ABSTRACT

Deep learning (DL) has become one of the most successful machine learning techniques. To achieve the optimal development result, recently there are emerging requirements for the interoperability between DL frameworks by re-utilizing existing model files and training/serving programs. Faithful model conversion is a promising technology to enhance the framework interoperability such that a source model is transformed into the semantic equivalent in target framework format. However, several major challenges need to be addressed. First, there are obvious discrepancies in both the computation graph constructs and supported features between DL frameworks. Second, semantic comprehension of model files could be quite complicated due to framework optimization or runtime execution mode. Lastly, there exist a large number of popular DL frameworks, bringing potential significant engineering efforts.

In this paper, we propose MMdnn, an open-sourced, comprehensive, and faithful model conversion tool for ten popular DL frameworks. MMdnn adopts a novel unified intermediate representation (IR)-based methodology to systematically handle such conversion challenges. The source model will be first transformed into an intermediate computation graph depicted with the simple graph-based IR of MMdnn, and then to the target framework format, which greatly reduces the engineering complexity. Since the model syntax expressed by developers may have been changed a lot 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 extendable 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, natural language processing, and board games. Developers use deep learning frameworks to design layered data representations called neural networks or 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 model construction, which greatly improves programmability and boosts productivity. There are already lots of widely used DL frameworks ¹ for both research and production: TensorFlow [1], PyTorch [40], Microsoft Cognitive Toolkit (CNTK) [46], Caffe [26], Apache MXNet [8], Keras [11], Baidu PaddlePaddle [4], Darknet [42], Apple Core ML [2], ONNX [36] ², etc., with each having its own uniqueness on expressivity, usability, and performance.

Recently, there are emerging requirements for the interoperability [14] between the above DL frameworks, such that available model files and training/serving programs implemented with one framework could be easily ported and re-utilized with another. 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 believe that 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 target 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 with the converted model file(s) and their new framework. 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 obvious discrepancies in both the computation graph constructs and supported features between DL frameworks. For example, PyTorch provides the `log1p` operator calculating `log(1 + x)`, which is unavailable in CNTK, Keras, and ONNX. Simply creating a placeholder node in the target model does not work, yet we must implement such `log1p` function

²ONNX provides not only "an open source format for AI models" but also "a cross-platform inferencing and training runtime" (aka.ms/onnxruntime).
with existing operators (e.g., using an Add operator followed immediately by a log operator). Another example is that not all frameworks support both NCHW and NHWC tensor layouts [24]. Hence, additional transpositions of the data and learnable parameters may be introduced in the target model.

(2) The source model syntax encoded in the file(s) can be quite different and complicated to that expressed by developers in the original training program due to framework optimization or runtime execution mode. This makes semantic comprehension much harder and results in 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 the PyTorch model file stores a dynamic computation graph defined by a real run, which could lose the conditional or loop information in code. Hence, reverse syntax transformation to recover the original DL constructs is needed for the source model.

(3) There are a large number of popular DL frameworks, bringing potential significant engineering efforts. Such an issue 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\(^3\), an open-sourced, comprehensive, and faithful model conversion tool. Comparing with existing 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 DL frameworks like ONNX, which aims at depicting as many IR constructs of current frameworks as possible and eliminating their syntactic differences (e.g., naming of operator arguments). The source model will be first transformed into an intermediate computation graph described with our IR and then to the target, using a node-to-node translation technique similar to compiler’s instruction selection [6]. Hence, the engineering complexity is significantly reduced from \(O(N \times M)\) to \(O(N + M)\), where \(N\) and \(M\) are the numbers of source and target frameworks, respectively. MMdnn also performs reverse syntax transformation on source models for better semantic comprehension via a pattern matching similar method (e.g., recovering a TensorFlow fully-connected structure from a subgraph). 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. MMdnn 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. The architecture enables modularization and automation, so as to ease 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 used it to deploy the converted production models (Caffe to ONNX). Up to now, MMdnn is able to convert VGG [47], ResNet [19], ResNetXt [52], Xception [10], Inception [48], MobileNets [21], SqueezeNet [22], NasNet [53], FaceNet [45], YOLO [43], etc. models for the previously mentioned ten frameworks\(^4\).

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 mathematical functions, models are formalized by DL frameworks as tensor-oriented computation graphs. Such a computation graph is a directed acyclic graph (DAG) in which graph/node inputs and outputs are tensor (multi-dimensional arrays of numerical values) variables, and each node represents the invocation of a mathematical operation called an operator (e.g., matrix multiplication). Because a node is completely decided by its invoked operator, we may use ‘node’ and ‘operator’ interchangeably in the sequel. A node has a list of tensor inputs with each referring to either a node output or a graph input. Similarly, it has a list of tensor outputs as the operator result with each either referring to a graph output or bringing a value to another node. Therefore, an edge on the graph pointing from one output of node \(A\) to one input of \(B\) allows tensor(s) to flow across and specifies the execution dependency. The node may additionally contain some or even massive numerical learnable parameters (i.e., weights and biases).

