TensorFlow distribuido

Diego Andrade Canosa

Contenidos del curso

- Breve introducción a TensorFlow y Keras
- Repaso de conceptos de entrenamiento distribuido
- Soporte nativo en TF para entrenamiento distribuido
  - MirroredStrategies
  - ParameterServer

TensorFlow

- TensorFlow (TF) es una plataforma de aprendizaje automática
  - Junto a Pytorch, su principal competidor, una de las más populares
- Proporciona:
  - Herramientas avanzadas para el procesamiento y carga de datos
  - Definición de modelos utilizando bloques constructores con distinto nivel de complejidad
  - Implementación de servidores de inferencia de modelos (en producción)
  - Técnicas de regularización
  - Herramientas auxiliares como tensorboard o tf profiler

Evolución histórica

- Versión actual: 2.14
- Año 2011: el *Google Brain Team* (Andrew Ng and Jeff Dean) empiezan un proyecto llamado DistBelief
  - Sistema de ML escalable y distribuido
- Año 2015: Google libera el código de DistBelief y lo renombra como **TensorFlow**
  - *El código abierto como acelerador de la innovación*
- Año 2019: Tensorflow 2.0
  - API más simple
  - Mejor rendimiento
  - Mejor integración con Keras

Ecosistema de TF

- TensorFlow.js (entornos web)
- TensorFlow Lite (IoT)
- TFX (eXtended TF)
- Keras (Bloques constructores más sofisticados)
- TensorBoard (Visualización)
- Jupyter, Colab
- Otras herramientas:
  - Kubeflow (Contenedores para ML)
  - Frameworks distribuidos: Ray, Horovod, etc...

TF vs Pytorch

- Static vs Dynamic Computation Graph
  - Static: Menos flexible, mejor rendimiento
  - Dynamic: Más flexible y fácil de usar, peor rendimiento

```python
import torch

# Define the neural network
class Net(torch.nn.Module):
    def forward(self, x, y):
        return x * y

# Create an instance of the neural network
net = Net()

# Define the input values
x = torch.tensor([2.0, 3.0])
y = torch.tensor([4.0, 5.0])

# Compute the output
output = net(x, y)
print(output)  # Output: tensor([ 8., 15.])
```

```python
import tensorflow as tf

# Define the input values
x = tf.constant([2.0, 3.0])
y = tf.constant([4.0, 5.0])

# Define the computation
output = x * y

# Create a session and run the computation
with tf.Session() as sess:
    result = sess.run(output)
    print(result)  # Output: [ 8. 15.]
```

https://medium.com/@gpj/comparing-pytorch-and-tensorflow-a-practical-guide-45bb5ebf91af#:~:text=In%20conclusion%2C%20PyTorch%20and%20TensorFlow,that%20is%20efficient%20and%20fast.

Dynamic Computation Graphs en TF

- Los DCGs están disponibles en TF a través del modo *eager*
  - Habilitado por defecto
  - Recomendable deshabilitarlo para modelos en producción (inferencia)

Tensorflow (TF): Conceptos básicos

- Características básicas:
  - Soporte para tensores (arrays multidimensionales)
  - Procesamiento en GPU y distribuido
  - Diferenciación automática
  - Definición de modelos, entrenamiento y exportación

TF: Tensores

```
import tensorflow as tf

x = tf.constant([[1., 2., 3.],
[4., 5., 6.]])

print(x)
print(x.shape)
print(x.dtype)
```

Tensores: operadores

- x+x
- 5*x
- transpose
- concat
- reduce_sum
- softmax
- ...
- **Variables**: Son la versión mutable de los tensores (usados para almacenar, por ejemplo, los parámetros entrenables del modelo)

Diferenciación automática

- El mecanismo de **autodiff** es similar al disponible en Pytorch
  - Construye un grafo con los nodos de la computación (durante la pasada *forward*) para calcular los gradientes de los pesos aplicables durante la pasada *backward*
- Se activa poniendo el código dentro del entorno

```
with tf.GradientTape() as tape:
    (…)
```

@tf.function

- Se trata de un decorador que aplicado a una función habilita varias características
  - Optimización del rendimiento
    - Acelera inferencia y entrenamiento
    - Exportación del modelo al final del entrenamiento
    - La primera vez que se ejecuta se genera un grafo de la computación que se utiliza para acelerar ejecuciones posteriores

