TensorFlow Code Generator for DNN

CSC512 Course Project

Abhisha Bhattacharyya
Computer Science Department
North Carolina State University
Raleigh NC USA
abhatt22@ncsu.edu

INTRODUCTION

Human beings are exposed to thousands of pixels of visual images every day and we analyze, label, recognize patterns and make plausible predictions about it without a lot of conscious effort all the time. The complex structure and interconnections of neurons in our brain play a major role in enabling us to process this huge volume of data effortlessly [22]. Even though we are born with this complex structure of neurons in our central nervous system a new born is not capable of labelling or recognizing patterns or making predictions. This goes to show that the ability to analyze and identify data is not just due to the structure of the brain but is acquired over the course of one's life. The way we human beings learn to identify, and label things is by example where we are told what something is and the next time we see it we can identify it.

Artificial Neural Network, which is a popular tool used in machine learning, is a class of algorithms inspired by the structure and functioning of the human brain. It involves setting up a network with interconnection of nodes which act as neurons of a brain.

A subset of machine learning techniques, deep learning involves using Artificial Neural Networks with the main the idea being computers might be able to learn actions exactly how humans learn: by example. Hence, deep learning involves creating a model where the computer is told what things are initially. This is called the training phase. Based on this training, the trained model will then be able to identify, analyze, and make predictions exactly like as a human being can.

Deep learning and deep neural network are a more refined form of Artificial Neural Networks. It is essentially a stack of layers and layers of neurons where the lowest (or first) layer takes unprocessed raw data like images, text, audio and so on. Each layer sends information to the layers above it and the higher the layer the more level of abstraction from the initial raw data.

There are several advantages to using deep learning in day-to-day applications. Firstly, deep neural networks are fault tolerant and they also scale well. Also, similar approaches work for a wide range of problems [27]. Due to all these advantages deep learning applications are being used almost everywhere from self-driven cars to medical research to automation in industries. Hence, this is an important aspect of computer science with a wide range of applications [1, 2].

A Convolutional Neural Network (CNN) is type of artificial neural network usually applied to analyzing images. A typical CNN consists of an input layer, an output layer and multiple hidden layers. The hidden layers are usually made up of convolutional layers, pooling layers, normalization layers and so on. The test files that have been used to test the compiler in this project all describe a CNN.

BACKGROUND AND MOTIVATION

Originally developed by researchers and engineers from the Google Brain team within Google's AI organization, Tensorflow [3] is the fastest growing framework when it comes to deep learning [4]. Developers using Tensorflow describe dataflow graphs which define how data travels through a network of nodes forming a graph. Each edge in the graph represents a tensor [26]. Its front-end API uses Python and it can be used to build applications which are then executed in high performance C++. Nodes and tensors in Tensorflow are recognized as python objects.

Caffe [5] is another widely used and popular deep learning framework, which focusses on the principles of expression, speed, modularity, openness and academic research in mind. Caffe also has a huge collection of ready-to-use layers. It has a useful abstraction that separates models from solvers. Models define the structure of the neural network being used for the application, and solvers define how gradient descent will be computed [25]. It has been developed by Berkeley AI Research (BAIR) [6] and by community contributors.

While both of these frameworks have their own pros and cons, when choosing between them beginners usually lean more towards Tensorflow because it creates networks using a programmatic approach and thus seem more familiar to developers who have a programming background [4]. Tensorflow also has a few other advantages over Caffe. For example, setup and deploying Tensorflow using Python pip package is much more user friendly than setting up Caffe which has to be compiled from source. Tensorflow also provides high level APIs that can be used to build models. These APIs can be invoked with a single

command, for example, "tf.contrib.keras" for using a pre-trained model.

Tensorflow also provides much better support for using GPUs and lastly Tensorflow primarily uses python which is rapidly becoming the programming language of choice for most developers [28]. Finally, since Tensorflow is becoming more popular, the users in the community discuss common problems on public platforms like stack overflow which leads to easier debugging of common issues for beginners.

The most important positive point in favor of Caffe is that it has been extensively used by the research community and so it boasts of a repository of models that developers have built for various applications and research and later shared. This Caffe model repository contains lots of pre-trained Caffe models for all types of tasks. Thus, for during development and testing of any new application finding suitable pre-trained and tested models is much easier when using Caffe's repository.