An intermediate representation (IR) for deep learning is the data structure or code used internally to represent the computation graph and its constructs. Some IRs such as Google MLIR [29] and TVM Relay [44] are extended to also represent the source code of host programming languages.

After the training finishes, the whole model including both its graph structure and learnable parameters will be serialized in a format to one or more disk files for later use. To be noted, a DL framework may support multiple model formats. For example, TensorFlow provides three options: checkpoint, frozen graph, and saved model.

3 METHODOLOGY

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

\[
M = (V = \{u_i\}_{i=1}^n, E = \{(u_i, u_j)\}, P = \{p_k\}_{k=1}^m)
\]

Each node \(u_i\) is an operator. A directed edge \((u_i, u_j)\) allows the output tensor(s) of node \(u_i\) to flow across as the input(s) of \(u_j\) and specifies that \(u_j\) can start execution only if \(u_i\) has finished. Each \(p_k\) is a hyperparameter such as batch size, learning rate, and dropout rate. We say that \(M' = (V', E', P')\) is a sub-model of \(M\) if \(V' \subseteq V, E' \subseteq E, \) and \(P' = P\).

A tensor is multi-dimensional arrays of numerical values, while its order is the number of its dimensions. Let \(X_{mp} = (X_1, X_2, \ldots, X_m)\)

\(^3\)MMdnn is an acronym for “model management for deep neural networks”: https://github.com/microsoft/MMdnn

\(^4\)Currently, Darknet and PaddlePaddle formats are for source only and ONNX format is for target only.

\(^5\)We require that hyperparameters are invariants under faithful model conversion.
be a tensor tuple, representing tensors associated with an edge as well as an operator’s multiple inputs/outputs. We use tuple instead of set because ordering among tensors can be critical to certain operators such as matrix multiplication. The tensor/edge/operator orderings are actually decided from program statements written by DL developers to construct the computation graph. Then, the mathematical representation of an operator \( u \) with \( m \)-inputs and \( n \)-outputs can be denoted as:

\[
(Z_1, Z_2, \cdots, Z_n) = F_u((X_1, X_2, \cdots, X_m))
\]  

(1)

For a DL model, its input tensor tuple is the concatenation of all input operators’ input tensor tuples according to the operator ordering. An input operator is the one without any predecessors on the computation graph. For a sub-model, its input tensor tuple is the concatenation of all tensor tuples that flow across edges from external operators. Similarly, we define the output tensor tuple and mathematical representation of a DL model/sub-model.

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, a legitimate target model should be produced:

\[ M_1 : (V_1, E_1, P_1) \vdash_{\mathcal{A}} M_2 : (V_2, E_2, P_2) \land (P_2 = P_1) \]

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

\[ Z_{IP} = F_{M_1}(X_{IP}) \vdash_{\mathcal{A}} Z_{IP} = F_{M_2}(X_{IP}) \]

where \( Z_{IP} \) and \( X_{IP} \) are the output and input tensor tuples, respectively.

MMdnn adopts a bottom-up approach to craft a model conversion algorithm \( \mathcal{A}^{\text{MMdnn}} \): for each source operator \( u_i \) (e.g., Conv2D of TensorFlow), an equivalent operator \( u_j \) (still Conv2D) is generated in the target model; if \( u_j \) has an edge pointing to \( u_i \), add an equivalent edge from \( v_j \) to \( v_i \). The faithfulness clearly holds. However, it is not always possible or efficient to perform such exact node-to-node translation thus there could exist transformation between subgraphs. In general, \( \mathcal{A}^{\text{MMdnn}} \) works as follows:

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

\[
(V_{1,i} \cup V_{1,j}) = \emptyset \land (\bigcup \{V_{1,i} \vdash V_1 \})
\]

For each \( u \in V_1 \), if there exists an equivalent in the target DL framework, \( u \) must belong to some single-element subset \( V_{1,i} \).

2. For each sub-model \( M_{1,i} = (V_{1,i}, E_{1,i}) \) \( \vdash_{\mathcal{A}} M_{2,i} = (V_{2,i}, E_{2,i}, P_{1,i}) \) such that:

(a) \( E_{2,i} \subseteq V_{2,i} \times V_{2,i} \), which is a set of internal edges.
(b) If \( V_{2,i} \) contains at least two operators, it must have only one input operator and one output operator. We can ensure this since it is always possible to use extra no-op operators just for relaying tensors.
(c) \( M_{1,i} \) and \( M_{2,i} \) are semantically equivalent:

\[ Z_{IP} = F_{M_{1,i}}(X_{IP}) \vdash_{\mathcal{A}^{\text{MMdnn}}} Z_{IP} = F_{M_{2,i}}(X_{IP}) \]

(3) Suppose that \( v_i \) is the output operator of \( V_{1,i} \), and \( v_j \) is the input operator of \( V_{2,i} \). \( \mathcal{A}^{\text{MMdnn}} \) adds a directed edge \((v_i, v_j)\) to the edge set \( E' \) if \( V_{1,i} \) has edge(s) to \( V_{1,i} \). The tensor tuple associated with \((v_i, v_j)\) is the concatenation of all tensor tuples that flow across such source edge(s).

