Datalog clauses can represent a wide range of constraints, some of the problems we are interested in make it unwieldy (and almost impossible) to use Datalog clauses (or even Horn clauses). For an example, see formula $\phi_3$ in the next section. We briefly list out and discuss a variety of other relevant approaches in Table 1.

### 3 Background and Problem Setting

While the framework we present is general enough to handle logics that have an inference system, we demonstrate using standard first-order logic. In first-order logic, formulae represent information about the world and can be used to encode knowledge and constraints about a problem or domain. For a brief overview of the syntax and a commonly used proof calculus for first-order logic, see Appendix A. For example, the formula $\phi_3$ below states that there are at most three red items.

$$\exists item_1, item_2, item_3 : \forall item \left( \left( \text{color}(item) = \text{red} \right) \rightarrow \begin{cases} item = item_1 \\ \lor \\ item = item_2 \\ \lor \\ item = item_3 \end{cases} \right)$$

As discussed above, there are several advantages to using full first-order logic over logics such as propositional logic or fragments of first-order logic. Particularly here, note that conditions on bounds $[\text{min}, \text{max}]$ of the number of objects are not expressible in propositional logic without restoring to an exhaustive enumeration of conditions (which can be exponential in $[\text{min}, \text{max}]$ in the worst case as opposed to linear in $[\text{min}, \text{max}]$ in first-order logic). The tradeoff in moving to a more expressive logic such as first-order logic is that inference can be more expensive. $\phi_3$ is not a Horn clause.

Inference in first-order logic is done through many different proof calculi. The broad goal in inference is to derive a goal formula $\phi$ from a set of formulae $\Gamma$. For instance, from the above formula stating that there are not more than three objects, we can derive a formula $\phi_3$ which states that there are not more than five objects. If $\phi$ is derivable by $\Gamma$ we denote that by $\Gamma \vdash \phi$. In this example, we have $\{\phi_3\} \vdash \phi_3$. If a set of formulae $\Gamma$ can derive all formulae in another set $\Gamma'$, we also represent that by $\Gamma \vdash \Gamma'$.

A proof is a record of one particular such derivation and can be represented by a sequence of formulae, and inference rule names. A proof is minimal if no premise can be derived from the rest of the premises. We denote the set of all minimal proofs of $\phi$ from $\Gamma$ by $\Lambda(\Gamma, \phi)$.
Over the last few decades, several systems and techniques have been developed that make inference and proof search in first-order logic tractable for many practical problems [29]. Moreover, in our framework logic-based inference is performed offline during model compilation time. No logic-based inference is then performed during training or deployment.

**Problem Definition:** Assume we have background knowledge encoded as a set of formulae \( \Gamma \). We have a neural network \( \eta \) that has as part of its output a vector of probabilities \( p \) for some set of formulae \( \mathcal{Y} = \{ y_1, y_2, \ldots, y_n \} \). Each component \( p_i \) of \( p \) gives us the probability for some formula \( y_i \in \mathcal{Y} \) holding, i.e., \( \text{prob}(y_i) = p_i \). As a shorthand, we denote probability for formula \( y_i \) holding by \( \eta(y_i) \). The set of formula is said to be logically mutually independent, if for all \( y_i, Y - y_i \not\vdash y_i \).

Define the minimal negation of a formula as below:

\[
\overline{\psi} = \begin{cases} 
\phi & \text{if } \psi \equiv \neg \phi \\
\neg \psi & \text{otherwise}
\end{cases}
\]

(1)

We extend the network to output probabilities for the minimal negation of each formula \( y \in \mathcal{Y} \) by taking \( \eta(\overline{y}) = 1 - \eta(y) \). If \( S \) is a set of formula, let \( \overline{S} \) denote the set with all the formulae in \( S \) minimally negated, \( \overline{S} = \{ \overline{s} \mid s \in S \} \). We also take \( \mathcal{Y} = \mathcal{Y} \cup \overline{\mathcal{Y}} \).

