Loading and Preprocessing Data with TensorFlow

The tf.data API

- The **tf.data** API can load and preprocess data efficiently. It can read from multiple files in parallel using multithreading, queuing, shuffling, batching samples, etc.
    - This API loads and preprocesses the next batch of data across multiple CPU cores while the GPUs or TPUs train the current batch of data.
- The API revolves around the **tf.data.Dataset()** class, which represents a sequence of data items.
    - We can create a dataset from a data tensor using the **from_tensor_slices(x)** function.
        - This function takes a tensor and creates a dataset whose elements are all x slices along the first dimension (e.g., number line).
        - When slicing a tuple, a dictionary, or a nested structure, the dataset will only slice the tensors it contains while preserving the tuple/dictionary structure.

Chaining Transformations

- We can apply several transformations to our new dataset.
    - The **repeat(x)** method returns a new dataset that repeats the items of the original dataset x times.
        - If no argument is given to the method, Python will repeat the original dataset infinitely.
    - The **batch(x)** method groups items of the previous dataset in batches of x items.
    - The **map(x)** method can apply preprocessing to the data based on a function x.
        - The method has a *num_parallel_calls* argument, which tells the method to use multiple threads to speed up the process.
    - We can filter the data using the **filter(x)** method based on a function x.
    - We can look at a few items from the dataset using **take(x),** where x is the number of tensors/rows.

Shuffling the Data

- Gradient descent is most efficient when the training set instances are independent and identically distributed. We can do this with the **shuffle()** method.
    - It will create a new dataset that will start by filling up a buffer with the first items of the source dataset.
    - Then, whenever asked for an item, it will pull one out randomly from the buffer and replace it with a fresh one from the source dataset until it has iterated entirely through the source dataset.

        - It will continue to pull out items randomly from the buffer until it is empty.
        - We must choose a sufficiently large buffer to make full use of shuffling. The buffer should not be too large; otherwise, we won't have enough RAM.
        - We can also specify a *seed* argument to maintain the same random order for the dataset.
- There may need to be more than a simple shuffle for a large dataset that fits in memory.
    - One solution is to shuffle the source data itself (for example, on Linux, you can shuffle text files using the **shuf** command)
    - Another approach is randomly picking multiple files and reading them simultaneously, interleaving their records.

Interleaving Lines from Multiple Lines

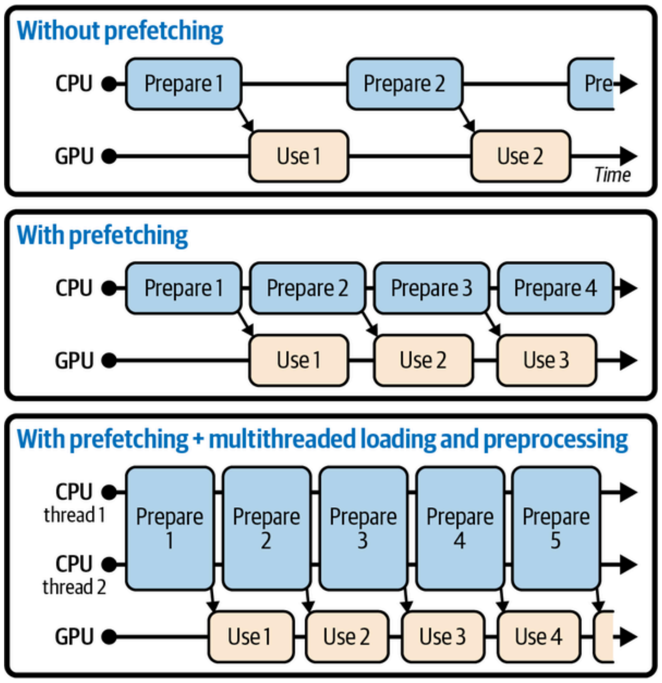
- First, we can load in a CSV file using the **list_files(x)** method, which returns a dataset that shuffles the file paths x.
- Then, we call the **interleave(x, cycle_length)** method, which reads cycle_length lines from the file paths using the function x.
    - We can use the **TextLineDataset()** function to create a new dataset from the file paths.
    - It is recommended that the files have similar lengths.
    - The interleave method can read lines in parallel by specifying the *num_parallel_calls* argument.
        - The **skip(x)** method skips the first x lines of each file (e.g., we can skip the header row of each file).
        - We can set this argument to **tf.data.AUTOTUNE** to make TF choose the right number of threads dynamically based on the available CPU.