Finally, \( \mathcal{A}^{\text{MMdnn}} \) outputs

\[ M_2 = \left( \bigcup V_{2,i}, \left( \bigcup E_{2,i} \right), P_i \right) \]

as the converted target model.

The above item 2c for a sub-model with two or more operators is the key to a faithful model conversion. MMdnn achieves this by the *operator selection* technique that many known subgraph patterns are prebuilt and matched. Details will be presented in Section 4.4. By induction, it is not hard to prove that the target model \( M_2 \) is semantically equivalent to \( M_1 \). Hence, \( \mathcal{A}^{\text{MMdnn}} \) satisfies the conditions of Definition 3.1.

4 DESIGN AND IMPLEMENTATION

4.1 Overview

Figure 1 illustrates the extensible conversion architecture of MMdnn. As mentioned previously, 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 input source model from disk files and reconstructs it to a computation graph depicted with the source framework IR. We implement a front-end parser for each supported format using the framework’s built-in model deserialization APIs. For instance, the PyTorch front-end parser calls the torch.load() function 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. For PyTorch models, a state_dict Python dictionary object is used to store the learnable parameters of each layer. Figure 2 shows a simplified PyTorch front-end parser. If possible, the parser invokes framework functionalities (e.g., TensorFlow strip_unused_lib and graph_transforms packages) to compact the source computation graph by removing all unnecessary graph nodes and folding constants for instance.

In the second phase, an intermediate model generator traverses the source computation graph in certain topological (linear) ordering and transforms it into an intermediate with the simple unified IR of MMdnn (to be described in Section 4.3). Briefly, the intermediate model generator performs the node-to-node translation. We employ the *operator selection* (Section 4.4) technique similar to compiler’s instruction selection [6], whose name comes from the fact that a computational graph node is completely decided by its invoked operator. Most often, the source operator has a semantic equivalent in MMdnn IR. We use *macro expansion* to emit a new node with our corresponding operator in the intermediate computation graph. Properties of the source node such as inputs, outputs, and attributes are translated to comply with MMdnn syntax. Learnable parameters are transformed into a NumPy [35] tensor and stored in a global dictionary object for centralized management. However, there are two more complex cases. First, a source subgraph is folded into a single intermediate node, which we call *operator fusion*. For instance, we need to recover fully connected structures in a TensorFlow model. DAG covering is used to discover predefined subgraph patterns. Second, a source node is unfolded into an intermediate
class PyTorchParser(Parser):
    def __init__(self, model_file_name, input_shape):
        super(PyTorchParser, self).__init__()
        # Load the model saved with torch.save().
        model = torch.load(model_file_name)
        self.weight_loaded = True
        self.pytorch_graph = PyTorchGraph(model)
        self.input_shape = tuple([1] + input_shape)
        # Construct the intermediate computation graph.
        self.pytorch_graph.build(self.input_shape)
        self.state_dict = self.pytorch_graph.state_dict
        self.shape_dict = self.pytorch_graph.shape_dict

def mnist_cnn():
    conv2d_inp = mx.sym.var('conv2d_inp')
    conv2d = mx.sym.Convolution(conv2d_inp,
                                kernel=(3,3), stride=(1,1), dilate=(1,1),
                                pad=(0,0), num_filter=32, num_group=1,
                                no_bias=False, layout='NCHW', name='conv2d')
    conv2d_act = mx.sym.Activation(data=conv2d,
                                act_type='relu', name='conv2d_act')
    maxpool2d = mx.sym.Pooling(conv2d_act,
                                global_pool=False, kernel=(2,2), pad=(0,0),
                                stride=(2,2), pool_type='max',
                                name='maxpool2d')
    dropout = mx.sym.Dropout(data=maxpool2d, p=0.25,
                              name='dropout')
    dense = mx.sym.FullyConnected(dropout,
                                  num_hidden=10, no_bias=False, name='dense')
    dense_act = mx.sym.SoftmaxOutput(dense, name='softmax')
    model = mx.mod.Module(dense_act,
                           context=mx.cpu(), data_names=['conv2d_inp'])
    return model

def set_params(model, params_file_path):
    arg_params = get_arg(params_file_path)
    aux_params = get_aux(params_file_path)
    model.bind(False, data_shapes=[('conv2d_inp', (1,1,28,28))])
    model.set_params(arg_params, aux_params, True)
    return model

if __name__ == '__main__':
    model = mnist_cnn()
    model = set_params(model, saved_params_file_path)
    model.save_checkpoint('mnist_cnn', epoch_num)
(2) **Inconsistent tensor layouts.** Some operators of the target framework may not support the original input tensor layout (NCHW vs. NHWC [24]). Therefore, the target model generator must carefully transpose input/output tensors at the proper places, modify those operators’ learnable parameters and attributes, or even reimplement them to ensure the faithfulness. Details will be discussed in Section 4.5.

(3) **Unsupported padding.** In convolutional layers, developers need some asymmetric padding if the filter size is even. However, a few frameworks like Caffe only support symmetric padding which may cause conversion failures. MMdnn will first pad a tensor with the maximum paddings and then crop it to the correct shape.

