# Enhancing the Interoperability between Deep Learning Frameworks by Model Conversion

Yu Liu*
Microsoft Research
National University of Singapore

Cheng Chen†
ByteDance Inc.

Ru Zhang
Microsoft Research

Tingting Qin
Microsoft Research

Xiang Ji
Microsoft Research

Haoxiang Lin‡
Microsoft Research

Mao Yang
Microsoft Research

## ABSTRACT

Deep learning (DL) has become one of the most successful machine learning techniques. To achieve the optimal development result, there are emerging requirements on the interoperability between DL frameworks that the trained model files and training/serving programs can be re-utilized. Faithful model conversion is a promising technology to enhance the framework interoperability in which a source model is transformed into the semantic equivalent in another target framework format. However, several major challenges need to be addressed. First, there are apparent discrepancies between DL frameworks. Second, understanding the semantics of a source model could be difficult due to the framework scheme and optimization. Lastly, there exist a large number of DL frameworks, bringing potential significant engineering efforts.

In this paper, we propose MMdnn, an open-sourced, comprehensive, and faithful model conversion tool for popular DL frameworks. MMdnn adopts a novel unified intermediate representation (IR)-based methodology to systematically handle the conversion challenges. The source model is first transformed into an intermediate computation graph represented by the simple graph-based IR of MMdnn and then to the target framework format, which greatly reduces the engineering complexity. Since the model structure expressed by developers may have been changed by DL frameworks (*e.g.*, graph optimization), MMdnn tries to recover the original high-level neural network layers for better semantic comprehension via a pattern matching similar method. In the meantime, a piece of model construction code is generated to facilitate later retraining or serving. MMdnn implements an extensible conversion architecture from the compilation point of view, which eases contribution from the community to support new DL operators and frameworks. MMdnn has reached good maturity and quality, and is applied for converting production models.

## CCS CONCEPTS

• **Software and its engineering** → **Interoperability**.

## KEYWORDS

deep learning, neural network, model conversion

## 1 INTRODUCTION

Deep learning (DL) has become one of the most successful machine learning techniques, and is applied to various application areas such as image recognition [18], natural language processing (NLP) [16], and board games [51]. Developers use *deep learning frameworks* to design layered data representations called deep neural networks or deep learning models, and train/serve them across various computing hardware like CPU, GPU, and TPU. Such DL frameworks provide high-level programming interfaces and basic building blocks for the model construction, which greatly improves programmability and boosts productivity. There are already many widely used DL frameworks [63] for both research and production: TensorFlow (TF) [1], PyTorch [41], Microsoft Cognitive Toolkit (CNTK) [50], Caffe [25], Apache MXNet [8], Keras [11], Baidu PaddlePaddle [4], Darknet [44], Apple Core ML [2], ONNX [37], [1] etc., with each having its own uniqueness on expressivity, usability, and runtime performance.

Recently, there are emerging requirements on the *interoperability* [64] between the above DL frameworks that the trained model files and training/serving programs could be easily ported and reutilized with another framework. This is because employing multiple frameworks becomes a common practice towards the optimal development result including not only model learning performance but also programming experience and DevOps productivity. However, existing DL frameworks mainly focus on runtime performance and expressivity while neglecting composability and portability, which makes the framework interoperability rather difficult.

We think that the faithful model conversion is a promising technology to enhance the interoperability between DL frameworks. The conversion works by transforming a source model into the semantic equivalent in another framework format and in the meantime generating a piece of model construction code. Developers may tune such code (*e.g.*, writing custom pre- or post-processing logic) for later retraining or serving. Model conversion is technically feasible because all current DL frameworks take the same abstraction to represent models as similar *tensor-oriented computation graphs* whose syntax and semantics are well defined. However, the following non-trivial challenges exist:

(1) There are apparent discrepancies in both the computation graph constructs and the supported features between DL

*Most of this author's work is done as an intern at Microsoft Research.
†This author's work is done as a full-time employee at Microsoft Research.
‡Corresponding author.

[1] ONNX provides not only "an open source format for AI models" but also "a cross-platform inferencing and training runtime" (aka.ms/onnxruntime).

frameworks. For example, PyTorch provides the *log1p* operator (aka a mathematical operation) calculating $\log(1 + x)$, which is not available in CNTK, Keras, and ONNX. Just creating a placeholder node in the target model does not work; we must implement *log1p* with other defined operators (*e.g.*, using *Add* plus the immediately followed *log*). Another example is that not all DL frameworks support both `NCHW` and `NHWC` tensor layouts [23]. Therefore, additional transpositions of the data and learnable parameters may be needed in the target model.

(2) The computational structure saved in a model file can be different and complex than that expressed by developers in the original training program due to the framework scheme and optimization. This makes semantic comprehension much harder and may result in a conversion failure or a non-optimal target model. For instance, TensorFlow translates neural network layers to low-level optimized tensor operations. Another example is that PyTorch adopts the "define-by-run" scheme such that a PyTorch model file stores a dynamic computation graph defined by a real run and may lose the conditional or loop information. Therefore, reverse syntax transformation is important for a source model to recover the original DL constructs.

(3) There are a large number of popular DL frameworks, bringing potential significant engineering efforts. Such a problem is very similar to that in compiler construction to support various front-end programming languages and back-end architectures. We could refer to compilers on the design and implementation.

In this paper, we propose **MMdnn**: **M**odel **M**anagement for **d**eep **n**eural **n**etworks, an open-sourced, comprehensive, and faithful model conversion tool [31]. Compared to other converters, our work has three unique contributions.

First, MMdnn adopts a novel unified intermediate representation (IR)-based methodology to systematically address the above challenges. We design a simple graph-based IR for model conversion by reference to existing DL frameworks like ONNX. Its purpose is to depict as many IR constructs of current frameworks as possible and eliminate their syntactic and semantic differences (*e.g.*, the *LRN* [26] operator has different argument definitions in Caffe and TensorFlow). The source model is first transformed into an intermediate computation graph represented by our IR and then to the target format, using a node-to-node translation technique similar to the compiler's instruction selection [6]. Hence, the engineering complexity is significantly reduced from $O(\mathrm{M} \times \mathrm{N})$ to $O(\mathrm{M} + \mathrm{N})$, where M and N are the numbers of the source and target frameworks, respectively. MMdnn also performs reverse syntax transformation on the source model for better semantic comprehension via a pattern matching similar method (*e.g.*, recovering a TensorFlow fully-connected structure from a subgraph consisting of several low-level computation nodes). We think that such a unified IR-based methodology is general for AI/DL tooling and could be further applied to other tools including DL transpiler (aka source-to-source compiler), visual DL programming, full model optimization, etc.

Second, MMdnn implements an extensible conversion architecture from the compilation point of view. It decouples the whole conversion pipeline into three major phases: source model reconstruction, intermediate model transformation, and target model generation, with their interfaces being clearly and carefully defined. Such an architecture enables modularization and automation, which eases contribution from the community to support new DL operators and frameworks.

Lastly, MMdnn has reached good maturity and quality for sophisticated models. A team of our company successfully converted and deployed the production models (Caffe to ONNX). Up to now, MMdnn is able to convert VGG [52], ResNet [18], ResNeXt [65], Xception [10], Inception [53], MobileNets [20], SqueezeNet [21], NasNet [66], FaceNet [49], YOLO [45], etc. models for the previously mentioned DL frameworks. [2]

The rest of the paper is organized as follows. In Section 2, we describe the background. Section 3 presents our methodology. Section 4 details the design and implementation of MMdnn. Section 5 demonstrates the evaluation results. We survey related work in Section 6 and conclude this paper in Section 7.

## 2 BACKGROUND

Deep learning (DL) is a subfield of machine learning to learn layered data representations called models. Python is the most popular programming language, while others like C++ or Julia [5] are also used in certain cases. Being essentially a mathematical function, a DL model is formalized by the frameworks as a tensor-oriented computation graph. It is a *directed acyclic graph* (DAG) in which each node represents the invocation of some mathematical operation (aka *operator. e.g.*, matrix multiplication). The node takes a list of tensors as the input and produces a list of tensors as the computation output. A *tensor* is a multi-dimensional array of numerical values, and its *order* is the number of the dimensions. An output tensor of some node $A$ is delivered to another node $B$ as one input tensor through a directed edge. Such an edge also specifies the execution dependency between $A$ and $B$. The node may additionally contain some or even massive numerical learnable parameters (*i.e.*, weights and biases), which are iteratively updated during training until the model learning performance (*e.g.*, accuracy and loss) meets the expectation.

