Distributed Multi-worker TensorFlow Training on Kubernetes

1 hour 30 minutesFree

GSP775

Google Cloud Self-Paced Labs

Overview

Large Deep Neural Networks (DNNs) have emerged as a critical component of many modern applications across all industries. Accelerating training for the ever increasing size of datasets and deep neural network models is a major challenge facing organizations adopting DNNs. The use of hardware accelerators and distributed clusters is becoming mainstream.

In this hands-on lab, you will explore using Google Kubernetes Engine (GKE) and Kubeflow TFJob to scale out TensorFlow distributed training.

What you'll learn

In this lab, you will learn how to:

- Deploy **TFJob** components to Google Kubernetes Engine.
- Configure multi-worker distributed training jobs using **TFJob**.
- Submit and monitor **TFJob** jobs.

Prerequisites

To successfully complete the lab you need to have a solid understanding of TensorFlow distributed training and a basic familiarity with Kubernetes concepts and architecture. Before proceeding with the lab we recommend reviewing the following resources:

- Distributed training with TensorFlow

- Kubernetes Overview

Lab scenario

You will train an MNIST classification model using TensorFlow multi-worker distributed strategy. You will use Kubeflow TFJob to configure, submit and monitor distributed training jobs on a Google Kubernetes Cluster (GKE).

TFJob is a Kubernetes custom resource designed to support TensorFlow distributed training algorithms. It is flexible enough to support process topologies for both Parameter Server and Mirrored distributed strategies.

TFJob supports the following distributed training roles:

- **Chief**. The chief is responsible for orchestrating training and performing tasks like checkpointing the model.
- **Ps**. The parameter servers provide a distributed data store for the model parameters.
- **Worker**. The workers do the actual work of training the model. In some cases, worker 0 might also act as the chief.
- **Evaluator**. The evaluators can be used to compute evaluation metrics as the model is trained.

TFJob automatically sets the `TF_CONFIG` environment variable in each of the configured pods to reflect the job's topology. The TF_CONFIG variable is required by TensorFlow for multi-worker settings.

In the lab, you will configure a job with three **Workers**. All workers use the same container image and execute the same training code. The training code checks the type of worker it is running on and performs additional tasks on the **Chief** (Worker with the index 0). Specifically, at the end of training it saves the trained model to a persistent storage location specified as one of the script's arguments. In the lab, you will use Cloud Storage.

The training code is designed to recover from failures that may happen during the training. It uses BackupAndRestore callback to automatically save checkpoints at the end of each training epoch. The checkpoints are also stored in **Cloud Storage**.

During the lab you will perform the following tasks:

- Create a **GKE** cluster
- Deploy **TFJob** components
- Configure a **TFJob** manifest
- Submit and monitor the configured **TFJob**

Setup and requirements

Before you click the Start Lab button

Read these instructions. Labs are timed and you cannot pause them. The timer, which starts when you click **Start Lab**, shows how long Google Cloud resources will be made available to you.

This hands-on lab lets you do the lab activities yourself in a real cloud environment, not in a simulation or demo environment. It does so by giving you new, temporary credentials that you use to sign in and access Google Cloud for the duration of the lab.

To complete this lab, you need:

- Access to a standard internet browser (Chrome browser recommended).

Note: Use an Incognito or private browser window to run this lab. This prevents any conflicts between your personal account and the Student account, which may cause extra charges incurred to your personal account.

- Time to complete the lab---remember, once you start, you cannot pause a lab.

Note: If you already have your own personal Google Cloud account or project, do not use it for this lab to avoid extra charges to your account.

How to start your lab and sign in to the Google Cloud Console

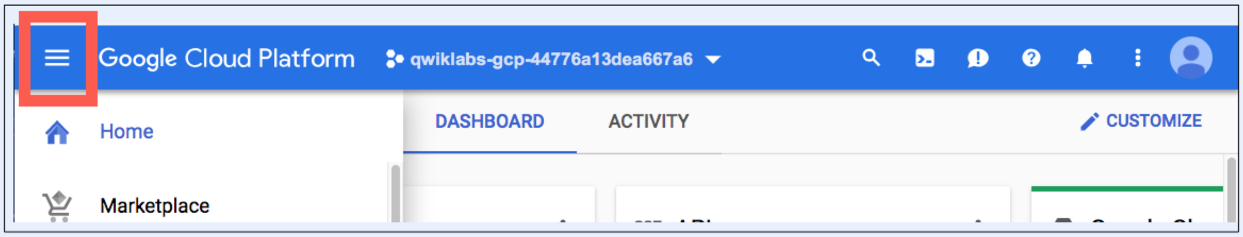
1. Click the **Start Lab** button. If you need to pay for the lab, a pop-up opens for you to select your payment method. On the left is the **Lab Details** panel with the following:
    - The **Open Google Console** button
    - Time remaining

    - The temporary credentials that you must use for this lab
    - Other information, if needed, to step through this lab
2. Click **Open Google Console**. The lab spins up resources, and then opens another tab that shows the **Sign in** page.

