▸ logsoftmax

▸ logsoftmax_axis

Loop

Generic Looping construct. This loop has multiple termination conditions:

1. Trip count. Iteration count specified at runtime. Set by specifying the input M. Optional. Set to empty string to omit. Note that a static trip count (specified at graph construction time) can be specified by passing in a constant node for input M.
2. Loop termination condition. This is an input to the op that determines whether to run the first iteration and also a loop-carried dependency for the body graph. The body graph must yield a value for the condition variable, whether this input is provided or not.

This table summarizes the operating modes of this operator with equivalent C-style code:

```
    Operator inputs defined as (max_trip_count, condition_var).

    input ("", ""):
        for (int i=0; ; ++i) {
          cond = ... // Note this value is ignored, but is required in
  the body
        }

    input ("", cond) // Note this is analogous to a while loop
        bool cond = ...;
        for (int i=0; cond; ++i) {
          cond = ...;
        }

    input ("", 1) // Note this is analogous to a do-while loop
        bool cond = true
        for (int i=0; cond; ++i) {
          cond = ...;
        }

    input (trip_count, "") // Note this is analogous to a for loop
        int trip_count = ...
        for (int i=0; i < trip_count; ++i) {
          cond = ...; // ignored
        }

    input (trip_count, cond)
        int trip_count = ...;
```

```
      bool cond = ...;
      for (int i=0; i < trip_count && cond; ++i) {
        cond = ...;
      }
```

Sample usage - cond as well as trip count

```
  graph predict-net {
    %a = Constant[value = <Scalar Tensor [3]>]()
    %b = Constant[value = <Scalar Tensor [6]>]()
    %keepgoing = Constant[value = <Scalar Tensor [1]>]()
    %max_trip_count = Constant[value = <Scalar Tensor [10]>]()
    %keepgoing_out, %b_out, %user_defined_vals = Loop[body = <graph
body-net>](%max_trip_count, %keepgoing, %b)
    return
  }

  graph body-net (
    %i[INT32, scalar]           // iteration number
    %keepgoing_in[BOOL, scalar] // incoming loop-termination-
condition; not used
    %b_in[INT32, scalar]        // incoming value of loop-carried-
dependency b
  ) {
    %my_local = Add(%a, %b_in)
    %b_out = Sub(%a, %b_in) // outgoing value of loop-carried-
dependency b
    %keepgoing_out = Greater(%my_local, %b_out) // outgoing loop-
termination-condition
    %user_defined_val = Add(%b_in, %b_in) // scan-output value to be
accumulated
    return %keepgoing_out, %b_out, %user_defined_val
  }
```

Sample equivalent C code

```
  {
    /* User-defined code (enclosing scope) */
    int a = 3, b = 6;
    bool keepgoing = true; // Analogous to input cond
    /* End user-defined code */

    /* Implicitly-defined code */
    const int max_trip_count = 10; // Analogous to input M
    int user_defined_vals[]; // Imagine this is resizable
    /* End implicitly-defined code */
    /* initialize loop-carried variables and scan-output variables */
    bool keepgoing_out = keepgoing
    int b_out = b
```

```
    for (int i=0; i < max_trip_count && keepgoing_out; ++i) {
      /* Implicitly-defined code: bind actual parameter values
         to formal parameter variables of loop-body */
      bool keepgoing_in = keepgoing_out;
      bool b_in = b_out;

      /* User-defined code (loop body) */
      int my_local = a + b_in; // Reading value "a" from the
enclosing scope is fine
      b_out = a - b_in;
      keepgoing_out = my_local > b_out;
      user_defined_val = b_in + b_in; // b_in and b_out are different
variables
      /* End user-defined code */

      /* Implicitly defined-code */
      user_defined_vals[i] = user_defined_val // accumulate scan-
output values
    }
    // int t = my_local; // Can't do this. my_local is not accessible
here.

    // The values below are bound to the output variables of the loop
and therefore accessible
    // b_out; user_defined_vals; keepgoing_out;
  }
```

There are several things of note in this code snippet:

1. Values from the enclosing scope (i.e. variable "a" here) are in scope and can be referenced in the inputs of the loop.
2. Any values computed in the loop body that needs to be used in a subsequent iteration or after the loop are modelled using a pair of variables in the loop-body, consisting of an input variable (eg., b_in) and an output variable (eg., b_out). These are referred to as loop-carried dependences. The loop operation node supplies the input value of the input variable for the first iteration, and returns the output value of the output variable produced by the final iteration.
3. Scan_output variables are used to implicitly concatenate values computed across all the iterations. In the above example, the value of user_defined_val computed over all iterations are concatenated and returned as the value of user_defined_vals after the loop.
4. Values created in the body cannot be accessed in the enclosing scope, except using the mechanism described above.

