Received: 23 April 2018 | Revised: 25 March 2019 | Accepted: 20 May 2019

DOI: 10.4218/etrij.2018-0135

ORIGINAL ARTICLE

WILEY ETRI Journal

Interworking technology of neural network and data among deep learning frameworks

Jaebok Park | **Seungmok Yoo** | **Seokjin Yoon** | **Kyunghee Lee** | **Changsik Cho**

Artificial Intelligence Research Laboratory, Electronics and Telecommunications Research Institute, Daejeon, Rep. of Korea

Correspondence
Jaebok Park, Artificial Intelligence Research Laboratory, Electronics and Telecommunications Research Institute, Daejeon, Rep. of Korea.
Email: parkjb@etri.re.kr

Funding information
Institute for Information & Communications Technology Promotion; Korea Government (MSIP), Grant/Award Number: 2017-0-00068

Based on the growing demand for neural network technologies, various neural network inference engines are being developed. However, each inference engine has its own neural network storage format. There is a growing demand for standardization to solve this problem. This study presents interworking techniques for ensuring the compatibility of neural networks and data among the various deep learning frameworks. The proposed technique standardizes the graphic expression grammar and learning data storage format using the Neural Network Exchange Format (NNEF) of Khronos. The proposed converter includes a lexical, syntax, and parser. This NNEF parser converts neural network information into a parsing tree and quantizes data. To validate the proposed system, we verified that MNIST is immediately executed by importing AlexNet's neural network and learned data. Therefore, this study contributes an efficient design technique for a converter that can execute a neural network and learned data in various frameworks regardless of the storage format of each framework.

KEYWORDS
AI, AlexNet, Caffe, CNN, deep learning, interworking, neural network, NNEF, parser, Tensorflow

1 | INTRODUCTION

Deep learning technology based on artificial neural networks is being actively studied and it is likely to be applied extensively in the future. However, because deep learning frameworks have individualized structures depending on the application field, they must be structured and implemented using individualized methods for each application field. Because such requirements lead to high costs in terms of the reusability and maintenance of software and code, a standardized data structure is required.

Nowadays, deep neural networks support various neural networks (such as AlexNet and LeNet) and an artificial intelligence system is implemented using a neural network suitable for the application. Several deep learning frameworks (such as Tensorflow and Caffe) can be supported by a system's GPU acceleration and operating system parallel support as well as large/small systems. Therefore, we need an interworking support framework that converts pre-created learning data and neural networks into a framework that can be supported by the device of interest.

Internationally, discussions are being held on standardizing methods for structuring and visualization and one of the results is the development of a method called the Neural Network Exchange Format (NNEF) [1]. NNEF is a standardization method for neural networks. A neural network graph defined using NNEF can be easily exchanged into various neural network configuration platforms.

In all deep-running frameworks, including Tensorflow, artificial neural networks are represented using computational graphs. This is similar to expressing the data (tensor) transferred between two nodes. However, there is a difference

in the manner in which each framework expresses the computational graph. NNEF follows a simple process that consists of learning neural networks through a consistent format. This has a major impact on the construction of neural networks used in the cross-platform. The purpose of NNEF is to represent a neural network computational graph in a unified format instead of in a separate format for each deep learning framework. In different deep learning frameworks, such as Tensorflow, artificial neural networks are represented using computational graphs. However, because the method of expressing computational graphs differs for each framework, a standardized method is required to implement neural networks in a unified format.

In this study, we propose a method that ensures interworking between an artificial intelligence system and a neural network processing system by supporting the interoperable neural network standard format established by Khronos [2]. We developed a parser for neural networks and data based on NNEF to achieve a convenient configuration for the neural networks used in a cross-platform. This study shows that conversion of a neural network and data can be performed using AlexNet [3] directly through our proposed converter. Thus, this study provides a design method and implementation technology for interoperable standard formats and interworking among neural network processing systems.

The rest of this paper is organized as follows. In Section 2, we discuss some basics and previous studies related to this research. Section 3 introduces an inference engine and learning data model for deep learning. In addition, we introduce the structure and design technologies of Khronos' NNEF. Section 4 presents the implementation methods of the proposed interworking framework. Section 5 shows that our framework is driven quickly using real neural networks and learning data. Finally, Section 6 concludes this paper and Section 7 highlights areas for future research.

2 | RELATED WORKS

Machine learning technology using deep neural networks is extremely important because it surpasses human performance in many areas. Owing to the particular attention being paid to artificial neural networks, several approaches have been developed to handle inference steps that are executed on inference engines by constructing and training neural networks [4].

Inference technologies using cloud and learning generally employ cloud-based inference engines such as Google's TPU, but they use similar hardware (mostly the GPU). In contrast, inference techniques for devices at edge points rely on optimized hardware accelerators and require special optimization techniques [5,6].

Several deep learning frameworks are currently known and in this section, we will briefly review the most commonly used frameworks. First, Tensorflow [7], which was developed by the Google Brain team, was made open source in 2015. Tensorflow is available for multiple CPUs and GPUs on all platforms, desktop and mobile, as a Python-based library [8]. In addition, Tensorflow can support other languages, such as C++ and R, and can directly create deep-running models. It can write models using a wrapper library such as Keras.

Caffe [9] is among the earliest developed deep learning frameworks; it was developed primarily at the Berkeley Vision and Learning Center (BVLC). It is also a C++ library with a Python interface, which it uses as a default application when modeling a convolutional neural network (CNN) [10]. One of the key benefits of using this library is that it can directly use many pretrained networks from the Caffe Model Zoo. Facebook [11] released a lightweight modular deep learning framework, Caffe2, to build a high-performance open learning model using Caffe.

Torch [12] is a Lua-based deep-running framework developed by large players such as Facebook, Twitter, and Google. Its parallel processing uses the C/C++ library and CUDA [13,14] for GPU processing. In addition, Torch's Python implementation, called PyTorch [15], is gaining popularity and is rapidly being adopted.

