1. What does a SavedModel contain? How do you inspect its content?

A SavedModel contains a complete TensorFlow program, including trained parameters (i.e, tf. Variable s) and computation. It does not require the original model building code to run, which makes it useful for sharing or deploying with TFLite, TensorFlow. js, TensorFlow Serving, or TensorFlow Hub.

A SavedModel directory has the following structure:

assets/

assets.extra/

variables/

variables.data-?????-of-?????

variables.index

saved\_model.pb

SavedModel protocol buffer

[saved\_model.pb](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/protobuf/saved_model.proto) or saved\_model.pbtxt

Includes the graph definitions as MetaGraphDef protocol buffers.

Assets

Subfolder called assets.

Contains auxiliary files such as vocabularies, etc.

Extra assets

Subfolder where higher-level libraries and users can add their own assets that co-exist with the model, but are not loaded by the graph.

This subfolder is not managed by the SavedModel libraries.

Variables

Subfolder called variables.

variables.data-?????-of-?????

variables.index

SavedModel in TensorFlow 1.x

SavedModel had slightly different semantics in TF 1.x. Conventions that are generally only supported in TF 1.x are noted as such.

Features

The following is a summary of the features in SavedModel:

(TF1-only) Multiple graphs sharing a single set of variables and assets can be added to a single SavedModel. Each graph is associated with a specific set of tags to allow identification during a load or restore operation.

(TF1-only) Support for SignatureDefs

Graphs that are used for inference tasks typically have a set of inputs and outputs. This is called a Signature.

SavedModel uses [SignatureDefs](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/protobuf/meta_graph.proto) to allow generic support for signatures that may need to be saved with the graphs.

For commonly used SignatureDefs in the context of TensorFlow Serving, please see documentation [here](https://github.com/tensorflow/serving/blob/master/tensorflow_serving/g3doc/signature_defs.md).

Support for Assets.

For cases where ops depend on external files for initialization, such as vocabularies, SavedModel supports this via assets.

Assets are copied to the SavedModel location and can be read when loading a specific meta graph def.

Support to clear devices before generating the SavedModel.

The following is a summary of features that are NOT supported in SavedModel. Higher-level frameworks and tools that use SavedModel may provide these.

Implicit versioning.

Garbage collection.

Atomic writes to the SavedModel location.

TF1 SavedModel Background

SavedModel manages and builds upon existing TensorFlow primitives such as TensorFlow Saver and MetaGraphDef. Specifically, SavedModel wraps a [TensorFlow Saver](https://github.com/tensorflow/tensorflow/tree/master/tensorflow/python/training/saver.py). The Saver is primarily used to generate the variable checkpoints. SavedModel will replace the existing [TensorFlow Inference Model Format](https://github.com/tensorflow/tensorflow/tree/r1.15/tensorflow/contrib/session_bundle#tensorflow-inference-model-format) as the canonical way to export TensorFlow graphs for serving.

APIs

The APIs for building and loading a SavedModel are described in this section.

(TF1-only) Builder

The SavedModel [builder](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/saved_model/builder.py) is implemented in Python.

The SavedModelBuilder class provides functionality to save multiple meta graph defs, associated variables and assets.

To build a SavedModel, the first meta graph must be saved with variables. Subsequent meta graphs will simply be saved with their graph definitions. If assets need to be saved and written or copied to disk, they can be provided when the meta graph def is added. If multiple meta graph defs are associated with an asset of the same name, only the first version is retained.

(TF1-only) Tags

Each meta graph added to the SavedModel must be annotated with user specified tags, which reflect the meta graph capabilities or use-cases. More specifically, these tags typically annotate a meta graph with its functionality (e.g. serving or training), and possibly hardware specific aspects such as GPU. In the SavedModel, the meta graph def whose tag-set exactly matches those specified in the loader API, will be the one loaded by the loader. If no meta graph def is found matching the specified tags, an error is returned. For example, a loader with a requirement to serve on GPU hardware would be able to load only meta graph annotated with tags='serve,gpu' by specifying this set of tags in tensorflow::LoadSavedModel(...).

Usage

The typical usage of builder is as follows:

export\_dir = ...

...

builder = tf.saved\_model.builder.SavedModelBuilder(export\_dir)

with tf.Session(graph=tf.Graph()) as sess:

...

builder.add\_meta\_graph\_and\_variables(sess,

[tf.saved\_model.tag\_constants.TRAINING],

signature\_def\_map=foo\_signatures,

assets\_collection=foo\_assets)

...

with tf.Session(graph=tf.Graph()) as sess:

...

builder.add\_meta\_graph(["bar-tag", "baz-tag"])

...

builder.save()

(TF1-only) Stripping Default valued attributes

The SavedModelBuilder class allows users to control whether default-valued attributes must be stripped from the NodeDefs while adding a meta graph to the SavedModel bundle. Both SavedModelBuilder.add\_meta\_graph\_and\_variables and SavedModelBuilder.add\_meta\_graph methods accept a Boolean flag strip\_default\_attrs that controls this behavior.

If strip\_default\_attrs is False, the exported MetaGraphDef will have the default valued attributes in all it's NodeDef instances. This can break forward compatibility with a sequence of events such as the following:

An existing Op (Foo) is updated to include a new attribute (T) with a default (bool) at version 101.

A model producer (such as a Trainer) binary picks up this change (version 101) to the OpDef and re-exports an existing model that uses Op Foo.

A model consumer (such as Tensorflow Serving) running an older binary (version 100) doesn't have attribute T for Op Foo, but tries to import this model. The model consumer doesn't recognize attribute T in a NodeDef that uses Op Foo and therefore fails to load the model.

