Optimize dynamic neural network models with control flow operators

Today we observe more and more dynamic neural network models, especially in the fields of natural language processing and graph analysis. The dynamics in these models come from multiple sources, including:

- Models are expressed with control flow, such as conditions and loops;
- NDArrays in a model may have dynamic shapes, meaning the NDArrays of a model or some of the NDArrays have different shapes for different batches;
- Models may want to use more dynamic data structures, such as lists or dictionaries.
- ...

It's natural to express the dynamic models in frameworks with the imperative programming interface (e.g., Gluon, Pytorch, TensorFlow Eager). In this interface, users can simply use Python control flows, or NDArrays with any shape at any moment, or use Python lists and dictionaries to store data as they want.

The problem of this approach is that it highly depends on the front-end programming languages (mainly Python). A model implemented in one language can only run in the same language. A common use case is that machine learning scientists want to develop their models in Python but engineers who deploy the models usually have to use a different language (e.g., Java and C). Gluon tries to close the gap between the model development and deployment. Machine learning scientists design and implement their models in Python with the imperative interface and Gluon turns the implementations into symbolic implementations by simply invoking hybridize() for model exporting. However, the current implementation of Gluon hybridization can't support dynamic models as I listed above.

This project is to address some of these limitations in Gluon. Overall, the goal of the project is to turn a dynamic neural network into a static computation graph (where the dynamic control flows are expressed by control flow operators) with Gluon hybridization and export them for deployment.

An additional benefit of this work is to improve the speed for both training and inference for dynamic models. First, expressing all computation in a graph and executing it in the backend saves the Python execution overhead. More importantly, expressing computation in a graph is an important step for many optimizations. For example, we can handle variable-length sequences efficiently by avoiding any unnecessary computation (this can be more efficient than bucketing). We can also integrate with TVM to accelerate the models by identifying the static part of the computation and using TVM to compile the static computation for acceleration.

Supporting dynamic features listed above require fundamental modification in the MXNet backend. As such, we'll take an incremental approach and support these features one by one. Currently, our plan is to to support control flow and dynamic shapes in Gluon. The entire project will take four steps:

- Add imperative and symbolic control flow operators to MXNet to switch between imperative and symbolic implementations in Gluon. Currently, we'll add three control flow operators: ``cond'', ``foreach'' and ``while_loop''. This proposal covers the detailed description of these operator APIs and their implementations.
- Support operators with dynamic shape. This happens in multiple cases: boolean index returns an NDArray whose shape depends on the values in the input NDArray; while_loop returns NDArrays whose shapes depend on the computation in the loop; a handful of operators can potentially be extended to return arrays whose shapes depend on the input shape symbol. The design proposal of dynamic shapes cover this task.
- We can go a step further by adding support for building static graphs directly from any Gluon models with Python control flow. This step requires to use the Python parser to reconstruct Python code to turn Python control flow into the code using MXNet control flow operators. While reconstructing the code, we need to analyze the liveness of Python variables and distinguish output variables of the loop and the variables carried over to the next iteration. This task is planned but there has been a design yet.
- To use TVM to compile and optimize dynamic models. Flow control operators understand the computation flow. It can reorganize subgraphs (e.g., the ones in the loop) and pass them to TVM for compilation to achieve better performance.

Implementation of control flow operators

In order to support Gluon hybridization, all control flow operators need to have two versions: one for NDArrays and the other for symbols. The NDArray version is easy to implement. We can simply use Python control flow to implement the operator. The discussion of this section focuses on the implementation of symbol control flow operators.

There are two possible approaches of implementing symbol control flow operators.

Approach 1: The approach is to keep all computation in a single graph and insert some special operators in the graph to change the executor behavior (e.g., avoid executing a sequence of operators, repeat execution of a sequence of operators). These special operators resembles the jump instruction in CPU. TensorFlow takes this approach. The detailed design can be found in http://download.tensorflow.org/paper/white_paper_tf_control_flow_implementation_2017_11_1.pdf

Approach 2: The approach is to maintain multiple computation graphs. There is a main graph that contains control flow operators. Each control flow operator contains its own computation graph(s) and is responsible for executing the computation graphs inside the operator. This approach represents computation more like how a high-level programming language handles control flows (the code is broken into basic block for compilation and execution).

Both approaches have their pros and cons. I think the second approach will be preferred in MXNet because the execution of MXNet is more static than TensorFlow (e.g., MXNet requires static shape and data type inference and static memory planning). The second approach also allows easier graph-level optimization and easier implementation. From here on, I'll discuss the second approach in more details.

Given the API (shown in the end of the proposal), the first step for approach 2 is to build a subgraph in the Python functions of a control flow operator and pass them to the operator. I'll use ``foreach'' as an example to describe the implementation of a symbolic control flow operator.

