SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

December 2020, Vol. 63 222103:1–222103:21
https://doi.org/10.1007/s11432-020-3097-4

Jittor: a novel deep learning framework with meta-operators and unified graph execution

Shi-Min HU[1,2*], Dun LIANG[1], Guo-Ye YANG[1],
Guo-Wei YANG[1] & Wen-Yang ZHOU[1]

[1]*Department of Computer Science and Technology, Tsinghua University, Beijing* 100084*, China;*
[2]*Beijing National Research Center for Information Science and Technology, Beijing* 100084*, China*

Received 25 August 2020/Accepted 23 September 2020/Published online 13 November 2020

Abstract This paper introduces Jittor, a fully just-in-time (JIT) compiled deep learning framework. With JIT compilation, we can achieve higher performance while making systems highly customizable. Jittor provides classes of Numpy-like operators, which we call meta-operators. A deep learning model built upon these meta-operators is compiled into high-performance CPU or GPU code in real-time. To manage meta-operators, Jittor uses a highly optimized way of executing computation graphs, which we call unified graph execution. This approach is as easy to use as dynamic graph execution yet has the efficiency of static graph execution. It also provides other improvements, including operator fusion, cross iteration fusion, and unified memory.

Keywords deep learning framework, meta-operator, unified graph execution, JIT compilation, generative adversarial network

Citation Hu S-M, Liang D, Yang G-Y, et al. Jittor: a novel deep learning framework with meta-operators and unified graph execution. Sci China Inf Sci, 2020, 63(12): 222103, https://doi.org/10.1007/s11432-020-3097-4

1 Introduction

In recent years, deep learning has developed rapidly. With big data science, deep learning has become a new paradigm for scientific research and engineering applications. As deep learning algorithms are typically complicated to implement, various deep learning libraries and frameworks have been developed, to provide researchers and developers with convenient ways to rapidly develop deep learning systems. The first of these was Torch, a modular machine learning software library [1]. In 2008, the MILA Laboratory of Montreal University, led by Yoshua Bengio, announced the Theano deep learning framework [2]. Its conceptual organisation has been adopted by subsequent deep learning frameworks. Python is used as a front-end language, while C, CUDA and other languages are used as back-end languages for acceleration, and computational graphs (also called dataflow graphs) provide a bridge between them. Later, several other frameworks were proposed, including Caffe [3], TensorFlow [4], and PyTorch [5]. Theano ceased to be maintained in 2017, and Caffe was merged with PyTorch in 2018. Thus, TensorFlow and PyTorch are the two main current frameworks used for deep learning; about 70% of CVPR 2020 deep learning papers used PyTorch.

Over time, these frameworks have evolved to provide many new features, which have become very popular and been widely used. These include hardware acceleration, automatic differentiation, dynamic

* Corresponding author (email: shimin@tsinghua.edu.cn)

binding, and eager execution. These features make the framework easier to use, or give it better performance. In particular, just-in-time (JIT) compilation is a new direction for deep learning frameworks, which is gradually becoming popular. TensorFlow provides an experimental back-end, XLA, which utlizes JIT compilation for acceleration. PyTorch also comes with JIT suites for industrial deployment and execution without the Python environment. There are also many compilers and libraries for JIT compilation, such as Intel's nGraph [6], Nvidia's TensorRT, JAX [7], and TVM [8], which can help users to easily develop fast kernels. JIT compilation technology not only improves the performance of deep learning frameworks, but can also make them easier to use. Thus Jittor, the new deep learning framework proposed in this paper, is fully JIT compilation based: all source code of Jittor is compiled at run-time for a better experience.

To support a variety of complicated deep learning tasks, computational graphs are widely used in deep learning frameworks. Each edge of a computational graph represents a dependency, and each node represents an operator; different tasks require different operators. Deep learning frameworks usually provide a wide selection of operators to support various tasks. This makes it easy for users to readily build learning models based on operators. However, from the perspective of the framework developer, it is tedious and difficult to ensure good performance for a large number of operators. From the perspective of the user, operators that are not well-optimized cause performance degradation. We thus propose a solution, via the concept of meta-operators and a fusion mechanism. Meta-operators are general operators, which can be thought of as classes of specialized operators, whose common properties allow the specialized operators to be optimized in a similar way. Operator fusion allows the combination of multiple operators into a single operator, with the advantage that intermediate results do not need to be stored in memory. Jittor has 3 types of meta-operators, which contain a total of 18 specialized operators. Many common deep learning operators, such as convolution and deconvolution, can be fused by use of these meta-operators. We only need to optimize the meta-operators and fusion strategy, to allow all computations built upon them to be uniformly optimized for better performance.

To train a deep learning model, back-propagation algorithms are typically used; the gradient of the training loss is propagated backwards to the parameters of the model. Each forward operator of the model requires a corresponding backward operator to back-propagating the gradient. Our meta-operators possess the useful property of backward closure, which means that backward propagation of a meta-operator is still a meta-operator. Fused operators for deep learning also have backward closure. The backward closure property furthermore allows for the use of higher order derivatives in back-propagation, resulting in faster model convergence.

The computational graph is the basic data structure in a deep learning framework. It can be static or dynamic. In the former case, the graph is first built, then executed, while in the latter case, the graph is built on the fly, as it is executed. TensorFlow 1.0 uses static graph execution, which is efficient and easy to optimize and deploy, but inflexible when changes to the graph are needed during execution or debugging is to be carried out. PyTorch instead utilizes dynamic graph execution, which allows modification during execution. This is more convenient for users, but is inefficient when training, and unsuited to reasoning about deep learning. We propose a new approach to computational graphs execution, which we call unified graph execution. It combines aspects of both static and dynamic graph execution, leading to both flexibility and efficiency. In detail, unified graph execution splits the computational graph into multiple sub-graphs. For each sub-graph, static graph execution is used, while for the global graph, dynamic execution is used.

The rest of this paper is organized as follows. Section 2 considers design principles and the architecture of Jittor. The front-end of Jittor, including meta-operators, their fusion and their back-propagation, is considered in Section 3. The back-end of Jittor, including the JIT compiler and unified graph execution is discussed in Section 4. Section 5 compares Jittor and PyTorch, both for inferencing, and for training a generative adversarial network. Section 6 gives a conclusion and discusses future work.

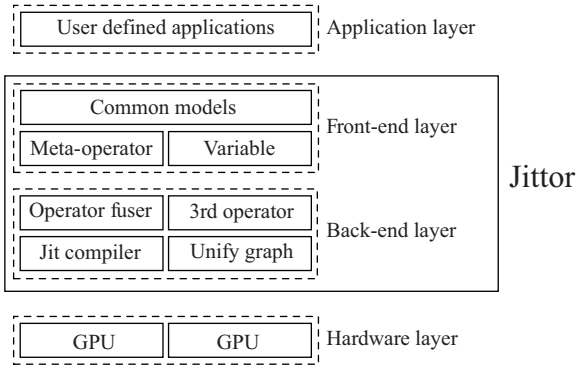

Figure 1 Jittor in use.

2 Design principles and architecture of Jittor

The development of deep learning frameworks revolves around improving human productivity and computational performance. To achieve good performance from modern processors, developers often need to write assembly language, use special instruction sets, or use specialised languages or libraries, such as shaders for GPU programming, CUDA [9], and OpenCL [10]. Although these provide excellent performance, they are difficult to use and debug, and furthermore, programmers need a good understanding of the underlying hardware. Scripting languages such as Python and JavaScript are interpreted, giving immediate feedback, further reducing the difficulty of programming, but this sacrifices performance.

To simultaneously improve productivity and performance, various scientific computing libraries and deep learning frameworks have been developed. A widely used optimization method is static compilation with dynamic binding. This optimization method uses C, C ++, CUDA [9], or other languages to statically compile the operators needed in deep learning, while the user dynamically applies them via scripting languages such as Python and Javascript. Many frameworks adopt this approach, including Numpy [11], Matlab, Theano [2], TensorFlow [4], MXNet [12], and PyTorch [5].

Dynamic binding allows users to take full advantage of the underlying hardware performance when using a scripting language, but it has a problem: all operations are statically compiled, making optimizations such as operator fusion difficult; this important optimization technique combines multiple operators into one operator, so that intermediate results do not need to be stored. Dynamic binding with a scripting language cannot use this optimization. For example, the user may need to calculate $d = ab + c$, where a, b, c are tensors. First, the scripting interpreter executes `tmp = TensorMul(a,b)` and then executes `d = TensorAdd(tmp,c)`. If we could compile the whole expression, rather than applying operators one by one, we could execute `d = TensorMulAndAdd(a,b,c)` directly without the need for temporary storage. This is significant, as on modern processors, memory access is often much slower than calculation. However, we cannot guess what combinations of operators the user may require, and static compilation of all possible combinations is obviously infeasible. To solve this problem, we may use JIT compilation technology to dynamically compile and optimize the operators that the user needs.

Jittor is a completely new design of deep learning framework based on JIT compilation technology. Following the above discussion, Jittor is designed based on the following principles.

• It should be highly customizable yet easy to use. Users should be able to define new operators and models with just a few lines of code.

• It should separate coding from optimization. Users should be able to focus on coding using the front-end interface, while the code is automatically optimized by the back-end. This improves readability of the front-end code, while well-tested, standard optimization code in the back-end ensures robustness.

• Everything should be compiled JIT. This includes the back-end and operations. Users should be able to change the source code at any time.

An overview of an application built with Jittor is provided in Figure 1. It has four layers, the middle two being provided by Jittor.

