CLARE: a General Interface Layer to Integrate AI and Deep Learning

Giuseppe Marra[1,2], Francesco Giannini[2], Michelangelo Diligenti[2], Marco Gori[2]

[1] Department of Information Engineering
University of Florence, Via di S. Marta 3, Firenze, Italy
g.marra@unifi.it
[2] Department of Information Engineering and Mathematics
University of Siena, Via Roma 56, Siena, Italy
{fgiannini,diligmic,marco}@diism.unisi.it
http://sailab.diism.unisi.it/

Abstract. In spite of the amazing results obtained by deep learning in many applications, a real intelligent behavior of an agent acting in a complex environment is likely to require some kind of higher-level symbolic inference. Therefore, there is a clear need for the definition of a general and tight integration between low-level tasks, processing sensorial data that can be effectively elaborated using deep learning techniques, and the logic reasoning that allows humans to take decisions in complex environments. This paper presents CLARE (Constrained Logic and Reasoning Environment), a generic interface layer for AI, which is implemented in TersorFlow (TF). CLARE provides an input language that allows to define arbitrary First Order Logic (FOL) background knowledge, including clauses, groundings and constants. The predicates and functions can be bound to any TF computational graph, while the formulas are converted by the framework into a set of real-valued constraints, which participate to the overall optimization problem. This allows to learn the weights of the learners, under the constraints imposed by the prior knowledge. The framework is extremely general as it imposes no restrictions in terms of which models or knowledge can be integrated. In this paper, we show the generality of the approach showing some use cases of the presented language, including generative models, logic reasoning, model checking and supervised learning.

Keywords: deep learning, prior knowledge injection, first order logic

1 Introduction

The success of deep learning relies on the availability of a large amount of supervised training data. This prevents a wider application of machine learning in real world applications, where the collection of training data is often a slow and expensive process, requiring an extensive human intervention. On the other hand, this raises concerns on what learning really means: how much deep learning simply relies on remembering previously seen patterns? How robust is such a

system in an adversarial or uncontrolled environment, where outliers are either unpredictable or even forged to fool the trained system?

The introduction of prior-knowledge into the learning process is a fundamental step in overtaking these limitations. First, it does not require the training process to induce the rules from the training set, therefore reducing the number of required training data. Secondly, the use of prior knowledge can be used to express the desired behavior of the learner on any input, providing better behavior guarantees in an adversarial or uncontrolled environment.

This paper presents CLARE (Constrained Logic and Reasoning Environment), a TensorFlow [1] environment based on a declarative language for integrating prior knowledge into machine learning. CLARE allows the full expressiveness of First Order Logic (FOL) to express the knowledge and it generalizes Semantic Based Regularization [5,6] to any set of learners, whereas Semantic Based Regularization was originally formulated to inject prior-knowledge into Kernel Machines. The presented declarative language provides a uniform platform to face both learning and inference tasks by requiring the satisfaction of a set of rules on the domain of discourse. CLARE can define any many-sorted first-order logical theory, allowing to declare domains of different sort, with constants, predicates and functions. CLARE provides a very tight integration of learning and logic as any computational graph can be bound to a FOL predicate. This allows to constrain the learner both during training and inference. Since the framework is agnostic to the learner that are bound to the predicates, the framework can be used in a vast range of applications including classification, generative or adversarial ML, sequence to sequence learning, collective classification, etc. We provide some examples of application in the next sections.

In the past few years many authors tackled specific applications by integrating logic and learning. Minervini et al. [12] proposes to use prior knowledge to correct the inconsistencies of an adversarial learner. Their methodology is designed ah-hoc for the tackled task, and limited to Horn clauses. An iterative rule knowledge distillation is presented in Hu et al. [9], which is also based on a fuzzy generalization of FOL. However, the definition of the framework is limited to universally quantified formulas and to a small set of logic operators with no discussion about the different selections of t-norms.

A more general attempt at creating a general formalism to integrate logic and learning is the work on Logic Tensor Networks [18], which is also based on a continuous generalization of First Order Logic. The framework however does not define a general language that can be easily reused in any context and it is limited to universally quantifies FOL clause with a specific form. Another line of research [16,4] attempts at using logical background knowledge to improve the embeddings for Relation Extraction. However, all these works are based on ad-hoc solutions that lack a common declarative mechanism that can be easily reused across all applications. They are all limited to a subset of FOL and they allow to injecting the knowledge at training time, with no guarantees that the output on the test set respect the knowledge. On the other hand, CLARE is a

general tool that allows to inject any background knowledge written in FOL logic with functions, both at training and test time.