The intermediate representation (IR) for deep learning is an abstract language used internally to represent computation graphs. Different DL frameworks have their own unique IRs, and some general IRs such as MLIR (Multi-Level Intermediate Representation) [27] and TVM Relay [46] are extended to represent programs written in the host programming languages.

After training has finished, the whole model (including its graph structure and learnable parameters) is serialized in some format to one or more disk files. Note that a DL framework may support multiple model formats. For example, TensorFlow provides three options: checkpoint, frozen graph, and saved model [58].

[2] Currently, the Darknet and PaddlePaddle formats are for source only, and the ONNX format is for target only, considering their popularity and our development resources.

# 3 METHODOLOGY

Formally, a DL model is represented as a directed acyclic graph (DAG):

$$\boldsymbol{M} = \langle V = \{u_i\}_{i=1}^{N}, E = \{(u_i, u_j)\}_{i \neq j}, P = \{p_i\}_{i=1}^{K} \rangle$$

Each node $u_i$ invokes some operator denoted by $u_i.op$. A directed edge $(u_i, u_j)$ delivers one of $u_i$'s output tensors to the node $u_j$ as an input tensor and specifies that $u_j$ can start execution only if $u_i$ has finished. Each $p_i$ is a hyperparameter such as batch size, learning rate, kernel size, and dropout rate. As mentioned above, the model $\boldsymbol{M}$ is essentially a mathematical function denoted by $\mathcal{F}_{\boldsymbol{M}}$. We say that $\boldsymbol{M}_s = \langle V_s, E_s, P_s \rangle$ is a submodel of $\boldsymbol{M}$ if $V_s \subseteq V$, $E_s \subseteq E$, and $P_s = P$.[3]

Suppose that $\mathcal{X}_{i \in [1,m]}, \mathcal{Z}_{i \in [1,n]}$ are tensors. We use tensor tuples $\mathcal{X}_{tp} = \langle \mathcal{X}_1, \mathcal{X}_2, \cdots, \mathcal{X}_m \rangle$ and $\mathcal{Z}_{tp} = \langle \mathcal{Z}_1, \mathcal{Z}_2, \cdots, \mathcal{Z}_n \rangle$ to denote the input and output of a node/operator, respectively. We favor the tuple over the set because the tensor ordering can be critical to certain operators such as matrix multiplication. The orderings of tensors, edges, and nodes are actually determined by the program statements written by developers to construct the computation graph. Let $\mathcal{F}_{op}$ be the mathematical function of the operator $op$ which takes $m$ input tensors and returns $n$ output tensors. Then, $op$ is denoted as follows:

$$\langle \mathcal{Z}_1, \mathcal{Z}_2, \cdots, \mathcal{Z}_n \rangle = \mathcal{F}_{op}(\langle \mathcal{X}_1, \mathcal{X}_2, \cdots, \mathcal{X}_m \rangle) \tag{1}$$

The model $\boldsymbol{M}$ has some nodes without predecessors (*e.g.*, reading input data from disk), which provide initial values to activate the DL computation. The tuple of their output tensors following the tensor and node orderings is called the input of $\boldsymbol{M}$. Similarly, $\boldsymbol{M}$'s output is denoted by the tuple of output tensors from the nodes with no successors. For the submodel $\boldsymbol{M}_s$, its input is the tensor tuple with each element being either delivered from an outside node or produced by some internal node as an input tensor of $\boldsymbol{M}$. The output of $\boldsymbol{M}_s$ can be defined similarly. With the above notations, we describe the faithful model conversion.

*Definition 3.1.* $\mathcal{A}$ is a faithful model conversion algorithm if the following two conditions are satisfied:

**Syntactic Legality.** Given an arbitrary source model $\boldsymbol{M}_1 = \langle V_1, E_1, P_1 \rangle$, a legitimate target model $\boldsymbol{M}_2 = \langle V_2, E_2, P_2 \rangle$ should be produced:

$$\boldsymbol{M}_1 \vdash_{\mathcal{A}} \boldsymbol{M}_2 \wedge (P_2 = P_1)$$

**Semantic Equivalence.** Given an arbitrary valid input, the source and target models should always return the same result:

$$\mathcal{Z}_{tp} = \mathcal{F}_{\boldsymbol{M}_1}(\mathcal{X}_{tp}) \vdash_{\mathcal{A}} \mathcal{Z}_{tp} = \mathcal{F}_{\boldsymbol{M}_2}(\mathcal{X}_{tp})$$

$\mathcal{Z}_{tp}, \mathcal{X}_{tp}$ are the output and input tensor tuples, respectively.

Ideally, MMdnn performs the *isomorphic* graph transformation or graph rewriting [47] on $\boldsymbol{M}_1$: for each source node $u_i$ (*e.g.*, a TensorFlow *Conv2D* node), a new node $v_i$ with the same $u_i.op$ operator (still *Conv2D* for the case) is generated in $\boldsymbol{M}_2$; if $u_i$ has an edge pointing from its $k$-th output tensor to the $l$-th input tensor of $u_j$ (*i.e.*, $(u_i, u_j) \in \boldsymbol{M}_1$), a corresponding edge from $v_i$'s $k$-th output to $v_j$'s $l$-th input is added. The faithfulness clearly holds. However, it may not always be possible or efficient to carry out such exact one-to-one node translations; therefore, subgraph transformations should be considered. We adopt the bottom-up approach to design a general model conversion algorithm $\mathcal{A}^{\text{MMdnn}}$:

(1) $\mathcal{A}^{\text{MMdnn}}$ identifies a finite *partition* $\Pi = \{V_{1,1}, V_{1,2}, \cdots, V_{1,t}\}$ of $V_1$. That is,

$$(V_{1,i} \cap V_{1,j} = \emptyset) \wedge (V_1 = \bigcup \{V_{1,i}\})$$

For a node $u \in V_1$, if $u.op$ is already implemented in the target DL framework, $u$ must belong to some single-element subset.

(2) For each submodel $\boldsymbol{M}_{1,i} = \langle V_{1,i}, E_1 \cap (V_{1,i} \times V_{1,i}), P_1 \rangle$, our algorithm produces a corresponding $\boldsymbol{M}_{2,i} = \langle V_{2,i}, E_{2,i}, P_1 \rangle$:
  (a) $E_{2,i} \subseteq V_{2,i} \times V_{2,i}$, which is a set of intra-partition edges.
  (b) $\boldsymbol{M}_{1,i}$ and $\boldsymbol{M}_{2,i}$ take the same input tensor tuple, and they are semantically equivalent:

$$\mathcal{Z}'_{tp} = \mathcal{F}_{\boldsymbol{M}_{1,i}}(\mathcal{X}'_{tp}) \vdash_{\mathcal{A}^{\text{MMdnn}}} \mathcal{Z}'_{tp} = \mathcal{F}_{\boldsymbol{M}_{2,i}}(\mathcal{X}'_{tp})$$

(3) Suppose that $V_{1,i}$ has an edge to $V_{1,j}$ which delivers the tensor $\mathcal{X}$. Let $v_i$ be a node of $V_{2,i}$ which outputs $\mathcal{X}$, and $v_j$ be a node of $V_{2,j}$ which takes $\mathcal{X}$ as an input tensor. Then, $\mathcal{A}^{\text{MMdnn}}$ adds an inter-partition edge $(v_i, v_j)$ to the auxiliary set $E'_2$. Similarly, edges from $V_{2,k}$ to $V_{2,i}$ are also added.

(4) Finally, $\mathcal{A}^{\text{MMdnn}}$ produces

$$\boldsymbol{M}_2 = \langle \bigcup \{V_{2,i}\},\ (\bigcup \{E_{2,j}\}) \cup E'_2,\ P_1 \rangle$$

as the converted target model.

