CHAPTER 19

Training and Deploying TensorFlow Models at Scale

Chapter 19 explains how to **serve, optimize, and scale** TensorFlow models in production, including TF Serving, cloud deployment, on-device inference, and distributed training with GPUs/TPUs and distribution strategies.

Serving models with TF Serving

Once a model is trained, it is exported as a **SavedModel**, a directory containing the computation graph (saved_model.pb), variables, and optional assets (e.g., vocab files), possibly with multiple tagged metagraphs for different usages (e.g., serve). SavedModels can be inspected with saved_model_cli, loaded via tf.saved_model.load() or keras.models.load_model(), and are the standard format consumed by TF Serving.

TensorFlow Serving is a high-performance C++ model server that:

- Hosts multiple models and versions, auto-reloading from a models directory, and supports **A/B testing** and canary deployments.
- Exposes **gRPC** and **REST** APIs (default ports 8500 and 8501) and can handle request batching and version routing.

A typical setup uses the official Docker image, mounts a host directory with versioned SavedModels (e.g., mymnistmodel/0001), and configures MODEL_NAME so TF Serving automatically serves the latest version.

REST and gRPC prediction APIs

For REST, clients send JSON like {"signature_name": "serving_default", "instances": [...]} to http://host:8501/v1/models/<name>:predict and receive predictions as JSON. For gRPC, clients use generated stubs to send PredictRequest protocol buffers, which are more efficient and better for high-QPS internal services.

On the client side, Keras models are usually exported with preprocessing layers included, so services can send raw inputs (e.g., images) rather than worrying about feature pipelines.

Edge deployment with TensorFlow Lite and JS

For **mobile and embedded** deployment, models are converted to **TensorFlow Lite** (flatbuffer format) using the TFLite Converter, which supports:

- **Post-training quantization** (e.g., weight-only or full integer), reducing size and latency at some accuracy cost.
- **Quantization-aware training**, inserting fake quantization nodes during training so the model learns to be robust to quantization noise.

For **web browsers**, **TensorFlow.js** can load Keras or SavedModels converted to TF.js format and run them with WebGL acceleration, enabling on-device inference directly in JavaScript without servers.

Using GPUs and executing across devices

TensorFlow places operations on devices automatically, preferring GPU 0 when available and falling back to CPU if no GPU kernel exists for an op. Device placement can be inspected with tensor.device and overridden via tf.device("GPU:0") or tf.device("CPU:0"); logging can be enabled with tf.debugging.set_log_device_placement(True). Multithreaded kernels and inter-op/intra-op thread pools exploit CPU cores and GPU threads in parallel.

Execution of tf.function graphs is **parallelized**: independent ops are scheduled based on dependency counts, dispatched to per-device queues, and run concurrently, while TensorFlow preserves program order for stateful updates (e.g., variable assign operations).

Data/model parallelism and distribution strategies

To scale training, the chapter contrasts:

- **Model parallelism**: splitting a single model across devices (rarely worth the complexity except for special architectures, e.g., very large CNNs or RNNs).
- **Data parallelism**: replicating the full model on each device, feeding each replica a different mini-batch, and aggregating gradients.

Two main data-parallel patterns:

- **Mirrored strategy**: parameters fully replicated on each GPU; each step, replicas compute gradients on their local batch, then an **AllReduce** operation (often NCCL-based) averages gradients, and each replica updates its local copy identically (synchronous SGD).
- **Centralized parameters**: parameters on CPU or parameter servers; workers compute gradients and either synchronize (barrier + average) or update asynchronously (risking **stale gradients** and instability).

Bandwidth limits become critical for large dense models, since parameter/gradient traffic can saturate network or PCIe; sparse models and reduced precision (e.g., bfloat16) scale better.

TensorFlow's **Distribution Strategies API** simplifies using these schemes:

- tf.distribute.MirroredStrategy: multi-GPU data parallelism on a single machine.
- tf.distribute.MultiWorkerMirroredStrategy: mirrored data parallelism across multiple workers in a cluster; workers are coordinated using collective ops (e.g., ring AllReduce, NCCL).
- tf.distribute.experimental.CentralStorageStrategy: parameters on CPU, workers on multiple GPUs of a single host.

- tf.distribute.experimental.ParameterServerStrategy: multi-worker setups with dedicated parameter servers; allows asynchronous or synchronous updates.
- tf.distribute.experimental.TPUStrategy: distribution over TPUs in Google Cloud.

Typical usage pattern:

python

```
strategy = tf.distribute.MirroredStrategy()
with strategy.scope():
    model = build_model()
    model.compile(...)
model.fit(dataset, batch_size=global_batch, ...)
```

The batch size must be divisible by the number of replicas because Keras automatically splits each batch per replica.

Clusters, AI Platform, and hyperparameter tuning

A multi-node **TensorFlow cluster** is described via the TF_CONFIG environment variable, which specifies roles (chief, worker, ps, evaluator) and their addresses. The same training code can then run on all tasks, and distribution strategies decide how to shard and synchronize work.

On **Google Cloud AI Platform**, users submit training jobs with gcloud ai-platform jobs submit training, specifying:

- Region and **scale tier** (e.g., number of workers and parameter servers).
- Runtime and Python version.
- Package path and main module.
- Job directory on GCS for checkpoints and final models.

AI Platform also supports **hyperparameter tuning** (black-box optimization) by running many trials with different hyperparameter combinations and reading metrics (e.g., accuracy) from TensorBoard event files in the job directory.