Building a subgraph: a control flow operator invokes the Python functions once to create a computation subgraph. The Python functions have arguments as input and can also access any variables in its scope. Therefore, the subgraph of ``foreach'' has four sets of inputs: slices from the input arrays we iterate over, the states from the previous iteration or the initial states, the variables outside the Python functions and the variables defined inside the Python functions. The initial states and the variables defined outside the Python functions reference to the symbols in the main graph and, thus, connect the subgraph in the Python functions with the main graph. However, the main graph and the subgraph have to be completely disjoint (otherwise, stateful operators may have different behaviors). To build a completely separate graph, we create new variable symbols to replace the ones in ``state'' and in the closure. It's easy to find the symbols passed in as ``state''. However, it's more difficult to identify the symbols in the closure and the ones defined inside the Python functions, which aren't the inputs of the control flow operator.

- For symbols in the closure, we perform a graph transformation. When we create a computation graph from the Python function, we mark the nodes with a special label. Then we traverse the nodes in the computation graph to find the unmarked nodes, which are the ones not created in the Python functions. We remove the nodes from the computation graph and create new symbols to reference them. Later on, we pass the output of the original nodes to the control flow operator as its inputs.
- For symbols defined inside the Python functions, we make a copy of the original symbols and pass them to the control flow operator as inputs.

Pass the subgraph to a control flow operator: The previous process creates a symbol that represent the computation graph in the Python function. Now we need pass it to the operator as a graph. By default, NNVM interprets input symbols of an operator as input data (see Symbol::Compose in NNVM) and use them to build connections between nodes, which results in a computation graph. To pass a symbol as a subgraph to an operator, we need to distinguish the graph symbols from the data symbols in Symbol::Compose. We create a new operator attribute (FInputGraph) to identify the graph symbols in the inputs of an operator. Once a graph symbol is spotted, nnvm stores it in NodeAttrs of the created node.

Shape and type inference:

- ``foreach'': shape and type inference on this operator is easy. Because the input shapes and types in each iteration is the same, we only need to infer the subgraph once. After having the shape and the type of the output of the subgraph, we can easily calculate the shape and the type of the output of ``foreach''.
- ``cond'': If we require the outputs of ``then_fun'' and ``else_func'' have the same shape and data type, the shape and type inference can also be simple.
- ``while_loop'': We can't perform static shape inference on ``while_loop'' because the shape of its output arrays depends on the number of iterations governed by the condition subgraph. As such, we need to extend MXNet to support dynamic shape inference.

Forward and backward execution: forward/backward computation of a control flow operator is to execute the corresponding subgraph imperatively. Here we can take advantage of CachedOp. However, there are some challenges in executing a subgraph inside an operator because operators are executed in the threaded engine currently:

- The execution engine requires that the data in the output arrays is valid once an operator completes. However, the operators in the subgraph are pushed to the engine for execution. As such, when a control flow operator completes, there is no guarantee that all computation in the subgraph is complete.
- If we wait for the completion of all computation in a subgraph, there is a potential deadlock. The deadlock happens when multiple control flow operators are waiting for subgraphs to complete and there are no more threads for executing tasks pushed by the control flow operators to the engine.

There are two potential solutions for executing a subgraph inside a control flow operator:

- We can execute the subgraph outside the execution engine in the main thread like a normal function call. In this way, the data in the output arrays of a control flow operator is only marked valid by the operators in the subgraph. In the case of ``cond'' and ``while_loop'', we need to wait for the condition subgraph to complete before we can proceed with ``then_func'' or ``else_func'' or the loop body. Potentially, we can still wait in the main thread for the condition subgraph to complete. However, blocking the main thread prevents multiple-GPU parallelism (MXNet has a single execution thread and uses asynchronous execution for multiple-GPU parallelism). As such, we have to allow the subgraph execution in the threaded engine.

Dynamic shape inference and execution: To execute a computation graph with operators that don't support static shape inference, the first step is to perform graph partitioning. Operators before ``while_loop'' and the ones after ``while_loop'' are grouped into separate subgraphs. As such, the original graph is split into multiple subgraphs and we can execute these subgraphs imperatively (just like Gluon does in Python). In this way, we can perform shape inference right before a subgraph is executed. Even though the shape of a subgraph may change in each mini-batch, we can still avoid memory allocation by reusing memory as long as the memory used in the previous execution is large enough.

