Correct by Construction Machine Learning

Artjoms Šinkarovs[0000−0003−3292−2985]

Southampton University, UK
a.sinkarovs@soton.ac.uk

Abstract. In this talk I present how theorem provers can be used to define machine learning applications together with correctness invariants of interest. This approach can be seen as a pathway towards encoding provably safe AI.

1 Introduction

With increased use of machine learning in safety-critical systems, there is a high demand in guaranteeing correctness of these systems. Unfortunately, the notion of correctness varies depending on the community. For example, machine learning experts may focus on minimising an average error when running applications on certain sets of inputs. Mathematicians may interpret machine learning applications as functions in vector spaces and require certain constraints on them. Programmers may be interested in ensuring that machine learning applications do not crash or throw exceptions at runtime.

While all of the mentioned notions are of high importance, we rarely see machine learning systems that make it possible to state and/or prove correctness properties about the actual applications. One of the reasons is that most of machine learning frameworks prioritise functionality over correctness. That is, one can quickly build applications, but there is no guarantee that they produce meaningful results. This means that all correctness guarantees about such applications have to be provided extrinsically.

There are several issues with extrinsic verification. First of all, there are significant limitations of what can be achieved automatically. For a given application written in weakly-typed languages such as C or Python, some properties are just undecidable. Secondly, extrinsic verification decouples the application from the correctness proof, meaning that one can run applications even if correctness proofs are incomplete or missing. Finally, each notion of correctness mentioned above is likely to require a separate verifier.

An alternative approach is to use intrinsic verification, defining machine learning applications together with correctness properties of interest. In particular, we can define machine learning applications within theorem provers, which makes it possible to attach arbitrary properties to the application. Such an approach guarantees that: (i) all properties of interest can be stated within a single framework; (ii) specifications are executable; (iii) correctness proofs cannot be decoupled from applications.

The main challenges of intrinsic verification are: (i) the entire application (including frameworks and libraries) have to be (re-)implemented within a theorem prover; (ii) a programmer becomes responsible for supplying all the required correctness proofs; (iii) there has to be a mechanism on running the verified code efficiently.

In the talk I will demonstrate that many of these challenges can be addressed in a reasonably straight-forward way. Concretely, I will present the proposed intrinsic approach within Agda — a theorem prover that is based on dependent types. I will use a simple convolutional neural network as a running example. I will demonstrate that with a reasonably small effort we can obtain a correct-by-construction implementation of the network that performs on par with state of the art competitors. I will explain the difficulties of the approach and further avenues that this feasibility study opens.

2 Technical Details

In this section I make an overview of the implementation, explaining how some of the above-mentioned challenges are addressed.

One of the key ingredient in any machine learning framework is tensor (in the machine learning sense of the word). In dependently-typed systems a type of a tensor can be indexed by its shape, which guarantees that: (i) all selections into tensors are within bounds; (ii) we can explain precisely how tensor shapes are related in the given operation; (iii) we have the ability to abstract over shapes resulting in very generic definitions, e.g. we can have a single definition of convolution for tensors of any rank.

Basic tensor operations are defined in less than a hundred lines of code with no explicit proofs. One operation that requires more work is a generalised convolution. The reason for this is that the shape of the output tensor depends on the shapes of the input ones in a non-trivial way. With tensor operations in-place the forward part of our example network can be defined in just 5 lines of code, which is comparable to machine-learning frameworks such as Tensorflow or PyTorch.

Another important component of machine learning frameworks is automatic differentiation which simplifies definition of training phases. This step is also implemented in Agda by defining a deeply-embedded DSL that wraps tensor operations. The implementation guarantees that tensor shapes are respected and variables within the DSL are well-typed and well-scoped. The core function that computes derivatives is compact taking about 30 lines of code.

The final step of the process is translation of the DSL expressions into high-performance languages. In particular, I can translate the DSL programs into C, SaC or Futhark. The latter generates the code that runs on GPUs. For our example, the runtime of the generated code is comparable with Tensorflow or Pytorch. An important step prior translation to high-performance languages is an optimisation of DSL expressions. This step is also implemented in Agda in semantically-preserving way, guaranteeing that optimisations do not change the value computed by the given expression.