Gluon Sparse API Design Proposal

This document discusses the API for the sparse feature in Gluon.

Prerequisite

This document assumes the reader

- is familiar with the existing sparse symbolic API
- is familiar with the existing (non-sparse) Gluon API

Problem Statement

Background: Data Parallel Training w/ High Dimensional Sparse Feature

During data parallel training, a copy of the parameters is stored on the parameter server. For each mini-batch, each worker fetches parameters from the server, computes loss for the mini-batch and the gradients, and sends gradients to parameter server to update the weight. When training a network with high dimensional sparse feature, the number of parameters is usually huge. Sending and receiving the large weights and gradients lead to significant communication overhead.

Opportunity: On-Demand Sparse Pull

To combat such issue, we can exploit the sparsity in the dataset and send only relevant parameters required for computation. To be specific, if the current mini-batch only contains 100 non-zero features, then only the parameters for those 100 features need to be sent over the network for computation.

Goal

PART I - beginners

- Users with a CPU machine, or a GPU device. No distributed training required.

- Provide basic sparse support to commonly used hybrid blocks such as nn.Embedding. They'll leverage sparse operators for computation when inputs are sparse, but won't perform on-demand sparse pull for data parallel training.

PART II - advanced users

- Users with multiple CPU machines, or one machine with multiple GPU devices. Multi-node CPU training, or single-node multi-GPU training is required.
- Provide sparse blocks which performs sparse pull for data parallel training for performance sensitive workloads.
- Provide interfaces for users to customize their own sparse blocks.

PART III - super advanced users

- Users with multiple machines and each machine with multiple GPUs. Multi-node multi-GPU training is required. This type of users is not the primary focus of this design.
- See future work section for more details.

Part I - Hybrid Blocks w/o Sparse Pull

The first part discusses how to enable sparse operators for existing hybrid blocks.

PARAMETER API

Parameter

Sparse parameters are declared with specific storage types, and could be pulled from KVStore via row_sparse_data based on row_ids.

- __init__(..., **stype=None**)
- row_sparse_data(self, ctx, row_id)

HYBRID BLOCK API

nn.Embedding

If sparse_grad is True, invoke SparseEmbedding (instead of Embedding).

- __init__(..., **sparse_grad=False**, **kwargs)

nn.Dense

If input is sparse, invoke sparse.dot (instead of FullyConnected).

- __init__(..., **kwargs)

DATASET API

data.SimpleDataset

A simple dataset with sparse data as inputs

- __init__(self, **data**)

data.ArrayDataset

An array dataset with arrays including sparse data

- __init__(self, ***args**)

EXAMPLE CODE

```
from mxnet.gluon import *
import scipy.sparse
ctx = [mx.gpu(0), mx.gpu(1)]
# construct a sparse dataset
sp_data = scipy.sparse.rand(10, 100000, format='csr')
dataset = data.SimpleDataset(sparse.csr_matrix(sp_data))
train_data = data.DataLoader(dataset, ...)

# declare a network for sparse data
net = nn.HybridSequential()
with net.name_scope():
  net.add(nn.Dense(num_hidden, sparse_grad=True))
  net.add(nn.Dense(num_outputs))
net.collect_params().initialize(...)
# the network is hybridizable with sparse data
net.hybridize()
loss = loss.SoftmaxCrossEntropyLoss()

trainer = gluon.Trainer(...)

for e in range(epochs):
  cumulative_loss = 0
  for batch in train_data:
      data = utils.split_and_load(batch.data[0], ctx)
      label = utils.split_and_load(batch.label[0], ctx)
      Ls = []
```

```
    with autograd.record():
        for x, y in zip(data, label):
            z = net(x)
            l = loss(z, y)
            Ls.append(l)
    for L in Ls:
        L.backward()
    trainer.step(...)
    cumulative_loss += nd.sum(Ls).asscalar()
```

Part II - Blocks w/ Sparse Pull

The second part discusses optimizations for data parallel training with sparse pull enabled. Such optimizations requires pauses during the execution, since the knowledge of what parameters are required may depend on an intermediate result of the partial graph. When the intermediate result is available, the request can be sent to parameter server to fetch the parameters. This is particularly important for Gluon, since the flexible interface allows users to do anything that APIs don't forbid. **Therefore, these blocks are not “hybridizable” due to extra interactions with KVStore.**

SPARSE BLOCK API

contrib.nn.SparseBlock(Block)

The SparseBlock class is the base class of all sparse pull enabled blocks with sparse operators. It adopts a similar interface compared to HybridBlock. The forward API performs some sanity checks and invokes sparse_forward. All blocks inheriting from SparseBlock only needs to implement the sparse_forward interface.

