Wootz - A TensorFlow Code Generator for DNN

Xiaorui Tang
Computer Science
North Carolina State University
xtang9@ncsu.edu

1 Introduction

This is a course project report of NCSU CSC522 Compiler Theory[1]. The main task is to develop a compiler called Wootz, which automatically generates TensorFlow (Slim) code from a Caffe Prototxt file.

2 Background and Motivation

Nowadays, deep learning is more and more widely used and there are several commonly used frameworks. TensorFlow and Caffe are the most popular ones of them[2][3]. Since there are many excellent models in both frameworks, it is very useful to know how to convert these models from one framework to the other. This project focuses on converting Caffe models to TensorFlow, where the main task is to generate TensorFlow (Slim) code from a Caffe Prototxt File.

TensorFlow (Slim)[4] is a deep learning framework from Google, commonly used for Deep Learning programming. Prototxt is a text format for defining the models and training schemes of a deep neural network (DNN) in Caffe. Its full name is plaintext protocol buffer schema (prototxt). It is defined by the Google Protocol Buffer (Protobuf) schema in caffe.proto.

Google Protocol Buffer (Protobuf)[5] has some appealing features in serving as a way to represent deep learning models: minimal-size binary strings when serialized, efficient serialization, a human-readable text format compatible with the binary version, and efficient interface implementations in multiple languages, most notably in C++ and Python. These are the reasons for Caffe to adopt Prototxt for its representation of deep learning models.

3 Related Work

There are several existing tools available to convert Caffe models to Tensorflow, such as caffe-tensorflow, MMdnn, nntools, etc. These tools can convert an existing well-trained Caffe model to Tensorflow. For example, the output of caffe-tensorflow consists of a python class that constructs the model's graph and a data file containing the model's learned parameters.

Our project only needs to consider network architecture defined in the prototxt format files. The values of model paramters (.caffemodel) can be ignored.

4 Objective

The objective of this project is to develop a complier named Wootz which which automatically generates TensorFlow (Slim) code from a Caffe Prototxt file. Unfortunately, we couldn't handle all the possible cases due to the huge number of layers and parameters defined in Caffe. But at least we could support some very common cases. We have been provided a sample prototxt file[6] and the layers and parameters inside are considered as a requirement. Layers included are Convolution, Batch Normalization, Scale, ReLU, Pooling, Dropout, Reshape, Softmax and Concat[7]. We will also find two more test cases and optionally cover the layers and parameters of them. We will implement all the basic components of a compiler which include definition of token language, grammar, scanner, parser and code generator. We also need to generate two type versions of code, which are normal and multiplexing. At the end, we will achieve the goal to automatically parse the network structures from 3 test cases and generate 2 different types of code for each of them.

5 Proposed Plan

According to the tasks, we initially plan to follow these steps:

1. Read tutorials of TensorFlow and Caffe, learn about Protobuf and Prototxt
2. Check out examples of DNN in Prototxt(Caffe) and TensorFlow, prepare three test files for the future test
3. Write a basic compiler for the conversion from a DNN in Prototxt to TensorFlow:
   a. Define the initial token language grammar of Prototxt file
   b. Define a template for TensorFlow (Slim) code
   c. Based on the properties of grammar, decide which type of parser should be used (LL(1), LR(1), etc)
   d. Refine the grammar to fit for target type of parser (e.g. convert to LL(1) grammar)
   e. Develop and test the scanner
   f. Develop and test the parser
   g. Design the inner data structure to store the network information conversion
   h. Figure out the rules and information we need to generate Tensorflow (Slim) code
   i. Develop and test the code generator
   j. Create the extended compiler which can generate multiplexing TensorFlow code
   k. Integrated test

4. Assess the project

6 Timeline

Preliminary study and preparation - Oct 25th
Prototxt and TensorFlow grammar design - Oct 31th
Development of the scanner and parser - Nov 10th
Development of the code generator - Nov 15th
Development of the extended comiler - Nov 25th
Update and debug - Nov 30th
Report, submit - Dec 12th

7 Update - Nov 8th

Up to now, I have finished all the preparation work include learning about Tensorflow, Caffe and Protobuf. As we know, protocol buffers (protobuf) are a flexible, efficient, automated mechanism for serializing structured data. The model format of Caffe is defined by the protobuf schema in "caffe.proto". And the models are defined in plaintext protocol buffer schema (prototxt) which is our focus. I have defined the language alphabet and the grammar of prototxt so that I can further use them in the implementation of the scanner and the parser. The Development of the scanner is almost done while the parser is on the way.