(1) Application layer. This layer is coded by the user in Python, using interfaces exposed by the front-end layer.

(2) Jittor front-end. This layer is written in Python. All interfaces in this layer are visible to users. It provides the meta-operator interface, allows manipulation of Jittor variables, and implements common models. See Section 3 for more details.

(3) Jittor back-end. This layer provides interface support to the front-end layer, and also manages the underlying hardware resources. It contains many modules, such as the operator fuser, which is used to dynamically fuse operators for improving performance. Third party operators provided by libraries such as CuDNN [13] and MKL can also be incorporated; the user can also define their own operators. It also includes the JIT compiler and the unified computational graph. See Section 4 for more details.

(4) Hardware. The back-end layer communicates with this layer for hardware acceleration. The current accelerator supports CPU and GPU hardware.

Our unified graph execution allows operators to be generated and automatic differentiation to be performed on the fly, as for a dynamic graph. The graph is cut into multiple static sub-graphs for optimization. The system also uniformly manages a variety of resources, such as forward and backward propagation operators, and computation graphs used in multiple iterations, allowing shared optimizations between these resources. For example, this permits optimization across successive iterations. This approach provides increased opportunities for operator fusion. It also uniformly manages CPU and GPU memory, swapping GPU memory into CPU memory when the former is insufficient. It schedules synchronous and asynchronous operations to ensure data consistency while allowing data reading, memory copying, and concurrent computing. Meta-operators called by Python are compiled into high-performance C++ or CUDA code. The JIT compiler is compatible with LLVM [14] and automatically optimizes the underlying source code.

3 The front-end

The front-end represents that part of Jittor which is visible to users. It is fully implemented in Python. It provides an interface to the meta-operators, commonly used layers and models, and allows manipulation of Jittor variables. We first explain the meta-operators in detail in Subsection 3.1. Then, we describe how variables and data are manipulated inside Jittor in Subsection 3.4. Finally, we give an example in Subsection 3.5 to illustrate the overall front-end interface.

3.1 Meta-operators

Meta-operators are a key concept in Jittor. In this subsection, we explain the concept of meta-operators, and how they may be fused and back-propagated. They allow easy development of operators by the user (a customized convolution example shown in Subsection 3.2 requires only 11 lines of code) while their uniformity ensures that the JIT compilation can perform optimization.

A meta-operator is a general operator, which when specialized gives a class of operators with common properties that make them particularly amenable to optimization. Deep learning frameworks usually provide many operators to make it easy for users to build learning models. In fact, many of them do similar things, and can be expressed as specializations of more general higher-level operators. In particular, reindex is a very useful meta-operator which provides an arbitrary one-to-many mapping between its input and output. Various specialized operators such as broadcast, pad, and slice are particular forms of this operator, and belong to the reindex operator class. Another important meta-operator is reindex-reduce, which provides a many-to-one mapping. Sum and product are particular examples of reindex-reduce operators. The third meta-operator class comprises element-wise operators. Each has one or more matrix inputs, which should all have the same shape; the output matrix also has this shape. Results are computed element by element. Matrix addition is an example of a binary element-wise operator.

These meta-operator classes are shown in Figure 2, which also shows how Jittor provides common,

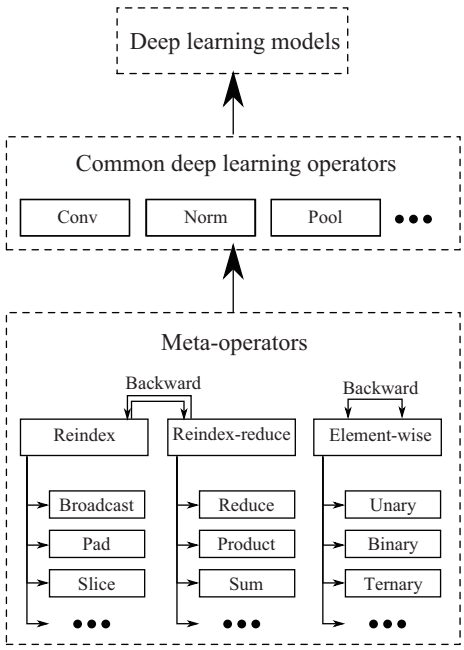

Figure 2 Building models from meta-operators. Operators from the three meta-operator classes, reindex, reindex-reduce, and element-wise, are fused to provide other common deep learning operators, which in turn are used to build the model.

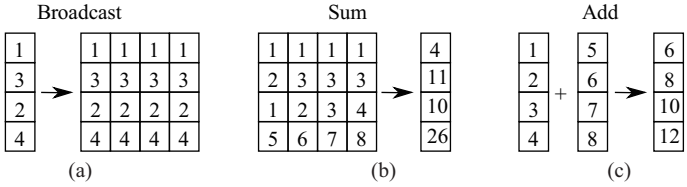

Figure 3 Three examples of different types of meta-operators. (a) The broadcast operator belongs to reindex type meta-operators. Each element of the operator's input vector will be broadcast to its corresponding row of output matrix. (b) The sum operator belongs to reindex-reduce type meta-operators. Each row of the operator's input matrix will be sum into its corresponding element of output vector. (c) The add operator belongs to element-wise operators.

higher-level, deep learning operators (e.g., convolution, normalization, and pooling), by fusing meta-operators.

In detail reindex, reindex-reduce and element-wise operators work as follows:

(1) **Reindex.** The interface of the reindex operator is defined as

$$\text{reindex}(I, S_O, f) \to O,$$

where I represents an N-dimensional input tensor, its input shape is defined as S_I, and its n-th index is defined as $i_n \in 0, \ldots, S_{I_m} - 1$. S_O represents the shape of an M-dimensional output tensor O. Its m-th index is defined as $o_m \in 0, \ldots, S_{O_n} - 1$. Any N-dimensional tensor T has a total Q_T elements given by $Q_T = S_{T_1} S_{T_2} \cdots S_{T_N}$. To locate a certain element $T_{t_1,\ldots,t_N}$, a tuple of indices $(t_1, \ldots, t_N)$ is required. The function $f : (o_1, \ldots, o_M) \to (i_1, \ldots, i_N)$ maps a tuple of output indexes $(o_1, \ldots, o_M)$ to a tuple of input indexes $(i_1, \ldots, i_N)$. The reindex operator returns an output tensor O with given shape S_O. Each element of output tensor $O_{o_1,\ldots,o_M}$ is given by

$$O_{o_1,\ldots,o_M} = I_{i_1,\ldots,i_N}.$$

In short, the reindex operator rearranges the input and stores it in appropriate positions of the output. Index bounds checking is also performed based on the indexing function.

For example, with suitable choice of f, the reindex meta-operator constitutes a broadcast operator. In Figure 3, a 1-D vector is broadcast into a 2-D matrix. Here, the input dimension $N = 1$, the output

dimension $M = 2$, and $f : (o_1, o_2) \to (o_1)$, where i_1 is defined as o_1 in this mapping function. The output matrix is thus determined by $O_{o_1,o_2} = I_{o_1}$; in other words, the operator performs broadcast by row.

(2) **Reindex-reduce.** The interface of the reindex-reduce operator is defined as

$$\text{reindex\_reduce}(I, S_O, f) \to O.$$

Arguments I and S_O have similar meanings to those for reindex. In this case, I is an N-dimensional tensor and O is an M-dimensional tensor. The n-th index of I is defined as $i_n \in X_n$, $X_n = 0, \ldots, S_{I_n} - 1$. The m-th index of O is defined as $o_m \in Y_m$, $Y_m = 0, \ldots, S_{O_m} - 1$. The function $f : (i_1, \ldots, i_N) \to (o_1, \ldots, o_M)$ maps a tuple of input indexes $(i_1, \ldots, i_N)$ to a tuple of output indexes $(o_1, \ldots, o_M)$. The operation of a reindex-reduce operator is given by

$$O_{o_1,\ldots,o_M} = \sum_{(i_1,\ldots,i_N)\in X_1\times\cdots\times X_N \wedge f(i_1,\ldots,i_N)=(o_1,\ldots,o_M)} I_{i_1,\ldots,i_N}.$$

For example, by suitably specifying f, the reindex-reduce operator can sum a 2-D matrix into a 1-D vector as shown in Figure 3. Here, input dimension $N = 2$, output dimension $M = 1$, and $f(i_1, i_2) = i_1$. The output matrix is given by: $O_{o_1} = \sum_{f(i_1,i_2)=o_1} I_{i_1,i_2}$, so $O_{o_1} = \sum I_{o_1,i_2}$. This is the common operator that sums rows of a matrix to give a vector of sums.

(3) **Element-wise operators.** Element-wise operators may be unary (e.g., unary minus), binary (e.g., $+ - \times \div$), or ternary (e.g., if-a-then-b-else-c). All input tensors must have the same shape, which is also the shape of the output tensor. Each element of the output tensor is calculated using corresponding elements in the same position in the input tensor(s).

3.2 Fusion of operators

In this subsection, we now explain fusion of operators from the meta-operator classes, using an example of computing a convolution.

Listing 1 shows how a convolution operator can be implemented in terms of meta-operators. A general reindex meta-operator is used, in addition to specialised broadcast and sum operators.

