Model Import and Export in Julia

GSoC 2018 Project Proposal

Ayush Shridhar

Introduction

Machine learning models can be really complex and large. Even state of the art models, such as VGG-16, VGG-19, ResNet have many layers and millions of parameters. Training such models is a cumbersome process, which requires a lot of time, patience and resources. In practice, very few people train an entire Convolutional Network from scratch (with random initialization), because it is relatively rare to have a dataset of sufficient size. Instead, it is common to pretrain a ConvNet on a very large dataset (e.g. ImageNet, which contains 1.2 million images with 1000 categories), and then use the ConvNet either as an initialization or a fixed feature extractor for the task of interest. This is where *Transfer learning* comes into the picture. *Transfer Learning* focuses on storing knowledge gained while solving one problem and applying it to a different but related problem. A good example for this can be pre-trained *word embeddings,* which are pretty common in the field of *Natural Language Processing*. They provide a dense representation of words and their relative meanings. They are learned from text data, and can be reused for later applications.

Over the summers, I plan to work on a few areas in this direction. This project can be broadly divided into three parts:

A. Model Import in Julia
B. Model Export using FluxJS.jl
C. Importing Pretrained models for Computer Vision based tasks, such as object detection in Julia.

A. Model Import

Why?

Implementing a neural network from scratch can be a difficult task. By difficult, I'm not referring to coding network, but training it. State of the art networks have millions of parameters. Training such networks on a regular machine is practically impossible. We need GBs of GPU capacity and RAM in order to accomplish this task in a reasonable amount of time. Importing models made using other high performance libraries directly into Julia can save both time and energy. In other words, rather than training a number of parameters, it would be a lot easier if we could load these models, which have already been trained. We could then use these pretrained models as the starting point in our case. This eliminates the need for training them ourselves, thus saving resources.

During the summers , I plan to implement readers for two major frameworks:

1. ONNX : ONNX, or Open Neural Network Exchange is a format for interchangeable Networks and models across several platforms. I have been in regular contact with the mentor, Mike Innes. We have discussed the intricacies of this project and how further steps should be taken in this domain.

2. Keras: Keras is powerful because it's really straightforward to create a deep learning model by stacking multiple layers. When using Keras, the user doesn't have to do the maths behind the layers. This seems ideal for quick prototyping and Keras is also a popular tool in various competitions.

B. Model Export

I plan on working towards exporting Julia models to other platforms such as the browser. This involves using the *deeplearnjs* API to visualize models on the browser.

Why?

Currently, the development of Machine learning systems is often restricted to those with computational resources and the technical expertise to work with commonly available Machine Learning libraries. Exporting models to the browser can help us in both training the network and visualizing it, and by making use of powerful graphics libraries like WebGL, we can significantly speed-up the process of backpropagation for training a neural network. Moreover, it makes the process of understanding and analyzing networks much more intuitive and easy to understand. Thus, with model export we could make Machine Learning a field not only for engineers, but also for the general audience.

During the span of three months of the summers, I plan to work on implementing models to be exported to the browser, and the core package itself. We can use *deeplearnjs* for this purpose. Deeplearnjs is an Open Source library by Google PAIR team. It is an open source WebGL-accelerated JavaScript library for machine learning. My main targets would be:

1. Working on Demos for FluxJS.jl : Implementing models of different types and architectures for the model-zoo demos of FluxJS.

2. Working on the core package FluxJS.jl: The package is in development stage and needs code tracing and other improvements.

C. Models for Computer Vision tasks

Apart from working on Model Import and Export, I'd also work on a few other packages like Metalhead.jl and Detectron model import ,focusing mainly on Computer Vision and Object detection in Julia.

Why?

Functionalities of Metalhead.jl can be extended to encompass many more functions. The package currently provides an easy approach for implementing Deep Learning models for Computer vision. Integrating this with other libraries can help us in using other functionalities for our benefit. We could use pretrained models read using ONNX.jl for performing computer vision tasks in Metalhead.jl.