Modules, layers, tensor, variables & models

- Como en Pytorch, existen varias abstracciones que actúan como contenedores o bloques constructores de modelos de ML
  - *Module:* Similar al concepto homónimo de Pytorch
    - Los modules sirven de contenedores para los modelos
  - *Layers* predefinidas: Evitan tener que definir tipos de capas comunes desde cero mediante tensores

Keras

- Es un API de nivel superior de TF
- Proporciona bloques constructores de alto nivel para aplicaciones de ML
  - Interfaz sencilla y consistente
  - Minimizar el código necesario para casos de uso comunes
  - Mejora la legibilidad del código

Keras: Layers y Models

- Layers: Encapsulan una capa de un modelo de ML: un estado (pesos) y alguna computación (call)
  - Los parámetros pueden ser entrenables o no
  - Las capas se pueden componer recursivamente
  - También se pueden usar para tareas de preprocesado
- Models: Son agrupaciones de capas
  - El modelo *Sequential* es el más común, se usa para agrupar una secuencia de capas
    - Arquitecturas más comunes se componen usando la *Keras functional API*
  - Proporciona:
    - Método fit: para entrenar el modelo
    - Método predict: para generar predicciones (inferencia) en base a *samples* de entrada
    - Método evaluate: para devolver la función de pérdida y otras métricas generadas en el momento de la compilación (*compile*) del modelo

Otros componentes de Keras

- Optimizers
- Metrics
- Losses
- Utilidades de carga de datos

Bucles de entrenamiento: Contexto

- Estructura de un script de entrenamiento
  1. Importar y procesar un conjunto de datos (*dataset*). Separar en:
     1. Entrenamiento
     2. Validación
  2. Definir la arquitectura del modelo e instanciarlo
  3. Bucle de entrenamiento
     1. Inferencia
     2. Cálculo de la función de pérdida
     3. Cálculo de los gradientes
     4. Aplicación de los gradientes a los parámetros del modelo
  4. Validación de la precisión del modelo

Bucle de entrenamiento en TF

```
model=keras.Sequential([layer1, layer2, layer2 …])
model.compile(optimizer="optname", loss="lossfuncname", metrics=['metric1', metric2, …])

for epoch in range(num_epochs):
        for i in range(0,len(train_data, batch_size):
            with tf.GradientTape as tape:
                predictions= model(batch_data)
                loss=tf.keras.loses.somelossfunction(batch_labels,predictions
            gradients = tape.gradient(loss, model.trainable_variables)
            optimizer.apply_gradients(zip(gradients, model.trainable_variables))

model.save('filename')
```

- Guía Keras: https://www.tensorflow.org/tutorials/quickstart/beginner?hl=es-419

- Guía TF: https://www.tensorflow.org/tutorials/quickstart/advanced?hl=es-419

Keras Distribution API

- Es una nueva característica disponible en Keras 3
  - Mediados de 2023
- Disponible para distribuir entrenamientos ocn varios backends: TensorFlow, Jax, Pytorch
  - Por ahora, solo implementada para Jax
- Desacopla la definición de la aplicación (script de entrenamiento) de las directivas de reparto
  - Útil para cambiar fácilmente de entornos de entrenamiento centralizados a distribuidos
  - Hacer uso de un número variable de recursos

- https://keras.io/guides/distribution/
- https://keras.io/api/distribution/

Keras Distribution API

- Conceptos clave
  - **keras.distribution.DeviceMesh**: representa el Cluster de dispositivos computacionales
  - **keras.distribution.TensorLayout**: representa la política de distribución de Tensores entre dispositivos
- Más control

```python
# Retrieve the local available gpu devices.
devices = jax.devices("gpu")  # Assume it has 8 local GPUs.

# Define a 2x4 device mesh with data and model parallel axes
mesh = keras.distribution.DeviceMesh(
    shape=(2, 4), axis_names=["data", "model"], devices=devices
)

# A 2D layout, which describes how a tensor is distributed across the
# mesh. The layout can be visualized as a 2D grid with "model" as rows and
# "data" as columns, and it is a [4, 2] grid when it mapped to the physical
# devices on the mesh.
layout_2d = keras.distribution.TensorLayout(axes=("model", "data"), device_mesh=mesh)

# A 4D layout which could be used for data parallel of a image input.
replicated_layout_4d = keras.distribution.TensorLayout(
    axes=("data", None, None, None), device_mesh=mesh
)
```