The integration of logic and learning has been attempted many times in the past few years. For example, Markov Logic Networks (MLN) [15] and Probabilistic Soft Logic (PSL) [10,2] are two frameworks that attempts at providing a generic AI interface layer for machine learning by implementing a probabilistic logic, whose parameters must be trained to determine the strength of the available knowledge in a given universe. MLN and PSL with their corresponding implementations have received lots of attention but they are limited by their shallow integration with the underlying learning processes working on the low-level sensorial data. In MLN and PSL, a low-level learner is trained independently, then frozen and stacked with the AI layer providing a higher-level inference mechanism. The language proposed in this paper instead allows to directly improve the underlying learner, while also providing the higher-level integration with logic. TensorLog [3] is a more recent framework to integrate probabilistic logical reasoning with the deep-learning infrastructure of TF, however TensorLog is limited to reasoning and does not allow to optimize the learners while performing inference. TensorFlow Distributions [7] and Edward [19] are also a related frameworks for integrating probability theory and deep learning, maintaining computational efficiency and numerical stability. However, these frameworks focus on probability theory and not the representation of logic and reasoning.

The paper is organized as follows. In Section 2 we introduce the framework, describe its declarative nature and delineate how first–order logic (FOL) formulas can be converted into a learning model. Section 3 shows the generality of the framework by showing a wide range of toy examples and applications. Finally, some conclusions are drawn in Section 4.

2 The CLARE framework

This Section presents how CLARE defines an TensorFlow environment in which learning and reasoning can take place at the same time. The definition of the knowledge in the CLARE environment starts by defining a certain number of domains in the considered world. A *domain* determines a collection of individuals of the world that share the same representation space and, thus, can be analyzed and manipulated in a homogeneous way. For example, a domain can collect the set of considered 30×30 pixel images, or the sentences of a book as bag-of-words, or all the points of the plane in their (x, y) form. The domains are then filled with their "inhabitants", on which the learning and reasoning will be carried.

For example, a domain called *Points* can be defined as:

```
Domain(label="Points", data=data)
```

where *data* is the placeholder of the input data. The elements of a domain are a sort of "anonymous" individuals that are collectively processed. On the other hand, an *individual* of a domain can also be separately specified, and a specific behavior can be defined for it. In particular, specific individuals can be added to a domain using the following construct:

```
Individual(label="p0", domain=("Points"), value=p0)
```

A CLARE *function* can be defined to map elements from the input domains into an element of an output domain. In particular, a unary function takes as input an element from a domain and transforms it into an element of the same or of another domain, while an n-ary function takes as input n elements, mapping them into an output element of its output domain. For example, it is possible to define arithmetic functions to operate over number domains, encoding functions to transform elements of a domain into a latent space, etc. The following example defines a CLARE "encoder" function:

```
Function(label="encoder",domains=("Images"),function=CNNEncoder)
```

where the FOL function is bound to its TF implementation, which in this case is the *CNNEncoder* function

CLARE allows also to define a set of *predicates*, which are functions mapping elements of the input domains to truth values, as for example: *isCat(x)*, or $f(x) > 3$. For example, a predicate A approximated by a neural network NN, taking as input the patterns in the domain *Points* can be defined as:

```
Predicate("A", domains=("Points"), function=NN)
```

Finally, it is possible to state the knowledge about the world by means of a set of *constraints*. Each constraint is a generic FOL formula using as atoms the previously defined functions and predicates. For instance, if we are given the domain *Animals* and two predicates *bird* and *flies* defined on it, the user can express the knowledge that all the birds fly by means of the constraint:

```
Constraint("forall x: bird(x) -> flies(x))
```

2.1 CLARE implementation

This Section shows how this high level description can be translated into an effective learning model that can be optimized to learn functions and predicates satisfying the given constraints.

CLARE is implemented using TensorFlow[3], an open source software library for high performance numerical computation. TF performs computations by building a computational graph, where nodes of the graph are operations which manipulate all the tensors represented by their incoming edges. One of the main features of TF is its capability of performing automatic differentiation of a generic function represented by a computational graph by the exploitation of the chain rule of calculus. Moreover, TF provides a wide family of gradient descent algorithms to optimize computational graphs which contain some variable tensors. CLARE exploits TensorFlow for implementing all components, thus any TensorFlow model can be integrated in CLARE as well.

The main feature of CLARE is the capability of translating a high level description of the knowledge into a big TF computational graph by composing

[3] `https://www.tensorflow.org/`

each piece of knowledge by means of the logical constraints. In the end, the overall graph is simultaneously optimized exploiting opportune TF procedures. In the following, we describe how this translation is performed.

Domains and Individuals. Domains and individuals allow users to provide data to the framework as tensors and represent the leaves of the computational graph. While domains are represented by constant tensors, individuals can be represented by both *constant* and *variable* tensors. In this way, the user is allowed to provide the knowledge of the existence of a certain individual, even if its feature representation is unknown, and its representation will be optimized in the overall process to be coherent with the other pieces of knowledge provided.

Functions and Predicates. Functions allow the mapping among different tensors of (possibly) different domains, while Predicates allow to express the truth degree of some property for those tensors. In CLARE, both functions and predicates can be implemented using any TF computational graph. If the graph does not contain any variable tensor (i.e. it is not parametric), then we say it to be *given*; otherwise all the variables will be automatically learned to maximize the constraints satisfaction and we say the graph to be *learnable*. Given functions are usually arithmetic functions (e.g. $+, =, \geq$), similarity or comparison functions, etc. Learnable functions are usually complex models such as (deep) neural networks, kernel machines, radial basis functions, whose weights must be learned.