I select 3 caffe models as test cases (and also the example inputs) which are "inception_v1", "inception_v2" and "AlexNet_Scale". "AlexNet_Scale" is relatively simple while the other two are more complicated. The benefit to choose these three models is that all of them are very classic CNN models with both robust Tensorflow and Caffe implementation. This makes them easy to refer and test.

I basically extract the grammar from one example prototxt file directly, then refine it with several other examples. Layers are the major components of the grammar and there are several types of layers in Caffe such as Convolution, Pooling, ReLU, etc. Considering all the types comprehensively is hard at this time so what I am doing now is to develop a initial version to satisfy the selected examples.

After extraction of the grammar, I refine it to meet the requirement of LL(1) grammar. Then the scanner is simply developed based on the grammar using the similar way we did in our first homework.

Of course, I have encountered several problems. As I am a new study of CNN, understanding and considering all the layer types with their various attributes is a little difficult. So I can just define the grammar roughly as all the attributes are now classified as "attribute". But this may need to be adjusted in the further. I believe the problem can be solved after I know more about CNN and how they will be used in our Tensorflow part.

I also make the adjustment that not to define the grammar of Tensorflow code at the start but leave it in the future. The reason is that I couldn't find a uniform implementation of these different models online. I think I need to design and customize the representation by myself. It's better to be done after I finished the parser as at that time I will understand the requirements better.

Overall, it's more difficult than I thought. So I referred some existing tools like "caffe-tensorflow" and tried to get some ideas from there. The previous timeline is still working and fitting, so I don't plan to adjust the timeline at this time.

8 Implementation

We use C++ to impelent all the components of our compiler. The develoment test environment is a personal x86 architecture PC which runs Ubuntu 18.04 LTS. Some basic information is showed in the table below.

Programming Language	C++
Platform	Ubuntu 18.04 LTS
IDE	VSCode
Libraries	C++11 STL
Tools	Netscope

It's worth mentioning that we use Netscope[8], a web-based tool for visualizing neural network architectures, to inspect prototxt files, which is really helpful during our development. Actually, the way we use to present and store the network layers inside the program is quite similar to the way the Netscope uses. We will take about the details later.

8.1 Prepare Test Cases

We already have a provided simple file and we use it as our first test case. It's a prototxt file which defines a Inception Network, an important milestone in the development of CNN classifiers. The provided one is the V1 version of Inception Network, and as our compiler implementation is somewhat implemented base on this file, we decide to use a Inception Network V2 as our second test case[9]. Inceoption_v2 has similar network structure with Inception_v1 and the kinds of layers they use are almost same, which is a good reason to become our test case. But both networks are heavy, with both containing over 100 layers. This may make our initial development difficult cause we always want to solve a simple problem before a hard one. So we choose AlexNet_Scale to be our third test case[10], which is relatively simpler than Inception.

8.2 Token Language

Token language is the fundamental of scanner, so naturally it becomes the first part we need to develop. What we need to scan is the network defined in the prototxt file so that's the format we need to focus on. As we mentioned above, prototxt is based on Protocol buffers from Google, which is a language-neutral, platform-neutral, extensible mechanism for serializing structured data. It is like XML, but smaller, faster, and simpler. Google provide an official tool to read and parse this format, but we don't plan to relay on that so that we develop everything from scratch. After analyzing the structure of the prototxt file, we found the following characteristics:

- The network is composed by lots of JSON-like definitions, with one nested inside another.
- The basic characters are letters, digits and symbols like colon.
- There are strings and some of them are the names of layers and network, some of them are fixed value of particular parameters.
- Different layers may have different parameters, further leading to different values the parameters could be.