Keras Distribution API

- **keras.distribution**:
  - Proporciona un API simplificada que ya implementa estrategias de entrenamiento distribuido comunes
    - keras.distribution.DataParallel
    - keras.distribution.ModelParallel

```
# Create DataParallel with list of devices.
# As a shortcut, the devices can be skipped,
# and Keras will detect all local available devices.
# E.g. data_parallel = DataParallel()
data_parallel = keras.distribution.DataParallel(devices=devices)

# Or you can choose to create DataParallel with a 1D `DeviceMesh`.
mesh_1d = keras.distribution.DeviceMesh(
    shape=(8,), axis_names=["data"], devices=devices
)
data_parallel = keras.distribution.DataParallel(device_mesh=mesh_1d)

inputs = np.random.normal(size=(128, 28, 28, 1))
labels = np.random.normal(size=(128, 10))
dataset = tf_data.Dataset.from_tensor_slices((inputs, labels)).batch(16)

# Set the global distribution.
keras.distribution.set_distribution(data_parallel)
```

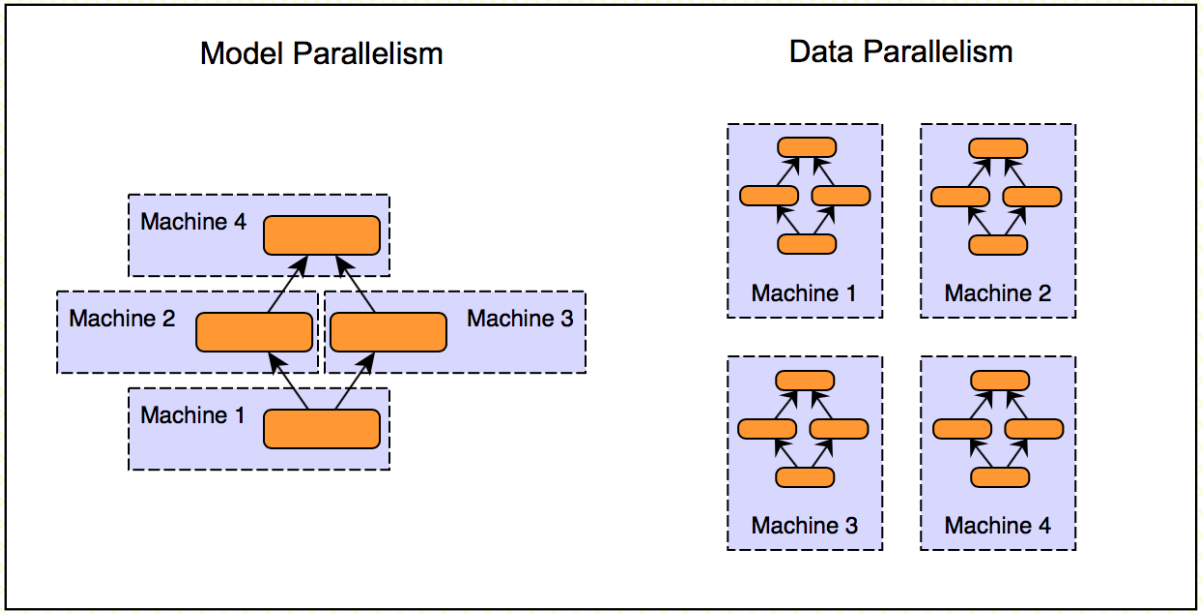
Keras Distribution API

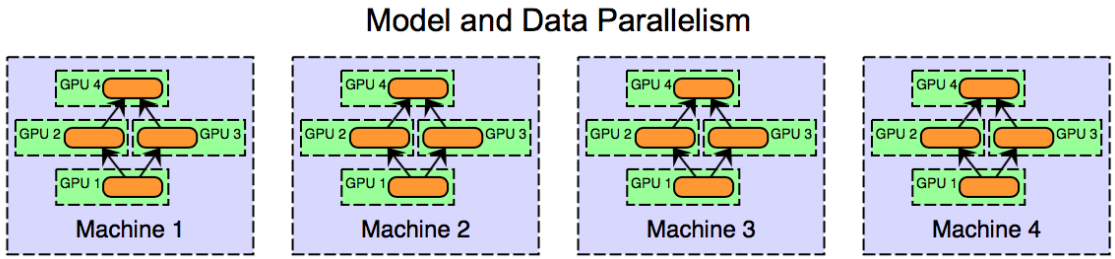
- **keras.distribution.LayoutMap**:
  - Permite tener más control sobre la distribución de los pesos sobre los dispositivos disponibles
    - Permitiendo especifica un TensorLayout
  - Su uso está prescrito para ajustar el rendimiento de la estrategia ModelParallel

```
mesh_2d = keras.distribution.DeviceMesh(
    shape=(2, 4), axis_names=["data", "model"], devices=devices
)
layout_map = keras.distribution.LayoutMap(mesh_2d)
# The rule below means that for any weights that match with d1/kernel, it
# will be sharded with model dimensions (4 devices), same for the d1/bias.
# All other weights will be fully replicated.
layout_map["d1/kernel"] = (None, "model")
layout_map["d1/bias"] = ("model",)

# You can also set the layout for the layer output like
layout_map["d2/output"] = ("data", None)

model_parallel = keras.distribution.ModelParallel(
    mesh_2d, layout_map, batch_dim_name="data"
)

keras.distribution.set_distribution(model_parallel)
```