Constraints. The main contribution of this framework is the capability of integrating learning models and logical reasoning. This is achieved by exploiting the conversion of logical rules into continuous real-valued constraints which can be optimized using gradient descent techniques on computational graphs. By means of the constraints, different elements in the environment correlate to enforce a desired behavior as stated by the formula.

Variables, functions, predicates and logical connectives can all be seen as nodes of an *expression tree* [11]. The evaluation of a constraint corresponds to a post-fix visit of the expression tree, where the visit action builds the correspondent portion of computational graph and, thus, varies from node to node. In particular:

- Visiting a *variable* corresponds to substituting the variable with the tensor of the domain it belongs to.
- Visiting a *function* or a *predicate* corresponds to providing input tensors to the TF models (either given or learnable) implementing those functions.
- Visiting the *connectives* corresponds to combining predicates by means of the real-valued operations associated to such connectives. This is obtained using the fuzzy generalization that was first proposed by Novak [14] and described in Appendix. Table 1 shows some possible implementation of the connectives using different *t-norms*.

As an example, we show in Figure 1 how to translate a simple logic formula into its expression tree and successively into a TensorFlow computational graph.

Table 1: The operations performed by the single units of an expression tree depending on the left and right inputs x, y and the t-norm used.

op \ tnorm	**Product**	**Minimum**	**Lukasiewicz**
$x \wedge y$	$x \cdot y$	$\min(x, y)$	$\max(0, 1 - x - y)$
$x \vee y$	$x + y - x \cdot y$	$\max(x, y)$	$\min(1, x + y)$
$\neg x$	$1 - x$	$1 - x$	$1 - x$
$x \Rightarrow y$	$\min(1, \frac{y}{x})$	$x < y ? 1 : y$	$x < y ? 1 : y - x$

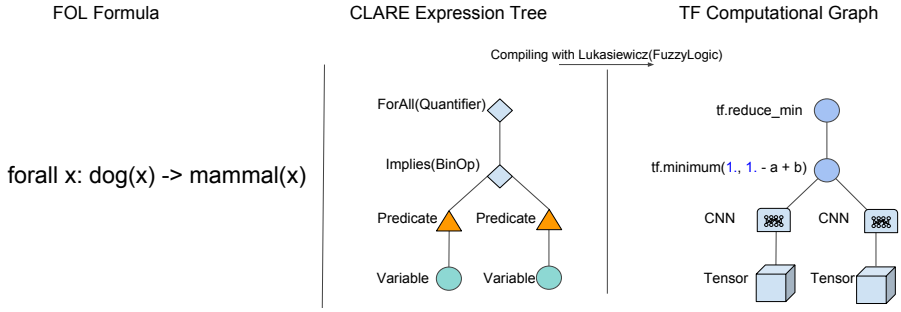


Fig. 1: CLARE translation of a FOL formula into a TF computational graph

Any available supervision for the learnable functions or predicates can be integrated into the learning problem. Whereas the fitting of the supervised data is very specific to the task at hand, CLARE provides a generic placeholder where this fitting is expressed, that is a construct called *PointwiseConstraint*. This construct defaults to a computational graph where an opportune loss for any specific TF model is exploited (e.g. cross-entropy for a neural network classifier).

```
PointwiseConstraint(model, labels, inputs)
```

where `model` is the function on which to enforce supervisions `labels` on data `inputs`.

Cost Function. Let us assume to be given a knowledge base KB, consisting of a set of T FOL formulas, where some of the elements (individuals, functions or predicates) are unknown. The CLARE learning process aims at finding a good approximation of each unknown element, so that the estimated values will satisfy the FOL formulas for the input samples.

For any single constraint a computational graph corresponding to its evaluation is built. Let $\boldsymbol{f}$ and $\boldsymbol{p}$ be the overall vectors of learnable functions and predicates respectively. Let $\Phi_j(\mathcal{X}_j, \boldsymbol{f}, \boldsymbol{p})$ denote the function returning the degree of satisfaction of the j-th constraint evaluated on all its inputs $\mathcal{X}_j$. It is implemented by the computational graph of the corresponding formula and its unknown components are optimized by gradient descent.

In order to satisfy all the logical constraints, we minimize the following term,

$$L_c(\mathcal{X}, \boldsymbol{f}, \boldsymbol{p}) = \sum_{j=1}^{T} \lambda_j (1 - \Phi_j(\mathcal{X}_j, \boldsymbol{f}, \boldsymbol{p})) \ ,$$

where $\mathcal{X}$ denotes the overall samples where the functions and predicates are evaluated and λ_j is a weight for the j-th logical constraint. However, CLARE allows to integrate classical supervised learning (by means of PointwiseConstraints) and learning from constraints modeling the prior knowledge. Therefore in general, the cost function that is optimized by CLARE is composed by three terms, one forcing the fitting of the supervised examples, one for regularization and the latter for the logical constraint satisfaction. We have:

$$\lambda_s \Big[L_s(\boldsymbol{f}, \mathcal{X}_s^f) + L_s(\boldsymbol{p}, \mathcal{X}_s^p) \Big] + \lambda_r \Big[||\boldsymbol{f}|| + ||\boldsymbol{p}|| \Big] + \lambda_c L_c(L_c(\mathcal{X}, \boldsymbol{f}, \boldsymbol{p}) \ ,$$

where $\mathcal{X}_s^f, \mathcal{X}_s^p$ denote the supervised data for functions and predicates respectively, $||\boldsymbol{f}||$ and $||\boldsymbol{p}||$ measure the complexity of the approximators, L_s is a loss function and λ_r, λ_c and λ_s indicate the weights in the optimization of the supervision, regularization and constraint terms, respectively.

