TensorFlow Code Generator for Deep Neural Network Models - Wootz Compiler

Vivek
Computer Science, North Carolina State University
Raleigh, NC
vivek.7266@gmail.com

ABSTRACT

Deep Neural Networks (DNN) are quite prevalent nowadays and so are the frameworks in which they are developed. There are many widely used frameworks available. Generally, researchers and academic practitioners use Caffe. And TensorFlow is more common in production environment. TensorFlow is a popular deep learning library available as open source software by Google. Caffe is different deep learning framework in use for Convolutional Neural Networks, in general. Deep learning model and schemes are represented in Prototxt. Transformation of DNN models created in one framework oftentimes is required to another. Conversion of a model developed by researchers in Prototxt needs to be converted TensorFlow. In this paper, an approach to convert Caffe Prototxt model to TensorFlow model is discussed. The approach is called Wootz, a source-to-source compiler. A grammar is defined for Caffe Prototxt models. Using this grammar, lexer and parser are generated using ANTLR. A general machine learning model is created with the correct order of layers. This model is used to generate TensorFlow code. This project discusses inception model along, with test cases of ResNet and AlexNet. This project is inspired caffe-translator of apache mxnet incubator. Limitations and future improvements are also discussed.

KEYWORDS

TensorFlow, Wootz, Deep Neural Networks, Code Generator

ACM Reference Format:
Vivek. 2018. TensorFlow Code Generator for Deep Neural Network Models - Wootz Compiler. In *Proceedings of Computer Science, North Carolina State University (CSC 512).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Deep Neural Network (DNN) models have increasingly become more and more profound in researches. It is not just that with more computation power, the research has become feasible, but also because of innovations in the field. Hinton et al [9][10] has shown that training times of these models can be decreased with pre-training. There are many active researches ongoing in deep learning. It is often a hindrance of knowledge transfer from one research to another. This is mainly because of the presence of many frameworks to code for deep learning models. Theano[18], Caffe Prototxt[11], TensorFlow[1], PyTorch [16], MXNet[4] and Keras [5] are some of the frameworks available. Innovations of each of these frameworks and specialization with respect to industry level deployment or large scale computation or flexibility of research drive the choice and usage of these frameworks. It becomes of paramount importance that we are able to share these innovations. Therefore, a compiler that facilitates transform a model defined in one framework to another is essential. However, due to their different constructs, it becomes a matter of feasibility study. Here, we focus on two frameworks TensorFlow and Caffe Prototxt. Our project aims at one particular problem of conversion on DNN model defined in Caffe Prototxt to TensorFlow. The particular model that we have taken as benchmark and inspiration for this project is *Inception V1*[17]. The motive also involved taking a recent research model and a seemingly complex one. The inception modules in this model make it more complex.

TensorFlow [1] is the leading Deep Learning framework used extensively nowadays. It is a Deep Learning API created by Google and is available as open source library. It is very effective in high performance in numerical computation. It can be deployed across many platforms, such as CPUs, GPUs or clusters. It ships with inbuilt support for Deep Learning and Machine Learning. It is very efficient in production setup. TensorFlow takes benefit of the data structure *tensor*. The APIs provided with TensorFlow actually creates *flow* over theses *tensors*. Another, important thing to note for TensorFlow code is executed with a *session*. Without *session*, all actions are stored in directed acyclic graphs. However, in this paper we aim to just transform the layer from Prototxt to TensorFlow.

Caffe stands for Convolutional Architecture for Fast Feature Embedding [11]. Caffe is another Deep Learning framework extensively used by researchers and scientists. Lot of models that are produced with research practitioners are in Caffe. Caffe stores the model in prototxt. Prototxt is a text format for defining models and training schemes for Deep Neural Networks. It is defined by Protobuf (Google Protocol Buffer) [7]. Caffe is a different framework from TensorFlow, so the representation of a DNN model in Caffe Prototxt is substantially different. Certain constructs such as padding in different layers can not be supported in TensorFlow. We will see about this later and the way we handle it.

