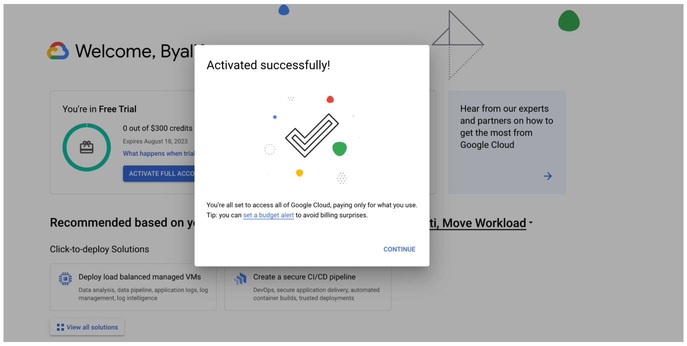
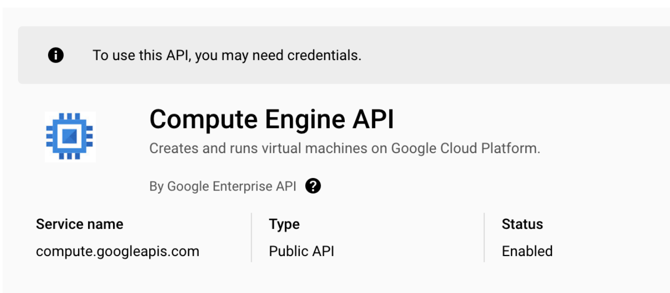
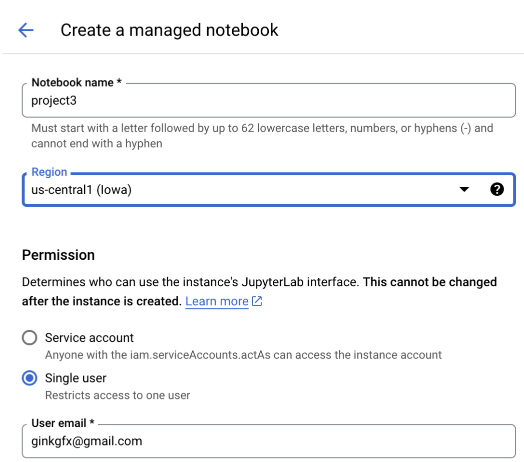
Alif Rahi
CSCI 381 - Project 3

