Neural Network Syntax Analyzer for Embedded Standardized Deep Learning

MyungJae Shin
Chung-Ang University
Seoul, Republic of Korea
mjshin.cau@gmail.com

Joongheon Kim
Chung-Ang University
Seoul, Republic of Korea
joongheon@cau.ac.kr

Aziz Mohaisen
University of Central Florida
Orlando, FL, USA
mohaisen@ucf.edu

Jaebok Park
ETRI
Daejeon, Republic of Korea
parkjb@etri.re.kr

Kyung Hee Lee
ETRI
Daejeon, Republic of Korea
kyunghee@etri.re.kr

ABSTRACT

Deep learning frameworks based on the neural network model have attracted a lot of attention recently for their potential in various applications. Accordingly, recent developments in the fields of deep learning configuration platforms have led to renewed interests in neural network unified format (NNUF) for standardized deep learning computation. The attempt of making NNUF becomes quite challenging because primarily used platforms change over time and the structures of deep learning computation models are continuously evolving. This paper presents the design and implementation of a parser of NNUF for standardized deep learning computation. We call the platform implemented with the neural network exchange framework (NNEF) standard as the NNUF. This framework provides platform-independent processes for configuring and training deep learning neural networks, where the independence is offered by the NNUF model. This model allows us to configure all components of neural network graphs. Our framework also allows the resulting graph to be easily shared with other platform-dependent descriptions which configure various neural network architectures in their own ways. This paper presents the details of the parser design, JavaCC-based implementation, and initial results.

CCS CONCEPTS

• **Computing methodologies** → **Machine learning**; *Artificial intelligence*; • **Information systems** → Information systems applications; • **Software and its engineering** → Software notations and tools;

KEYWORDS

Deep learning, machine learning, neural network unified format, TensorFlow, standardization

ACM Reference Format:
MyungJae Shin, Joongheon Kim, Aziz Mohaisen, Jaebok Park, and Kyung Hee Lee. 2018. Neural Network Syntax Analyzer for Embedded Standardized Deep Learning. In *EMDL'18: 2nd International Workshop on Embedded and Mobile Deep Learning , June 15, 2018, Munich, Germany*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3212725.3212727

1 INTRODUCTION

Theories, techniques, and applications of deep learning have recently witnessed an outburst, resulting in a large number of investigations that looked into the effect of deep learning methods on various learning tasks and domains. Organizations are increasingly interested in the potential of deep learning techniques' research and development, and many have started adopting those techniques to obtain more accurate machine learning results using data-driven supervised approaches in multiple domains and fields [2]. The large number activities surrounding deep learning research [12] and development of this research have resulted in a large number of platforms, which help programmers to simply configure and train neural network architectures. As more platforms are developed every day, more platform-specific workloads to utilize those platforms are required. To alleviate this issue, the standardization of neural network descriptions would be necessary to make use of advances in deep learning approaches. Particularly, it becomes important to shift the focus from the development of new frameworks into a standardized formats that allow defined neural networks to simply transform to platform-dependent descriptions.

From an application viewpoint, neural networks can be utilized in (i) systems for solving pattern recognition tasks, (ii) models for understanding biological neural systems, (iii) systems to characterize parallel computing architectures, and (iv) models that capture behaviors in physical systems [10], among others. However different they are, those applications can be abstracted to a standard format, and treated with the same approach. To this end, in this paper, we focused on one such an approach: neural network that conduct predictions and classifications, especially convolutional neural network. Neural network models are defined by various pieces of information, such as input variables, output variables, and network graph. It is important to note that the neural model consists of information other than the neural network itself [10]. This paper further introduces neural network unified format ideas and proposes to standardized deep learning computation frameworks.

```
//Tensorflow
x=tf.placeholder(tf.float32)
y=tf.placeholder(tf.float32)
z=tf.placeholder(tf.float32)
a=x * y
b=a + z
c=tf.reduce_sum(b)
with tf.Session as sess:
values = {
  x:np.random.randn(3, 4),
  y:np.random.randn(3, 4),
  z:np.random.randn(3, 4),
}
```

```
//Pytorch
x=Variable(tourch.randn(3,
     4).cuda(),
     requires_grad=True)
y=Variable(tourch.randn(3,
     4).cuda(),
     requires_grad=True)
z=Variable(tourch.randn(3,
     4).cuda(),
     requires_grad=True)
a=x * y
b=a + z
c=torch.sum(b)
```

Figure 1: Difference of structures at each platform.

