Implementing New Operators for MXNet Design Proposal

Link to dev list discussion

TODO

Feature Shepherd

Anirudh Subramanian

Problem

Multiple Apache MXNet user studies reveal that despite its performance benefits, users still find the alternatives as a better choice for an open-source deep learning framework. At the heart of all these issues, **Usability** seems to be the main cause of concern voiced by the users. As a testimony, the number of open issues continues to hover around the 800 mark since few months. While the number has not risen, it only goes to show that despite the bug bashes, issue fix sprints the approach for solving issues isn't sustainable. Moreover, the types of issues encompass a wide array of domains - including but not restricted to *kvstore*, *operators*, *io* (input output), *profiler*, *imperative*, et al. Realizing that the main logic behind user-facing APIs (front-end) resides in the operators (back-end), I decided to focus on *Operator Improvements*. Hence, with a target of taking down user-critical issues, I shortlisted these - *RandInt, Debug, Hardmax* and *Constant Initializer support for NDArray*.

RandInt Operator

Use Case

A certain MXNet user was building a recommendation model. They used Tensorflow and MXNet to ensure the model performed as expected and performance regressions, if any, were identified. During one such test, it was found that the MXNet's version of Recommendation model saw a drop in its accuracy from 60% to 30%. On the contrary, its equivalent Tensorflow version was performing as per expectation.

The model build recommendations for a user based on past data (previous samples). Each batch consisted of positive and negative (randomly sampled) samples. The embeddings corresponding to both these samples were updated in every batch. It so happened, that the last sample was present only in positive samples and not present in negative samples. Moreover, this bias was relevant to all the users. This was due to incorrect sampling as described here:

> After carefully checking for the numerical stability of random uniform, it appears that it is always possible for the result to be equal to MAX_VAL, which would result in invalid embedding. The solution is to simply clip the output to MAX_VAL-1 after casting it to integer:

```
neg_samples = mx.sym.cast(mx.sym.clip(mx.sym.random_uniform(
low=0, high=opts['MAX_VAL'], shape=(opts['N_NEG_SAMPLES'],)), 0, opts['MAX_VAL'] - 1),
```

> However overall all these solutions have an upper limit of 16.6e6 on MAX_VAL value beyond which there is no one to one mapping between integer and float values.

Conclusion - Upon root cause analysis of the issue, it was found that the unavailability of an out-of-the-box randint operator caused the team to write their own makeshift version of randint. That makeshift random integer sampler was implemented incorrectly. Hence it is necessary to write a specific randint function for supporting such use cases.

However, as mentioned above, overall user experience suffers from

1. Incomplete API that does not address user needs, and requires users to add repeatable custom code
2. Incomplete and hence inaccurate results since all these solutions have an upper limit of 16.6e6 on `num_playlists` value beyond which there is no one to one mapping between integer and float values.

Proposed Approach

To implement the randint function, following steps would be taken

1. Front-end API - Python wrapper to call the `randint` operator in order to put it in the correct `mx.nd.random` namespace and to maintain the consistency.
2. Backend (C/C++) implemented as rand() and rand_int64() functions in `random_generator.h` CPU implementation uses MT19937 random number generator (RNG) while GPU uses Philox. CPython, the reference implementation of Python written in C and Python uses MT19937 as RNG.

It was ensured to include support for `int32` and `int64` for CPU as well as GPU.

Addition of new APIs

```
mx.nd.random.randint (low, high, shape=_Null, dtype=_Null, ctx=None, out=None, **kwargs):
mx.sym.random.randint (low, high, shape=_Null, dtype=_Null, ctx=None, out=None, **kwargs):
```

Performance Considerations

Following table shows the time required for performing randint operation using MXNet's randint vs Numpy's randint. Different permutation of [low,high] values and shapes were tried to justify usage of in-built MXNet `randint` operator.