- __init__(self, prefix=None, params=None)
- forward(self, *args)
- sparse_forward(self, x, *args, **kwargs)
- save_params()

contrib.nn.SparseEmbedding(SparseBlock)

A sparse block which invokes SparseEmbedding and pulls row_sparse parameter during sparse_forward

- __init__(self, ...)
- sparse_forward(self, x, weight_param)

contrib.nn.SparseLinear(SparseBlock)

A sparse block which invokes sparse.dot and pulls row_sparse parameter during sparse_forward

- __init__(..., data_stype=None, weight_stype=None, **kwargs)
- sparse_forward(self, F, x, weight_param, bias=None)

EXAMPLE: MULTI-DEVICE TRAINING

```
from mxnet.gluon import *
ctx = [mx.gpu(0), mx.gpu(1)]
train_data = data.DataLoader(...)

# declare network with sparse blocks
net = nn.Sequential()
with net.name_scope():
  net.add(contrib.nn.SparseEmbedding(..))
  net.add(nn.Dense(num_outputs))
net.initialize(ctx)

loss = loss.SoftmaxCrossEntropyLoss()

trainer = gluon.Trainer(net.collect_params(), ctx, ...)

for e in range(epochs):
  cumulative_loss = 0
  for batch in train_data:
    data = utils.split_and_load(batch[0], ctx)
    labels = utils.split_and_load(batch[0], ctx)
    Ls = []
    with autograd.record():
      for input, label in zip(data, label):
        z = net(input)
        L = loss(z, label)
        Ls.append(L)
    for L in Ls:
      L.backward()
    trainer.step(...)
    cumulative_loss += nd.sum(Ls).asscalar()
```

EXAMPLE: INVALID BLOCK w/ ROW_SPARSE WEIGHT

```
class InvalidBlock(Block):

  def __init__(self, **kwargs):
    super(InvalidBlock).__init__(**kwargs)

    self.w = self.params.get('weight', shape=(1000, 1), stype='row_sparse')
    # Exception: 'row_sparse' parameter is not allowed. Please create a
    # SparseBlock if needed.
```

EXAMPLE: CUSTOM SPARSE BLOCK

```
class CustomBlock(SparseBlock):
  def __init__(self, **kwargs):
    super(CustomBlock).__init__(**kwargs)
    # register a sparse parameter
    self.w = self.params.get('weight', shape=(1000, 1), stype='row_sparse')

  def sparse_forward(self, x, w_param):
    # pull row_sparse parameter from KVStore
    w = w_param.row_sparse_data(x.ctx, x.indices)
    # invoke a sparse operator
    result = sparse.dot(x, w)
    return result
```

EXAMPLE: CUSTOM SPARSE BLOCK w/o ROW_SPARSE PULL

```
class CustomBlock(SparseBlock):
  def __init__(self, **kwargs):
    super(CustomBlock).__init__(**kwargs)
    self.w = self.params.get('weight', shape=(1000, 1), stype='row_sparse')

  def sparse_forward(self, x, w):
    result = sparse.dot(x, w)
    return result

x = mx.nd.arange(0, 1000)
net = CustomBlock()
net.initialize(ctx=[mx.gpu(0), mx.gpu(1)])
net(x)
# Exception: parameter 'weight' was not pulled during sparse_forward. Its value
# maybe stale. Please pull the parameter before using it.
```

