Interoperating Deep Learning models with ONNX.jl

Ayush Shridhar[1] and Mike Innes[2]

[1]International Institute of Information Technology, Bhubaneswar, India
[2]Julia Computing

ABSTRACT

Flux[4] is a machine learning framework, written using the numerical computing language Julia[2]. The framework makes writing layers as simple as writing mathematical formulae, and it's advanced AD, Zygote[3], applies automatic differentiation (AD) to calculate derivatives and train the model. It makes heavy use of Julia's language and compiler features to carry out code analysis and make optimisations. For example, Julia's GPU compilation support[1] can be used to JIT-compile custom GPU kernels for model layers[5]. Flux also supports a number of a hardware options, from CPUs, GPUs and even including TPUs.

ONNX.jl is an Open Neural Network Exchange backend for the Flux.jl deep learning framework. ONNX.jl supports directly importing high quality ONNX standard models into Flux, thus saving time and reducing the need for additional computation resources. In this paper we'll look into finer details of how ONNX.jl produces Julia code of a model from a serialized file.

Keywords

Julia, Machine Learning, Deep Learning, Transfer Learning

1. Introduction

Open Neural Network Exchange (ONNX) is an open ecosystem that empowers AI developers to choose the right tools as their project evolves. ONNX provides an open source format for AI models, both deep learning and traditional machine learning. It defines an extensible computation graph model, as well as definitions of built-in operators and standard data types. It provides a set of specifications to convert a model to a basic ONNX format, and another set of specifications to get the model back from this ONNX form.
At a high level, ONNX is designed to allow framework interoperability. There are many excellent machine learning libraries in various languages : PyTorch, TensorFlow, MXNet, and Caffe are just a few that have become very popular in recent years, but there are many others as well.

1.1 A simple use case

ONNX is simplifying AI choices by defining a standard which is here to stay, and is usable anywhere from small mobile devices to large server farms, across chipsets and vendors, and with extensive runtimes and tools support. ONNX reduces the friction of moving trained AI models among your favorite tools and frameworks and platforms. A simple example of how ONNX is ideal for ML is the case when large deep learning models need to be deployed.

Consider the simple case of deploying a Deep Learning model to an iOS application. This particular model can be implemented in any framework : TensorFlow, PyTorch, MXNet just to name a few. However, iOS applications expect to use CoreML inside the application. Up until now, developers have been porting large models to different frameworks, which is a waste of time and energy, better spent somewhere else. This is also retraining the entire model from scratch, which isn't efficient. This makes the entire process cumbersome and impractical. ONNX exists to solve this very problem : By connecting the common dots from different frameworks, ONNX makes it possible to express a model of type A to type B, thus saving time and the need to train the model again.

2. ONNX backend in Julia: ONNX.jl

At the heart of it, ONNX.jl solves a compiler problem by dealing with intermediate code representations to generate readable graphs. While doing this, ONNX operators are mapped to corresponding Flux layers, thus tracing out the model's computation graph at the end. This graph can then be travelled to generate the Julia code for the model.

3. Structure of an ONNX serialized model

ONNX uses Google's Protocol Buffers to serialize and de-serialize models. Julia has a Protocol Buffer backend via ProtoBuf.jl.
Since ONNX tries to inherit properties from diverse frameworks, ONNX serialized models can be large and complicated. In general, the model is essentially a *ModelProto* object, that consists a number of sub-fields. The *ModelProto* structure looks something like:

```
mutable struct ModelProto <: ProtoType
    ir_version::Int64
    opset_import::Vector{OperatorSetIdProto}
    producer_name::AbstractString
    producer_version::AbstractString
    domain::AbstractString
    model_version::Int64
    doc_string::AbstractString
    graph::GraphProto
    metadata_props::Vector{StringStringEntryProto}
end
```

Of all these fields, *graph* is the most essential attribute. It captures the entire computation graph of the model. The *GraphProto* follows the structure:

```
mutable struct GraphProto <: ProtoType
    node::Vector{NodeProto}
    name::AbstractString
    initializer::Vector{TensorProto}
    doc_string::AbstractString
    input::Vector{ValueInfoProto}
    output::Vector{ValueInfoProto}
    value_info::Vector{ValueInfoProto}
end
```

Any model graph consists of a number of interconnected nodes. In ONNX, every such node is a *NodeProto* type. It shows the inputs to the node and the output of the node.

```
mutable struct NodeProto <: ProtoType
    input::Vector{AbstractString}
    output::Vector{AbstractString}
    name::AbstractString
    op_type::AbstractString
    domain::AbstractString
    attribute::Vector{AttributeProto}
    doc_string::AbstractString
end
```

op_type is the metadata that stores the type of the operation being done in the Node. This includes Convolution, Dense, different types of normalization layers to name a few.

