Open Neural Network Exchange (ONNX) Support for Julia Developers

Ege Ersü
Koç University
Artificial Intelligence Laboratory
Istanbul, Turkey
egeersu.github.io

Abstract

Developers often have to stick to a single programming language and framework for their Deep Learning projects. Sharing pre-trained models between frameworks is a tedious task, requiring the preservation of the structure of the neural network, as well as its individual layers and parameters. We developed **KnetOnnx.jl**, a software package in Julia that makes use of the Open Neural Network Exchange (ONNX) format to automate this process, giving Julia developers the tools to read ONNX files and re-construct the corresponding models in Knet. Given the ONNX representation of a model, we provide the user with a Knet Model that can be re-designed, re-trained or simply used for inference. Our current build can convert all multi-input & multi-output graphical models, as long as the operators are supported by the ONNX format and our package. We hope that this technical report will also serve as a reference to developers who wish to make their Deep Learning Framework compatible with ONNX.

1 Introduction

Due to the immense number of tools, frameworks and programming languages that are being used in the field of Machine Learning, transferring pre-trained models from one framework to another is a tedious task. This difficulty consists of two major problems: (1) different implementations of the same model in distinct languages & frameworks, and (2) sharing the parameters of the model in a way such that when they are read into the target environment, we know to which layer the parameters belong to and that the right data structure is used to re-create it in the new environment. Although there has been many attempts by the community to ameliorate the problem of sharing weights, Open Neural Network Exchange (ONNX) is the largest open ecosystem so far that aims to finally solve both problems.

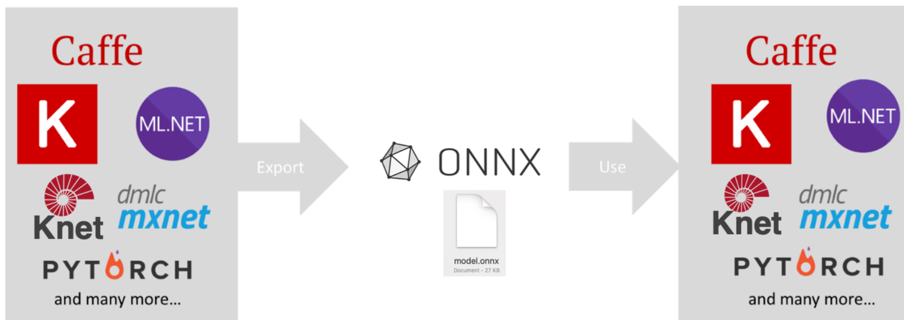

Figure 1: Import & Export

1.1 Open Neural Network Exchange Format

Deep learning models make their predictions through a computation over dataflow graphs. The idea is that if there was a universal representation of neural networks, independent of the programming language (or specific implementation) that is being used, one could always represent their neural network in that format. ONNX provides that intermediate representation we are looking for, capturing the essential properties of a neural network such as its inputs, outputs, structure and also information regarding each individual layer that the neural network consists of. More specifically, it provides (1) a definition of an extensible computation graph model, (2) definitions of standard data types and (3) definitions of built-in operators. Once we have a fixed format for representation, each Deep Learning Framework ought to provide the import and export functionality. The export function takes a model implemented in Framework X, and constructs a representation of it using the ONNX guideline and creates the file: model.onnx. The import function handles the other direction: reading the ONNX file and implementing the corresponding model in Framework X. Once a framework implements both functions it is ONNX-complete, meaning that the developer can transfer their model in and out of the framework.

1.2 The ONNX Graph

The ONNX Graph is structured as a list of nodes that form an acyclic graph. Each **node** of the graph represents one of the built-in operators and its attributes. To give an example, a node could be the Convolution operation, and its attributes would contain information regarding the padding and stride that must be used. Every ONNX-complete framework shall provide implementations of these operators on the applicable data types. Each edge of the graph represents a piece of data. So the input to the Convolution operation might be a tensor with the name: "input1". As a side note, the nodes contain the names of the incoming and outgoing edges (names of the tensors), so it is sufficient to only process the nodes. The graph also has metadata to help document its name, purpose, author, etc.

Finally, the ONNX Graph has the field: **initializer**, a dictionary from the names of the parameters to their actual values. In order to preserve the parameters of the original model, the target framework must use these values to initialize each individual layer. So the Convolution layer corresponding to our hypothetical node should be initialized by first looking at the name of its parameters, then using the initializer to get the actual values of the parameters.