Note that the semantics of this op support "diagonal" or "wavefront" execution. (See Step 3 here for an example: https://devblogs.nvidia.com/optimizing-recurrent-neural-networks-cudnn-5/). Frontends should emit multi-layer RNNs as a series of While operators (with time being the inner looping dimension), with each successive layer consuming the scan_outputs from the previous layer, possibly going through several point-wise operators (e.g. dropout, residual connections, linear layer).

The input/output of subgraph (produced by loop node) matching is based on order instead of name. The implementation will figure out the names based on this order.

Version

This version of the operator has been available since version 16 of the default ONNX operator set.

Other versions of this operator: 1, 11, 13

Attributes

***`body`* : graph (required)**
The graph run each iteration. It has 2+N inputs: (iteration_num, condition, loop carried dependencies...). It has 1+N+K outputs: (condition, loop carried dependencies..., scan_outputs...). Each scan_output is created by concatenating the value of the specified output value at the end of each iteration of the loop. It is an error if the dimensions or data type of these scan_outputs change across loop iterations.

Inputs (2 - ∞)

***`M`* (optional) : I**
A maximum trip-count for the loop specified at runtime. Optional. Pass empty string to skip.

***`cond`* (optional) : B**
A boolean termination condition. Optional. Pass empty string to skip.

***`v_initial`* (variadic, heterogeneous) : V**
The initial values of any loop-carried dependencies (values that change across loop iterations)

Outputs (1 - ∞)

***`v_final_and_scan_outputs`* (variadic, heterogeneous) : V**
Final N loop carried dependency values then K scan_outputs. Scan outputs must be Tensors.

Type Constraints

V : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool), tensor(complex64), tensor(complex128), seq(tensor(uint8)), seq(tensor(uint16)), seq(tensor(uint32)), seq(tensor(uint64)), seq(tensor(int8)), seq(tensor(int16)), seq(tensor(int32)), seq(tensor(int64)), seq(tensor(bfloat16)), seq(tensor(float16)), seq(tensor(float)), seq(tensor(double)), seq(tensor(string)), seq(tensor(bool)), seq(tensor(complex64)), seq(tensor(complex128)), optional(seq(tensor(uint8))), optional(seq(tensor(uint16))), optional(seq(tensor(uint32))), optional(seq(tensor(uint64))), optional(seq(tensor(int8))), optional(seq(tensor(int16))), optional(seq(tensor(int32))), optional(seq(tensor(int64))), optional(seq(tensor(bfloat16))), optional(seq(tensor(float16))), optional(seq(tensor(float))), optional(seq(tensor(double))), optional(seq(tensor(string))), optional(seq(tensor(bool))), optional(seq(tensor(complex64))), optional(seq(tensor(complex128))), optional(tensor(uint8)), optional(tensor(uint16)), optional(tensor(uint32)), optional(tensor(uint64)), optional(tensor(int8)), optional(tensor(int16)), optional(tensor(int32)), optional(tensor(int64)), optional(tensor(bfloat16)), optional(tensor(float16)), optional(tensor(float)), optional(tensor(double)), optional(tensor(string)), optional(tensor(bool)), optional(tensor(complex64)), optional(tensor(complex128))

All Tensor, Sequence(Tensor), Optional(Tensor), and Optional(Sequence(Tensor)) types

I : tensor(int64)

tensor of int64, which should be a scalar.

B : tensor(bool)

tensor of bool, which should be a scalar.

Examples

▸ loop_11

▸ loop_13

▸ loop_16_none

LpNormalization

Given a matrix, apply Lp-normalization along the provided axis.