The above item 2b is the key to correctness. MMdnn employs the *operator selection* technique to achieve the goal: it identifies known subgraph patterns in advance and carefully builds the semantically equivalent subgraphs as templates; once a source subgraph is matched, the target equivalent is produced. Details of operator selection are presented in Section 4.4. By induction, it is not hard to prove that the target model $\boldsymbol{M}_2$ is semantically equivalent to $\boldsymbol{M}_1$. Hence, $\mathcal{A}^{\text{MMdnn}}$ satisfies the conditions of Definition 3.1.

# 4 DESIGN AND IMPLEMENTATION

## 4.1 Overview

Figure 1 shows the extensible architecture of MMdnn. As mentioned earlier, the whole conversion pipeline is divided into three phases from the compilation point of view: source model reconstruction, intermediate model transformation, and target model generation.

In the first phase, a *front-end parser* reads the source model from disk and reconstructs it to a computation graph represented by the source framework's intermediate representation (IR). We implement one front-end parser for each supported model format using the framework built-in model deserialization API. For example, the PyTorch front-end parser calls `torch.load()` to deserialize pickled object files to memory. Learnable parameters (*i.e.*, weights and biases) are loaded into either individual graph nodes or some global object depending on the framework implementation. Figure 2 demonstrates the simplified PyTorch front-end parser, which uses a Python dictionary `state_dict` to store the learnable parameters of each layer. If possible, the parser invokes framework functionalities (*e.g.*, TensorFlow's `strip_unused_lib` and

[3] For simplicity, we require that the hyperparameter set is invariant under the faithful model conversion.

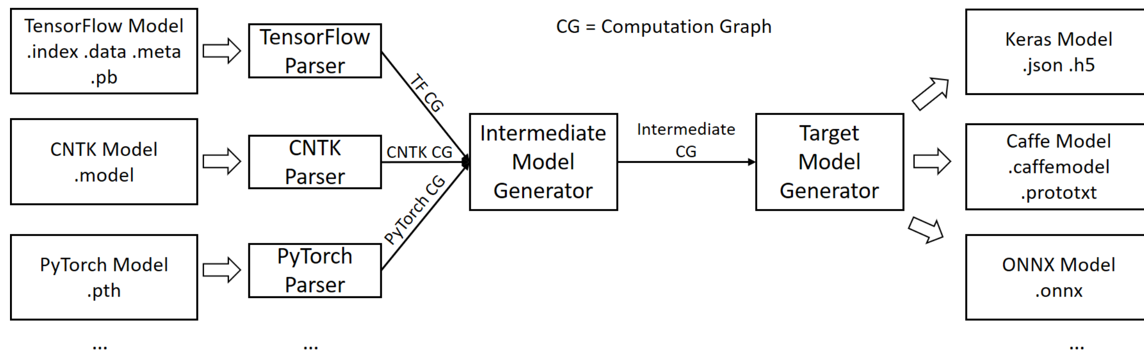


**Figure 1: Architecture of MMdnn.**

graph_transforms packages) to compact the source computation graph by removing unnecessary nodes and folding constants.

In the second phase, an *intermediate model generator* traverses the source computation graph in certain topological (linear) ordering and transforms it into an intermediate computation graph represented by the simple unified IR of MMdnn (Section 4.3). The generator employs *operator selection*,[4] a technique similar to compiler's instruction selection [6], to perform the node-to-node translation (Section 4.4). Most often, the operator of a source node has been defined or has an equivalent counterpart in MMdnn. Then, *macro expansion* is used for adding a new node with the corresponding operator to the intermediate computation graph. Properties of the source node such as input tensors, output tensors, and attributes are translated to comply with our syntax. Learnable parameters are transformed into NumPy [36] tensors and stored in a global dictionary object for centralized management. Furthermore, there exist complex cases involving the subgraph transformation. An example is to recover a TensorFlow fully-connected structure from low-level computation nodes. *DAG covering* is used to match predefined subgraph patterns of the source framework and emit intermediate subgraphs.

In the final phase, the intermediate computation graph is fed to a *target model generator*, which also employs the operator selection technique. However, it emits a piece of model construction code implemented with the target framework instead of an in-memory computation graph. Such code, together with the pre- and post-processing templates, is finally presented to developers for later retraining or serving. To export the target model file(s), the generator saves the converted learnable parameters and runs a dynamically crafted serialization program using the above model construction code. Figure 3 illustrates a simplified MXNet serialization program which constructs the MNIST [28] model, associates the saved learnable parameters with their belonging nodes, and calls the built-in model serialization API.

## 4.2 Technical Difficulties

In this section, we briefly mention some technical difficulties rooted from the discrepancies between DL frameworks:

[4]The name comes from the fact that a graph node is completely determined by its invoked operator.

```
1  class PyTorchParser(Parser):
2      def __init__(self, model_file_name, input_shape):
3          super(PyTorchParser, self).__init__()
4          # Load the model saved with torch.save().
5          model = torch.load(model_file_name)
6          self.weight_loaded = True
7          self.pytorch_graph = PyTorchGraph(model)
8          self.input_shape = tuple([1] + input_shape)
9          # Construct the intermediate computation graph.
10         self.pytorch_graph.build(self.input_shape)
11         self.state_dict = self.pytorch_graph.state_dict
12         self.shape_dict = self.pytorch_graph.shape_dict
```

**Figure 2: Simplified PyTorch front-end parser.**

(1) **Unavailable operators.** A source operator may not be available in the target framework. If it can be composed of other defined operators (*e.g.*, PyTorch's *log1p*), MMdnn translates the source node to a semantically equivalent subgraph. Otherwise (*e.g.*, PyTorch's *adaptive_avg_pool2d*), MMdnn defines it with a unified syntax and emits the placeholder code. Users need to implement such an operator in the target framework by reference to its original source code.
(2) **Inconsistent tensor layouts.** Some operators of the target framework may not support the original input tensor layout (NCHW vs. NHWC [23]). For example, the convolutional operators of PyTorch support only NCHW on GPU devices. Therefore, the target model generator must carefully transpose input/output tensors at the proper places, modify the learnable parameters and attributes, or even re-implement such operators to ensure the faithfulness. Details are presented in Section 4.5.
(3) **Unsupported padding.** Convolutional operators need an asymmetric padding if the kernel size is even. However, a few DL frameworks like Caffe only support symmetric padding which may cause conversion failures. MMdnn first pads a tensor with the maximum paddings and then crops it to the correct shape.
(4) **Incompatible argument type.** Non-constant operator arguments are usually defined as tensor variables; however,

```
1 def mnist_cnn():
2     conv2d_inp = mx.sym.var('conv2d_inp')
3     conv2d = mx.sym.Convolution(conv2d_inp,
4         kernel=(3,3), stride=(1,1), dilate=(1,1),
5         pad=(0,0), num_filter=32, num_group=1,
6         no_bias=False, layout='NCHW', name='conv2d'
7         )
8     conv2d_act = mx.sym.Activation(data=conv2d,
9         act_type='relu', name='conv2d_act')
10    ... # Omit some layers.
11    maxpool2d = mx.sym.Pooling(conv2d_2_act,
12        global_pool=False, kernel=(2,2), pad=(0,0),
13        stride=(2,2), pool_type='max',
14        name='maxpool2d')
15    dropout = mx.sym.Dropout(data=maxpool2d, p=0.25,
16        mode='training', name='dropout')
17    ... # Omit some layers.
18    dense = mx.sym.FullyConnected(dropout, num_hidden=10,
19        no_bias=False, name='dense')
20    dense_act = mx.sym.SoftmaxOutput(dense, 'softmax')
21    model = mx.mod.Module(dense_act, context=mx.cpu(),
22        data_names=['conv2d_inp'])
23    return model
24
25 def set_params(model, params_file_path):
26    arg_params = get_arg(params_file_path)
27    aux_params = get_aux(params_file_path)
28    model.bind(False,
29        data_shapes=[('conv2d_inp', (1,1,28,28))])
30    model.set_params(arg_params, aux_params, True)
31    return model
32
33 if __name__ == '__main__':
34    model = mnist_cnn()
35    model = set_params(model, saved_params_file_path)
36    model.save_checkpoint('mnist_cnn', epoch_num)
```

**Figure 3: Simplified serialization program to output the MXNet MNIST model whose source framework is Keras. `mnist_cnn()` constructs the model structure, `set_params()` associates the converted learnable parameters to their belonging nodes, and `model.save_checkpoint()` exports the model file.**

some DL frameworks allow to use Python objects instead. For example, the `begin` argument of the MXNet *slice* operator is a Python tuple [34]. Currently, MMdnn translates a Python object from or to a tensor variable by value. If the argument is not a constant, users need to fill in the most likely value. However, this method may lead to a conversion failure since the full computation history of the argument is lost. The root cause is that the computation of Python is separate from that of deep learning. It is possible to tackle the problem by performing code analysis on the entire DL program. A more complete solution depends on a general IR (*e.g.*, MLIR) which can uniformly represent the above two kinds of computation.