(4) **Incompatible argument type.** Some frameworks can use objects of the host programming language instead of tensors as the operator arguments. For example, the `begin` argument of MXNet `slice` operator is a Python tuple. When outputting the model, only the final value of `begin` will be written while its full computation history is lost. Therefore, this kind of hybrid programming could result in conversion failure if the source framework (e.g., TensorFlow) defines the same argument as a tensor. The reason is that maybe we cannot decide the value of such an argument since its full computation history is possibly a subgraph of the source model. MMdnn currently adopts a simple solution: only if the tensor is a constant, we query its value and conduct the tensor-to-value translation. For an arbitrary tensor argument, users need to fill in the most likely value. In the future, we plan to apply the symbolic execution [27] or dynamic profiling techniques to obtain the candidate. However, the complete solution relies on whether the target framework has a built-in tensor-to-value capability so that we could directly convert the subgraph representing a full computation history of the argument.

### 4.3 Intermediate Representation

The intermediate representation of MMdnn plays a critical role to represent models from various DL frameworks as unified general-purpose computation graphs. We discard ad-hoc direct conversion between every two frameworks in order to make MMdnn more extensible and reduce the difficulty of implementation and debugging. The goal of MMdnn is faithful model conversion instead of extensible and reduce the difficulty of implementation and debugging. Therefore, the target model generator must carefully transpose input/output tensors at the proper places, modify those operators’ learnable parameters and attributes, or even reimplement them to ensure the faithfulness. Details will be discussed in Section 4.5.

MMdnn refers to existing DL frameworks like ONNX. Protocol Buffers [50] are used to describe the graph schema, which includes the following main constructs:

1. **GraphDef.** This is the top-level construct representing a complete computation graph. Edges are not explicitly defined on the graph since node inputs encode them.

2. **NodeDef.** It represents a graph node with the fields of name, operator type, a list of zero or more named inputs of type `string`, and a map of zero or more named attributes. Each input has the `dep_node: dep_output` format which means the input is actually the output tensor with the name “dep_output” of a predecessor node named “dep_node”. Since node outputs are determined by the corresponding operator, graph edges are therefore implicitly constructed so we take them away from the node structure.

3. **AttrValue.** Attributes represent the operator arguments other than the input tensors. An attribute value is a runtime constant specified by users or inferred during the computation graph construction. Attribute names are not stored together since they appear in the node and operator structures.

4. **op.** This represents an operator, the mathematical operation invoked by a node. It consists of a name, a list of input and output tensor arguments, 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. The `LiteralTensor` represents a serialized tensor value, consisting of a tensor shape, an element data type, and a flattened array of elements.

Initially, the operator set consists of essential operators available in various frameworks, such as basic mathematical operators of addition, exponentiation and division, and layer operators of convolution and batch normalization [25]. To be noted, it is possible that DL frameworks have different syntax on the same operator. Let us take local response normalization (LRN) [28] 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 \( \left\lfloor \frac{2n+1}{2} \right\rfloor \) (rounding down). In CNTK (`BlockApiSetup.lrn`), `n` is used and equals \( \lceil \frac{2n+1}{2} \rceil \) (rounding up). MMdnn unifies the operator syntax: naming and ordering of inputs, outputs, 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. Then, when converting the Tensorflow `LRN` to ours, MMdnn must store \((\text{depth}\_\text{radius}\times 2+1)\) to the size attribute.

The operator set gradually grows on demand. When a new framework operator is met, we usually formalize its syntax and add it to the operator set directly, even if it is exclusive but semantically equivalent to a composition of defined operators. To achieve an optimal converted model, MMdnn tends to avoid decomposing a source operator in the phase of intermediate model generation since certain target frameworks may have 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 its two input tensors should be transposed and conjugated in advance. 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. Currently, MMdnn has defined 80+ frequently-used operators and handled 300+ framework operators in total.

4.4 Operator Selection

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

Macro expansion works by checking the node’s operator, locating a matched expander function, and running the code. Most often, a single output node with the equivalent operator is emitted. Figure 5 and Figure 6 illustrate two expander functions that translate a source TensorFlow Exp node to the intermediate node with MMdnn IR and then to a piece of target CNTK Exp node construction code. To be noted, the expanded output could be a subgraph under certain cases. One example is that the target model generator of CNTK will expand a log1p node to an Add followed by a log.

The input computation graph sometimes could be rather complicated and huge due to the alteration by DL frameworks. For example, TensorFlow’s FullyConnected and BatchNormalization have been optimized to a subgraph of low-level computation operators in the model file. Translating one node by another using macro expansion could ensure the correctness, but it may not achieve an optimal model structure if target frameworks have the same high-level DL constructs. A possible method is DAG covering, which tries to recover the original user-specified graph structures by searching for predefined subgraph patterns on the input computation graph.

def Exp_from_tf(self, source):
    ir = self.IR_Graph.add()
    ir.inputs = source.inputs
    ir.attrs = source.attrs
    ir.name = source.name
    ir.type = source.type

    return code

Figure 5: Macro expansion from a TensorFlow Exp (exponential operator) node to an equivalent intermediate node.

def Exp_to_cntk(self, ir):
    parent = self.parent(ir, pos=0)
    code = "{} = cntk.{}
    .format(ir.name, parent.name, ir.name)

    return code

Figure 6: Macro expansion from the intermediate Exp node in Figure 5 to node construction code of its equivalent in CNTK.

def FullyConnected_from_tf(self, source):
    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 = self.child(source, 0)

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

Figure 7: DAG covering from a subgraph containing TensorFlow MatMul and Variable nodes to an intermediate Fully-Connected node.