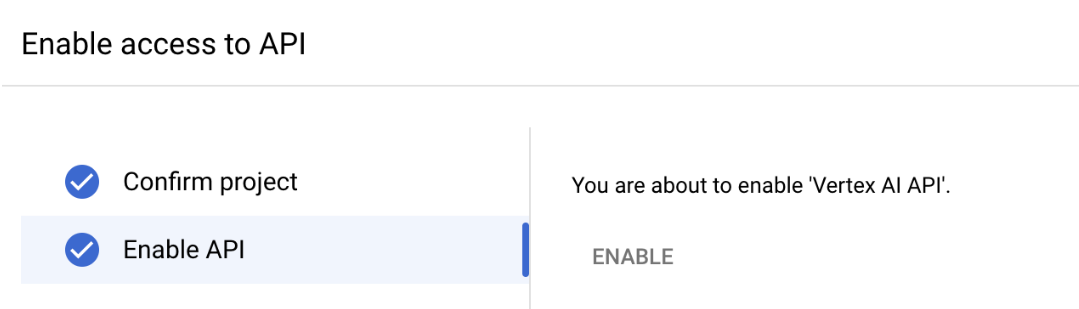
Step 1: Set up environment

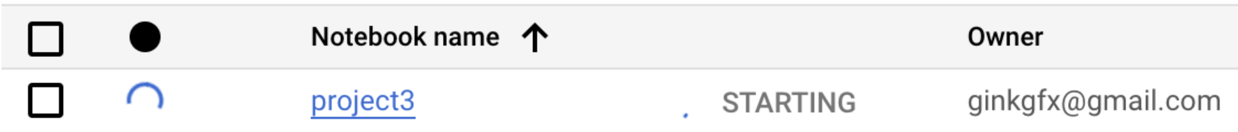


Enable extensions

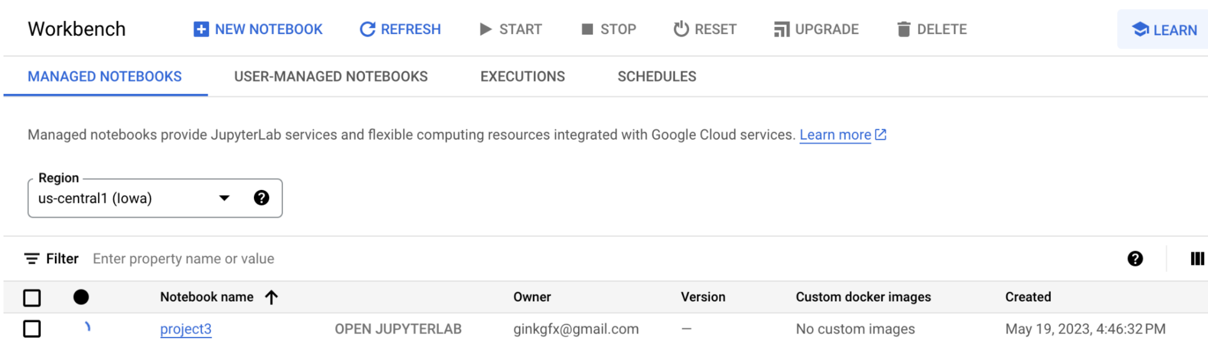




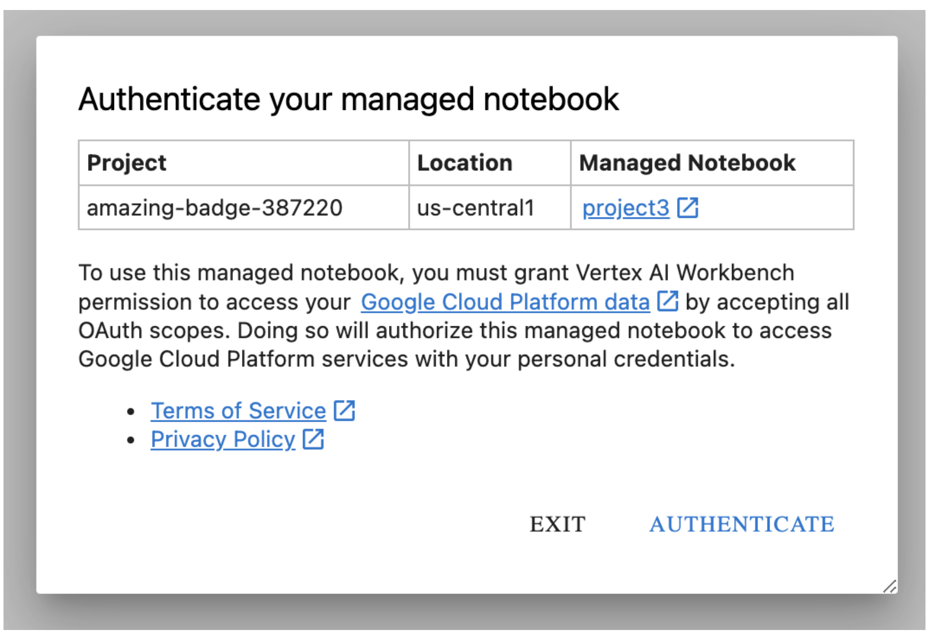




Notebook name
Owner
project3
STARTING
ginkgfx@gmail.com



Workbench
NEW NOTEBOOK
REFRESH
START
STOP
RESET
UPGRADE
DELETE
LEARN
MANAGED NOTEBOOKS
USER-MANAGED NOTEBOOKS
EXECUTIONS
SCHEDULES
Managed notebooks provide JupyterLab services and flexible computing resources integrated with Google Cloud services. Learn more
Region
us-central1 (Iowa)
Filter
Enter property name or value
Notebook name
Owner
Version
Custom docker images
Created
project3
OPEN JUPYTERLAB
ginkgfx@gmail.com
—
No custom images
May 19, 2023, 4:46:32 PM



