Pylon:
A PyTorch Framework for Learning with Constraints

Kareem Ahmed
University of California, Los Angeles

Tao Li
University of Utah

Thy Ton
University of California, Irvine

Quan Guo
Sichuan University

Kai-Wei Chang
University of California, Los Angeles

Parisa Kordjamshidi
Michigan State University

Vivek Srikumar
University of Utah

Guy Van den Broeck
University of California, Los Angeles

Sameer Singh
University of California, Irvine

Editors: Douwe Kiela, Marco Ciccone, Barbara Caputo

Abstract
Deep learning excels at learning low-level task information from large amounts of data, but struggles with learning high-level domain knowledge, which can often be directly and succinctly expressed. In this work, we introduce Pylon, a neuro-symbolic training framework that builds on PyTorch to augment procedurally trained neural networks with declaratively specified knowledge. Pylon allows users to programmatically specify constraints as PyTorch functions, and compiles them into a differentiable loss, thus training predictive models that fit the data whilst satisfying the specified constraints. Pylon includes both exact as well as approximate compilers to efficiently compute the loss, employing fuzzy logic, sampling methods, and circuits, ensuring scalability even to complex models and constraints. A guiding principle in designing Pylon has been the ease with which any existing deep learning codebase can be extended to learn from constraints using only a few lines: a function expressing the constraint and a single line of code to compile it into a loss. We include case studies from natural language processing, computer vision, logical games, and knowledge graphs, that can be interactively trained, and highlights Pylon’s usage.

Keywords: Neuro-symbolic Learning, Learning with Constraints, Deep Learning

Introduction

Deep learning models are able to learn even the most complex of tasks, provided enough data is available. However, they often struggle with learning high-level domain knowledge that can often be much more succinctly expressed declaratively, such as using programmatic constraints. Unfortunately, existing frameworks are not able to learn from such declarative knowledge, and instead attempt to learn it from available data, leading to overfitting to spurious patterns, learning functions that are unfaithful to rules of the underlying task.

Neuro-symbolic reasoning methods aim to straddle the line between deep learning and symbolic reasoning, combining high-level procedural knowledge with data, during learning. They aim to learn functions that fit the data while remaining faithful to the rules of the underlying domain, which empirically translates into performance improvements and more efficient learning. These systems are not without their challenges, however. Most frameworks make use of custom languages (Rajaby Faghihi et al.; Guo et al., 2020; Manhaeve et al., 2018; Stewart and Ermon, 2017) or logic (Bach et al., 2017; Diligenti et al., 2017; Fischer et al., 2019; Hu et al., 2016; Li and Srikumar, 2019; Nandwani et al., 2019; Rocktäschel et al., 2015; Xu et al., 2018; Zhang et al., 2016) to express such knowledge, making it unnatural, unwieldy, or even impossible to express many forms of knowledge. Furthermore, they often require porting to their own ecosystem, making them arduous to integrate with existing code. Finally, each such method presents with a unique set of trade-offs, and is effective on a limited set of domains and constraints, often unbeknownst to the user.

We introduce Pylon, a package built on top of PyTorch that offers practitioners the ability to seamlessly integrate declarative knowledge into deep learning models. The user expresses the knowledge as a Python function that defines the constraint in terms of PyTorch tensors. Pylon compiles the function into an efficient, differentiable loss that is compatible with PyTorch trainers, providing a unifying interface to existing neural-symbolic methods that integrate declarative knowledge into learning.

Pylon Overview

Example Consider the code snippet in Figure 1, where we consider the task of entity-relation extraction. That is, given a sentence $x$, the model on line 14 classifies each word into a corresponding entity (e.g. person, organization), and for every pair of entities whether they are related, and if so, the type of relation that holds between them (e.g. works for).

We wish to enforce two constraints that capture our domain knowledge on the learned model: 1) the subject of a lives in relation is always a person, and 2) that the majority of predicted entities are not person. The above constraints are

![Figure 1: Enforcing a constraint using Pylon](https://pylon-lib.github.io/)

---

1. Pylon website is available at https://pylon-lib.github.io/
expressed as the PyTorch function `check_livesin_subj` defined in Figure 1 on lines 2-4 and the lambda function on line 10, respectively.

The challenge then is, how to integrate these discrete, Boolean functions with differentiable learning. We will show how this can be achieved by interpreting the model outputs as inducing a distribution over the output space, and reducing our problem to one of probabilistic reasoning: we wish to find the set of parameters that maximize the probability of satisfying the user-defined constraints under the network’s probability distribution.

**A Probability Distribution over Structured Outputs**

Let $\theta$ be the parameters of the neural network model defined over a set of variables $Y = \{Y_1, \ldots, Y_n\}$, where each $Y_i$ denotes a target. Let $p$ be a vector of probabilities for the variables $Y$, where $p_i$ denotes the predicted probability of variable $Y_i$ and corresponds to a single output of the network. The network’s outputs induce a distribution $p_\theta(\cdot)$ over all possible instantiations $y$ of $Y$

$$p_\theta(y|x) = \prod_{i:y=Y_i} p_i \prod_{i:y=\neg Y_i} (1 - p_i)$$ (1)

where $y \models Y_i$ and $y \models \neg Y_i$ denote that $Y_i$ is true or false in the instantiation $y$, respectively. Although we use boolean valued variables for notational simplicity, the ideas directly extend to other discrete valued variables as well.