Theano [16] is very useful for numerical calculations with CPUs and GPUs. It has a low-level library and can simplify processes by directly creating a deep learning model or by applying the wrapper library on top of it. However, unlike other extended learning frameworks, it is not scalable and lacks support for multiple GPUs.

Keras [17] was developed as a simplified interface for efficient neural network construction and can be configured to work with Theano or Tensorflow. It is written in Python and is very light and easy to learn. Its greatest advantage is that it can be used to create neural networks from a few lines of code. Table 1 shows the features of each of these described frameworks.

In recent years, there has been a growing demand for neural network technology, which has resulted in an increasing interest in its standardization. An application programming interface (API)-independent file format standard has been established for data exchange between deep learning systems and interface engines. NNEF and ONNX [18,19] support deep-learning interworking technologies [20–22] that can easily transform neural network graphs into other neural network configuration platforms.

ONNX is an open source library and acts as a serialization format that encodes and decodes in-depth learning models. ONNX is supported on Apache MXNet [23], PyTorch, TensorRT [24], and other well-known in-depth learning frameworks. In contrast, NNEF was developed by a group of non-profit standards organizations that include

TABLE 1 Features of each framework

Framework	Interface	Language	Platform	CUDA	OpenMP	OpenCL	RBM/DBNs
Tensorflow	Python, (C/C++ public API only for executing graphs)	C++, Python, Java	Linux, Mac, Windows	Y	N	SYCL Tri-sycl tf-coriander	Y
Caffe	C++, Command line, Python, MATLAB	C++, MATLAB	Linux, OS X, AWS Mac	Y	N	Roadmap only	Y
Torch	Lua, LuaJIT, C, C++	C, Lua	Cross-Platform	Y	Y	Under development	Y
Theano	Python	Python	Cross-Platform	Y	Y	Under development	N
Keras	Python	Python	Linux, Mac, Windows	Y	Y (Theano) N (TF)	Roadmap only	?

many hardware and software developers as members; companies or universities can participate in the standardization process through a proven multicompany management model. NNEF is a popular standardization method for neural networks. The neural network graph defined by NNEF can be exchanged with several other neural network configuration platforms.

Representatively, the Khronos Group has attempted to connect software with hardware and proposed an open standardization method to abstract hardware details on which software developers can work on a single platform. For example, the OpenVX [25] standard provides graph-based APIs for accelerating computer vision hardware, including neural-based systems, by bringing trained networks into the graph. A common example of NNEFs is the transformation of a neural network between corresponding inference engines using a framework such as Caffe or Tensorflow. The purpose of NNEF is to make available a source code to generate neural networks with a hierarchical design in an intuitive manner. We primarily use it to load trained networks for inference using NNEF. Many research groups, including the Khronos Group, are involved in preparing a standard exchange format to link learning frameworks with inference engines. NNEF makes it easy to access all frameworks, thus allowing engineers to support a cross-platform in an open extensible transport format. However, the NNEF standard is still being defined in global research.

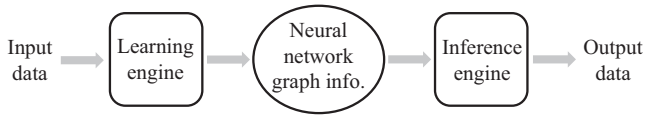

FIGURE 1 Separated learning and inference systems

3 | NEURAL NETWORK INTERWORKING ARCHITECTURE

The process of an artificial intelligence neural network can be roughly divided into a learning engine and an inference engine for determining output data from given input data, as shown in Figure 1. The learning engine determines the operating functions and parameters in the neural network so that the user can generate the desired output through sample input data. The inference engine performs a series of processes that can generate output data from new input data using the neural network structure information learned through the learning engine.

Most learning and inference engines consist of a single set. Each of them can be separated but the structure of the storage method of the learned neural network, which depends on the product used, developer, and other factors, may be different between the learning engine and inference engine. Therefore, various neural network inference engines are being developed. Each inference engine has its own neural network storage format.

To solve this problem, an interworking framework is necessary between the learning system model and inference model. Figure 2 shows the current network format structure, interworking problems, and the need for a neural network format.

The standards organization, Khronos Group, established the NNEF to solve this problem. NNEF enables collaboration between the learning engine and inference engine by defining a standardized neural network structure storage format.

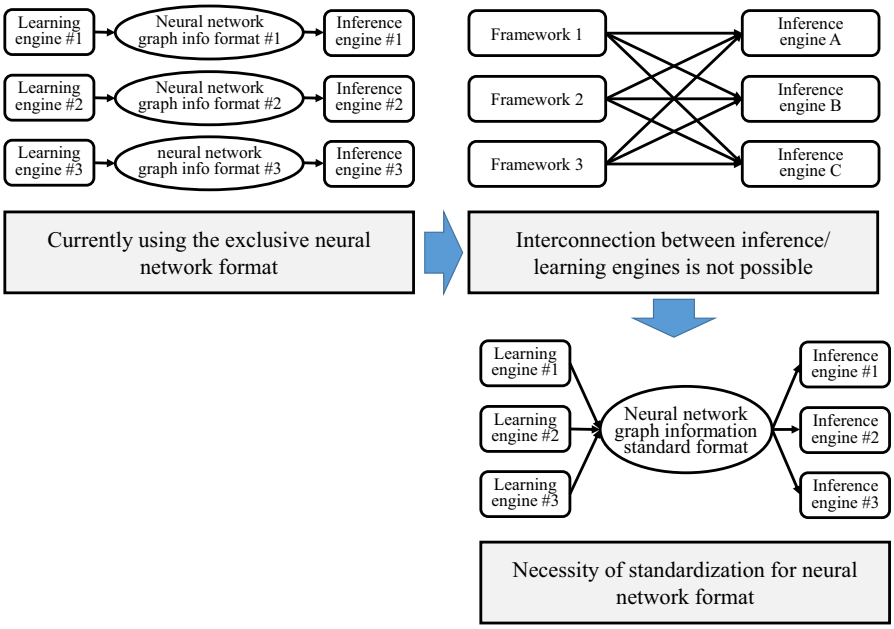

FIGURE 2 Necessity of standardizing neural networks

The Khronos Group is working on the development of a draft standard for graphical representation grammar and learning data storage formats for neural network inference engines. Version 1.0 is currently under revision.

