scikit-learn、Keras、TensorFlow による 実践機械学習 第 2 版 第 13 章 前半 TensorFlow によるデータのロードと前処理

データ工学研究室 輪読会

AL18036 片岡 凪

芝浦工業大学 工学部 情報工学科 4 年

May 13, 2021

発表者

- 片岡 凪
- 芝浦工業大学 工学部 情報工学科 4 年
- データ工学研究室（木村昌臣研究室）
- Github KataokaNagi
- Twitter @calm_IRL

目次

TensorFlow のデータ API

- 大規模なデータセットに対応
- データの入手元と変換方法のみを指示
- 以下は TF（Tensorflow）が代行
  - マルチスレッド管理
  - キューイング
  - バッチへの分割
  - プリフェッチ

データ API での入力形式

- 拡張前
  - CSV などのテキストファイル
  - レコードが固定長のバイナリファイル
  - レコードが可変長の TFRecord 形式
    - 柔軟で効率が良い
    - 通常はプロトコルバッファを格納（オープンソースのバイナリフォーマット）
- その他、BigQuery などに拡張可能

前処理

- 主な対象
  - スケールの異なる数値フィールド群
  - テキスト特徴量
  - カテゴリ特徴量
- 主な対処
  - 正規化
  - ワンホットエンコーディング（13 章後半）
  - バッグオブワーズエンコーディング
  - 埋め込み（embedding）（13 章後半）

便利な TF ライブラリ

- TF Transform (tf.Transform)
  - 訓練前：訓練セット全体を高速なバッチモードで前処理
  - 訓練後：TF 関数として訓練済みモデルに組み込んで前処理
    - 本番環境で新たなインスタンスをその場で処理可能
- TF Datasets（TFDS）
  - 大規模なデータセットをダウンロード可能な関数を提供
  - データ API で上記関数を操作可能なデータセットオブジェクトを提供

目次

データセット

- ディスクなどから少しずつ読み出していくデータ要素のシーケンス
- 例：from_tensor_slices(X)
  - テンソル X の第 1 次元のスライスであるデータセットを返す
    - つまり、tf のスライスを Dataset のスライスに変換？

```
[ ]  X = tf.range(10)
     dataset = tf.data.Dataset.from_tensor_slices(X)

     dataset = tf.data.Dataset.range(10) # same

     print(X)
     print(dataset)
```

```
tf.Tensor([0 1 2 3 4 5 6 7 8 9], shape=(10,), dtype=int32)
<TensorSliceDataset shapes: (), types: tf.int32>
```

```
[ ]  for item in dataset:
         print(item)
```

```
tf.Tensor(0, shape=(), dtype=int64)
tf.Tensor(1, shape=(), dtype=int64)
tf.Tensor(2, shape=(), dtype=int64)
tf.Tensor(3, shape=(), dtype=int64)
tf.Tensor(4, shape=(), dtype=int64)
tf.Tensor(5, shape=(), dtype=int64)
tf.Tensor(6, shape=(), dtype=int64)
tf.Tensor(7, shape=(), dtype=int64)
tf.Tensor(8, shape=(), dtype=int64)
tf.Tensor(9, shape=(), dtype=int64)
```

目次

別のデータセットに変換するメソッド 1/4

- repeat(n)
  - 1 つのスライスを n 回連結
  - コピーではない（高速かつ小容量？）
- batch(n)
  - スライスを n 個ずつのスライスに分割
  - 最後のバッチが n 個未満の場合、引数に drop_remainder=True で削除可能

```
[ ]  # kataoka added
     dataset = dataset.repeat(3)
     for item in dataset:
         print(item)
```

```
tf.Tensor(0, shape=(), dtype=int64)
tf.Tensor(1, shape=(), dtype=int64)
tf.Tensor(2, shape=(), dtype=int64)
tf.Tensor(3, shape=(), dtype=int64)
tf.Tensor(4, shape=(), dtype=int64)
tf.Tensor(5, shape=(), dtype=int64)
tf.Tensor(6, shape=(), dtype=int64)
tf.Tensor(7, shape=(), dtype=int64)
tf.Tensor(8, shape=(), dtype=int64)
tf.Tensor(9, shape=(), dtype=int64)
tf.Tensor(0, shape=(), dtype=int64)
tf.Tensor(1, shape=(), dtype=int64)
tf.Tensor(2, shape=(), dtype=int64)
tf.Tensor(3, shape=(), dtype=int64)
```

```
[ ]  dataset = dataset.repeat(3).batch(7)
     for item in dataset:
         print(item)
```

```
tf.Tensor([0 1 2 3 4 5 6], shape=(7,), dtype=int64)
tf.Tensor([7 8 9 0 1 2 3], shape=(7,), dtype=int64)
tf.Tensor([4 5 6 7 8 9 0], shape=(7,), dtype=int64)
tf.Tensor([1 2 3 4 5 6 7], shape=(7,), dtype=int64)
tf.Tensor([8 9 0 1 2 3 4], shape=(7,), dtype=int64)
tf.Tensor([5 6 7 8 9 0 1], shape=(7,), dtype=int64)
```