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 Fully-Connected node in Figure 7 to node construction code of its equivalent in Caffe.

One pattern comes from the above example in which a high-level operator is broken down to a few low-level ones. For instance, the TensorFlow FullyConnected operator is implemented by a MatMul node, MatMul’s grandfather Variable node, and surrounding others. We locate certain representative nodes (e.g., MatMul) and look around their predecessors and successors (e.g., Variable). Figure 7 and Figure 8 show how to recover an actual TensorFlow FullyConnected node and convert it to the equivalent in Caffe. Patterns for RNN (recurrent neural network) are usually more sophisticated because DL frameworks could unroll an RNN layer (e.g., LSTM [20] and GRU [12]) to a cell sequence whose length is not explicitly encoded in the source model. Furthermore, TensorFlow unfolds each RNN cell to a subgraph of low-level operators too. After having identified all the RNN cells by matching cell patterns if needed, MMdnn infers the sequence length from the shape information of
RNN weights and finally recovers the original RNN layer. Nevertheless, when handling a TensorFlow RNN structure, we sometimes noticed unexpected degradation of the learning performance no matter a high-level RNN layer or a sequence of RNN cells were generated in the target model. We guess that those 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 above, tensors are multi-dimensional arrays. Tensor layout refers to the dimensional ordering, which is critical in DL computation since these dimensions usually have specific semantics. For example, 4D tensors representing images have four dimensions: \( N, C, H, \) and \( W \), which denote 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 NHWC (channels first) and NCHW (channels last) are widely used by different computing devices [24]. However, not all DL frameworks always support both. For instance, on GPU devices, TensorFlow accepts both while PyTorch supports NCHW only.

Let us take an example to illustrate the possible conversion issue caused by discrepant assumptions on the tensor layout. Suppose that we are converting a TensorFlow model trained with the NHWC input data to PyTorch format. We require that the initial input of the target model does not change its tensor layout and will discuss this condition later. If the source model contains operators (graph nodes) like Conv2D, MaxPooling, and BatchNormalization, direct conversion can bring a totally wrong result. The reason lies in that Conv2D etc. operators in PyTorch presume the NCHW layout while their inputs’ actual layout should be NHWC, which we call inconsistent tensor layouts. It is inadequate to simply insert an NHWC-to-NCHW tensor transpose just before and an NCHW-to-NHWC immediately after each of those operators. 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. Actually, the issue caused by inconsistent tensor layouts is rooted in the fact that tensor transpose and some operators are not commutative. In the following, we give the formal notations [39] and definition.

Assume set \( S = \{1, 2, \ldots, n\} \), \( \sigma \) a permutation of \( S \) and \( \sigma^{-1} \) the inverse of \( \sigma \). \( \sigma \) is then a one-to-one function from \( S \) onto \( S \), and is denoted as follows, with \( \sigma(i) = k_i \):

\[
\sigma = \begin{pmatrix}
1 & 2 & 3 & \cdots & n \\
1 & 2 & 3 & \cdots & k_n
\end{pmatrix}
\]

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

\[
\sigma_{\text{NCHW}} = \begin{pmatrix}
1 & 2 & 3 & 4 \\
1 & 4 & 2 & 3
\end{pmatrix}
\]

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

A tensor \( Y \) is called the transpose of \( X \) associated with \( \sigma \) and denoted by \( X^T_{\sigma} \), if

\[
Y(i_{\sigma(1)}, i_{\sigma(2)}, \ldots, i_{\sigma(n)}) = X(i_1, i_2, \ldots, i_n)
\]

Let \( X_{\sigma} = (X_1, X_2, \ldots, X_m) \) be a tensor tuple which represents an operator’s multiple inputs or outputs. \( \sigma_{\text{Tp}} = (\sigma_1, \sigma_2, \ldots, \sigma_m) \) is a permutation tuple with each \( \sigma_i \) being a permutation of \( X_i \)’s \( S \) set. \( \sigma_{\text{Tp}} \) is called a tensor layout of \( X_{\text{Tp}} \). We then denote the transpose of \( X_{\text{Tp}} \) as follows:

\[
X_{\text{Tp}}^{T_{\sigma_{\text{Tp}}}} = (X_1^{T_{\sigma_1}}, X_2^{T_{\sigma_2}}, \ldots, X_m^{T_{\sigma_m}})
\]

Now we define the commutativity of tensor transpose and operators.