NNEF can be used as a document to store graph information and data learned through a learning engine. It can also update the content described in the NNEF by extending and learning from existing stored data. In addition, through the parser, the target inference engine can generate code to implement the application service. The NNEF includes a primitive operation corresponding to a mathematical equation, which represents the operation of the neural network unit, as shown in Figure 3. It also includes a compound operation that provides a level of interface similar to the functions of C language by binding several primitive operations. In addition, it includes a graph that describes the interface defined at the top of the input and output of the neural network. The syntax of this NNEF is defined as a Backus-Naur Form (BNF).

NNEF defines a fragment by grouping a set of primitive operations and compound operations to define the language grammatically and represent nodes in the neural network. A fragment expresses functions in the text format for neural network data storage or operation. A fragment can receive parameter values similar to C/C++ functions or class definitions and can conduct neural network operations or call other fragments within a fragment.

Figure 4 shows the syntax, specifications, declaration, and type, which are part of the syntax structure of the NNEF format. In addition, the syntax structure of graph fragments is a fragment prototype that defines names, type parameters, and types, as shown in Figure 5. Fragment bodies can define other fragments, primitive invocations, parameter values of the type parameters, and operation behaviors.

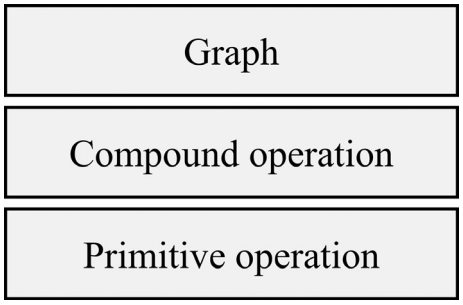

FIGURE 3 Example of a neural network graph structure in NNEF

```
<operation-declaration> ::= "fragment" <identifier>
          "(" <parameter-list> ")" "->" "(" <result-list> ")"
<parameter-list> ::= <parameter> ("," <parameter>)*
<parameter> ::= <identifier> ":" <type-spec> ["=" <literal-
expr>]
<result-list> ::= <result> ("," <result>)*
<result> ::= <identifier> ":" <type-spec>
<type-name> ::= "tensor" | "extent" | "scalar" | "logical" |
"string"
<array-type-spec> ::= <type-spec> "[]"
<tuple-type-spec> ::= "(" <type-spec> ("," <type-spec>)+ ")"
<type-spec> ::= <type-name> | <array-type-spec> | <tuple-type-
spec>
fragment foo( input: tensor, size: extent[] = [1] ) ->
( output: tensor )
```

FIGURE 4 Syntax structure of the NNEF format

```
graph barfoo( input ) -> ( output )
{
input = external(shape = [1,10])
intermediate, extra = bar(input, alpha = 2)
output = foo(intermediate, size = [3,5])
}
```

FIGURE 5 Example of a neural network graph expressed by NNEF

4 | IMPLEMENTATION

A neural network interoperability format converter (NNEF parser) performs conversion into codes that can recognize inference engines or other learning engines to input a document written in the NNEF grammar, as shown in Figure 6.

The transformable neural network proposed in this study had the same structure as a convolutional neural network (eg, AlexNet, GoogleNet, and LeNet), which includes repeated convolutions and pools, as shown in Figure 7. The implemented source code for neural network structures is shown in Figure 8. These neural networks were converted into the desired framework structure. To efficiently translate these structures, we developed a framework to translate the desired framework code using a parser tree through symbol rules according to NNEF basic grammar, which was redefined by the Khronos group.

The learning of deep neural networks requires considerable processing capacity and time, and therefore, it is carried out on devices with high computational power such as GPU processors. Hence, depending on the framework supported by small devices such as embedded systems, a technique capable of converting and driving learning data and neural networks is required, which is one of the objectives of the technique proposed in this study.

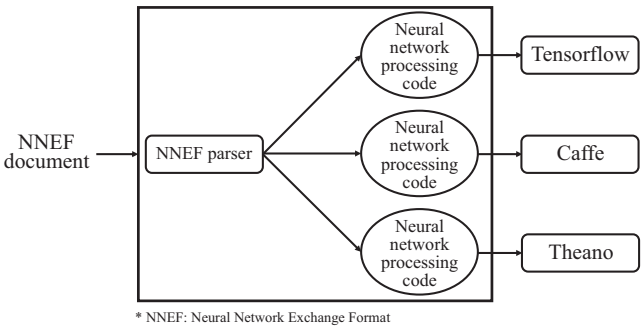

FIGURE 6 Conceptual diagram of a neural network information storage model

The NNEF parser was designed as an open source. We used LEX [26] to support lexical analysis and YACC [27] to automate parser generation. Moreover, we defined a token analysis rule that can recognize symbols defined in the NNEF document using LEX and YACC. In order to enable YACC to analyze sentences, we converted it into a grammar that YACC can recognize; this grammar is based on BNF, which is defined in NNEF.

As shown in Figure 9, the NNEF parser is designed to generate a parse tree corresponding to the NNEF document through YACC. We also added a plug-in conversion code generator that converts this parse tree into a code that can be recognized by the target inference engine.

As shown in Figure 10, the structural design of the neural network interworking module generates an NNEF file that represents the node structure of the neural network. It also generates a file that stores the learning data accompanying the neural network structure. Each inference engine can proceed by further learning or modifying the graphical structure using NNEF files. In addition, it is designed as an API that allows the inference to be executed using the syntax expressed in NNEF.

Considering prototype implementation and API design of a lexical analyzer in the interworking support module of a neural network, we employed token separation functions, such as the character set, literal, and symbol, defined in NNEF, as shown in Figure 11. In addition, we defined the main keywords and separation functions to define grammar. We also added LEX rules for processing comments, new lines, and other aspects.