別のデータセットに変換するメソッド 2/4

- map(lambda x: x の式)
  - 要素をラムダ式で柔軟に変換するなど
  - TF 関数に変換可能なラムダ式のみ対応（12 章）
  - 引数 num_parallel_calls=tf.data.experimental.AUTOTUNE で並列化＆高速化

```
[ ]  dataset = dataset.map(lambda x: x * 2)

[ ]  for item in dataset:

         print(item)

tf.Tensor([ 0  2  4  6  8 10 12], shape=(7,), dtype=int64)
tf.Tensor([14 16 18  0  2  4  6], shape=(7,), dtype=int64)
tf.Tensor([ 8 10 12 14 16 18  0], shape=(7,), dtype=int64)
tf.Tensor([ 2  4  6  8 10 12 14], shape=(7,), dtype=int64)
tf.Tensor([18 18  0  2  4  6  8], shape=(7,), dtype=int64)
```

別のデータセットに変換するメソッド 3/4

- unbach()
  - bach() で作ったデータセットを解体
  - 試験段階のメソッドで、collab ではエラー
- apply()
  - [] で囲まれたデータセット全体に適用
  - 引数に dataset.unbach() などを用いる

```
[ ]  # Error
     # dataset = dataset.apply(tf.data.experimental.unbatch()) # Now deprecated
     # dataset = dataset.apply(dataset.unbatch())
     # dataset = dataset.unbatch()
```

別のデータセットに変換するメソッド 4/4

- filter(lambda x: x を用いた bool 演算)
  - ラムダ式が true の要素 x のみのスライスを返す
- take(n)
  - 先頭から n 個の要素を用いる

```
[ ]  # Error
     # dataset = dataset.filter(lambda x: x < 10)  # keep only items < 10
```

```
[ ]  for item in dataset.take(3):
         print(item)
```

```
tf.Tensor([ 0  2  4  6  8 10 12], shape=(7,), dtype=int64)
tf.Tensor([14 16 18  0  2  4  6], shape=(7,), dtype=int64)
tf.Tensor([ 8 10 12 14 16 18  0], shape=(7,), dtype=int64)
```

目次

Dataset のシャッフル 1/2

- shuffle(buffer_size=n, seed=m)
  - 要素のシャッフル
  - 独立同分布（同一の分布から独立に抽出された状態）でよく機能する勾配降下法などで使用　(4 章)
  - 大規模なデータからサイズ n のバッファを経由してシャッフル
  - 大規模データセットに見合った大きいバッファサイズが必要
  - RAM の容量に注意
  - shuffle() 後に repeat(3) しても、3 回とも異なるリピートとなる（次頁）
    - 引数 reshuffle_each_iteration=False で同じリピートにできる

Dataset のシャッフル 2/2

```
[ ] # シャッフル後にリピートしても、リピートごとに異なる要素となる
    tf.random.set_seed(42)

    dataset = tf.data.Dataset.range(10)
    dataset = dataset.shuffle(buffer_size=3, seed=42).batch(7)
    dataset = dataset.repeat(3)
    for item in dataset:
        print(item)
```

```
tf.Tensor([1 3 0 4 2 5 6], shape=(7,), dtype=int64)
tf.Tensor([8 7 9], shape=(3,), dtype=int64)
tf.Tensor([1 3 0 2 8 7 5], shape=(7,), dtype=int64)
tf.Tensor([8 9 4], shape=(3,), dtype=int64)
tf.Tensor([0 1 3 4 5 6 7], shape=(7,), dtype=int64)
tf.Tensor([2 9 8], shape=(3,), dtype=int64)
```

```
[ ] # シャッフル後にリピートしても、リピートごとに異なる要素となるが、
    # reshuffle_each_iteration=Falseでリピートごとに同じ要素にできる
    tf.random.set_seed(42)

    dataset = tf.data.Dataset.range(10)
    dataset = dataset.shuffle(buffer_size=3, seed=42, reshuffle_each_iteration=False).batch(7)
    dataset = dataset.repeat(3)
    for item in dataset:
        print(item)
```

```
tf.Tensor([0 2 3 5 6 4 8], shape=(7,), dtype=int64)
tf.Tensor([9 1 7], shape=(3,), dtype=int64)
tf.Tensor([0 2 3 5 6 4 8], shape=(7,), dtype=int64)
tf.Tensor([9 1 7], shape=(3,), dtype=int64)
tf.Tensor([0 2 3 5 6 4 8], shape=(7,), dtype=int64)
tf.Tensor([9 1 7], shape=(3,), dtype=int64)
```