   Tip: Arrange the tabs in separate windows, side-by-side.
3. **Note:** If you see the **Choose an account** dialog, click **Use Another Account**.
4. If necessary, copy the **Username** from the **Lab Details** panel and paste it into the **Sign in** dialog. Click **Next**.
5. Copy the **Password** from the **Lab Details** panel and paste it into the **Welcome** dialog. Click **Next**.
6. **Important:** You must use the credentials from the left panel. Do not use your Google Cloud Skills Boost credentials.
7. **Note:** Using your own Google Cloud account for this lab may incur extra charges.
8. Click through the subsequent pages:
    - Accept the terms and conditions.
    - Do not add recovery options or two-factor authentication (because this is a temporary account).
    - Do not sign up for free trials.

After a few moments, the Cloud Console opens in this tab.

Note: You can view the menu with a list of Google Cloud Products and Services by clicking the **Navigation menu** at the top-left.



Activate Cloud Shell

Cloud Shell is a virtual machine that is loaded with development tools. It offers a persistent 5GB home directory and runs on the Google Cloud. Cloud Shell provides command-line access to your Google Cloud resources.

1. Click **Activate Cloud Shell** at the top of the Google Cloud console.
2. Click **Continue**.

It takes a few moments to provision and connect to the environment. When you are connected, you are already authenticated, and the project is set to your **PROJECT_ID**. The output contains a line that declares the **PROJECT_ID** for this session:

```
Your Cloud Platform project in this session is set to YOUR_PROJECT_ID
```

`gcloud` is the command-line tool for Google Cloud. It comes pre-installed on Cloud Shell and supports tab-completion.

3. (Optional) You can list the active account name with this command:

```
gcloud auth list
```

Copied!

content_copy

Output:

```
ACTIVE: *
ACCOUNT: student-01-xxxxxxxxxxxx@qwiklabs.net
To set the active account, run:
    $ gcloud config set account `ACCOUNT`
```

4. (Optional) You can list the project ID with this command:

```
gcloud config list project
```

Copied!

content_copy

Output:

```
[core]
project = <project_ID>
```

Example output:

```
[core]
project = qwiklabs-gcp-44776a13dea667a6
```

Note: For full documentation of `gcloud`, in Google Cloud, refer to the gcloud CLI overview guide.

Lab tasks

You will use **Cloud Shell** for all of the tasks in the lab. Some tasks require you to edit text files. You can use any of the classic command line text editors pre-installed in **Cloud Shell**, including *vim*, *emacs*, or *nano*. You can also use the built-in Cloud Shell Editor.

Before proceeding, make sure that you completed the **Activate Cloud Shell** step in the **Qwiklabs setup** instructions and your **Cloud Shell** is open and ready.

Task 1. Creating a GKE cluster

For the purpose of the lab, you will create a small, CPU-based GKE cluster. The MNIST classifier DNN used in the lab is very simple so the training process does not require accelerated hardware or a large number of nodes. In most commercial settings, where you train/fine-tune industrial grade NLP or Computer Vision models, larger clusters with accelerated hardware will be necessary. Nevertheless, the techniques and patterns demonstrated in this lab using a simplified cluster configuration are transferable to more complex scenarios.

1. Start by setting the default compute zone and a couple of environment variables:

```
gcloud config set compute/zone us-central1-f
PROJECT_ID=$(gcloud config get-value project)
```

```
CLUSTER_NAME=cluster-1
```

Copied!

content_copy

2. Now, create the cluster. The below command may take a few minutes to complete:

```
gcloud container clusters create $CLUSTER_NAME \
  --project=$PROJECT_ID \
  --release-channel=stable \
  --machine-type=n1-standard-4 \
  --scopes compute-rw,gke-default,storage-rw \
  --num-nodes=3
```

Copied!

content_copy

3. After the cluster has started, configure access credentials so you can interact with the cluster using `kubectl`:

```
gcloud container clusters get-credentials $CLUSTER_NAME
```

Copied!

content_copy

Click *Check my progress* to verify the objective.