**Training objective**

Having defined a distribution over all possible outputs, we now consider the problem of learning with constraints through a probabilistic lens: the problem of integrating our declaratively-defined functions into the learning process reduces to optimizing for the set of network parameters such that the probability allocated by the network to satisfying the constraints is maximized. Formally, we would like to minimize the following:

$$\arg\min_\theta \mathcal{L}(\theta|C, x) = \arg\min_\theta -\log \mathbb{E}_{y \sim p_\theta(\cdot|x)} [1\{C(y)\}]$$ (2)

where, for a given constraint $C$, we penalize the network with a loss that is proportional to the extent to which the network’s beliefs violate the constraint, as measured by the probability mass allocated by the network to all instantiations violating the constraint $C$.

Calculating the above expectation naively requires enumerating all instantiations $y$ in a brute force manner, of which there are exponentially many, and is feasible only for the simplest of constraints. For example, for a model defined over the edges in a $n \times n$ grid (i.e. $2n^2 - 2n$ boolean variables), an instantiation is an assignment to each of the variables, of which there are $2^{2n^2-2n}$ many possibilities.

**Constraint functions**

We encode the aforementioned declarative knowledge by means of constraint functions. A constraint function is a Python function that accepts any number of tensor arguments, each of shape (batch size,...) and returns a Boolean tensor of shape (batch size,). Each argument corresponds to a (batched) decoding from a model. A decoding is an assignment to all variables of a model, each variable sampled with a probability corresponding to its likelihood under the model’s posterior. For example, in our entity-relation extraction example, a decoding of relation logits (or entity logits) constitutes a relation (or entity, resp.) assigned to each word in the sentence.

A constraint function defines a predicate $C$ on the decodings of any number of models, and returns whether or not the given decodings satisfy the constraint. For instance, lines
2-4 define a constraint function over the decodings of the entity and relations classifiers that captures that the subject of a lives in relation is always a person. Line 10 defines a lambda constraint function over the decoding of the entity classifier, and encodes that the majority of entities are not person. While the first constraint can be easily expressed in logic, the same does not hold true for the second constraint: we would need to conjoin all decodings satisfying the constraint, which would scale exponentially with the length of the sentence — unless we resort to introducing auxiliary variables. Using Python/PyTorch, we can capture the constraint succinctly.

**Exploiting Structure of Constraint Definition**  Although the user can use all of PyTorch/Python syntax to write the constraint, we parse the constraint function to see if it expresses known structures, such as logic. When the constraints exhibit structural properties that allow us to reuse intermediate computations, we can sidestep the intractability of Eq (2) by compiling them into logical circuits (Xu et al., 2018). This does not, in general, escape the complexity of Eq (2) as the circuit grows exponentially in the constraint size. In these cases, we can utilize approximations based on fuzzy logic, computing differentiable probabilities of logical statements without grounding them, such as using product (Rocktäschel et al., 2015), or Lukasiewicz (Bach et al., 2017; Kimmig et al., 2012) T-norms.

**Black-box Optimization**  Alternatively, we can also approximate the loss in Eq (2) by sampling decodings from the model posterior. We can use the REINFORCE gradient estimator (Glynn, 1990; Williams, 1992) to rewrite the gradient of the expectation in Eq (2) as the expectation of the gradient, which can be estimated using Monte Carlo sampling

$$
\nabla_\theta \mathbb{E}_{y \sim p_\theta(\cdot|x)}[1\{C(y)\}] = \mathbb{E}_{y \sim p_\theta(\cdot|x)}[\nabla_\theta 1\{C(y)\} \log p_\theta(y|x)]
$$

(3)

This not only enables us to estimate the probability of otherwise-intractable constraints but also enables greater flexibility in defining our constraint functions: we can issue calls to non-differentiable resources and continue to yield a differentiable loss, hence black-box.

**Pylon** uses implementations of these approaches that are directly compatible with PyTorch, as seen in lines 16 and 17, including ones that utilize the structure in the user-defined code for efficiency (T-norm and circuit-based losses) and ones that work for any implementation (brute-force and sampling), and is easily extensible to other techniques.

**Case Studies**

We include four case studies that vary in their domain, and exhibit the versatility of Pylon, spanning vision, NLP and logical games (e.g. Li et al., 2019; Punyakanok et al., 2008): 

- **MNIST Addition**: Presented with two MNIST digits, we require the model’s predictions add to their summation. Model learns to predict single digits at test time.

- **NLI Transitivity**: The model is presented with sentence triples and predicts how each pair is related. The model’s predictions are constrained to satisfy transitivity.

- **SRL Unique Role**: For each predicate in the semantic role labeling task, the model’s predictions are constrained such that each core argument span appears at most once.

- **Sudoku**: Given a Sudoku, the model is trained to predict the missing entries in the puzzle such that the elements in each individual row, column and square are unique.
Acknowledgements

We would like to thank Sebastian Riedel and Sanjay Subramanian for important suggestions and feedback for the project. This work is supported in part by NSF grants #IIS-2046873, #CCF-1837129, #IIS-1956441, #IIS-1943641, #CNS-1801446, #IIS-1822877, NSF Career award #2028626, Office of Naval Research grant #N00014-20-1-2005, a Sloan Fellowship, and a gift from Allen Institute for AI.

References