2 OBJECTIVE AND PLAN

The objective of this project is to create a compiler that can transform Deep Neural Network (DNN) model described in Caffe Prototxt to TensorFlow. Following plan was published for this project:

(1) Thorough study of TensorFlow, Caffe Prototxt and compiler tools through their documentations.
(2) Derive a grammar for parsing Caffe Prototxt language constructs.
(3) Generate a lexer and a parser for Caffe Prototxt DNN models.
(4) Parse through the abstract syntax tree created with helper of generated parser to translate language. constructs of the DNN model to TensorFlow.
(5) Feasibility study of parameters that do not have a direct conversion for default values.

The above plan aims to achieve the objective in the order it has been mentioned. Another part of the initial plan is the feasibility study of the tools and frameworks that is to be used.

3 BACKGROUND AND RELATED WORK

The objective of this project, conversion of Caffe Prototxt to TensorFlow, can be seen in multiple software projects. Some of these projects are being mentioned below:

- **MMdnn**: A comprehensive, cross-framework solution to convert, visualize and diagnose deep neural network models [14]. The "MM" in MMdnn stands for model management and "dnn" is an acronym for the deep neural network. This project supports multiple frameworks. It is built in python.
- **nn_tools**: This provides a tool for some convert models in different neural network frameworks [19]. This is built in python and like MMdnn, it supports many frameworks.
- **caffe-tensorflow**: This is the standard script used to convert Caffe models to TensorFlow models [6]. This script is also written in python.
- **Caffe Translator**: Caffe Translator is a migration tool that helps developers migrate their existing Caffe code to MXNet and continue further development using MXNet [3]. Our project is heavily inspired by this translator. Although, the transformation of Caffe Prototxt models is to MXNet, we have drawn inspiration and adapted it for TensorFlow.
- **ONNX**: Open Neural Network Exchange is built in python and aims at interoperability between different deep learning framework [2]. It's TensorFlow conversion is still under making.

4 RESEARCH METHODOLOGY

4.1 Prototxt analysis

For the purpose of this study we take "Inception V1" prototxt file. The definition of deep learning model is defined in this file. Prominently, it has all the layers of the model. There are definitions for input names and input shape. Each layer has following characteristics as described as follows:

- **name**: Describes the name of the layer. In case of, *Mixed* layers, this describes the upper and bottom layer int the branching.
- **type**: Describes the type of layer. There are following types of layers:
  - Convolution - For 2d Convolution layer.
  - Pooling - For Max Pooling and Average pooling layers.
  - ReLU - For Rectified Linear Units after each convolution layer.
  - BatchNorm - For defining Normalization.
  - Concat - For concating layers, in case of Mixed branching.
  - Scale - For scaling layers.
  - Dropout - For Dropout layer in case of Logits.
  - Reshape - For final reshape to Softmax layer.
  - Softmax - For Softmax layer mapping to Predictions.
- **top**: Describes the layer that resides on top of this layer
- **bottom**: Describes the bottom layer residing after this layer. In case of *Concat* layers, multiple *bottoms* are defined. Both *top* and *bottom* help in defining the ordering of the layers.
- **type params**: *convolution_param* and *pooling_param* are defined in case of *Convolution* and *Pooling* layers, respectively. These fields have the values corresponding to *kernel_size*, *num_output* and *stride*.

There are more fields in the layer description. For the purpose of this study, we are interested in the ones described above. Along with the description of layers and inputs, *name* of the model is also defined.

4.2 Grammar for Prototxt

The grammar that we have defined is very inspired from Apache MXNet. What we see in the definition of a Prototxt file for the deep learning model, is the definition of different layers. The symbols are as follows:

- **DIGIT** [0-9] defines the numerical digits.
- **LETTER** defines the upper case, lower case and utf encoded characters.
- **ID**, defined as name of parameters. It is generally the left hand side of all statements.
- **COLON** (':'), marks the separator of every pair.
- **STRING** is the terminal value for any string like values.
- **NUMBER** is the terminal value for any numeric like values.
- **LPAREN** ('{') denotes the opening parenthesis, generally defining a object containing more pairs.
- **RPAREN** ('}') denotes the closing parenthesis, generally defining closing of an object.
- **value** is a terminal value such as STRING or NUMBER or ID. It can also be an object.
- **pair** It contains an ID followed by a colon or a value. It is the core definition for every key value pair.
- **object** contains LPAREN and RPAREN with one or more pairs in between.
- **layer** defines each layer. It starts with "layer" followed by an object.