For \( \psi \notin \mathcal{Y} \), we define \( \eta \) to be as follows:

\[
\eta(\psi) = \begin{cases} 
\prod_{y \in \mathcal{Y}} \eta(y) & \text{if } \mathcal{Y} \vdash y \\
1 & \text{otherwise}
\end{cases}
\]

(2)

**Example:** Consider a neural network classifying images of digits as odd, even or divisible by four. \( \Gamma \) could then be the following set of constraints:

\[
\Gamma = \left\{ \begin{array}{c}
\text{odd} \rightarrow \neg \text{even} \\
\text{divisible}_4 \rightarrow \text{even}
\end{array} \right\}
\]

The network takes as input an image and has three outputs \( \langle p_1, p_2, p_3 \rangle \) such that \( \text{prob}(\text{odd}) = p_1, \text{prob}(\text{divisible}_4) = p_2, \text{prob}(\text{even}) = p_3, \text{prob}(\neg \text{odd}) = 1 - p_1, \text{prob}(\neg \text{divisible}_4) = 1 - p_2, \) and \( \text{prob}(\neg \text{even}) = 1 - p_3 \).

### 4 A Differentiable Function from Constraints

We now define a function \( d_\Gamma(\eta) \), a function of the set of first-order formula \( \Gamma \) and the vector \( p \), that conveys how well the output of the neural network adheres to the constraints in \( \Gamma \). The function \( d_\Gamma(p) \) should be differentiable so that it can be used in gradient descent training methods.

First, we define the weight of proving \( \phi \) from a set of formula \( \Gamma \) as shown below in [3] If we can not derive \( \phi \) from \( \Gamma \), the weight is zero. Otherwise, the weight is the product of the probabilities of all the formulae in \( \mathcal{Y} \).

\[
\omega(\Gamma, \phi, \eta) = \max_{\rho \in \Lambda(\Gamma, \phi)} \prod_{y \in \text{premises}(\rho)} \eta(y)
\]

(3)

We specify \( d_\Gamma(\eta) \) as the sum of the weights of all possible ways to reach a contradiction when we add to the constraints formulae in \( \mathcal{Y} \).

\[
d_\Gamma(\eta) = \sum_{Y \subseteq 2^\mathcal{Y}} \omega(\Gamma \cup Y, \bot, \eta)
\]

(4)

\( d_\Gamma(\eta) \) is a sum of product terms, where each product term is either \( \eta(y_i) \) standing in for \( y_i \), or \( 1 - \eta(y_i) \) standing in for \( \neg y_i \), as shown in Figure [4]

---

1 More precisely, this is equivalent to saying that the formula \( y_i \) is true in some interpretation or model, e.g. the real world, \( \text{prob}(W \models y_i) = p_i \).
For a given neural network architecture, we have a fixed set of formulae $Y = \{y_1, \ldots, y_n\}$ that are assigned weights or probabilities by the network. Equation 4 contains a sum over a powerset of a set of formulae. While this term is computed only once per network architecture and set of constraints, this computation can grow quite large if we add constraints during the course of a neural network’s deployment. For monotonic logics, such as first-order logic, we note that if $Y' \subset Y$ and $\Gamma \cup Y \vdash \phi$, then $\Gamma \cup Y' \vdash \phi$. We can use this information to reduce the amount of computation. If $\omega(\Gamma \cup Y, \bot, \eta) = 0$ then $\omega(\Gamma \cup Y', \bot, \eta) = 0$ for all $Y' \subset Y$.

Note $\omega(\Gamma, \phi, \eta)$ is differentiable if $\eta$ is differentiable. Therefore, the constraint loss term $d_\Gamma(\eta)$ is differentiable if $\eta$ is differentiable.

The number of terms in $d_\Gamma(\eta)$ can grow quite large. To address this issue, we simply $d_\Gamma(\eta)$ by applying the Quine McCluskey algorithm [22] for simplification of logic formulae. For computing $\Gamma \vdash \phi$, we use Vampire, a state-of-the-art theorem prover for first-order logic [30, 24, 29].

**Adding $d_\Gamma(\eta)$ to a Neural Network** Given $d_\Gamma(\eta)$, we can either add it to the loss term in a neural network (or any differentiable form of learning system) or use it to modify the outputs of any other system. In the first approach, we use a new loss term as shown below:

$$
\text{loss}(y_{\text{true}}, y_{\text{pred}}) = c_1 \cdot \text{loss}(y_{\text{true}}, y_{\text{pred}}) + c_2 \cdot d_\Gamma(\eta)(y_{\text{pred}})
$$

In the second approach, given that output vector $\hat{y}$ from $\eta$, we modify it by taking small steps in the direction opposite of the gradient of the loss term:

$$
\hat{y} = \hat{y} - \alpha \nabla (d_\Gamma(\eta))
$$

**Interpretation** The logic formulae $\Gamma$ can be thought of as dictating the outputs of the neural network. The derived loss function can be thought of as increasing the loss when the network strays away from the logic formulae $\Gamma$.