Token language should cover all the valid characters and token appear in the content. Thus, based on the characteristics we got above, we design the token language which showed in the following table:

Token	**Value**
bool	true | false
number	<digit>+ | <digit>+ . <digit>+
name	any string between (and including) the closest pair of quotation marks
pooling_method	MAX | AVE | STOCHASTIC
option	input | dim | input_shape | layer | ...
type	"Convolution" | "BatchNorm" | "Scale" | ...
weight_filler_type	"xavier" | "gaussian"
bias_filler_type	"constant"
symbol	{ | } | :

Basically, we implement all the parameter options as token <option> and thus all of them are reserved words. But they could appear inside a string because we will treat anything between a pair of quotation as string. For some parameter, they have a specific set of possible values, that's why we have tokens like <weight_filler_type>. The possible values for these token are extracted separately and all of them further become reserved words. Some of these values are string-like format with a pair of quotation, so this means they should used as values of <name>.

The complete Token Language definition is in the file "Token Language". The code related to token are located in files "Token.hh" and "Token.cc".

8.3 Scanner

After we have the token language, scanner implementation is not hard. In our project, the scanner is responsible for scanning tokens from a prototxt file one by one. If the token is a valid token in our token language, then the scanner will set proper label for the token and feed it to the parser, which is the next part of our compiler.

The most functions of scanner are defined in the files "Scanner.cc" and "Scanner.hh". Part of functions are implemented in "Util.hh" and "Util.cc".

8.4 Grammar

Grammar is a big part of a compiler, as one of the responsibility of the compiler is to check the syntax validation of input. The grammar define the structure of the content, in our case is the network defined in the prototxt file. After further inspection of our prototxt file, we found following information related to grammar:

- The network always start with its name and definition of input.
- After the input, the remaining content of the file is about layers.
- Each layer definition contains its name, type, top, bottom and corresponding parameters.
- The combinations of these parameters are complicated, but there are still some rules to follow.
- For a specific type of layer, the parameter options and their order are almost same.

As the final grammar is really complicated, we don't plan to show it here, but the full grammar could be referred from file "Grammar". The design and development of the grammar is not a one-pass job. Actually, after we have got our initial grammar, we think it may be a good fit for LL(1). Hence we need to check if it is a LL(1) grammar. We have refined the grammar by strategies like left factoring and eliminate left recursions in the grammar. We also compute the first plus set for each entries to check if it is a valid LL(1) grammar. This will be helpful in the development of parser as well.

The functions related to grammar have been implemented in the files "Grammar.hh" and "Grammar.cc".

8.5 Parser

One of the most important and difficult part of a compiler is the parser. In our project, the parser is responsible for receiving tokens from scanner, then use the grammar to check its validation. The parser drives scanner to retrieve the token from the file, while in the same time, parsing it and filling the information into inner network structure. The parser knows the information each token contains. For example, when the parser see the first <name> token, it knows that it is the network's name and set it to the network. When the parser see the next <name> token, it knows that it could only be a layer's name because we have already had the network's name. The parser also understand the parameters by looking at which type of layer it encounters. Same strategies used thought out the parser and finally the parser will build up a complete network by using the valid tokens it parses.

The main functions of parser have been implemented in the files "Parser.hh" and "Parser.cc". The functions to filling information to the network is integrated into "Grammar.cc".

8.6 Network Structure

As we could see in the final generated code, the parser should have the ability to reorganize the network and understand some basic parts of it in order to output the code. Therefor, it is necessary to have a intermediate place to store and analyze the network information rather than generated code directly after parsing the tokens.

As we mentioned earlier, the Netscope tool is so helpful that it gives me some good inspirations. What the tool does is to reorganize the layers by putting the layers with same <top> value together, merges them as one node and uses a directed acyclic graph to represent the network. We consider this is a good solution to go so we use the same strategy in our code.

First, layers with same <top> values will be gathered together. The <top> and <bottom> values of a layer in Caffe actually represent the input and output blob of one layer. By observing layers and generated code, we find that adjacent layers with same <top> sometimes output a code module together. That makes our way of organization more sense.