3 Learning and Reasoning with CLARE

This section presents a list of examples illustrating the range of learning tasks that can be expressed in the proposed framework. In particular, it is shown how it is possible to force label coherence in semi-supervised or transductive learning tasks, how to implement collective classification over the test set, rule deduction from the learned predicates as in classical Inductive logic programming (ILP), pure logical reasoning and how to address generative tasks or pattern completion in the case of missing features. The software of both the framework and the experiments is made available at `https://github.com/GiuseppeMarra/CLAREecml`

Semi-Supervised Learning. In this task we assume to have available a set of 420 points distributed along an outer and inner circles. The inner and outer points belong and do not belong to some given class A, respectively. A random selection of 20 points is supervised (either positively or negatively), as shown in Figure 2(a). The remaining points are split into 200 unsupervised training point, shown in Figure 2(b) and 200 points left as test set. A neural network is assumed to have been created in TF to approximate the predicate A.

The network can be trained by making it fit the supervised data. So, given the vector of data `X`, a neural network `NN_A` and the vector of supervised data `X_s`, with the vector of associated labels `y_s`, the supervised training of the network can be expressed in CLARE by the following code:

```
# Definition of the domain of the data points.
Domain(label="Points", data=X)
# Approximating the predicate A via a NN.
```

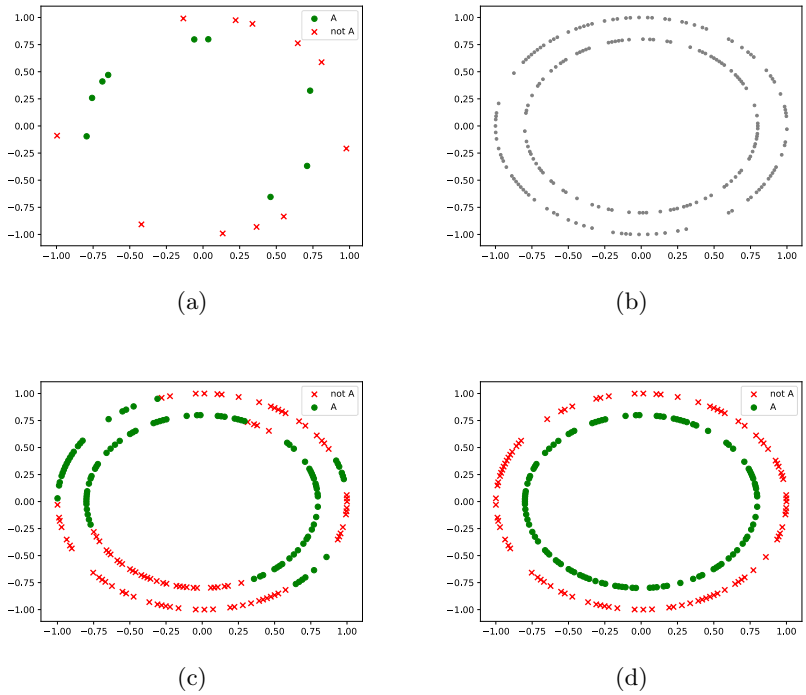


Fig. 2: Semi-Supervised Learning: (a) data that is provided with a positive and negative supervision for class A; (b) the unsupervised data provided to the learner; (c) class assignments using only the supervised examples; (d) class assignments using learning from examples and constraints in CLARE.

```
Predicate("A", ("Points"), function=NN_A)
# Fit the supervisions
PointwiseConstraint(A, y_s, X_s)
```

Let's now assume that we want to express manifold regularization for the learned function: this states that points that are close should be similarly classified. This extension can be expressed in CLARE adding the following lines:

```
# Given predicate stating whether two patterns are "close"
Predicate("Close", ("Points","Points"), function=f_close)
# The constraint implementing manifold regularization.
Constraint("forall p:forall q: Close(p,q)->(A(p)<->A(q))")
```

where `f_close` is a given function determining if two patterns are close. The training is then re-executed starting from the same initial conditions as in the supervised-only case.

Figure 2(c) shows the class assignments of the patterns in the test set, when using only classical learning from supervised examples. Finally, Figure 2(d) presents the assignments when learning from examples and constraints.