便利なシャッフル

- list_files(DATA_FILE_PATH, seed=n)
  - データをシャッフルしてからロード
  - 引数 suffle=False も指定可能
- interleave(lambda file_name: **ラムダ式**, cycle_length=n)
  - n 個のファイルを同時に無作為に読み出す
  - n 個のデータセットをもつ 1 個のデータセットを作成
  - n 個のデータセット間の同じ長さの部分が互い違いになる
    - 引数 num_parallel_calls=tf.data.experimental.AUTOTUNE で初めて並列化する
  - コードでは、ファイルパスが尽きるまで 5 個ずつ取り出して
    インターリーブする操作を繰り返している

目次

CSV データのスケーリング

- TF で上手く平均 0 分散 1 に統一したい
- tf.io.decode_csv(LINE, record=[...])
  - 第 1 引数：CSV の 1 行
  - 第 2 引数：1 行の列数とデータ型がわかる初期値を格納した配列
    - 敢えて 0. を入れると欠損時に例外を吐く
  - 1 レコードの 1 次元テンソル [[...], ..., [...]] を返す
    - tf.stack(fields[スライス表現]) で [...,] に直す

```
[ ]  n_inputs = 8 # X_train.shape[-1]

     @tf.function
     def preprocess(line):
         defs = [0.] * n_inputs + [tf.constant([], dtype=tf.float32)]
         fields = tf.io.decode_csv(line, record_defaults=defs)
         x = tf.stack(fields[:-1])
         y = tf.stack(fields[-1:])
         return (x - X_mean) / X_std, y
```

目次

これまでの処理を関数化

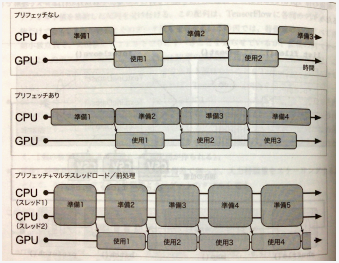
- 新要素は最後の行の prefetch(1) のみ（後述）

```
[ ]  def csv_reader_dataset(filepaths, repeat=1, n_readers=5,
                            n_read_threads=None, shuffle_buffer_size=10000,
                            n_parse_threads=5, batch_size=32):
         dataset = tf.data.Dataset.list_files(filepaths).repeat(repeat)
         dataset = dataset.interleave(
             lambda filepath: tf.data.TextLineDataset(filepath).skip(1),
             cycle_length=n_readers, num_parallel_calls=n_read_threads)
         dataset = dataset.shuffle(shuffle_buffer_size)
         dataset = dataset.map(preprocess, num_parallel_calls=n_parse_threads)
         dataset = dataset.batch(batch_size)
         return dataset.prefetch(1)
```

目次

プリフェッチによる並列化

- prefetch(n)
  - n 個のバッチを CPU と GPU で並列・高速化
  - 一般に n=1 でよい
  - GPU の RAM の容量や帯域幅（速度）が重要
  - 引数 num_parallel_calls
    =tf.data.experimental.AUTOTUNE
    - CPU 内で並列化
    - n を自動調節



その他の便利な関数

- cache()
  - データセットがメモリに入る程度に小さいときに高速化
  - ロード前処理とシャッフルの間で実行
- concatenate()
- zip()
- window()
- reduce()
- shard() = 破片
- flat_map()
- padded_batch()
- from_generator()
- from_tensors()

目次

データセットの応用 1/3

- 種々のデータセットの作成

```
[ ]  train_set = csv_reader_dataset(train_filepaths, repeat=None)
     valid_set = csv_reader_dataset(valid_filepaths)
     test_set = csv_reader_dataset(test_filepaths)
```

データセットの応用 2/3

- モデルの構築と訓練

```
[ ]  keras.backend.clear_session()
     np.random.seed(42)
     tf.random.set_seed(42)

     model = keras.models.Sequential([
         keras.layers.Dense(30, activation="relu", input_shape=X_train.shape[1:]),
         keras.layers.Dense(1),
     ])
```

```
[ ]  model.compile(loss="mse", optimizer=keras.optimizers.SGD(lr=1e-3))
```

```
[ ]  batch_size = 32
     model.fit(train_set, steps_per_epoch=len(X_train) // batch_size, epochs=10,
               validation_data=valid_set)
```

```
Epoch 1/10
362/362 [==============================] - 2s 4ms/step - loss: 2.0914 - val_loss: 21.5124
Epoch 2/10
362/362 [==============================] - 1s 3ms/step - loss: 0.8428 - val_loss: 0.8648
Epoch 3/10
362/362 [==============================] - 1s 3ms/step - loss: 0.6329 - val_loss: 0.6196
```

