Old OI Design doc

Operator Improvement Design Doc

Link to dev list discussion

TODO

Feature Shepherd

Chaitanya Bapat

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* during my 15-month long Fall internship. With a target of taking down user-critical issues, I shortlisted these - *RandInt, Debug, Hardmax* and *Constant Initializer support for NDArray*.

Main premise behind selecting these 4 operators was :-

- Criticality and business need
- Operators that need to be Efficient, Easy-to-use for certain models
- Opportunity for me personally to get an end-to-end (backend as well as front-end) exposure of implementing and applying models using Apache MXNet

RandInt Operator

Use Case

Amazon Music, an internal user of MXNet, was building a Music Recommendation model. It used Tensorflow and MXNet to ensure the model performed as expected and performance regressions, if any, were found. 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. Following which, a ticket was file (TT0156262539 [1]) by the Amazon Music team. The recommendation model helps build a playlist for a user based on previous songs (samples). In every batch, we have positive and negative (randomly sampled) samples. The embeddings corresponding to both these samples are updated in every batch. It so happened, that the last sample was present only in positive samples and not present in negative samples. This bias is relevant to all the users and got amplified by having fewer playlists compared to other entities. This was due to incorrect sampling as described in the ticket:

> The last playlist, since it is not often negative sampled will often appear in the recommendations, so it'll harm the metrics since it's not a very popular one.

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

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

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

Conclusion - Upon root cause analysis of the TT, 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.

User Experience

For recommending Music to users, sampling had to be done for positive as well as negative data.

- Tensorflow negative sampling
  - Use `tf.nn.uniform_candidate_sampler`
- Corresponding MXNet negative sampling
  - `negative_samples = random.randint(0, num_playlists)`
    - Ideal scenario (but doesn't exist currently)
  - `negative_samples = random.uniform(0, num_playlists- 1)`
    - Makeshift version which uses NDArray `random.uniform`
    - Issue - https://tt.amazon.com/0156262539
      - [ ]
        ```
        neg_samples = mx.sym.cast(mx.sym.clip(mx.sym.random_uniform(
            low=0, high=opts['N_TRACK'], shape=(opts['N_NEG_SAMPLES'],)), 0,
        opts['N_TRACK'] - 1), dtype='int32')
        ```

However, as mentioned above, overall user experience suffers from

1. Shaky UX (need to write custom code for a lack of simple function)
2. Incomplete and hence inaccurate reults
   a. all these solutions have an upper limit of 16.6e6 on num_playlistsvalue 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)
   a. Symbol as well NDArray
      i. simply call the existing `random_helper` function
2. Backend (C/C++)
   a. Random number generator (RNG) specific code - `random_generator.h`
   b. Operator specific
      i. Defining the operator in `sample_op.h`

ii. Implementing CPU and GPU specific code in `*.cc` and `*.cu` respectively

Support for `int32` and `int64` for CPU as well as GPU.