Rest of the symbols denote name, or is the starting rule for the grammar. The exact definitions will be shared with the code.

4.3 ANother Tool for Language Recognition - ANTLR

This tool is a parser generator for reading and processing structured texts like source code. ANTLR parser recognizes the elements present in the source code and build a parse tree metamodel. ANTLR is an easy to use parser available in language of our choice i.e. Java.

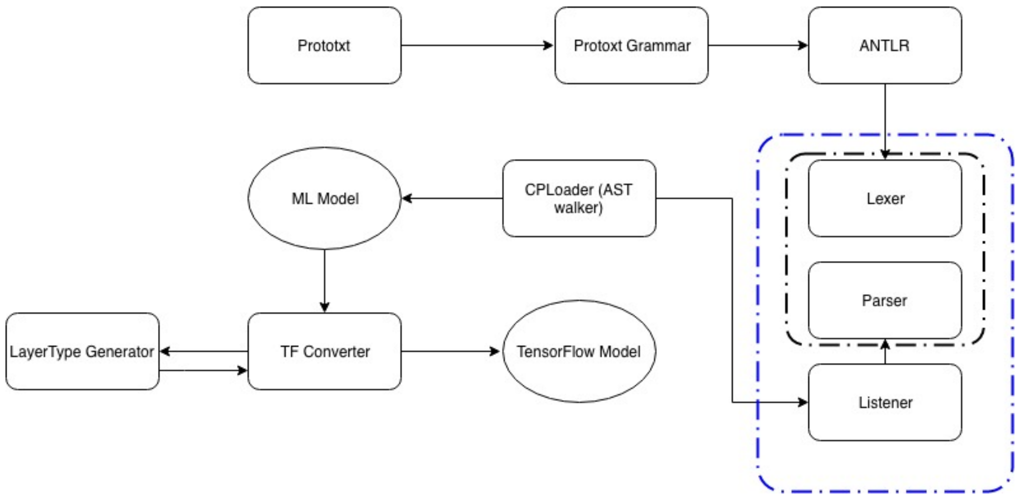

Figure 1: Wootz compiler - construction and workflow

It works on LL grammar. We have already defined LL grammar based on sample prototxt file of 'Inception V1'. ANTLR is a parser generator program that generates code to translate a specified input language into a nice, tidy data structure [15]. ANTLR can be used to language to language converter such as is our use case. We use version 3 of ANTLR that runs on Java 8 and uses maven. The main classes that we get are as follows:

- **`CaffePrototxtParser.java`**: ANTLR generated parser.
- **`CaffePrototxtLexer.java`**: ANTLR generated lexer.
- **`CaffePrototxtBaseListener.java`**: ANTLR generated listener that is used for parsing the abstract syntax tree that ANTLR will generate by use of parser. We use `ParseTreeWalker` to move through parse tree and create the DNN model.

For this study, we have installed ANTLR version 4.7 and of maven version. This helps us to structure further parts of this project that involves walking through the parse tree and generating TensorFlow code later.

4.4 Deep Neural Network Model Generation

For generating ML model, we make use of Java. A maven project is created. This project contains all the required files generated by ANTLR, mainly the lexer, parser and listener.