Preprocessing the Data

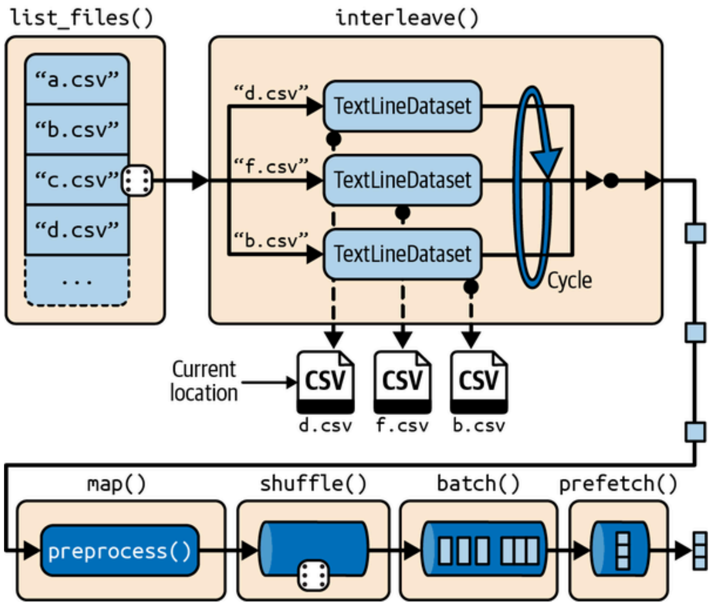
- We can write custom functions that preprocess the data.
    - We can use the **tf.io.decode_csv()** function to parse each line of the CSV file.
        - This function takes two arguments: the first indicates the line to parse, and the second contains each column's default value and type.
        - It returns a list of scalar tensors (one per column). If we need a 1D tensor array, we can use the **tf.stack()** method.
    - We will still need a custom function that reads each line of the file.

Prefetching

- While the training algorithm works on one batch, the dataset will already work in parallel to prepare the next batch (reading the data and preprocessing it).
- This diagram demonstrates the benefits of prefetching.

- ○ 

- ○ We can use multiple CPU cores to speed up the training algorithm.

- If a dataset is small enough to fit in memory, we can use its **cache()** method to cache its content to RAM, which speeds up training significantly.
- The entire loading and preprocessing procedure:
  - ○ 

  - ○ Some other methods of interest are the **concatenate()**, **zip()**, **window()**, **reduce()**, **shard()**, **flat_map()**, **apply()**, **unbatch()**, **padded_batch()**, **from_generator()**, and **from_tensors()**.

The TFRecord Format

- The TFRecord format is TF's preferred format for storing and reading large amounts of data efficiently.
  - We can create a TFRecord file using the **tf.io.TFRecordWriter()** class.
    - We can use the class's **write(x)** method to create a tensor with the contents x.
  - This class creates a **tf.data.TFRecordDataset** object, which we can use to read one or more TFRecord files.
  - By default, TFRecordDataset reads files individually, but we can specify the *num_parallel_reads* argument to make it read multiple files simultaneously.

Compressed TFRecord Files

- We can compress TFRecord files by setting the *options* argument.
  - Using the **tf.io.TFRecordOptions()**, we can specify a *compression_type* argument, such as GZIP, and pass this object into TFRecordWriter().
  - Also, we can create a compressed TFRecordDataset by setting its compression_type argument.

Protocol Buffers

- TFRecord files usually contain serialized protocol buffers (protobufs).
  - The protobufs definition looks similar to a Java class definition.
- We can create protobufs using the **tensorflow.train()** class.
  - Note: I had trouble loading this class into Python. I found that it is stored under a different name in my system. When in doubt, we can print the class name we want to use, which tells us where the class is located.
  - The main protobuf typically used in a TFRecord file is the **Example()** protobuf, representing one dataset instance.
  - Then, we can create a series of **Feature()** instances within the Example, which can contain **BytesList(), FloatList(),** or **Int64List()** classes, which specify the types of the protobuf's features.
  - Lastly, we can serialize the protobuf by calling its **SerializeToString()** method and write the resulting data to a TFRecord file.

Loading and Parsing Examples

- We can parse each Example using **tf.io.parse_single_example()**.
  - This method requires at least two arguments:
    - A string scalar tensor containing the serialized data.
    - A description of each feature.
      - This description is a dictionary that maps each feature to either a **tf.io.FixedLenFeature** or **tf.io.VarLenFeature** descriptor.
      - The FixedLenFeature descriptor indicates the feature's shape, type, and default value.
      - The VarLenFeature descriptor indicates only the type if the length of the feature's list may vary.

- A BytesList can contain any binary data.
    - For example, we can use **tf.io.encode_jpeg()** or **tf.io.decode_image()** to encode an image using a specific format. This is useful for an image classification system.

Keras Preprocessing Layers

The Normalization Layer

- We can normalize the training set before training.
    - First, we create a Normalization() layer. Then, we use its adapt() method on the training set.
    - After that, we can use the layer like a function and pass the training and validation set into it.
    - Thus, when we create the model, we don't have to create a normalization layer within the model.
- We can pass a tf.data.Dataset to a preprocessing layer's adapt() method.