The API deign of syntax analysis in the neural network support module changed the NNEF grammar defined by

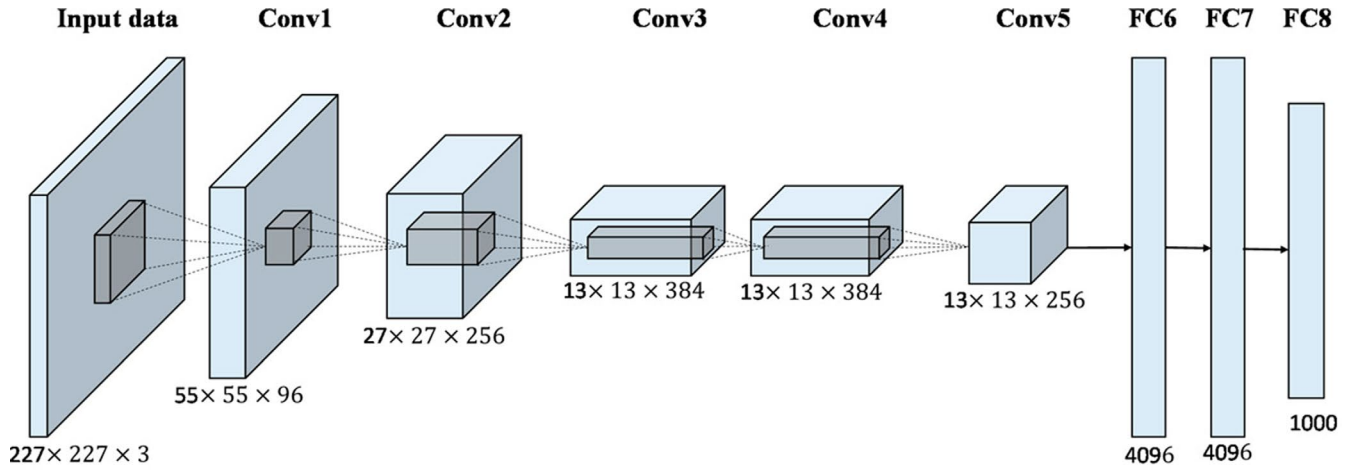

FIGURE 7 Structure of convolutional neural networks, such as AlexNet: Reprinted from X. Han et al., *Pre-trained AlexNet architecture with pyramid pooling and supervision for high spatial resolution remote sensing image scene classification*, Remote Sensing, **9** (2017) doi:10.3390/rs9080848 , CC BY 4.0.

```
graph AlexNet( input = tensor([-1, 28, 28, 1]) ) -> ( output: tensor )
{
    kernel1 = variable(shape = [3, 3, 1, 64], label = 'alexnet_v2/conv1/kernel');
    bias1 = variable(shape = [64], label = 'alexnet_v2/conv1/bias');
    conv1 = conv(input, filter = kernel1, strides=[1, 1, 1, 1], padding="SAME");
    add1 = add(conv1, bias1);
    relu1 = relu(add1);
    pool1 = max_pool(relu1,  ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding="SAME");
    norm1=local_response_normalization(pool1, depth_radius=5.0, bias=2.0, alpha=1e-4, beta=0.75);
    norm1= dropout(norm1, keep_prob);
    kernel2 = variable(shape = [3, 3, 64, 128], label = 'alexnet_v2/conv2/kernel');
    bias2 = variable(shape = [128], label = 'alexnet_v2/conv2/bias');
    conv2 = conv(norm1, kernel2, strides=[1, 1, 1, 1], padding="SAME");
    add2 = add(conv2, bias2);
    relu2 = relu(add2);
    pool2 = max_pool(relu2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding="SAME");
    norm2=local_response_normalization(pool2, depth_radius=5.0, bias=2.0, alpha=1e-4, beta=0.75);
    norm2= dropout(norm2, keep_prob);
    kernel3 = variable(shape = [3, 3, 128, 256], label = 'alexnet_v2/conv3/kernel');
    bias3 = variable(shape = [256], label = 'alexnet_v2/conv3/bias');
    conv3 = conv(norm2, kernel3, strides=[1, 1, 1, 1], padding="SAME");
    add3 = add(conv3, bias3);
    relu3 = relu(add3);
    pool3 = max_pool(relu3, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding="SAME");
    norm3=local_response_normalization(pool3, depth_radius=5.0, bias=2.0, alpha=1e-4, beta=0.75);
    norm3= dropout(norm3, keep_prob);
    kernel4 = variable(shape = [4*4*256, 1024], label = 'alexnet_v2/fc4/kernel');
    bias4 = variable(shape = [1024], label = 'alexnet_v2/fc4/bias');
    resh4 = reshape(norm3, [-1, kernel4.get_shape().as_list()[0]] );
    mul4 = matmul(resh4, kernel4);
    add4 = add(mul4, bias4);
    relu4 = relu(add4);
    kernel5 = variable(shape = [1024, 1024], label = 'alexnet_v2/fc5/kernel');
    bias5 = variable(shape = [1024], label = 'alexnet_v2/fc5/bias');
    resh5 = reshape(relu4, [-1, kernel5.get_shape().as_list()[0]] );
    mul5 = matmul(resh5, kernel5);
    add5 = add(mul5, bias5);
    output = relu(add5)
}
```

(A)

FIGURE 8 Generating Tensorflow codes using the proposed converter: (A) NNEF-based AlexNet and (B) Tensorflow-based neural network codes generated by the proposed framework

BNF into YACC. We also changed the grammar of ambiguously defined syntax in the NNEF standard document. Thus, an interface could be implemented between the LEX and YACC modules. Figure 12 shows some codes used for the syntax analyzer of the NNEF parser.

The learning data storage model processes learning data and saves it in a data format supported by NNEF by combining the NNEF header and data. Table 2 shows the structure of the NNEF-based data format header. Data quantization is achieved by quantizing and storing data to develop a quantized function according to the recommendations of the Khronos group's NNEF. In contrast, when fetching data, quantized data is released using the imquantize function.