On the other hand, object detection hasn't been implemented in Julia. Detectron is a Facebook Research platform for object detection. Loading and running a Detectron model in Julia can help us achieve the goal of Object Detection with high degrees of accuracy. Normally, when we look at a still picture or video, we see multiple overlapping things and calculate their relative distances; relationship to each other; the fact that foreground and background objects are affected by the focal length of the camera (or the human eye); as well as where one object ends and the other begins. If your training data includes moving pics, you'll might need to look at relative size, positioning, or partial obstruction from frame to frame. These are some of the problems that ML researchers in the computer vision game –and specifically in object detection – are attempting to solve.
Over the summers, I plan on working on two projects in this direction:

1. Integrating Metalhead.jl with ONNX.jl.

2. Working on loading Detectron models for Object Detection in Julia.

Project Details:

Background

A. Model Import:

I have been working on the ONNX.jl package for a few months now. You can find my code in the FluxML/ONNX.jl repository on GitHub. Apart from this, no work has been done towards importing models from other frameworks.

How?

ONNX models are basically serialized models, which use Google's Protocol Buffers for serializing and deserializing models. These models can be read using the Julia implementation of Protocol Buffers, ProtoBuf.jl . This produces the data structure file in Julia. Now, we need to convert everything into a Julian format and trace the Julia code to produce the dataflow graph. This can be done using DataFlow.jl . With this, we can extract the structure and weights from the ONNX file. Finally, we need to build the model step by step in Julia. This successfully builds the entire ONNX model in Julia.

On the other hand, Keras models need to be read from the Keras checkpoint file, which stores all details of the model (structure, weights) in a nested dictionary format. Since it is stored in a HDF5 format, we can use HDF5.jl to read this checkpoint file. Then we can trace the code to get the structure and weights from this file. This can be done using DataFlow.jl, which allows us to treat everything as graphs, with output form one going in as the input to another. Finally, we need to build this model step by step in Julia.

B. Model Export:

Models can be exported to the browser using FluxJS.jl. FluxJS.jl makes the use of *deeplearnjs* API to make function calls. Hence, it allows us to use the power of *deeplearnjs* in our Julia models. Now, FluxJS.jl is a relatively new package and needs code tracing for a few functionalities. It can be used to generate the JavaScript code from a Julia function which accepts an array as its argument. Apart from this, Models are also needed for the FluxJS.jl model zoo. I plan to implement models of different architectures (ConvNets, LSTM) in order to show how different models can be incorporated within FluxJS.

How?

In order to generate a model using FluxJS.jl, we first need to get the JavaScript format for the model. Since a model in Flux is essentially a function, we can generate the JavaScript format of this model. This is as simple as doing:

@code_js(model(rand(10)))

@code_js is a macro and will generate the entire code in JavaScript for us.
Then, we need to create the HTML file and embed this code in it, either externally or putting the entire JavaScript code in *<script> tag*. The most important part in FluxJS.jl is perhaps the *compile* function.

FluxJS.compile("model", model ,rand(10))

Generates two files:
1. The Javascript code file, which consists the JavaScript format for the model.

2. The *model.bson* file, which is produced using *BSON.jl* and contains the weights in a binary serialized format.

Now that we have the JavaScript code and the weights, we can use them in our HTML file. I will be using *Angular*, an efficient front-end framework for implementing these models.

C. Computer Vision based tasks:

Metalhead.jl makes the task of classifying images pretty simple. It uses state of the art models to classify images into various categories. The models that can be used in Metalhead.jl currently are limited. However, this can be resolved using ONNX.jl.
On the other hand, Detectron can be used effectively for object detection. No step has been taken towards implementing a Detectron model in Julia till date. I'd love to kickstart work in this direction.

How?

Rather than training models and using them to classify images, we can use ONNX.jl to load pretrained models and use them for our tasks. This would involve integrating ONNX.jl with Metalhead.jl, so that it can use functionalities of the earlier to load and read models. Hence a function like:

model = Metalhead.load_model(pretrained_model_file)
classify(model, img)

Will classify images using the pretrained model. (Note: This will be using ONNX.jl in the background)

On the other hand, Detectron models are stored in a Python *Pickle* format. *Pickle* is essentially used for serializing and de-serializing Python data structures. Once we have the pretrained Detectron model, we need to run the model in Python. These models make use of the *Caffe2* framework and the *Coco* API. Now that we have the model up and running in Python, we need to import it into Julia. For this, we can first export the models using ONNX. An implementation of *Caffe2 to ONNX* is available for this purpose. When we have the ONNX serialized format of the model, we can import it into Julia using ONNX.jl.