Authenticate your managed notebook
Project
Location
Managed Notebook
amazing-badge-387220
us-central1
project3
To use this managed notebook, you must grant Vertex AI Workbench permission to access your Google Cloud Platform data by accepting all OAuth scopes. Doing so will authorize this managed notebook to access Google Cloud Platform services with your personal credentials.
Terms of Service
Privacy Policy
EXIT
AUTHENTICATE

Step 2: CNN code synch SGD

This code trains a simple CNN model on the MNIST dataset for 5 epochs and saves the trained model to disk. It's important to note that the model training is performed on a single machine, and this code assumes that the necessary dependencies and packages are already installed.

file1.py

```
import tensorflow as tf
from tensorflow.keras.datasets import mnist

# Load the MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Normalize the pixel values to a range of [0, 1]
x_train, x_test = x_train / 255.0, x_test / 255.0

# Define the Convolutional Neural Network (CNN) model
model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28,
28, 1)),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(10, activation='softmax')
])

# Compile the model with Stochastic Gradient Descent (SGD) optimizer and
Cross-Entropy loss function
model.compile(optimizer='sgd',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Train the model on a single machine
model.fit(x_train.reshape(-1, 28, 28, 1), y_train, epochs=5,
          validation_data=(x_test.reshape(-1, 28, 28, 1), y_test))

# Save the trained model
model.save('mnist_cnn.h5')
```

The code above loads the MNIST dataset using TensorFlow's Keras API and splits it into training and testing datasets. The dataset contains grayscale images of size 28x28 pixels, and there are 10 classes representing the digits 0 to 9. To ensure consistent input, the pixel values are normalized to a range between 0 and 1. Next, a simple convolutional neural network (CNN) model is defined using the TensorFlow Keras API. The model consists of two convolutional layers and two fully connected layers. These layers are designed to extract features from the images and classify them into the appropriate digit category. After defining the model architecture, it is compiled using the Adam optimizer and categorical cross-entropy loss. The Adam optimizer adjusts the model's parameters to minimize the loss function and improve accuracy during training. Finally, the model is trained on the training set for 5 epochs, with a batch size of 32. This means that the model will process the training data in batches of 32 samples at a time and iterate over the entire training dataset 5 times, adjusting its parameters to improve its predictions. This code loads the MNIST dataset, defines a CNN model, compiles it, and trains it on the training data to classify handwritten digits with the goal of improving accuracy.