Memory management: Inference and training tasks require very different memory management strategies. For inference, we only need to allocate memory for an iteration and the final output NDArrays. The memory used by the previous iteration can be reused in the next iteration. However, it's completely a different story for the training. Backward propagation for an iteration requires the intermediate computation results in the forward path. One approach is saving all intermediate computation results in the forward path. If a model requires many iterations (e.g., long sequences) or uses the attention mechanism, we need to allocate astonishing amount of memory, which can be a problem for GPU. To enable training on these workloads, we may have to use small batch sizes. Another possibility is to drop some memory in the forward path, e.g., only saving the outputs of each iteration and recompute the intermediate results in the forward path whenever it's needed. This can save significant amount of memory at the cost of computing the forward path twice. In the case of attention, we should develop a mechanism that systematically identifies the memory generated by cheap computation and drop it in the forward path (Pytorch allows developers to mark these memory manually).

===================================== Attachment ===

The proposed APIs of the flow control operators are listed below:

``cond''

``cond'' invokes different computations based on a certain condition.

> `cond(pred, then_func, else_func)`

Input arguments:

- ``pred'' is a symbol/NDArray that contains a boolean scalar to define which function to compute.
- ``then_func'' and ``else_func'' are Python functions whose signature is defined below.

Return value:

- A list of symbols/NDArrays returned from one of the Python functions.

The signature of ``then_func'' and ``else_func'' is

> def func(): outputs

where ``outputs'' is a list of symbols/NDArrays.

``then_func'' and ``else_func'' should return the same number of outputs with the same types and shapes. This is compatible with ``cond'' in TensorFlow, except the restriction in shapes.

Foreach

``foreach'' is a special form of loops. It's designed to have easy shape inference and other optimizations. It iterates over the first dimension of the input NDArray/Symbol, so the number of iterations is determined before entering the loop.

> *foreach(body, input, state)*

Input arguments:

- ``input'' is a symbol/NDArray or a list of symbols/NDArrays.
- ``body'' is a Python function that defines computation for each iteration.
- ``state'' is a list of symbols/NDArrays passed to ``body'' as part of the inputs for the first iteration.

Return values:

- A tuple of (out_data, state), where ``out_data'' is a symbol/NDArray or a list of symbols/NDArrays that is a concatenation of all outputs from ``body'' and ``state'' is the output state in the last iteration.

The signature of ``body'' is

> *def body(input, state): output, new_state*

``input'' is a symbol/NDArray or a list of symbols/NDArrays that is a slice from the input arrays of ``foreach''; ``state'' is a list of symbols/NDArrays that represent data from the previous iteration; ``output'' is a symbol/NDArray or a list of symbols/NDArrays that contains the output data generated in this iteration; ``new_state'' is a list of symbols/NDArrays that contain data passed to the next iteration. All ``output'' from this function are concatenated as the output of ``foreach''. As such, the shape and type of ``output'' from each iteration should always be the same. ``body'' is invoked once to generate a symbol that represents the computation in the function.

``foreach'' is similar to ``scan'' in TensorFlow. The difference is that ``body'' in ``foreach'' has two types of outputs: one is concatenated as the output of ``foreach''; the other outputs are passed to ``body'' as input for the next iteration. In contrast, ``scan'' concatenates the outputs of ``body'' as the output of ``scan'' and also passes them to ``body'' as one of the inputs for the next iteration. This difference makes the implementation of LSTM with ``foreach'' simpler and more efficient than ``scan'' in TensorFlow. In addition, ``foreach'' allows ``body'' to only return empty ``output'' or empty ``state''. If ``state'' is empty, ``foreach'' becomes map. If ``output'' is empty, ``foreach'' becomes scan.

``while_loop''

``while_loop'' is the general form of a loop: at the beginning of each iteration, it checks a condition function to determine the termination of the loop. It is difficult to determine the number of iterations in advance and is more difficult to optimize ``while_loop'' than ``foreach''.

> while_loop(cond, func, loop_vars, max_iterations)

Input arguments:

- ``cond'' is a Python function that takes ``loop_vars'' as input and return a boolean scalar symbol/NDArray to determine the termination of the loop.
- ``func'' is a Python function that takes ``loop_vars'' as input and performs computation of an iteration.
- ``loop_vars'' is a list of symbols that represent NDArrays.
- ``max_iterations'' is a python scalar or that defines the maximal number of iterations.

Return value:

- Depending on the signature of ``func'', there are two potential ways of returning values.

The signature of ``cond'':

> def cond(loop_vars): boolean scalar

The signature of ``func'':

> def func(loop_vars): (output, new_loop_vars)

In this option, ``output'' from each iteration will be concatenated and returned as the output of ``while_loop''. We require ``output'' to have the same shape and data type. We probably require arrays in ``new_loop_vars'' to have the same shape and data type as ``loop_vars''. This interface is similar to the definition of ``loop'' in ONNX and is more restrictive.

It is difficult to inference the shape of the output of ``while_loop'' statically because we cannot determine the number of iterations required by ``while_loop'' in advance. Because MXNet symbols don't support dynamic shape, we currently use ``max_iterations'' to determine the first dimension of the output arrays.