By setting strip\_default\_attrs to True, the model producers can strip away any default valued attributes in the NodeDefs. This helps ensure that newly added attributes with defaults don't cause older model consumers to fail loading models regenerated with newer training binaries.

TIP: If you care about forward compatibility, then set strip\_default\_attrs to True while using SavedModelBuilder.add\_meta\_graph\_and\_variables and SavedModelBuilder.add\_meta\_graph.

2.When should you use TF Serving? What are its main features? What are some tools you can use to deploy it?

TensorFlow Serving is a flexible, high-performance serving system for machine learning models, designed for production environments. TensorFlow Serving makes it easy to deploy new algorithms and experiments, while keeping the same server architecture and APIs. TensorFlow Serving provides out-of-the-box integration with TensorFlow models, but can be easily extended to serve other types of models and data.

📖 Introduction

Currently there are a lot of different solutions to serve ML models in production with the growth that MLOps is having nowadays as the standard procedure to work with ML models during all their lifecycle. Maybe the most popular one is [TensorFlow Serving](https://www.tensorflow.org/tfx/guide/serving) developed by TensorFlow so as to server their models in production environments.

This post is a guide on how to train, save, serve and use TensorFlow ML models in production environments. Along the GitHub repository linked to this post we will prepare and train a custom CNN model for image classification of [The Simpsons Characters Data dataset](https://www.kaggle.com/alexattia/the-simpsons-characters-dataset), that will be later deployed using [TensorFlow Serving](https://www.tensorflow.org/tfx/guide/serving).

So as to get a better understanding on all the process that is presented in this post, as a personal recommendation, you should read it while you check the resources available in the repository, as well as trying to reproduce it with the same or with a different TensorFlow model, as “practice makes the master”.

Requirements

First of all you need to make sure that you have all the requirements installed, but before proceeding you need to know that TF-Serving is just available for Ubuntu, which means that, in order to use it you will either need a Ubuntu VM or just Docker installed in your OS so as to run a Docker container which deploys TF-Serving.

So, from your Ubuntu VM you need to install tensorflow-model-server, but before being able to install it you need to add the TF-Serving distribution URI as a package source as it follows:

And then you will be able to install tensorflow-model-server using APT-GET as it follows:

apt-get update && apt-get install tensorflow-model-server

Finally, for the client side you need to install the Python package tensorflow-serving-api, which is useful towards using the gRPC API; and, tensorflow. Note that the versions of both packages should match.

pip install tensorflow-serving-api==2.4.1  
pip install tensorflow==2.4.1

If you have any problems regarding the TensorFlow installation, visit [Installation | TensorFlow](https://www.tensorflow.org/install?hl=es-419).

Dataset

The dataset that is going to be used to train the image classification model is “[The Simpsons Characters Data](https://www.kaggle.com/alexattia/the-simpsons-characters-dataset)”, which is a big Kaggle dataset that contains RGB images of some of the main The Simpsons characters including Homer, Marge, Bart, Lisa, Maggie, Barney, and much more.

The original dataset contains 42 classes of The Simpsons characters, with an unbalanced number of samples per class, and a total of 20,935 training images and 990 test images in JPG format, and the images in different sizes, but as all of them are small, we will be resizing them to 64x64px when training the model.

Anyway, we will create a custom slice of the original dataset keeping just the training set, and using a random 80/20 train-test split and removing the classes with less than 50 images. So on, we will be have 32 classes, with 13,210 training images, 3,286 validation images, and 4,142 testing images.

Source: Random images of the 10 most populated classes of The Simpsons Characters Dataset

Modelling

Once the data has been explored, we are going to proceed with the definition of the ML model, that in this case will be a CNN (Convolutional Neural Network) as we are facing an image classification problem.

The created model architecture consists on an initial Conv2D layer (that also indicates the input\_shape of the net), which is a 2D convolutional layer that produces 16 filters as output of windows of 3x3 convolutions, followed by a MaxPooling2D in order to downsample the Tensor resulting from the previous convolutional layer. Usually, you will find this layer after two consecutive convolutions, but for the sake of simplicity, here we will be downsampling the data after each convolution, as this is a simple CNN with a relatively small dataset (less than 20k images).

Then we will include another combination of Conv2D and MaxPooling2D layers as increasing the number of convolutional filters means that we will provide more data to the CNN as it is capturing more combinations of pixel values from the input image Tensor.

After applying the convolutional operations, we will include a Flatten layer in order to transform the image Tensor into a 1D Tensor which prepares the data that goes through the CNN so as to include a few fully connected layers after it.

Finally, we will include some Dense fully connected layers so as to assign the final weights of the net, and some Dropout layers to avoid overfitting during the training phase. You also need to take into consideration that the latest Dense layer contains as much units as the total labels to predict, which in this case is the number of The Simpsons Characters available in the training set.

The trained model has been named SimpsonsNet (this name will be used later while serving the model as its identifier) and its architecture looks like:

Finally, once trained we will need to dump the model in SavedModel format, which is the universal serialization format for the TensorFlow models. This format provides a language-neutral format to save ML models that is recoverable and hermetic. It enables higher-level systems and tools to produce, consume and transform TensorFlow models.

3.How do you deploy a model across multiple TF Serving instances?

The final mile in any machine learning project is deployment of the solution so that it can do what it was created to do: improve the lives of people. Once our models have been trained and we are satisfied with the model accuracy, the next thing is to deploy. And if our intention is to deploy into production at scale, then TensorFlow Serving on GPUs is currently the way to go. Additionally, depending on the complexity of the data science solution, multiple models are often used in ensemble or in linear cascade. Here I discuss how to:

Setup a TensorFlow model Server on a GPU-enabled machine

Host multiple models on the server simultaneously, and

Send image classification requests to the server from a RESTful API python client.

