Capita Selecta AI 2023-2024

Implementation of a Neurosymbolic system and application to MNIST Addition

Giuseppe Marra, Robin Manhaeve

November 2023

1 Introduction

In this assignment, you will dive into the field of neuro-symbolic AI. You will implement a neuro-symbolic system and experiment with it on the MNIST Addition task.

2 Practicalities

The assignment can be completed alone or in pairs. You must implement the required algorithms by yourself and you are not allowed to share code with others.
You are allowed to use standard Python libraries (e.g. Matplotlib, Scikit-learn, PyTorch, ...) but you must implement every project-related algorithm yourself.
Your assignment must adhere to general scientific standards of credit attribution. More details on this can be found at `https://www.kuleuven.be/english/education/plagiarism/`.

Questions

Please direct any questions that you may have about the assignment to the Toledo forum such that all students can benefit from the discussion. An FAQ section will be available on Toledo where you can ask questions.

3 Submission

Your submission should include three items:

1. A **report** in **.pdf** containing the output of the different tasks (max 6 pages);
2. A **code package** in **.zip** format
3. A **proposal for a new application** in **.pdf** to be implemented during Phase 3 (max 1 page).

4 Deadlines

Submit on Toledo **Before February 19th, 23:59**

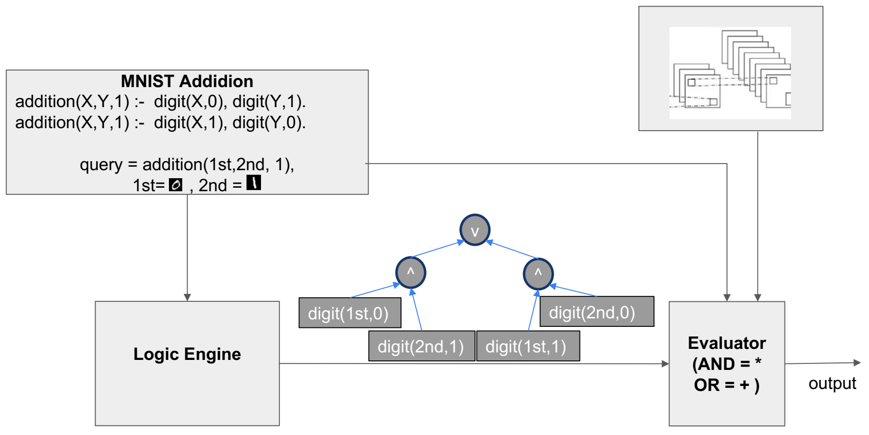

Figure 1: An high level overview of the system instantiated on the MNISTAddition task.

5 Score

The assignment will be scored on a **5 points** basis, jointly based on your report and code.

Note: Feedback on the report will be sent after the deadline. Correctly implementing the suggested changes in the final report is worth another 5 points. Having a better initial report makes it easier to earn the additional 5 points.

6 Installation

You will be working with **Python**. We have provided a skeleton for you to start from. It requires a Python version >= 3.10. The required packages are specified in `requirements.txt`, and can be installed with the following command:

```
$ pip install −r requirements.txt
```

Make sure that the `nesy/` folder is in the PYTHONPATH.

7 High Level Description

In this assignment you are going to implement a simple Neuro-Symbolic system and apply it to the MNIST Addition task. Figure 2 shows an high level overview of the system. There are 4 main components, which we describe in the following.

7.1 The Task

The **MNISTAdditionTask** (`example/dataset.py`) contains the task of computing the addition of two digits represented by two MNIST dataset images. A task is the collection of:

- a **logic program**; here the one telling which numbers sum to what:

$$addition(X,Y,Z) \leftarrow digit(X,N1), digit(Y,N2), add(N1,N2,N).$$

The provided code uses Prolog[1] syntax without functors (i.e. the Datalog fragment). In line with the majority of probabilistic and neurosymbolic logic programs, we extend such syntax with weighted facts (i.e. $weight :: fact.$). Important for the project are neural predicates, e.g.:

$$nn(digit, X, Y) :: digit(X,Y)$$

where the weight of the fact is computed by calling a neural network (see Section 7.3).