5 Experiments

In this section, we present several experiments demonstrating the use of constraints in empirical settings. For all the experiments, we use a system that we have implemented that takes in constraints represented in standard TPTP notation and converts them into a Python function that can be used with standard deep learning frameworks. See the appendix for an example.

**Experiment 1: MNIST Fashion Collection** We consider labeling of multiple objects in an image. We use items from MNIST fashion as objects. MNIST fashion is a drop-in replacement for MNIST digits and is supposed to be more reflective of real world image processing tasks [32]. Assume that we are given a set of images that contain objects from MNIST fashion. We might know that the images satisfy certain conditions. For example, one set of conditions could specify there should be

---

[2] This system will be released with an open-source license.
items in the image that let us assemble a complete set of clothing (footwear items, lower and upper clothing items). The constraints could also specify that there should not be pairs of clothing items that usually do not go together in the image (e.g., dresses and sneakers). We could also have constraints that specify the minimum and/or maximum number of items in the images.

To model this scenario, we generate data that satisfies one such particular such set of conditions. Some examples are shown in Figure 2. The constraints are shown in TPTP format in Appendix B. To simulate missing data, we randomly drop a fraction of the labels. We then train and test neural networks with and without the constraints at different levels of training data and with different amounts of labels missing. Results are shown in figures 9 and 4. We can see that with a large fraction of labels missing from the training data, constraints are helpful even when we train with large amounts of data. When we have all training labels, we see there is not much improvement in using information in the constraints. When we use only the constraint loss term, we see that performance of the network collapses even when trained with more data.

**Experiment 2: Transparently Debiasing with Constraints**

In the last few decades, there has been a huge increase in machine learning applications that have had significant impact on human lives. For example, machine learning systems are used to process loan applications, decide whether

Figure 3: MNIST Fashion Multi-object Accuracies (low training data). Results are over 5 randomized runs. We use $c_1 = 1 - \text{scaling factor}$, $c_2 = \text{scaling factor}$. Darker colors show increasing training data. The horizontal axis shows the scaling factor and the vertical axis is accuracy. The top row shows networks trained with constraint loss functions and the bottom row shows output modification applied to the networks. Columns indicate fraction of missing labels.
defendants should be granted bail, screen job candidates etc. This has led to societal and historical biases creeping into machine learning systems. To address this, there has been a flurry of research into quantifying and removing bias in such systems [31]. Implementations such as AIF360 [2] provide many such measures and algorithms for removing bias. It should be noted that, in general, it is not possible for a system to satisfy multiple definitions of bias [4].

In this experiment, we apply constraints to debias a random forest classifier when trained on the income dataset [17]. The goal is to predict whether a person earns above $50,000 given a certain set of features. By convention, a label of 1 is desirable and 0 is undesirable. We are interested in detecting and removing bias with respect to two features, race and sex. Many debiasing algorithms remove bias by changing predictions that are in some ambiguous or uncertain zone. For example, the algorithm presented by Kamiran et al. [15] achieves debiasing by flipping labels from 0 to 1 if the data instance belongs to an unprivileged class (and vice-versa for privileged instances) and the prediction probability for the label is in a small region around 0.5.

With constraints, we can develop debiasing algorithms that are more transparent and achieve a similar effect. For instance, we can represent a procedure similar to the one by Kamiran et al. [15] by using the constraint:

$$\Gamma = \{ (\text{unprivileged} \text{(race)} \lor \text{unprivileged} \text{(sex)}) \rightarrow 1 \}$$

We then modify the outputs of the classifier using the above constraint and using 6 with $\alpha = 0.1$ with 10 iterations. Figures 5a and 5b show the how the modified classifier performs with respect to the base classifier. As the figures show, there is a slight improvement in many bias scores (lower score magnitudes are better) without much drop in accuracy. Eventually, such constraints can be checked against formalization of legal requirements that have been cast in first-order logic [9].

**Experiment 3: MNIST Divisibility Classification** In this experiment, we look at classifying handwritten digits as odd, even and/or divisible by four. Our inputs are MNIST images [18] and outputs are three binary labels indicating whether the input is odd, even or divisible by four. While given enough data, modern neural networks can learn a proper distribution of the three outputs, we consider a scenario where we have noisy labeled training data, as is common in many applications. To simulate noisy labeled data, we flip each of the three labels with a probability $p$ that we term as the noise factor.

$$\Gamma = \{ \text{odd} \rightarrow \neg \text{even} \}$$

We train with different amounts of training data and labeling noise, and test with 10,000 samples. For adding in the constraint loss term, we set $c_1 = c_2 = 1$. Results are shown in Figure 6. As can be seen
in the figure, when labeling noise is absent, there is a small improvement in accuracy when we train with a very small amount of data. As the noise increases, we get an improvement with constraints even when we train with larger amounts of data.