Listing 1 Python implementation of convolution using three operators: reindex, broadcast, and sum

```
1: def conv(x, p):
2:     N,C,H,W = x.shape
3:     o,i,h,w = p.shape
4:     xx = x.reindex(
5:          shape=(N,o,H,W,i,h,w),
6:          indices=("i0", "i4", "i2-i5", "i3-i6")
7:     )
8:     pp = p.broadcast(xx.shape, dims=(0,2,3))
9:     yy = xx*pp
10:    y = yy.sum(dims=(4,5,6))
11:    return y
```

Line 1 shows that `conv` takes two arguments: `x` is an image tensor, and `p` is a parameter tensor. Lines 2 and 3 unpack information about the shapes of the parameters. The layout of the image tensor `x` is: number of batches (`N`), number of channels (`C`), image height (`H`), and image width (`W`). The layout of the parameter tensor `p` is: number of output channels (`o`), number of input channels (`i`), kernel height (`h`), and kernel width (`w`). Lines 4–7 call the reindex operator with input tensor `x` and output tensor `xx`. The result is:

$$\texttt{xx}(\texttt{i0}, \texttt{i1}, \texttt{i2}, \texttt{i3}, \texttt{i4}, \texttt{i5}, \texttt{i6}) = \texttt{x}(\texttt{i0}, \texttt{i4}, \texttt{i2–i5}, \texttt{i3–i6}). \tag{1}$$

Line 8 broadcasts the parameter tensor `p` to output tensor `pp`, with the same shape as `xx`. The broadcast operator is a specialization of the reindex operator, equivalent to `pp = p.reindex(x.shape, indices=(i1,i4,i5,i6))`. The result of this broadcast operator is

$$\texttt{pp}(\texttt{i0}, \texttt{i1}, \texttt{i2}, \texttt{i3}, \texttt{i4}, \texttt{i5}, \texttt{i6}) = \texttt{p}(\texttt{i1}, \texttt{i4}, \texttt{i5}, \texttt{i6}).$$

Line 9 performs element-wise multiplication with result

$$\texttt{yy(i0,i1,i2,i3,i4,i5,i6)} = \texttt{xx(i0,i1,i2,i3,i4,i5,i6)}\,\texttt{pp(i0,i1,i2,i3,i4,i5,i6)}.$$

Line 10 uses a sum operator, a specialization of reindex-reduce, to compute

$$\texttt{y(i0,i1,i2,i3)} = \sum_{\texttt{i4,i5,i6}} \texttt{yy(i0,i1,i2,i3,i4,i5,i6)}.$$

The above example shows how a convolution can be implemented in terms of 4 calls to meta-operators. Jittor is able to fuse these 4 meta-operators into a single operator, such that intermediate variables `xx`, `pp`, `yy` do not need to actually be calculated. Fusing all 4 meta-operators leads to the final expression:

$$\texttt{y(i0,i1,i2,i3)} = \sum_{\texttt{i4,i5,i6}} \texttt{x(i0,i4,i2–i5,i3–i6)}\,\texttt{p(i1,i4,i5,i6)}.$$

In a similar way, meta-operators can also be used to implement various convolution variants, such as dilation convolution and group convolution.

3.3 Back-propagation of meta-operators

Back-propagation [15] is the foundation of neural network training, where errors in predictions are propagated back to their sources to adjust the parameters of the model. In this subsection, we explain the back-propagation mechanism for meta-operators, and backward closure.

During back-propagation, we need to do the following. Given an operator $f(x) = y$, and an error e_y in the output, we need to calculate the corresponding error e_x in the input. This is done via the corresponding backward operator: $g(x, e_y) = e_x$. To find g, we need to calculate the gradient of the whole model. The chain rule is applied and each operator in the forward dataflow graph is mapped to its gradient operator, where overall we have

$$f(x) = y \underset{\text{backward}}{\Rightarrow} e_x = g(x, e_y) = f'(x)\, e_y. \tag{2}$$

Backward closure is an important property provided by our meta-operators. A set $\boldsymbol{B}$ of operators is said to have backward closure if the backward operator of each operator in $\boldsymbol{B}$ is also in $\boldsymbol{B}$. Backward closure is especially important in training, when back-propagation is performed using a series of backward operators of the model. If backward operators of those used in the model are unavailable, it will not be possible to propagate the gradients back through the model. Normally, the user thus needs to implement corresponding backward operators. With backward closure, however, derivatives, including higher-order derivatives, can be determined automatically, saving manual effort.

Backpropagation of the reindex and reindex-reduce operators is performed as follows. Let x be the input and S_x be the shape of x, and y be the output and S_y be the shape of y. Let e_y be the output error to be back-propagated, and f be the index mapping function. From Eq. (2), and noting that the reindex-reduce operator is the backward operator of the reindex operator, we have

$$y = \text{reindex}(x, S_y, f) \underset{\text{backward}}{\Rightarrow} e_x = \text{reindex\_reduce}(e_y, S_x, f). \tag{3}$$

Similarly, the reindex operator is the backward operator of the reindex-reduce operator:

$$y = \text{reindex\_reduce}(x, S_y, f) \underset{\text{backward}}{\Rightarrow} e_x = \text{reindex}(e_y, S_x, f). \tag{4}$$

The backward operator of an element-wise operator is still an element-wise operator. For example, for the operator $f(x) = x^2$, the corresponding backward operator is $g(x, e_y) = f'(x)e_y = 2xe_y$.

Thus all of our meta-operators are inside a backward closure. With backward closure, forward and backward operators can be handled in the same way during unified graph execution, allowing more opportunities for operator fusion and optimization.

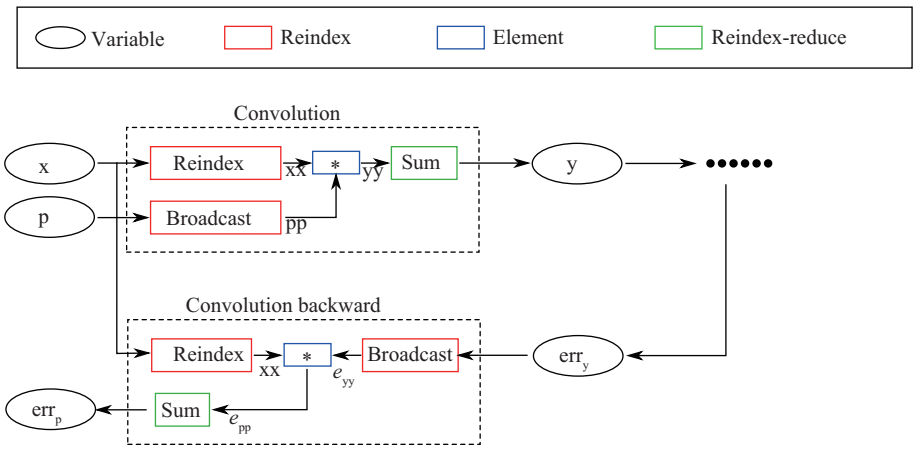

Figure 4 (Color online) Automatic differentiation of a convolution operator implemented using meta-operators.

Let us return to our forward convolution example given earlier in Listing 1, shown again in Figure 4. It consists of four meta-operators, reindex, broadcast, multiplication and sum. We also show the corresponding backward operation. The sum operator (green box) is replaced by the broadcast operator (red box). $\mathrm{err_y}$ is back-propagated to $e_{\mathtt{yy}}$ by the broadcast operator. The corresponding formulation of the broadcast operator is

$$\mathtt{y}_{i0,i1,i2,i3} = \sum_{i4,i5,i6} \mathtt{yy}_{i0,i1,i2,i3,i4,i5,i6} \underset{\text{backward}}{\Rightarrow} e_{\mathtt{yy}\ i0,i1,i2,i3,i4,i5,i6} = e_{\mathtt{y}\ i0,i1,i2,i3}. \tag{5}$$

Because Figure 4 only shows error propagation from output to parameters, error propagation to the input data is omitted. The back-propagation path does not pass through the reindex operator, so the reindexing operator in forward graph is duplicated directly in the backward graph.

The backward operator corresponding to the element-wise multiplication operator is

$$\mathtt{yy} = \mathtt{xx}\ \mathtt{pp} \underset{\text{backward}}{\Rightarrow} e_{\mathtt{pp}} = \mathtt{xx}\ e_{\mathtt{yy}}. \tag{6}$$

The broadcast operator in the forward graph becomes a sum operator in the backward graph, as follows:

$$\mathtt{pp}_{i0,i1,i2,i3,i4,i5,i6} = \mathtt{p}_{i1,i4,i5,i6} \underset{\text{backward}}{\Rightarrow} e_{\mathtt{p}\ i1,i4,i5,i6} = \sum_{i0,i2,i3,i4} e_{\mathtt{pp}\ i0,i1,i2,i3,i4,i5,i6}. \tag{7}$$

Combining Eqs. (1), (5)–(7), the fused backward convolution operation is given by

$$\mathtt{y}_{i0,i1,i2,i3} = \sum_{i4,i5,i6} \mathtt{x}_{i0,i4,i2-i5,i3-i6}\ \mathtt{p}_{i1,i4,i5,i6} \underset{\text{backward}}{\Rightarrow} e_{\mathtt{y}\ i0,i1,i2,i3} = \sum_{i0,i2,i3,i4} \mathtt{x}_{i0,i4,i2-i5,i3-i6}\ e_{\mathtt{p}\ i1,i4,i5,i6}. \tag{8}$$

3.4 Variables

Variables are the basic elements used by Jittor to store data. The inputs and outputs of all operators are variables. Variables are tensors with the following properties:

- A `shape` attribute.
- A data type attribute, `dtype`, e.g., `float` or `int`.
- A `stop_grad` attribute to prevent gradient back propagation for this variable.
- A `stop_fuse` attribute to prevent fusion of operators associated with this variable.