- a **list of queries**, specifying which the images and their corresponding sum. We used the notation $tensor(domain, const\_id)$ to symbolically refer to the $const\_id$-th element of the tensor $domain$. For example:

$$q = addition(tensor(images,0), tensor(images,1), 1)$$

asks for the probability that the 0-th and the 1-th elements of the tensor identified by $images$ sums to a 1.

- a list of **probabilistic labels**; i.e. the ground truth probability of the query being true

$$p(q) = 1$$

- a **tensor source**, which is a dictionary mapping domain names to the corresponding tensors. Each of these tensors has shape $[n, m, h1, ..., hn]$, where n is the number of queries in the batch, m is the number of constants in each query, indexed by $const\_id$, and $h1, ..., hn$ are the dimensions of each single input tensor. For example, for a MNISTAddition task for the 2-digit addition, with a batch size of 5, and grey-scale images of shape $[1, 28, 28]$, we will have a total shape of $[5, 2, 1, 28, 28]$.

There is an important difference between queries used during train and queries used during test. During train, we know the true label of the addition and we will only ask this unique query to the model. During test, we will ask for all possible labels of pair of images and then we will compute the most probable result. Therefore, during training, given a pair of images, a single query will be asked. During testing, given a pair of images, multiple queries will be asked. The system needs to handle both these two cases.

We will provide the MNIST Task for the 2-digit addition. At the end of your assignment, you will be asked to extend the task to multiple digits.

7.2 The Logic Engine

The logic engine (`src/nesy/logic.py`) is a purely symbolic component that takes a program and a set of queries and returns an And-Or tree [1, 2] containing all the proofs for the queries. An example of an And-Or tree for the query $q = addition(tensor(images,0), tensor(images,1), 1)$ is shown in Figure 2. There are many engines that can be used to this end. In this assignment, you will be asked to implement a simple forward chaining engine. In forward chaining, one applies those rules for which all the elements in the body are known to be true and adds the corresponding heads to the known facts. Such process is repeated until the query is found or no new facts can be inferred. A detailed description of a forward chaining algorithm for first order logic programs can be found in [2], Chapter on Inference in First Order Logic.

[1] https://en.wikipedia.org/wiki/Prolog

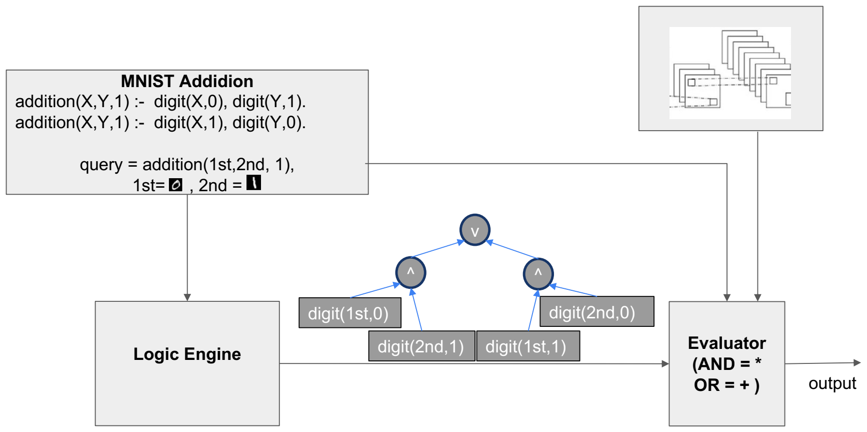

Figure 2: An high level overview of the system instantiated on the MNISTAddition task.

7.3 The Neural Predicates

The neural predicates are facts in your program that take a tensor as input (e.g. $tensor(images, 0)$)). For example, in the provided program, we have:

$$nn(digit, tensor(images, 0), 2) :: digit(tensor(images, 0), 2).$$

which means that the probability for the atom $digit(tensor(images, 0), 2)$ can be obtained by calling the neural network $digit$, passing the tensor $tensor(images, 0)$ as input and getting its 2-nd output.

Therefore, one has to provide, for each neural predicate type, e.g. $digit$, a corresponding neural network. We will provide a trivial neural net in the code (`example/run.py`), but you are allowed to change it to your needs.

7.4 The Evaluator

The Evaluator (`src/nesy/evaluator.py`) is responsible of evaluating the And-Or tree to provide the final score for each query.