Reference Model. In the literature, several neural network definition models have been created, including models using XML type language. Among XML type languages is the neural network markup language (NNML), which aims to develop and provide reconfiguration of neural networks. As shown in [8], neural network can not be entirely reconfigured without information about the environment in which it was created (i.e., application). In other words description of neural network must contain information about the structure of data dictionary, preprocessing methods, postprocessing methods, and additional information about the use model [8, 10]. In this model, it is necessary to assign (trained) values of variables at the time of defining the neural network model through NNML. NNML can lead to compact code that represents neural networks.
Contributions. The contributions of this work are as follows. First, we introduce the design of a parser of NNUF for standardized deep learning computation, called the standard NNEF, which provides platform-independent processes for configuring and training deep learning neural networks. Second, through implementation and demonstration on a Raspberry Pi, we show that our proposed design allows the resulting learning graph to be easily shared with other platform-dependent descriptions.
Organization. The organization of the rest of this paper is as follows. In section 2, we introduce an overview of the standard NNUF. In section 3 we review the standard framework, including design rationale. In section 4 we review the implementation of our standard NNUF. In section 5, we draw concluding remarks and outline our future work.

2 NNUF: AN OVERVIEW

As mentioned earlier, neural networks are adopted as a tool to obtain best-results from data in broad fields [2, 11, 12]. Given the unprecedented volumes of application-specific data delivered daily, it is ideal to use machine learning in many of those applications. However, using advances in developed machine learning tools in those fields is nontrivial. For example, neural networks built on Python-based artificial neural network platforms cannot be implemented without the platform-dependent features.

As shown in Fig. 1, we observe that the same neural network is implemented very differently based on the framework. The more complex the neural network operations being used, the greater is this difference. As a result, when a programmer implements the same model to take advantage of each framework, it becomes necessary to understand the framework from scratch as a baseline [5, 12].

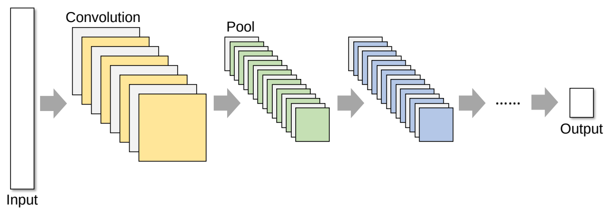

Figure 2: NNUF neural network model representation example, convolutional neural network (CNN).

The neural network model description in NNUF focuses on this problem. In the implementation of the model description using NNUF, we first have to create a model and add values separately. The input of the data is delegated to the framework in which the model is used, so that the model is not dependent on the input data. This method is similar to the model implementation in TensorFlow. We call the platform implemented according to the neural network exchange framework (NNEF) the standard NNUF [6].

In previous neural network XML-based unified model, the focus has been on correct reconstruction of computational models of the neural network [10]. However, for correct reconfiguration, information that is not actually required in the processing platform must be filled in the model description. If the data pre-/post-processing and training are delegated to other frameworks, it may become more appropriate to introduce methods to reconstruct the artificial neural networks to work with implementation of those frameworks.

The strength of standardization can exert its power at the construction phase of a neural network. Through standardization, NNUF builds an artificial neural network in a form that is independent of the used frameworks, such as Tensorflow and Pytorch, stores the network in a protocol buffer format, and transfers it to various frameworks. With standardization, we do not need to be aware of the specific methods of the frameworks when deploying artificial neural networks in multiple environments. Furthermore, with such a standardization, it is easy to implement operations commonly used in neural networks (such as relu, tanh, etc.), as well as new operations based on the functional format of NNUF.