## 4.3 Intermediate Representation

The intermediate representation of MMdnn plays an important role to represent models from various DL frameworks as unified general-purpose computation graphs. We reject the ad-hoc direct conversion between every two frameworks to make MMdnn more extensible and reduce the difficulty of implementation and debugging. The goal of MMdnn is converting models faithfully instead of formulating a common model format; hence, the IR design can be simplified to focus on the graph syntax without caring a lot about the implementation of various IR constructs.

MMdnn refers to existing DL frameworks like ONNX and uses Protocol Buffers [59] to describe the following graph schema:

(1) **GraphDef**. This is the top-level construct for representing a computation graph. Edges are not explicitly defined since the node input tensors encode them.
(2) **NodeDef**. It represents a graph node with the fields of a name, an operator type, a list of zero or more named input tensors of type `string`, and a map of zero or more named attributes. Each input tensor has the "dep_node : dep_output" format which means that it is actually an output tensor named "dep_output" of the "dep_node" node. Therefore, the graph edges can be implicitly constructed from the corresponding tensor/node names. We also take the output tensors away because they are defined in the operator data structure.
(3) **AttrValue**. Attributes represent the node/operator arguments other than the input tensors. An attribute value is a runtime constant specified by developers or inferred by frameworks. Attribute names are not stored together since they appear in the node and operator data structures.
(4) **op**. This represents an operator, the mathematical operation invoked by a node. It consists of a name, two lists for the input and output tensor arguments respectively, and a list of named attributes.
(5) **TensorShape** and **LiteralTensor**. A tensor shape is a list of element numbers in each dimension, where $-1$ stands for an unknown dimension. A literal tensor represents a serialized tensor value which consists of a tensor shape, an element data type, and a flattened array of elements.

Initially, the operator set includes essential operators available in various frameworks, such as basic mathematical operators of addition, exponentiation and division, and layer operators of convolution and batch normalization [24]. To be noted, it is possible that DL frameworks have different syntax on the same operator. Let us take the *local response normalization* (LRN) [26] as an example. ONNX defines a `size` attribute (argument) which represents an odd number of channels to sum over. However, TensorFlow uses `depth_radius` instead, being equal to $\lfloor \frac{\texttt{size}}{2} \rfloor$ (rounding down). In CNTK (*BlockApiSetup.lrn*), `n` is used and equals $\lceil \frac{\texttt{size}}{2} \rceil$ (rounding up). MMdnn unifies the operator syntax: naming and ordering of input tensors, output tensors, and attributes. For instance, MMdnn refers to ONNX and adopts `size` as the formal attribute name of our *LRN* operator, whose definition is shown in Figure 4. Thus, when converting a TensorFlow *LRN* node, MMdnn should store ($\texttt{depth_radius} \times 2 + 1$) to the `size` attribute.

The operator set gradually grows on demand. When a new framework operator is met, we usually define it and formalize its syntax,

```
1 op {
2     name: "LRN"
3     input_arg  {name: "x" type_attr: "T"}
4     output_arg {name: "z" type_attr: "T"}
5     attr {name: "size"  type: "int"}
6     attr {name: "alpha" type: "float"}
7     attr {name: "beta"  type: "float"}
8     attr {name: "bias"  type: "float"}
9     summary: "Local Response Normalization."
10 }
```

**Figure 4: Definition of the *LRN* operator in MMdnn. The descriptions of each field are not shown.**

```
1 def Exp_from_tf(self, source):
2     ir = self.IR_Graph.add()
3     ir.op = 'Exp'
4     ir.name = source.name
5     ir.inputs = convert_input(source.inputs)
6     ir.attrs  = convert_attribute(source.attrs)
```

**Figure 5: Macro expansion from a TensorFlow *Exp* (exponentiation) node to an intermediate *Exp* node.**

even if it is semantically equivalent to a composition of other defined operators. To achieve an optimal converted model, MMdnn avoids decomposing a source operator at the phase of intermediate model generation since some target frameworks may have already implemented it. Note that a framework operator can act quite differently with certain framework-specific arguments. For example, the TensorFlow *MatMul* (matrix multiplication) operator has four unique Boolean arguments to control whether the two input tensors should be transposed and conjugated in advance [55]. Under such circumstances, adding all arguments to the formal operator definition for deferred processing, decomposing the source operator, or defining multiple new operators (*e.g.*, *TransposedMatMul*) is possible depending on the complexity.

## 4.4 Operator Selection

Operator selection is used by the intermediate and target model generators for the node-to-node translation. Given a computation graph, the generators traverse it and apply *macro expansion* on a node or *DAG covering* on a subgraph.

Macro expansion works by checking the operator of a node, locating the matched *expander function*, and executing the function code. Most often, a single node with the same or equivalent operator is emitted. Figure 5 and Figure 6 illustrate two expander functions that translate a TensorFlow *Exp* (exponentiation) node to an intermediate *Exp* node and then to a piece of code for generating a CNTK *Exp* node. Note that the expander function can emit a subgraph under certain cases (*e.g.*, expanding a *log1p* node to an *Add* node followed by a *log*).

The input computation graph is sometimes complicated and huge by the alteration of DL frameworks. For example, the TensorFlow *FullyConnected* and *BatchNormalization* nodes have been

```
1 def Exp_to_cntk(self, ir):
2     parent = self.parent(ir, pos=0)
3     code = "{} = cntk.exp({}, name='{}')"
4     .format(ir.name, parent.name, ir.name)
5 
6     return code
```

**Figure 6: Macro expansion from the intermediate *Exp* node in Figure 5 to a piece of code for generating a CNTK *Exp* node.**

optimized to subgraphs of low-level computation in the model file. Although translating one node by another using macro expansion ensures the correctness, it may not achieve an optimal converted model if the target framework has implemented the same high-level operators. MMdnn uses DAG covering to recover the original user-specified graph structures by searching for predefined *subgraph patterns* on the input computation graph. For instance, the *Fully-Connected* structure is implemented in TensorFlow by a *MatMul* node, a *Variable* node as *MatMul*'s grandfather, and surrounding others. Figure 7 and Figure 8 demonstrate how to recover such a TensorFlow *FullyConnected* structure from representative *MatMul* and *Variable* nodes and then translate it to an equivalent Caffe node using the *InnerProduct* operator.

RNN (recurrent neural network) [61] patterns are usually more sophisticated because DL frameworks unroll an RNN node (*e.g.*, LSTM [19] and GRU [12]) to a cell sequence whose length is not explicitly specified in the source model. Furthermore, TensorFlow unfolds each RNN cell to a subgraph of low-level computation nodes too. After having identified all the RNN cells by matching cell patterns, MMdnn infers the sequence length from the shape of RNN weights to recover the original RNN node. Nevertheless, when handling a TensorFlow RNN structure, we sometimes noticed unexpected degradation of the model learning performance no matter a high-level RNN node or a sequence of RNN cells were generated in the target model. We guess that the target frameworks may implement RNN cells in a different internal computation ordering. One workaround is to craft a sequence of custom nodes with each invoking a synthesized *cell function* identical to that of the corresponding TensorFlow RNN cell.

