Project Documentation

AriadneDL

Deep Learning for Ariadne

Mirco De Marchi

Contents

Chapter 1

Introduction

Ariadne is a tool for reachability analysis and model checking of hybrid systems. Additionally, Ariadne is a framework for rigorous computation featuring arithmetic, linear algebra, calculus, geometry, algebraic and differential equations, and optimization solvers.

AriadneDL wants to be a library that extends Ariadne functionalities in order to provide online estimations of the running system to improve its execution through Machine Learning and Deep Learning techniques. In particular, AriadneDL focuses on 2 main use case:

- Task execution time estimation;
- Adaptation of the dynamic part of a hybrid system model;

The implementation has to be done in C++ and it can be based on a library for Machine Learning and Deep Learning, with strict compatibility for the major operating systems. As alternative, the library can be developed from zero, in order to build an ad-hoc implementation. The library has to be compatible with MacOS, Ubuntu, Debian and Windows, in particular the library has to be provided by the following package manager: Homebrew, Aptitude and vcpkg. The libraries discussed will be the following:

- Tensorflow: github.com/tensorflow, /www.tensorflow.org;
- Caffe: github.com/caffe, caffe.org;
- Cognitive Toolkit (CNTK): github.com/CNTK, docs.microsoft.com/cognitive-toolkit;
- dynet: github.com/dynet, dynet.io;
- shogun: github.com/shogun, www.shogun.ml;

- FANN: github.com/fann, leenissen.dk/fann;
- Shark library: github.com/Shark, www.shark-ml.org;
- OpenNN: github.com/OpenNN, www.opennn.net;
- mlpack: github.com/mlpack, www.mlpack.org;
- Boost: github.com/boost, www.boost.org;
- Eigen3: gitlab.com/eigen, eigen.tuxfamily.org;
- Armadillo: gitlab.com/armadillo, arma.sourceforge.net

The following list provides some useful links related to the implementation and other resources:

- Ariadne main repository: github.com/ariadne-cps/ariadne;
- AriadneDL extension repository: github.com/mircodemarchi/AriadneDL;

Chapter 2

Requirements

2.1 Task execution time estimation

The first Ariadne integration is the **estimation of task execution time**. Ariadne can execute in parallel multiple tasks, that are represented as numeric integrations of a set that evolves over time. The tasks amount executed during the system evolution before the global integration finishes, is quite a large number. The objective is to find the best schedule of the task with different configurations, in order to obtain the best result in terms of execution time. This implementation can be useful in Ariadne, because different tasks could have really different execution time and the predicted execution time of a specified task configuration can be used to improve the threads scheduling.

Let assume to take in input a vector of bounded integer values, that represents the task parameters configuration for the numeric integration, the output has to be the estimated execution time of input task in microseconds. The task takes another input, in addition to the numeric configuration, that is the initial state of the set. The set is represented as a Taylor expansion with dozen of terms related with the continuous variables of the set space, that is more difficult to represent in the training model of a neural network. The set state has to be taken in consideration during the evolution of system tasks, because the task execution time tends to vary during the temporal evolution of the set, according to its complexity.

The deep learning model can be initially trained with a sequence of task parameters configurations labelled with their execution time. Then the estimator model has to take in input a task configuration and produce the predicted execution time. Even if the system has to perform multiple task in concurrence, the estimator model is interested in each single task execution. For this reason the most suitable solution for this integration would be a simple Feedforward Neural Network (FNN), without too many layers, in order to avoid an excessive storage of training data.

The prediction model can be improved over time, because when tasks finish their execution, Ariadne keeps track of real execution time. Furthermore, the deep learning model has to work well for the last evolution time interval rather than for all evolution, therefore it has to loose memory of oldest executions during online learning and overfitting has to be avoided. To implement this, the estimator has to provide an online learning, in order to give more importance to the latest execution time results obtained.

2.2 Parametric dynamic system adaptation

The second Ariadne integration is the **adaptation of a parametric function that represents the dynamic part of a hybrid system model**. Given a vector of samples, defined on continuous variables, that represent the states of a real system, and an associated hybrid model, we want to get the best accuracy of the hybrid model compared to the real system. Since time evolution is faster than real system evolution, we can use the hybrid model to predict in advance the points that the real system will reach, in order to catch dangerous situations. However, if the hybrid model is not so accurate, the performed prediction will be inaccurate.

