ONNX Sparse Tensor Design

V2: Updated July 1, 2019

## Motivation

There are several situations where tensors are sparse, and we can achieve better efficiency during inference by making use of sparse tensor representations and operations on sparse tensors. One common scenario is the use of feature vectors that are sparse. While typical usage involves either sparse vectors or sparse matrices (that is, tensors of rank 1 or 2), batching could add one extra dimension to this.

## Type System

We introduce a SparseTensor type analogous to the existing Tensor type (which is interpreted as a “Dense Tensor”). One key implication of this design is that operator signatures will have to explicitly indicate when they will accept sparse-tensors as input and when they will produce sparse-tensors as output. (Thus, to apply an operator that requires a Tensor input to a SparseTensor value, there needs to be an explicit conversion from SparseTensor to Tensor using conversion operators.)

Proposal: Promote the existing SparseTensor type from ONNXML to ONNX.

|  |
| --- |
| message TypeProto {  …  message SparseTensor {  // This field MUST NOT have the value of UNDEFINED  // This field MUST have a valid TensorProto.DataType value  // This field MUST be present for this version of the IR.  optional int32 elem\_type = 1;  optional TensorShapeProto shape = 2;  };  oneof value {  …  SparseTensor sparse\_tensor\_type = 8;  …  }  } |

## Representation

ONNX does not impose any specific representation for sparse tensors, and the choice of the representation is up to the runtime. However, it is still useful to consider the variety of representations for sparse tensors.

Sparse vectors have an obvious representation as a pair of vectors (arrays) of (non-zero) values and indices. Sparse matrices have multiple possible representations, such as CSR (Compressed Sparse Row), CSC (Compressed Sparse Column), and COO (Coordinate format). The COO format is the direct generalization of the sparse-vector representation: we have an array of the non-zero values and a (2-dimensional) array of the indices of these non-zero values. The CSR format compresses the representation of indices further, while providing direct (constant-time) access to each row by utilizing an array of column-indices of all non-zero values, and an array of offsets (into the other two arrays) for each row. Thus, the CSR format is typically more compact than the COO format, unless the matrix has many empty rows (rows with all-zero entries).

For tensors of rank more than 2, the number of possible representations increases even further. In principle, it is possible to choose a dense-or-sparse representation for each dimension. A useful sub-class of these representations is obtained if we impose the restriction that the dense-dimensions are an initial prefix of the list of all dimensions, while the rest are all sparse-dimensions.

Representing the batch-dimension as dense has certain advantages: it enables constant-time access to the data corresponding to a given batch, which is convenient when exploiting batch-level parallelism. On the other hand, the COO format enables uniform access to all values, allowing partitioning that is not based on batches.

## Sparse Tensors As Attributes

We extend AttributeProto to support a representation of sparse tensors as attributes. We permit the use of either the COO format (the index of every value is specified as a tuple ) or linearized index values. (A linearized index of an element is the index that element has if the tensor is reshaped into a 1-dimensional tensor: an index tuple is converted into a single index value where size denotes the value . )

Proposal: Extend AttributeProto as below:

message SparseTensorProto {

// The sequence of non-default values are encoded as a tensor of shape [NNZ].

// The default-value is zero/empty-string for numeric/string tensors.

optional TensorProto values = 1;

// The indices of the non-default values may be stored in one of two formats.

// (a) Indices can be a tensor of shape [NNZ, rank] with the [i,j]-th value

// corresponding to the j-th index of the i-th value (in the values tensor).

// (b) Indices can be a tensor of shape [NNZ], in which case the i-th value

// must be the linearized-index of the i-th value (in the values tensor).

// The linearized-index can be converted into an index tuple (k\_1,...,k\_rank)

// using the shape provided below.

// The indices must appear in ascending order without duplication.

// In the first format, the ordering is lexicographic-ordering:

// e.g., index-value [1,4] must appear before [2,1]

optional TensorProto indices = 2;

// The shape of the underlying dense-tensor: [dim\_1, dim\_2, ... dim\_rank]

repeated int64 dims = 3;

}

message AttributeProto {

…

…

optional SparseTensorProto sparse\_tensor;

repeated SparseTensorProto sparse\_tensors;

}

Note that indices is required to be an int64 tensor in the above representation. We considered the alternative of making indices a “repeated int64” field, but the current TensorProto definition gives us some flexibility in representing very large sequences of values.

In addition, generation of SparseTensors from other formats and representations can be flexibly supported via ops as illustrated below.

## Ops

The specification of ops will indicate which inputs/outputs may be sparse and/or dense. For ops with more than one argument, whether the output is sparse or dense may depend on the combination of input-types. For example, “Mul(Sparse, Dense)” can return a SparseTensor, while “Add(Sparse,Dense)” can return a (dense) Tensor, while “Add(Sparse, Sparse)” can return a SparseTensor. The specification of ops will indicate the behavior (through the TypeAndShapeInferenceFunction).

**Example**: A simplified op-schema for MatMul appears below. We will discuss some other issues relating to this later (in the “Operator Schema Type Specification Language” section).

ONNX\_OPERATOR\_SET\_SCHEMA(

MatMul,

11,

OpSchema()

.Input(0, "A", "N-dimensional matrix A", "T")

.Input(1, "B", "N-dimensional matrix B", "T")

.Output(0, "Y", "Matrix multiply results from A \* B", "T")

.TypeConstraint(

"T",

{"tensor(float16)", “sparsetensor(float16)”, ... },

"Constrain input and output types to float/int tensors.")

...