Creating a GKE cluster

Check my progress

Task 2. Deploying `TFJob` components

TFJob is a component of Kubeflow. It is usually deployed as part of a full **Kubeflow** installation but can also be used in a standalone configuration. In this lab, you will install **TFJob** as a standalone component.

TFJob consists of two parts: a Kubernetes custom resource and an operator implementing the job management logic. Kubernetes manifests for both the custom resource definition and the operator are managed in **Kubeflow GitHub** repository.

Instead of cloning the whole repository you will retrieve the **TFJob** manifests only using an OSS tool - kpt - that is pre-installed in **Cloud Shell**.

1. Get the manifests for TFJob from v1.1.0 of Kubeflow:

```
cd
SRC_REPO=https://github.com/kubeflow/manifests
kpt pkg get $SRC_REPO/tf-training@v1.1.0 tf-training
```

Copied!

content_copy

2. Create a Kubernetes namespace to host the **TFJob** operator:

```
kubectl create namespace kubeflow
```

Copied!

content_copy

3. Install the **TFJob** custom resource:

```
kubectl apply  --kustomize tf-training/tf-job-crds/base
```

Copied!

content_copy

4. Install the **TFJob** operator:

```
kubectl apply  --kustomize tf-training/tf-job-operator/base
```

Copied!

content_copy

5. Verify the installation:

```
kubectl get deployments -n kubeflow
```

Copied!

content_copy

Notice that the TFJob operator is running as a Kubernetes Deployment in the `kubeflow` namespace.

It may take a couple of minutes before the deployment is ready.

Click *Check my progress* to verify the objective.

Deploying TFJob components

Check my progress

Task 3. Creating a Cloud Storage bucket

As described in the lab overview, the distributed training script stores training checkpoints and the trained model in the *SavedModel* format to the storage location passed as one of the script's arguments. You will use a **Cloud Storage** bucket as a shared persistent storage.

1. Since storage buckets are a global resource in Google Cloud you have to use a unique bucket name. For the purpose of this lab, you can use your project id as a name prefix:

```
export TFJOB_BUCKET=${PROJECT_ID}-bucket
gsutil mb gs://${TFJOB_BUCKET}
```

Copied!

content_copy

2. Verify that the bucket was successfuly created:

```
gsutil ls
```

Copied!

content_copy

Click *Check my progress* to verify the objective.

Creating a Cloud Storage bucket

Check my progress

Task 4. Preparing TFJob

Your distributed training environment is ready and you can now prepare and submit distributed training jobs.

The TensorFlow training code and the **TFJob** manifest template used in the lab can be retrieved from **GitHub**.

```
cd
SRC_REPO=https://github.com/GoogleCloudPlatform/mlops-on-gcp
kpt pkg get $SRC_REPO/workshops/mlep-qwiklabs/distributed-training-gke
lab-files
cd lab-files
```

Copied!

content_copy

The training module is in the `mnist` folder. The `model.py` file contains a function to create a simple convolutional network.

The `main.py` file contains data preprocessing routines and a distributed training loop. Review the files. Notice how you can use a `tf.distribute.experimental.MultiWorkerMirrorStrategy()` object to retrieve information about the topology of the distributed cluster running a job.

```
   strategy = tf.distribute.experimental.MultiWorkerMirroredStrategy()
    task_type = strategy.cluster_resolver.task_type
    task_id = strategy.cluster_resolver.task_id
    global_batch_size = per_worker_batch * strategy.num_replicas_in_sync
```

Copied!

content_copy

You can also see how to configure automatic checkpointing using `tf.keras.callbacks.experimental.BackupAndRestore()`.

```
  callbacks = [
        tf.keras.callbacks.experimental.BackupAndRestore(checkpoint_path)
    ]
    multi_worker_model.fit(dataset,
                           epochs=epochs,
                           steps_per_epoch=steps_per_epoch,
                           callbacks=callbacks)
```

Copied!

content_copy

You can control the training loop by passing command line arguments to the `main.py` script. We will use it when configuring a **TFJob** manifest.