An Evaluator visits the tree and assigns values to each node, all the way to the top (which represents the query). In particular,

- to each *leaf* node, we assign the value 1 if it is a known fact (e.g. $add(0, 3, 3)$) or the corresponding neural network output if it is a neural predicate (e.g. $nn(digit, tensor(images, 0), 2)$);
- to each logic operator (AND,OR), we assign the result of a continuous operation of the values of its children. For example, AND(a,b) is mapped to $min(a, b)$ using a Godel t-norm. Different semantics (`src/nesy/semantics.py`) of such continuous operations exist and are based on probabilistic logic or fuzzy logic. You will be asked to implement and compare these semantics.

8 Structure of the code

You will be provided with the following source files:

- `example/dataset.py` It containes the 2-digit MNIST Addition task.
- `example/run.py` It is the main script to run the training and the evaluation of the system. **Run this script to start getting used with the code.**
- `src/parser` It is a package containing parsing utilities.
- `src/term.py` It contains classes for encoding logic programs.
- `src/logic.py` It contains the LogicEngine interface.
- `src/evaluator.py` It contains the Evaluator interface.
- `src/model.py` It contains a PyTorch Lightning wrapper of the entire Neurosymbolic model. This is the model that will be instantiated in the `run.py` script.

We have provided you with lightweight, often under-specified interfaces. **This code is supposed to be a starting point for you and not a constraint. You are allowed to change such interfaces to fit your needs (e.g. adding arguments to functions, adding functions, etc).** The adherence of your code to the provided interfaces will not be considered or scored. The interfaces are only intended to help you start thinking about the implementation.

We also provided some dummy hard coded part of the solution, to give you some intuitions about what the outcome *could* look like.

9 Make it small!

The class MNISTAdditionTask (`example/dataset.py`) allows you to simplify the task by reducing: (1) the number of examples (`MNISTAdditionTask.nr_examples`) and, (2) the number of classes or possible digits (`MNISTAdditionTask.n_classes`). For example, `n_classes = 2` means that you will only add images representing 0s and the 1s.

10 Tasks

We group the task by type: implementation, experiments and new application. However, some experiments can already be performed before the full implementation is completed. Perform the experiments as soon as possible, so that you can use these experiments to further test your implementation.

Task 1: Implementation

You will need to implement:

- A `ForwardChaining` logic engine (`src/logic.py`);
- The `SumProductSemiring` semantics and three fuzzy logic semantics `LukasieviczTNorm`, `GoedelTNorm` and `ProductTNorm` (`src/semantics.py`). Check [1] for reference.
- The `Evaluator` (`src/evaluator.py`).
- The **n**-digit MNISTAddition (`example/dataset.py`). Extend the existing class to handle the sum of more than 2 digits. *Note:* here with multiple digits sum, we mean the task of summing multiple single digit numbers. In other words, you need to implement $x_1 + x_2 + ... + x_n$, where each x_i is an MNIST image between 0 and 9.

Note: You will need to change and implement utilities in other scripts as well, for example, to adapt the model interfaces to the functions you implement. As mentioned before, the provided code is supposed to be just a starting point.

Task 2: Experiments

Perform experiments to answer the following research questions:

Q1 How scalable is the logic engine when increasing the allowed number of digit classes (`n_classes`)?

Q2 How does the accuracy evolve during training?

Q3 How do the training curves of different semantics compare to each other?

Q4 How does the system scale when increasing the number of digits in the addition? This requires the implementation of the n-MNISTAddition task.

Q5-Q6 Design two further research question and perform experiments to answer them.

For Q5-Q6, possible questions are those evaluating the impact of your design choices when improving scalability. For example, if you find that some operations are repeated (*spoiler:* many operations are), can you cache them? How does impact your performances? Other possible questions may concern the robustness w.r.t. alterations of the input dataset. For example, adding noise to the input or to the label.

Task 3: New Application

Finally, you will need to propose (but not implement!) a new application (i.e. a new task) for your system. Take the answers to the research questions into account when choosing the task, in particular scalability. Choose a task on which your system can scale (probably, not much more difficult than the MNISTAddition).

References

[1] Giuseppe Marra et al. "From statistical relational to neural symbolic artificial intelligence: a survey". In: *arXiv preprint arXiv:2108.11451* (2021).

[2] Stuart J Russell and Peter Norvig. *Artificial intelligence a modern approach*. London, 2010.