MXNet end to end models

- Link to dev list discussion
- Feature Shepherd
- Problem
- Goals/Use cases
- Open Questions
- Proposed Approach
- Addition of New APIs
- Backward compatibility
- Performance Considerations
- Technical Challenges
- Milestones (Development Plan)
  - Phase 1 - Image Classification
  - Phase 2 - Object Detection
  - Phase 3 - Image Segmentation
  - Phase 4 - NLP use cases
  - Other ideas and future work items
- Test Plan
- Alternative Approach - 1
- Cons of this Approach
- Appendix
- FAQ
- Performance Benchmarks
- References

Link to dev list discussion
<TBD>

Feature Shepherd

Sandeep (skm@apache.org)


Problem

Data pre-processing and post-processing are commonly used when training a deep learning model. Technically these data processing is called data transformations. Generally, Data Transformation are applied on train, val, and test dataset. In most cases, data transformations used on validation data during training phase, is applicable in the inference phase. However, MXNet models do not contain information about data transformations, creating a disconnect and barrier for easy movement of models from training to production inference deployment. Below are the problems we are aiming to solve in this work:

1. Input/Output data transformations are not part of MXNet model
2. Input/Output data transformations are currently support CPU only
3. Few Input/Output data transformations are Python specific
4. Not all data transformers are Hybrid Blocks in Gluon, hence, cannot be exported as symbol graph
5. Input/Output data transformations takes single input only (Ex: Normalize takes 3D tensor - 1 image only as input)

NOTE

Other notable problems for production model deployment are listed below. However, these problems are not addressed in this work and will be taken up in the next iteration:

1. Input/Output signature: Saved model missing the information about the input/output descriptions, like name/shape, making the saved model unusable out of the box.
2. File name, multiple files: Managing multiple files representing one model. Knowing epoch number. With this release, MXNet addresses these limitations by providing easy to use end-to-end-model APIs for saving the model along with the input/output data transformations and input/output data descriptions.

Goals/Use cases

1. As a data scientist, from MXNet Gluon, I should be able to concatenate data transformations with the neural networks, and export the end-to-end MXNet model using Gluon export APIs.
2. As a user, I want to be able to load the end-to-end MXNet model and run inference (single/batch) in Gluon (Python), Module (Python), Scala and Java inference APIs. I should not be required to rewrite data transformations and I expect them to be part of the model.
3. As a user, I should be able to run inference on these end-to-end model on CPU or GPU machine.
4. As a user, I should be able to run single or batch request inference with different shapes on these end-to-end model on CPU or GPU machine.
5. As a user, when I run inference with end-to-end models on CPU or GPU machine, I should not lose performance compared to running data transformations separately on CPU followed by inference on CPU/GPU.
Open Questions

1. Can we have list of NDArrays as input? How does it work when bound to module for doing batch inference?
   a. For example, before resize data transformation operator, users may have images of different shapes, so they will not be able to have batch of NDArray (N, c, h, w). Instead, they will have a list of NDArrays (c, h1, w1), (c, h2, w2) and so on. How does this work?
2. Can we have an operator in MXNet that can take a buffer (ex: image) or string (ex: filepath) as input?
   a. If we can achieve this, we can have, Image decode as first node in the graph enabling users to directly feed the raw data and get the output predictions totally simplifying model deployment and inference code.
3. Can we have an operator in MXNet that can output String (ex: class name) or Vector (ex: Bounding box co-ordinates) (Non-NDArray) as output?
   a. If we can achieve this, we can have, post processing transformation operators in the graph, enabling users to get the ready to consume output predictions. For example, a model that can take raw input image as input and gives out class name as output! Totally, hiding framework specificities like NDArray in production deployment inference code.

Proposed Approach

1. Make data transformation operators as any other MXNet operator i.e., they are available via `nd` and `sym` packages.
2. Implement CPU and GPU support for data transformation operators.
3. Support single input (3D Tensor), batch input (4D Tensor) and list input (list of 3D Tensor) as inputs to data transformation operators.
4. Create a new data transformation operator - "list_to_batch". This will enable converting a list of NDArray to a batch data (4D Tensor) before pushing it to neural network for inference.
5. Make all data transformations as Hybrid Block. This enables users to be able to concatenate data transformation blocks to neural network block and export the end to end model.

