CHAPTER 13

Loading and Preprocessing Data

with TensorFlow

Chapter 13 explains how to use TensorFlow's **Data API**, TFRecords, and preprocessing tools to build efficient, scalable input pipelines that feed large datasets to Keras models.

Data API basics

The **Data API** centers on tf.data.Dataset, which represents a sequence of elements (often feature/label pairs) and supports chaining transformations. You can create datasets from memory (from_tensor_slices, range), files (TextLineDataset, TFRecordDataset, list_files), generators, or tensors, then transform them using methods like map, filter, batch, repeat, shuffle, apply, concatenate, zip, and prefetch. Typical training pipelines:

- shuffle(buffer_size) to randomize order with a sliding buffer.
- repeat() to iterate for multiple epochs.
- map(preprocess, num_parallel_calls=AUTOTUNE) to decode/transform records.
- batch(batch_size) to group examples.
- prefetch(1) to overlap CPU preprocessing and GPU training, improving utilization.

The chapter shows how to interleave records from multiple files using list_files(...).interleave(TextLineDataset(...).skip(1), ...), enabling better shuffling for large datasets split across many CSVs. Datasets integrate directly with **Keras**: you can pass training/validation/test datasets to model.fit, evaluate, and predict, or iterate over them in custom training loops.

Parsing CSV and building an efficient reader

For the California housing example, each CSV line is parsed with tf.io.decode_csv using default values to define column types and missing-value behavior. A preprocess function:

- Parses a line into fields.
- Stacks feature fields into a vector, separates the target.
- Standardizes features using precomputed training means and standard deviations.

A reusable helper csv_reader_dataset then:

- Lists training files, interleaves lines from several readers in parallel.
- Applies preprocess in parallel with map.

- Shuffles with a buffer, optionally repeats, batches, and **prefetches** one batch ahead. This pattern yields a high-throughput pipeline that keeps the GPU busy while reading from multiple files concurrently.

TFRecord format and protocol buffers

TFRecord is TensorFlow's preferred binary record format: a sequence of length-prefixed records with checksums; it is efficient and flexible for large datasets and complex data (e.g., images, audio). TFRecords typically store serialized **protocol buffers**, especially:

- **Example**: a mapping from string feature names to Feature values, where each Feature is a list of bytes, floats, or int64s.
- **SequenceExample**: for lists of lists (e.g., sequences of sentences), with context features and feature_lists for variable-length sequences.

Writing: construct an Example (or SequenceExample), serialize it with SerializeToString(), and write to a TFRecordWriter (optionally with compression like GZIP). Reading:

- Use TFRecordDataset (with compression_type if compressed).
- Parse each record with tf.io.parse_single_example or batched parse_example, given a feature description mapping names to FixedLenFeature or VarLenFeature descriptors.
- Variable-length features become **sparse tensors**, convertible via tf.sparse.to_dense or by using .values.

Binary fields can hold JPEG-encoded images (tf.io.encode_jpeg / decode_jpeg) or serialized tensors (tf.io.serialize_tensor / parse_tensor).

Feature preprocessing and categorical encoding

The chapter covers several ways to **preprocess input features**:

- **Numerical features**: standardization via a Lambda or preprocessing layer that subtracts means and divides by stds (e.g., using keras.layers.Lambda).
- **Categorical features**:
    - **One-hot encoding** using lookup tables (tf.lookup.StaticVocabularyTable with optional OOV buckets) and tf.one_hot; suitable when the vocabulary is small (e.g., <10–50 categories).
    - **Embeddings** using tf.nn.embedding_lookup or keras.layers.Embedding, mapping category indices to dense vectors (10–300 dimensions typical) that are learned during training; preferred for larger vocabularies or hashed categories.

The upcoming **Keras preprocessing layers** (e.g., TextVectorization, Normalization, Discretization) provide standard, serializable layers whose adapt method learns statistics (vocabularies, means/stds) from a sample, and whose call method applies the transformation in the model.

TF Transform and TFDS

TF Transform (tft), part of TFX, lets you define preprocessing once in Python using TensorFlow ops and tft analyzers
(e.g., tft.scale_to_z_score, tft.compute_and_apply_vocabulary). It:

- Runs preprocessing in **batch** over the full training set (often via Apache Beam), computing global statistics.
- Exports an equivalent **TF Function** that is embedded in the serving model, ensuring identical preprocessing in training and production and eliminating training/serving skew.

TensorFlow Datasets (TFDS) provides ready-made datasets (from MNIST to ImageNet) and tf.data pipelines:

- tfds.load(name, batch_size=..., as_supervised=True) downloads, caches, and returns training/test datasets as tf.data.Dataset objects.
- You can then add shuffle, map, batch, and prefetch, or let load handle batch size and supervision, and pass these datasets directly to model.fit.

For your Word summary, useful figures from Chapter 13 include: the dataset transformation chain (repeat/batch/map/shuffle/prefetch), the CPU–GPU overlap with prefetching, the CSV multi-file interleaving diagram, the TFRecord/Example structure, and the dataflow for TF Transform and TFDS