Developers would get the best of both worlds if they could take these pre-trained models from Caffe's repository and use them in the more popular and user-friendly Tensorflow applications. Hence, it would be quite useful to be able to convert Caffe models to Tensorflow and such a tool that can make this conversion effectively, efficiently, correctly and also be user friendly would be very useful indeed.

THE PROBLEM

Prototxt (protocol buffer definition file) is a text format for defining the models and training schemes of a deep neural network and Caffe models use prototxt file format to define the model architecture. The model architecture includes definition of the different layers in the deep learning network and also the connectivity between each layer. Layers can be one after the other in series or branched in parallel to one another. Thus, the first step towards converting Caffe models to Tensorflow is to convert prototxt files to python which is a file format that can be used in Tensorflow.

This project involves implementation of the Wootz compiler which tackles this first step of converting Caffe models to Tensorflow code, i.e, the conversion of prototxt model architecture files to code that can be used in Tensorflow. This will involve mapping the functionalities in the Caffe prototxt file to the corresponding Tensorflow operators. The resulting python file from this Wootz compiler can be used to define the layers of a Deep Neural Network (DNN) in a Tensorflow project.

RELATED WORK

There are a few tools available that convert Caffe models to Tensorflow code. Below are the most popular tools available.

1. Caffe to TensorFlow

This project [9] provides a simple method of converting Caffe models to Tensorflow code. They provide a convert.py python script which takes a Caffe model as input and produces a data file and a python class as outputs. The output data file contains the model's learned parameters and the Python class is used to construct the model's graph. It currently supports only a subset of Caffe layers and the input Caffe model must be in the new format. This tool has been developed by Saumitro Dasgupta from Stanford University.

2. MMdnn

Model Management Deep Neural Network or MMdnn [7] is a framework that can be used to convert DNN models between multiple popular frameworks. The tool provides a model converter that can be easily used to convert models generated on one framework to another through an intermediate representation. This tool also provides the developer with a collection of popular models to choose from, a model visualizer to display the network architecture and options to retrain the model. After training a model on a particular framework, this tool can be used to convert the model and the deploy it on a different framework. Some of the frameworks supported by this project are Tensorflow, CNTK, Keras, Caffe, PyTorch, MXNet, CoreML among others. This project has been developed by Microsoft Research [8].

3. nn_tools

Neural Network Tools or nn_tools [10] consists of Converter, Constructor and Analyser. This tool can convert deep learning models between frameworks. The analyser can be used to analyse parameters such as input_size, output_size, multiplication ops, addition ops, comparation ops, tot ops, weight size etc in the model's layers. This tool has been developed by Hahn Zhihang Yuan from Peking University.

APPROACH

The first step towards approaching the problem was to define the grammar that can comprehensively describe the code in prototxt file format. This grammar must take into account all legal commands allowed in the prototxt file format and also in Caffe models. The complete set of legal commands possible in a prototxt file is defined in caffe.proto [24]. However, for the purposes of this project only the following types has been considered: *Convolution, Pooling, BatchNorm, Scale, ReLU, Dropout, Softmax, Concat, Reshape*.

Also, the current Wootz compiler developed as part of this course project will be able to handle only those prototxt files which have layers structured in a certain format. For example, in any particular layer the following sequence of fields is legal according to the definition in caffe.proto: *bottom, top, name, type* then respective parameters for that type of layer. However, my implementation of the compiler will not be able to handle such a file.

Once the grammar was successfully defined, the initial plan was to develop the scanner, which will be able to correctly identify and tag every token in any given correct prototxt file, and the parser, which would throw errors if it finds that the given input

prototxt file does not follow the grammar defined in the previous step. However, once I started working on the project I came across Antlr [14] which is a parser generator for reading, processing, executing, or translating structured text or binary files. This tool was ideal for the requirements of the project and eliminated the need for developing a scanner and parser for the input prototxt file from scratch.