**Definition 4.1.** Assume that \( u \) is an operator, and

\[
X_{\text{Tp}} = (X_1, X_2, \ldots, X_m), \quad Z_{\text{Tp}} = (Z_1, Z_2, \ldots, Z_n)
\]

are \( u \)’s input/output tensor tuples satisfying Equation 1.

\[
\sigma_{\text{Tp}} = (\sigma_1, \sigma_2, \ldots, \sigma_m), \quad \sigma_{\text{Tp}}' = (\sigma_1', \sigma_2', \ldots, \sigma_n')
\]

are tensor layouts of \( X_{\text{Tp}} \) and \( Z_{\text{Tp}} \), respectively. The operator \( u \) is called to be transpose-commutative associated with \( \sigma_{\text{Tp}} \) if

\[
Z_{\text{Tp}}^{T_{\sigma_{\text{Tp}}'}} = f_u(X_{\text{Tp}}^{T_{\sigma_{\text{Tp}}}})
\]

In case \( \sigma_{\text{Tp}} \) is arbitrary and there always exists the corresponding \( \sigma_{\text{Tp}}' \), \( u \) is transpose-commutative.

It is not hard to reason 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 inconsistent tensor layouts, inserting two corresponding tensor transposes just before and immediately after it can solve the issue, given the previously mentioned \( \sigma_{\text{Tp}} \) and \( \sigma_{\text{Tp}}' \). The core task is to pre-identify those operators in the target DL model which confront inconsistent tensor layouts yet are non-transpose-commutative under such circumstances. We then make them transpose-commutative associated with the new target input tensor layout by modification on the learnable parameters/attributes or re-implementation. For the TensorFlow-to-PyTorch example, we transpose Conv2D’s filters from [filter_height, filter_width, in_channels, out_channels] to [out_channels, in_channels, filter_height, filter_width]. To be noted, DL frameworks may implement self-adaptive operators to work with multiple tensor layouts. Users only need to explicitly specify the correct input tensor layout. For instance, the TensorFlow Conv2D operator has an optional string argument data_format which defaults to NHWC. Thus, we think of Conv2D as being transpose-commutative associated with \( \sigma_{\text{NCHW}} \) in TensorFlow, assuming that \( \sigma_{\text{NCHW}} \) is the identity permutation. Algorithm 1 proposes a general approach to handle inconsistent tensor layouts, with the optimization of eliminating consecutive reciprocal tensor transposes.

At the beginning of this subsection, we assume that the tensor layout of initial input will not change in the target model. Hence, enough necessary tensor transposes will be inserted which may significantly reduce the runtime performance. A more attractive idea is only transposing the initial input and final output to avoid those inserted tensor transposes. However, such a method requires that we must completely determine the actual tensor layouts of each
Algorithm 1: Handle inconsistent tensor layouts in the target model generation.

Input: The intermediate model inter_model
Output: The target model target_model with possible operator amendment and inserted tensor transposes

1. inter_topo ← GetTopologicalOrder(inter_model);
2. foreach op ∈ inter_topo do
3.     old_in_layout ← GetInputLayout(op); // A tuple
4.         (l₁,...,lᵣ) whose lᵢ is the layout of op’s i-th input.
5.     allowed_input_layouts ← GetTargetAllowedInputLayouts(op);
6.     if old_in_layout ∈ allowed_input_layouts then
7.         // Direct conversion for consistent input layouts.
8.         EmitNode(target_model, op);
9.         continue;
10.    end
11.    new_in_layout ← SelectTargetInputLayout(allowed_input_layouts);
12.    parents ← GetParents(inter_model, op);
13.    for i ← 1 to parents.GetLength() do
14.        parent ← parents[i];
15.        parent_transpose ← GetTransposeAfter(target_model, parent); // Emitted by parent at Line 36
16.        if parent_transpose ≠ NULL then
17.            lᵢ ← old_in_layout [i];
18.            lᵢ ← new_in_layout [i];
19.            if IsInverse(parent_transpose, lᵢ, lᵢ) then
20.                // Optimize parent transpose away.
21.                RemoveNode(target_model, parent_transpose);
22.            else
23.                EmitTransposeNode(target_model, lᵢ, lᵢ);
24.            end
25.        end
26.        EmitNode(target_model, new_op);
27.    end
28.    // Assume old_in_layout is the identity layout.
29.    if IsCommutativeInTarget(op, old_in_layout, new_in_layout) then
30.        new_op ← op;
31.    else
32.        new_op ← MakeCommutativeInTarget(op);
33.    end
34.    EmitNode(target_model, new_op);
35.    // Emit possible tensor transposes for new_op’s outputs.
36.    old_out_layout ← GetOutputLayout(op, old_in_layout);
37.    new_out_layout ← GetOutputLayout(new_op, new_in_layout);
38.    for i ← 1 to old_out_layout.GetLength() do
39.        lᵢ ← old_out_layout [i];
40.        lᵢ ← new_out_layout [i];
41.        if lᵢ ≠ lᵢ then // Emit an lᵢ-to-lᵢ tensor transpose.
42.            EmitTransposeNode(target_model, lᵢ, lᵢ);
43.        end
44.    end