2 System Design

The current version of our software package (KnetOnnx v0.1.0) only provides the import functionality, and the export function is currently left as future work. Let us motivate the design of our system by first dividing it to its sub-components. The system must first read the ONNX file and create a graph in Julia. This is done by our ONNX Reader (subsection 2.1). Once the graph is created in Julia, a program should iterate over all nodes and create the corresponding layers by using the information stored within the nodes. This is achieved with the use of operator-specific converters (subsection 2.2). In order to create the corresponding layers in Knet, we also need those layers to be implemented in Julia. Since Knet does not currently have a complete library of layers, we had to construct our own mini library of layers (subsection 2.3). Finally, we designed a Knet Model class which would be the model we return to the user (subsection 2.4).

2.1 ONNX Reader

An ONNX file is a Google Protobuf binary file. A Protocol Buffer is a language-neutral, platform-neutral, extensible way of serializing structured data for use in communications protocols, data storage, and more [1]. In order to extract the information, we made use of a Julia Package called ProtoBuf.jl. The resulting piece of data is composed of data structures that are not native to Julia, so we created our own Graph class and converted Proto objects to Julia types. The resulting Graph is easy to read, process and print. The code can be found in the graph directory of our source code.

2.2 Converters

A converter looks at the representation of an operator as specified by the Julia graph (the output of our ONNX reader), and returns the implementation of that layer in Julia. There is a unique converter for each operator, since all layers depend on a different set of attributes. We make use of a general-purpose **convert** function that takes the entire Julia graph as input and iterates over all nodes one by one. After determining the type of the operation specified by a node, it calls the unique converter for that specific type. All the resulting layers are collected in a list, which will later be used to create a Knet Model. If the converter for an operation does not yet exist the convert function throws an error, notifying the user that the model cannot be re-constructed due to that certain operation. In order to add a new converter, one must first ensure that there exists a Knet Layer that can be used to implement that layer in Julia.

2.3 Knet Layers

Currently Knet does not provide a complete library of neural network layers. It provides a variety of operations like convolution or pooling; but the user must define their own layers by using these more primitive operations. KnetLayers.jl was developed by Ekin Akyürek to address this issue [2], but it only covers a tiny set of operators that an ONNX-complete framework must implement. We still greatly appreciate Ekin's effort, since our own Layer library is built on top the latest version of KnetLayers.jl by simply adding new layers as it was needed. Please realize that to make our package support some operator O, we need to add two components to our package: (1) A KnetLayer K that implements operator O, and (2) a converter that looks at a node representing, and constructs a K object with those attributes. Once the KnetLayer is implemented successfully, implementing the converter for an operation amounts to gathering the parameters (if there are any) from the graph's initializer, the attributes from the node itself, and calling the corresponding KnetLayer constructor with the right arguments. In our current build, we also return the local inputs and outputs of a node (which are the names of incoming & outgoing tensors) to make the forward pass of a Knet Model faster.

2.4 Knet Model

Knet Model is what our system returns as its final output to the user. It can be thought of as an implementation of a computational graph where each node is a Model Layer. A ModelLayer is nothing but a KnetLayer, accompanied by the names of the input tensors going in and the names of the output tensors going out of that layer. The **layers** field of the Knet Model contains all ModelLayers, so local connections can be inferred from it. The Knet Model also stores the names of the inputs and the outputs of the entire model, so that we know which tensors to return to the user when a forward pass is completed.

So where are all these tensors stored? The final field of the Knet Model is **tensors**, a Julia dictionary that maps tensor names to actual tensors. When a model is created, all tensors are initialized to null values. When the forward function of a Knet Model is called, the input tensors given as arguments to the forward function are placed into the tensors dictionary. Then, we start the process of randomly picking a layer and computing its output. A layer's output can be computed only if its inputs are already computed, so we repeat this process until all tensors are calculated. Once a valid path is found, it is saved to be used in further forward calculations. This process might be replaced by a smarter algorithm in the future, but currently is not necessary since the path is computed only once, before the first forward pass.

3 Analysis & Results

3.1 Operators

An ONNX-complete framework must support all operators that are specified by ONNX [3]. Although we are still working towards that goal, here is a list of operators that our current build supports by providing a KnetLayer implementation, and a converter for their nodes in an ONNX graph.