I used the Antlr tool to generate the parse tree for the input prototxt file. For doing so I downloaded the latest version of ANTLR (version 4.7.1), released in December 10, 2017. After downloading I added the Antlr 4 IDE [15] in Eclipse IDE [16] and also added the Antlr jar in eclipse library path. The code for running Antlr takes the prototxt file from Caffe as the input and using the previously written Prototxt grammar produces the parse tree for the input prototxt file.

Once I was able to get the parse tree correctly, the last piece of development was the code generator which takes the output of the parser, i.e., the parse tree for the input prototxt file, and produces the final Tensorflow code in python language. This code generator produces the final output for the whole project and took the longest time to develop.

The code generator is divided into four primary methods along with methods for the *Convolution* and *Pooling* layers, a method to get the default number of classes for the main python function in the output file and a method to write to the output file. The first important method in the code generator is the *createEndPoints* method which is responsible for traversing through all the layers in the parse tree and identifying the end points and mapping the branched layers.

The next two methods *generateCodeSimple* for generating the simple Tensorflow code and *generateCodeMultiplexing* for generating the multiplexing Tensorflow code do the bulk of the work of creating the output file. These methods traverse the whole parse tree layer by layer and execute different parts of the code depending on the type of layer. After these methods comes the *generateLogitsCode* which generates the python code for the logits layer which is the layer that feeds into the last layer of the network: usually the “Softmax” layer.

Once the issues with the simple Tensorflow code were resolved, writing the code for the multiplexing Tensorflow code was quite straightforward. For the multiplexing Tensorflow code a few lines of extra template code needed to be added to the previously written simple code. Also, the function in the python file will have and extra *config* argument.

The program takes two arguments. The first argument is the input filename. The program assumes that the input file will be present in the same folder where the .java files are present if the argument provided is only the filename without any preceding path. If the first argument provided is the filename with the full path location of the file then the compiler will pick the input file from that path. The output file is always created in the same folder as the input file.

Since the input file is always picked from the argument: whether with the preceding path or not, this eliminates the risk of having explicit file paths which might cause the compiler to fail to run on a different machine. Not using explicit path in the code also ensures that there is no chance of the compiler encountering errors due to different methods of writing paths in different OS. For example, Windows uses backslash (\) to denote directory separator while most other OS uses forward slash (/) to denote the same thing. Thus, if there was an explicit path “C:\CSC512\Project” in the code, the compiler will not face any problems running on Windows. However, this will fail on Ubuntu or any other Linux based machines because Linux uses forward slash as directory separator.

The second argument is either “simple” or “multiplexing”. If the second argument is the former then the *generateCodeSimple* method is called. If the second argument passed to the main function is the latter, i.e., “multiplexing” then the *generateCodeMultiplexing* method is called.

1. Challenges Faced

1.1 Space Token in Grammar. I faced considerable challenge while writing the grammar for prototxt as I was not sure how to start writing the grammar. After going through a few example grammars provided in online Antlr tutorials I started trying to formulate the grammar for prototxt by looking at the inception_v1.prototxt given in the project description GitHub page. Post a lot of revisions the grammar was working fine for every token in the inception_v1.prototxt file except for the space token.

I struggled a lot while trying to fix the issue of the space token. While facing this issue, I was testing the grammar by first removing all spaces in the input prototxt file and then running it through the Antlr code to generate the corresponding parse tree which can then be traversed using the code generator methods. Finally, I found that adding the below statement in the grammar ensures that all whitespaces, new lines and tabs are ignored.

“WhiteSpace: [\n\t\r]+ -> skip;”

Before finding this solution I was also accounting for all new lines in each grammar rule. This was making the resulting parse tree very large and cumbersome. Previously, I also needed to make sure there are no unexpected new lines between layers. Hence, adding this line provided a solution for multiple issues with the grammar and parse tree.

1.2 Generalization of the Grammar. Since I had initially started writing the grammar based on the single example prototxt file provided, it was not general enough to work for other prototxt files whose layer types are in the scope of this project but have slightly different structures. For example, I found a lot of Caffe models where the *Convolution* layer does not have the following fields: *bias_term*, *pad*, *stride* and *weight_filler*. To ensure that my compiler is able to handle prototxt files from such Caffe models I tweaked the grammar around. I also found an extra type in the

weight_filler: “gaussian”. Changes in grammar must be accompanied by corresponding changes in code generator because any change in the grammar changes the shape of the parse tree.