Collective Classification. In this example we show how to implement collective classification [17] in CLARE. We assume that in a supervised learning stage the supervised examples are used to train a classifier. In particular, we assume that the patterns are represented as $\mathbb{R}^2$ datapoints. The classification task is a multi-label problem where the patterns belongs to three classes A, B, C. In particular, the class assignments are defined by the following membership regions:

$$\mathbf{A} = [-2, 1] \times [-2, 2], \quad \mathbf{B} = [-1, 2] \times [-2, 2], \quad \mathbf{C} = [-1, 1] \times [-2, 2]$$

These regions correspond to three overlapping rectangles as shown in Figure 3(a). The examples are partially labeled and drawn from a uniform distribution on both the positive and negative regions for all the classes.

In a first stage, the classifiers for the three classes are learned by a two-layer neural network taking four positive and four negative examples for each class. This can be done using the following CLARE declaration:

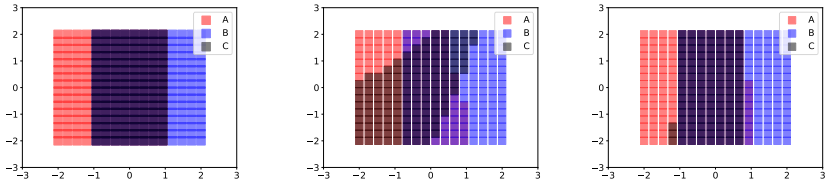
```
Domain(label="Points", data=X)
Predicate(label="A", domains=("Points"), function=NN_A)
Predicate(label="B", domains=("Points"), function=NN_B)
Predicate(label="C", domains=("Points"), function=NN_C)
PointwiseConstraint(NN_A, y_A, X_A)
PointwiseConstraint(NN_B, y_B, X_B)
PointwiseConstraint(NN_C, y_C, X_C)
```

The evaluation is carried out on a sample of 256 test points and the assignments performed by the classifiers after the supervised step are reported in Figure 3(b).

In a second stage, it is assumed that it is available some prior knowledge about the task at hand. In particular, any pattern must belongs to (at least) one of the classes A or B. Furthermore, it is known that class C is defined as the intersection of A and B. The classifiers trained in the first stage provide some initial predictions for each pattern. The collective classification step is performed by seeking the class assignments that are close to the initial classifier predictions but also respect the logical constraints on the test set. This is expressed n CLARE as:

```
# Prior knowledge expressed by logical constraints
Constraint("forall x: A(x) or B(x)")
Constraint("forall x: (A(x) and B(x)) <-> C(x)")
# Minimize the distance from the prior values
PointwiseConstraint(A, priorsA, X_test)
PointwiseConstraint(B, priorsB, X_test)
PointwiseConstraint(C, priorsC, X_test)
```

where `priorsA`, `priorsB`, `priorsC` are the outputs of the classifiers trained in the previous step for A, B, C, respectively, and `X_test` the set of test datapoints. As we can see from Fig.3(c), the collective step fixes some wrong predictions for the predicates.



(a) (b) (c)

Fig. 3: Collective classification: (a) classes assignments; (b) the predictions after the supervised step; (c) the predictions with collective classification and rules satisfaction (best viewed in colors).

Rule deduction and model checking. In this example, we show how CLARE can be used to deduct rules and perform model checking on a set of datapoints. In this section, a brute force approach was applied, but the Inductive Logic Programming [13] community has designed more scalable algorithms that could be exploited in this context. Let us Consider a simple multi-label classification task where the patterns belong to two classes A and B, where B is contained in A. This case can be seen as very simple hierarchical classification task. In particular, the classes are defined by the following membership regions:

$$\mathbf{A} = [-2, 2] \times [-2, 2], \quad \mathbf{B} = [-1, 1] \times [-1, 1].$$

Two neural network classifiers are trained to identify points drawn from a uniform distribution in the $[-3, 3] \times [-3, 3]$ square. In CLARE this correspond to:

```
Domain(label="Points", data=X)
Predicate(label="A", domains=("Points"), function=NN_A)
Predicate(label="B", domains=("Points"), function=NN_B)
PointwiseConstraint(NN_A, y_A, X)
PointwiseConstraint(NN_B, y_B, X)
```

It could be interesting to discover which logical relations are learned by the classifiers. In order to achieve this goal, we build all possible formulas in Disjunctive Normal Form (DNF) that are universally quantified with a single variable. One constraint is built and evaluated for each formula, and CLARE measures the degree of satisfaction of the rule over the data. In the considered task, one constraint get a very high degree of satisfaction:

```
Constraint("forall x: (not A(x) and not B(x)) or (A(x) and not B(x)) or (
    A(x) and B(x))")
```

As one could expect, the only fully-satisfied constraint (translated from DNF to its minimal form) is indeed $\forall x B(x) \rightarrow A(x)$, that states the inclusion of B in A.

Logic Reasoning. While CLARE is mainly designed for integrating logic and deep learning, it is also possible to use it as a tool for pure logical reasoning. This

case is illustrated by the following example, where a few individuals are added to the domain *People* without any underlying data representation. This can be defined in the CLARE environment as:

```
Domain(label="People")
Individual(label="Marco", "People")
Individual(label="Giuseppe", "People")
Individual(label="Michelangelo", "People")
Individual(label="Francesco", "People")
Individual(label="Franco", "People")
Individual(label="Andrea", "People")
```

The individuals are assumed to be related via parental relations defined by the following predicates:

```
Predicate(label="fatherOf", ("People", "People"))
Predicate(label="grandFatherOf", ("People", "People"))
Predicate(label="eq", ("People", "People"), function=eq)
```

where the given binary predicate `eq` holds true iff the two input individuals are the same person.