4. Extracting data from the *ModelProto*

Every ONNX serialized model is stored in an *ModelProto* object. This captures all attributes, parameters and hyper-parameters needed to load the model. Internally, *ModelProto* consists of several other *Proto* objects. The major ones that capture the entire nature of the model are *GraphProto , NodeProto, TensorProto and AttributeProto*. Each serves a different purpose, as follows:

—*GraphProto* stores the entire computation graph for the deep learning model.

—*NodeProto* stores attributes of individual nodes of the graph. This mostly includes the several layers used in the model.

—*AttributeProto* stores the weights of the model via *TensorProto*. It also stores other miscellaneous attributes. This includes properties such as data type, operator version, onnx version.

—*TensorProto* stores the parameters associated with the model. In the case of Deep Learning model, this is the weights of the models.

Internally, *ProtoBuf.jl* is used to read the ONNX graph and *BSON.jl* is used to read and write the model weights.

5. The Julian approach : Converting ONNX graph to DataFlow graph

DataFlow.jl is a code intermediate representation (IR) format. Unlike SSA, which allows only statements, DataFlow allows only expressions. A data flow graph is a bit like an expression tree without variables; functions always refer to their inputs directly. Underneath it's a directed graph linking the output of one function call to the input of another. DataFlow.jl provides functions like prewalk and postwalk which allow you to do crazy graph-restructuring operations with minimal code, even on cyclic graphs.

6. Interface

ONNX.jl provides a minimal interface to the user. The API exposes just a handful of functions. Out of these, the *load_model* function is the most essential function. It takes in the path to the ONNX serialized model as a argument, and generates two files:

(1) The *model.jl* file : This is essentially the Julia code for the entire model. The model code uses purely Base and Flux operations.

(2) The *weights.bson* file: This is a binary file which stores a dictionary mapping the name of the layer to the weights associated with the corresponding layer.

7. Internals

Internally, the task of generating the aforementioned files can be summarized in four main steps:

(1) Data Cleanup : ONNX graphs consist of a number of redundant fields. These need to be removed in the earlier stages to prevent complications later.

(2) Conversion of ONNX graph to DataFlow graph.

(3) Replacing the nodes of the graph, which are actually layers of the model, with Flux operators.

(4) Generating the code and weight files.

8. Additional features

Unlike other ONNX backends, the result of the ONNX.jl is the Julia code for the model. This makes it much more intuitive and ideal for using this as a starting point for other tasks, say neural style transfer or semantic segmentation.

9. Related Work

Another recent development in this direction is Google's *Multi-Level Intermediate Representation*, or *mlir*. Unlike ONNX, mlir aims an implementing a common Intermediate Representation for deep learning models, thus unifying TensorFlow and other frameworks.

10. Example Usage:

Loading a model is as simple as :

```
using ONNX, Flux
ONNX.load_model("path_to_onnx_file")
weights = ONNX.load_weights("weights.bson")
model = include("model.jl")
```

model above is corresponding model, and can be treated like any other Flux model.

11. Conclusion

In conclusion, ONNX.jl presents a effective and efficient way to enable researchers and developers to make use of the power of the Julia langauge, the elegance of Flux and the availability of a vast number of pre-trained models. This enables researchers to spend time focusing on the real issues, rather than model portability.

12. References

[1] Tim Besard, Christophe Foket, and Bjorn De Sutter. Effective extensible programming: Unleashing Julia on GPUs. *arXiv*, abs/11712.03112, 2017.

[2] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[3] Michael Innes. Don't unroll adjoint: Differentiating ssa-form programs. *CoRR*, abs/1810.07951, 2018.

[4] Mike Innes. Flux: Elegant machine learning with julia. *Journal of Open Source Software*, 2018.

[5] Mike Innes et al. Generic gpu kernels, 2017.