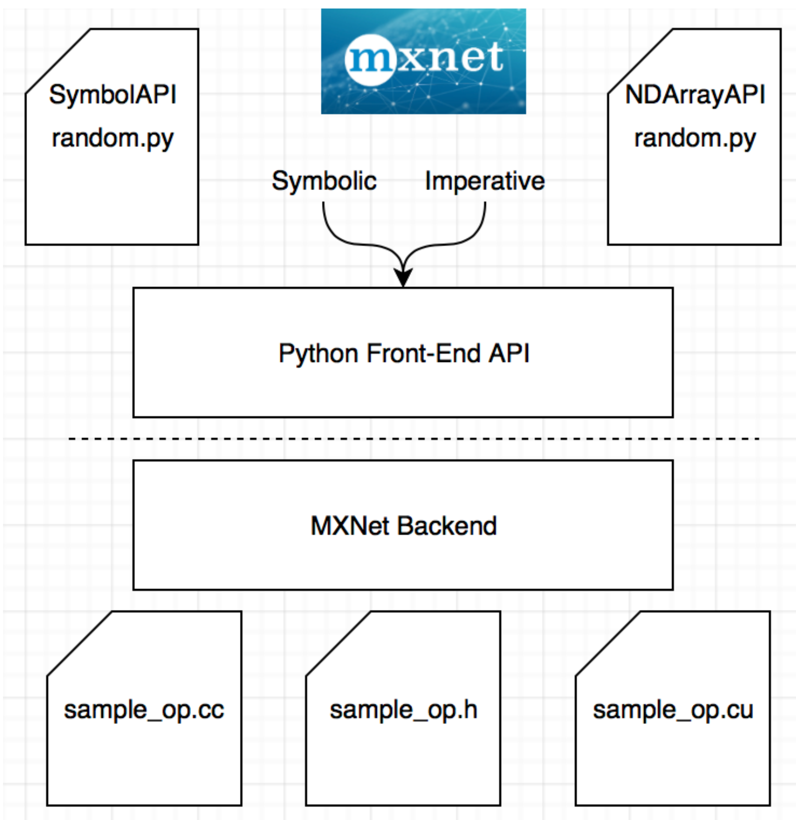
Known Difficulties

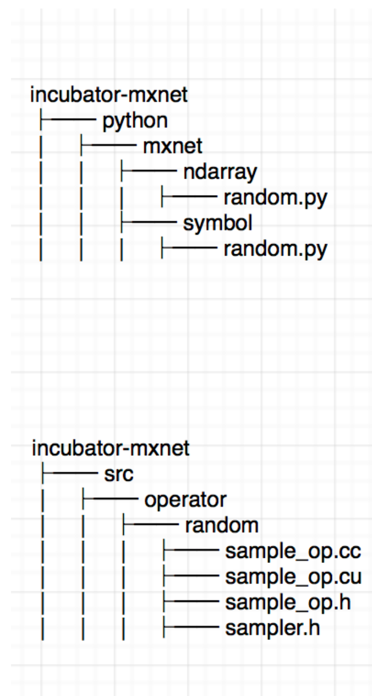
- `int64` support for GPU

In, `random_generator.h` - we need to extend support for `int64`.
However, since the existing `rand` function returns `int32_t`, I'll add separate function `rand_int64()`. In this way, we don't break the existing codebase.
But, *Curand (NVidia CUDA toolkit's random number generator library)* doesn't support int64. Hence, following workaround will just generate two random numbers and put them together:

```
static_cast<int64_t>(engine_->operator()() << 31) + engine_->operator()();
```

Class Design





Design Considerations

- **Integer support for 9 MXNet random functions vs Separate randint function**

In order to tackle the randint problem, we had two options to choose from:

1. Include support for all 9 fxns in MXNet with integer
2. Similar to *Numpy,* have a seperate randint function (basically discrete uniform distribution)

However, upon discussion, decision was made to ensure **consistency** with Numpy, and to continue only floating point support for existing 9 random functions. As a result, I worked on the additional randint function (a.k.a. “discrete” uniform distribution).

- **`std::mt19937.rand()` vs `std::uniform_int_distribution()`**

RandInt is basically discretized form of uniform distribution. Currently, there exists both CPU and GPU specific implementations of `mx.nd.random.uniform`. It uses the `std:uniform_real_distribution` and `std:uniform_int_distribution`
So one way would be to implement `mx.nd.random.randint` based of off the predefined std functions.

However, the random number generator class uses 2 different types of RNG functions

1. Mersene Twister 19937 (MT19937) for CPU
2. Philox for GPU

Moreover, funcrtion `rand()` uses RNG→operator()() (it generates a pseudorandom number)
It is not known whether this pseudorandom number follows what distribution
Hence, according to this discussion, it was decided to choose existing `rand()` function instead of uniform_int_distribution()
Why? - it will confuse the user whether or not to use rand and what would be its return type.

Result - use `rand()`

- Confusing definition of low and high of numpy

Numpy needs 1 value
According to documentation, low is required, while High is optional

`numpy.random.randint(`*`low`*`, `*`high=None`*`, `*`size=None`*`, `*`dtype='l'`*`)`

However, upon closer look it turns out that conceptually it is the other way round where low is implied as 0 and high is required.
Yet another instance of funny, contrived documentation -

> If *high* is None (the default), then results are from [0, *low*).