The proposed system constructs an inference engine based on Tensorflow by importing AlexNet's neural network information and learning data based on NNEF. For the first time, we implemented NNEF_Data_Reading and NNEF_Network_Reading functions to read neural network graphs and learn data. NNEF_Network_Reading replaces the neural network based on NNEF with Tensorflow codes using our interworking support framework and compiles it to run the transformed code in Tensorflow. Figure 13 illustrates a NNEF file reading function for conversion into executable code in Tensorflow after importing the NNEF-based neural network information. NNEF_Data_Reading creates Tensorflow variables that can be employed by interpreting NNEF protocol-based data files using the learning data interoperability support framework. Figure 14 shows the process of replacing the data that can be run on Tensorflow by analyzing NNEF-based data. It should be noted that parameter 12 indicates 12 data files. "/Temp/dat/" represents the folder with data.

5 | RESULT

The interpreter of the proposed interworking converts an NNEF-based neural network, shown in Figure 8A, into neural network codes based on Tensorflow, shown in Figure 8B.

The implementation results of the proposed system can be verified through MNIST operation, which was conducted to import AlexNet neural network information and learning data based on NNEF in Tensorflow, as shown in Figure 15.

Figure 16 shows that results can be obtained quickly by constructing an inference engine using Tensorflow codes, as shown in Figure 15. Thus, the proposed system makes it possible to quickly run an AI inference engine by importing neural network information and learning data based on NNEF.

```
input = tf.reshape(inputs, [-1, 28, 28, 1])
kernel1 = tf.Variable(tf.random_normal([3, 3, 1, 64]), name='alexnet_v2/conv1/kernel')
bias1 = tf.Variable(tf.random_normal([64]), name='alexnet_v2/conv1/bias')
conv1 = tf.nn.conv2d(input, kernel1, strides=[1, 1, 1, 1], padding="SAME")
add1  = tf.add(conv1, bias1)
relu1 = tf.nn.relu(add1)
pool1 = tf.nn.max_pool(relu1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding="SAME")
norm1 = tf.nn.local_response_normalization(pool1, depth_radius=5.0, bias=2.0, alpha=1e-4, beta=0.75)
norm1 = tf.nn.dropout(norm1, keep_prob)
kernel2 = tf.Variable(tf.random_normal([3, 3, 64, 128]), name='alexnet_v2/conv2/kernel')
bias2 = tf.Variable(tf.random_normal([128]), name='alexnet_v2/conv2/bias')
conv2 = tf.nn.conv2d(norm1, kernel2, strides=[1, 1, 1, 1], padding="SAME")
add2  = tf.add(conv2, bias2)
relu2 = tf.nn.relu(add2)
pool2 = tf.nn.max_pool(relu2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding="SAME")
norm2 = tf.nn.local_response_normalization(pool2, depth_radius=5.0, bias=2.0, alpha=1e-4, beta=0.75)
norm2 = tf.nn.dropout(norm2, keep_prob)
kernel3 = tf.Variable(tf.random_normal([3, 3, 128, 256]), name='alexnet_v2/conv3/kernel')
bias3 = tf.Variable(tf.random_normal([256]), name='alexnet_v2/conv3/bias')
conv3 = tf.nn.conv2d(norm2, kernel3, strides=[1, 1, 1, 1], padding="SAME")
add3  = tf.add(conv3, bias3)
relu3 = tf.nn.relu(add3)
pool3 = tf.nn.max_pool(relu3, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding="SAME")
norm3 = tf.nn.local_response_normalization(pool3, depth_radius=5.0, bias=2.0, alpha=1e-4, beta=0.75)
norm3 = tf.nn.dropout(norm3, keep_prob)
kernel4 = tf.Variable(tf.random_normal([4*4*256, 1024]), name='alexnet_v2/fc4/kernel')
bias4 = tf.Variable(tf.random_normal([1024]), name='alexnet_v2/fc4/bias')
resh4 = tf.reshape(norm3, [-1, kernel4.get_shape().as_list()[0] ] )
mul4 = tf.matmul(resh4, kernel4)
add4  = tf.add(mul4, bias4)
relu4 = tf.nn.relu(add4)
kernel5 = tf.Variable(tf.random_normal([1024, 1024]), name='alexnet_v2/fc5/kernel')
bias5 = tf.Variable(tf.random_normal([1024]), name='alexnet_v2/fc5/bias')
resh5 = tf.reshape(relu4, [-1, kernel5.get_shape().as_list()[0] ] )
mul5 = tf.matmul(resh5, kernel5)
add5 = tf.add(mul5, bias5)
output = tf.nn.relu(add5)
```

(B)

FIGURE 8 (Continued)