Yu Liu, Cheng Chen, Ru Zhang, Tingting Qin, Xiang Ji, Haoxiang Lin, and Mao Yang

5 EVALUATION

MMDan is implemented in Python with over 29,000 lines of code. We evaluate a converted model by comparing its learning performance with that of the source model. The ResNet-152 models of TensorFlow [49], PyTorch [41], CNTK [13], MXNet [33], and Caffe [7] from the official galleries are used as our experimental objects. Except that the TensorFlow source model is trained with the NHWC tensor layout, others are with NCHW. For the targets, TensorFlow models use NHWC while others keep NCHW. Target frameworks additionally include Keras and ONNX. We fix the version number for each framework (Table 1), and MMDan invokes that specific version to load and export models. Our experiments are conducted on an Ubuntu 16.04 workstation with 16 Intel Xeon E5-2665 CPUs and 128 GB main memory.

The first 10,000 of COCO [30] 2017 Test images⁹ in RGB/BGR formats are selected as the input. We preprocess them 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.

Test images are divided into 20 subsets and we evaluate 500 images per round.

Suppose that there are m input image tensors 𝑋ᵢ ∈ ℜ[^1×m]. Let yᵢ and zᵢ be two output vectors in the Euclidean space ℜ[^n], computed by the source and target models on 𝑋ᵢ, respectively. Let yᵢj and zᵢj be their j-th components. Then yᵢj, zᵢj ≥ 0 if j ∈ [1, n]. For each input image, we calculate three measurements: MRE (max relative error [31]), SNR (signal-to-noise ratio [16]), and PSNR (peak signal-to-noise ratio [15]):

\[ MRE_{ij} = \begin{cases} 0 & \text{if } y_{ij} = z_{ij} = 0, \\ \frac{\max |y_{ij} - z_{ij}|}{\max (y_{ij}, z_{ij})} & \text{otherwise.} \end{cases} \]

\[ MRE_{i} = \max_{1 \leq j \leq n} MRE_{ij} \]

\[ SNR_{i} = 10 \times \frac{\|z_{i}\|^2}{n} - 10 \times \log \frac{\|y_{i} - z_{i}\|^2}{n} \]

\[ PSNR_{i} = 10 \times \log(\max_{1 \leq j \leq n} \|z_{ij}\|^2) - 10 \times \log \frac{\|y_{ij} - z_{ij}\|^2}{n} \]

Then, we measure the experiment:

\[ MRE = \max_{1 \leq i \leq m} MRE_{i} \]

\[ SNR = \min_{1 \leq i \leq n} SNR_{i} \]

\[ PSNR = \min_{1 \leq i \leq n} PSNR_{i} \]

⁹There are 41,000 images whose total compressed size is 6 GB.
We claim a faithful model conversion if \( MRE < 0.15 \), \( SNR > 12 \), and \( PSNR > 30 \) are all satisfied since the converted model has a comparable learning performance. Those thresholds are set based on our experience.

Table 1 shows the evaluation results, in which all of them meet the above conditions for faithfulness. Each cell has three lines of values, reporting \( MRE \), \( SNR \), and \( PSNR \) respectively on the full 10,000 images.