1.3 Determining the Branching. The next major challenge that I faced was determining the branching structure of the input prototxt file and finding the end points for each layer. In prototxt the layers are written one after the other with only the *bottom* field in each layer indicating the layer that feeds into the current layer. However, when writing the output python file all the parallel branches between a pair of end points must be grouped together under one variable scope. Thus, it is essential to be able to determine the correct branched structure of the layers to ensure the resulting output python file is correct.

For this challenge the solution I settled with is inefficient, but it was the only one that worked. Before starting to traverse the parse tree layer by layer and writing to output file my code traverses the entire tree once to find the end points for the Tensorflow code. During this traversal the code fills up a map data structure where the key is the layer name and the value is the list of layers that succeed the key layer. This map is used during the second traversal when the actual output file is created and populated. There are two separate bodies of code that get executed depending on whether the current layer has a single branch, or the current layer is part of multiple parallel branches between a pair of end points.

For the non-branched layers, they are traversed and corresponding Tensorflow code is written out to the output file in the same iteration. For the branched layers, from the beginning of the branching till the *Concat* layer is reached Tensorflow code is added to a String content variable. Once the program detects the *Concat* layer (which merges all the different branches into one end point) the *content* variable containing code for all of the branches are written to the output file before moving on to the next layer in the parse tree.

1.4 Layer Identification. Initially I had almost completed writing the code generator with a very wrong assumption. I had assumed that the names of the layers would be indicative of their content. Hence, I had written my code generator such that it was deciding which part of the code to execute based on the name of the layers. I later came to realize that the example file inception_v1.prototxt had such indicative names only for easier understanding. In a more generalized case, however, the names can be any strings which have no link whatsoever with the actual contents of the layer.

I had to rewrite most of my code to rectify this issue. Instead of looking for a specific string in the name I added two main checks. The new checks are as follows:

- Creating the map data structure which indicated the branched structure of the network. This data structure was then used to identify branched layers and non-branched layers.
- Using the *type* field to determine which section of code to be executed.

Post the above modifications, the code generator is able to identify the different layers correctly and call the correct section of code.

1.5 Finding Test Files: Identifying test files which have the structure that is accepted by my compiler was a major challenge that I faced. Most files I found on the GitHub pages of the projects listed in the Caffe Model Zoo have layer types which are out of scope for this compiler. Almost all the files have the “InnerProduct” layer acting as the Logits layer feeding into the last Softmax layer. A lot of the prototxt files have other types like “Eltwise” or “Data”. Some have a different sequence of fields within each layer.

After spending a lot of time trying to find suitable test files I realized that it will not be possible to find any prototxt file that will work with the version of code that I had at that time. Hence, I started with the file that was most similar to the inception_v1.prototxt and began modifications on my code to ensure it is able to handle slightly different structure. In this method I was able to identify and successfully create python version of the three test files mentioned in the Test Files section in Evaluation.

2. Assumptions

The following assumptions have been made while developing this project.

- The given input file will only have the following types: *Convolution, Pooling, BatchNorm, Scale, ReLU, Dropout, Softmax, Concat, Reshape*.
- The structure of each layer would be same as that given in the example inception_v1.prototxt file. The expected order of fields is as follows: name, type, bottom, top, parameters. Each type has its own parameter name field, for example, *Convolution* layer has *convolution_param*, *BatchNorm* layer has *batch_norm_param*, *Pooling* layer has *pooling_param* and so on.
- Average pooling layer must have stride and kernel_size fields in the pooling_param section. The grammar is written such that it expects these two fields in all pooling layers and the code picks these two values from the parse tree.
- The default argument scope assumes that each *Convolution* layer is followed by the corresponding *BatchNorm*, *Scale* and *ReLU* layers. Hence, if the *BatchNorm* layer is not present the code adds normalizer_fn=None to the respective code in the output python file. If the *ReLU* layer is not present for a particular *Convolution* layer, the code adds activation_fn=None to the respective code in the output python file. If both these layers are not present then both normalizer_fn and activation_fn are set as None in the output file.