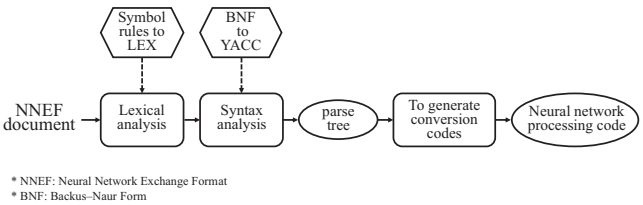

FIGURE 9 NNEF parsing process sequence

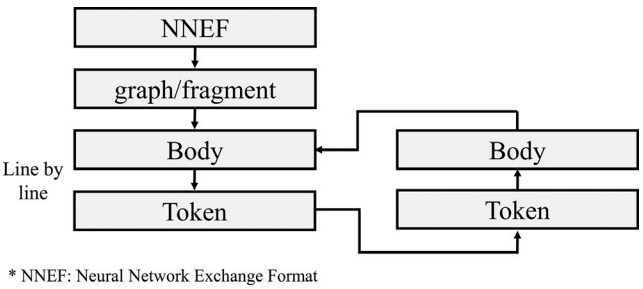

FIGURE 10 Operation structure of the NNEF parser

6 | CONCLUSION

Owing to their different storage structures, AI deep learning frameworks require standardization for code reuse and maintenance. The standardized NNEF can execute stored neural

```
# Ignored characters
t_ignore                        = " \t"
t_ignore_COMMENT                = r'\#.*'
# Tokens
t_single_quote_literal          =r'\'([^\\\n]|(\\.))*?\''
t_double_quote_literal          =r'\"([^\\\n]|(\\.))*?\"'
t_hyphen_angle_braket_token     =r'->'
t_open_angle_braket_equal_token =r'<='
t_close_angle_braket_equal_token =r'>='
t_equal_equal_token             =r'=='
t_exclam_equal_token            =r'!='
t_and_and_token                 =r'&&'
t_or_or_token                   =r'\|\|'
t_numeric_literal     =r'([0-9]+)(\.[0-9]*)?([eE][-+]?[0-9]+)?'
t_open_angle_braket_token       =r'<'
t_close_angle_braket_token      =r'>'
t_open_paren_token              =r'\('
t_close_paren_token             =r'\)'
t_open_braket_token             =r'\['
t_close_braket_token            =r'\]'
t_open_brace_token              =r'\{'
t_close_brace_token             =r'\}'
t_comma_token                   =r','
t_colon_token                   =r':'
t_semi_colon_token              =r';'
t_equal_token                   =r'='
t_plus_token                    =r'\+'
t_minus_token                   =r'-'
t_star_token                    =r'\*'
t_slash_token                   =r'/'
t_circum_token                  =r'\^'
t_exclam_token                  =r'!'

def t_identifier(t):
    r'[_a-zA-Z][_a-zA-Z0-9]*'
    t.type = reserved.get(t.value, 'identifier' )
    return t
```

FIGURE 11 Some codes used for the lexical analyzer

```
########################################
def p_unary_plus_expression(t):
    print('p_unary_plus_expression   ')
    pass
def p_unary_minus_expression(t):
    print('p_unary_minus_expression   ')
    pass
##############################################
def p_error(t):
    if t:
        print('\n$$$$$$$$$\nSyntax error at line =====>',
                                            t.lexer.lineno)
        print('Symbol===>  ', t.value)
    else:
        print("Syntax error at EOF")
##############################################
# Build the lexer
import ply.lex as lex
import ply.yacc as yacc
def p_error(t):
    if t:
        print('\n$$$$$$$$$\nSyntax error at line =====>',
t.lexer.lineno)
        print('Symbol===>  ', t.value)
    else:
        print("Syntax error at EOF")
##############################################
# Build the lexer
import ply.lex as lex
import ply.yacc as yacc
lex.lex()
yacc.yacc()
yacc.parse(open('nnef3.txt').read())
```

FIGURE 12 Some codes used for the syntax analyzer

TABLE 2 Data format structure based on NNEF

Item	Bit
NNEF data identification number	0x4E 0xEF
Version information	0x00 0x01
Offset information of actual data	0x00 0x00 0x00 0xFF
Order of tensor	0x00 0x00 0x00 0x00
Range of tensor	0x00 0x00 0x00 0x00
Type of tensor	0x00: 0bit: 0: float values: 1: quantized values 2: signed integer values 3: unsigned integer values
Bit width for each item of Tensor	0x00
Quantization algorithm length	0x00 0x00

networks and data (inference engine) with a unique format on various deep learning frameworks without having to follow the particular format of each framework.

This study describes the design of converters for neural network interoperability based on NNEF. The proposed system presents an interworking framework in which a NNEF-based neural network can be transformed into a computational graph for various deep learning frameworks. In particular, the proposed system can be run directly on the desired deep-running framework by converting the file based on NNEF using top-down parsing-based lexical, syntax,

Call Function in Tensorflow:

```
exec(compile(NNEF_Network_Reading
    ("/Temp/network.nnef"),'<string>', 'exec'))
```

Implemented Function:

```
def NNEF_Network_Reading(file):
    nncode = ""
    infile = open(file, "r")
    for line in infile:
        for word2 in line.split("\n"):
            if len(word2) != 0:
                nncode = nncode + word2 + "\n"
    infile.close()
    return nncode
```

FIGURE 13 Reading function for an NNEF-based neural network

Call Function in Tensorflow:

```
ND = NNEF_Data_Reading(12, "/Temp/dat/")
```

Implemented Function:

```
def NNEF_Data_Reading(ii, FilePath):
    WW = [0 for i in range(ii)]
    filenames = os.listdir(FilePath)
    filenum = 0
    for filename in filenames:
        full_filename = os.path.join(FilePath, filename)
        infile = open(full_filename, "r")
        l = d = 0
        for line in infile:
            for word2 in line.split():
                if word2[1] == 'x':
                    d = d + 1
                    if d == 9:
                        num1 = int(word2, 16)
        #code skip..
    return WW
```

FIGURE 14 Reading function for NNEF-based data

and semantic analyses. We included the conversion technologies for neural networks and learned data. Additionally, we tested the running of AlexNet on Tensorflow by converting a neural network and data based on NNEF.

7 | FUTURE WORK

In the future, NNEF and ONNX are to be studied for developing neural network standards. Both technologies have their own advantages and disadvantages. If compatibility can be achieved between these two standards, it would be a landmark achievement. In addition, ONNX's neural networks and data formats should be further standardized using protocol buffers. We believe that NNEF's formats can improve interoperability using a formal format, such as protocol buffer. The interworking technology should be studied so that the existing neural network and learning data can be

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
from NNEFDATA import NNEF_Data_Reading, NNEF_Network_Reading
sess = tf.InteractiveSession()
sess.run(tf.global_variables_initializer())
inputs = tf.placeholder(tf.float32, shape=[None, 784])
labels = tf.placeholder(tf.float32, shape=[None, 10])
keep_prob = tf.placeholder(tf.float32)

ND = [0 for i in range(12)]
ND = NNEF_Data_Reading(12, "/Temp/dat/")
exec(compile(NNEF_Network_Reading("/Temp/network.nnef"), '<string>', 'exec'))