6 Conclusion

We have presented a new form of differentiable functions derived via proofs from information expressed in logics such as first-order logic. We then use these functions to integrate knowledge and constraints about a problem or domain either in the loss function of a neural network or in a modification layer after the outputs are computed by the neural network (or any other system). Notably, the new function is syntactic in nature as opposed to semantic forms of similar loss functions considered previously. This has advantages in translation of large constraints, and in the use of the constraints themselves, as syntactic reasoning is more scalable for large domains. As far as we are aware of, this is the first such work that combines general constraints in standard first-order logic with a neural network. In future work, we will also be looking at more real-world datasets and constraints. We also note that we have not specified the formal semantics for our system and we will be investigating how our system fits in formally with classic logic-based systems.
References


A First-order Logic Overview

We give a quick overview of the grammar of first-order logic below. For any given domain, we have a signature that specifies the symbols that we can use to build formulae. We have a set of variables \( x_0, x_1, \ldots \), a set of constants \( c_0, c_1, \ldots \) and a set of function symbols \( f_1, f_2, \ldots \). Each function symbol has an arity specified by \( \text{arity}[f_i] \). A term is any variable \( x_i \), constant \( c_j \) or a function symbol combined with other terms \( f_k(t_1, \ldots, t_n) \) such that \( n = \text{arity}[f_k] \). The signature in the domain also specifies a set of predicate symbols \( P_1, P_2 \ldots \). The predicate symbols have arities given by \( \text{arity}[P_i] \). A predicate symbol applied to a set of terms \( P_k(t_1, \ldots, t_n) \) such that \( n = \text{arity}[P_k] \) gives us an atomic formula \( A \).

From the atomic formulae, we can build formulae using the following grammar:

\[
\phi := \begin{cases} 
A; & A \text{ is atomic} \\
\phi_1 \land \phi_2 \\
\phi_1 \lor \phi_2, & \phi_1 \rightarrow \phi_2 \\
\neg \phi \\
\forall x_i \phi, & \exists x_i \phi
\end{cases}
\]

**Semantics:** An interpretation (or world) is any function from the atoms to truth values \( \{\top, \bot\} \). Associated with each of the connectives \( \{\land, \lor, \rightarrow, \neg\} \) are truth functions that assign truth values to the output formula given truth values for the input formulae. A set of formulae \( \Gamma \) is said to model or satisfy \( \phi \) if all interpretations that make all formulae in \( \Gamma \) true also make \( \phi \) true. This denoted by \( \Gamma \models \phi \). To keep the presentation brief, we present semantics and inference only for the propositional subset.

**Inference:** Given a set of formulae \( \Gamma \), we can syntactically manipulate the formulae to derive new formulae without considering truth value assignments. Such manipulations are known as proofs or derivations. There are many such inference systems that let us check whether a formula \( \phi \) is derivable from \( \Gamma \) (such as natural deduction, tableau, resolution) [3]. We focus on resolution which is commonly used in many state-of-the-art automated reasoning systems.

In resolution, we convert a set of formulae into a canonical form, the conjunctive normal form. Every formula is represented as set of conjunctions \( (c_1 \land c_2 \land \ldots) \) and every conjunct \( c_i \) is a disjunction of literals \( (c_1 \equiv d_1 \lor d_2 \lor \ldots) \). A literal \( d_i \) is either an atom \( Y \) or its negation \( \neg Y \).

There is one single rule in resolution (which amounts to a generalized form of modus ponens):

**Resolution Scheme**

\[
\begin{array}{c}
d_1 \lor d_2 \ldots q \ldots \lor d_n \\
e_1 \lor e_2 \ldots \neg q \ldots \lor e_m
\end{array}
\rightarrow

\begin{array}{c}
d_1 \lor d_2 \ldots \lor d_n \lor e_1 \lor d_2 \ldots \lor e_m
\end{array}
\]

The rule takes as input two clauses. If one of the clauses has an atom \( q \) that is negated in the other clause \( (\neg q) \), the two clauses can be combined. The output clause is a new clause with all the literals in the two clauses except for \( q \) and \( \neg q \). We have a proof \( \Gamma \vdash \phi \) if there exists some sequence of resolution operations that produce \( \phi \) given \( \Gamma \). Inference is then just simply search over the space of clauses using the resolution scheme.
## B MNIST Constraints