<table>
<thead>
<tr>
<th>Source</th>
<th>Target</th>
<th>TensorFlow (Version: 1.13.1)</th>
<th>PyTorch (0.4.0)</th>
<th>CNTK (2.6)</th>
<th>MXNet (1.2.0)</th>
<th>Caffe (1.0)</th>
<th>Keras (2.2.4)</th>
<th>ONNX (1.4.1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TensorFlow</td>
<td>Zero Center</td>
<td>( 1.60 \times 10^{-5} )</td>
<td>( 3.80 \times 10^{-5} )</td>
<td>( 5.61 \times 10^{-6} )</td>
<td>( 4.11 \times 10^{-5} )</td>
<td>( 3.64 \times 10^{-5} )</td>
<td>0</td>
<td>1.24 \times 10^{-5}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( SNR: 72.51 )</td>
<td>72.51</td>
<td>79.24</td>
<td>72.51</td>
<td>72.51</td>
<td>72.51</td>
<td>71.81</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( PSNR: 85.56 )</td>
<td>85.56</td>
<td>95.68</td>
<td>85.56</td>
<td>85.56</td>
<td>85.56</td>
<td>90.41</td>
</tr>
<tr>
<td>PyTorch</td>
<td>Standard</td>
<td>( 1.97 \times 10^{-5} )</td>
<td>( 5.19 \times 10^{-6} )</td>
<td>( 1.79 \times 10^{-6} )</td>
<td>( 8.82 \times 10^{-6} )</td>
<td>( 2.15 \times 10^{-5} )</td>
<td>( 1.69 \times 10^{-4} )</td>
<td>( 2.80 \times 10^{-6} )</td>
</tr>
<tr>
<td></td>
<td>Transposition</td>
<td>72.88</td>
<td>72.88</td>
<td>72.88</td>
<td>72.88</td>
<td>72.88</td>
<td>69.25</td>
<td>75.78</td>
</tr>
<tr>
<td></td>
<td></td>
<td>84.66</td>
<td>84.66</td>
<td>84.66</td>
<td>84.66</td>
<td>84.66</td>
<td>80.32</td>
<td>89.49</td>
</tr>
<tr>
<td>CNTK</td>
<td>Identity</td>
<td>( 3.58 \times 10^{-6} )</td>
<td>( 1.73 \times 10^{-6} )</td>
<td>( 1.50 \times 10^{-5} )</td>
<td>( 1.81 \times 10^{-5} )</td>
<td>( 1.58 \times 10^{-5} )</td>
<td>3.82 \times 10^{-6}</td>
<td>7.15 \times 10^{-6}</td>
</tr>
<tr>
<td></td>
<td>Transposition</td>
<td>78.48</td>
<td>71.67</td>
<td>71.67</td>
<td>71.67</td>
<td>71.67</td>
<td>80.15</td>
<td>77.25</td>
</tr>
<tr>
<td></td>
<td></td>
<td>92.42</td>
<td>85.75</td>
<td>85.75</td>
<td>85.75</td>
<td>85.75</td>
<td>92.26</td>
<td>90.85</td>
</tr>
<tr>
<td>MXNet</td>
<td>Identity</td>
<td>( 1.38 \times 10^{-5} )</td>
<td>( 7.21 \times 10^{-6} )</td>
<td>( 5.04 \times 10^{-6} )</td>
<td>0</td>
<td>( 8.61 \times 10^{-6} )</td>
<td>8.11 \times 10^{-6}</td>
<td>1.07 \times 10^{-6}</td>
</tr>
<tr>
<td></td>
<td>Transposition</td>
<td>18.78</td>
<td>18.78</td>
<td>18.78</td>
<td>18.78</td>
<td>18.78</td>
<td>18.78</td>
<td>33.30</td>
</tr>
<tr>
<td></td>
<td></td>
<td>41.37</td>
<td>41.37</td>
<td>41.37</td>
<td>41.37</td>
<td>41.37</td>
<td>41.37</td>
<td>62.58</td>
</tr>
<tr>
<td>Caffe</td>
<td>Zero Center</td>
<td>( 9.00 \times 10^{-6} )</td>
<td>( 1.06 \times 10^{-5} )</td>
<td>( 5.90 \times 10^{-6} )</td>
<td>( 9.12 \times 10^{-6} )</td>
<td>0</td>
<td>1.30 \times 10^{-5}</td>
<td>8.40 \times 10^{-6}</td>
</tr>
<tr>
<td></td>
<td>Transposition</td>
<td>22.06</td>
<td>22.06</td>
<td>24.16</td>
<td>22.06</td>
<td>22.06</td>
<td>22.06</td>
<td>21.26</td>
</tr>
<tr>
<td></td>
<td></td>
<td>43.09</td>
<td>43.09</td>
<td>47.96</td>
<td>43.09</td>
<td>43.09</td>
<td>43.09</td>
<td>44.20</td>
</tr>
</tbody>
</table>

Table 1: Evaluation results on the ResNet-152 models.

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 [23], built upon the Julia programming language [5], provides a new and elegant machine learning stack for Julia developers. TensorFlow XLA [1] and DLVM [51] are domain-specific compilers, optimizing on high-level computation graphs. They aim at achieving the best training/serving performance. Although MMdnn adopts compiler principles (e.g., IR and processing phases), it only focuses on faithful model conversion.

Common model formats. ONNX [36] and NNEF [34] are open neural network exchange formats, with a similar motivation for the framework interoperability. However, their IRs include only essential operators which are much 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 a model uses one out-of-range operator, it cannot be converted or directly exported to such two formats. And the extensibility of ONNX and NNEF must align with the supporting frameworks and hardware vendors. 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. It is more likely that the target framework already has an equivalent for the source operator. And MMdnn can further extend its IR via operator decomposition and porting. Hence, MMdnn has the potential to quickly support more DL operators, models, and frameworks for model conversion.

Model converters. There are a number of model converters such as caffe-tensorflow [17], ONNXMLTools [37], WinMLTools [32], Core ML Community Tools [3], tf2onnx [38], NNEF-Tools [18], 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 support more DL operators such as 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 operators are increasing so rapidly, it is inefficient to manually understand their semantics, discover equivalents, and port them to another framework. Some program analysis techniques may be developed to facilitate the process. However, we think that using certain standard domain-specific language (DSL) for operator semantic description can be very useful to not only model conversion but also training/serving performance. As DL frameworks are evolving fast, we will continuously keep up with their latest versions. Another work is to improve the runtime performance for handling inconsistent tensor

---

10 CNTK/Caffe2/MXNet/PyTorch can export models to ONNX format.
layout cases in the number of inserted tensor transposes is high. We also hear from developers that they want to port existing DL programs to another framework faithfully for later training, which requires a new DL transpiler (source-to-source compiler).

ACKNOWLEDGEMENTS

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 Jiachao Yao, Yuhao Zhou, Tong Zhan, and Qianwen Wang. We thank all the contributors for their submitted questions, suggestions, documents, features, bug fixes, etc.

REFERENCES


Enhancing the Interoperability between Deep Learning Frameworks by Model Conversion


