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

**Ans : A SavedModel contains a TensorFlow model, including its architecture (a computation graph) and its weights. It is stored as a directory containing a saved\_model.pb file, which defines the computation graph (represented as a serialized protocol buffer), and a variables subdirectory containing the variable values. For models containing a large number of weights, these variable values may be split across multiple files. A SavedModel also includes an assets subdirectory that may contain additional data, such as vocabulary files, class names, or some example instances for this model. To be more accurate, a SavedModel can contain one or more metagraphs. A metagraph is a computation graph plus some function signature definitions (including their input and output names, types, and shapes). Each metagraph is identified by a set of tags. To inspect a SavedModel, you can use the command-line tool saved\_model\_cli or just load it using tf.saved\_model.load() and inspect it in Python.**

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

**Ans : TF Serving allows you to deploy multiple TensorFlow models (or multiple versions of the same model) and make them accessible to all your applications easily via a REST API or a gRPC API. Using your models directly in your applications would make it harder to deploy a new version of a model across all applications. Implementing your own microservice to wrap a TF model would require extra work, and it would be hard to match TF Serving’s features. TF Serving has many features: it can monitor a directory and autodeploy the models that are placed there, and you won’t have to change or even restart any of your applications to benefit from the new model versions; it’s fast, well tested, and scales very well; and it supports A/B testing of experimental models and deploying a new model version to just a subset of your users (in this case the model is called a canary). TF Serving is also capable of grouping individual requests into batches to run them jointly on the GPU. To deploy TF Serving, you can install it from source, but it is much simpler to install it using a Docker image. To deploy a cluster of TF Serving Docker images, you can use an orchestration tool such as Kubernetes, or use a fully hosted solution such as Google Cloud AI Platform.**

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

**Ans : To deploy a model across multiple TF Serving instances, all you need to do is configure these TF Serving instances to monitor the same models directory, and then export your new model as a SavedModel into a subdirectory.**

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

**Ans : The gRPC API is more efficient than the REST API. However, its client libraries are not as widely available, and if you activate compression when using the REST API, you can get almost the same performance. So, the gRPC API is most useful when you need the highest possible performance and the clients are not limited to the REST API.**

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

**Ans : To reduce a model’s size so it can run on a mobile or embedded device, TFLite uses several techniques:**

* **It provides a converter which can optimize a SavedModel: it shrinks the model and reduces its latency. To do this, it prunes all the operations that are not needed to make predictions (such as training operations), and it optimizes and fuses operations whenever possible.**
* **The converter can also perform post-training quantization: this technique dramatically reduces the model’s size, so it’s much faster to download and store.**
* **It saves the optimized model using the FlatBuffer format, which can be loaded to RAM directly, without parsing. This reduces the loading time and memory footprint.**

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

**Ans : Quantization-aware training consists in adding fake quantization operations to the model during training. This allows the model to learn to ignore the quantization noise; the final weights will be more robust to quantization.**

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

**Ans : Model parallelism means chopping your model into multiple parts and running them in parallel across multiple devices, hopefully speeding up the model during training or inference. Data parallelism means creating multiple exact replicas of your model and deploying them across multiple devices. At each iteration during training, each replica is given a different batch of data, and it computes the gradients of the loss with regard to the model parameters. In synchronous data parallelism, the gradients from all replicas are then aggregated and the optimizer performs a Gradient Descent step. The parameters may be centralized (e.g., on parameter servers) or replicated across all replicas and kept in sync using AllReduce. In asynchronous data parallelism, the parameters are centralized and the replicas run independently from each other, each updating the central parameters directly at the end of each training iteration, without having to wait for the other replicas. To speed up training, data parallelism turns out to work better than model parallelism, in general. This is mostly because it requires less communication across devices. Moreover, it is much easier to implement, and it works the same way for any model, whereas model parallelism requires analyzing the model to determine the best way to chop it into pieces.**

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

**Ans : When training a model across multiple servers, you can use the following distribution strategies:**

* **The MultiWorkerMirroredStrategy performs mirrored data parallelism. The model is replicated across all available servers and devices, and each replica gets a different batch of data at each training iteration and computes its own gradients. The mean of the gradients is computed and shared across all replicas using a distributed AllReduce implementation (NCCL by default), and all replicas perform the same Gradient Descent step. This strategy is the simplest to use since all servers and devices are treated in exactly the same way, and it performs fairly well. In general, you should use this strategy. Its main limitation is that it requires the model to fit in RAM on every replica.**
* **The ParameterServerStrategy performs asynchronous data parallelism. The model is replicated across all devices on all workers, and the parameters are sharded across all parameter servers. Each worker has its own training loop, running asynchronously with the other workers; at each training iteration, each worker gets its own batch of data and fetches the latest version of the model parameters from the parameter servers, then it computes the gradients of the loss with regard to these parameters, and it sends them to the parameter servers. Lastly, the parameter servers perform a Gradient Descent step using these gradients. This strategy is generally slower than the previous strategy, and a bit harder to deploy, since it requires managing parameter servers. However, it is useful to train huge models that don’t fit in GPU RAM.**