A) BASIC INSTALLATIONS:

To use GPUs you cannot use garden variety docker, but will instead need to install a version of nvidia-docker (1 or 2). I strongly recommend nvidia-docker 2. Of note, nvidia-docker has been deprecated and is already having compatibility issues with TF versions etc. If you have nvidia-docker on your machine, you will need to remove it with the following code (If you don’t have nvidia-docker already installed then skip to next block). The following is the nvidia-docker 2.0 installation process for Ubuntu 18.04 operating system which is what I use:  
  
$ distribution=$(. /etc/os-release;echo $ID$VERSION\_ID)  
  
$ curl -s -L https://nvidia.github.io/nvidia-docker/$distribution/nvidia-docker.list | \  
sudo tee /etc/apt/sources.list.d/nvidia-docker.list  
  
$ sudo apt-get update  
$ sudo apt-get install -y nvidia-docker2  
$ sudo pkill -SIGHUP dockerd

NEXT Run NVIDIA-SMI to verify that everything works, and to ensure compatibility between CUDA version and nvidia-driver version. On my machine I got the following:

On my system, I had to particularly install a driver (430.26) that was compatible with my CUDA version 10.2. The table below shows what nvidia driver versions are compatible with what CUDA versions:

B) SETUP TENSORFLOW SERVING

Obtain latest tensorflow/serving docker image for gpu by the following command:

$ docker pull tensorflow/serving:latest-gpu

Next, clone the tensorFlow serving repository as follows:

The above code can be understood as follows, piece-by-piece: It instructs for gpu usage (--runtime=nvidia) and opens up a port 8501 designated for RESTful messaging. It then binds the docker container's 8501 port to the host machine's 8501 port (-p 8501:8501). Next the location of the ML model on the host machine is bound to the location to which it will be copied onto in the docker container (--mount type=bind,source=<model location on host>,target=<model location in container>). Next the environmental variable representing the model name is explicitly changed to whatever name we've chosen to call our model (-e MODEL\_NAME=<model name>). And finally we specify that we are running tensorFlow serving (-t tensorflow/serving:latest-gpu). Notably, each of the pieces in the code have default settings which we should be aware of.

E) MODEL SERVER FOR MULTIPLE MODELS

The Model\_Config\_File is the key ingredient needed to setup a tensorFlow server that can hold multiple models in the same docker container and serve through a common port.

There are a number of ways to implement this. My preferred approach is to launch the server de novo in such a manner that all ports and resources are bound and wired at launch time. This then enables us use essentially the same python client we used for the single model case. This approach is therefore both easier to set-up and easier to use. The alternative approach requires a gRPC client, and involves several more steps at set-up.

F) PYTHON CLIENT FOR MULTIPLE MODEL

The python client for multiple model server is essentially identical to that for single model. The only difference is that you specifically call the model you want by indicating it in the URL of the request.

SUCCESS! Together, we have setup a tensorFlow server on a GPU-enabled host machine; we’ve hosted two different machine learning models on the server and exposed a single RESTful port, 8501; we queried each of the ML models on the server about the class of an image; and finally, we received prediction responses which we successfully examined.

4.When should you use the gRPC API rather than the REST API to query a model served by TF Serving?

REST dominates the modern API landscape, especially when it comes to web applications and microservices-based infrastructures. However, REST isn’t the only API architecture available, and for a certain set of use-cases, the gRPC model has begun to play a small but crucial role.

Whether you’re trying to figure out what “gRPC” means — or you’re considering gRPC as an alternative to REST APIs for your next development project — this guide will help you understand what gRPC is, why people use it, and how a gRCP API compares to a RESTful API.

Overview of REST vs. gRPC

To understand REST and gRPC, we need to start with APIs (application programming interfaces). APIs provide rules and definitions that allow applications to communicate and interact with each other. An API defines the types of calls and requests that one application can make to another, how to make those requests, the data formats to be used, and the conventions that clients must follow.

APIs also support the “pluggability” of applications that form a larger system because they allow two applications – even if they were written in different programming languages and running on different platforms – to communicate and interact with each other.

REST APIs and gRPC APIs refer to different architectural styles for building APIs. Here’s a brief definition of both:

REST (Representational State Transfer) API: REST is the most popular architectural style for building APIs, particularly for web-based applications and microservices-based infrastructures. REST defines specific constraints that support interoperability between microservices and internet-based applications. Although a REST API can receive and return messages written in a variety of formats, the most common format used is JSON. JSON is a text-based, human-readable format that is flexible, efficient, and language/platform agnostic.

 gRPC (Google Remote Procedure Call): gRPC is an open-source RPC architecture designed by Google to achieve high-speed communication between microservices. gRPC allows developers to integrate services programmed in different languages. gRPC uses the Protobuf (protocol buffers) messaging format, which is a highly-packed, highly-efficient messaging format for serializing structured data. For a specific set of use-cases, a gRPC API can serve as a more efficient alternative to a REST API (more on this later).

Here’s a simple matrix that compares the basics of REST APIs and gRPC:

|  |  |  |
| --- | --- | --- |
| Characteristic | gRPC | REST API |
| HTTP Protocol | HTTP 2 | HTTP 1.1 |
| Messaging Format | Protobuf (Protocol Buffers) | JSON (usually) or XML and others |
| Code Generation | Native Protoc Compiler | Third-Party Solutions Like Swagger |
| Communication | Unary Client-Request or Bidirectional/Streaming | Client-Request Only |
| Implementation Time | 45 Minutes | 10 Minutes |

Understanding Microservices, REST APIs and RPC APIs

One of the best ways to understand the topic of APIs is to view APIs in the context of modern, microservices-based application development. Not only do APIs make microservices-based applications possible, but it’s in the context of microservices-based applications that gRPC APIs can — in some cases — serve as an alternative to REST APIs.