## 4.5 Tensor Layout

As mentioned earlier, tensors are multi-dimensional arrays. The tensor layout refers to the dimensional ordering, which is important because such dimensions usually have specific semantics. For example, 4D image tensors have four dimensions: *N*, *C*, *H*, and *W*, which represent the batch size, color channel, height, and width, respectively. Depending on whether the *C* dimension ranks ahead of the *H* and *W* dimensions, two tensor layouts `NCHW` (channels first) and `NHWC` (channels last) are widely used by different computing devices [23].

However, DL frameworks do not always support any tensor layouts. For instance, on GPU devices, TensorFlow accepts both `NCHW` and `NHWC` but PyTorch supports `NCHW` only. An unexpected tensor layout can cause the conversion to fail. Suppose that we

```python
def FullyConnected_from_tf(self, source):
    # self is a MatMul node.
    parent = self.parent(source, 1)
    W = self.parent(parent, 0)
    if 'Variable' in W.type:
        ir = self.IR_Graph.add()
        ir = self.copy(source, 'FullyConnected')
        units = source.attr['units']
        ir.attr['units'].i = units
        self.set_weight(source.name,
            'kernel', self.ckpt[W.name])
        add_node = self.child(source, 0)

    if add_node is not None:
        add_node.covered = True # Cover the add node.
        B = self.parent(self.parent(source, 1), 0)
        if B is not None:
            self.set_weight(source.name,
                'bias', self.ckpt_data[B.name])
            ir.attr['use_bias'].b = True
        else: # No bias.
            assign_ir(ir, {'use_bias': False})
```

**Figure 7: Simplified DAG covering from a subgraph containing TensorFlow *MatMul, Variable*, etc. nodes to an intermediate *FullyConnected* node.**

```python
def FullyConnected_to_caffe(self, ir):
    # check whether to transpose the weights
    if self.check(ir):
        transpose_ir_weight(ir)

    code = "n.{:<15} = L.InnerProduct(n.{}, num_output={},
    ↪  bias_term={}, ntop=1)"
    .format(ir.name, self.parent_name(ir),
        ir.layer.attr["units"].i,
        ir.get_attr('use_bias', False))

  return code
```

**Figure 8: Macro expansion from the intermediate *FullyConnected* node in Figure 7 to a piece of code for generating an equivalent Caffe node using the *InnerProduct* operator.**

are converting a TensorFlow model trained with NHWC input data to the PyTorch format. We assume that the target model still uses NHWC input data. If the source model contains *Conv2D*, *MaxPooling*, *BatchNormalization*, etc. nodes, the converted PyThorch model will produce totally wrong results. This is because the target *Conv2D* etc. nodes encounter an *inconsistent tensor layout* other than NCHW, which is either unsupported (*e.g.*, NHWC) or completely unknown (*e.g.*, the input is produced by another convolutional node which has not handled the layout issue). Simply inserting an NHWC-to-NCHW tensor transpose just before and an NCHW-to-NHWC immediately after each of them does not solve the problem. The reason is that their learnable parameters and attributes, being part of the operator implementation, are bound to the source NHWC layout and should be amended too. In fact, the root cause comes from that tensor transpose and certain operators are not *commutative*. In the following, we give the formal notations [40] and definition.

Assume that set $S = \{1, 2, \cdots, n\}$, $\mathcal{X}$ is an $n$-order tensor, $\sigma$ is a permutation of $S$, and $\sigma^{-1}$ is the inverse of $\sigma$. $\sigma$ is actually a one-to-one function from $S$ onto $S$ and is denoted as follows, with $\sigma(i) = k_i$:

$$\sigma = \begin{pmatrix} 1 & 2 & 3 & \cdots & n \\ k_1 & k_2 & k_3 & \cdots & k_n \end{pmatrix}$$

We call $\sigma$ a *tensor layout* of $\mathcal{X}$. For instance, if we take NHWC as the identity layout (permutation), NCHW and its inverse are then denoted as follows:

$$\sigma_{\text{NCHW}} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 4 & 2 & 3 \end{pmatrix} \qquad {\sigma_{\text{NCHW}}}^{-1} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 3 & 4 & 2 \end{pmatrix}$$

An $n$-order tensor $\mathcal{Y}$ is called the *transpose* of $\mathcal{X}$ associated with $\sigma$ and denoted by $\mathcal{X}^{T_\sigma}$, if

$$\mathcal{Y}(i_{\sigma(1)}, i_{\sigma(2)}, \cdots, i_{\sigma(n)}) = \mathcal{X}(i_1, i_2, \cdots, i_n)$$

Let $\mathcal{X}_{tp} = \langle \mathcal{X}_1, \mathcal{X}_2, \cdots, \mathcal{X}_m \rangle$ be a tensor tuple which represents the input or output of a node/operator. $\sigma_{tp} = \langle \sigma_1, \sigma_2, \cdots, \sigma_m \rangle$ is a permutation tuple with each $\sigma_i$ being a permutation of $\mathcal{X}_i$'s $S$ set. $\sigma_{tp}$ is called a *tensor layout* of $\mathcal{X}_{tp}$. We then denote the transpose of $\mathcal{X}_{tp}$ as follows:

$${\mathcal{X}_{tp}}^{T_{\sigma_{tp}}} = \langle {\mathcal{X}_1}^{T_{\sigma_1}}, {\mathcal{X}_2}^{T_{\sigma_2}}, \cdots, {\mathcal{X}_m}^{T_{\sigma_m}} \rangle$$

Now we define the commutativity of tensor transpose and operators.

*Definition 4.1.* Assume that $op$ is an operator, and

$$\mathcal{X}_{tp} = \langle \mathcal{X}_1, \mathcal{X}_2, \cdots, \mathcal{X}_m \rangle, \quad \mathcal{Z}_{tp} = \langle \mathcal{Z}_1, \mathcal{Z}_2, \cdots, \mathcal{Z}_n \rangle$$

are $op$'s input/output tensor tuples satisfying Equation 1.

$$\sigma_{tp} = \langle \sigma_1, \sigma_2, \cdots, \sigma_m \rangle, \quad {\sigma'}_{tp} = \langle {\sigma'}_1, {\sigma'}_2, \cdots, {\sigma'}_n \rangle$$

are tensor layouts of $\mathcal{X}_{tp}$ and $\mathcal{Z}_{tp}$, respectively. The operator $op$ is called to be *transpose-commutative* associated with $\sigma_{tp}$ if

$${\mathcal{Z}_{tp}}^{T_{{\sigma'}_{tp}}} = \mathcal{F}_{op}({\mathcal{X}_{tp}}^{T_{\sigma_{tp}}}) \tag{2}$$

In case $\sigma_{tp}$ is arbitrary and there always exists the corresponding ${\sigma'}_{tp}$, $op$ is transpose-commutative.

It is not hard to realize that operators such as tensor element-wise addition, subtraction, and multiplication with a scalar are transpose-commutative, while the above mentioned *Conv2D*, *MaxPooling*, and *BatchNormalization* are not. If a transpose-commutative operator (associated with certain permutations) encounters an inconsistent tensor layout, inserting two tensor transposes just before and immediately after it guarantees the correctness, given the previously mentioned $\sigma_{tp}$ and ${\sigma'}_{tp}$. The core task is to pre-identify the target nodes whose operators are non-transpose-commutative under current tensor layout assumption. We then make them transpose-commutative associated with the assumed tensor layout by modifying the learnable parameters/attributes or re-implementing the operators. For the TensorFlow-to-PyTorch example, we transpose the PyTorch *Conv2D* node's filters from [filter_height, filter_width,

in_channels, out_channels] to [out_channels, in_channels, filter_height, filter_width]. Note that some DL frameworks implement *self-adaptive* operators to support multiple tensor layouts. Users only need to explicitly specify the actual input tensor layout. For instance, the TensorFlow *Conv2D* operator has an optional string argument `data_format` which defaults to `NHWC`. Hence, we think that *Conv2D* is transpose-commutative associated with $\sigma_{\mathtt{NCHW}}$ in TensorFlow, assuming that $\sigma_{\mathtt{NHWC}}$ is the identity permutation. Algorithm 1 suggests a general approach to handle inconsistent tensor layouts, with the optimization of eliminating consecutive reciprocal tensor transposes.