	Shape	Low,High = [-1000, 1000]	Low,High = [-1000, 1000]	Low,High = [0, 100000]	Low,High = [0, 100000]
1	(i,i)	MXNet	Numpy	MXNet	Numpy
2	1000	0.00146	0.01475	0.35822	0.44711
3	10000	0.00031	1.25686	0.00073	1.26803
4	"32500"	0.00048	16.39452	0.0016	20.00214

Test Plan

Identified 3 types of tests that have been tested on both CPU and GPU:

1. verifying if the distribution is accurate
    a. leverage the existing `verify_generator()`
2. verifying for extreme values
    a. If it is able to handle the large `int32` and `int64` values
3. checking if the upper and lower bounds work

4. checking for permissible and non-permissible dtypes

Alternative Approaches

Used `std::uniform_int_distribution` and it works too. However, was replaced by `std::mt19937::operator()()` function to ensure consistency (Refer - https://github.com/apache/incubator-mxnet/pull/12749/commits/e7e622c25f50ad0588ee9107e903279aea102ed3)

Future Scope

Numpy supports 35 different type of random functions vs 9 of MXNet. In the future, we could consider extending support to the rest of the 26 random functions. This would further enhance the operator coverage and aid to the completeness of MXNet operators.

Milestones

Implementation can be found in the PR - https://github.com/apache/incubator-mxnet/pull/12749

Debug Operators

Use Cases

These are the two types of use cases where Debug operators find themselves relevant and useful :-

1. General Sanity Check (error handling and prevention)
2. Implementing Half-precision floating-point format (FP16) Dynamic Loss Scaling (DLS)
    a. Without FP16, most networks won't be trained with FP16. E.g. Transformer, ConvSeq2Seq

Here's a snapshot of the steps needed to choose a scaling factor (DLS)

```
1. Maintain a master copy of weights in FP32.
2. Initialize 'S' to a large value.
3. For each iteration:
    a. Make an FP16 copy of the weights.
    b. Forward propagation (FP16 weights and activations).
    c. Multiply the resulting loss with the scaling factor 'S'.
    d. Backward propagation (FP16 weights, activations, and their gradients).
    e. If there is an Inf or NaN in weight gradients:
        i. Reduce S.
        ii. Skip the weight update and move to the next iteration.
    f. Multiple the weight gradient with 1/'S'
    g. Complete the weight update (including gradient clipping, etc.).
    h. If there hasn't been an Inf or NaN in the last N iterations, increase 'S'.
```

In the above pseudo-code, step ***3.e.*** verifies if the weight gradients contain `Inf` or `NaN` value. At such instances, an out-of-the-box debug operator that supports large multi-dimensional NDArrays would work wonders (in terms of speed, performance). Verifying if the gradients have absurd values is pretty common use case that calls for having debug operators supported for NDArray in MXNet (instead of relying on corresponding slower Numpy functions).

Addition of new APIs

`mx.nd.contrib.isfinite`
`mx.nd.contrib.isinf`
`mx.nd.contrib.isnan`

Backward Compatibility

This is the first version of this feature and therefore there is no backward compatibility concern. It also does not impact the existing work flow in MXNet.

Performance Considerations

Since we are using NDArray specific operators, it should be faster than converting to Numpy and then using the Numpy equivalents.

	Shape	Load Time (NDArray)	isfinite	isfinite	is_inf	is_inf	is_nan	is_nan
1			MXNet	Numpy	MXNet	Numpy	MXNet	Numpy
2	1000	0.03021	0.0043	0.00083	0.08217	0.01037	0.03103	0.00971
3	10000	3.97593	0.00099	0.80442	0.00122	0.42341	0.00115	0.41738
4	32500	121.64969	0.13902	78.64407	0.01056	76.96974	0.01172	60.25667

Test Plan

```
Create a random dimension, random shape NDArray.
Ensure it has extreme values - np.inf, -np.inf (np.NINF), np.nan
Assert if the output of NDArray Debug ops is equivalent to corresponding Numpy functic
```

Future Scope

- Create seperate `mx.nd.debug`
- Incorporate other debug operators (`assert`, `print`, `verify_tensor_all_finite`)

Milestones

Implementation can be found in the PR -https://github.com/apache/incubator-mxnet/pull/12967

