Bucketing in MXNet

Introduction

Bucketing is a way to train multiple networks with "different but similar" architectures that share the same set of parameters. A typical application is in recurrent neural networks (RNNs). Unrolling the network explicitly in time is commonly used to implement RNNs in toolkits that use symbolic network definition. For explicit unrolling, we need to know the sequence length in advance. In order to handle all the sequences, we need to unroll the network to the maximum possible sequence length. However, this could be wasteful because on shorter sequences, most of the computations are on padded data.

Bucketing, borrowed from Tensorflow's sequence training example, is a simple solution to this: instead of unrolling the network to the maximum possible sequence, we unroll multiple instance of different lengths (e.g. length 5, 10, 20, 30). During training, the most appropriate unrolled model is used for each mini-batch data with different sequence length. Note for RNNs, although those models have different architecture, because the parameters are shared in time. So although models in different buckets are selected to train in different mini-batches, essentially the same set of parameters are being optimized. MXNet re-use the internal memory buffers among all executors.

For simple RNNs, (potentially at the cost of slower speed) one can use a for loop to explicitly go over the input sequences and do *back-propagate through time* by maintaining the connection of the states and gradients through time. This naturally works with variable length sequences. However, for more complicated models (e.g. seq2seq used translation), explicitly unrolling is the easiest way. In this tutorial, we introduce the APIs in MXNet that allows us to implement bucketing.

Variable-length Sequence Training for PTB

Taking the PennTreeBank language model example as our demonstration here. If you are not familiar with this example, please also refer to this tutorial (in Julia).

The architecture used in the example is a simple word-embedding layer followed by two LSTM layers. In the original example, the model is unrolled explicitly in time for a fixed length of 32. In this demo, we will show how to use bucketing to implement variable-

length sequence training.

In order to enable bucketing, MXNet need to know how to construct a new unrolled symbolic architecture for a different sequence length. To achieve this, instead of constructing a model with a fixed `Symbol`, we use a callback function that generating new `Symbol` on a *bucket key*.

```
model = mx.model.FeedForward(
        ctx     = contexts,
        symbol  = sym_gen)
```

Here `sym_gen` must be a function taking one argument `bucket_key` and returns a `Symbol` for this bucket. In our example, we will use the sequence length as the bucket key. But in general, a bucket key could be anything. For example, in neural translation, since different combination of input-output sequence lengths correspond to different unrolling, the bucket key could be a pair of lengths.

```
def sym_gen(seq_len):
    return lstm_unroll(num_lstm_layer, seq_len, len(vocab),
                       num_hidden=num_hidden, num_embed=num_embed,
                       num_label=len(vocab))
```

The data iterator need to report the `default_bucket_key`, which allows MXNet to do some parameter initialization before actually reading the data. Now the model is capable of training with different buckets by sharing the parameters as well as intermediate computation buffers between bucket executors.

However, to achieve training, we still need to add some extra bits to our `DataIter`. Apart from reporting the `default_bucket_key` as mentioned above, we also need to report the current `bucket_key` for each mini-batch. More specifically, the `DataBatch` object returned in each mini-batch by the `DataIter` should contain the following *extra* properties

- `bucket_key`: the bucket key corresponding to this batch of data. In our example, it will be the sequence length for this batch of data. If the executors corresponding to this bucket key has not yet been created, they will be constructed according to the symbolic model returned by `gen_sym` on this bucket key. Created executors will be cached for future use. Note the generated `Symbol` could be arbitrary, but they should all have the same trainable parameters and auxiliary states.

- `provide_data`: this is the same information reported by the `DataIter` object. Since now each bucket corresponds to different architecture, they could have different input data. Also, one **should** make sure that the `provide_data` information returned by the `DataIter` object is compatible with the architecture for `default_bucket_key`.
- `provide_label`: the same as `provide_data`.

Now the `DataIter` is responsible for grouping the data into different buckets. Assuming randomization is enabled, in each mini-batch, it choose a random bucket (according to a distribution balanced by the bucket sizes), and then randomly choose sequences from that bucket to form a mini-batch. And do some proper *padding* for sequences of different length *within* the mini-batch if necessary.

Please refer to example/rnn/lstm_ptb_bucketing.py for the full implementation of a `DataIter` that read text sequences implement the API shown above.

Beyond Sequence Training

We briefly explained how the bucketing API looks like in the example above. However, as it might be already clear: the API is not limited to bucketing according to the sequence lengths. The bucket key could be arbitrary objects. As long as the architecture returned by `gen_sym` is compatible with each other (having the same set of parameters).