- To avoid any explicit path for the input file which might lead to errors when the code is run on a different machine with different folder structure, the code uses only the file name without any preceding path name. Due to this the code assumes that the input file is in the same folder as the .java files. If the input filename given as argument is not present in the same folder as the .java files, the code will fail with file not found exception. The other alternative to this is to provide the input filename with the location path of the file as argument to the code. The output file is always created in the same folder as the input file. Since explicit paths are not hard-coded in the compiler, and the filename with or without the preceding path is to be picked from the user provided argument there is no chance of the compiler failing due to hard-coding of paths.

EVALUATION

Evaluation of my implementation of the Wootz compiler has been done using multiple test prototxt files most of which have been taken from the Caffe Model Zoo [11] and this GitHub repository[17]. The model zoo contains Caffe models for different tasks that have been developed by researchers and engineers for varied real-world problems ranging from simple regression, to large-scale visual classification, to robotics applications.

The initial grammar was written based on the inception_v1.prototxt and the corresponding code generator was also written based on the inception_v1_simple.py and inception_v1_multiplexing.py which were provided as example samples. However, while testing on the prototxt files taken from the model zoo I came across some minor differences in the structure of the layers. Hence, I updated the initial grammar and code generator such that my implementation of the Wootz compiler can handle more generalized layer structures.

1. Test Files

The first two test files inception_v2.prototxt [18] and xception-dw.prototxt [19] was taken from this GitHub page [17]. This repository contains a lot of models for DNNs like resnet, resnext, inception, inception_resnet, wider_resnet, densenet, etc. Some minor changes where required to be made in these test files. For example, both these files had a type: "InnerProduct" which was not present in the grammar. Since it was only one layer (the logits layer – the layer just before the Softmax layer), I changed the type of the layer to “Convolution” in both test files before testing to ensure the code will be able to handle it.

The third test file SqueezeNet.prototxt [20] was taken from one of the models listed in the Caffe model zoo. This model is based on AlexNet DNN and has been developed as part of the work for the paper SqueezeNet: AlexNet-Level Accuracy with 50x fewer parameters and <0.5MB model size [21].

The development of the compiler and the preliminary testing has been performed on a Windows 10 machine using eclipse IDE [16] running on java version 1.8.0_191. Additional testing has also been done on Ubuntu 18.04 which was running openjdk version 10.0.2.

2. Testing method

For preliminary testing and testing while development was ongoing I used the Eclipse IDE’s console. Once development was complete I wanted to make sure that my implementation of the Wootz compiler will also work outside of the Eclipse IDE. Hence I shifted my testing process to Ubuntu 18.04 OS. To make testing easy and fast on Ubuntu I created the following four shell scripts.

- init.sh – This script performs all the initialization steps. It sets up the Antlr jar path in the classpath variable, then runs the Antlr jar to compile the grammar file which produces the required java files required for using Antlr to create the parse tree. Lastly, this script also compiles all java files including the Antlr’s java files as well as java files written as part of the implementation of the Wootz compiler and creates the corresponding class files.

- run_simple.sh – This script can be used to generate the simple Tensorflow code for any prototxt file. This script takes one argument which is the name of the input prototxt file.

- run_multiplexing.sh - This script is similar to the script mentioned in the previous point. The only difference is that this script generates the multiplexing Tensorflow code for any prototxt file. This script takes one argument which is the name of the input prototxt file.

- clear.sh – This script should be run only when all testing is finished. This script cleans up the testing directory by removing all the unnecessary class files, Antlr’s java, interp and token files. After this file has been run further testing can be done only after running the init.sh.

RESULTS

The below table shows the run time of the different files for generating simple and multiplexing Tensorflow outputs

	Run Time in seconds			
Files	**1st run**	**2nd run**	**3rd run**	**Average time over 3 runs**
Initital setup				
init.sh	4.114	3.031	4.659	3.935
Simple				
inception_v2.prototxt	0.479	0.437	0.421	0.446
SqueezeNet.prototxt	0.385	0.429	0.438	0.417
xception-dw.prototxt	0.769	0.563	0.495	0.609
Multiplexing				
inception_v2.prototxt	0.462	0.409	0.504	0.458
SqueezeNet.prototxt	0.472	0.459	0.432	0.454
xception-dw.prototxt	0.841	0.470	0.852	0.721

As was expected the initial setup using init.sh took the maximum time to run. This is due to the fact that the init.sh executes Antlr's code using the grammar producing the corresponding java files and then compiles all the java files to create class files. All the prototxt files take similar time to run with the xception-dw.prototxt taking slightly more time than the other two files. However, none of the files takes too long to run even though all of them have a lot of layers.