Project Goals

This project involves several stages. These are

- Reading and writing the raw model (ONNX) formats.
- Model Import: Making a reader to import models from Keras into Julia.

- Model Export: FluxJS can be used to export models to the browser.
- FluxJS Demos: Implementing models to export to the browser.
- Extending Metalhead.jl functionalities.
- Loading and running Detectron models.

Implementation Plan

1. **Raw model formats**: Formats like ONNX can be read using the ProtoBuf.jl library. I have already started working on this. The code for the progress till date is hosted on Github (ONNX.jl)You can find my commits here . I'd have finished this reader before the beginning of the official coding period. Hence , I have a good idea of the package and its working. The process involved here is

   - Using ProtoBuf.jl to get the data structures from the model file.
   - Extracting information from these data structures by converting the raw format into a more general graph format, such as the DataFlow.jl graph.
   - Converting everything into a Julian format.
   - Dumping the model graph as Julia code.
   - Generating the weight and Julia file using BSON.jl.
   - Building the model step by step by reading the raw model from the Julia file and weights from the bson file.

Final Aim:

The final aim is to have functions similar to *load_model()*, *load_weights()* and *model()* which work as follows:

```
>>> using ONNX
>>> using Flux
>>> ONNX.load_model ( name_of_model )  // Creates the Julia and weight
                                                                 files

>>> weights = ONNX.load_weights() // Read and load weights from the bson
                                                                 file.

>>> model = ONNX.model() // Load the entire model by reading the weights.
```

Two files will be generated: The weight file (*weights.bson*), which will store the weights in binary format and the *model.jl* file, which contains the entire model, layer by layer.

Current Status:
As mentioned earlier, I have already started work on this. Currently, I am able to generate the code for the Julia model. However , there are some errors in this generated code. These errors are:

1. The method signatures don't match the corresponding methods in Flux. As a result, the proper method isn't being called. This leads to a *MethodError*

   MethodError: No method matching reshape(#NNlib.relu, ::Array{Float64,4},::Array{Float64,4}, Tuple{Int64, Int64})

   A way around this is redefining these signatures in our package. I'm currently working on rewriting these signatures.

2. There are a few layers that are being called without being initialized.

3. Work still needs to be done to map how a few keys like *:Constant, :LRN* map into Flux. Until this is done, we won't be able to import these models.

4. A few layers have a *stride = (0,0).* This is logically incorrect. This is also happening because of the incorrect signatures.

2. Model Export: FluxJS.jl is the Julia package which helps us in exporting models to the browser. Exporting them to the browser makes them easier to visualize, train and test. In order to do this, FluxJS makes use of DeeplearnJS API. This runs of powerful WebGL engine that makes this process of exporting a Julia model to the browser pretty fast. FluxJS is a relatively new package and needs a lot of work. There are still a lot of features which haven't been incorporated in FluxJS. This also includes working on the FluxJS model zoo. Currently , there are no model zoo examples/demos for FluxJS. The various steps involved in exporting a model are:

- Since Julia models are also functions, we use FluxJS to convert the model into the corresponding JavaScript code.
- Once we have the JavaScript format of the model, we need to embed this model in our HTML file.

```
>>> m = Chain(Dense(10, 5, relu), Dense(5, 2), softmax)
>>> @code_js m(rand(10))

        let model = (function () {
        let math = dl.ENV.math;
        function ram(waterbuffalo) {
         return math.add(math.matrixTimesVector(model.weights[0],
        waterbuffalo), model.weights[1]);
```

```
 };
 function raccoon(raven) {
 return
math.relu(math.add(math.matrixTimesVector(model.weights[2],
          raven), model.weights[3]));
 };
 function model(panda) {
  return math.softmax(ram(raccoon(panda)));
 };
 model.weights = [];
  return model;
 })();
 flux.fetchWeights("model.bson").then((function (ws) {
 return model.weights = ws;
 }));
```

- This code generated can be embedded in the HTML file, either directly using the <*script*> tag or by importing the JavaScript file.

There are also various front-end frameworks available and I will be using *Angular,* because I'm pretty proficient in it.

Final Aim:
The final aim of this will be to work on model-zoo examples for FluxJS.jl and the package itself. As far as the model-zoo is considered, I'd be implementing a few different types of Neural Networks to show the working of FluxJS in each case. These include:

1. **Abstract patterns:** Based on the paper you can find here, I plan on implementing a network which will generate random color patterns. This could be used as a header in the Flux website (the exact way *deeplearnjs* website implents it), as it goes the show the power and beauty of Neural Networks on the browser.