The accuracy of this hybrid model can be improved through some configurable parameters that are given to the functions associated with the model dynamic at each location. The parameters can be defined as point values or, more generally, intervals, and allow to include the samples evolution. In fact, the hybrid model evolution can be represented as a flow channel that is the time evolution of a non-punctual set. The narrower the flow channel, the more accurate the model prediction and the greater the likelihood of including samples of real evolution in the flow channel defined by the sequence of chosen parameters.

The hybrid model evolution is periodically repeated starting from the last sample received by the real system. Therefore, the hybrid model can learn how to improve the sequence of chosen parameter through its errors. The proposed solution is to integrate a deep learning model inside the hybrid model that periodically takes as input the real system state, defined as a sequence of continuous variable samples, and it has to give as output the parameters values for the parametric functions of each dynamic model location that allow the hybrid model to best return a set of points that includes the future real system samples. The output has to be given as an interval, that is a minimum-maximum pair.

The deep learning model can be trained with a sequence of samples produced by the real system and a related sequence of simulated values produced by the hybrid model of the real system, with a defined parametric function for each hybrid model locations. In addition, the estimator has to be able to improve periodically the accuracy during system execution, for this reason the model has to perform an online learning. Ideally, in a real system that perform over a loop, the learning phase should be executed along all the samples collected in the loop. However, since the loop period is not easily findable, the number of samples on which perform the learning phase is not defined. Finally, this solution needs to define a quality metric of the estimator.

Furthermore, the prediction could be improved with an inference performed along a sequence of samples, because a future state can be found more accurately if the learning model had some of the past sequence of states available. For example a past sequence of states might suggest a direction of the states evolution, if the states evolution is located around a limited area or if it spreads over a large area. For this reason the most suitable solution would be a Recurrent Neural Network (RNN), because it suits really well with temporal sequences. However, the RNN learning model works very well if the real system future state depends on a sequence of previous states, but if the evolution states of a real system depends only on the current state, the usefulness of RNN decreases.

Since there is not a strict necessity to avoid overfitting and the complexity of this use case, compared to the first one, is higher, a good solution could be to implement a Feedforward Neural Network (FNN) with a larger amount of layers, in order to manage better a difficult problem. Therefore, if the RNN does not provide good results then you could opt for the FNN solution.

Chapter 3

C++ Libraries

3.1 Deep Learning libraries

The most famous deep learning libraries are: TensorFlow, Caffe and Cognitive Toolkit. These libraries are complete, really optimized, with every tool that you need and easy to use because based on a simple and abstract interface. However, they are not easy install, and not available on all operating system and package manager. In particular, TensorFlow is a Google library, really used, mostly in Python, for mobile and IoT applications. The following are the main features:

- Programming languages support: Python, C++, Javascript, Java, Go, Swift;
- CUDA support;
- Installation: pip, docker or homebrew;

Caffe is an efficient, speed and modular library with also a command line interface. The following are Caffe's main features:

- Programming languages support: C++, Python, Matlab;
- CUDA support;
- Installation: docker, apt, homebrew, and, for Windows, you need to build the project from source in the windows branch.
- Math libraries dependencies: BLAS and BOOST;

Cognitive Toolkit is a library developed by Microsoft that allows the user to easily realize and combine popular model types. The following are CNTK's main features:

- Programming languages support: C++, Python, C# or its own model description language (BrainScript);
- CUDA support on multiple GPUs and servers;
- Installation: docker, pip, but generally it is installed from source through an installation script. It is not supported for MacOS.

After that, there are some libraries, not very well known, but quite optimized and specific that could be interesting: DyNet, Shogun, FANN, Shark Library, OpenNN and mlpack. DyNet is a C++ library that works well with networks that have dynamic structures that change for every training instance. These kinds of networks are particularly important in natural language processing tasks. The following are DyNet's main features:

- Programming languages support: C++, Python;
- CUDA support;
- Installation: pip, homebrew.
- Math libraries dependencies: Eigen;

Shogun is a library not really specialized in deep learning models, but it offers mostly a wide range of efficient and unified machine learning methods. The following are Shogun's main features:

- Programming languages support: C++, Python, Octave, R, Java/Scala, Lua, C#, Ruby;
- No CUDA support;
- Installation: in the official installation website you can find the instructions for apt, homebrew, docker and pip package managers. Shogun natively compiles under Windows using MSVC.