User experience - Model Training and Export

```python
import mxnet as mx
from mxnet import gluon
from mxnet.gluon.data.vision import transforms

#### ..... Train/Validation dataset anddataloaders and more ..... ####
# A very simple 2 layer network definition for illustration
net = gluon.nn.HybridSequential()
with net.name_scope():
    net.add(gluon.nn.Conv2D(channels=20, kernel_size=5, activation='relu'))
    net.add(gluon.nn.Dense(10))

#### ..... Model Training Part ..... ####
# Now, user wants to export the model
end_to_end_model = gluon.nn.HybridSequential()
with net.name_scope():
    end_to_end_model.add(transforms.ToTensor())
    end_to_end_model.add(transforms.Normalize(mean=(123.675, 116.28, 103.53), std=(58.395, 57.12, 57.375)))
    end_to_end_model.add(transforms.Normalize(mean=(123.675, 116.28, 103.53), std=(58.395, 57.12, 57.375)))
end_to_end_model.hybridize()
inp = nd.random_normal(shape=(1, 224, 224, 3))
end_to_end_model(inp)
end_to_end_model.export("end_to_end_img_classification")

# Generates 2 files - end_to_end_img_classification-symbol.json and end_to_end_img_classification-0000.params
```
Symbolic graph of above exported model looks like below:

```plaintext
Activation sigmoid

FullyConnected 10

Convolution 2x2/1x1, 8

image_normalize

image_to_tensor

data
```

**User experience - Model Inference in Python Module**

```python
import mxnet as mx

# Load the model as usual
sym, args, aux = mx.model.load_checkpoint('end_to_end_img_classification', 0)
mod = mx.mod.Module(symbol=sym, context=ctx, label_names=None)
mod.bind(for_training=False, data_shapes=[('data', (1,224,224,3))],
         label_shapes=mod._label_shapes)
mod.set_params(arg_params, aux_params, allow_missing=True)

# Inference is just reading an image and pushing it to model which has transformation followed by network
img = mx.image.imread(fname)
prob = mod.forward(img)
```

**User experience - Model Inference in Java Inference API**

```java
List<DataDesc> inputDesc = new ArrayList<>();
Shape inputShape = new Shape(new int[]{1, 224, 224, 3});
inputDesc.add(new DataDesc("data", inputShape, DType.Float32(), "NCHW"));
Predictor predictor = new Predictor(inst.modelPathPrefix, inputDesc, context, 0);
NDArray img = Image.imRead(inst.inputImagePath, 1, true);
float[][] result = predictor.predict(new float[][][img.toArray()]);
```

**Addition of New APIs**

There are NO NEW APIs introduced as part of this work in phase 1.
Backward compatibility

1. All API changes and functionality addition is backward compatible. No existing functionality should be broken.
2. No performance impact for existing use cases.

Performance Considerations

1. For training jobs, there is no change in existing behavior, hence we do not expect any performance changes.
2. For single/batch inference (on CPU), since the number of operators (transformations + network) are the same, we do not expect any performance changes.
3. For single/batch inference (on GPU), with addition of transformation operator support on GPU, expect to see increased throughput and reduced latency (TBD Benchmarks and POC in progress)
4. However, if the input data is small, number of data transformations are minimal and inference is running on a GPU, then running all transformations on GPU may slow things down compared to a multi-threaded CPU data transformation operation.

Technical Challenges

1. GPU implementation of data transformation operators.
2. Handling Single, Batch, List inputs for data transformation operators.
3. Handling backward pass (very rarely used if at all) for data transformation operators.

Milestones (Development Plan)

Phase 1 - Image Classification

In Phase 1, we will be targeting image classification use cases with following transformation being commonly used during inference.

exists => Functionality already exists currently
TODO/PR/Done => Tasks of this current work

