Chapter 13: Memuat dan Memproses Data dengan TensorFlow

Pendahuluan

Meskipun arsitektur model dan algoritma pelatihan sering menjadi pusat perhatian, proses memuat dan memproses data (*data loading and preprocessing*) adalah bagian yang sangat krusial dan seringkali menjadi *bottleneck* (penghambat) dalam sebuah proyek Machine Learning. Terutama saat pelatihan dilakukan pada akselerator seperti GPU atau TPU, jika pipeline data tidak efisien, maka akselerator akan "kelaparan" (menunggu data) dan sumber daya komputasi yang mahal menjadi sia-sia. Chapter ini berfokus pada cara membangun pipeline data yang efisien, dapat diskalakan, dan portabel menggunakan alat-alat yang disediakan oleh TensorFlow.

1. The Data API (tf.data)

Untuk mengatasi masalah performa data, TensorFlow menyediakan **Data API (tf.data)**. API ini memungkinkan kita membangun pipeline input yang sangat efisien yang mampu memproses dataset dalam jumlah besar, bahkan yang tidak muat dalam memori.

Konsep Inti:

Inti dari tf.data API adalah objek tf.data.Dataset, yang merepresentasikan urutan item data (misalnya, baris dari file CSV atau gambar). Setelah dataset dibuat, kita dapat menerapkan serangkaian transformasi secara berantai (chaining) untuk memproses data.

Transformasi Utama dalam Pipeline Data:

1. **Creating a Dataset**: Langkah pertama adalah membuat objek Dataset dari sumber data, seperti dari tensor di memori, dari nama-nama file, atau langsung dari file teks (seperti CSV).
2. **map()**: Transformasi ini menerapkan fungsi yang kita tentukan pada setiap item dalam dataset. map() adalah tempat utama untuk melakukan parsing dan pra-pemrosesan, seperti mengubah baris teks menjadi tensor numerik atau melakukan augmentasi gambar. Proses ini dapat dipercepat secara signifikan dengan menjalankannya secara paralel.
3. **shuffle()**: Mengacak urutan item dalam dataset untuk memastikan bahwa model melihat data dalam urutan yang bervariasi setiap *epoch*, yang penting untuk konvergensi Gradient Descent. Metode ini bekerja dengan mengisi sebuah *buffer* dan mengambil sampel acak dari *buffer* tersebut.
4. **batch()**: Mengelompokkan item-item yang berurutan ke dalam *batch* (kelompok) dengan ukuran yang ditentukan. Pelatihan model hampir selalu dilakukan dalam *batch*.
5. **prefetch()**: Ini adalah transformasi optimisasi kinerja yang paling penting. prefetch(N) akan membuat pipeline data menyiapkan N *batch* berikutnya di latar belakang (menggunakan CPU) sementara GPU/TPU sedang sibuk melatih model pada *batch* saat ini. Ini memastikan bahwa akselerator tidak pernah perlu menunggu data, sehingga memaksimalkan utilisasi.
6. **repeat()**: Mengulang dataset untuk sejumlah *epoch* yang ditentukan.

Urutan transformasi yang efisien sangat penting. Praktik terbaik umumnya adalah mengacak nama file terlebih dahulu, kemudian membaca dan memproses beberapa file secara bersamaan (*interleaving*), baru kemudian melakukan pengacakan tingkat instance, *batching*, dan *prefetching*.

2. Format File TFRecord

Untuk dataset yang sangat besar, memproses dari ribuan file kecil bisa menjadi tidak efisien. TensorFlow merekomendasikan format file biner standar yang disebut **TFRecord**. TFRecord adalah format yang sangat efisien untuk menyimpan dan membaca data dalam jumlah besar.

Struktur TFRecord:

Sebuah file TFRecord pada dasarnya adalah urutan dari catatan biner (binary records). Setiap catatan berisi panjang data, CRC checksum untuk validasi integritas, dan data itu sendiri dalam bentuk biner. Format ini dioptimalkan untuk pembacaan sekuensial dan dapat dengan mudah dibaca secara paralel.