The `stop_grad` attribute is usually used in testing or inferencing, while the `stop_fuse` attribute provides control over operator fusion: the user may get better performance by careful use of these attributes. For example, operator fusion will consume register resources in a GPU. With sufficient resources, fusion will always improve performance, but fusing hundreds of operators will exhaust resources and cause performance degradation.

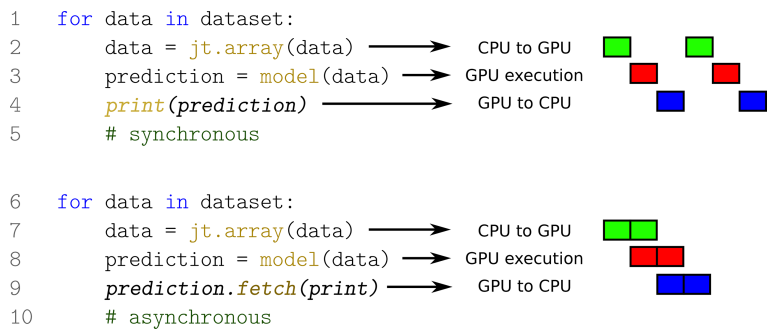

Figure 5 (Color online) Coding and timeline differences between synchronous and asynchronous interfaces.

Jittor allows synchronous and asynchronous data access. Synchronous access is simple to use and can be used interactively from the command line in a similar way to access Numpy arrays. Asynchronous access is more efficient as it does not generate synchronous instructions, and the computing pipeline is not locked; a callback function is required to access the data. Figure 5 shows how the user writes code in these two different ways, and diagrammatically indicates the difference in performance.

Line 1 loops over the entire dataset. Line 2 copies the data into the GPU. Line 3 executes the model on the GPU. Line 4 copies the data back to the CPU and prints it. In the synchronous case, because the print function is synchronous, it blocks iterations (see Subsection 3.4), so model execution and memory copying cannot be overlapped. The asynchronous version instead uses `print` as a callback function of data fetching. This callback is called after the data transfer is complete, allowing model execution and memory copy to be overlapped. The timelines for the two approaches thus differ: overlaps in the asynchronous version provide better performance.

3.5 End-user example

We now give an example in Listings 2–4 showing how an end-user would use Jittor to model a two-layer neural network and train it, in just a few lines of Python code. In Listing 2, we define a linear layer and a sigmoid activation function. (In practice, both are already provided by the Jittor framework, so would not need to be defined. We do so here for the sake of a simple example.) In Listing 3, we define our model, a two-layer fully connected neural network with sigmoid activation. Listings 2 and 3 provide definitions for Listing 4, which creates and trains the model.

Listing 2 Layer definitions

```
 1: import jittor as jt
 2:
 3: class Linear(jt.Module):
 4:     def __init__(n_in, n_out):
 5:         self.w = jt.random((n_in, n_out))
 6:         self.b = jt.random((n_out,))
 7:
 8:     def execute(self, x):
 9:         return jt.matmul(x, self.w) + self.b
10:
11: def sigmoid(x):
12:     return jt.exp(x) / (jt.exp(x) + 1)
```

Listing 2 shows how to implement layers and functions in Jittor. It defines a linear layer class and a sigmoid activation function. Layers are classes as their instances store their parameters. A linear layer is a commonly used fully connected layer, with two parameters: weights $\boldsymbol{W}$ and bias $\boldsymbol{B}$. Its input is $\boldsymbol{X}$; its output is $\boldsymbol{XW} + \boldsymbol{B}$. All of these quantities are matrices. In lines 5 and 6 `self.w` represents the weights and `self.b` represents the bias. Both are initialized using a uniform random operator. Line 9 defines the execution function of the linear layer, where `jt.matmul` performs matrix multiplication.

Listing 3 Model definition

```
1: class Model(jt.Module):
2:     def __init__(self):
3:         self.net = jt.Sequential(
4:             Linear(1, 10),
5:             sigmoid,
6:             Linear(10, 1)
7:         )
8:
9:     def execute(self, x):
10:        return self.net(x)
```

Listing 4 Training with gradient descent

```
1: learning_rate = 0.1
2: model = Model()
3: params = model.parameters()
4: for data, label in user_defined_data:
5:     data, label = jt.array(data), jt.array(label)
6:     predictions = model(data)
7:     loss = (predictions - labels)**2
8:     loss.fetch(print)
9:     grads = jt.grad(loss, params)
10:    for p,g in zip(params, grads):
11:        p.update(p - g * learning_rate)
```

A sigmoid is a commonly used activation function in neural networks: $S(x) = e^x/(e^x + 1)$. If such a definition were used in traditional imperative style deep learning, as used in PyTorch [5], four operators would be executed to evaluate a sigmoid: exp (twice), + and /. Because intermediate values would need to be stored in memory, and memory access consumes time, this would be slow, so PyTorch provides a dedicated operator `torch.sigmoid` to eliminate the need for intermediate values, improving performance. The user must use the provided function to avoid performance degradation. In Jittor, users do not need to consider such performance issues. JIT compilation allows the Jittor back-end to automatically fuse these operators into a single operator (see Section 4). This is a major contribution of Jittor: we provide high performance and customization via user-defined functions at the same time.

Listing 3 defines our model, a two-layer neural network. `jt.Sequential` indicates that the model is executed from the first item (layer or function) to the last item sequentially. In lines 4 and 6, the arguments to `Linear` are the numbers of input and output channels. Thus, the size of the hidden layer is 10, and a sigmoid activation function is used.

After defining the model class, we can now create an instance of it and train it, as shown in Listing 4. Line 2 makes an instance of our model. Line 3 obtains its parameters, in this case the weight and bias of each of the two `Linear` layers. Line 4 traverses the entire user defined dataset. In each iteration, line 5 converts `data` and `label` into Jittor variables, line 6 makes a prediction using the model, line 7 calculates the L_2 loss, line 8 prints the loss (asynchronously, rather synchronously, for reasons explained earlier), and lines 9–11 train this model using simple gradient descent to update the parameters.

Jittor applies graph- and operator-level optimizations via the back-end. In this example, these optimizations include the following. Operator fusion allows the activation function and loss function to be fused. Parallelism improves the performance of compute-intensive operations on multi-core CPUs and GPUs. Concurrency: for example, the `jt.array` operation copies the CPU host memory into GPU device memory if the GPU is used, and memory copy and execution can be concurrent and overlapped.

3.6 Other features

Jittor also provides a code operator, a JIT compilation operator based on a high-performance language, allowing users to directly inline C++ or CUDA code in Python. Jittor also provides distributed operators

for distributed training, based on MPI [16] and NCCL[1)]. NCCL is a dedicated communication library for Nvidia GPUs. Data parallel distributed training can be achieved via use of an appropriate command line.

4 The back-end

This section gives details of the back-end. The back-end is responsible for resource management, process scheduling and compilation optimization. It includes the operator fuser, which decides the fusing strategy used for the meta-operators, external operators, which are customized operators provided by users or third-party libraries, the JIT compiler, the integrated compiler used to optimize meta-operators, and the unified graph execution, which unifies static and dynamic graphs execution.

4.1 Operator fuser

The operator fuser is an important part of the back-end of Jittor. It is responsible for operator fusion optimization in arbitrary computational graphs. In Subsection 3.2, we showed an example of operator fusion using a convolution computation. In real applications, the computational graph produced by the front-end is much more complicated. To optimize arbitrarily cases, we consider the computational graph as a directed acyclic graph of vertices and edges, $G = (V, E)$, in which each node v represents an operator, while each edge e represents a variable. We wish to partition G into multiple sub-graphs $G'_i \subseteq G$, where each sub-graph $G'_i = (V'_i, E'_i)$ represents a fused operator, each node belongs to exactly one sub-graph, and each edge may belong to a sub-graph or link two sub-graphs. The aim is to select one partitioning for which the cost of executing all sub-graphs is minimised. However, accurate prediction of actual execution costs is infeasible: they depend on many aspects of the hardware and other factors. So we use a simplified approach to determine the sub-graphs by defining the cost C as

$$C = \sum_{e \in E \ \wedge \ \forall i (e \notin E'_i)} w_e, \tag{9}$$

where w_e is simply the size of the variable represented by edge e. Eq. (9) sums the weight of each edge e that links two different sub-graphs, and so does not belong to any sub-graph G'_i. This cost is equivalent to the total number of read and write instructions. This approach is justified as most deep learning models are constrained by memory bandwidth. Fusion can improve performance by reducing memory operations. While minimizing the cost, the following rules need to be met.

• **Rule 1.** Reindex operators may not be fused with preceding meta-operators, as such fusion usually causes performance degradation. For example, consider two operators, an element-wise add operator (Eq. (10)), and a reindex broadcast operator (Eq. (11)). The fusion of those two operators is defined in Eq. (12).

$$\forall i \in [0, n) \quad c_i = a_i + b_i, \tag{10}$$

$$\forall j \in [0, m), \ \forall i \in [0, n) \quad d_{j,i} = c_i, \tag{11}$$

$$\text{fused}: \ \forall j \in [0, m), \ \forall i \in [0, n) \quad d_{j,i} = a_i + b_i, \tag{12}$$

Eq. (10) has n addition instructions, $2n$ read instructions and n write instructions. Eq. (11) has nm read instructions and nm write instructions. However, Eq. (12) has nm addition instructions, $2nm$ read instructions and nm write instructions. The cost of Eq. (12) is greater than the sum of costs of Eqs. (10) and (11), illustrating why such fusion is forbidden.