file2.py

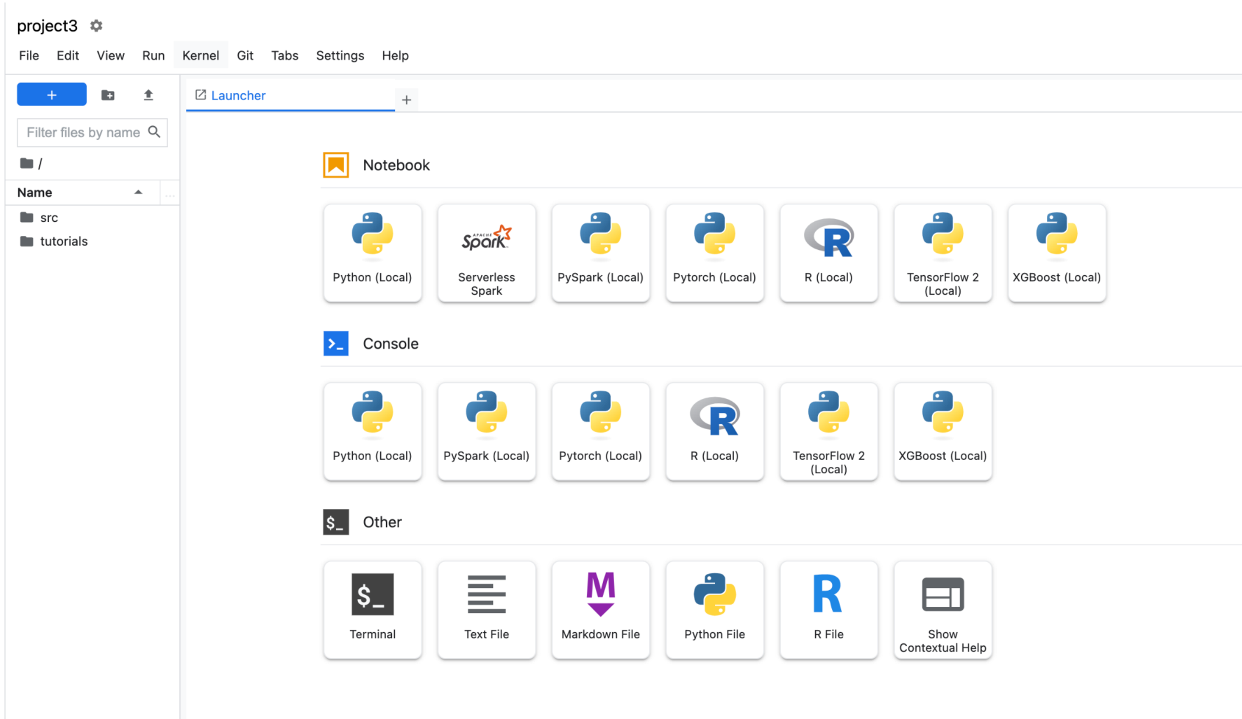
```
import tensorflow as tf
import matplotlib.pyplot as plt
import os

def create_distributed_dataset(strategy, batch_size, dataset_func,
data_dir):
    options = tf.data.Options()
    options.experimental_distribute.auto_shard_policy =
tf.data.experimental.AutoShardPolicy.DATA

    filenames = tf.io.gfile.glob(os.path.join(data_dir, 'mnist-train*'))
    dataset = tf.data.TFRecordDataset(filenames,
num_parallel_reads=tf.data.experimental.AUTOTUNE)
    dataset = dataset.with_options(options)

    feature_description = {
        'image': tf.io.FixedLenFeature([], tf.string),
        'label': tf.io.FixedLenFeature([], tf.int64),
    }

    def _parse_function(example_proto):
        parsed_example = tf.io.parse_single_example(example_proto,
feature_description)
        image = tf.io.decode_raw(parsed_example['image'], tf.uint8)
        image = tf.cast(image, tf.float32) / 255.0
        image = tf.reshape(image, [28, 28, 1])
        label = tf.cast(parsed_example['label'], tf.int32)
        return image, label

    dataset = dataset.map(_parse_function,
num_parallel_calls=tf.data.experimental.AUTOTUNE)
    dataset = dataset.shuffle(batch_size * 10).batch(batch_size).repeat()
```

```
    distributed_dataset =
strategy.experimental_distribute_dataset(dataset)
    return distributed_dataset

def create_distributed_model(strategy, learning_rate=0.001,
sync_mode=True):
    with strategy.scope():
        model = tf.keras.Sequential([
            tf.keras.layers.Conv2D(32, 3, activation='relu',
input_shape=(28, 28, 1)),
            tf.keras.layers.MaxPooling2D(),
            tf.keras.layers.Flatten(),
            tf.keras.layers.Dense(64, activation='relu'),
            tf.keras.layers.Dense(10)
        ])

        optimizer = tf.keras.optimizers.SGD(learning_rate=learning_rate)
if sync_mode else tf.keras.optimizers.Adam(learning_rate=learning_rate)
        optimizer = tf.keras.mixed_precision.LossScaleOptimizer(optimizer,
"dynamic")

        model.compile(

loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
            optimizer=optimizer,
            metrics=['accuracy']
        )

    return model

batch_size = 128
epochs = 5

strategy = tf.distribute.MirroredStrategy()
train_dataset = create_distributed_dataset(strategy, batch_size,
tf.data.TFRecordDataset, './mnist_data')

sync_model = create_distributed_model(strategy, learning_rate=0.001,
sync_mode=True)
history = sync_model.fit(train_dataset, epochs=epochs)

async_model = create_distributed_model(strategy, learning_rate=0.001,
sync_mode=False)
history = async_model.fit(train_dataset, epochs=epochs)
```