Entrenamiento distribuido

- Paralelismo de datos
- Paralelismo de modelo
- Paralelismo híbrido

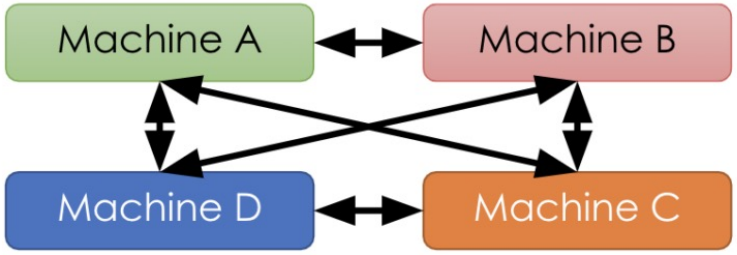




citic

Centralizado vs descentralizado

- Copias espejo (*Mirror*)
  - allreduce
- Parameter Server
  - 1 PS – n trabajadores
  - n PS – n trabajadores



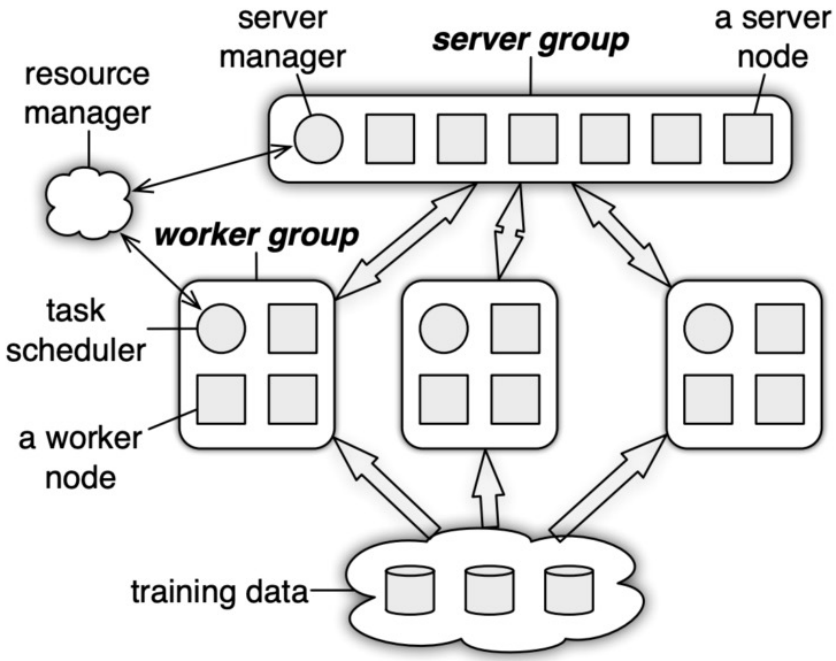
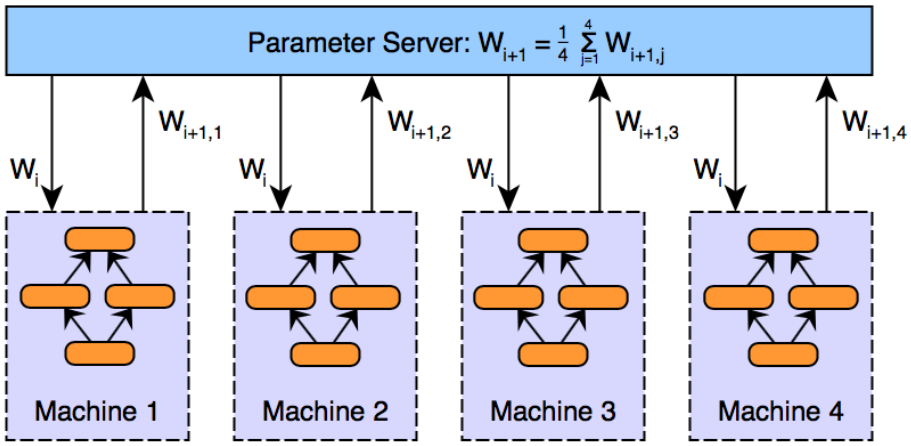