2. **Convolutonal Neural Net**: The starting point for this is perhaps the MNIST network. Since MNIST is pretty famous, this looks like a suitable starting point.

3. **LSTM :** A good way of showing the working of LSTM would be sentence generation. The user should be able to enter a piece of text in the browser. The network will then complete the text by adding a few sentences.

(Note: I'm willing to add more models to the model-zoo as per requirements)

3. Metalhead.jl: Metalhead.jl can be used for implementing computer vision models that run directly on top of the Flux library. Running these models in Metalhead is as simple as:

classify(vgg, img)

Which implements the pretrained VGG model to classify the image (img). Implementing and training the VGG model can be a really cumbersome due the large number of parameters. Steps involved in this process are:

- Using the already implemented ONNX.jl to load pretrained models.
- Using this newly imported model to classify images.
- Integrating ONNX.jl into Metalhead.jl, so that pretrained models can be directly imported and run in Metalhead.jl.

Final Aim:
The final aim of this is to have a function similar to *load()*, such that:

classify(load(model_name), img)

Classifies the image from the pretrained model. The key part here is integrating ONNX.jl with Metalhead.jl, so that we can use functionalities offered by ONNX.jl in the later.

5. Detectron: It'd be fun to load and show the working of one of the Detectron models. Detectron is a platform for Object Detection. This is done by implementing various powerful networks like Mask R-CNN and RetinaNet. The final aim of this would be to be able to import a detectron model in Julia and use it for object detection.

About Detectron models:

Detectron models are *Pickle* serialized models. They are made using the *Caffe2* framework , *Coco API* and have achieved state-of-the art results as far as object detection is concerned.
Generating a reader for Detectron pretrained models in Julia would involve several steps such as:

- The Detectron models are based on the Python *Pickle* format, which is used for serializing and deserializing Python data structure.
- First of all, the model needs to be build in Python.
- Once we have the model up and running, we can use the *Caffe-ONNX* import to convert this model into an ONNX format.
- Importing this newly generated ONNX format can be done using ONNX.jl.
- Finally, now that we have the Julia model, this can be used for Object detection.

Final Aim:
The final aim of this project will be to load a detectron model in Julia. After successfully loading the model, it should be able detect objects from images.

6. Keras model Import: Similar to ONNX reader, Keras reader would involve generating the dataflow graph from the Keras model checkpoint file (.hdf5 format) , which contains the structure and the weights of the entire model. The HDF5 checkpoint file contains data in a nested dictionary format. Dataflow graph generation can be done using DataFlow.jl. After generating the dataflow graph , we need to transform the complete graph to generate the Julia model.
Also, I would have to extract the weights from the file and store them in a separate *BSON* file, which the model will later read from.

Final Aim:
The final aim of this part of the project would be having a function similar to *load()*, such that:

model = Keras.load(name_of_checkpoint_file)

Loads the Julia format of the model.

Timeline:

Bonding Period (April 24 – May 13)
I plan to spend this time getting to know the community, reading and getting myself acquainted with computation graphs. I'd also finalize my plan for the next few weeks. I also plan on continuing work on the ONNX readers. My initial step would be to get this up and running for Squeezenet, which is perhaps the simplest kind of Neural network. Once this is running for Squeezenet, I can adapt the code to incorporate features for a lot of other models.

First Week (May 14 – May 21)

With the beginning of the official coding period, I hope to be able to read majority of models and incorporate them into ONNX.jl. The tougher task here is tracing code and finding out how that maps to Flux, and finally implementing it in Julia. For example, suppose several models have a *Constant* layer. In order to read this layer in Julia, we need to trace the functioning of this layer in Julia, else we would encounter a *KerError*, since there's no definition for the layer.

Second Week (May 22 – May 28)

I'll wrap up ONNX.jl by incorporating features from a lot of different models.

Third & Fourth Week (May 29 – June 12)

Start work on Keras.jl. The tougher task here is perhaps extracting data from the dataflow graph. The best way of doing this perhaps using DataFlow.jl. But, there are various problems that this presents:

- Most of the times, the generated code isn't the ideal code we wanted to have. As a result, new signatures have to be written for the corresponding functions, so as to adapt it to the generated code. This can be time consuming.
- We need to map how functions in the generated code maps to *Flux*. This is generally the problem in case of models having diverse architectures.