• **Rule 2.** Reindex-reduce operators may not be fused with following meta-operators. Such fusion does not improve performance. For example, consider two operators, a reindex-reduce sum operator

1) https://docs.nvidia.com/deeplearning/nccl/.

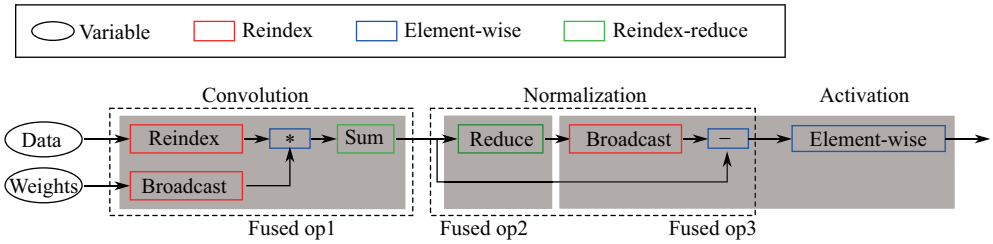

Figure 6 (Color online) Fused operation for a classic network combination, convolve-normalize-activate. Note how operators are fused across convolution, normalization and activation layers.

(Eq. (13)), and an element-wise add operator (Eq. (14)). The fusion of these two operators is defined in Eq. (15).

$$\forall i \in [0,n) \quad b_i = \sum_{j\in[0,m)} a_{j,i}, \tag{13}$$

$$\forall i \in [0,n) \quad d_i = b_i + c_i, \tag{14}$$

$$\text{fused}: \ \forall i \in [0,n) \quad d_i = \sum_{j\in[0,m)} a_{j,i} + c_i. \tag{15}$$

Eq. (13) has nm addition instructions, nm read instructions and n write instructions. Eq. (14) has n addition instructions, $2n$ read instructions and n write instructions. However, Eq. (15) has $nm+n$ addition instructions, $nm+n$ read instructions and n write instructions. While the cost of Eq. (15) is slightly smaller than sum of costs of Eqs. (13) and (14), in our experiments, we find that such fusion does not improve performance. Furthermore, the more complicated the fused operator is, the more difficult it is for the compiler to optimize it. So, such fusion is forbidden.

• **Rule 3.** Fusion should not create directed cycles between sub-graphs. For example, given a graph with three nodes and three edges: $(1 \to 2)$, $(2 \to 3)$, $(1 \to 3)$, if the third edge were to be fused, it would produce a cycle between sub-graph $(1,3)$ and (the trivial) sub-graph 2 in the result: $(1,3) \leftrightharpoons 2$.

A greedy algorithm is used to minimize cost: in each iteration, we select an edge $e = (v_{\text{start}}, v_{\text{end}})$ that satisfies Rules 1–3, and fuse $v_{\text{start}}, v_{\text{end}}$ into the sub-graph G'_i that v_{start} belongs to, repeating until no edge satisfying Rules 1–3 can be found. In practice, a dynamic programming labeling algorithm is used to avoid repeatedly searching for an edge that satisfies the rules. This algorithm works well and achieves competitive performance for most neural networks.

Figure 6 illustrates operation fusion for a classic network combination, convolve-normalize-activate. The convolution layer is composed of two reindex operators, one element-wise operator and one reindex-reduce operator. The normalization layer is composed of one reindex operator, one element-wise operator and one reindex-reduce operator. The activation layer is composed of multiple element-wise operators. In this case, operators can be fused across convolution, normalization and activation layers.

4.2 JIT compiler

The JIT compiler is the built-in compiler used to optimize fused meta-operators. After the operator fuser does its work, the JIT compiler compiles the fused operators into high-performance C++ code.

Listing 5 shows a Jittor Python implementation of instance norm, which normalizes an input tensor `x` into an output tensor `norm_x`. The memory layout is `[N,C,H,W]`, `N` denotes the batch size, `C` denotes the number of channels, `H` is the image height, and `W` is the image width. We calculate mean and variance by reducing the first, third, and fourth dimensions.

Listing 6 gives the corresponding C++ code output by the JIT compiler. (This is only one possible compilation result. Actual results may vary due to the runtime environment.)

In Listing 5, operators created by lines 2–4 are fused into a single nested C++ loop (lines 1–15 in Listing 6). Line 2 in Listing 5 specifies that this operator is to take averages along the three dimensions (batch, image height and image width). `keepdims` indicates that the shape of the output is [1,C,1,1]

Listing 5 Jittor implementation of instance norm in Python

```
1: def instance_norm(x, eps=1e-5):
2:     xmean = jt.mean(x, dims=[0,2,3], keepdims=True)
3:     x2mean = jt.mean(x*x, dims=[0,2,3], keepdims=True)
4:     xvar = x2mean-xmean*xmean
5:     norm_x = (x-xmean)/jt.sqrt(xvar+eps)
6:     return norm_x
```

Listing 6 Results of JIT compilation

```
1: for (int i1=0; i1<x.shape(1); i1++) {
2:     xmean(i1) = 0;
3:     x2mean(i1) = 0;
4:     for (int i0=0; i0<x.shape(0); i0++) {
5:         for (int i2=0; i2<x.shape(2); i2++) {
6:             for (int i3=0; i3<x.shape(3); i3++) {
7:                 xmean(i1) += x(i0,i1,i2,i3);
8:                 x2mean(i1) += sqr(x(i0,i1,i2,i3));
9:             }
10:        }
11:    }
12:    xmean(i1) /= x.shape(0)*x.shape(2)*x.shape(3);
13:    x2mean(i1) /= x.shape(0)*x.shape(2)*x.shape(3);
14:    xvar(i1) = x2mean(i1) - sqr(xmean(i1));
15: }
16: for (int i0=0; i0<x.shape(0); i0++) {
17:    for (int i1=0; i1<x.shape(1); i1++) {
18:        for (int i2=0; i2<x.shape(2); i2++) {
19:            for (int i3=0; i3<x.shape(3); i3++) {
20:                norm_x(i0,i1,i2,i3) =
21:                  (x(i0,i1,i2,i3)-xmean(i1))/
22:                  sqrt(xvar(i1)+eps);
23:            }
24:        }
25:    }
26: }
```

rather than [C], for the convenience of subsequent operations. In Listing 6, the outermost loop variable is `x.shape(1)`, and inner loop variables are `x.shape(0)`, `x.shape(2)`, and `x.shape(3)`, as this order provides the best memory continuity in the `jt.mean` operator. Line 5 in Listing 5 creates 4 element-wise operators: subtraction, division, square root and addition, which are fused into one nested C++ loop (lines 16–26 in Listing 6). The order of nesting of loops in this case is `0,1,2,3`, unlike the previous nested loop, as this is the order with the best memory continuity for these element-wise operators.

The C++ code is further optimized by LLVM compatible passes, based on the specific hardware environment. Common optimizations provided here include loop reordering, loop fission, loop fusing, data packing, vectorizing, and CUDA / OpenMP parallelization.

The JIT compiler and meta-operators make development of custom operators a much easier process. To develop a high-performance custom operator in a traditional deep learning framework, a developer typically needs to develop a CPU forward operator, a CPU backward operator, a GPU forward operator, a GPU backward operator, with possibly separate float32 and float64 versions, and so on. This is a huge programming workload. Jittor can do all of this automatically for the developer.

4.3 Unified graph execution

Unified graph execution is another major contribution of Jittor. According to the execution method of computational graphs, deep-learning frameworks can be based either on a static graph execution (also called a define-and-run approach) or a dynamic graph execution (define-by-run, eager execution). Static graph based frameworks are efficient and easy to optimize, and dynamic graph based frameworks are