EXAMPLE: CUSTOM BLOCK w/ SPARSE CHILD BLOCK

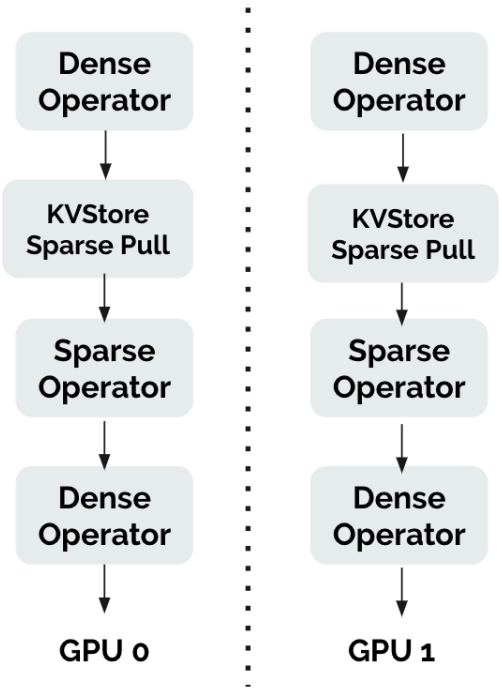
```
class CustomBlock(Block):
  def __init__(self, **kwargs):
      super(CustomBlock).__init__(num_units, **kwargs)
      self.linear = contrib.nn.SparseLinear(num_units)

  def forward(self, x):
      result = self.linear(x)
      return result
```

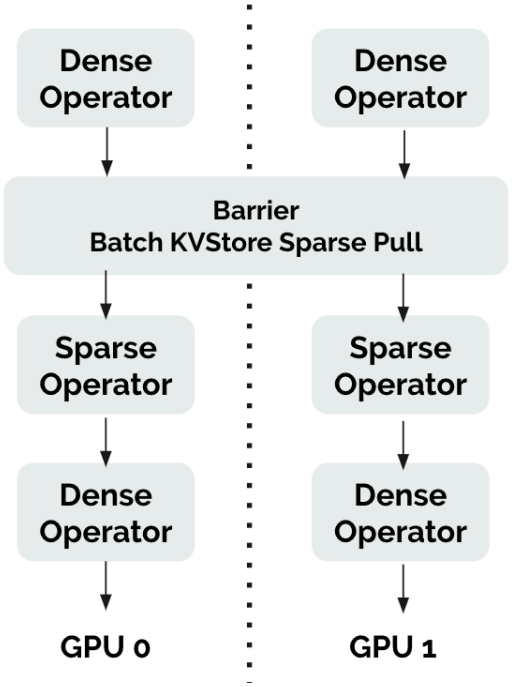
Future Work

Cross-device Synchronization for Multi-machine Multi-GPU Training

When the parameter server sits on CPU or remote machines, for any sparse layer in the network, the request of fetching the parameters should be done in **batch mode** to reduce network & CPU-GPU communication, instead of fetching once per GPU. What this entails is that the execution on different contexts need to be synchronized whenever such optimized sparse operators occur.



(a) normal sparse pull



(b) batched sparse pull to reduce network and CPU to GPU communications

PARALLEL API

contrib.parallel.DataParallel(Object)

The data parallel object launches multiple threads for forward and backward execution. Usually each thread is responsible for one context and push the operations to backend engine.

- __init__(self, block, ctx)
- __call__(self, x, *args, **kwargs)
- _set_ctx

contrib.parallel.ParallelState(Object)

This object provides a **barrier** among multiple threads, which can be used to synchronize multiple threads.

- __init__(self, reducer_broadcaster)
- push(self, ctx, x)
- pull(self, ctx)
- set_ctx(self, ctxes)

contrib.parallel.ParallelSparseState(Object)

A better abstraction of ParallelState for sparse blocks.

- __init__(self)
- push(self, ctx, index, param)
- pull(self, ctx)

contrib.parallel.SparseBlock(Block)

- __init__(self, prefix=None, params=None)
- forward(self, *args)
- sparse_forward(self, x, *args, **kwargs)
- save_params()
- .parallel_states

contrib.parallel.SparseEmbedding(SparseBlock)

- __init__(self, ...)
- sparse_forward(self, x, weight)

contrib.parallel.SparseLinear(SparseBlock)

- __init__(..., data_stype=None, weight_stype=None, **kwargs)
- hybrid_forward(self, F, x, weight_param, bias=None)