Finally, some known relations are known between the individuals:

```
Constraint("fatherOf(Marco, Giuseppe)")
Constraint("fatherOf(Giuseppe, Michelangelo)")
Constraint("fatherOf(Giuseppe, Francesco)")
Constraint("fatherOf(Franco, Andrea)")
```

The prior knowledge provided for this task expresses some well-known semantics about parental constraints. For example, CLARE allows to express that nobody can be father or grandfather of himself as:

```
Constraint("forall x: not fatherOf(x,x)")
Constraint("forall x: not grandFatherOf(x,x)")
```

Another two rules state that fathership is an asymmetric relation, so that if you are father or grandfather of someone, he can not be your father or grandfather. Furthermore, someone can not be father and grandfather of someone at the same time, these are expressed in CLARE as:

```
Constraint("forall x: forall y: fatherOf(x,y) -> not fatherOf(y,x)")
Constraint("forall x: forall y: grandFatherOf(x,y)
     -> not grandFatherOf(y,x)")
Constraint("forall x: forall y: fatherOf(x,y) -> not grandFatherOf(x,y)")
Constraint("forall x: forall y: grandFatherOf(x,y) -> not fatherOf(x,y)")
```

Another rule expresses that the father of the father is a grandfather, and that one person has at most one father in the considered world:

```
Constraint("forall x: forall y: forall z: fatherOf(x,z) and fatherOf(z,y)
     -> grandFatherOf(x,y)")
Constraint("forall x: forall y: forall z: (fatherOf(x,y) and not eq(x,z))
     -> not fatherOf(z,y)")
```

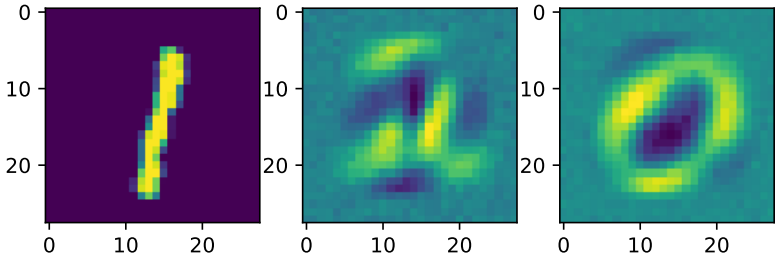


Fig. 4: An example of the trained generative functions, The middle and right pictures shown the outputs of the functions `next` and `previous` functions fed with the image on the left, respectively.

The learning task seeks to infer the unknown relations among the individuals. After starting the learning phase, CLARE outputs the predicate values for all the groundings, and it correctly concludes that the following facts hold true: `grandFatherOf("Marco", "Michelangelo")`, ¬`grandFatherOf("Marco", "Giuseppe")`, `grandFatherOf("Marco", "Francesco")`, etc. On the other hand nothing can be concluded regarding who is the grandfather of "Franco" and "Andrea", so leaving these values to be equal to their prior values.

One the training has been performed and the grounded predicates have been computed, CLARE provides an easy interface for performing model checking also in this symbolic environment. For example, let's suppose that we want to check whether the following rule holds true according to the computed assignments:

```
Constraint("forall x: forall y: forall z: grandFatherOf(x,z) and
    fatherOf(y,z) -> fatherOf(x,y)")
```

As expected, the evaluation of the rule performed by returns that the rules is perfectly verified by the computed assignments.

Pattern Generation. In this task, we exploit a set of around 15000 images of handwritten digits, obtained extracting only the 0, 1 and 2 digits from the MNIST dataset. We want to solve both a classification task, aiming at identifying which digit an image represents, and a generation task, learning some generative functions producing images from images. In particular, we want to learn two generative functions, `next` and `previous`, which, given an image of a digit, will produce an image of the next and previous digit, respectively. In order to give each digit a next and a previous digit in the chosen set, we used a circular mapping such that 0 is the next digit of 2 and 2 is the previous digit of 0.

This generative task can be solved in two steps: first, we learn the classifier in a purely supervised fashion, then the image generator is trained in a purely unsupervised fashion by simply exploiting the knowledge of the relations among classes and the inverse nature of the `next` and `previous` operators.