At the beginning of this section, we assume that the target model uses the same initial input tensor layout as the source model. Therefore, necessary tensor transposes must be inserted, which may significantly reduce the runtime performance. A more attractive idea is transposing only the initial input and final output to avoid those inserted tensor transposes. However, such a method requires that we can always determine the actual input and output tensor layouts of each node/operator, which is challenging due to the dynamic node/operator amendment. Moreover, extra tensor transposes may also be imposed. It remains an interesting future research problem to seek the best initial tensor layout that achieves both correctness and optimal runtime performance.

**Algorithm 1:** Handle inconsistent tensor layouts at the target model generation phase.

```
Input: The intermediate model inter_model;
Output: The target model target_model with inserted tensor
        transposes and possible node amendment;
 1 inter_topo ← GetTopologicalOrder(inter_model) ;
 2 foreach node ∈ inter_topo do
 3     new_node ← EmitNode (target_model, node) ;
 4     old_in_layout ← GetInputLayout(new_node) ; // A tuple
         ⟨l1, ··· , lm⟩ whose li is the layout of i-th input tensor.
 5     allowed_input_layouts ←
         GetTargetAllowedInputLayouts(new_node.op) ;
 6     if old_in_layout ∈ allowed_input_layouts then
 7         continue;
 8     end
 9     old_out_layout ← GetOutputLayout(new_node) ;
10     new_in_layout ← SelectTargetInputLayout(new_node,
         allowed_input_layouts) ;
       // Assume old_in_layout is the identity layout.
11     if not IsCommutativeInTarget(new_node, old_in_layout,
         new_in_layout) then
12         MakeCommutativeInTarget(new_node, old_in_layout,
             new_in_layout) ;
13     end
       // Emit necessary tensor transposes for input tensors.
14     parents ← GetParents(inter_model, node) ;
15     for i ← 1 to parents.GetLength() do
16         parent ← parents [i] ;
17         parent_transpose ← GetOutputTranspose(target_model,
             parent, new_node) ; // Emitted by parent at Line 33
18         if parent_transpose ≠ NULL then
19             l1 ← old_in_layout [i] ;
20             l2 ← new_in_layout [i] ;
21             if IsInverse(parent_transpose, l1, l2) then
                   // Optimize parent_transpose away.
22                 RemoveNode(target_model, parent_transpose) ;
23             else
24                 EmitInputTranspose(target_model, new_node,
                     l1, l2) ;
25             end
26         end
27     end
       // Emit necessary tensor transposes for output tensors.
28     new_out_layout ← GetOutputLayout(new_node.op,
         new_in_layout) ;
29     for i ← 1 to old_out_layout.GetLength() do
30         l1 ← old_out_layout [i] ;
31         l2 ← new_out_layout [i] ;
32         if l1 ≠ l2 then
               // Emit an l2-to-l1 tensor transpose.
33             EmitOutputTranspose(target_model, new_node, l2,
                 l1) ;
34         end
35     end
36 end
```

# 5 EVALUATION

MMdnn is implemented in Python with over 29,000 lines of code. We choose official TensorFlow/PyTorch/CNTK/MXNet/Caffe ResNet-152 models [7, 14, 32, 43, 57] and TensorFlow/PyTorch/CNTK/MXNet Inception-V3 models [13, 33, 42, 56] as our experimental objects. The target frameworks additionally include Keras (TensorFlow backend) and ONNX. The source models were trained using ImageNet classification dataset with 1,000 classes of objects [48]. Except that the source and target models of TensorFlow use the `NHWC` tensor layout, other models use `NCHW`. The framework versions are shown in Table 1 and the MMdnn version is v0.2.5. Our experiments are conducted on an Ubuntu 16.04 workstation with 16 Intel Xeon E5-2665 CPUs and 128 GB main memory.

We randomly select 4,000 images in RGB/BGR formats from COCO [29] 2017 Test dataset[5] to evaluate the effectiveness of MMdnn. The test images are preprocessed by size refactoring (to 224), normalization, and transposition. Normalization includes the following three methods:

(1) Standard: divide each pixel by 255, then subtract 0.5, and finally multiply by 2.
(2) Zero Center: subtract 123.68, 116.779, and 103.939 from the three color channels, respectively.
(3) Identity: no processing.

Suppose that $m$ is the number of test images and $\mathcal{X}_1, \mathcal{X}_2, \cdots, \mathcal{X}_m$ denote the image tensors. Let $\mathbf{y}_i$ and $\mathbf{z}_i$ ($i \in [1, m]$) be two result vectors in the Euclidean space $\mathbb{R}^n$, computed by the source and target models on $\mathcal{X}_i$, respectively. $n$ is the number of object classes, which equals 1,000.[6] $y_{ij}$ and $z_{ij}$ denote the $j$-th components of $\mathbf{y}_i$ and $\mathbf{z}_i$, respectively. To evaluate whether the target model classifies

[5] There are 41,000 images in total.

[6] $n$ = 1,001 for the source TensorFlow Inception-V3 model "since 0 is reserved for the background class" [54].

**Table 1: Evaluation results of ResNet-152 models.**

| Source | Data Preprocessing | TensorFlow (Version: 1.13.1) | PyTorch (0.4.0) | CNTK (2.6) | MXNet (1.2.0) | Caffe (1.0) | Keras (2.2.4) | ONNX (1.4.1) |
|---|---|---|---|---|---|---|---|---|
| TensorFlow | Zero Center | Top-10 ACC (%): 100<br>MRE: 4.25e-6 | 100<br>7.11e-6 | 100<br>0.443 | 100<br>7.37e-6 | 100<br>6.68e-6 | 100<br>0 | 100<br>0.254 |
| PyTorch | Standard<br>Transposition | 100<br>6.78e-6 | 100<br>2.00e-6 | 100<br>6.72e-6 | 100<br>3.80e-6 | 100<br>5.57e-6 | 100<br>7.31e-4 | 100<br>6.76e-6 |
| CNTK | Identity<br>Transposition | 100<br>0.359 | 100<br>6.45e-6 | 100<br>4.96e-6 | 100<br>6.74e-6 | 100<br>8.13e-6 | 100<br>0.359 | 100<br>5.82e-6 |
| MXNet | Identity<br>Transposition | 100<br>6.73e-6 | 100<br>2.28e-6 | 100<br>5.59e-6 | 100<br>0 | 100<br>3.95e-6 | 100<br>4.22e-6 | 100<br>6.87e-6 |
| Caffe | Zero Center<br>Transposition | 100<br>4.39e-6 | 100<br>3.75e-6 | 100<br>5.99e-6 | 100<br>3.82e-6 | 100<br>0 | 100<br>1.22e-5 | 100<br>1.552 |

**Table 2: Evaluation results of Inception-V3 models.**

| Source | Data Preprocessing | TensorFlow (Version: 1.13.1) | PyTorch (0.4.0) | CNTK (2.6) | MXNet (1.2.0) | Caffe (1.0) | Keras (2.2.4) | ONNX (1.4.1) |
|---|---|---|---|---|---|---|---|---|
| TensorFlow | Standard | Top-10 ACC (%): 100<br>MRE: 2.94e-5 | 100<br>7.29e-5 | 100<br>4.37e-5 | 100<br>1.206 | 100<br>1.206 | 100<br>0 | 100<br>4.64e-5 |
| PyTorch | Standard<br>Transposition | 100<br>2.38e-5 | 100<br>5.01e-6 | 100<br>1.74e-5 | 100<br>0.196 | 100<br>0.196 | 100<br>1.99e-5 | 100<br>2.31e-5 |
| CNTK | Identity<br>Transposition | 100<br>1.40e-5 | 100<br>1.44e-5 | 100<br>1.29e-5 | 100<br>0.284 | 100<br>0.284 | 100<br>2.67e-4 | 100<br>9.84e-6 |
| MXNet | Identity<br>Transposition | 100<br>0.305 | 100<br>0.305 | 100<br>0.305 | 100<br>0 | 100<br>3.32e-5 | 100<br>0.305 | 100<br>0.305 |

an image correctly, we use *Top-10 accuracy*: if the indices of the 10 largest components in $\mathbf{z}_i$ are identical to those in $\mathbf{y}_i$, the accuracy on $\mathcal{X}_i$ ($\text{ACC}_i$) is 1; otherwise, it is 0. We also calculate *relative error* ($\text{RE}_i$) [62] to assess the distance between $\mathbf{y}_i$ and $\mathbf{z}_i$.