Constraints for the MNIST Fashion experiment expressed in the TPTP format [28]. Atoms $y(i)$ correspond to class $i$ in the MNIST Fashion dataset. $a_1$ specifies that we need to have complete set of clothing. $a_2$ specifies a complete set of clothing has upper, lower and footwear. $a_3, a_4,$ and $a_5$ declare what goes into upper, lower and footwear categories. $a_6$ declares dresses and sneakers don’t go together. $atleast\_three\_objs$ declares that we have at least three objects.

```plaintext
tff(dec1, type, y: $int > $o).
tff(a1, axiom, complete).
tff(a2, axiom, upper & lower & footwear <=> complete).
tff(a3, axiom, $(y(0) \lor y(6) \lor y(3))$ <=> upper).
tff(a4, axiom, $(y(1) \lor y(3))$ <=> lower).
tff(a5, axiom, $(y(5) \lor y(7) \lor y(9))$ <=> footwear).
tff(a6, axiom, $(\neg y(3) \land y(7))$).
```

The listing below shows the generated Python code for the loss function derived from the above constraints.

```python
def constraint_loss_single(y_pred):
    def y(i):
        return y_pred[i]
    def _y(i):
        return 1 - y_pred[i]
    loss = (_y(5) * _y(7) * _y(9)) + \
      (_y(1) * _y(3)) + \
      (_y(0) * _y(1) * _y(2) * _y(4) * _y(6) * _y(8) * _y(9)) + \
      (_y(0) * _y(3) * _y(6)) + \
      (_y(0) * _y(1) * _y(2) * _y(4) * _y(5) * _y(6) * _y(8)) + \
      (y(3) * y(7))
    return loss
```
C Properties

Lemma 1 (Monotonicity 1). If \( \Gamma \subseteq \Gamma' \), then \( d_\Gamma(\eta) \leq d_{\Gamma'}(\eta) \)

Proof. Since \( \Gamma \subseteq \Gamma' \), we have \( \omega(\Gamma, \phi, \eta) \leq \omega(\Gamma', \phi, \eta) \).

Lemma 2 (Monotonicity 2). If \( \Gamma' \vdash \Gamma \), then \( d_{\Gamma'}(\eta) \geq d_\Gamma(\eta) \)

Proof. Since \( \Gamma' \vdash \Gamma \), for every \( \Gamma \cup Y \vdash \rho \phi \) we have a corresponding \( \Gamma' \cup Y \vdash \rho' \phi \) such that \( \text{premises}(\rho') \vdash \text{premises}(\rho) \). Using the fact that the proofs are minimal we can prove that \( \eta(\rho') \geq \eta(\rho) \), therefore \( \omega(\Gamma, \phi, \eta) \leq \omega(\Gamma', \phi, \eta) \).

Theorem 1 (Equivalence). If \( \bigwedge \Gamma' \iff \bigwedge \Gamma \), then \( d_{\Gamma'}(\eta) = d_\Gamma(\eta) \)

Proof. If \( \bigwedge \Gamma' \iff \bigwedge \Gamma \), then we have \( \Gamma' \vdash \Gamma \) and \( \Gamma \vdash \Gamma' \) and we can use the previous property.
D Models Used