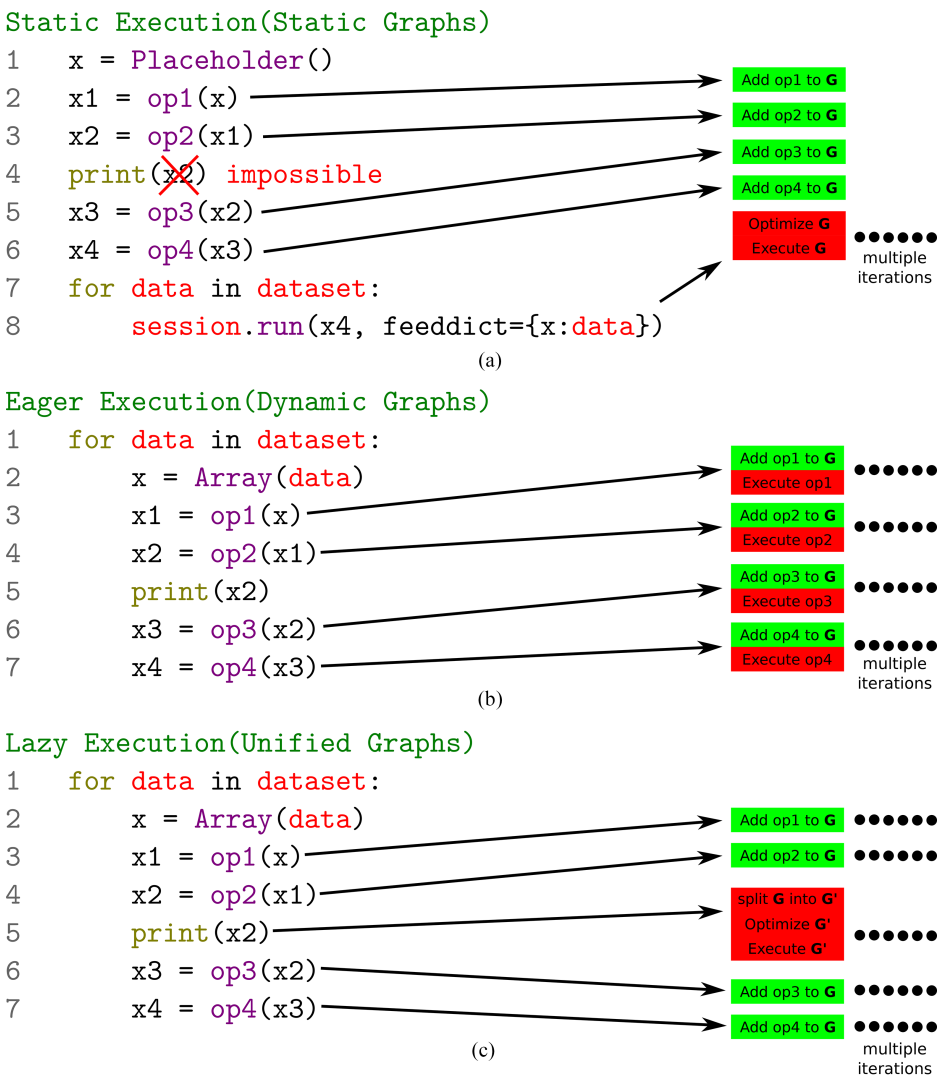

Figure 7 (Color online) A comparison of execution methods using (a) static graphs, (b) dynamic graphs, and (c) unified graphs.

easy-to-use and flexible. Most current frameworks, including TensorFlow 2.0's eager mode, PyTorch, and Chainer, support dynamic graphs.

As an alternative, we propose our unified graph execution approach. Unified graph execution provides an imperative style interface which has the same flexibility as a dynamic graph. And it is also as efficient as a static graph. Figure 7 compares the different execution methods of static graphs, dynamic graphs and unified graphs.

Static graph execution defines the model before running, and then executes the plan with batches of data. In Figure 7(a), line 1 uses a placeholder method to represent input of the data needed for the whole computational graph. Most static execution frameworks use this method, e.g., TensorFlow's `tf.placeholder()`. The operator is added to the computational graph G first (lines 1–6), and then the graph G is optimized and executed in multiple runs (lines 7 and 8). Line 4 is invalid: `x2` cannot be printed as nothing is executed until `session.run` is called. The advantage of static graphs is that they are simple, and easy to optimize and deploy. The graph in Figure 7(a), for example, is built from all four operators before execution, allowing operator fusion to be performed first. The disadvantage is lack of flexibility: debugging a static graph is difficult; e.g., we cannot print intermediate results during building as suggested by line 4. While there are workarounds to this problem, they are unnatural. It is even harder to change the model according to intermediate results found during training. To do so, the user

must rebuild the whole graph, and the resultant performance degradation is unacceptable.

To solve these problems, dynamic graphs were proposed [5, 17], using eager execution. Using dynamic graphs, the graph is rebuilt and executed on each iteration, allowing the graph to be changed during execution. Eager execution executes each operator immediately when it is added to the graph. Because addition of operators is performed on the CPU while they are executed on the GPU, eager execution will lower the latency between CPU and GPU, thus reducing overheads, allowing this approach to achieve competitive performance with the static graph approach. As shown in Figure 7(b), this furthermore allows the user to manipulate intermediate results during model building. This provides users with a great deal of flexibility: for example data can be printed, and the model can be changed according to the intermediate results obtained, which is hard to do with static graphs, and is essential in applications such as generative adversarial networks (GANs) [18] and reinforcement learning [19]. For example, when training a GAN, the computation graph keeps changing between the discriminator and generator. This flexibility has made dynamic graphs popular, and most frameworks (such as TensorFlow 2.0's eager mode, PyTorch, and Chainer) currently support them.

However, dynamic graphs preclude graph optimization as each operation is executed as it is built. This prevents operations from being inter-operator optimized, such as operator fusion. When static graphs are used, the entire graph is available before execution, allowing operator fusion for more efficient execution.

To obtain the benefits of both approaches, without their drawbacks, we use a unified graph execution approach. It provides the full flexibility of a dynamic graph, and the graph can be rebuilt frequently without performance degradation, yet operator fusion is still possible. This is achieved by lazy execution. See Figure 7(c). Operators interpreted by Python are not executed immediately, but delayed until their results are needed. `op1` in line 3 is not executed until `x2` is printed: `x2` is needed at that point, and it depends on `x1` which in turn requires `op1` to be executed. During printing in line 5, three things happen. First, unified execution will select all those operators in graph G that are required by printing, and split them off into a new sub-graph G'; in Figure 7 this is `op1` and `op2`. The sub-graph G' is then optimized using the operator fusion process in Subsection 4.1: the operator fuser takes G' as input, and partitions G' into multiple sub-graphs G''_i, where each sub-graph represents one fused operator. Finally, sub-graph G' is executed. In this very simple example, as `op1` and `op2` are executed together, there is an opportunity to fuse them before doing so. While addition and execution of operators is coupled in the dynamic graph, it is decoupled in the unified graph.

Note that if there is no dependency between two sub-graphs, then they can be executed concurrently, if hardware resources permit. Jittor's algorithm for concurrency control keeps track of the state (running, pending or finished) for each executing sub-graph.

In addition to lazy execution, unified graph execution also provides two other features: cross iteration fusion and asynchronous operation. As operators for different iterations may exist in the unified graph G at the same time, this allows operators be fused across different iterations. For synchronous interfaces, the CPU will start all operators of G' on the GPU and then wait until the GPU finishes its job. However, for an asynchronmous interface using a callback, the CPU does not have to wait, and instead the GPU will call the callback when finishing its job. This reduces the time the CPU spends for waiting, thereby improving efficiency.

Figure 8 illustrates the differences between dynamic graph execution and unified graph execution. The top part explains the symbols used. The three types of meta-operators are given different colors. Two operator pipelines are shown: ovals indicate addition of operators to the graph in the order users create them, where operators are added when Python interprets a line containing them; boxes indicate optimization and execution of operators, and show the order in which operators are executed.

The second part shows part of a computational graph dynamically created by the user. `op1` is an element-wise operator, whose input is the output of previous operators, omitted here. `op2` is a reindex operator, whose input is also from previous operators. `op3` is an element-wise operator, with input from `op2`. `op4` is a reindex-reduce operator with input from `op3`. `op5` has two inputs, which come from previous operators and `op1`. The input for `op6` comes from `op5`. `op7`'s inputs come from `op4` and `op6`. `op8`'s input comes from `op7`, and its output is used by upcoming operators.

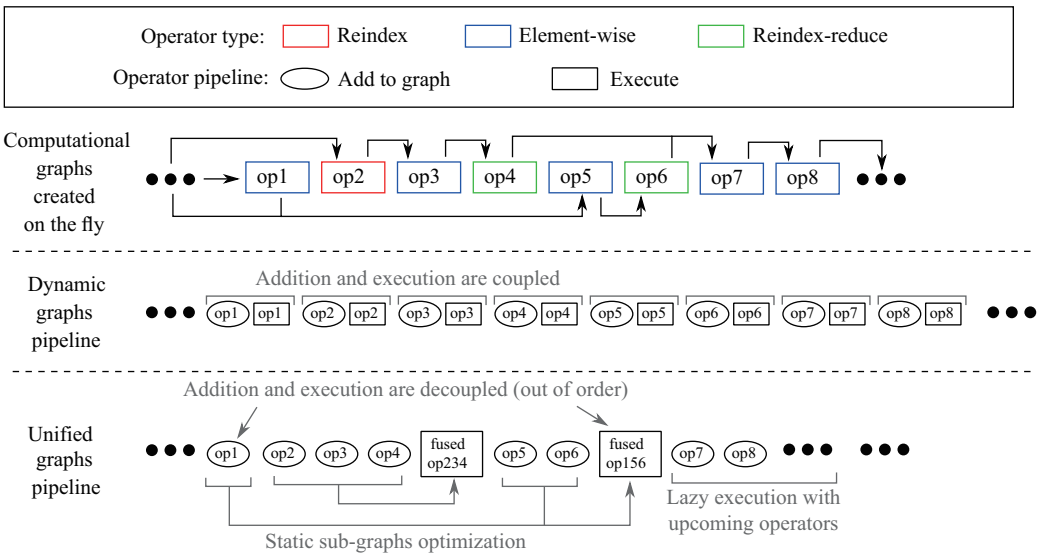

Figure 8 (Color online) Execution pipelines for dynamic and unified graphs. The unified graph can be executed out of order; addition and execution of operators are decoupled. This permits optimization and fusion of operators in static sub-graphs.

The third part shows the execution pipeline for a dynamic graph. This pipeline is very simple and straightforward. `op1` is executed immediately after it is created, and so are `op2`, `op3`, ... Creation and execution are coupled, so there is no scope for optimization.

The fourth part shows the execution pipeline for a unified graph. Those operators are executed out of order: the addition and execution orders differ. With this improvement, unified graph execution can optimize and fuse operators on the fly. First, `op1` is created, but it is not executed immediately because of lazy evaluation. In turn `op2`, `op3`, `op4` are also created in order. However, using our previous methods, `op2`, `op3`, and `op4` can be fused before execution. The static sub-graph is further optimized by the JIT compiler. After `op2`, `op3`, `op4` have been executed, `op5` and `op6` are created, and they can be fused with `op1` and executed with it. While `op1` was created before `op2`, it ends up being executed after `op2`. This out-of-order execution or lazy evaluation approach allows `op1` to be fused with subsequent operators. Similarly, `op7` and `op8` are also not evaluated until needed by upcoming operators.

