Tensor Notes

Andrew H. Fagg
Symbiotic Computing Laboratory
University of Oklahoma

Tensorflow Tensors

Two types:

- Symbolic tensors:
    - Contain some shape information, as well as dtype
    - Used to connect data flow graphs together
- Eager tensors:
    - Well-defined mathematical tensors
    - Shape, dtype and values

Keras Tensors

- All are symbolic tensors
- Keras libraries (Layers, Models, …) operate on these tensors (but in some cases can also work with symbolic TF tensors)

Execution Modes

Eager mode:

- Tensors are always Eager tensors
- Code is procedural: executes immediately (so, more intuitive from a programming language point of view)
- Helpful for interactive debugging

Symbolic Tensor Mode

- The code produces a data-flow graph
- This code works in terms of non-Eager tensors
- Methods including model.fit() .predict() .evaluate()
  - Push Eager tensors through the data-flow graph

Eager vs Symbolic Tensors

- Keras classes: (essentially) always non-Eager
- Tensorflow classes: mix of eager and non-Eager operators (some operators will work with either)

Loss Functions

- Translate a set of examples (input/desired output pairs) into ‘loss’
- model.fit(), .evaluate() will accumulate the data across the set of batches to compute the scalar loss
- Ultimately, loss functions are evaluated with Eager tensors, but often use in graph mode
  - Requires bridging the gap between symbolic and Eager tensors

Loss Example

```
@classmethod

def mdn_loss(cls, y, dist):

    return -dist.log_prob(y)
```

- Referenced in the compile step
- Computes loss for each example individually
- Expressed using Eager tensors
- compile() adds a wrapper that connects this loss into the symbolic graph that is being created

Keras Loss Example

Relationship of symbolic and Eager tensors is more explicit:

```
class WeightedLogLikelihood(Loss):
    def __init__(self, name="weighted_log_likelihood"):
        super().__init__(name=name)

    def call(self, y_true, y_pred, sample_weight=None):
        n_log_prob = -y_pred.log_prob(y_true)

        if sample_weight is not None:
            log_prob *= sample_weight
            loss = tf.reduce_sum(n_log_prob) / tf.reduce_sum(sample_weight)
        else:
            loss = tf.reduce_mean(n_log_prob)

    return loss
```

Metrics

- Applied to the entirety of a data set
- … even when that data set is composed of multiple batches
- For custom metrics, must provide methods for:
  - Accumulating the needed data across batches
  - Generating the output metric given the accumulated data
  - Resetting the accumulators

Metric Example

```
class CustomMSE(Metric):
     def __init__(self, name="custom_mse", **kwargs):
          super().__init__(name=name, **kwargs)
          # Variables for holding the accumulated values (numerator and denominator)
          self.total = self.add_weight(name="total", initializer="zeros")
          self.count = self.add_weight(name="count", initializer="zeros")

     def update_state(self, y_true, y_pred, sample_weight=None):
          # One batch
          error = tf.square(y_true - y_pred)
          if sample_weight is not None:
               error *= tf.cast(sample_weight, error.dtype)
          self.total.assign_add(tf.reduce_sum(error))
          self.count.assign_add(tf.cast(tf.size(y_true), tf.float32))

     def result(self):
          return self.total / self.count

     def reset_state(self):
          self.total.assign(0.0)
          self.count.assign(0.0)
```