What Are Microservices-Based Applications?

Microservices-based applications overcome the biggest limitations of traditional, monolithic applications. A monolithic application contains the programming for all of its services and features within a single, indivisible code-base that manages all of the services and features of the application.

The problem is that — as developers bolt new services and features on top of the existing framework — it becomes increasingly difficult to modify, upgrade, and scale the application. Changing one part of the app can negatively impact other areas. After scaling, updating, and changing a monolith multiple times, the codebase eventually becomes so interdependent and difficult to understand that it’s necessary to redesign the entire system from scratch.

A microservices-based application architecture resolves this problem. This architectural style involves breaking a monolith into its component services and running each component as an autonomous application (called a microservice). These individual microservices then use APIs to interact and interact with each other. Together, this API-connected group of microservices forms the larger application architecture.

Because of the way APIs allow autonomously-running microservices — even those coded in different languages and running on different platforms — to connect with each other, APIs allow you to achieve a pluggable, component-based system. Upgrading individual microservices is dramatically faster and easier because the changes you make to an autonomously-running service have less impact on the entire system. Scaling is easier and more efficient because resources can be diverted to the microservices that need it based on usage demands. Moreover, if one microservice fails or slows down, it’s less likely to bring down the entire infrastructure. All of this translates into more efficient, resilient, scalable, and flexible systems — and APIs are what make it all possible.

The most widely-used architectural style for APIs is the REST API. However, there are also RPC APIs and gRPC APIs. In the next sections, we’ll look at how these APIs compare to each other.

REST APIs

REST (Representational State Transfer) describes a client-server organization in which back-end data is made available to clients through the JSON or XML messaging format. According to[Roy Fielding](https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm), an API qualifies as “RESTful” when it meets the following constraints:

A uniform interface: An API must expose specific application resources to API consumers.

Client-server independence: The client and the server function independently. The client will only know the URIs that point to the application’s resources. These are usually published in the API documentation.

Stateless: The server doesn’t save any data pertaining to the client request. The client saves this “state data” on its end (via a cache). Learn more about stateful vs. stateless systems here.

Cacheable: Application resources exposed by the API need to be cacheable.

Layered: The architecture is layered, which allows different components to be maintained on different servers.

Code-on-Demand (COD): This is the only optional REST constraint. This allows the client to receive executable code as a response from the server. In other words, it’s the server that determines how specific things get done.

Finally, the REST API architecture generally relies on HTTP protocol, and REST APIs are the most common format for building web applications and connecting microservices. When a REST API is made publicly available as a web service, each component (or service) provided by the web service is presented to clients as a resource. Clients can access these resources via a common interface that accepts different HTTP commands like GET, POST, DELETE, and PUT.

RPC APIs

As a predecessor of REST, RPC (Remote Procedure Call) is a software architecture dating back to the 1970s. RPC allows you to invoke a function on a remote server in a particular format and receive a response in the same format. It doesn’t matter what format the server executing the request uses, and it doesn’t matter if it’s a local server or a remote server. RPC allows you to invoke a function on the server and receive the result in the same format.

The basic concept of an RPC API is similar to that of a REST API. The RPC API defines the rules of interaction and what methods a client can use to interact with it. Clients submit calls that use “arguments” to invoke these methods. However, with an RPC API, the method is found in the URL. The arguments that invoke the methods are found in the query string. To illustrate this, here’s how an RPC API request compares to a REST API request:

RPC: An RPC API request might use POST /deleteResource and have a query string that says { “id”: 3 }

REST: A REST API request would write this request as DELETE /resource/2.

Understanding gRPC APIs

As a variant of the RPC architecture, gRPC was created by Google to speed up data transmission between microservices and other systems that need to interact with each other. Compared to REST APIs, gRPC APIs are unique in the following ways:

Protobuf Instead of JSON

Built on HTTP 2 Instead of HTTP 1.1

In-Born Code Generation Instead of Using Third-Party Tools Like Swagger

7 to 10 times Faster Message Transmission

Slower Implementation than REST

Let’s take a closer look at each of these differences between REST and gRPC APIs.

(1) Protobuf Instead of JSON/XML

Both REST APIs and RPC APIs send and receive messages using the JSON or XML messaging formats. They can use other formats too, but JSON and XML are the most common. Of these, JSON has become the most popular format because it is flexible, efficient, platform neutral, and language agnostic. It’s also text-based and human-readable, which makes it easy for human operators to work with. The problem is that for certain use-cases, JSON isn’t fast enough or light-weight enough when transmitting data between systems.

In contrast to REST and RPC, gRPC overcomes issues related to speed and weight — and offers greater efficiency when transmitting messages — by using the [Protobuf](https://medium.com/better-programming/understanding-protocol-buffers-43c5bced0d47) (protocol buffers) messaging format. Here are a few details about Protobuf:

Platform and language agnostic like JSON

Serializes and deserializes structured data to communicate via binary.

As a highly-compressed format, it doesn’t achieve JSON’s level of human-readability.

Speeds up data transmission by removing a lot of the responsibilities that JSON manages so it can focus strictly on serializing and deserializing data.

Data transmission is faster because Protobuf reduces the size of messages and serves as a lightweight messaging format.

(2) Built on HTTP 2 Instead of HTTP 1.1

Another way that gRPC boosts efficiency is through its use of the HTTP 2 protocol.

REST APIs are usually built on HTTP 1.1, which uses a request-response model of communication. This means that when a microservice receives multiple requests from more than one client, it has to serve them one at a time, which slows the entire system. REST APIs can also use HTTP 2, they are still limited to the request-response model and don’t take advantage of HTTP 2 support for bidirectional, streaming communication.