In summary, our unified graph execution approach has the following characteristics.

• It unifies dynamic and static computational graph approaches, making it simultaneously easy to use and efficient.

• Backward closure of meta-operators allows the fusion of the forward and backward computational graphs.

• It uniformly manages computational graphs across multiple iterations, i.e., cross iteration fusion can be achieved. This optimization is especially useful in certain inferencing scenarios, such as fusing multiple small batches into one big batch.

• It uniformly manages CPU and GPU memory, swapping GPU memory into CPU memory when there is insufficient GPU memory. This is especially useful in certain training scenarios. Users can do training tasks using larger batch sizes beyond the GPU memory limit, and using CUDA unified memory.

• Synchronous and asynchronous interfaces are both provided. Synchronous and asynchronous operations are scheduled to ensure data consistency during concurrent data reading, memory copying, and computing.

These improvements enable unified graph execution to maximize the advantage of JIT optimization, while providing an easy-to-use interface for users.

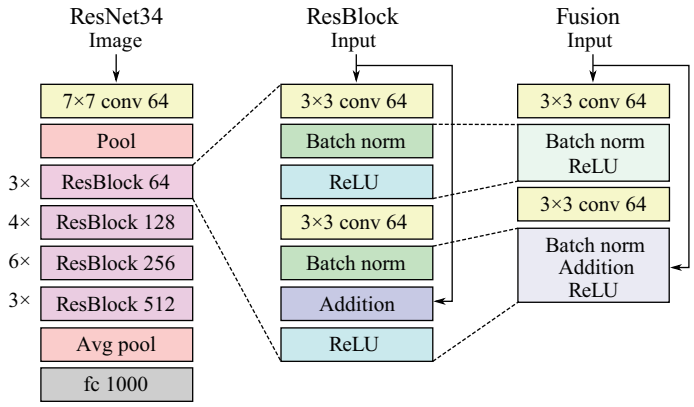

Figure 9 (Color online) Components of ResNet34 and the way it is fused and optimized.

4.4 Analysis on ResNet34

In this subsection, we will take ResNet34 [20] as an example, and analyze how unified graph execution improves performance. ResNet34 model has 33 convolution layers and 1 fully connected layer in total. The header of ResNet34 consists of one convolution layer (7×7 kernel size, 64 output channels) and one pool layer. The tail of ResNet34 consists of one average pool layer and a fully connected layer (the number of output channels is 1000). All five ResNets (ResNet18, 34, 50, 101, 152) share the same header and tail. In a classification task, an extra softmax layer is also added in the tail of ResNet. The number of layers in the header and tail is much smaller than the number of layers in the body. So the body is the most time-consuming part of the whole ResNet. The body of ResNet34 consists of 16 ResBlocks with different channel sizes. ResBlock is a basic component of ResNet. One ResBlock consists of 2 convolution layers, 2 batch norm layers, 2 ReLU activation functions, and one residual learning addition. When the ResNet34 is implemented by Jittor, the unified graph will analyze it and split it into multiple sub-static graphs. In Figure 9, the first batch norm and the first ReLU will be fused; the second batch norm, the addition, and the second ReLU will be fused. The fusion is impossible when eager execution is used, but easy to achieve when lazy execution is used. In this example, 7 operators per block will be executed in eager execution. However, in lazy execution, only 4 operators per block will be executed after operator fusion. This saves lots of memory I/O and makes the model more efficient and execute faster.

Because of the backward closure of meta-operators, back-propagation can also get the performance improvement brought by operator fusion. In Jittor, this improves the performance of training and testing at the same time.

5 Evaluation

Recently, PyTorch has become the most popular deep learning platform because of its use of dynamic graphs, making it convenient for users to design and debug deep learning models. About 70% of works published in CVPR 2020 were based on PyTorch. Because Jittor adopts unified graph execution and operator fusion strategies, it provides advantages over PyTorch in both performance and convenience, as we now show in two experiments in this section. The first experiment considers evaluation of backbone networks (i.e., inferencing), which are the most important part of deep learning models: most of the computational cost of a model comes from its backbone network. The second experiment concerns image generation with a GAN [18], a very popular research field.

5.1 Backbone networks

Ten of the most successful and popular backbone networks for image recognition were chosen for this experiment: AlexNet [21], VGG [22], ResNet50, ResNet152 [20], Wide ResNet50, Wide ResNet101 [23], SqueezeNet [24], ResNEXT50, ResNEXT101 [25], and Res2Net [26]. Res2Net50 [26] is the newest, and

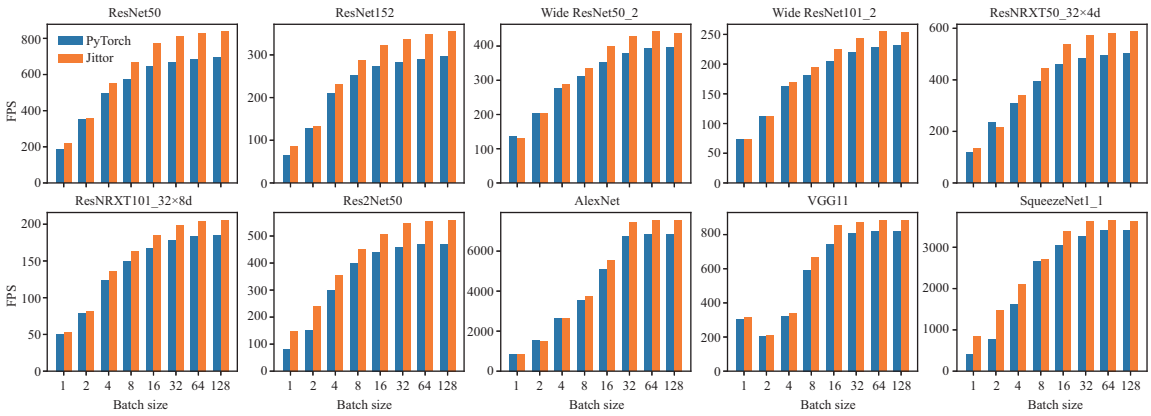

Figure 10 (Color online) Inferencing speed comparison of Jittor and PyTorch.

Table 1 Inferencing speed comparison (FPS) between Jittor and PyTorch[a)]

Model	Batch size															
	1		2		4		8		16		32		64		128	
	PT	JT	PT	JT	PT	JT	PT	JT	PT	JT	PT	JT	PT	JT	PT	JT
ResNet50 [20]	185	**220**	353	**357**	492	**548**	575	**667**	643	**773**	668	**810**	680	**829**	692	**836**
ResNet152 [20]	64	**86**	126	**132**	209	**231**	251	**287**	273	**321**	283	**335**	288	**346**	296	**354**
Wide ResNet50_2 [23]	134	**131**	**204**	202	275	**288**	310	**335**	353	**397**	379	**426**	391	**441**	397	**437**
Wide ResNet101_2 [23]	**73**	72	111	**112**	162	**169**	180	**194**	203	**225**	220	**243**	228	**254**	230	**253**
ResNEXT50_32×4d [25]	118	**132**	236	**216**	310	**341**	393	**445**	458	**536**	484	**572**	495	**579**	503	**586**
ResNEXT101_32×8d [25]	50	**53**	78	**81**	123	**136**	149	**162**	166	**185**	178	**198**	183	**204**	185	**205**
Res2Net50 [26]	79	**149**	152	**240**	299	**355**	399	**451**	441	**507**	460	**547**	468	**553**	468	**559**
AlexNet [21]	**865**	818	**1562**	1500	**2626**	2622	3553	**3745**	5070	**5546**	6736	**7431**	6836	**7531**	6856	**7551**
VGG11 [22]	303	**315**	201	**208**	322	**337**	593	**665**	741	**855**	808	**873**	818	**883**	820	**885**
SqueezeNet1_1 [24]	404	**842**	769	**1461**	1619	**2102**	2656	**2700**	3035	**3382**	3258	**3628**	3406	**3658**	3407	**3631**

a) The boldface represents the faster framework for the same model and batch size.

the state-of-the-art model for most image recognition tasks. AlexNet [21] won first place in ImageNet ILSVRC-2012; its performance far exceeded that of the second placed method. ResNet [20] won first place in ImageNet ILSVRC-2016, by a large margin. These ten networks summarize the development of convolution backbone networks from 2012 to the present.

Figure 10 shows an inferencing speed comparison between Jittor and PyTorch. The horizontal axis shows different batch sizes, while the vertical axis is number of frames processed per second (FPS). A quantitative assessment is given in Table 1; PT means PyTorch and JT means Jittor. The experimental environment is an 11 GB Nvidia 1080ti GPU, an Intel i7-6850K CPU, 32 GB RAM, FP32, PyTorch benchmark switch on. Taking an average over the above 10 networks, the frame rate of Jittor is 26% higher than that of PyTorch for small batches and 13% higher for large batches. More noteworthy, the frame rate of Jittor is 101% higher for a small batch with Res2Net50. Because of the operator fusion optimization in Jittor, small batches are processed much faster than in PyTorch. For some classic networks like AlexNet, the frame rate of Jittor is 10% higher for large batches but 5% lower for small batches. This is because the architecture of AlexNet is much simpler than that of Res2Net50, so Jittor has less opportunities for optimization, and furthermore, AlexNet is well optimized by underlying libraries like CUDNN.