データセットの応用 3/3

- テストの評価と新インスタンスの予測

```
[ ]  model.evaluate(test_set, steps=len(X_test) // batch_size)

     161/161 [==============================] - 0s 2ms/step - loss: 0.4788
     0.4787752032279968

[ ]  new_set = test_set.map(lambda X, y: X) # we could instead just pass test_set, Keras would ignore the labels
     X_new = X_test
     model.predict(new_set, steps=len(X_new) // batch_size)

     array([[2.3576407],
            [2.255291 ],
            [1.4437604],
            ...,
            [0.5654392],
            [3.9442453],
            [1.0232248]], dtype=float32)
```

独自の TF 訓練関数（12 章と同様）

- p,401 の自動微分のコードと比較するとよい

```
[ ]  optimizer = keras.optimizers.Nadam(lr=0.01)
     loss_fn = keras.losses.mean_squared_error

     @tf.function
     def train(model, n_epochs, batch_size=32,
               n_readers=5, n_read_threads=5, shuffle_buffer_size=10000, n_parse_threads=5):
         train_set = csv_reader_dataset(train_filepaths, repeat=n_epochs, n_readers=n_readers,
                           n_read_threads=n_read_threads, shuffle_buffer_size=shuffle_buffer_size,
                           n_parse_threads=n_parse_threads, batch_size=batch_size)
         for X_batch, y_batch in train_set:
             with tf.GradientTape() as tape:
                 y_pred = model(X_batch)
                 main_loss = tf.reduce_mean(loss_fn(y_batch, y_pred))
                 loss = tf.add_n([main_loss] + model.losses)
             gradients = tape.gradient(loss, model.trainable_variables)
             optimizer.apply_gradients(zip(gradients, model.trainable_variables))
```

目次

TFRecord 概要

- TF データのロードやパースがボトルネックなら用いる
- 大規模なデータの効率的な格納・読み出しが可能
- 可変長バイナリレコードのシーケンス
  - 長さ情報
  - 長さ情報の CRC チェックサム
  - 実データ
  - 実データのチェックサム

TFRecord の利用

- 書き込み

```
[ ]  with tf.io.TFRecordWriter("my_data.tfrecord") as f:
         f.write(b"This is the first record")
         f.write(b"And this is the second record")
```

- 読み出し・出力

```
[ ]  filepaths = ["my_data.tfrecord"]
     dataset = tf.data.TFRecordDataset(filepaths)
     for item in dataset:
         print(item)
```

```
tf.Tensor(b'This is the first record', shape=(), dtype=string)
tf.Tensor(b'And this is the second record', shape=(), dtype=string)
```

目次

TFRecord ファイルの圧縮

- TF データの送受信に利用
- option 引数で指定
- 解凍

```
[ ]  options = tf.io.TFRecordOptions(compression_type="GZIP")
     with tf.io.TFRecordWriter("my_compressed.tfrecord", options) as f:
         f.write(b"This is the first record")
         f.write(b"And this is the second record")
```

```
[ ]  dataset = tf.data.TFRecordDataset(["my_compressed.tfrecord"],
                                        compression_type="GZIP")
     for item in dataset:
         print(item)
```

```
tf.Tensor(b'This is the first record', shape=(), dtype=string)
tf.Tensor(b'And this is the second record', shape=(), dtype=string)
```

目次

protobuf（プロトコルバッファ）

- 通常の TFRecord で使われるシリアライズ化されたバイナリデータ
- 可搬性と拡張性に優れる
- protoc：protobuf コンパイラ
- .protoc ファイル内の定義例

```
[ ]  %%writefile person.proto
     syntax = "proto3";
     message Person {
       string name = 1;
       int32 id = 2;
       repeated string email = 3;
     }

     Writing person.proto
```

Python の protobuf アクセスクラスの使用例

- person インスタンスの可視化や読み書き
- SerializeToString() でシリアライズ
- ParseFromString() でデシリアライズ

```
[ ] from person_pb2 import Person

    person = Person(name="Al", id=123, email=["a@b.com"])  # create a Person
    print(person)  # display the Person
```

```
name: "Al"
id: 123
email: "a@b.com"
```

```
[ ] s = person.SerializeToString()  # serialize to a byte string
    s
```

```
b'¥n¥x05Alice¥x10{¥x1a¥x07a@b.com¥x1a¥x07c@d.com'
```

```
[ ] person2 = Person()  # create a new Person
    person2.ParseFromString(s)  # parse the byte string (27 bytes)
```

```
27
```

```
[ ] person == person2  # now they are equal
```

```
True
```

protobuf と TF との関係

- protoc ファイルで定義したクラスは TF オペレーションでない
  - TF 関数に単純に組み込めない
  - tf.py_function() で組み込むと速度と可搬性が落ちる
  - TF で定義された特別な protobuf 定義で解決する
- その他詳細
  - https://developers.google.com/protocol-buffers/