In contrast, gRPC uses HTTP 2 and it takes advantage of HTTP 2 support for both client-response communication and bidirectional communication. As such, gRPC can manage “Unary” interactions that are similar to HTTP 1.1 (where the client sends one request and the server sends one response). At the same time, clients can also open long-lived connections where each RPC call opens a new HTTP 2 stream — also known as bidirectional, “multiplexing,” or streaming communication.

In HTTP 2, when a microservice receives multiple requests from more than one client, it achieves multiplexing by serving many requests and responses simultaneously. In this respect, gRPC APIs depart from the limitations of REST APIs in their capacity to stream information constantly.

There are three types of streaming that gRPC makes available:

Server-side: A client sends a request message to a server. The server returns a stream of responses back to the client. After completing the responses, the server sends a status message (and in some cases trailing metadata), which completes the process. After receiving all of the responses, the client completes its process.

Client-side: A client sends a stream of request messages to a server. The server returns one response back to the client. It (usually) sends the response after receiving all of the requests from the client and a status message (and in some cases trailing metadata).

Bidirectional: A client and server transmit data to one another in no special order. The client is the one that initiates this kind of bidirectional streaming. The client also ends the connection.

(3) In-Born Code Generation Instead of Using Third-Party Tools

Code generation features are native to gRPC via its in-built protoc compiler. With REST APIs, it’s necessary to use a third-party tool such as Swagger to auto-generate the code for API calls in various languages.

The protoc compiler that comes with gRPC is compatible with a wide range of programming languages. This makes gRPC excellent for polyglot environments — where you are connecting a lot of different microservices that are coded in different languages and running on different platforms.

In contrast, REST API does not offer native code generation features. You have to use a third-party tool like Swagger to generate code for API calls in different languages. This isn’t an inconvenience, but it’s worth noting that gRPC doesn’t depend on any external tools for code generation.

(4) 7 to 10 Times Faster Message Transmission

According to [widely-cited tests published by Ruwan Fernando](https://medium.com/@EmperorRXF/evaluating-performance-of-rest-vs-grpc-1b8bdf0b22da#:~:text=gRPC%20is%20roughly%207%20times,of%20HTTP%2F2%20by%20gRPC.), gRPC API connections are considerably faster than REST API connections. In fact, he reported that they are 7 to 10 times faster:

“gRPC is roughly 7 times faster than REST when receiving data & roughly 10 times faster than REST when sending data for this specific payload. This is mainly due to the tight packing of the Protocol Buffers and the use of HTTP/2 by gRPC.”

(5) Slower Implementation than REST

Despite the benefits in message transmission speed, gRPC API implementation is a great deal slower than REST API implementation. [According to Ruan Fernando](https://medium.com/@EmperorRXF/evaluating-performance-of-rest-vs-grpc-1b8bdf0b22da#:~:text=gRPC%20is%20roughly%207%20times,of%20HTTP%2F2%20by%20gRPC.), it takes approximately 45 minutes to implement a simple gRPC Service. It only takes about 10 minutes to implement a Web or REST API.

Fernando reports that the additional implementation time reflects the lack of in-built support for gRPC in third-party tools. This is primarily because gRPC is not yet widely adopted, especially compared to the ubiquity of REST APIs. Here’s [what Fernando says](https://medium.com/@EmperorRXF/evaluating-performance-of-rest-vs-grpc-1b8bdf0b22da#:~:text=gRPC%20is%20roughly%207%20times,of%20HTTP%2F2%20by%20gRPC.) about gRPC implementation time:

“I had to spend roughly 45 mins implementing this simple gRPC Service, where I only spent around 10 mins building the WebAPI. This is mainly due to REST becoming mainstream a long time back, and most major frameworks (i.e. ASP.NET Core MVC) having built-in support to quickly spin up such services (through convention & patterns).”

5.What are the different ways TFLite reduces a model’s size to make it run on a mobile or embedded device?

TensorFlow Lite

TensorFlow Lite is a set of tools that enables on-device machine learning by helping developers run their models on mobile, embedded, and edge devices.

Key features

Optimized for on-device machine learning, by addressing 5 key constraints: latency (there's no round-trip to a server), privacy (no personal data leaves the device), connectivity (internet connectivity is not required), size (reduced model and binary size) and power consumption (efficient inference and a lack of network connections).

Multiple platform support, covering [Android](https://www.tensorflow.org/lite/android) and [iOS](https://www.tensorflow.org/lite/guide/ios) devices, [embedded Linux](https://www.tensorflow.org/lite/guide/python), and [microcontrollers](https://www.tensorflow.org/lite/microcontrollers).

Diverse language support, which includes Java, Swift, Objective-C, C++, and Python.

High performance, with [hardware acceleration](https://www.tensorflow.org/lite/performance/delegates) and [model optimization](https://www.tensorflow.org/lite/performance/model_optimization).

End-to-end [examples](https://www.tensorflow.org/lite/examples), for common machine learning tasks such as image classification, object detection, pose estimation, question answering, text classification, etc. on multiple platforms.

1. Generate a TensorFlow Lite model

A TensorFlow Lite model is represented in a special efficient portable format known as [FlatBuffers](https://google.github.io/flatbuffers/) (identified by the .tflite file extension). This provides several advantages over TensorFlow's protocol buffer model format such as reduced size (small code footprint) and faster inference (data is directly accessed without an extra parsing/unpacking step) that enables TensorFlow Lite to execute efficiently on devices with limited compute and memory resources.