Packaging training code in a docker image

Before submitting the job, the training code must be packaged in a docker image and pushed into your project's Container Registry. You can find the Dockerfile that creates the image in the `lab-files` folder. You do not need to modify the Dockerfile.

- To build the image and push it to the registry execute the below commands.

```
IMAGE_NAME=mnist-train
docker build -t gcr.io/${PROJECT_ID}/${IMAGE_NAME} .
docker push gcr.io/${PROJECT_ID}/${IMAGE_NAME}
```

Copied!

content_copy

Updating the TFJob manifest

The `tfjob.yaml` file is an example TFJob manifest:

```
apiVersion: kubeflow.org/v1
kind: TFJob
metadata:
  name: multi-worker
spec:
  cleanPodPolicy: None
  tfReplicaSpecs:
    Worker:
      replicas: 3
      template:
        spec:
          containers:
            - name: tensorflow
              image: mnist
              args:
                - --epochs=5
                - --steps_per_epoch=100
                - --per_worker_batch=64
                - --saved_model_path=gs://bucket/saved_model_dir
                - --checkpoint_path=gs://bucket/checkpoints
```

Copied!

content_copy

As noted in the lab overview, you have a lot of flexibility in defining the job's process topology and allocating hardware resources. Please refer to the TFJob guide for more information.

The key field in the TFJob manifest is `tfReplicaSpecs`, which defines the number and the types of replicas (pods) created by a job. In our case, the job will start 3 workers using the container image defined in the `image` field and command line arguments defined in the `args` field.

Before submitting a job, you need to update the `image` and `args` fields with the values reflecting your environment.

Use your preferred command line editor or **Cloud Shell Editor** to update the `image` field with a full name of the image you created and pushed to your **Container Registry** in the previous step.

1. You can retrieve the image name using the following command:

```
gcloud container images list
```

Copied!

content_copy

The name should have the following format:

```
gcr.io/<YOUR_PROJECT_ID>/mnist-train
```

Copied!

content_copy

2. Next, update the `--saved_model_path` and `--checkpoint_path` arguments by replacing the `bucket` token with the name of your Cloud storage bucket. Recall that your bucket name is `[YOUR_PROJECT_ID]-bucket`.

The updated manifest should look similar to the one below:

```
apiVersion: kubeflow.org/v1
kind: TFJob
```

```
metadata:
  name: multi-worker
spec:
  cleanPodPolicy: None
  tfReplicaSpecs:
    Worker:
      replicas: 3
      template:
        spec:
          containers:
          - name: tensorflow
            image: gcr.io/qwiklabs-gcp-01-93af833e6576/mnist-train
            args:
            - --epochs=5
            - --steps_per_epoch=100
            - --per_worker_batch=64
            -
--saved_model_path=gs://qwiklabs-gcp-01-93af833e6576-bucket/saved_model_di
r
            -
--checkpoint_path=gs://qwiklabs-gcp-01-93af833e6576-bucket/checkpoints
```

Copied!

content_copy

Task 5. Submitting the TFJob

- You can now submit the job using `kubectl`.

```
kubectl apply -f tfjob.yaml
```

Copied!

content_copy

Task 6. Monitoring the TFJob

During execution, TFJob will emit events to indicate the status of the job, including creation/deletion of pods and services.

1. You can retrieve the recent events and other information about the job by executing the following command:

```
JOB_NAME=multi-worker
kubectl describe tfjob $JOB_NAME
```

Copied!

content_copy

Recall that the job name was specified in the job manifest.

2. To retrieve logs generated by the training code you can use the `kubectl logs` command. Start by listing all pods created by the job:

```
kubectl get pods
```

Copied!

content_copy

Notice that the pods are named using the following convention `[JOB_NAME]-worker-[WORKER_INDEX]`.

Wait till the status of all pods changes to `Running`.

3. To retrieve the logs for the chief (worker 0) execute the following command. It will continue streaming the logs till the training program completes:

```
kubectl logs --follow ${JOB_NAME}-worker-0
```

Copied!

content_copy

4. After the job completes, the pods are not removed to allow for later inspection of logs. For example, to check the logs created by worker 1:

```
kubectl logs ${JOB_NAME}-worker-1
```

Copied!

content_copy

Click *Check my progress* to verify the objective.

Submitting the TFJob

Check my progress

5. To remove the job and the associated pods:

```
kubectl delete tfjob $JOB_NAME
```

Copied!

content_copy

Congratulations