The neural network model in Fig. 2 shows the neural network unified format grammar. The structure is based on a neural network called the convolutional neural network (CNN), an artificial neural network with multiple hidden layers; the multi-layer nonlinear structure provides it with a powerful feature expression ability. The CNN contains three kinds of basic structures, the convolutional layer, the pooling layer, and the fully-connected layer [3, 7]. Note that the NNUF was intended to construct the CNN.

3 STANDARDIZED FRAMEWORK

In this section, we show the design rationale of NNUF software, which is based on the concept of "*Define and Run*". We discuss the base model of the NNUF, which uses an XML-like format.

3.1 Design Rationale

The main purpose of the unit description rule in NNUF is to simplify the representation of the underlying mathematical procedures.

```
// Neural network definition
graph CustomNet(inputVar) -> (output Var) {
    // Variables definition
    Input = reshape(inputVar, [-1, 28, 28, 1] );
    Kernel = variable(shape=[3, 3, 1, 64],
         label="conv1/kernel");
    Bias1 = variable(shape=[64], label="conv1/bias");
    // Operations definition
    conv1 = conv(input, filter=kernel1, strides=[1, 1,
         1, 1], padding="SAME");
    add1 = add(conv1, bias1);
    outputVar = relu(add1);
}
```

Figure 3: NNUF description of neural network model.

Figure 4: Neural network exchange framework.

Since there is a lack of uni ed standard terminologies for neural networks implementations, we have decided not to use existing terms. Therefore, terms are de ned and used in a form that is intuitive and understandable by humans; e.g., add and sub.

The declaration of variables and computations of the neural network model are represented as functions. The input and output values of the neural network are speci ed in the form of the parameter of graph function, and de ne these declarations in an internal implementation. There are no special order constraints on the structure of these declarations in graph function, therefore the network structure is explicitly determined as a combination of declarations. This procedure is illustrated in Fig. 4.

In this work, we designed and implemented a framework that con gures and trains various types of models using NNUF. Furthermore, we design a parser with JavaCC for understanding codes based on NNUF; used as the core structure of neural network exchange framework. When switching to TensorFlow code through the NNUF parser, the training done by the appropriate data set is fed into the model; then, the trained model is stored according to the protocol bu er structure through the framework. When the neural network de ned by NNUF is fed into the framework, the framework generates les as parsing results as follows:

- Full TensorFlow Code: A code that can perform framework exchange based on the model de ned by NNUF is generated. It is fully converted to the code of the neural network, which is implemented by TensorFlow. In this code, pre-processing of data and training options can be lled through TensorFlow format. Then, the training can be performed in a wat similar to writing codes within the pure TensorFlow framework.

```
SKIP: { "" | "\r" | "\t" | "\n" }
TOKEN:{
  <IDENTIFIER: (["a"-"z", "A"-"Z"])
           + (["a"-"z", "A"-"Z", "0"-"9", "_"])*>
  |<METHOD: (<IDENTIFIER> ("." <IDENTIFIER> ("("
       ")"|"["(["0"-"9"])* "]")*)+) >
  |<NUMERIC_LITERAL: (["+", "-"])? (["0"-"9"])+("."
       (["0"-"9"])+)?(["E", "e"] (["+", "-"])?
       (["0"-"9"])+)? >
  |<STRING_LITERAL: ("" | "\"") (["a"-"z", "A"-"Z",
       "/", "_", "0"-"9"])* ("     ' " | "\"")>
         ...
  |<SEMI_COLON: ";">
  |<QUESTION: "?">
  |<ARROW: "->">
}
```

Figure 5: Lexical analysis (.jj le format).

```
String argument():
{String arg, exp, name, res; Token id;}
{
  (
    ((id=<IDENTIFIER>)<ASSIGN>(exp=expression())){
      name = id.toString();
      switch(name) {
        case "label": res="name";arg+=res+"="+exp; break;
        case "filter": arg+=exp; break;
        case "size": res="shape="+exp; arg+=res; break;
        case "type": res="dtype=tf."+exp;arg+=res; break;
        default: arg+=name+"="+exp; break;
      }
    } | (exp = expression()){ arg += exp; }
  ) {return arg;}
}
```

Figure 6: Syntax analysis (.jj le format).