A TensorFlow Lite model can optionally include metadata that has human-readable model description and machine-readable data for automatic generation of pre- and post-processing pipelines during on-device inference. Refer to [Add metadata](https://www.tensorflow.org/lite/models/convert/metadata) for more details.

You can generate a TensorFlow Lite model in the following ways:

Use an existing TensorFlow Lite model: Refer to [TensorFlow Lite Examples](https://www.tensorflow.org/lite/examples) to pick an existing model. Models may or may not contain metadata.

Create a TensorFlow Lite model: Use the [TensorFlow Lite Model Maker](https://www.tensorflow.org/lite/models/modify/model_maker) to create a model with your own custom dataset. By default, all models contain metadata.

Convert a TensorFlow model into a TensorFlow Lite model: Use the [TensorFlow Lite Converter](https://www.tensorflow.org/lite/models/convert/) to convert a TensorFlow model into a TensorFlow Lite model. During conversion, you can apply [optimizations](https://www.tensorflow.org/lite/performance/model_optimization) such as [quantization](https://www.tensorflow.org/lite/performance/post_training_quantization) to reduce model size and latency with minimal or no loss in accuracy. By default, all models don't contain metadata.

2. Run Inference

Inference refers to the process of executing a TensorFlow Lite model on-device to make predictions based on input data. You can run inference in the following ways based on the model type:

Models without metadata: Use the [TensorFlow Lite Interpreter](https://www.tensorflow.org/lite/guide/inference) API. Supported on multiple platforms and languages such as Java, Swift, C++, Objective-C and Python.

Models with metadata: You can either leverage the out-of-box APIs using the [TensorFlow Lite Task Library](https://www.tensorflow.org/lite/inference_with_metadata/task_library/overview) or build custom inference pipelines with the [TensorFlow Lite Support Library](https://www.tensorflow.org/lite/inference_with_metadata/lite_support). On android devices, users can automatically generate code wrappers using the [Android Studio ML Model Binding](https://www.tensorflow.org/lite/inference_with_metadata/codegen#mlbinding) or the [TensorFlow Lite Code Generator](https://www.tensorflow.org/lite/inference_with_metadata/codegen#codegen). Supported only on Java (Android) while Swift (iOS) and C++ is work in progress.

On Android and iOS devices, you can improve performance using hardware acceleration. On either platforms you can use a [GPU Delegate](https://www.tensorflow.org/lite/performance/gpu), on android you can either use the [NNAPI Delegate](https://www.tensorflow.org/lite/android/delegates/nnapi) (for newer devices) or the [Hexagon Delegate](https://www.tensorflow.org/lite/android/delegates/hexagon) (on older devices) and on iOS you can use the [Core ML Delegate](https://www.tensorflow.org/lite/performance/coreml_delegate). To add support for new hardware accelerators, you can [define your own delegate](https://www.tensorflow.org/lite/performance/implementing_delegate).

6.What is quantization-aware training, and why would you need it?

Quantization aware training emulates inference-time quantization, creating a model that downstream tools will use to produce actually quantized models. The quantized models use lower-precision (e.g. 8-bit instead of 32-bit float), leading to benefits during deployment.

There are two forms of quantization: post-training quantization and quantization aware training. Start with [post-training quantization](https://www.tensorflow.org/model_optimization/guide/quantization/post_training) since it's easier to use, though quantization aware training is often better for model accuracy.

This page provides an overview on quantization aware training to help you determine how it fits with your use case.

To dive right into an end-to-end example, see the [quantization aware training example](https://www.tensorflow.org/model_optimization/guide/quantization/training_example).

To quickly find the APIs you need for your use case, see the [quantization aware training comprehensive guide](https://www.tensorflow.org/model_optimization/guide/quantization/training_comprehensive_guide).

Overview

Quantization aware training emulates inference-time quantization, creating a model that downstream tools will use to produce actually quantized models. The quantized models use lower-precision (e.g. 8-bit instead of 32-bit float), leading to benefits during deployment.

Deploy with quantization

Quantization brings improvements via model compression and latency reduction. With the API defaults, the model size shrinks by 4x, and we typically see between 1.5 - 4x improvements in CPU latency in the tested backends. Eventually, latency improvements can be seen on compatible machine learning accelerators, such as the [EdgeTPU](https://coral.ai/docs/edgetpu/benchmarks/) and NNAPI.

The technique is used in production in speech, vision, text, and translate use cases. The code currently supports a [subset of these models](https://www.tensorflow.org/model_optimization/guide/quantization/training#general_support_matrix).

Experiment with quantization and associated hardware

Users can configure the quantization parameters (e.g. number of bits) and to some degree, the underlying algorithms. Note that with these changes from the API defaults, there is currently no supported path for deployment to a backend. For instance, TFLite conversion and kernel implementations only support 8-bit quantization.

APIs specific to this configuration are experimental and not subject to backward compatibility.

API compatibility

Users can apply quantization with the following APIs:

Model building: [tf.keras](https://www.tensorflow.org/api_docs/python/tf/keras) with only Sequential and Functional models.

TensorFlow versions: TF 2.x for tf-nightly.

[tf.compat.v1](https://www.tensorflow.org/api_docs/python/tf/compat/v1) with a TF 2.X package is not supported.

TensorFlow execution mode: eager execution

It is on our roadmap to add support in the following areas:

Model building: clarify how Subclassed Models have limited to no support

Distributed training: [tf.distribute](https://www.tensorflow.org/api_docs/python/tf/distribute)

General support matrix

Support is available in the following areas:

Model coverage: models using [allowlisted layers](https://github.com/tensorflow/model-optimization/tree/master/tensorflow_model_optimization/python/core/quantization/keras/default_8bit/default_8bit_quantize_registry.py), BatchNormalization when it follows Conv2D and DepthwiseConv2D layers, and in limited cases, Concat.