In particular, CLARE is used to define the domain of images and the binding of the predicates *one*, *two*, *three* to the three output neurons of the classifier (thanks to the *Slice* construct), which is trained as defined in the *PointwiseConstraint* function using a cross-entropy loss:

```
Domain("Images", data=X)
Predicate("zero",("Images"),function=Slice(NN, 0))
Predicate("one",("Images"),function=Slice(NN, 1))
Predicate("two",("Images"),function=Slice(NN, 2))
PointwiseConstraint(NN, y, X)
```

Once the `NN` function has been learned, the generative functions are trained as:

```
Predicate("eq",("Images", "Images"), function=eq)
Function("next",("Images"), function=NN_next)
Function("previous", ("Images"), function=NN_prev)

Constraint("forall x: zero(x) -> one(next(x))")
Constraint("forall x: one(x) -> two(next(x))")
Constraint("forall x: two(x) -> zero(next(x))")
Constraint("forall x: zero(x) -> two(previous(x))")
Constraint("forall x: one(x) -> zero(previous(x))")
Constraint("forall x: two(x) -> one(previous(x))")

Constraint("forall x: eq(previous(next(x)),x)")
Constraint("forall x: eq(next(previous(x)),x)")
```

where the function *eq* is implemented as the cosine-distance re-scaled into $[0, 1]$. The first six rules define the meaning of the *next* and *previous* mapping and the last two constraints, by enforcing the inverse property, implement a neural auto-encoder.

In Figure 4, it is shown an input image (left) and the output of the functions `next` (center) and `previous` (right).

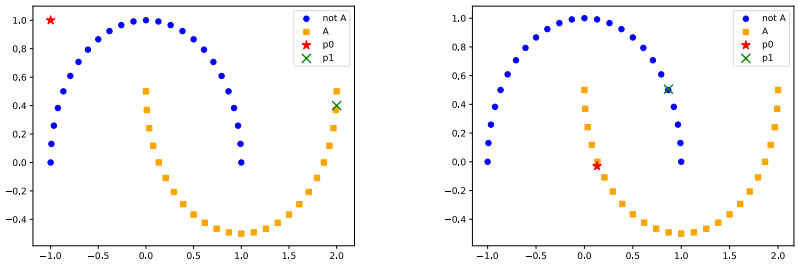
Missing Features. In this task we assume to have available a set of patterns drawn from a double moon shaped distribution as show in Figure 5(a). The patterns distributed along the lower moon belong to class A, while patterns along the lower one do not belong to the class.

This simple supervised learning task can be expressed in CLARE as:

```
Domain(label="Points", data=X)
Predicate("A", "Points", function=NN_A)
PointwiseConstraint(NN_A, y_A, X)
```

Let us now assume that there are two new individuals $p0$ and $p1$ for which no feature representation is available, but it is known that $p0$ and $p1$ belong and not belong to class A, respectively. This can be expressed in CLARE as:

```
p0 = Individual(label="p0", ("Points"))
p1 = Individual(label="p1", ("Points"))
```



(a) (b)

Fig. 5: Missing Features: (a) data that is provided with a positive and negative supervision for class A and initial random positions of points p_0 and p_1 (b) final position of the points p_0 and p_1 after constraints' satisfaction

```
Constraint("A(p0)")
Constraint("not A(p1)")
```

We assume to know in advance that the first and second individuals are close to a positive and negative example for class A, respectively. This can be stated in CLARE as:

```
Predicate("Close", ("Points", "Points"), function=f_close)
Predicate("eq", ("Points", "Points"), function=eq)
Constraint("exists q: not eq(q,p0) and A(q) and Close(q,p0)")
Constraint("exists q: not eq(q,p1) and not A(q) and Close(q,p1)")
```

where `f_close` is a given predicate deciding whether two points are close and `eq` implements a differentiable equality function defined by $1 - tanh(||x - y||^2 - \epsilon)$, where ϵ is a threshold.

Since the feature representations of $p0$ and $p1$ are not defined and are left as free variables, CLARE will learn them in order to respect the constraints defined by the logic rules. Figure 5(b) shows the feature values for the individuals, and CLARE has correctly placed $p0$ and $p1$, where the data distribution of the corresponding classes of the points is not null.

4 Conclusions

This paper presents a novel and general framework, called CLARE, to bridge logic reasoning and deep learning. The framework is directly implemented in TensorFlow, allowing a seamingless integration that is architecture agnostic. The CLARE frontend is a declarative language based on the full expressivity of First Order Logic (FOL). This paper presents a set of simple examples illustrating the generality and expressivity of the framework, which can be applied to a large

range of tasks, including classification, pattern generation, symbolic reasoning and rule induction.

Appendix

A *T-norm fuzzy logic* [8] generalizes Boolean logic to variables assuming values in $[0, 1]$. A T-norm fuzzy logic is defined by its T-norm $t(a_1, a_2)$ that models the logical AND. Several T-norm fuzzy logics have been proposed like the Łukasiewicz T-norm defined as $(\bar{a}_1 \wedge \bar{a}_2) \quad \rightarrow \quad t(a_1, a_2) = \max(0, a_1 + a_2 - 1)$. , where $\bar{a}_1, \bar{a}_2$ indicate two boolean values and their continuous generalizations a_1, a_2 in $[0, 1]$, respectively. The negation $\neg\bar{a}$ of a variable corresponds to $1 - a$ in the Łukasiewicz T-norm. From the definition of the $\wedge$ and $\neg$ logic operators, the generalized formulation for the $\vee$ operator can be derived via the DeMorgan rule and the implication $\Rightarrow$ operator via the T-norm residuum. Another T-norm that is commonly employed is the *product T-norm* defined as $(\bar{a}_1 \wedge \bar{a}_2) \rightarrow t(a_1, a_2) = a_1 \cdot a_2$ and the *minimum T-norm* defined as $(\bar{a}_1 \wedge \bar{a}_2) \rightarrow t(a_1, a_2) = \min(a_1, a_2)$.