<table>
<thead>
<tr>
<th>Transformer</th>
<th>Gluon Hybrid Block</th>
<th>CPU Operator</th>
<th>GPU Operator</th>
<th>Single Input (3D)</th>
<th>Batch Input (4D)</th>
<th>List Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Normalize</td>
<td>Exists</td>
<td>Exists</td>
<td>WIP - PR - #13802</td>
<td>Exists</td>
<td>WIP - PR - #13802</td>
<td>TODO</td>
</tr>
<tr>
<td>2 To Tensor</td>
<td>Exists</td>
<td>Exists</td>
<td>WIP - PR - #13837</td>
<td>Exists</td>
<td>WIP - PR - #13837</td>
<td>TODO</td>
</tr>
<tr>
<td>3 Resize</td>
<td>WIP - PR - #13611</td>
<td>Exists</td>
<td>TODO</td>
<td>Exists</td>
<td>WIP - PR</td>
<td>TODO</td>
</tr>
<tr>
<td>4 list_to_batch</td>
<td>WIP</td>
<td>Exists</td>
<td>Needs some update</td>
<td>Exists as concat operator.</td>
<td>Needs some update</td>
<td>TODO</td>
</tr>
<tr>
<td>5 RandomResizedCrop</td>
<td>TODO</td>
<td>Exists</td>
<td>Requires restructuring</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>6 CenterCrop</td>
<td>PR - #13694</td>
<td>Exists</td>
<td>Requires restructuring</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>7 Crop</td>
<td>PR - #13679</td>
<td>PR - #13679</td>
<td>N/A</td>
<td>PR - #13679</td>
<td>N/a</td>
<td>N/A</td>
</tr>
<tr>
<td>8 Java API image decoding</td>
<td><a href="https://github.com/apache/incubator-mxnet/pull/13807">https://github.com/apache/incubator-mxnet/pull/13807</a></td>
<td>By Qing Lan</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9 End to end example - Image classification - Train on Gluon, Inference on Java</td>
<td>(TODO)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 Benchmarks - Above benchmarks with Java Predictor API on CPU, GPU comparing existing numbers with a fused single graph.</td>
<td>(TODO)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 Blogpost and other user documentations</td>
<td>(TODO)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Phase 2 - Object Detection

Data transformation operators specifically for object detection use cases.

Ex: SSDRandomCrop, BoundingBoxFlip

Phase 3 - Image Segmentation

Data transformation operators specifically for image segmentation use cases.
**Phase 4 - NLP use cases**

Data transformation operators specifically for nlp use cases.

**Other ideas and future work items**

1. Fused transformation operators for the common combination.
   a. **ResizeCropNormalize**: Perform fused resize, crop and normalization
   b. **CropMirrorNormalize**: Perform fused cropping, normalization, format conversion (NHWC to NCHW) if desired, and type casting.
   c. **FastResizeCropMirror**: Perform a fused resize, crop, mirror operation. Handles both fixed and random resizing and cropping. Backprojects the desired crop through the resize operation to reduce the amount of work performed.
   d. **RandomResizedCrop**: Perform a crop with randomly chosen area and aspect ratio, then resize it to given size.
   e. **ReshapeCropMirror**: Perform a fused resize, crop, mirror operation. Handles both fixed and random resizing and cropping.

2. Integration with NVIDIA DALI.

3. Integration with other Data Processing engines - RAPIDS and Apache Arrow will have multiple advantages - Non-image based accelerated transformations on columnar data, out of the box support for various stable data loaders to read parquet, protobuf and such data formats, ability for users to use Pandas, Spark and cross framework data processing libraries and feed the data to MXNet.

**Test Plan**

1. Test, able to load models trained in the previous versions of MXNet. (Backward compatible)
2. Test, able to concatenate transformation to the network and export an end-to-end model
3. Test, running inference on the saved end-to-end model with Python Module and Gluon SymbolBlock APIs. On CPU and GPU. Single, Batch and List input inference.
4. Test, running inference on the saved end-to-end model with Java Inference APIs, Scala Inference APIs. On CPU and GPU. Single, Batch and List input inference.
5. Benchmark and verify performance running inference with end-to-end model versus transformations separately on CPU followed by network prediction.
6. All these tests will be added to CI tests.

**Alternative Approach - 1**

Create a new end-to-end model export API. End to end model is not just a sym and param file. It is an archive network graph and params, transformations graph and params, input/output signature and auxiliary resources like Synset file, sample input/output and more. In this approach two main changes:

1. **Extend Hybrid Block export API**: In Gluon Hybrid Block export API, provide additional options for users to specify input/output signatures, and any other graphs (Hybrid blocks) to be exported (Ex: transformations hybrid sequential block). Export API will save the symbol graphs for each of these hybrid blocks tagged with a pre-defined name or user provide name. See an example symbol file below.

2. **Extend model import APIs**: In Gluon/module/Java/Scala inference APIs, we create new APIs for importing end to end models.

See below for code samples:
Export APIs for end to end models from Gluon

Export the HybridBlock as MXNet Model. You can include additional graphs such data transformations, signature as part of the model.

Parameters
---------

- **path**: str
  - Path to save the model. Two files `path-symbol.json` and `path-xxxx.params` will be created, where xxxx is the 4 digits epoch number.

- **epoch**: int
  - Epoch number of saved model.

- **signature**: dict of tuples
  - Input/Output signature, i.e., name and shape, for the model.