Hardware acceleration: our API defaults are compatible with acceleration on EdgeTPU, NNAPI, and TFLite backends, amongst others. See the caveat in the roadmap.

Deploy with quantization: only per-axis quantization for convolutional layers, not per-tensor quantization, is currently supported.

It is on our roadmap to add support in the following areas:

Model coverage: extended to include RNN/LSTMs and general Concat support.

Hardware acceleration: ensure the TFLite converter can produce full-integer models. See [this issue](https://github.com/tensorflow/tensorflow/issues/38285) for details.

Experiment with quantization use cases:

Experiment with quantization algorithms that span Keras layers or require the training step.

Stabilize APIs.

7.What are model parallelism and data parallelism? Why is the latter generally recommended?

Recent years have witnessed exponential growth in the scale of distributed parallel training and the size of deep learning models. In particular, Transformer-based language models have been stealing the show. The notorious GPT-3 blew out with 175 billion parameters and 96 attention layers with a 3.2 M batch size and 499 billion words. Exactly half a year later, Google published [Switch Transformer](https://arxiv.org/abs/2101.03961) with 1.6 trillion parameters. On the same day (1/11/2021), the Beijing Academy of Artificial Intelligence (BAAI) released the initial [Wu Dao](https://en.wikipedia.org/wiki/Wu_Dao) 1.0. Before long, Wu Dao 2.0 debuted on 5/31/2021 as the largest language model with 1.75 trillion parameters and ten times GPT-3 parameters.

Suppose we train GPT-3 on 240 ml.p4d.24xlarge instances of the Amazon SageMaker training platform, the whole model will take [25 days](https://towardsdatascience.com/distributed-parallel-training-model-parallel-training-a768058aa02a) to train. The challenge is not just processing but also memory. Wu Tao 2.0 appears to need more than [1000 GPUs](https://youtu.be/tgB671SFS4w?t=418) only to store its parameters.

It is imperative to employ distributed parallel training for deep learning large models like GPT-3 and DALL-E 2. There are two primary types of distributed parallel training: data parallelism and model parallelism. We further divide the latter into two subtypes: pipeline parallelism and tensor parallelism. We will cover all distributed parallel training here and demonstrate how to develop in PyTorch.

Understanding Distributed Parallel Training

Distributed parallel training has two high-level concepts: parallelism and distribution.

Parallelism is a framework strategy to tackle the size of large models or improve training efficiency, and distribution is an infrastructure architecture to scale out.

In addition to the two basic types of parallelism, there are many more variants, such as [expert parallelism](https://arxiv.org/pdf/2101.03961.pdf). Furthermore, they can be mixed with two or all, such as data and model mixed parallelism. It’s common to mix both model and data parallelism for large-scale models. For instance, the largest [T5](https://arxiv.org/abs/1910.10683) models and [GPT-3](https://arxiv.org/abs/2005.14165) employ a model and data combined parallelism. However, all of these should be part of the strategy of the DL modeling framework.

On the other side, distribution eventually scales the parallelism out in the cloud or a cluster. Containerization makes it easy to scale nodes, and Kubernetes or cloud solutions can orchestrate them effectively. Each node can have multiple GPUs (or TPUs and other devices) and various containers in a container cluster. In the cloud-native solution, the nodes can be hidden from users. A container manages one or more GPUs. The parallelism can be dispatched across a cluster of distributed GPU containers. So the distribution is the implementation of infrastructure architecture.

Data and weight partitioning strategies of Google Switch Transformers (source: [Fedus et al., 2021](https://arxiv.org/pdf/2101.03961.pdf" \t "_blank))

The above illustrates data and weight partitioning strategies in Google Switch Transfers. Each 4×4 dotted-line grid represents 16 cores, and the shaded squares are the data on that core (either model weights or batch of tokens). It demonstrates how the model weights and the data tensors are split for each strategy. The first row illustrates how model weights are divided across the cores. Shapes of different sizes in this row represent larger weight matrices in the Feed Forward Network (FFN) layers (e.g., larger dff sizes). Each color of the shaded squares identifies a unique weight matrix. The number of parameters per core is fixed, but larger weight matrices will apply more computation to each token. The second row demonstrates how the data batch is split across cores. Each core holds the same number of tokens, maintaining a fixed memory usage across all strategies. The partitioning strategies have different properties, allowing each core to have the same or different tokens across cores in different colors.

Data Parallelism in PyTorch

Data parallelism shards data across all cores with the same model. A data parallelism framework like PyTorch Distributed Data Parallel, SageMaker Distributed, and Horovod mainly accomplishes the following three tasks:

First, it creates and dispatches copies of the model, one copy per each accelerator.

It shards the data and then distributes it to the corresponding devices.

It finally aggregates all results together in the backpropagation step.

So we can see that the first task should happen once per training, but the last two tasks should occur in each iteration.

PyTorch [Distributed Data Parallel](https://pytorch.org/tutorials/intermediate/ddp_tutorial.html) (DDP) implements data parallelism at the module level for running across multiple machines. It can work together with the PyTorch model parallel. DDP applications should spawn multiple processes and create a DDP instance per process. DDP uses collective communications in the torch.distributed package to synchronize gradients and buffers. Furthermore, DDP registers an autograd hook for each parameter from model.parameters(), and it will fire when the corresponding gradient is computed in the backward pass. DDP then uses that signal to trigger gradient synchronization across processes.  
def run\_dummy(dummy\_fn, world\_size):  
 mp.spawn(dummy\_fn,  
 args=(world\_size,),  
 nprocs=world\_size,  
 join=True)

Model Parallelism in PyTorch

Model parallelism shards a model (i.e., its layers or tensors) across multiple cores, unlike data parallelism, replicating the same model for all training cores. PyTorch alleviates the parallel implementation and wraps it with minimal changes.