- Trained Model Data: The training is performed with the converted model, where the input data and pre-processing processes are provided. The trained model is then stored in the form of a protocol bu er in TensorFlow.

3.2 Parser Implementation with JavaCC

The neural network de ned by NNUF is required to analyze whether the given input is grammatically and semantically correct or not. For this analysis, we implemented a parser that analyzes the graph description de ned by the grammar of NNUF. This NNUF parser is implemented using JavaCC [1], which is one of well-known lexical analysis and parsing (syntax analysis) tools based on Java programming language. JavaCC is fundamentally based on the concept of top-down parsing. The parser generation using JavaCC is done with pure Java and Java Virtual Machine (JVM) code. Therefore, Jython 2.7 is used for integrating Python and Java codes, and the overall framework structure is implemented with python syntax using the Jython. The major components of the compiler generation using JavaCC is jj le, which is based on language formal speci cation and JavaCC grammar [1, 11, 14]. In the NNUF parser implemented with JavaCC, the jj le have the following items:

- Lexical Analysis Rules: This information can be abbreviated. If it exists, it can be one or some of SKIP, TOKEN, SPECIAL TOKEN, and MORE. The meaning of them is available in JavaCC syntax description [1, 14]. In Fig. 5, an example lexical analysis code implemented using JavaCC is provided.

Figure 7: A snapshot of the graphical user interface (GUI) of, capturing the NNUF graph and TensorFlow graph de nitions.

Syntax Analysis Rules with Enhanced Backus Naur Form (EBNF): Each syntax grammar in NNUF speci cation can be implemented with EBNF as presented in Fig. 6 (le name nnuf.jj).

As shown in Fig. 6, the NNUF parser takes the results of the grammar and the token (generated by lexical analysis). According to the results, the NNUF parser matches the code to be converted one and returns it. The input NNUF graph is transformed into the TensorFlow code according to the above procedure. Finally, our standardized deep learning computation framework gets the input as NNUF-based le and returns the code which can be executed by TensorFlow computation engine.

4 IMPLEMENTATION

In this section, we outline our NNUF parser implementation in embedded open-source platforms. The implementation of one of the most popular image classi cation CNN models, the AlexNet, is introduced in Section 4.1, and the demonstration of the implementation on an embedded Raspberry Pi is presented in Section 4.2.

4.1 AlexNet/CNN Implementation

The neural network model, e.g., example model in Fig. 2, for the de nition of neural network uni ed the format grammar. The structure of our example model is based on AlexNet [7], based on CNN. The AlexNet is a popular neural networks that is specialized in image classi cation. AlexNet consists of eight layers and perfect connection layers. The output of the last fully connected layer in AlexNet is the probability distribution for 1000 image classes [7].

Based on the NNUF-based standardized parsing, AlexNet, which is one of well-known deep convolutional neural network models, is implemented [7]. The AlexNet is specialized in image classi cation where it consists of eight layers and fully-connected layers. Based on the NNUF-based AlexNet implementation, MNIST dataset based executions are performed. As an input MNIST datset, we used 28-by-28 pixels as shown in Fig. 8(a). After successful computation with our NNUF-based standardized parser, the activation map for the given MNIST dataset input are successfully obtained as illustrated in Fig. 8(b).

(a) MNIST dataset initial input

(b) Activation map for the MNIST dataset input

Figure 8: The MNIST dataset inputs (Fig. 8(a)) and the computation result as an activation map via NNUF-based AlexNet (Fig. 8(b)).

4.2 Demonstration in Linux/Raspberry Pi Embedded Platforms

The prototype of NNUF is implemented using Jython and TensorFlow in Ubuntu/Raspberry Pi embedded platforms, as shown in Fig. 9. Our standardized deep neural network learning computation framework dedicates the pre-processing, post-processing, and training to TensorFlow. For our demonstration, we employ AlexNet with the MNIST ne-grained sample image data sets. The description