5.2 Generative adversarial networks

We compared the training speed of Jittor and PyTorch for four GAN models: WGAN-GP [27], DCGAN [28], LSGAN [29], and CycleGAN [30]. Table 2 shows the number of iterations per second, and relative speed of Jittor on two different datasets, MNIST [31] and cityscapes [32], for 4 GAN models.

Table 2 Training speed comparison (number of iterations per second) of 4 GAN models

	WGAN-GP	DCGAN	LSGAN	CycleGAN
Dataset	MNIST	MNIST	MNIST	cityscapes
PyTorch	52.35	37.73	38.61	10.08
Jittor	149.25	78.74	78.74	13.96
Jittor relative speed	2.9	2.1	2.0	1.4

Table 3 Ablation studies about asynchronous interface, cross iteration fusion, unified memory and lazy execution

	FPS	Speedup ratio
PyTorch	253.1	0.965
Without asynchronous interface	254.9	0.972
Without cross iteration fusion	260.4	0.993
Without unified memory	264.7	1.009
Without lazy execution	73.2	0.279
All-features-on	262.2	1.000

Benefiting from meta-operators and unified graph execution, with these GANs, Jittor achieves a maximum relative speed of 2.9 and a minimum of 1.4, with an average relative speed of 2.1.

The performance improvement of WGAN-GP is the most significant. This is because WGAN-GP requires second-order back-propagation, which means the back-propagation of the model requires back-propagation again. The backward closure feature makes our second order back-propagation operators perform better.

All test results are public, and a high-performance GAN library has been released on our website[2)].

5.3 Ablation study

In this subsection, four ablation studies are performed on ResNet50 training. The experimental environment is a 24 GB Nvidia RTX Titan, 64 GB RAM, FP32, and the batch size is 64. Table 3 shows the training FPS of PyTorch and different modes of Jittor, and the speedup ratio compared to the all-features-on mode of Jittor.

The **asynchronous interface** improves performance by overlapping communication and computation. During ResNet50 training, communication between CPU and GPU is not heavy compared to computation, so this only has 3% performance loss on FPS compared to the all-features-on mode.

Cross iteration fusion improves performance by fusion operators across iterations. However, during ResNet50 training, only a part of the batch norm layer can be fused across iterations. So this only has 1% performance loss on FPS.

Unified memory makes memory of CPU and GPU can be swapped. This feature can keep the program running even when there is not enough memory. It costs about 1% performance loss, but it is worthy.

Lazy execution is the fundamental feature of Jittor. Without this feature, all operators cannot be fused at all. Because most operators of Jittor build upon meta-operators, turning off this feature will greatly reduce performance.

6 Conclusion and future work

This paper presents the deep learning framework Jittor, with two major contributions: meta-operators and unified graph execution. Jittor has been released online[3)]. Use of meta-operators makes development and optimization of deep learning operators much easier, and unified graph execution is both easy to use and efficient.

2) https://github.com/Jittor/gan-Jittor.
3) https://github.com/Jittor/.

In future, we plan to develop further libraries for object detection, image segmentation, 3D point cloud processing, 3D mesh processing and related topics. We hope to further optimize performance, usability, and scalability of Jittor. We also note that differentiable programming is gradually gaining attention as a new paradigm to solve machine learning problems, in areas such as differentiable Monte Carlo ray tracing [33], differentiable rasterizers [34] for reverse rendering problems, and DiffTaichi [35] for trainable physical simulation. Future versions of Jittor will support differentiable programming to better solve learning problems in visual computing and geometry.

Acknowledgements This work was supported by National Natural Science Foundation of China (Grant No. 61521002). We would like to thank anonymous reviewers for their helpful review comments, and Professor Ralph R. MARTIN for his useful suggestions and great help in paper writting.

References

1 Collobert R, Bengio S, Mariethoz J. Torch: a modular machine learning software library. In: Proceedings of IDIAP Research Report, 2002
2 Al-Rfou R, Alain G, Almahairi A, et al. Theano: a python framework for fast computation of mathematical expressions. 2016. ArXiv:1605.02688
3 Jia Y, Shelhamer E, Donahue J, et al. Caffe: convolutional architecture for fast feature embedding. In: Proceedings of ACM International Conference on Multimedia, 2014. 675–678
4 Abadi M, Barham P, Chen J, et al. Tensorflow: a system for large-scale machine learning. In: Proceedings of the 12th Symposium on Operating Systems Design and Implementation, 2016. 265–283
5 Paszke A, Gross S, Massa F, et al. Pytorch: an imperative style, high-performance deep learning library. In: Proceedings of Advances in Neural Information Processing Systems, 2019. 8024–8035
6 Cyphers D S, Bansal A K, Bhiwandiwalla A, et al. Intel nGraph: an intermediate representation, compiler, and executor for deep learning. 2018. ArXiv:1801.08058
7 Schoenholz S S, Cubuk E D. JAX, M.D.: end-to-end differentiable, hardware accelerated, molecular dynamics in pure Python. 2019. ArXiv:1912.04232
8 Chen T, Moreau T, Jiang Z, et al. TVM: an automated end-to-end optimizing compiler for deep learning. In: Proceedings of the 13th Symposium on Operating Systems Design and Implementation, 2018. 578–594
9 Nickolls J, Buck I, Garland M, et al. Scalable parallel programming with CUDA. In: Proceedings of IEEE Hot Chips 20 Symposium (HCS), 2008
10 Thompson J A, Schlachter K. An introduction to the OpenCL programming model. 2012. https://cims.nyu.edu/~schlacht/OpenCLModel.pdf
11 Oliphant T E. Guide to NumPy. North Charleston: CreateSpace Publishing, 2015
12 Chen T Q, Li M, Li Y T, et al. MXNet: a flexible and efficient machine learning library for heterogeneous distributed systems. 2015. ArXiv:1512.01274
13 Chetlur S, Woolley C, Vandermersch P, et al. cuDNN: efficient primitives for deep learning. 2014. ArXiv:1410.0759
14 Lattner C, Adve V S. LLVM: a compilation framework for lifelong program analysis & transformation. In: Proceedings of International Symposium on Code Generation and Optimization, 2004. 97–104
15 Rumelhart D E, Hinton G E, Williams R J. Learning representations by back-propagating errors. Nature, 1986, 323: 533–536
16 Gabriel E, Fagg G E, Bosilca G, et al. Open MPI: goals, concept, and design of a next generation MPI implementation. In: Proceedings of European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting, 2004. 97–104
17 Tokui S, Oono K. Chainer: a next-generation open source framework for deep learning. In: Proceedings of Workshop on Machine Learning Systems (LearningSys), 2015
18 Goodfellow I J, Pouget-Abadie J, Mirza M, et al. Generative adversarial nets. In: Proceedings of Advances in Neural Information Processing Systems, 2014
19 Sutton R S, Barto A G. Reinforcement Learning: An Introduction. Cambridge: MIT Press, 2018
20 He K M, Zhang X Y, Ren S Q, et al. Deep residual learning for image recognition. In: Proceedings of IEEE Conference on Computer Vision and Pattern Recognition, 2016. 770–778
21 Krizhevsky A, Sutskever I, Hinton G E. Imagenet classification with deep convolutional neural networks. In: Proceedings of Advances in Neural Information Processing Systems, 2012. 1106–1114
22 Simonyan K, Zisserman A. Very deep convolutional networks for large-scale image recognition. 2015. ArXiv:1409.1556
23 Zagoruyko S, Komodakis N. Wide residual networks. 2016. ArXiv:1605.07146
24 Iandola F N, Han S, Moskewicz M W, et al. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5 MB model size. 2017. ArXiv:1602.07360
25 Xie S N, Girshick R, Dollár P, et al. Aggregated residual transformations for deep neural networks. In: Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017. 5987–5995
26 Gao S H, Cheng M M, Zhao K, et al. Res2Net: a new multi-scale backbone architecture. IEEE Trans Pattern Anal Mach Intell, 2019. doi: 10.1109/TPAMI.2019.2938758
27 Gulrajani I, Ahmed F, Arjovsky M, et al. Improved training of wasserstein GANs. In: Proceedings of Advances in

Neural Information Processing Systems, 2017. 5767–5777
28 Radford A, Metz L, Chintala S. Unsupervised representation learning with deep convolutional generative adversarial networks. In: Proceedings of the 4th International Conference on Learning Representations, 2016
29 Mao X D, Li Q, Xie H R, et al. Least squares generative adversarial networks. In: Proceedings of IEEE International Conference on Computer Vision, 2017. 2813–2821
30 Zhu J Y, Park T, Isola P, et al. Unpaired image-to-image translation using cycle-consistent adversarial networks. In: Proceedings of IEEE International Conference on Computer Vision, 2017. 2223–2232
31 LeCun Y, Cortes C, Burges C J C. The MNIST database of handwritten digits. 2005. http://yann.lecun.com/exdb/mnist/
32 Cordts M, Omran M, Ramos S, et al. The cityscapes dataset for semantic urban scene understanding. In: Proceedings of IEEE Conference on Computer Vision and Pattern Recognition, 2016
33 Li T M. Differentiable visual computing. 2019. ArXiv:1904.12228
34 Kato H, Ushiku Y, Harada T. Neural 3D mesh renderer. In: Proceedings of IEEE Conference on Computer Vision and Pattern Recognition, 2018. 3907–3916
35 Hu Y M, Anderson L, Li T M, et al. DiffTaichi: differentiable programming for physical simulation. In: Proceedings of the 8th International Conference on Learning Representations, 2020