- **auxiliary_graphs**: dict of Hybrid Blocks
  - Additional helper graphs to be saved as part of the model.
  - Key -> Name of the graph. Use predefined constant names or custom name.
  - Value -> Hybrid Block representing the graph.

```python
net.export(path="my_model", epoch=0,
           signature = {constants.INPUT_DESC : ('data', Shape(1, 3, 224, 224)),
                        constants.OUTPUT_DESC : ('softmax', shape(1, 10))},
           auxiliary_graphs =
                        {constants.TRAIN_INPUT_TRANSFORMS = my_train_transforms,
                         constants.VAL_INPUT_TRANSFORMS = my_val_transforms,
                         constants.PRED_INPUT_TRANSFORMS = my_val_transforms
                        })
```

# Generates following files
# `my_model-symbol.json` => transformation + network + signature details
# `my_model-0000.params`

Module Inference with End to End models

# Module is bound to a fused symbol graph of transformations
# and neural network.
# You can directly call mod.forward(raw_image_data) and get predictions.
mod = mx.mod.Module.from_end_to_end_model(symbol_file = "my_model-symbol.json",
param_file = "my_model-0000.params",
load_transforms = True, ctx = 'cpu',
batch_size = 1)

# Inference
mod.forward(data)
Java Inference API with End to end models

```python
# Predictor object is bound to a fused symbol graph of transformations
# and neural network.
Predictor predictor = Predictor.from_end_to_end_model(
    symbol_file = "my_model-symbol.json",
    param_file = "my_model-0000.params",
    load_transforms = True,
    context = context,
    batch_size = 1)

// Run Inference with raw input image data
List<List<Float>> result = predictor.predict(inputFloatList);
```

Cons of this Approach

1. New concept of end-to-end models for users. May create confusions and regression on existing models and new end-to-end models.
2. Sharing data transformations from a training job to another training job is fairly straightforward as sharing the code files. In majority of cases all model building experiments happen on Python.
3. For inference, there is typically simple and single transformation graph. This solution tries and enables more information to be saved than necessary.
4. New Export, import APIs, model formats - Requires more understanding the problem domain.

Appendix

FAQ

Q1) Can I implement my own Image Transformation Block and later use it in inference?

A1) As long as your own Image Transformation Block is HybridBlock which takes advantage of existing MXNet operator, then the answer is yes.

If it’s not, unfortunately, you need to implement the backend operator first. But current transform cover most of the use case for image task.

Performance Benchmarks

- Pre-processing - Resize(224, 224), ToTensor, Normalize(mean=(0.485, 0.456, 0.406), std=(0.229, 0.224, 0.225))
- We take average of 500 runs
- Single Request Inference - Input Data - Synthetic (random.uniform(0, 255, shape=(1, 300, 300, 3))
- Batch Inference - Input Data - Synthetic (random.uniform(0, 255, shape=(25, 300, 300, 3))
- Below time gives - Average Prediction Time Per Sample

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Non End to End Models (ms)</th>
<th>End to End Models (ms)</th>
<th>Boost %</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU (C2.2X)</strong></td>
<td>Single Request Inference</td>
<td>Python (Module API)</td>
<td>17</td>
<td>14</td>
<td>17.65%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Java Inference APIs</td>
<td>17.09</td>
<td>14.16</td>
<td>17.14%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Scala Inference APIs</td>
<td>17.93</td>
<td>13.19</td>
<td>26.44%</td>
</tr>
<tr>
<td><strong>Batch Inference (Batch size = 25)</strong></td>
<td></td>
<td>Python (Module API)</td>
<td>15.18</td>
<td>12.57</td>
<td>17.19%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Java Inference APIs</td>
<td>18.54</td>
<td>13</td>
<td>29.88%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Scala Inference APIs</td>
<td>17</td>
<td>13.26</td>
<td>22.00%</td>
</tr>
<tr>
<td><strong>GPU (P3.16X)</strong></td>
<td>Single Request Inference</td>
<td>Python (Module API)</td>
<td>5.78</td>
<td>3.14</td>
<td>45.67%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Java Inference APIs</td>
<td>8.95</td>
<td>4.26</td>
<td>52.40%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Scala Inference APIs</td>
<td>9.14</td>
<td>4.42</td>
<td>51.64%</td>
</tr>
<tr>
<td><strong>Batch Inference (Batch size = 25)</strong></td>
<td></td>
<td>Python (Module API)</td>
<td>2.61</td>
<td>1.31</td>
<td>49.81%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Java Inference APIs</td>
<td>8.03</td>
<td>5.53</td>
<td>31.13%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Scala Inference APIs</td>
<td>7.86</td>
<td>5.52</td>
<td>29.77%</td>
</tr>
</tbody>
</table>
References