Hardmax operator

Use Case

While ensuring consistency of operators supported by ONNX, it was found that currently MXNet doesn't support a few operators out-of-the-box. Case in point - `Hardmax`. It involves use of a convoluted way involving `reshape` and `argmax` for achieving the same.

```
# Compute Hardmax with axis=1

x = np.random.rand(2,3,4)
xn = mx.nd.array(x)
```

```
xn_r = mx.nd.reshape(xn, shape=(2,12))
xn_e = mx.nd.eye(xn_r.shape[1], dtype=x.dtype)[mx.nd.argmax(xn_r, axis=1)]

hardmax_output = mx.nd.reshape(xn_e, shape=xn.shape)

print(hardmax_output)
```

Direct hardmax implementation would be more convenient and useful for users who would want to build their networks with MXNet as opposed to just importing from ONNX. Hardmax finds its application in Natural Language Processing and Reinforcement Learning. Few examples :-

1. Language model [5]
2. Movie dialogue generation [6]
3. Generative text models [7]

Hardmax is an ONNX operator unsupported in MXNet. MXNet needs to support ONNX operators for compatibility reasons. By including this operator, we are making Apache MXNet as a package more complete and holistic.

Design Considerations

In order to implement Hardmax, it's important to understand the distinction between Softmax, Max, Argmax and Hardmax At times they wrongly get used interchangeably.

Let's see how these operators behave differently, for the same input

```
input_array = mx.nd.array([[ 0., 1., 2.],[ 3., 4., 5.]])
```

Softmax or **normalized exponential function -** generalization of the logistic function that "squashes" a *K*-dimensional vector of arbitrary real values to a *K*-dimensional vector of real values, where each entry is in the range (0, 1), and all the entries add up to 1.

```
>>> mx.nd.SoftmaxActivation(input_array).asnumpy()
array([[0.09003057, 0.24472848, 0.66524094],
[0.09003057, 0.24472848, 0.66524094]], dtype=float32)
```

Argmax - Returns indices of the maximum values along an axis

```
>>> np.argmax(input_array.asnumpy())
5
>>> np.argmax(input_array.asnumpy(),axis=0)
array([1, 1, 1])
>>> np.argmax(input_array.asnumpy(),axis=1)
array([2, 2])
```

Numpy NDarray Max (numpy.ndarray.max) - Return the maximum along a given axis.

```
>> np.max(input_array.asnumpy(),axis=0)
array([3., 4., 5.])
>>> np.max(input_array.asnumpy(),axis=1)
array([2., 5.])
```

However, ideally, we want Hardmax to behave like this

```
>>> mx.nd.contrib.hardmax(xn)

[[0. 0. 1.]
[0. 0. 1.]]
<NDArray 3x2 @cpu(0)>
```

Addition of new APIs

`mx.nd.contrib.hardmax(x)`

Backward Compatibility

This is the first version of this feature and therefore there is no backward compatibility concern. It also does not impact the existing work flow in MXNet.

Test Plan

Since there is no numpy implementation for hardmax, one way to test the function is compute it on paper and verify that for a given input, does it match.

Technical Challenges

- Compelling Usecases / Examples
    - Hardmax offers little use from Backpropagation standpoint since it is hard (non-differentiable) i.e. gradients can't be computed
    - However, one advantage it has over others is the performance/speed apart from being one-hot encoded
    - So it was tough to find use of hardmax in literature/production

Milestones

Implementation can be found in the PR - https://github.com/apache/incubator-mxnet/pull/13083

References

1. CPython - https://github.com/python/cpython/blob/b8e689a6e8134e88f857a55e50b6a4977967e385/Lib/random.py#L33
2. Confusing argument order for random.randint in numpy/numpy Github repo - https://github.com/numpy/numpy/issues/9573
3. JIRA ticket - Hardmax feature request - https://issues.apache.org/jira/browse/MXNET-376
4. Choosing a scaling Factor (Nvidia) - DLS - https://docs.nvidia.com/deeplearning/sdk/mixed-precision-training/index.html#scalefactor
5. Hardmax Usecase - Language Model - https://github.com/v-shmyhlo/machine-learning-playground/blob/432475235169de00b86f786fd0f9ee1e6b7b5685/rnn.ipynb
6. Hardmax Usecase - Movie dialogue generation - https://github.com/vineetjohn/movie-dialogue-generation/blob/60d943632c5459aae28c2b5e1073a2801b9fa127/movie-dialogue-generation-tensorflow.ipynb

7. Hardmax Usecase - Generative text model - https://github.com/vineetjohn/tf-generative-