The quantifier-free part of the expression is an assertion in fuzzy propositional logic once all the quantified variables are grounded. Let's consider a FOL formula with variables $x_1, x_2, \ldots$, and let $\mathcal{P}$ indicate the vector of predicates and $\mathcal{P}(\mathcal{X})$ be the set of all grounded predicates.

The degree of truth of a formula containing an expression E with a universally quantified variable x_i is the average of the T-norm generalization $t_E(\cdot)$, when grounding x_i over $\mathcal{X}_i$:

$$\forall x_i \; E\big(\mathcal{P}(\mathcal{X})\big) \longrightarrow \Phi_\forall(\mathcal{P}\big(\mathcal{X})\big) = \frac{1}{|\mathcal{X}_i|} \sum_{x_i \in \mathcal{X}_i} t_E\big(\mathcal{P}(\mathcal{X})\big)$$

The truth degree of the existential quantifier is instead defined as the *maximum* of the T-norm expression over the domain of the quantified variable:

$$\exists x_i \; E\big(\mathcal{P}(\mathcal{X})\big) \quad \longrightarrow \quad \Phi_\exists\big(\mathcal{P}(\mathcal{X})\big) = \max_{x_i \in \mathcal{X}_i} t_E\big(\mathcal{P}(\mathcal{X})\big)$$

When multiple universally or existentially quantified variables are present, the conversion is recursively performed from the outer to the inner variables. Please note that the fuzzy formula expression is continuous and differentiable with respect to the fuzzy value of a predicate, and it can therefore easily be integrated into learning.

References

1. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al.: Tensorflow: A system for large-scale machine learning. In: OSDI. vol. 16, pp. 265–283 (2016)
2. Bach, S.H., Broecheler, M., Huang, B., Getoor, L.: Hinge-loss markov random fields and probabilistic soft logic. arXiv preprint arXiv:1505.04406 (2015)

3. Cohen, W.W.: Tensorlog: A differentiable deductive database. arXiv preprint arXiv:1605.06523 (2016)
4. Demeester, T., Rocktäschel, T., Riedel, S.: Lifted rule injection for relation embeddings. arXiv preprint arXiv:1606.08359 (2016)
5. Diligenti, M., Gori, M., Maggini, M., Rigutini, L.: Bridging logic and kernel machines. Machine learning **86**(1), 57–88 (2012)
6. Diligenti, M., Gori, M., Saccà, C.: Semantic-based regularization for learning and inference. Artificial Intelligence (2015)
7. Dillon, J.V., Langmore, I., Tran, D., Brevdo, E., Vasudevan, S., Moore, D., Patton, B., Alemi, A., Hoffman, M., Saurous, R.A.: Tensorflow distributions. arXiv preprint arXiv:1711.10604 (2017)
8. Hájek, P.: Metamathematics of fuzzy logic, vol. 4. Springer Science & Business Media (1998)
9. Hu, Z., Ma, X., Liu, Z., Hovy, E., Xing, E.: Harnessing deep neural networks with logic rules. arXiv preprint arXiv:1603.06318 (2016)
10. Kimmig, A., Bach, S., Broecheler, M., Huang, B., Getoor, L.: A short introduction to probabilistic soft logic. In: Proceedings of the NIPS Workshop on Probabilistic Programming: Foundations and Applications. pp. 1–4 (2012)
11. Michelangelo Diligenti, S.R., Gori, M.: Image classification using deep learning and prior knowledge. In: Proceedings of Third International Workshop on Declarative Learning Based Programming (DeLBP) (February 2018)
12. Minervini, P., Demeester, T., Rocktäschel, T., Riedel, S.: Adversarial sets for regularising neural link predictors. arXiv preprint arXiv:1707.07596 (2017)
13. Muggleton, S., De Raedt, L.: Inductive logic programming: Theory and methods. The Journal of Logic Programming **19**, 629–679 (1994)
14. Novák, V., Perfilieva, I., Močkoř, J.: Mathematical principles of fuzzy logic. 1999
15. Richardson, M., Domingos, P.: Markov logic networks. Machine learning **62**(1), 107–136 (2006)
16. Rocktäschel, T., Singh, S., Riedel, S.: Injecting logical background knowledge into embeddings for relation extraction. In: Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. pp. 1119–1129 (2015)
17. Sen, P., Namata, G., Bilgic, M., Getoor, L., Galligher, B., Eliassi-Rad, T.: Collective classification in network data. AI magazine **29**(3), 93 (2008)
18. Serafini, L., Garcez, A.S.d.: Learning and reasoning with logic tensor networks. In: AI* IA. pp. 334–348 (2016)
19. Tran, D., Hoffman, M.D., Saurous, R.A., Brevdo, E., Murphy, K., Blei, D.M.: Deep probabilistic programming. In: International Conference on Learning Representations (2017)