$$\text{ACC}_i = \begin{cases} 1 & \text{if } \mathbf{y}_i \text{ and } \mathbf{z}_i \text{ have identical Top-10 component indices,} \\ 0 & \text{otherwise.} \end{cases}$$

$$\text{RE}_{ij} = \left| \frac{z_{ij} - y_{ij}}{y_{ij}} \right|$$

$$\text{RE}_i = \frac{\sum_{j=1}^{n} \text{RE}_{ij}}{n}$$

Then, we measure the experiment using Top-10 accuracy (ACC) in percent and mean relative error (MRE):

$$\%\ \text{ACC} = \frac{\sum_{i=1}^{m} \text{ACC}_i}{m} \times 100$$

$$\text{MRE} = \frac{\sum_{i=1}^{m} \text{RE}_i}{m}$$

A larger ACC means that the learning performances of the source and target models are closer; a smaller MRE indicates that the two result vectors have minor difference.

Table 1 and Table 2 demonstrate the evaluation results of ResNet-152 and Inception-V3 models. Each cell has two lines of values, reporting Top-10 accuracy and MRE, respectively. From the results, we find that MMdnn achieves perfect Top-10 accuracy (100%), confirming its effectiveness. We also notice that MRE $< 0.001$ in many cases, which indicates that the overall difference between the two result vectors is small. Some MRE values are relatively large, for example, MRE of the Caffe-to-ONNX case in Table 1 reaches 1.552. However, since the Top-10 accuracy is 100%, we think that the differences come from insignificant vector components and do not influence the model learning performance.

## 6 RELATED WORK

**Machine learning/deep learning compilers.** TVM [9] is a deep learning compiler stack that compiles models into minimum deployable modules on diverse hardware backends. Flux [22], built upon the Julia programming language [5], provides a new and elegant machine learning stack for Julia developers. TensorFlow XLA [1] and DLVM [60] are domain-specific compilers, optimizing on high-level computation graphs. Such compilers aim at achieving the optimal training/serving runtime performance. Although MMdnn adopts compiler principles (*e.g.*, IR and processing phases), it only focuses on the faithful model conversion.

**Common model formats.** ONNX [37] and NNEF [35] are open neural network exchange formats, with a similar motivation for the framework interoperability. They include essential operators which are less than those in popular DL frameworks. For example, there are about 137 and 115 operators in ONNX v1.5 and NNEF 1.0.1 respectively, while TensorFlow r1.13 has more than 500. Therefore,

if the source model uses an out-of-range operator, it cannot be converted or directly exported[7] to such two formats. ONNX and NNEF need to keep pace with the frameworks and hardware vendors to support new operators, which limits their extensibility. MMdnn refers to the syntax of ONNX and other frameworks to design its simple yet unified IR, with the purpose of an intermediate medium to describe as many IR constructs as possible. For a source operator, it is more likely that the target framework has already implemented the same or an equivalent operator. In addition, MMdnn can extend its operator set via operator decomposition and porting. Hence, MMdnn has the potential to quickly support more DL operators, models, and frameworks for the model conversion.

**Model converters.** There are a number of model converters such as caffe-tensorflow [15], ONNXMLTools [38], WinMLTools [30], Core ML Tools [3], tf2onnx [39], NNEF-Tools [17], etc. They provide unidirectional conversion, requiring that the operators are from the intersection of both source and target frameworks. MMdnn adopts a unified IR-based methodology to perform bidirectional conversion between more DL frameworks and support a broader range of operators.

## 7 CONCLUSION AND ON-GOING WORK

MMdnn is an open-sourced, comprehensive, and faithful model conversion tool to enhance the interoperability between popular DL frameworks. It adopts a novel unified intermediate representation-based methodology and implements an extensible conversion architecture to ease contribution from the community. MMdnn has reached good maturity and quality, and is applied for converting production models.

One immediate on-going work is to convert NLP pre-trained models like BERT [16], which requires to support more operators such as *LayerNorm*, *BatchMatMul*, *Cast*. We also wish to support control-flow constructs (*e.g.*, *tensorflow::ops::Switch*, *mxnet.ndarray.contrib.while_loop*, and ONNX *Loop*) and dynamic RNNs (*e.g.*, *tf.nn.dynamic_rnn*). Since the number of operators is increasing rapidly, it is inefficient to manually understand their semantics, identify the equivalents, and port them to another framework. Some program analysis techniques may be developed to facilitate the process. However, we think that using some standard domain-specific language (DSL) to define both syntax and semantics of the operators could help not only convert models but also increase the training/serving runtime performance. DL frameworks are evolving fast, so we will continuously keep up with their latest versions. Another work is to reduce the overhead of inserted tensor transposes when handling the inconsistent tensor layout. We also hear from developers that they want to port existing DL programs to another framework faithfully for training, which needs a new DL transpiler (aka source-to-source compiler).

## ACKNOWLEDGMENTS

We gratefully acknowledge the valuable feedback and contributions from the community that greatly influenced the design and development of MMdnn. We thank all the previous MMdnn team members including Jiahao Yao, Yuhao Zhou, Tong Zhan, and Qianwen Wang. We thank all the contributors for their submitted questions, suggestions, documents, features, bug fixes, etc.

[7] CNTK/Caffe2/MXNet/PyTorch can export models to the ONNX format.

## REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, and et al. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI '16)*. USENIX Association, USA, 265–283.
[2] Apple. 2017. Apple Core ML. https://developer.apple.com/documentation/coreml.
[3] Apple. 2017. Core ML Tools. https://coremltools.readme.io/docs.
[4] Baidu. 2016. PaddlePaddle: PArallel Distributed Deep LEarning. https://github.com/paddlepaddle/paddle.
[5] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. 2017. Julia: A fresh approach to numerical computing. *SIAM review* 59, 1 (2017), 65–98. https://doi.org/10.1137/141000671
[6] G.H. Blindell. 2016. *Instruction Selection: Principles, Methods, and Applications*. Springer International Publishing. 1–177 pages.
[7] Caffe. 2019. Caffe ResNet-152. prototxt: http://data.mxnet.io/models/imagenet/test/caffe/ResNet-152-deploy.prototxt params: http://data.mxnet.io/models/imagenet/test/caffe/ResNet-152-deploy.prototxt.
[8] 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
[9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, and et al. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI '18)*. USENIX Association, USA, 579–594.
[10] François Chollet. 2016. Xception: Deep Learning with Depthwise Separable Convolutions. *CoRR* abs/1610.02357 (2016). arXiv:1610.02357 http://arxiv.org/abs/1610.02357
[11] François Chollet et al. 2015. Keras. https://keras.io.
[12] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. 2014. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. *CoRR* abs/1412.3555 (2014). arXiv:1412.3555 http://arxiv.org/abs/1412.3555
[13] CNTK. 2019. CNTK Inception-V3. https://www.cntk.ai/Models/CNTK_Pretrained/InceptionV3_ImageNet_CNTK.model.
[14] CNTK. 2019. CNTK ResNet-152. https://www.cntk.ai/Models/CNTK_Pretrained/ResNet152_ImageNet_CNTK.model.
[15] Saumitro Dasgupta. 2015. Caffe to TensorFlow. https://github.com/ethereon/caffe-tensorflow.
[16] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR* abs/1810.04805 (2018). arXiv:1810.04805 http://arxiv.org/abs/1810.04805
[17] Khronos Group. 2017. NNEF-Tools. https://github.com/KhronosGroup/NNEF-Tools.
[18] 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
[19] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (Nov. 1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735
[20] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *CoRR* abs/1704.04861 (2017). arXiv:1704.04861 http://arxiv.org/abs/1704.04861
[21] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. *CoRR* abs/1602.07360 (2016). arXiv:1602.07360 http://arxiv.org/abs/1602.07360
[22] Michael Innes, Elliot Saba, Keno Fischer, Dhairya Gandhi, Marco Concetto Rudilosso, Neethu Mariya Joy, Tejan Karmali, Avik Pal, and Viral Shah. 2018. Fashionable Modelling with Flux. *CoRR* abs/1811.01457 (2018). arXiv:1811.01457 http://arxiv.org/abs/1811.01457
[23] Intel. 2019. Understanding Memory Formats. https://intel.github.io/mkl-dnn/understanding_memory_formats.html.
[24] Sergey Ioffe and Christian Szegedy. 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37* (Lille, France) *(ICML '15)*. JMLR.org, 448–456.