FANN is a easy to use, versatile, well documented, and fast library, which implements multilayer artificial neural networks in C with support for both fully connected and sparsely connected networks. The following are FANN's main features:

- Programming languages support: C++, Python, Octave, R, Java/Scala, Lua, C#, Ruby, Matlab, Perl, PHP, Javascript and others;
- No CUDA support;

- Installation: vcpkg install support in Windows, but only from source for Linux systems. See official github repository for instructions.

Shark Library is a fast, modular, general open-source C++ machine learning library, with support for Feedforward Neural Network and Autoencoders. This library is not so big and it can be useful to use as a source from start to build a new custom ad-hoc library for our use cases. The following are Shark's main features:

- Programming languages support: C++;
- No CUDA support (only experimental OpenCL support on earlier releases through BLAS library);
- Installation: works on Windows, MacOS X, and Linux but only from source.
- Math libraries dependencies: Boost and BLAS;

OpenNN is a high performance library in terms of execution speed and memory allocation, with neural networks and machine learning algorithms. As in the case of Shark Library, also OpenNN library is easy to read from source and it can be used as starting point for developing a new ad-hoc library. The following are OpenNN's main features:

- Programming languages support: C++ and Python;
- support CPU parallelization by means of OpenMP and GPU acceleration with CUDA;
- Installation: only from source.
- Math libraries dependencies: Eigen;

Mlpack is an intuitive, fast, and flexible C++ machine learning library. This is the only library that provides more compatibilities for operating systems and package managers. The following are Mlpack's main features:

- Programming languages support: C++, Python, Julia, Go, R and provides a command line interface;
- No native CUDA support, but indirectly inherits CUDA support through Armadillo library dependency;

- Installation: in the official get stared guide there are all the informations to install mlpack on MacOS and Linux (Debian), with support for homebrew, Pkg.jl, apt, pip, docker or from source. Install from source or through vcpkg for Windows.
- Math libraries dependencies: BLAS, Armadillo and Boost;

Finally, a good solution could be the one to develop a custom ad-hoc library. A custom ad-hoc library would result lighter and more efficient, because specific for the task to solve. On the other hand, a third-party library would result in faster, easier developing and a wide choice availability of machine learning algorithms that can be used also for future implementations. The following is a list of useful reference from which to start developing a custom ad-hoc C++ library:

- github.com/3ammor/SimpleNeuralNet
- github.com/huangzehao/SimpleNeuralNetwork
- github.com/jeremyong/cpp_nn_in_a_weekend
- github.com/Whiax/NeuralNetworkCpp

3.2 Math libraries

A custom ad-hoc implementation could be developed with a library that supports mathematical operations along huge arrays and matrices. The following are libraries that can be used in C++ for machine learning and deep learning applications: Boost, Eigen3 and Armadillo.

Boost is a set of libraries for the C++ programming language that provides support for tasks and structures such as linear algebra, pseudorandom number generation, multithreading, image processing, regular expressions, and unit testing. Boost library is not built to support CUDA internally but from CUDA 2.2 you can use Boost functionalities inside CUDA kernels. Boost supports the main installation package managers: homebrew, apt, vcpkg.

Eigen3 is a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms. Similarly to Boost library, Eigen3 doesn't support CUDA internally but the library functionalities can be partially used in CUDA kernels. Eigen3 supports the main installation package managers: homebrew, apt, but it doesn't support vcpkg.

Armadillo is a high quality linear algebra library for the C++ language, aiming towards a good balance between speed and ease of use. Armadillo is a library of higher abstraction level than Boost or Eigen3 libraries, in fact it uses BLAS library as dependency. In order to obtain GPU speed up on large matrix multiplications, you can link Armadillo with NVBLAS, which is a GPU-accelerated implementation in CUDA of BLAS. Armadillo supports the main installation package managers: homebrew, apt, vcpkg.

Chapter 4

Implementation

Chapter 5

Conclusion