This code ^ utilizes TensorFlow's MirroredStrategy to enable distributed training. MirroredStrategy is a synchronization-based training strategy that duplicates the model across multiple devices. It processes mini-batches of data in parallel, with each device handling a subset of the data. The devices independently compute gradients and send them to a parameter server. The server then aggregates the gradients and broadcasts the updated parameters back to each device. This iterative process continues until convergence. MirroredStrategy is commonly employed for distributed training on multiple GPUs within a single machine or across multiple machines. By leveraging MirroredStrategy, the code can effectively scale the training process and reduce training time when dealing with large datasets.

- The code begins by defining a function called create_distributed_dataset() that generates a distributed dataset. This function requires a TensorFlow distribution strategy, batch size, dataset function (such as TFRecordDataset), and data directory as inputs. It then proceeds to read TFRecord files from the specified directory, parse them, shuffle and batch the data, and distribute it using the provided distribution strategy.
- Subsequently, the code establishes another function called create_distributed_model() to construct a distributed model. This function expects a TensorFlow distribution strategy, learning rate, and a synchronous mode flag, and returns a compiled Keras model. If the synchronous mode flag is set to True, the function employs synchronous stochastic gradient descent (SGD) for training, whereas asynchronous SGD is used if it is set to False.
- The code proceeds by defining the batch size and number of epochs, and creating a MirroredStrategy to facilitate training the model. MirroredStrategy is a synchronous distributed strategy that synchronously replicates the model across multiple GPUs or CPUs.
- Two models are subsequently created using the aforementioned strategies: one utilizing synchronous SGD and the other utilizing asynchronous SGD. The create_distributed_model() function is invoked to generate these models. The models are then trained using the fit() method, with the distributed dataset serving as input and the specified number of epochs.
- In synchronous SGD, all workers update their weights simultaneously after processing a batch of data, whereas asynchronous SGD allows each worker to independently and asynchronously update its weights without considering other workers. Synchronous SGD is easier to implement but may be slower due to synchronization requirements. On the other hand, asynchronous SGD can be faster but needs careful implementation to avoid convergence issues.

Step 3: Uploading code to Vertex AI



Install dependencies

```
(base) jupyter@vm-b06ce8cf-55c9-4b03-bd8e-1a063069f687:~$ pip install matplotlib
Requirement already satisfied: matplotlib in /opt/conda/lib/python3.7/site-packages (3.5.3)
Requirement already satisfied: packaging>=20.0 in /opt/conda/lib/python3.7/site-packages (from matplotlib) (23.0)
Requirement already satisfied: numpy>=1.17 in /opt/conda/lib/python3.7/site-packages (from matplotlib) (1.21.6)
Requirement already satisfied: python-dateutil>=2.7 in /opt/conda/lib/python3.7/site-packages (from matplotlib) (2.8.2)
Requirement already satisfied: pyparsing>=2.2.1 in /opt/conda/lib/python3.7/site-packages (from matplotlib) (3.0.9)
Requirement already satisfied: cycler>=0.10 in /opt/conda/lib/python3.7/site-packages (from matplotlib) (0.11.0)
Requirement already satisfied: pillow>=6.2.0 in /opt/conda/lib/python3.7/site-packages (from matplotlib) (9.4.0)
Requirement already satisfied: fonttools>=4.22.0 in /opt/conda/lib/python3.7/site-packages (from matplotlib) (4.38.0)
Requirement already satisfied: kiwisolver>=1.0.1 in /opt/conda/lib/python3.7/site-packages (from matplotlib) (1.4.4)
Requirement already satisfied: typing-extensions in /opt/conda/lib/python3.7/site-packages (from kiwisolver>=1.0.1->matplotlib) (4.5.0)
Requirement already satisfied: six>=1.5 in /opt/conda/lib/python3.7/site-packages (from python-dateutil>=2.7->matplotlib) (1.16.0)
(base) jupyter@vm-b06ce8cf-55c9-4b03-bd8e-1a063069f687:~$
```

```
(base) jupyter@vm-b06ce8cf-55c9-4b03-bd8e-1a063069f687:~$ pip install tensorflow
Collecting tensorflow
  Downloading tensorflow-2.11.0-cp37-cp37m-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (588.3 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 588.3/588.3 MB 68.4 MB/s eta 0:00:01
```

```
(base) jupyter@vm-b06ce8cf-55c9-4b03-bd8e-1a063069f687:~$ pip install numpy
Requirement already satisfied: numpy in /opt/conda/lib/python3.7/site-packages (1.21.6)
(base) jupyter@vm-b06ce8cf-55c9-4b03-bd8e-1a063069f687:~$
```