REMAINING ISSUES AND FUTURE WORK

Even though this implementation of the Wootz compiler is complete for the purposes for this course project, in the big picture view this only completes the first step of converting Caffe code to Tensorflow code. There are more steps that need to be finished before this Wootz compiler can be used to fully convert all possible Caffe models to Tensorflow format.

To create a fully functional generalized compiler the following tasks need to be finished.

- Generalizing the grammar and the code generator such that it is able to handle all possible prototxt files. To do this the grammar must take into account all possible types of layers and all possible variations of layer structures and must follow the definition of prototxt as given in caffe.proto.
- Once all possible models are accepted, then solvers must also be converted to a format compatible with Tensorflow.

CONCLUSION

Machine learning and Artificial Intelligence are flooding every aspect of our lives right now in the form of Cortana or Siri on our phones and Alexa in our smart homes. Another avenue where deep learning is making significant progress is self-driving cars. Given how rapidly Artificial Neural Network in general and deep learning in particular has developed in the recent years, this is one aspect of computer science that will stay and grow in importance in the foreseeable future. Hence, it is a good idea to learn more about deep learning tools. This project has helped me a great deal in solidifying my understanding of Caffe models as well as Tensorflow framework.

REFERENCES

[1] https://www.mathworks.com/discovery/deep-learning.html
[2] https://machinelearningmastery.com/what-is-deep-learning
[3] https://www.tensorflow.org
[4] https://www.analyticsindiamag.com/tensorflow-vs-caffe-which-machine-learning-framework-should-you-opt-for
[5] http://caffe.berkeleyvision.org/
[6] https://bair.berkeley.edu
[7] https://github.com/Microsoft/MMdnn
[8] https://www.microsoft.com/en-us/research/group/systems-research-group-asia/
[9] https://github.com/ethereon/caffe-tensorflow
[10] https://github.com/hahnyuan/nn_tools
[11] https://github.com/BVLC/caffe/wiki/Model-Zoo
[12] https://blog.wildcat.io/2018/04/a-simple-tutorial-about-caffe-tensorflow-model-conversion/
[13] https://ndres.me/post/convert-caffe-to-tensorflow/
[14] http://www.antlr.org/
[15] https://marketplace.eclipse.org/content/antlr-4-ide
[16] https://www.eclipse.org/
[17] https://github.com/soeaver/caffe-model
[18] https://github.com/soeaver/caffe-model/blob/master/cls/inception/deploy_inception-v2.prototxt
[19] https://github.com/soeaver/caffe-model/blob/master/cls/inception/deploy_xception-dw.prototxt
[20] https://github.com/DeepScale/SqueezeNet/blob/master/SqueezeNet_v1.0/deploy.prototxt
[21] Forrest N. Iandola and Song Han and Matthew W. Moskewicz and Khalid Ashraf and William J. Dally and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and $<$0.5MB model size, *arXiv:1602.07360*
[22] https://medium.freecodecamp.org/an-intuitive-guide-to-convolutional-neural-networks-260c2de0a050
[23] https://www.digitaltrends.com/cool-tech/what-is-an-artificial-neural-network/
[24] https://github.com/BVLC/caffe/blob/master/src/caffe/proto/caffe.proto
[25] http://shengshuyang.github.io/A-step-by-step-guide-to-Caffe.html
[26] https://www.infoworld.com/article/3278008/tensorflow/what-is-tensorflow-the-machine-learning-library-explained.html
[27] https://www.linkedin.com/pulse/why-should-you-learn-deep-learning-vipin-tyagi/
[28] https://thenewstack.io/three-ways-google-tensorflow-beats-caffe/