.TypeAndShapeInferenceFunction([](InferenceContext& ctx) {

// Determine if output is sparse or dense based on input types

}));

For some ops, it may be desirable to let the user specify the desired output format (as sparse or dense). We will use attributes (recommendation: a boolean attribute “sparseoutput”). For example, we will update the existing ops such as Binarizer or TfidfVectorizer by adding a Boolean attribute “sparseoutput” to indicate that the output should be produced as a SparseTensor.

The example below illustrate this.

The following is a categorization of the various ops that can be extended to support or exploit sparse tensors.

ONNX\_OPERATOR\_SET\_SCHEMA(

TfIdfVectorizer,

11,

OpSchema()

.Input(0, "X", "Input for n-gram extraction", "T")

.Output(0, "Y", "Ngram results", "T1")

...

.TypeConstraint("T1", {"tensor(float)", “sparsetensor(float)”},

"1-D (sparse or dense) tensor of floats")

.Attr("sparseoutput",

"An optional flag. The default value of 0 indicates that the output should be produced as a dense tensor and a value of 1 indicates that the output should be produced as a sparse tensor.",

AttributeProto::INT,

false)

...

.TypeAndShapeInferenceFunction(

// Infer output type based on value of “sparseoutput”

)

* Matrix multiplication variants (e.g., MatMul and Gemm)
* Unary element-wise ops (e.g., Abs and Neg)
* Binary element-wise ops (e.g., Add and Mul).
* Reduction ops (such as ReduceSum, ReduceSumSquare, ReduceL1, ReduceL2, ReduceLogSum)
* Generators and constructors (ONNXML): Ops such as Binarizer, DictVectorizer, OneHotEncoder, and TfidfVectorizer can be extended to generate a sparse representation of their outputs.
* Classifiers and Regressors (ONNXML), such as LinearClassifier and LinearRegressor, can exploit sparsity in their inputs.
* Converters: We will add new ops to convert between different representations, such as from a SparseTensor to Tensor.

Further, generation of SparseTensors from other representations will be supported by new ops, as illustrated below:

|  |
| --- |
| ONNX\_OPERATOR\_SCHEMA(SparseTensor)  .SinceVersion(…)  .SetDoc(R"DOC("Create a sparse tensor"))  .Input(0, "shape", "Shape of underlying dense tensor", "T1")  .Input(1, "indices", "Indices of non-zero values", "T2")  .Input(2, "values", "Non-zero values in the tensor", "T3")  .Output(0, "output", "A sparse tensor", "T4")  .TypeConstraint(“T1”, {“tensor(int64)”}, “Shape must be an int64 tensor”)  .TypeConstraint(“T2”, {“tensor(int64)”}, “Indices must be an int64 tensor”)  .TypeConstraint(“T3”, OpSchema::all\_tensor\_types(), “Any dense tensor type”)  .TypeConstraint(“T4”, OpSchema::all\_sparse\_tensor\_types(), “Any sparse tensor type”)  .TypeAndShapeInferenceFunction([](InferenceContext& ctx) {  // Shape and type inference  }); |

## Shape Inference For SparseTensors

As before, we will use shape-inference methods to enable a more efficient implementation in the runtime. In the case of sparse-tensors, the shape-inference methods can be utilized to infer the “dense-shape” (this is the shape of the tensor that will be produced if we convert the sparse-tensor to a dense-tensor). In principle, we can also try to compute (a symbolic) NNZ (the number of non-zero values in the sparse-tensor). However, with the exception of unary element-wise ops, it is unlikely we can compute a useful value for NNZ. So, currently, we do not propose to infer NNZ.

## Operator Schema Type Specification Language

ONNX uses a sub-language for specifying the type-signature of operators. In its current form, it makes it somewhat awkward and verbose for specifying the type-signature of an op (say MatMul) when it supports both sparse and dense tensors. In particular, we would like MatMul to permit either of its two operands to be sparse or dense, but we would like to constraint them to be of the same numeric type (e.g., float or double). Ideally, we should extend the op schema type specification language so that we can write the following:

ONNX\_OPERATOR\_SET\_SCHEMA(

MatMul,

11,

OpSchema()

.Input(0, "A", "N-dimensional matrix A", {“sparse(T)”, “dense(T)”})

.Input(1, "B", "N-dimensional matrix B", {“sparse(T)”, “dense(T)”})

.Output(0, "Y", "Matrix multiply results from A \* B", {“sparse(T)”, “dense(T)”})

.TypeConstraint(

"T",

{"float16", “float16”, “double”},

"Constrain input and output types to float tensors.")

...

.TypeAndShapeInferenceFunction([](InferenceContext& ctx) {

// Determine if output is sparse or dense based on input types

}));

However, this extension requires some care to ensure compatibility, because of the existing logic for handling scalar types.

## Design Choices

* We propose to extend existing ops (such as MatMul or Add) to support sparse-tensor types (instead of, for example, introducing new ops such as “SparseMatMul” or “SparseAdd”). This means that the type-and-shape-inference functions for such “overloaded” ops will be slightly more complex, but this seems preferable.
* Sparse-tensors have a default value which is zero for numeric types and empty-string for string tensors. While this can be generalized to support other default values, we don’t propose to do this currently. Supporting other non-default values enables efficient implementations of some ops: e.g., applying the “Cos” operator to a sparse-tensor input. Since Cos(0) is 1, the output can be represented as a sparse-tensor if we can set its default value to be 1. Otherwise, it is better to go with a dense-tensor representation. However, a default value other than zero will complicate the implementation of other ops (such as “Add” or “Mul”). So, it does not seem justified to add this complexity at this stage.