Second, when we generate the code, we need to iterate through the layers and output code in the order of layers from bottom to top, which means a linked structure is necessary. Although the order of occurrence of each layer almost corresponding to the order we need, it is without guarantee. Directed acyclic graph is a suitable way to perform the task.
Hence we decide to merge the layers with the same <top> together into a new data structure called "TFNode", then link them together to form a directed acyclic graph. The <bottom> value indicates the order of the layers so we could use it to link nodes even they come in with random order.
The definition of TFNode and its fully implementation is located in files "TFNode.hh" and "TFNode.cc".

8.7 Code generator

After having a very comprehensive network, we reach the last step that is to generate code. The key point of this part is to follow the rules of target file, which sounds easy but actually complex and hard to implement.

Again, we need to look at our simple tensorflow code file and try to find some ruls. Here is what we got:

- Part of the code is template code which is fixed and we don't need to change it.
- All the code the compiler generated located in a function, which named by the network's name.
- Some of the parameters of the function are fixed while the others need to vary according to the network information.
- The sequence of layers in prototxt: Convolution->BatchNorm->Scale->Relu can be represented using slim.conv2d.
- The set of layers under the name "Mixed_3d" is treated as one component of the network. Unlike the code of other layers, their code is stacked in a more structured way.
- Optional parameters like "stride" of conv2d will appear depending on the value of it. If the value is the default value 1, then it doesn't need to be generated.
- Layers are stacked one by one and they connect using the blob. In the code, the input blob of a layer should be the output of the last layer, so we need to track while generating the code.

As we have seen, so many rules are there and some many details need to be paid attention. Thus, it actually a tough work to take care of all these cases. Because of the different patterns the different layers should take, we implement our code to handle different types of layers separately. What we do is to take a further processing step to classify different types of layers. For example, if we see the sequence of layers in Convolution->BatchNorm->Scale->ReLU, we will classify the TFNode as CONV2D type. Then we will know what pattern should we use for this type of layers during the generation period.

The way we generate code is a little tricky. Instead of generate every word of the file, we simply prepare a template code file in advance. When it comes to the generate period, the generator will load in the content from the template file, then do some modification. We also use placeholder to indicate the locations that should modified or replaced later, like we use NETWORK_NAME in the template to indicate that this should be replaced later by the real network name. This strategy is actually inspired by Tensorflow, which we know uses placeholder in its network implementation.

There definitely are many other difficulties and challenges during the development. What makes this part complicated and difficult is the Concat layer, which mix several branches of layers together and generate a module code for the entire part. Find a way to tracking output and input blobs for each layer is also a interesting problem. We will talk about these difficulties later.

The most functions of this part could be found in the files "TFGenerator.hh" and "TFGenerator.cc".

8.8 Extended Code Generator

Multiplexing means that one can generate different network structures by calling the function with different arguments. This is useful as not every time we want everything fixed without any possibility of modification. In our project, we need to make the code in the Concat layer multiplex, following specific rules. If the convolution layer is the last layer of current branch, then we will not apply multiplexing, otherwise we apply to it by modifying number of output from a fixed number to a function.

Comparing to the previous part, this seems not that hard to do. We already have the template code, what we need to do are just some slight modifications. The modification made in this part could be found in the files "TFGenerator.hh" and "TFGenerator.cc"

8.9 Finish Implementation

After much of effort we make, the code generator could finally generate code as what we expect. For the provided simple, the generated code is almost exactly same with the sample file. For the test cases we find by ourselves, the compiler could scan and parse them correctly and generate code properly.

9 Assess and Test

Totally, we have three test cases, which are provided simple file "inception_v1.prototxt", and files we found online: "inception_v2.prototxt" and "alexnet_scale.prototxt". Finally, our compiler could scan and parse all the three test cases correctly and generate proper code. The code generated from the simple file is almost same with the simple code provided.

Unfortunately, as the way we design and develop the compiler relays on these three test cases, the compiler could not handle all kinds of prototxt files. Prototxt files which contains tokens that doesn't appear in the three test cases may cause the scan error. Prototxt files which contains layers that doesn't appear in the three test cases may lack code of that layer.