ANTLR generates an Abstract Syntax Tree using the grammar. The Abstract Syntax Tree (AST) is generated on top of lexer that ANTLR had earlier generated. We walk through the AST and generate a Machine Learning model in a general sense with all the layers. The main purpose of this first pass is to get the metadata associated with all the layers as java `ArrayList<Layer>`. Our implementation in Java, an object-oriented programming language that facilitates the use of creating class and pass by reference values. We will make one pass at the AST and generate MLModel. MLModel has the following attributes:

- `name`: name of the model.
- `layerList`: top and bottom matched list of all the `layer` object.
- `inputImageSize`: size of the input image
- `inputName`: name of the layer that describes the input data. This layer becomes input to the first layer.
- `layerLookup`: this is a map of `layerName` to `Layer` object. We use this on multiple occasion when we want to fetch layer object by the name.

As we exit the last `prototxt` block, we re-order the layers initially stored in the order they appear in the Prototxt description. Most of the Java classes are inspired by the apache mxnet incubator project that contains *caffe-translator* [3].

Then we make one pass through all the layers of MLModel and generate TensorFlow code. For every layer we enter, we use a Stack to keep track of the tokens in Pair. A pair as defined earlier is can be fully formed at the leaf. So we create the rule at the `exitLeaf` method in the `CaffePrototxtBaseListener` class. Figure-1 shows the process of construction of compiler and also the work flow to create the TensorFlow DNN model from a Caffe Prototxt model. The process of generating a lexer and parser through use of ANTLR is a one time operation.

5 CHALLENGES AND SOLUTIONS

5.1 Order of the Layers

Our first implementation meant creating an ordered list of layers. To determine the In this implementation, MLModel class has a an ordered list of layers. This list is generated by matching tops and bottoms of each layer. Generally, each layer is either a Convolution or Pooling, save for the other type of layers.

Finding the order of the layers required moving through the layers and determining the order from "top" and "bottom" mentioned in each layer. This is taken care by `CaffePrototxtBaseListener` class. We make a child class of `CaffePrototxtBaseListener` for our implmentation. `CPLoader` class is defined for this purpose. We instantiate a new `Layer` class as we enter a layer. Attributes of layer class are instantiated. We add this `Layer` object in the `MLModel` class attribute `layerList`. This `layerList` at this point of time contains list in the order they appear in the Prototxt file. However, our requirment is a list of layers basis the *top* and *bottom*. So, once we

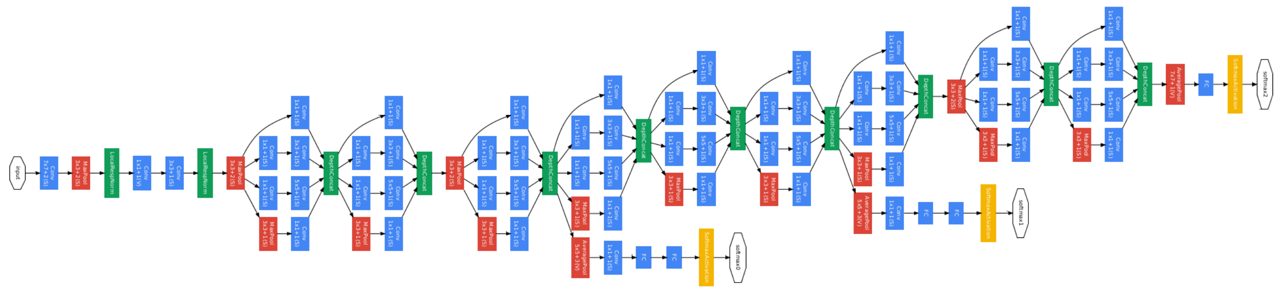

Figure 2: Inception v3 - DNN Model

exit `Protoxt`, we re-order the whole list using *bottoms* in one pass. This was facilitated by a `layerLookup` Map that we keep for every layer. At the end of this parsing, we have a `layerList` in correct order.