project3

File Edit View Run Kernel Git Tabs Settings Help

Terminal 1 | file1.py

Filter files by name

/ proj3 /

Name

file1.py

```python
import tensorflow as tf
from tensorflow.keras.datasets import mnist

# Load the MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Normalize the pixel values to a range of [0, 1]
x_train, x_test = x_train / 255.0, x_test / 255.0

# Define the Convolutional Neural Network (CNN) model
model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(10, activation='softmax')
])

# Compile the model with Stochastic Gradient Descent (SGD) optimizer and Cross-Entropy loss function
model.compile(optimizer='sgd',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Train the model on a single machine
model.fit(x_train.reshape(-1, 28, 28, 1), y_train, epochs=5,
          validation_data=(x_test.reshape(-1, 28, 28, 1), y_test))

# Save the trained model
model.save('mnist_cnn.h5')
```

project3

File Edit View Run Kernel Git Tabs Settings Help

Terminal 1 | file1.py | file2.py

Filter files by name

/ proj3 /

Name

file1.py

file2.py

```python
import tensorflow as tf
import matplotlib.pyplot as plt
import os

def create_distributed_dataset(strategy, batch_size, dataset_func, data_dir):
    options = tf.data.Options()
    options.experimental_distribute.auto_shard_policy = tf.data.experimental.AutoShardPolicy.DATA

    filenames = tf.io.gfile.glob(os.path.join(data_dir, 'mnist-train*'))
    dataset = tf.data.TFRecordDataset(filenames, num_parallel_reads=tf.data.experimental.AUTOTUNE)
    dataset = dataset.with_options(options)

    feature_description = {
        'image': tf.io.FixedLenFeature([], tf.string),
        'label': tf.io.FixedLenFeature([], tf.int64),
    }

    def _parse_function(example_proto):
        parsed_example = tf.io.parse_single_example(example_proto, feature_description)
        image = tf.io.decode_raw(parsed_example['image'], tf.uint8)
        image = tf.cast(image, tf.float32) / 255.0
        image = tf.reshape(image, [28, 28, 1])
        label = tf.cast(parsed_example['label'], tf.int32)
        return image, label

    dataset = dataset.map(_parse_function, num_parallel_calls=tf.data.experimental.AUTOTUNE)
    dataset = dataset.shuffle(batch_size * 10).batch(batch_size).repeat()

    distributed_dataset = strategy.experimental_distribute_dataset(dataset)
    return distributed_dataset

def create_distributed_model(strategy, learning_rate=0.001, sync_mode=True):
    with strategy.scope():
        model = tf.keras.Sequential([
            tf.keras.layers.Conv2D(32, 3, activation='relu', input_shape=(28, 28, 1)),
            tf.keras.layers.MaxPooling2D(),
            tf.keras.layers.Flatten(),
            tf.keras.layers.Dense(64, activation='relu'),
```

```
(base) jupyter@vm-b06ce8cf-55c9-4b03-bd8e-1a063069f687:~/proj3$ python file1.py
2023-05-20 00:57:27.413329: I tensorflow/core/platform/cpu_feature_guard.cc:193] This TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the
 following CPU instructions in performance-critical operations:  AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
2023-05-20 00:57:29.955016: W tensorflow/compiler/xla/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic library 'libnvinfer.so.7'; dlerror: libnvinfer.s
o.7: cannot open shared object file: No such file or directory; LD_LIBRARY_PATH: /usr/local/cuda/lib64:/usr/local/cuda/lib:/usr/local/lib/x86_64-linux-gnu:/usr/local/nvidia/lib
:/usr/local/nvidia/lib64:/usr/local/nvidia/lib:/usr/local/nvidia/lib64
2023-05-20 00:57:29.955219: W tensorflow/compiler/xla/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic library 'libnvinfer_plugin.so.7'; dlerror: libnv