10 Challenges and Solution

We have met so many challenges during the development but most of them have been overcome. Here, we'd like to list several significant Challenges and their corresponding solutions.

1. Challenge: It may be ugly to hard code all the template code inside our compiler. We only need to care and focus on the code we generate and for those fixed template code, we just need to joint them with our generated code. However, we still need to make some small changes to the template code like filling in the network name. We need to find an elegant way to solve the problem.
   Solution: We prepare template files to store those template code. We use placeholder to indicate the place that need to be changed.
2. Challenge: Generating code for Concat layer is somewhat difficult as we need to have concat layer's name at first to output a line of code looks like "end_point = 'Mixed_5b', but the node containing this name information comes last during our iteration. We need find a way to solve the problem.
   Solution: It's very easy to come up with an idea that to change the place of output by changing the pointer of file writer. But it will become mess and this will leave a lot of confusion seek() code in our program. We actually find another feasible way by utilizing the encapsulation properly.
3. Challenge: When generating code for Concat layer, we first see a node with more than one ajacent nodes in our graph, which means the start of a new concat layer. Then we have several branches, and finially they will converge into a Concat type layer, which means the end of this concat layer. We need to find a proper way to iterate a oncat layer, in the order of branch.
   Solution: We just iterate branch using DFS, and return when we met a Concat type layer.
4. Challenge: Tracking input and output blobs for each layer is kind of tricky, especially comes to the case of Concat layer. We need to find a proper way to solve the problem.
   Solution: By passing a variable between functions, we track this value when generating the layer code. This is somewhat similar to the namespace we used in our programming. We treat layer's input and output as a scope and modified it when necessary.

11 Remaining issues

Although we have achieve our goal, there are still some remaining issues in our project:

11.1 Couldn't Handle Other Prototxt Files

Our compiler could now work on our test cases perfectly. But it's not the case when it comes to other prototxt files. One of the reason is that we haven't implement and support all the layer types and functions. But another important reason is that it is quite tough to handle things comprehensively. Besides the huge space of possible combinations of layers and parameters, we find that even the order of these tokens could vary on different files.

This need us to design a very robust grammar and flexible enough to extend in the future. In fact, when we need to add some new tokens or grammars into our program, we have already find it a bit hard to do, especially when tokens or grammars become more and more complex.

Possible solution maybe redesign the token and grammar, as I think the present way is not that perfect. The better I think is to have a well-design grammar which could easily cover these variation, then parse all the necessary information to build a comprehensive network, finally map the network information to Tensorflow Slim API. If the compiler has the information of one input parameter, then fill it, otherwise skip it. This may be the most proper way to solve the problem, but which I think will be big project with large amount of work to do.

11.2 Couldn't Handle Concat Layer In Concat Layer

For now, Concat layer in the form like the simple file provides could be handled correctly. But if the case become more complex, for example, if we have a concat layer inside a concat layer, the compiler will output a "concat in concat" error.

The reason of this is simply that we didn't handle this case. And the reason why we didn't handle this case is that we haven't find a proper simple for this case to show what should a correct result looks like. So as all our test cases don't have this case, we temporarily assume that the input protxtfile shouldn't have this case either.

11.3 Conclusion

By implementing Wootz, we gain not only the knowledge of compiler, deep learning, Tensorflow and Caffe, but also the valuable project experience. We find that a good framework design is critical which ensures the program could cover more cases and be extended easily. By implementing a compiler from scratch, we become familiar with the theory about token, scanner, grammar, parser, IR, etc. What's more, we already met and solve some issues that we should take care for the next time.

12 Reference

[1] Course Project Introduction: link
[2] Tensorflow Introduction: link
[3] Caffe Introduction: link
[4] Tensorflow Slim API Introduction: link
[6] Protocol Buffers Introduction: link
[6] Provided Simple: link
[7] Caffe Layers Introduction: link
[8] Netscope Introduction: link
[9] Second Test Case: link
[10] Second Test Case: link