Figure 4: Architecture of a parameter server communicating with several groups of workers.

http://www.juyang.co/distributed-model-training-ii-parameter-server-and-allreduce/

citic

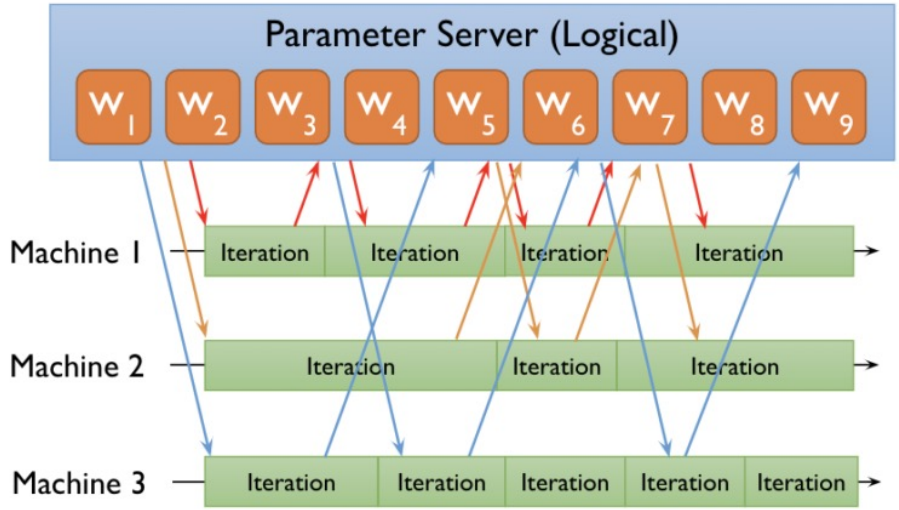
Promediado de pesos



citic

Promediado síncrono vs asíncrono

- Síncrono: Hay que esperar por todos los trabajadores para promediar los pesos
  - La sincronización supone un cuello de botella
- Asíncrono: El promediado se produce sin asegurar la sincronización de los trabajadores
  - Convergencia más lenta del modelo



citic

Estrategias avanzadas

- Paralelismo de tensores
- Paralelismo multinivel
  - Data + Model + Tensor
- Estrategias ad-hoc para ciertas arquitecturas de modelo
- Estrategias avanzadas como Zero-DeepSpeed

Actividad: Conf. y prueba del entorno

- Creación y configuración del entorno
  - Conectarse a FT3
  - compute –gpu
  - cd $STORE/CFR24/scripts
  - source createVENVTF.sh
  - source $STORE/mytf/bin/activate
- Comprobar la instalación
  - python

    >> import tensorflow as tf

    >> print("Num GPUs Available: ", len(tf.config.list_physical_devices('GPU')))

Explotación de una GPU

- TF puede utilizar una GPU de forma totalmente transparente
  - No necesita cambios en el código
- En FT3 es necesario estar en un *compute node* con una GPU disponible: *compute –gpu*

```
import tensorflow as tf
print("Num GPUs Available:
",len(tf.config.list_physical_devices('GPU'))
)
```

Fuente: https://www.tensorflow.org/guide/

Explotación de una GPU: modo de explotación

- Cuando haya una CPU y una GPU disponibles
  - TF prioriza la GPU si la operación a ejecutar tiene una implementación específica para GPU
  - En caso contrario, se ejecuta en la CPU
- El siguiente código permite saber dónde se ejecuta una función (ej. matmul)

```
tf.debugging.set_log_device_placement(True)

# Create some tensors
a = tf.constant([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
b = tf.constant([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]])
c = tf.matmul(a, b)

print(c)
```

Exploración de una GPU: modo de explotación

- Existe una forma de forzar una ubicación para unos cálculos

```
tf.debugging.set_log_device_placement(True)

# Place tensors on the CPU
with tf.device('/CPU:0'):
    a = tf.constant([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
    b = tf.constant([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]])

# Run on the CPU
c = tf.matmul(a, b)
print(c)
```