infer_plugin.so.7: cannot open shared object file: No such file or directory; LD_LIBRARY_PATH: /usr/local/cuda/lib64:/usr/local/cuda/lib:/usr/local/lib/x86_64-linux-gnu:/usr/lo
cal/nvidia/lib:/usr/local/nvidia/lib64:/usr/local/nvidia/lib:/usr/local/nvidia/lib64
2023-05-20 00:57:29.955245: W tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-TRT Warning: Cannot dlopen some TensorRT libraries. If you would like to use Nvidia GPU w
ith TensorRT, please make sure the missing libraries mentioned above are installed properly.
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 0s 0us/step
2023-05-20 00:57:34.284245: W tensorflow/compiler/xla/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic library 'libcuda.so.1'; dlerror: libcuda.so.1: c
annot open shared object file: No such file or directory; LD_LIBRARY_PATH: /usr/local/cuda/lib64:/usr/local/cuda/lib:/usr/local/lib/x86_64-linux-gnu:/usr/local/nvidia/lib:/usr/
local/nvidia/lib64:/usr/local/nvidia/lib:/usr/local/nvidia/lib64
2023-05-20 00:57:34.284322: W tensorflow/compiler/xla/stream_executor/cuda/cuda_driver.cc:265] failed call to cuInit: UNKNOWN ERROR (303)
2023-05-20 00:57:34.284367: I tensorflow/compiler/xla/stream_executor/cuda/cuda_diagnostics.cc:156] kernel driver does not appear to be running on this host (vm-b06ce8cf-55c9-4
b03-bd8e-1a063069f687): /proc/driver/nvidia/version does not exist
2023-05-20 00:57:34.284779: I tensorflow/core/platform/cpu_feature_guard.cc:193] This TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the
 following CPU instructions in performance-critical operations:  AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
Epoch 1/5
1349/1875 [====================>.........] - ETA: 5s - loss: 0.6256 - accuracy: 0.8326
```

Step 4: Monitor code

To enable real-time monitoring of system resources, such as CPU usage, during training, we will utilize dstat and sar tools. To incorporate them into Jupyter Lab

To plot the performance and test error for both synchronous and asynchronous modes, as well as monitor CPU/Memory/Network usage during training, we can:

1. Update the package information
   sudo apt-get update
2. Install the dstat tool:
   sudo apt-get install dstat

By running these commands, you will be able to employ dstat and sar for monitoring the CPU usage while conducting training in Jupyter Lab.

For some reason the commands above did not allow me to ssh in the terminal. It was asking for a password which was not provided to me.

```
(base) jupyter@vm-b06ce8cf-55c9-4b03-bd8e-1a063069f687:~/proj3$
(base) jupyter@vm-b06ce8cf-55c9-4b03-bd8e-1a063069f687:~/proj3$ sudo apt-get update
[sudo] password for jupyter:
```

Pytorch

The distributed training in multiple GPUs can be implemented in PyTorch using two methods: DataParallel or nn. To utilize data parallelism, we can employ the DataParallel class provided by PyTorch. By specifying the GPU IDs and initializing the network with a DataParallel object, we can achieve parallel execution across multiple GPUs. The following steps outline the process:

1. Install PyTorch on your local machine using the Python Installation Package. The specific command used for installation may vary depending on your system configuration.
2. Install torchvision
3. Install matplotlib

To employ data parallelism with PyTorch, you implement the DataParallel class. This class enables the definition of GPU IDs and the initialization of our network using a Module Object accompanied by a DataParallel object. By leveraging the DataParallel class, we can distribute computations across multiple GPUs and enhance the training process's efficiency.