Refer - Confusing argument order for random.randint in numpy/numpy Github repo

Result - ensure both low and high are mandatory (instead of using numpy's confusing definition)

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):
```

Backward compatibility

Since it belongs to the class of random operators, it has been written in a way that's consistent with them. However, being a new operator itself, there are no issues as far as backward compatibility is concerned.

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]	C	Low,High = [0, 100000]	E
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

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

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)

Technical Challenges

1. Reproducing the CI test failures
2. Found definition mismatch of randn (symbolic api)
   a. Closed PR since API Breaking change (to be reopened for MXNet 2.0) #12775

Future Scope

Numpy supports 35 different type of random functions vs 9 of MXNet

	Numpy	-	MXNet
1			
2	beta		exponential
3	binomial		gamma
4	chisquare		generalized_negative_binomial
5	dirichlet		negative_binomial
6	exponential		normal
7	f		poisson
8	gamma		uniform
9	geometric		multinomial
10	gumbel		shuffle
11	hypergeometric		
12	laplace		
13	logistic		
14	lognormal		
15	logseries		
16	multinomial		
17	multivariate_normal		
18	negative_binomial		
19	noncentral_chisquare(df, nonc[, size])		
20	noncentral_f(dfnum, dfden, nonc[, size])		
21	normal([loc, scale, size])		
22	pareto(a[, size])		
23	poisson([lam, size])		
24	power(a[, size])		
25	rayleigh([scale, size])		
26	standard_cauchy([size])		
27	standard_exponential([size])		
28	standard_gamma(shape[, size])		
29	standard_normal([size])		
30	standard_t(df[, size])		
31	triangular(left, mode, right[, size])		
32	uniform([low, high, size])		
33	vonmises(mu, kappa[, size])		

	size])		
34	`wald`(mean, scale[, size])		
35	`weibull`(a[, size])		
36	`zipf`(a[, size])		

Status - Merged

https://github.com/apache/incubator-mxnet/pull/12749

Debug Operators

Use cases

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).

User Experience

The current user experience is as follows

```
>>> import mxnet as mx
>>> import numpy as np
>>> a = mx.nd.zeros(1)
>>> print(a)
[0.]
<NDArray 1 @cpu(0)>
```

```
>>> print(np.isinf(a.asnumpy()))
[False]
>>> b = mx.nd.array(np.inf)
>>> print(b)

[inf]
<NDArray 1 @cpu(0)
>>>> print(np.isnan(b.asnumpy()))
[False]
```

Basically, user has to

1. convert an NDArray into corresponding numpy array
2. Use the numpy's debug functions

Problem with existing user experience is two-fold
a. Shaky/Broken user experience
b. Slower

In **Tensorflow**, on the other hand, there are 8 debug operators supported out of the box.

```
import tensorflow as tf
import numpy as np
with tf.Session() as sess:
    print(sess.run(tf.is_finite(tf.constant(4.0))))
    print(sess.run(tf.is_inf(tf.constant([4.0, np.inf, np.NINF]))))
    print(sess.run(tf.is_nan(tf.constant(4.0))))
    print(sess.run(tf.verify_tensor_all_finite(tf.constant(4.0))))
    print(sess.run(tf.check_numerics(tf.constant(4.0))))
    print(sess.run(tf.add_check_numeric_ops(tf.constant(4.0))))
    print(sess.run(tf.Asset(tf.constant(4.0))))
    print(sess.run(tf.Print(tf.constant(4.0),[tf.constant(4.0)],"Yo")))
```

Proposed Approach

Debug operator was identified to be needed only for NDArray API.
Reason - NDArray is the basis for computation and hence ultimately one would need to debug for values such as NaN, infinity, etc for an NDArray.

Hence identified that `/python/mxnet/ndarray/contrib.py` is the right place for incorporating the 3 debug operators.
Reason behind selecting the 3 debug operators is because the broad use and relevant user feedback.