5.2 Inception Modules

In this Inception model, we find a different kind of layer that branches. In Figure-2, it can be seen that some branches appear in the modules. These modules are called "inception" module. The structure of these layers are a bit different. These branches go on repeating one after the other. These branches can itself be very different as we can see the *inception* module in the far right towards the output is significantly different from the one in the mid. While in Caffe Prototxt it is easier to describe this, as individual layers become just layers with different "tops" and "bottoms". However, in Tensorflow this is represented differently. Converting from protoxt to tensorflow code now becomes difficult as we need to keep track of which branch starts from which layer. The reordering of Layers basis *top* and *bottom* was helpful. This is because the next layer will be the next element in the list. However, we need to append layer information accordingly as one branch can span to multiple layers. To solve this we created a palceholder that kept the generated TensorFlow code until *Concat* layer comes, wherein we generate the final code using the placeholder information for that inception layer. Then the placeholder is reset.

5.3 Branch Variable Name in Inception Module

In Caffe Prototxt description all branch layers are also described in the similar way as it was just another layer in the whole model description. However, it has certain top and bottom, that lets us figure out the branch. Finally, a *Concat* layer instructs that the earlier defined layer, were in fact branches of inception module. This requires different scope constructs in TensorFlow. Declaring, the variable name and understanding that last branch is still in the same `variable_scope`, we maintain two global flags.

- **`lastBranchName`**: This maintains the variable scope, if there is another scope starting, we add another scope as it is decalred in TensorFlow `variable_scope` and reset this flag.
- **`lastVarCount`**: This tracks the name of the branches within the scope. We should note that the `LayerName` can not be used as variable name because it can be numeric and we can not declare a numeric variable name. This flag gets reset with the change in variable scope.

5.4 Padding

Some of the language constructs needed to be analyzed for their feasibility of conversion. For instance, Caffe supports arbitrary padding in layers, but the same is not applicable in TensorFlow. So some of the constructs such as padding have been defaulted to a particular value in conversion. We default padding to `SAME`.

5.5 ReLU, BatchNorm and Decay

In the Caffe Prototxt definitions we see that each *Convolution* layer is followed by a *BatchNorm*, then a *Scale* and finally a *ReLU* layer. We can accomplish the same thing by using TensorFlow slim. So for each convolution layer we use `slim.conv2d`. BatchNorm and decay are set in `default_arg_scope`. This is in congruence with Inception module definitions for TensorFlow.

5.6 Reshape

For *Reshape* layer, we need to convert it to appropriate TensorFlow code. `tf.squeeze` in TensorFlow is used in its transformation.

5.7 Indentation

Python works with indentation to understand the scope. As TensorFlow is written in python, it becomes increasingly important that we indent the generated code properly. For maintaining correct indentation we maintain a global variable that increases and decrease depending on scope. So, every string we append to exist line, we need to first append that amount of indentation first. Only, after indentation transformed TensorFlow code could be appended.

5.8 Multiplexing Code

Multiplexing means that one can generate different network structures by calling the function with different arguments. For this purpose, once again we need to add some changes in the branches

Table 1: Layer Translation

Caffe Prototxt	TensorFlow
Convolution	slim.conv2d
Pooling - MAX	slim.max_pool2d
Pooling - AVE	slim.avg_pool2d
Concat	tf.concat
Softmax	slim.softmax
SoftmaxWithLoss	slim.softmax
InnerProduct	slim.fully_connected
Dropout	slim.dropout
Reshape	tf.squeeze

of inception module. Even before that we need to add the templates required to read output shapes from configuration file. We maintain another flag in `ConvolutionGenerator` class that keeps track of number of layers in a particular branch. For this we keep `branchVarCount`. The top layer in each branch that can be multiplexed, we call the depth function.

5.9 Layer Translation

Layer translation required understanding of both Caffe and TensorFlow. Layer translation required a one to one mapping. Sometimes more than one line of TensorFlow code is required. The generator layer that were transformed to corresponding TensorFlow code are as in Table-1.