The easiest and most efficient way to accomplish this is by successfully loading a basic model first, and then adapting the code for other models of gradually increasing complexity. This would help me in fitting the model in accordance with other models.

Fifth Week (June 13 – June 20)

I'd finish up with Keras reader, trying to incorporate as many features as possible.

Sixth, Seventh & Eighth Week (June 21 – July 11)

Start working on demos for FluxJS models. I'd be using the *Angular* front-end framework for this purpose. The models I'll implement would be highly interactive, to show the flexibility of FluxJS. This can be time consuming, as it involves writing the code for the front-end and checking it simultaneously. Moreover, depending on the comlpexity of models, these interactive models can take time to build.

Ninth & Tenth Week (July 12 – July 26)

Get familiar with the Metalhead.jl package. Start working on enhancing and increasing Metalhead..jl functionalities. The main challenge here would be studying the way models are called and loaded in Metalhead.jl, and investigating how ONNX.jl could be integrated within this. After that, it's all about writing the appropriate functions that'll act as a a buffer between the two packages.

Eleventh and Twelfth Week (July 26 – August 11)

Understand and get familiar with the working of the Detectron platform. Implement and show the working of one of the Detectron models in Julia. For this, I'll be using the *Caffe2-ONNX* for the export and ONNX.jl for the import.

Thirteenth, Fourteenth Week (August 12 – August 26)

Completing work on importing Detectron model. Wrapping up all remaining work. Making sure that I've reached all project goals.

Previous Contributions:

I've already begun working on the ONNX reader. I collaborated with the mentor, Mike Innes to implement it. The code can be found in the Github repository, under that FluxML organization (here). Some of my major commits are:

1. Commit f5cb01d: Added the data structure file generated using Protocol Buffer compiler and ProtoBuf.jl.

2. Commit 2e507c6: Added function to retrieve data from these data structures into a Dictionary of type Dict{Symbol, Any}.

3. Commit a235bba: Designed newer data types. These were analogues to the previously generated data structures, but stored data in a more straight forward and Julian format, making them easier to use.

4. Commit 897b0d7: Made the additional changes in the new data structure format.

5. Commit 213005c: Added new functions to convert an object of a primitive type to the newly defined type. Also, made the required changes in the data formats.

6. Commit 1f02836: Added function to read the weights from the model and store them in a BSON file. Added another function to retrieve data from this newly generated binary weight file.

7. Commit d7cf11c: Added functionalities for creating a seperate *model.jl* which contains the entire Julian format for the model. This also contains the model as a lambda function. Hence, doing

   model = include('model.jl')

   Loads the model. This can be called with appropriate arguments, such as *model(rand(10))* or so.

About me:

My name is Ayush Shridhar. I am a second year Computer Science undergraduate in International Institute of Information Technology, Bhubaneswar, India. I spend most of my time contributing to open source projects and implementing machine learning algorithms. I'm confident that I can finish this project in time and meet all objectives as

- I have a good experience with open source software development.
- I'm familiar with technologies such as Git and Github.
- I have a knack for programming. I have experience of programming in C, C++, Python, Julia, JavaScript. I developed a keen interest in Julia seeing how easy and efficient numerical computing in Julia is.
- I love collaborating with others on different projects.
- I have taken online courses on Machine learning and I'm pretty good with the basics.
- I'm familiar with different machine learning libraries such as TensorFlow, PyTorch and Keras.
- I have myself faced problems implementing algorithms due to presence of different frameworks. I'd love to make machine learning more flexible by removing this barrier.
- Last but not the least, I have an excellent mentor Mike Innes who will definitely help me whenever I face any problems. I'll get a chance to learn a lot from him.

Contact:

You can contact me on different forums:

GitHub: ayush1999
Linkedin: Ayush Shridhar
Email Id: ayush.shridhar1999@gmail.com

Apart from this, you can also find me on the Julia slack channel.

References:

1. ONNX (Open Neural Network Exchange)
2. TensorFlow, Keras, PyTorch, MXNet, Caffe2
3. Protocol Buffers
4. ProtoBuf.jl
5. ONNX.jl
6. Flux.jl
7. FluxJS.jl