CNN Model for MNIST Digits and Fashion-MNIST

<table>
<thead>
<tr>
<th>Layer (type)</th>
<th>Output Shape</th>
<th>Param #</th>
</tr>
</thead>
<tbody>
<tr>
<td>input_1 (InputLayer)</td>
<td>(None, 84, 84, 1)</td>
<td>0</td>
</tr>
<tr>
<td>conv2d_1 (Conv2D)</td>
<td>(None, 82, 82, 64)</td>
<td>640</td>
</tr>
<tr>
<td>max_pooling2d_1 (MaxPooling2D)</td>
<td>(None, 41, 41, 64)</td>
<td>0</td>
</tr>
<tr>
<td>conv2d_2 (Conv2D)</td>
<td>(None, 39, 39, 32)</td>
<td>18464</td>
</tr>
<tr>
<td>max_pooling2d_2 (MaxPooling2D)</td>
<td>(None, 19, 19, 32)</td>
<td>0</td>
</tr>
<tr>
<td>conv2d_3 (Conv2D)</td>
<td>(None, 17, 17, 16)</td>
<td>4624</td>
</tr>
<tr>
<td>max_pooling2d_3 (MaxPooling2D)</td>
<td>(None, 8, 8, 16)</td>
<td>0</td>
</tr>
<tr>
<td>dropout_1 (Dropout)</td>
<td>(None, 8, 8, 16)</td>
<td>0</td>
</tr>
<tr>
<td>flatten_1 (Flatten)</td>
<td>(None, 1024)</td>
<td>0</td>
</tr>
<tr>
<td>dense_1 (Dense)</td>
<td>(None, 256)</td>
<td>262400</td>
</tr>
<tr>
<td>dropout_2 (Dropout)</td>
<td>(None, 256)</td>
<td>0</td>
</tr>
<tr>
<td>dense_2 (Dense)</td>
<td>(None, 128)</td>
<td>32896</td>
</tr>
<tr>
<td>dropout_3 (Dropout)</td>
<td>(None, 128)</td>
<td>0</td>
</tr>
<tr>
<td>dense_3 (Dense)</td>
<td>(None, 64)</td>
<td>8256</td>
</tr>
<tr>
<td>dropout_4 (Dropout)</td>
<td>(None, 64)</td>
<td>0</td>
</tr>
<tr>
<td>dense_4 (Dense)</td>
<td>(None, 32)</td>
<td>2080</td>
</tr>
<tr>
<td>d1 (Dense)</td>
<td>(None, 10)</td>
<td>330</td>
</tr>
</tbody>
</table>

Total params: 329,690
Trainable params: 329,690
Non-trainable params: 0

Random Forest Classifier for Income Classification

<table>
<thead>
<tr>
<th>Param</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>bootstrap</td>
<td>True</td>
</tr>
<tr>
<td>class_weight</td>
<td>None</td>
</tr>
<tr>
<td>criterion</td>
<td>gini</td>
</tr>
<tr>
<td>max_depth</td>
<td>None</td>
</tr>
<tr>
<td>max_features</td>
<td>auto</td>
</tr>
<tr>
<td>max_leaf_nodes</td>
<td>None</td>
</tr>
<tr>
<td>min_impurity_decrease</td>
<td>0.0</td>
</tr>
<tr>
<td>min_impurity_split</td>
<td>None</td>
</tr>
<tr>
<td>min_samples_leaf</td>
<td>1</td>
</tr>
<tr>
<td>min_samples_split</td>
<td>2</td>
</tr>
<tr>
<td>min_weight_fraction_leaf</td>
<td>0.0</td>
</tr>
<tr>
<td>n_estimators</td>
<td>10</td>
</tr>
<tr>
<td>n_jobs</td>
<td>None</td>
</tr>
<tr>
<td>oob_score</td>
<td>False</td>
</tr>
<tr>
<td>random_state</td>
<td>None</td>
</tr>
<tr>
<td>verbose</td>
<td>0</td>
</tr>
<tr>
<td>warm_start</td>
<td>False</td>
</tr>
</tbody>
</table>
E  MNIST Fashion: Expanded Results

Figure 7: MNIST Fashion Multi-object Accuracies (low training data). Results are over 5 randomized runs. We use $c_1 = 1 - scaling\_factor, c_2 = scaling\_factor$. Darker colors show increasing training data. The horizontal axis shows the scaling factor and the vertical axis is accuracy. The top row shows networks trained with constraint loss functions and the bottom row shows output modification applied to the networks. Columns indicate fraction of missing labels. Probability of missing a label: 0 and 0.1.
Figure 8: MNIST Fashion Multi-object Accuracies (low training data). Results are over 5 randomized runs. We use $c_1 = 1 - \text{scaling factor}, c_2 = \text{scaling factor}$. Darker colors show increasing training data. The horizontal axis shows the scaling factor and the vertical axis is accuracy. The top row shows networks trained with constraint loss functions and the bottom row shows output modification applied to the networks. Columns indicate fraction of missing labels. Probability of missing a label: 0.1 and 0.3.
Figure 9: MNIST Fashion Multi-object Accuracies (low training data). Results are over 5 randomized runs. We use $c_1 = 1 - scaling\_factor$, $c_2 = scaling\_factor$. Darker colors show increasing training data. The horizontal axis shows the scaling factor and the vertical axis is accuracy. The top row shows networks trained with constraint loss functions and the bottom row shows output modification applied to the networks. Columns indicate fraction of missing labels. Probability of missing a label: 0.7 and 0.9