6 LESSONS AND EXPERIENCES

6.1 TensorFlow and Caffe Prototxt

TensorFlow and Caffe are both very popular deep learning libraries. Thousands of projects use them and malleability of these framework allows them to moulded in so many ways. For instance, inception module of Inception DNN model are flexible enough to not just stack layers but branch to different layers. As for Caffe description of this layer is as simple as any other layer in Protoxt. However, the same bunch of layers are quite differently stacked in TensorFlow, but with same flexibility. Moreover, in TensorFlow depth of inception convolution layers can be manipulated using config parameters. This project required understanding these small intricacies of well built frameworks. Learning TensorFlow and Caffe, also gave us a view of the Deep Learning field in general. The availability of these frameworks as open source helps the deep learning community. However, at the same time due to availability of multiple deep learning frameworks, sharing of important researches and breakthrough becomes difficult. A project like Wootz is the necessity of the hour to help the deep learning community to collaborate efficiently.

6.2 Selection of Software Tools

In the intial part of this project, we required to read and understand some softwares and framework. First of all, we needed to understand how to create a compiler for parsing any simple language. We explored ANTLR. ANTLR as described in earlier section came in very handy. It also required us to write an LL grammar. While choosing ANTLR, we chose Java as our preferred choice of language. This is because we understood the necessity for object oriented programming. Moreover, we could use features such as Factory Pattern, Singleton, interfaces and polymorphism. Walking through the parse tree using these features made it more convenient. Also, to manage all the dependencies in the code piece, we used maven compiler. Apache Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven manages dependencies, build goals from single source. It becomes easier to not include the jar files of all dependencies in our code and just download it when required by the project while compiling.

6.3 Use of Generators

We find that in Caff Prototxt the layers have a specific types. Each specific type of layer has different parsing and transformation for TensorFlow code. Initially, it required a lot of if conditions to properly match correctly to layer type. However, through the use of generator interface with different implementations for each type of layer. This facilitated one pass through the layers and handling each unique layer according to its implementation. To add to this, `ConvolutionGenerator` had to add multiple type of layers, like in case of inception layer, which has both convolution and pooling layers.

6.4 Input Name and Shape

Input name and shape are defined in different format different from layers. Name is parsed and save at the time of walking through the parse tree in the ML model itself. This helps us segregate the information between two kinds of layers. An intermediate layer and a data layer. A data layer is the one where information is present about the name of the model and the input dimensions.

6.5 The Impact of Compiler

A very diverse field with many communities doing research simultaneously, along with the industry adopting the the research as soon as it proves success, can not remain a collaborative environment if there is no interoperability between the tools used. While, we studied compiler in a way where it does machine level code translation from high level code. Wootz is a compiler that translates high level code to high level. In a way, it is a source-to-source compiler [13]. Source-to-source compiler enables us to build a software in a particular language and share in some other language construct. We can say that they achieve collaboaration efficiency. The power of compiler to combine two different widely accepted framework to collaborate together is something we see in this project. There are many such collaborative opportunities required in multiple frameworks that was mentioned earlier.

7 FUTURE WORK AND POSSIBLE SOLUTIONS

7.1 Segregation of Data and Non-data layers

Wootz currently keeps all the layers are non-data layer, unless specified in Caffe Prototxt as `type: "data"`. Data layer has more information sometimes, regarding the test and training phase of the model. The inputs to the model. Handling of a data layer and a non-data layer is different as they pertain to different sections of compiler code. Handling of data layer can be developed in a similar way MLModel is developed. Wootz model has the ability to extract these symbols. However, the implementation to use data layers and transform it to appropriate TensorFlow code can be done.

7.2 Different Types of Layers

Currenntly, Wootz compiler has generators for the following types of Caffe Prototxt layers:

- Convolution
- Pooling
- Concat
- ReLU
- BatchNorm
- Scale
- Dropout
- Reshape
- Softmax
- SoftmaxWithLoss
- InnerProduct

These layers correspond to the usage of in Inception [17], AlexNet [12] and ResNet [8] DNN architectures. However, there are other layer types that may be required such as, deconvolution, flatten, eltwise, permute, accuracy metrics, etc. An exhaustive case of generators and appropriate implementation in TensorFlow would enable conversion of many more deep neural networks.