The Discretization Layer

- A discretization layer transforms a numerical feature into a categorical one by mapping value ranges (bins) to categories.
    - The discretization layer is available in Keras's **layers.Discretization (x)** method, which takes in a range of values representing the bin boundaries.
    - This method also has *num_bins* argument, which lets the layer find the appropriate boundaries based on the value percentiles.
        - For example, if x is 3, the layer places the bin boundaries at the 33rd and 66th percentile.
        - In this case, we must use the adapt() method.
- Category identifiers should never be passed directly to a neural network since the network cannot meaningfully compare their values.
    - These identifiers should be encoded, such as using one-hot encoding.

The Category Encoding Layer

- Keras provides the **CategoryEncoding()** layer, which lets us apply one-hot encoding (or other encoding types).
    - If we try to encode more than one categorical feature at a time, the class will perform multi-hot encoding.
    - The output tensor will contain a 1 for each category present in any input feature.

The String Lookup Layer

- We can use Keras's **StringLookup()** layer to encode categorical text features.
    - We must use its adapt() method on a list of text features.
    - Each category is encoded as an integer.

    - Known categories are numbered starting with 1, from most frequent to least, and unknown categories are mapped to 0.
    - Setting the layer's *output_mode* argument to "one_hot" makes it output a one-hot vector for each category instead of an integer.
- For significantly large datasets, the layer's adapt() method may miss some rarer categories.
    - By default, it maps them to 0, making them indistinguishable by the model.
    - We can set the *num_oov_indices* argument to an integer greater than one.
        - This specifies the number of out-of-vocabulary (OOV) buckets.
        - Each unknown category will get mapped to one of the buckers using a hash function modulo the number of buckers.
        - This allows the model to distinguish some of the rare categories.
        - However, some categories may get mapped to the same bucket. This is called a hashing collision.
        - We can increase the number of buckets to reduce the number of collisions, but this also increases the amount of RAM.
        - Note that Keras provides a **Hasing()** layer, which implements hashing.

The Hashing Layer

- The Hashing layer computes a hash and modulates the number of buckets (or bins).
    - The mapping is pseudorandom but stable across runs and platforms (the same category will always be mapped to the same integer as long as the number of bins is unchanged).
    - The benefit of this layer is that we don't need to call its adap() method.

Encoding Categorical Features using Embeddings

- An embedding is a dense representation of some higher-dimensional data, such as a category or a word in a vocabulary.
- In deep learning, embeddings are usually initialized randomly, and they are then trained by gradient descent, along with the other model parameters.
- The better the representation, the easier it will be for the neural network to make accurate predictions.
- Keras provides an **Embedding()** layer, which wraps an embedding matrix: this matrix has one row per category and one column per embedding dimension.
    - To convert a category ID to an embedding, the layer finds and returns the row that corresponds to that category.
    - An Embedding layer is randomly initialized, so we shouldn't use it outside a model as a standalone preprocessing layer.
    - To embed a categorical text attribute, we can chain a StringLookup and Embedding layer.

Text Preprocessing

- Keras provides a **TextVectorization()** layer for basic text preprocessing.
    - We pass the training data into its adapt() method.

    - First, the layer converts the training sentences to lowercase and removes punctuation.
    - Then, the layer splits the sentences based on whitespace, and the resulting words are sorted by descending frequency, producing the final vocabulary.
    - We can set the *output_mode* as "tf_idf", which downweighs frequent words and weighs rare words.
        - We do this if the training data has several unmeaningful words, such as "the," and focus on rarer words.
        - TF-IDF stands for term-frequency x inverse-document-frequency.
        - The weight formula is $weight = log(1 + \frac{d}{(f+1)})$, where d is the total number of sentences and f counts how many of these sentences contain the given word.
    - Although TextVectorization() is useful, it only works with languages that separate words with spaces, and it doesn't distinguish between homonyms.

Using pre-trained Language Model Components

- The TensorFlow Hub library allows us to reuse pre-trained model components in our models for text, image, audio, etc. These model components are called <u>modules</u>.
- The modules are available in the **tensorflow_hub** library.
    - We provide a URL to the **hub.KerasLayer()** function, which downloads the model from the URL.

Image Preprocessing Layers

- Keras offers several image preprocessing layers.
    - **tf.keras.layers.Resizing()** resizes the input images to the desired size (height x width).
    - **tf.keras.layers.Rescaling()** rescales the pixel values.
    - **tf.keras.layers.CenterCrop()** crops the image, keeping only a center patch of the desired height and width.
- We can access image datasets with Scikit-Learn's **load_sample_images()**.

The TensorFlow Datasets Project

- The TDFS facilitates loading common datasets, including audio, image, text, video, time series, and other datasets.
- The datasets are available through the **tensorflow_datasets** library.
    - The library's **load()** function loads a dataset by providing it with the dataset's name, split parameters, etc.