Protocol Buffers dan tf.train.Example

Data di dalam setiap catatan TFRecord biasanya diserialisasi menggunakan Protocol Buffers (protobuf), sebuah format serialisasi biner yang fleksibel dan efisien. Protobuf utama yang digunakan adalah tf.train.Example. Sebuah tf.train.Example pada dasarnya adalah sebuah kamus (map) di mana key adalah nama fitur (string) dan value adalah tf.train.Feature, yang dapat berisi salah satu dari tiga tipe daftar:

- FloatList: untuk nilai-nilai float.
- Int64List: untuk nilai-nilai integer.
- BytesList: untuk string atau data biner mentah (seperti gambar yang di-encode).

Dengan mengubah dataset menjadi format TFRecord, kita dapat meningkatkan kecepatan I/O (Input/Output) secara dramatis, yang sangat penting untuk menghilangkan *bottleneck* data.

3. Pra-pemrosesan Fitur (Feature Preprocessing)

Data mentah jarang bisa langsung digunakan oleh model. Ia perlu diubah menjadi representasi numerik yang sesuai.

One-Hot Encoding vs. Embedding

- **One-Hot Encoding**: Teknik standar untuk fitur kategorikal dengan jumlah kategori yang sedikit. Setiap kategori diubah menjadi vektor biner di mana hanya satu elemen yang bernilai 1.
- **Embedding**: Untuk fitur kategorikal dengan kosakata yang sangat besar (misalnya, jutaan kata atau ID produk), one-hot encoding menjadi tidak praktis. **Embedding** adalah solusi yang lebih baik. Embedding merepresentasikan setiap kategori sebagai sebuah vektor numerik yang padat (*dense vector*) dan berdimensi relatif rendah. Yang terpenting, vektor-vektor ini **dapat dilatih** (*trainable*). Selama pelatihan, model akan

belajar untuk menempatkan kategori-kategori yang mirip secara konseptual berdekatan satu sama lain di dalam ruang embedding.

4. Layer Pra-pemrosesan Keras (Keras Preprocessing Layers)

Secara tradisional, pra-pemrosesan data dilakukan sebelum data dimasukkan ke dalam model. Namun, pendekatan modern yang lebih disukai adalah dengan **memasukkan logika pra-pemrosesan ke dalam model itu sendiri** sebagai bagian dari arsitekturnya. Keras menyediakan serangkaian *layer* khusus untuk tujuan ini.

Keuntungan:

- **Portabilitas**: Model yang disimpan sudah mencakup semua logika pra-pemrosesan. Saat model digunakan di lingkungan produksi, kita tidak perlu menulis ulang logika pra-pemrosesan, cukup memberikan data mentah.
- **Menghindari Train/Serve Skew**: Ini mengurangi risiko perbedaan antara pra-pemrosesan saat pelatihan dan saat *serving* (produksi), yang merupakan sumber kesalahan yang umum.

Layer Pra-pemrosesan Utama:

- **Normalization**: Untuk menstandarisasi fitur numerik (mengurangkan *mean* dan membagi dengan standar deviasi). *Layer* ini dapat "beradaptasi" dengan data pelatihan untuk mempelajari *mean* dan standar deviasinya.
- **TextVectorization**: Menangani pra-pemrosesan teks, termasuk standardisasi, pemisahan menjadi token (kata), dan pemetaan setiap kata ke indeks integer.
- **Discretization**: Mengubah fitur numerik kontinu menjadi fitur kategorikal dengan membaginya ke dalam beberapa interval (*binning*).
- **Embedding**: *Layer* untuk mengubah indeks integer dari fitur kategorikal menjadi vektor embedding yang padat dan dapat dilatih.

Kesimpulan

Membangun pipeline input yang efisien adalah kunci untuk pelatihan model yang cepat dan dapat diskalakan. **Data API (tf.data)** menyediakan alat yang kuat untuk membaca, memproses, dan menyajikan data ke model. Menggabungkannya dengan format file yang dioptimalkan seperti **TFRecord** dapat memaksimalkan throughput data. Lebih lanjut, dengan mengintegrasikan logika pra-pemrosesan langsung ke dalam model menggunakan **layer pra-pemrosesan Keras**, kita dapat membangun model yang lebih sederhana, lebih portabel, dan lebih andal, mengurangi kesenjangan antara lingkungan penelitian dan produksi.