7.3 Inception Module Identification

Wootz relies on the naming convention for identifying an Inception Module. If it did not rely on the name, it may not be able to identify the starting layers for Inception module. This may not be the correct way of identification. If the names are do not follow convention, then Wootz may no not identify the Inception module. It may even happen that Wootz mistakenly identifies a layer to be part of inception module. This may lead to incorrect model altogether. Possibly, the *Concat* layer may help in fixing this problem. We may work backwards from the this layer looking at all the `bottoms` that follow with the names in the bottom. And then work our way backwards following the `tops`. This may give the correct order, but requires either a Stack based approach or a recursive approach.

Since inception module identification may go wrong, this inherently implies that multiplexing code can also go wrong. However, `selectdepth` function takes care of it. But if there is a matching configuration then wrong depth may be applied at a layer. Moreover, a multiplexing may be applied to a layer that should not be. Hence, it is very important to correctly identify inception module. Identifying them on the basis of name may not be the correct approach.

7.4 Inorrect Input

Currently, Wootz is designed to parse the Prototxt file completely without any final validation. If Wootz is fed with incorrect layer information or malformed layering, not following correct top and bottom, Wootz will still parse it, as long as the input Prototxt file complies with the grammar. We need to add a logical validation to throw compiler error, if the layers do not follow the language constructs that require code transformations to TensorFlow. A quantifiable output like the number of inception modules, of number of layers, number of unique layers, etc can be employed for sanity and the compiler can throw an error saying what went wrong even though the file complied to correct grammar.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[2] ONNX Amazon/Facebook/Microsoft. 2017. ONNX. https://github.com/onnx/onnx.

[3] Apache/MXNet. 2015. Apache MXNet (Incubating), Caffe Translator. https://github.com/apache/incubator-mxnet/tree/master/tools/caffe_translator.

[4] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR* abs/1512.01274 (2015). arXiv:1512.01274 http://arxiv.org/abs/1512.01274

[5] FranÃğois Chollet. 2015. keras. https://github.com/fchollet/keras.

[6] Saumitro Dasgupta. 2016. caffe-tensorflow. https://github.com/ethereon/caffe-tensorflow.

[7] Google. 2008. Protocol Buffers - Google's data interchange format. https://github.com/protocolbuffers/protobuf.

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *CoRR* abs/1512.03385 (2015). arXiv:1512.03385 http://arxiv.org/abs/1512.03385

[9] Geoffrey E. Hinton. 2007. Learning multiple layers of representation. *Trends in Cognitive Sciences* 11, 10 (2007), 428–434.

[10] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. 2006. A Fast Learning Algorithm for Deep Belief Nets. *Neural Computation* 18, 7 (2006), 1527–1554.

[11] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proceedings of the 22Nd ACM International Conference on Multimedia (MM '14)*. ACM, New York, NY, USA, 675–678. https://doi.org/10.1145/2647868.2654889

[12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2017. ImageNet Classification with Deep Convolutional Neural Networks. *Commun. ACM* 60, 6 (May 2017), 84–90. https://doi.org/10.1145/3065386

[13] Sang-Ik Lee, Troy A. Johnson, and Rudolf Eigenmann. 2004. Cetus – An Extensible Compiler Infrastructure for Source-to-Source Transformation. In *Languages and Compilers for Parallel Computing*, Lawrence Rauchwerger (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 539–553.

[14] Microsoft. 2017. MMdnn. https://github.com/Microsoft/MMdnn.

[15] Terence Parr. 2007. *The complete ANTLR reference guide.* Pragmatic ;O'Reilly [distributor], Farnham;Lewisville, Tex;.

[16] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).

[17] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2014. Going Deeper with Convolutions. *CoRR* abs/1409.4842 (2014). arXiv:1409.4842 http://arxiv.org/abs/1409.4842

[18] Theano Development Team. 2016. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints* abs/1605.02688 (May 2016). http://arxiv.org/abs/1605.02688

[19] Hahn Yuan. 2016. nn_tools. https://github.com/hahnyuan/nn_tools.