#########################################################################################################
out_weights = tf.Variable(tf.random_normal([1024, 10]))
out_biases = tf.Variable(tf.random_normal([10]))
pred = tf.add(tf.matmul(output, out_weights), out_biases)
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=pred, labels=labels))
train_op = tf.train.AdamOptimizer(learning_rate=0.001).minimize(cost)
correct_prediction = tf.equal(tf.argmax(pred,1), tf.argmax(labels,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
#########################################################################################################

print("AlexNet-based MNIST accuracy: %g" % sess.run(accuracy,
                                    feed_dict={kernel1: ND[7], bias1: ND[1],
                                               kernel2: ND[8], bias2: ND[2], kernel3: ND[9],
                                               bias3: ND[3], kernel4: ND[10], bias4: ND[4],
                                               kernel5: ND[11], bias5: ND[5], out_weights: ND[6],
                                               out_biases: ND[0], inputs: mnist.test.images[:256],
                                               labels: mnist.test.labels[:256], keep_prob: 1.0}))
```

Import learning data

Import neural network

FIGURE 15 Importing NNEF files on Tensorflow and MNIST operation screen

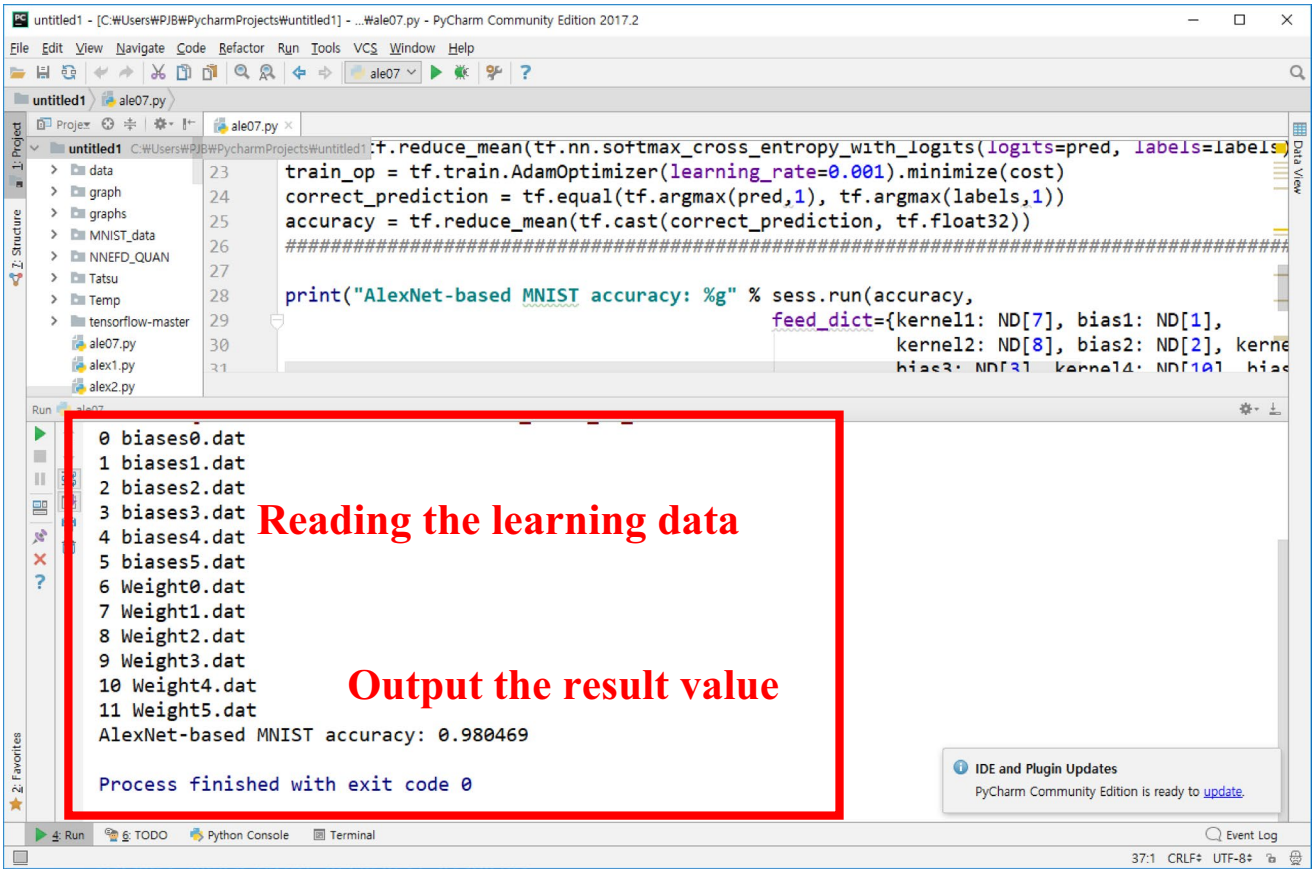

FIGURE 16 Output of MNIST accuracy based on AlexNet

optimally transformed for high performance beyond a simple transformation.

FUNDING INFORMATION

This work was supported by the Institute for Information & Communications Technology Promotion (IITP) grant funded by the Korea Government (MSIP) (No. 2017-0-00068, A Development of Driving Decision Engine for Autonomous Driving (4th) using Driving Experience Information).

ORCID

Jaebok Park https://orcid.org/0000-0003-4160-7805

REFERENCES

1. Khronos, Neural network exchange format (NNEF) specification, Beaverton, OR, USA, 2018.
2. H. Deepika, N. Mangala, and S.C. Babu, *Automatic program generation for heterogeneous architectures*, in Proc. Int. Adv. Comput., Commun. Inf. Conf., Jaipur, India, Sept. 2016, pp. 102–109.
3. A. Krizhevsky, I. Sutskever, and G.E. Hinton, *Imagenet classification with deep convolutional neural networks*, in Proc. Int. Neural Inf. Process. Syst. Conf., Lake Tahoe, NV, USA, Dec. 2012, pp. 1097–1105.
4. K. Simonyan and A. Zisserman, *Very deep convolutional networks for large-scale image recognition*, in Proc. Int. Learn. Representations Conf., San Diego, CA, USA, May 2015, pp. 1–14.
5. Y. LeCun, Y. Bengio, and G. Hinton, *Deep learning*, Nature J. **521** (2015), 436–444.
6. J. Deng et al., Imagenet: A large-scale hierarchical image database, in Proc. Int. IEEE Comput. Vision Pattern Recogn. Conf., Miami, FL, USA, June 2009, pp. 248–255.
7. M. Abadi et al., *TensorFlow: Large-scale machine learning on heterogeneous distributed systems*, in Proc. Int. USENIX Symp. Oper. Syst. Des. Implement., Savannah, GA, USA, 2016, pp. 265–283.
8. F. Chollet, *Deep learning with python*, Manning Publications, Shelter Island, NY, 2017.
9. Y. Jia et al., *Caffe: Convolutional architecture for fast feature embedding*, in Proc. Int. ACM Multimedia Conf., Orlando, FL, USA, Nov. 2014, pp. 675–678.
10. I. Serrano et al., *Fight recognition in video using hough forests and 2D convolutional neural network*, IEEE Trans. Image Process. **27** (2018), no. 10, 4787–4797.
11. Facebook Open Source, Caffe2: a new lightweight, modular, and scalable deep learning framework, Accessed Jan. 2018, http://caffe2.ai
12. R. Collobert, K. Kavukcuoglu, and C. Farabet, *Torch7: A MATLAB-like environment for machine learning*, BigLearn, NIPS Workshop, no. EPFL-CONF-192376, 2011.
13. H. Adie, I. Paradana, and Pranowo, *Parallel computing accelerated image inpainting using GPU CUDA, Theano, and Tensorflow*, in Proc Int. Inf. Technol. Electr. Eng., Kuta, Indonesia, July 2018, pp. 621–625.
14. S. Chetlur et al., *CUDNN: Efficient primitives for deep learning*, CoRR (2014). http://arxiv.org/abs/1410.0759