1. ReLU
2. LeakyReLU
3. Conv
4. MaxPool
5. Dropout
6. Flatten
7. Gemm
8. Add
9. BatchNormalization
10. ImageScaler
11. RNN
12. Unsqueeze
13. Squeeze
14. Concatenate
15. ConstantOfShape
16. Shape
17. Constant

3.2 Models

In our proposal we specified a list of models that we were planning to make sure our software package can read and re-construct without error. We are pleased to report that they have all been copied over successfully, and their demos can be found on our repository's demo section. We also show that the Knet Model can be trained on the MNIST dataset successfully, by copying over a Convolutional Neural Network from PyTorch using ONNX and re-training it on Knet.

1. Multilayer Perceptron (MNIST Demo) [4]
2. Multiple-input, Single-output models
3. Single-input, Multiple-output models
4. Multiple-input, Multiple-output models
5. Branching Models [5]
6. Convolutional Neural Network (MNIST Demo) [6]
7. Recurrent Neural Networks [7]
8. VGG-16 [8]

Models listed above were crucial as a success metric, and it also shows our order of implementation and testing. Copying over a Mutlilayer Perceptron requires that the ONNX Reader, Parameter Trasnfer and two types of Knet-Layers and converters work as intended. A successful copy of a Multiple-input, Multiple-output model requires that the KnetModel knows which tensors to treat as inputs and which tensors to return as outputs. A branching model is any model where at least one layer has more than one output, and each output is input to a different layer. We have experimented with models with multiple branches, and shown that the Knet Model can always find a valid path and that the forward computation halts when all local computations are over. Convolutionals Neural Networks, Recurrent Neural Networks and VGG-16 were also tested to show that our package can handle certain operations that popular deep learning models usually make use of.

4 Conclusion

We believe that KnetONNX.jl is a software package that has the potential to be used by many Julia developers who wish to read ONNX files in Julia or simply transfer their models over to Knet. If further-developed and advertised, it could help Julia and Knet gain new users.

In terms of future work, we must first address the elephant in the room: the export function. The harder solution would be to do it as PyTorch does, by running a dummy input through any function and identifying the operations that are being used; and writing that to a binary protobuf file. The easier but less general solution would be to handle each KnetLayer separately, by writing a KnetLayer to ONNX node converter one by one. This would unfortunately only handle user models that are instantiated as Knet Models, since the graph structure would have to be extracted from its Model Layers.

4.1 Project Links

Project Video: youtube.com/watch?v=0OZIyi_0_70
GitHub Repository: github.com/egeersu/KnetOnnx.jl
Julia Registries: https://github.com/JuliaRegistries/General/tree/master/K/KnetOnnx

References

[1] developers.google.com/protocol-buffers/docs/overview

[2] github.com/ekinakyurek/KnetLayers.jl

[3] github.com/onnx/onnx/blob/master/docs/Operators.md

[4] github.com/egeersu/KnetOnnx.jl/tree/master/test/ONNX_files/mlp.onnx

[5] https://github.com/egeersu/KnetOnnx.jl/tree/master/test/ONNX_files/branch.onnx

[6] https://github.com/egeersu/KnetOnnx.jl/tree/master/test/PosterDemo/Knet_MNIST.ipynb

[7] https://github.com/egeersu/KnetOnnx.jl/tree/master/test/ONNX_files/rnn.onnx

[8] https://github.com/egeersu/KnetOnnx.jl/tree/master/test/ONNX_files/VGG.ipynb

5 Appendix

```
julia> PrintGraph(graph)
model inputs: ["input.1"]
model outputs: ["16"]
(op1) Conv
    input1: input.1
    input2: conv1.weight
    input3: conv1.bias
    output1: 9
(op2) Relu
    input1: 9
    output1: 10
(op3) Conv
    input1: 10
    input2: conv2.weight
    input3: conv2.bias
    output1: 11
(op4) MaxPool
    input1: 11
    output1: 12
(op5) Flatten
    input1: 12
    output1: 13
(op6) Gemm
    input1: 13
    input2: fc1.weight
    input3: fc1.bias
    output1: 14
(op7) Relu
    input1: 14
    output1: 15
(op8) Gemm
    input1: 15
    input2: fc2.weight
    input3: fc2.bias
    output1: 16
```

Figure 2: An ONNX Graph in Julia