OP 1
OP 2
OP 1
A
A
A
TensorFlow

Introducing tf.data

Better input pipelines for TensorFlow

Derek Murray
@mrry

Why are we here?

Input data is the lifeblood of machine learning

Why are we here?

Input data is the lifeblood of machine learning

Modern accelerators need faster input pipelines

Why are we here?

Input data is the lifeblood of machine learning

Modern accelerators need faster input pipelines

A better way to get your data into TensorFlow

Feeding

```
sess.run(…,
    feed_dict={x: features,
               y: labels})
```

All the flexibility of Python, but potentially poor performance.

Queues

```
files = string_input_producer(…)
record = TFRecordReader().read(files)
parsed = parse_example(record, …)
batch = batch(parsed, 32)
```

Uses TensorFlow ops to perform preprocessing, but driven by client threads, and complex.

Input pipelines = lazy lists

Functional programming to the rescue!

Input pipelines = lazy lists

Functional programming to the rescue!

Data elements have the same type

Input pipelines = lazy lists

Functional programming to the rescue!

Data elements have the same type

Dataset might be too large to materialize all at once… or infinite

Input pipelines = lazy lists

Functional programming to the rescue!

Data elements have the same type

Dataset might be too large to materialize all at once... or infinite

Compose functions like `map()` and `filter()` to preprocess

Input pipelines = lazy lists

Functional programming to the rescue!

A well-studied area, applied in existing languages.

- C# LINQ, Scala collections, Java Streams

Huge literature on optimization (stream fusion etc.)

Introducing `tf.data`

Functional input pipelines in TensorFlow

The `Dataset` interface

Data sources and functional transformations

The `Dataset` interface

Data sources and functional transformations

Create a `Dataset` from one or more `tf.Tensor` objects:

The `Dataset` interface

Data sources and functional transformations

Create a `Dataset` from one or more `tf.Tensor` objects:

```
Dataset.from_tensors((features, labels))
```

The `Dataset` interface

Data sources and functional transformations

Create a `Dataset` from one or more `tf.Tensor` objects:

```
Dataset.from_tensors((features, labels))
Dataset.from_tensor_slices((features, labels))
```

The `Dataset` interface

Data sources and functional transformations

Create a `Dataset` from one or more `tf.Tensor` objects:

```
Dataset.from_tensors((features, labels))
Dataset.from_tensor_slices((features, labels))
TextLineDataset(filenames)
```

The `Dataset` interface

Data sources and functional transformations

Or create a `Dataset` from another `Dataset`:

The `Dataset` interface

Data sources and functional transformations

Or create a `Dataset` from another `Dataset`:

```
dataset.map(lambda x: tf.decode_jpeg(x))
```

The `Dataset` interface

Data sources and functional transformations

Or create a `Dataset` from another `Dataset`:

```
dataset.map(lambda x: tf.decode_jpeg(x))
dataset.repeat(NUM_EPOCHS)
```

The `Dataset` interface

Data sources and functional transformations

Or create a `Dataset` from another `Dataset`:

```
dataset.map(lambda x: tf.decode_jpeg(x))
dataset.repeat(NUM_EPOCHS)
dataset.batch(BATCH_SIZE)
```

The `Dataset` interface

Data sources and functional transformations

Or create a `Dataset` from another `Dataset`:

```
dataset.map(lambda x: tf.decode_jpeg(x))
dataset.repeat(NUM_EPOCHS)
dataset.batch(BATCH_SIZE)
```

...and many more.

The `Dataset` interface

Data sources and functional transformations

Or (in TensorFlow 1.4) create a `Dataset` from a Python generator:

```
def generator():
  while True:
    yield ...

Dataset.from_generator(generator, tf.int32)
```

```python
# Read records from a list of files.
dataset = TFRecordDataset(["file1.tfrecord", "file2.tfrecord", …])

# Parse string values into tensors.
dataset = dataset.map(lambda record: tf.parse_single_example(record, …))

# Randomly shuffle using a buffer of 10000 examples.
dataset = dataset.shuffle(10000)

# Repeat for 100 epochs.
dataset = dataset.repeat(100)

# Combine 128 consecutive elements into a batch.
dataset = dataset.batch(128)
```

The `Iterator` interface

Sequential access to `Dataset` elements

The `Iterator` interface

Sequential access to `Dataset` elements

Create an `Iterator` from a `Dataset`:

The `Iterator` interface

Sequential access to `Dataset` elements

Create an `Iterator` from a `Dataset`:

```
dataset.make_one_shot_iterator()
```

The `Iterator` interface

Sequential access to `Dataset` elements

Create an `Iterator` from a `Dataset`:

```
dataset.make_one_shot_iterator()
```

```
dataset.make_initializable_iterator()
```