15. T.M. Breuel, *High performance text recognition using a hybrid convolutional-LSTM implementation*, in Proc. Int. IEEE IAPR Document Anal. Recogn., Kyoto, Japan, Nov. 2017, pp. 11–16.
16. The Theano Development Team, *Theano: a python framework for fast computation of mathematical expressions*, arXiv preprint, 2016, arXiv:1605.02688.
17. F. Chollet, *Keras: deep learning library for Theano and Tensorflow*, GitHub Repository, Tech. Rep., 2015, https://keras.io/getting-started/faq/#how-should-i-cite-keras
18. H. Kim et al., *Applied machine learning at Facebook: a datacenter infrastructure perspective*, in Proc. Int. IEEE Symp. High Performance Comput. Architecture, Vienna, Austria, Feb. 2018, pp. 620–629.
19. J.Q. Candela, *Facebook and Microsoft introduce new open ecosystem for interchangeable AI frameworks*, Sept. 2017.
20. J. Ambrosi et al., *Hardware-software co-design for an analog-digital accelerator for machine learning*, in Proc. Int. IEEE Rebooting Comput., Mclean, VA, USA, Nov. 2018, pp. 1–13.
21. B. Seo et al., *Top-down parsing for Neural Network Exchange Format (NNEF) in TensorFlow-based deep learning computation*, in Proc. Int. Inf. Netw., Chiang Mai, Thailand, Jan. 2018, pp. 522–524.
22. M. Shin et al., *Neural network syntax analyzer for embedded standardized deep learning*, in Proc. Int. Workshop Embedded Mobile Deep Learn., Munich, Germany, June 2018, pp. 37–41.
23. T. Chen et al., MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems, arXiv preprint, 2015, arXiv:1512.01274.
24. J. Hanhirova et al., *Latency and throughput characterization of convolutional neural networks for mobile computer vision*, in Proc. Int. ACM Multimedia Syst. Conf., Amsterdam, Netherlands, June 2018, pp. 204–215.
25. Z. Guo, J. Han, and T. Li, *Implementing OpenVX on a polymorphous array processor*, in Proc. Int. IEEE Commun. Technol. Conf., Hangzhou, China, Oct. 2015, pp. 598–601.
26. M. Upadhyaya, *Simple calculator compiler using Lex and YACC*, in Proc. Int. IEEE Electron. Comput. Technol. Conf., Kanyakumari, India, Apr. 2011, pp. 182–187.
27. P. Nakwijit and P. Ratanaworabhan, *A parser generator using the Grammar Flow Graph*, in Proc. Int. IEEE Comput. Sci. Eng. Conf., Chiang Mai, Thiland, Nov. 2015, pp. 1–6.

AUTHOR BIOGRAPHIES

Jaebok Park received his PhD degree in Computer Engineering from Chonbuk National University, Jeonju, Korea in 2011 and MS degree in Computer Engineering from Chungnam National University, Daejeon, Korea in 2007. He joined the Electronics and Telecommunications Research Institute (ETRI), Daejeon, Korea in September 2011. His research interests include localization, tracking, machine learning frameworks, and inference engines for embedded systems.

Seungmok Yoo is a principal researcher at ETRI. He received his BS and MS degrees in Computer Engineering from Kyungpook National University, Daegu, Korea in 1994 and 1996, respectively. He received a PhD in Electrical and Computer Engineering from UC Irvine, Irvine, CA, USA in 2007. He was a researcher at the Agency for Defense Development, Daejeon, Korea from 1996 to 2001. His research interests include machine learning frameworks and inference engines, node architecture design, MAC and routing protocol design, and distributed real-time system in wireless sensor networks and embedded systems.

Seokjin Yoon received his BS and MS degrees from Chung-Ang University, Seoul, Korea in 1992 and 1994, respectively. He is now a principal member of the research staff at ETRI. His main research interests are high speed computing platforms and mobile systems.

Kyunghee Lee received his BS and MS degrees from Kyungpook National University, Daegu, Korea in 1990 and 1992, respectively. He is now a principal member of the research staff at ETRI. His main research interests are embedded systems, high speed computing platforms, and real-time systems.

Changsik Cho received his PhD from ChungNam National University, Korea in 2011 and BS and MS degrees from KyungPook National University, Korea in 1993 and 1995, respectively. In January 1995, he joined ETRI, where he is currently a principal researcher. His research interests are machine learning frameworks and inference engines for embedded systems.