```
isinf - To check for infinite, all we had to do was compare absolute value of input wi
isnan - To check for NotANumber, use the property of NaN that NaN == NaN returns false
isfinite - To check for finite value, (in that order)
Step 1 - Ensure that we check for NaN
Step 2 - Check for np.inf
```

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	D	is_inf	F	is_nan	H
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`)

Status - Merged

https://github.com/apache/incubator-mxnet/pull/12967

Hardmax operator

Problem

While ensuring consistency of operators supported by ONNX, it was found 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. 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.

User Experience

The current user experience is as follows

```
# 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)
```

Refer -

Use cases

Hardmax finds its application in Natural Language Processing and Reinforcement Learning

1. Language model
2. Movie dialogue generation
3. Generative text models

Proposed Approach

Similar to Hardmax implementation of tensorflow, I would take an argmax upon one_hot_encoding the data based on the given shape. Front-end approach.

Back-end - For the time being, we can circumvent actual implementation of hardmax on backend by using CustomOp Basic steps would be

- Subclass the CustomOp class

```
class Hardmax(mx.operator.CustomOp):
    def forward(self, is_train, req, in_data, out_data, aux):
        <forward computation>
    def backward(self, req, out_grad, in_data, out_data, in_grad, aux):
        <backward computation>
```

- Registering the new op

```
@mx.operator.register("hardmax")class HardmaxProp(mx.operator.CustomOpProp):
```

- Initializations

```
def __init__(self):
    super(HardmaxProp, self).__init__(need_top_grad=False)
```

- Declaration of Input, Output

```
def list_arguments(self):
    return ['data', 'label']
```

```
def list_outputs(self):
    return ['output']
```

- Shape Inference

```
def infer_shape(self, in_shape):
    data_shape = in_shape[0]
    label_shape = (in_shape[0][0],)
    output_shape = in_shape[0]
    return [data_shape, label_shape], [output_shape], []
```

- Type Inference

```
def infer_type(self, in_type):
    dtype = in_type[0]
    return [dtype, dtype], [dtype], []
```

- Backend Function Call

```
def create_operator(self, ctx, shapes, dtypes):
    return Hardmax()
```

Design Considerations

Understanding the distinction between Softmax, Argmax and Hardmax (since at times they wrongly get used interchangeably). However, they all mean different things.

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),[a] and all the entries add up to 1.

```
>>> input_array = mx.nd.array([[3., 0.5, -0.5, 2., 7.],
>>>                            [2., -.4, 7.,   3., 0.2]])
>>> softmax_act = mx.nd.SoftmaxActivation(input_array)
>>> print softmax_act.asnumpy()
[[  1.78322066e-02   1.46375655e-03   5.38485940e-04   6.56010211e-03   9.73605454e-01
 [  6.56221947e-03   5.95310994e-04   9.73919690e-01   1.78379621e-02   1.08472735e-03
```

Argmax - Returns indices of the maximum values along an axis

```
x = [[ 0.,  1.,  2.],[ 3.,  4.,  5.]]
// argmax along axis 0argmax(x, axis=0) = [ 1.,  1.,  1.]
// argmax along axis 1argmax(x, axis=1) = [ 2.,  2.]
// argmax along axis 1 keeping same dims as an input array
// argmax(x, axis=1, keepdims=True) = [[ 2.],[ 2.]]
```

Hardmax -

```
>>> xn
[[[[2. 3. 4.]
```

```
[1. 2. 3.]]
[[1. 1. 1.]
[1. 2. 3.]]]

[[[4. 5. 6.]
[4. 4. 4.]]
[[1. 1. 1.]
[1. 2. 3.]]]]
<NDArray 2x2x2x3 @cpu(0)>
>>> mx.nd.contrib.hardmax(xn)

[[[[0. 0. 1.]
[0. 0. 1.]]
[[1. 0. 0.]
[0. 0. 1.]]]

[[[0. 0. 1.]
[1. 0. 0.]]
[[1. 0. 0.]
[0. 0. 1.]]]]
<NDArray 2x2x2x3 @cpu(0)>
```

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

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

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

Model

Building a language model
Dataset utilized - Text8
3 layered LSTM-RNN model that utilizes Hardmax as prediction for first layer input.

References

1. Trouble Ticket - Amazon music - https://tt.amazon.com/0156262539
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-model/blob/108a5bf470b292b3b80c76cf01e0c669635f0db9/tf-generative-model.ipynb