[25] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross B. Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. *CoRR* abs/1408.5093 (2014). arXiv:1408.5093 http://arxiv.org/abs/1408.5093
[26] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems* (Lake Tahoe, Nevada) *(NIPS '12)*. Curran Associates Inc., Red Hook, NY, USA, 1097–1105.
[27] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A Compiler Infrastructure for the End of Moore's Law. arXiv:2002.11054 [cs.PL] https://arxiv.org/abs/2002.11054
[28] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.
[29] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, Lubomir D. Bourdev, Ross B. Girshick, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. 2014. Microsoft COCO: Common Objects in Context. *CoRR* abs/1405.0312 (2014). arXiv:1405.0312 http://arxiv.org/abs/1405.0312
[30] Microsoft. 2018. Convert ML models to ONNX with WinMLTools. https://docs.microsoft.com/en-us/windows/ai/windows-ml/convert-model-winmltools.
[31] MMdnn. 2018. MMdnn: Model Management for deep neural networks. https://github.com/microsoft/MMdnn.
[32] MXNet. 2019. MXNet imagenet1k-resnet-152. symbol: http://data.mxnet.io/models/imagenet/resnet/152-layers/resnet-152-symbol.json params: http://data.mxnet.io/models/imagenet/resnet/152-layers/resnet-152-0000.params.
[33] MXNet. 2019. MXNet Inception-V3. symbol: http://data.mxnet.io/models/imagenet/inception-bn/Inception-BN-symbol.json params: http://data.mxnet.io/models/imagenet/inception-bn/Inception-BN-0126.params.
[34] MXNet. 2020. The Slice Symbol API of MXNet. https://mxnet.incubator.apache.org/versions/1.6/api/python/docs/api/symbol/symbol.html#mxnet.symbol.slice.
[35] NNEF. 2017. Neural Network Exchange Format. https://www.khronos.org/nnef.
[36] Travis Oliphant. 2006–. NumPy: A guide to NumPy. USA: Trelgol Publishing. http://www.numpy.org/
[37] ONNX. 2017. Open Neural Network Exchange. https://onnx.ai/.
[38] ONNX. 2018. ONNXMLTools. https://github.com/onnx/onnxmltools.
[39] ONNX. 2018. tf2onnx - Convert TensorFlow models to ONNX. https://github.com/onnx/tensorflow-onnx.
[40] Ran Pan. 2014. Tensor Transpose and Its Properties. *CoRR* abs/1411.1503 (2014). arXiv:1411.1503 http://arxiv.org/abs/1411.1503
[41] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 8024–8035.
[42] PyTorch. 2019. PyTorch Inception-V3. https://download.pytorch.org/models/inception_v3_google-1a9a5a14.pth.
[43] PyTorch. 2019. PyTorch ResNet-152. resnet152 of torchvision 0.2.1 https://download.pytorch.org/models/resnet152-b121ed2d.pth.
[44] Joseph Redmon. 2013–2016. Darknet: Open Source Neural Networks in C. http://pjreddie.com/darknet/.
[45] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. 2015. You Only Look Once: Unified, Real-Time Object Detection. *CoRR* abs/1506.02640 (2015). arXiv:1506.02640 http://arxiv.org/abs/1506.02640
[46] Jared Roesch, Steven Lyubomirsky, Marisa Kirisame, Josh Pollock, Logan Weber, Ziheng Jiang, Tianqi Chen, Thierry Moreau, and Zachary Tatlock. 2019. Relay: A High-Level IR for Deep Learning. *CoRR* abs/1904.08368 (2019). arXiv:1904.08368 http://arxiv.org/abs/1904.08368
[47] Grzegorz Rozenberg (Ed.). 1997. *Handbook of Graph Grammars and Computing by Graph Transformations*. World Scientific.
[48] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)* 115, 3 (2015), 211–252. https://doi.org/10.1007/s11263-015-0816-y
[49] Florian Schroff, Dmitry Kalenichenko, and James Philbin. 2015. FaceNet: A Unified Embedding for Face Recognition and Clustering. *CoRR* abs/1503.03832 (2015). arXiv:1503.03832 http://arxiv.org/abs/1503.03832
[50] Frank Seide and Amit Agarwal. 2016. CNTK: Microsoft's Open-Source Deep-Learning Toolkit. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD '16)*. Association for Computing Machinery, New York, NY, USA, 2135. https://doi.org/10.1145/2939672.2945397
[51] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science* 362, 6419 (2018), 1140–1144. https://doi.org/10.1126/science.aar6404 arXiv:https://science.sciencemag.org/content/362/6419/1140.full.pdf
[52] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. http://arxiv.org/abs/1409.1556
[53] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. 2015. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 1–9.
[54] TensorFlow. 2019. Data for the ImageNet ILSVRC 2012 Dataset plus some bounding boxes. https://github.com/tensorflow/models/blob/v1.13.0/research/slim/datasets/imagenet.py#L83.
[55] TensorFlow. 2019. The Matrix Multiplication of TensorFlow. https://github.com/tensorflow/docs/blob/r1.13/site/en/api_docs/python/tf/linalg/matmul.md.
[56] TensorFlow. 2019. TensorFLow Inception-V3. http://download.tensorflow.org/models/inception_v3_2016_08_28.tar.gz.
[57] TensorFlow. 2019. TensorFLow ResNet-V1-152. pre-trained model: http://download.tensorflow.org/models/resnet_v1_152_2016_08_28.tar.gz; dataset: ILSVRC-2012-CLS.
[58] TensorFlow. 2020. Save and load models. https://www.tensorflow.org/tutorials/keras/save_and_load#save_checkpoints_during_training.
[59] Kenton Varda. 2008. *Protocol Buffers: Google's Data Interchange Format*. Technical Report. Google. http://google-opensource.blogspot.com/2008/07/protocol-buffers-googles-data.html
[60] Richard Wei, Vikram S. Adve, and Lane Schwartz. 2017. DLVM: A modern compiler infrastructure for deep learning systems. *CoRR* abs/1711.03016 (2017). arXiv:1711.03016 http://arxiv.org/abs/1711.03016
[61] P. J. Werbos. 1990. Backpropagation through time: what it does and how to do it. *Proc. IEEE* 78, 10 (1990), 1550–1560.
[62] Wikipedia. 2020. Approximation error — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Approximation%20error&oldid=961618101. [Online; accessed 06-September-2020].
[63] Wikipedia. 2020. Comparison of deep-learning software — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Comparison%20of%20deep-learning%20software&oldid=971790766. [Online; accessed 06-September-2020].
[64] Wikipedia. 2020. Interoperability — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Interoperability&oldid=976199023. [Online; accessed 06-September-2020].
[65] Saining Xie, Ross B. Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. 2016. Aggregated Residual Transformations for Deep Neural Networks. *CoRR* abs/1611.05431 (2016). arXiv:1611.05431 http://arxiv.org/abs/1611.05431
[66] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. 2017. Learning Transferable Architectures for Scalable Image Recognition. *CoRR* abs/1707.07012 (2017). arXiv:1707.07012 http://arxiv.org/abs/1707.07012