In a nutshell, you need to specify neural network layers and immediate outputs to the desired cores via “to(device)” in three corresponding areas: modeling definition, “forward” method, and “backward” method while calling the loss function. PyTorch will handle all the rest behind the scenes. Please see the example code [here](https://towardsdatascience.com/distributed-parallel-training-model-parallel-training-a768058aa02a).

This may not be straightforward in the real world of large model parallelism. It often needs extra efforts to improve training efficiency and resource utilization. Taking pipeline parallelism as an example, [PipeDream](https://dl.acm.org/doi/10.1145/3341301.3359646" \t "_blank) improves pipeline efficiency by sacrificing memory to store multiple copies of weights. [TeraPipe](https://arxiv.org/abs/2102.07988" \t "_blank) introduces another pipelining specific to single-transformer architectures, where pipelining occurs across tokens rather than micro-batches. Also, [Mesh-TensorFlow](https://arxiv.org/abs/1811.02084) and [Megatron-LM](https://arxiv.org/abs/1909.08053) create a tensor parallelism framework for optimally training billion-parameter models based on TensorFlow and PyTorch, respectively.

Amazon SageMaker [model parallelism](https://arxiv.org/pdf/2111.05972.pdf) is a software library on top of PyTorch. It is a general and flexible framework, supporting pipeline and tensor parallelism with memory-saving features. Its pipeline parallelism engine enables load-balancing auto-partitioning and pipelining runtime for arbitrary model architectures based on module-server design. As with pipeline parallelism, the fundamental computational unit for tensor parallelism is nn.Module. In essence, tensor parallelism consists in traversing the model and replacing specific submodules of the model with their distributed implementations.

Take Aways

Distributed parallel training has two high-level concepts of parallelism and distribution. Parallelism is a framework strategy, and distribution is an infrastructure architecture. Distributed parallel training is critical but still nascent in the industry and research. We can look forward to three creative areas happening in the future.

Parallelism shards data, models, or mixed to make large model training work. As data and models grow exponentially, optimizing memory usage and processing efficiency becomes vital.

It’s expensive to train a large model. Transfer learning for reusing the trained layers will change the game of large-scale distributed parallel training.

The ML lifecycle involves multiple distributed systems, from data collection to processing, model training, and serving. ML platform is often hindered by complexity, data communication costs, and system instability. Unifying all distributed systems for ML will be significant.

8.When training a model across multiple servers, what distribution strategies can you use? How do you choose which one to use?

Overview

[tf.distribute.Strategy](https://www.tensorflow.org/api_docs/python/tf/distribute/Strategy) is a TensorFlow API to distribute training across multiple GPUs, multiple machines, or TPUs. Using this API, you can distribute your existing models and training code with minimal code changes.

[tf.distribute.Strategy](https://www.tensorflow.org/api_docs/python/tf/distribute/Strategy) has been designed with these key goals in mind:

Easy to use and support multiple user segments, including researchers, machine learning engineers, etc.

Provide good performance out of the box.

Easy switching between strategies.

You can distribute training using [tf.distribute.Strategy](https://www.tensorflow.org/api_docs/python/tf/distribute/Strategy) with a high-level API like Keras [Model.fit](https://www.tensorflow.org/api_docs/python/tf/keras/Model" \l "fit), as well as [custom training loops](https://www.tensorflow.org/guide/keras/writing_a_training_loop_from_scratch) (and, in general, any computation using TensorFlow).

In TensorFlow 2.x, you can execute your programs eagerly, or in a graph using [tf.function](https://www.tensorflow.org/guide/function). [tf.distribute.Strategy](https://www.tensorflow.org/api_docs/python/tf/distribute/Strategy) intends to support both these modes of execution, but works best with [tf.function](https://www.tensorflow.org/api_docs/python/tf/function). Eager mode is only recommended for debugging purposes and not supported for [tf.distribute.TPUStrategy](https://www.tensorflow.org/api_docs/python/tf/distribute/TPUStrategy). Although training is the focus of this guide, this API can also be used for distributing evaluation and prediction on different platforms.

You can use [tf.distribute.Strategy](https://www.tensorflow.org/api_docs/python/tf/distribute/Strategy) with very few changes to your code, because the underlying components of TensorFlow have been changed to become strategy-aware. This includes variables, layers, models, optimizers, metrics, summaries, and checkpoints.

In this guide, you will learn about various types of strategies and how you can use them in different situations. To learn how to debug performance issues, check out the [Optimize TensorFlow GPU performance](https://www.tensorflow.org/guide/gpu_performance_analysis) guide.

Types of strategies

[tf.distribute.Strategy](https://www.tensorflow.org/api_docs/python/tf/distribute/Strategy) intends to cover a number of use cases along different axes. Some of these combinations are currently supported and others will be added in the future. Some of these axes are:

Synchronous vs asynchronous training: These are two common ways of distributing training with data parallelism. In sync training, all workers train over different slices of input data in sync, and aggregating gradients at each step. In async training, all workers are independently training over the input data and updating variables asynchronously. Typically sync training is supported via all-reduce and async through parameter server architecture.

Hardware platform: You may want to scale your training onto multiple GPUs on one machine, or multiple machines in a network (with 0 or more GPUs each), or on Cloud TPUs.

In order to support these use cases, TensorFlow has MirroredStrategy, TPUStrategy, MultiWorkerMirroredStrategy, ParameterServerStrategy, CentralStorageStrategy, as well as other strategies available. The next section explains which of these are supported in which scenarios in TensorFlow.