The `Iterator` interface

Sequential access to `Dataset` elements

Initialize the `Iterator` if necessary:

```
sess.run(iterator.initializer, feed_dict=PARAMS)
```

The `Iterator` interface

Sequential access to `Dataset` elements

Get the next element from the `Iterator`:

```
next_element = iterator.get_next()
while …:
  sess.run(next_element)
```

```python
dataset = …

# A one-shot iterator automatically initializes itself on first use.
iterator = dataset.make_one_shot_iterator()

# The return value of get_next() matches the dataset element type.
images, labels = iterator.get_next()
train_op = model_and_optimizer(images, labels)

# Loop until all elements have been consumed.
try:
  while True:
    sess.run(train_op)
except tf.errors.OutOfRangeError:
  pass
```

```python
def input_fn():
  dataset = …
  # A one-shot iterator automatically initializes itself on first use.
  iterator = dataset.make_one_shot_iterator()
  # The return value of get_next() matches the dataset element type.
  images, labels = iterator.get_next()
  return images, labels
# The input_fn can be used as a regular Estimator input function.
estimator = tf.estimator.Estimator(…)
estimator.train(train_input_fn=input_fn, …)
```

```python
dataset = …

# An initializable iterator can be re-initialized before each epoch.
iterator = dataset.make_initializable_iterator()

images, labels = iterator.get_next()
train_op = f(images, labels)

for i in NUM_EPOCHS:
  # Initialize iterator for epoch i.
  sess.run(iterator.initializer)
  try:
    while True:
      sess.run(train_op)
  except tf.errors.OutOfRangeError:
    pass
  # Perform end-of-epoch computation here.
```

`tf.data` API

`tf.data.Dataset`

Represents input pipeline using functional transformations

`tf.data.Iterator`

Provides sequential access to elements of a `Dataset`

Tuning `tf.data` performance

Functional input pipelines in TensorFlow

Tuning performance

`tf.data` is implemented in C++ to avoid Python overhead

Tuning performance

`tf.data` is implemented in C++ to avoid Python overhead

Execution is deterministic, sequential and synchronous *by default*

```
dataset = TFRecordDataset(["file1.tfrecord", "file2.tfrecord", …])

dataset = dataset.map(lambda record: tf.parse_single_example(record, …))

dataset = dataset.shuffle(10000)
dataset = dataset.repeat(100)
dataset = dataset.batch(128)
```

```
dataset = TFRecordDataset(["file1.tfrecord", "file2.tfrecord", …])

dataset = dataset.map(lambda record: tf.parse_single_example(record, …))

dataset = dataset.shuffle(10000)
dataset = dataset.repeat(100)
dataset = dataset.batch(128)

# Use prefetch() to overlap the producer and consumer.
dataset = dataset.prefetch(1)
```

```python
dataset = TFRecordDataset(["file1.tfrecord", "file2.tfrecord", …])


# Use num_parallel_calls to parallelize map().
dataset = dataset.map(lambda record: tf.parse_single_example(record, …),
                      num_parallel_calls=64)

dataset = dataset.shuffle(10000)
dataset = dataset.repeat(100)
dataset = dataset.batch(128)

# Use prefetch() to overlap the producer and consumer.
dataset = dataset.prefetch(1)
```

```python
# Use interleave() and prefetch() to read many files concurrently.
files = Dataset.list_files("*.tfrecord")
dataset = files.interleave(lambda x: TFRecordDataset(x).prefetch(100),
                           cycle_length=8)

# Use num_parallel_calls to parallelize map().
dataset = dataset.map(lambda record: tf.parse_single_example(record, …),
                      num_parallel_calls=64)

dataset = dataset.shuffle(10000)
dataset = dataset.repeat(100)
dataset = dataset.batch(128)

# Use prefetch() to overlap the producer and consumer.
dataset = dataset.prefetch(1)
```

Looking to the future

This month: API moving to `tf.data` in TensorFlow 1.4
(update: done!)

Looking to the future

This month: API moving to `tf.data` in TensorFlow 1.4

Short-term: Automatic staging to GPU memory

Looking to the future

This month: API moving to `tf.data` in TensorFlow 1.4

Short-term: Automatic staging to GPU memory

Long-term: Automatic performance optimization

Conclusion

Getting your data into TensorFlow with `tf.data`

- Simple
- Fast
- Flexible

Blogpost: **https://goo.gl/RyLuUw**

Thank you!

Derek Murray
@mrry