Explotación de una GPU: Límite de memoria

- Por defecto, TF se asigna a toda la memoria de todas la GPUs visibles
  - Podemos usar el método *set_visible_devices* para limitarlo

```
gpus = tf.config.list_physical_devices('GPU')
if gpus:
    # Restrict TensorFlow to only use the first GPU
    try:
            tf.config.set_visible_devices(gpus[0], 'GPU’)
            logical_gpus = tf.config.list_logical_devices('GPU’)
            print(len(gpus), "Physical GPUs,", len(logical_gpus), "Logical GPU")
    except RuntimeError as e:
            # Visible devices must be set before GPUs have been initialized
            print(e)
```

Explotación de una GPU: Límite de memoria

- También podemos usar el mecanismo experimental *set_memory_growth*
  - Hace un aumento paulatino de la reserva de memoria bajo demanda

```
(…)
for gpu in gpus:
        tf.config.experimental.set_memory_growth(gpu, True)
        logical_gpus = tf.config.list_logical_devices('GPU')
        print(len(gpus), "Physical GPUs,", len(logical_gpus), "Logical GPUs")
(…)
```

Explotación de una GPU: Límite de memoria

- También se puede establecer un límite fijo a través del mecanismo *set_logical_device_configuration*

```
# Place tensors on the CPU
with tf.device('/CPU:0'):
    a = tf.constant([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
    b = tf.constant([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]])

    gpus = tf.config.list_physical_devices('GPU')
    if gpus:
    # Restrict TensorFlow to only allocate 1GB of memory on the first GPU
    try:
        tf.config.set_logical_device_configuration(
        gpus[0],
        [tf.config.LogicalDeviceConfiguration(memory_limit=1024)])
        logical_gpus = tf.config.list_logical_devices('GPU')
        print(len(gpus), "Physical GPUs,", len(logical_gpus), "Logical GPUs")
    except RuntimeError as e:
        # Virtual devices must be set before GPUs have been initialized
        print(e)
```

Explotación de una GPU única en un sistema multi-GPU

```
tf.debugging.set_log_device_placement(True)

try:
   # Specify an invalid GPU device
   with tf.device('/device:GPU:0’):
   …
except RuntimeError as e:
   print(e)
```

Tensorflow ClusterResolver

- Es una librería que permite tener acceso a los recursos computacionales reservados en entornos de supercomputación
- Se asocia con el framework ClusterSpec
- Soporta varios sistemas:
  - GCE
  - Kubernetes
  - **Slurm**
  - ...
- Fuente:[Module: tf.distribute.cluster_resolver | TensorFlow v2.11.0]

Tensorflow ClusterResolver

- SlurmClusterResolver es el que corresponde con Slurm el sistema de colas de FT3
- Devuelve un objeto ClusterResolver que puede ser usado directamente en TF
- El método cluster_spec devuelve un objeto ClusterSpec para ser usado en Distributed TF

```
tf.distribute.cluster_resolver.SlurmClusterResolver(
    jobs=None,
    port_base=8888,
    gpus_per_node=None,
    gpus_per_task=None,
    tasks_per_node=None,
    auto_set_gpu=True,
    rpc_layer='grpc'
)
```

Actividad: ClusterResolver

- https://github.com/diegoandradecanosa/CFR24/tree/main/tf_dist/002

Distribute.strategy de TF: CrossDeviceOps

- Clase para seleccionar la implementación a usar para los algoritmos de
  - Reducción
  - Broadcasting
- Es uno de los parámetros que podemos pasar a la MirroredStrategy
- Implementaciones:
  - tf.distribute.ReductionToOneDevice
    - Copia todos los valores a un dispositivo donde se hará la reducción de forma centralizada
  - tf.distribute.NcclAllReduce
    - Usa la implementación de Nvidia NCCL para el all reduce
  - tf.distribute.HierarchicalCopyAllReduce
    - Utiliza un algoritmo de reducción jerárquica
    - Pensado para Nvidia-DGX1
      - Asume que las GPUs están interconectadas como en ese tipo de nodo

Fuente: https://www.tensorflow.org/api_docs/python/tf/distribute/CrossDeviceOps

Distribute.strategy de TF: DistributedDataSet

- Clase que permite definir un *dataset* distribuido entre varios nodos
  - Apropiado para su uso con el módulo tf.distribute.strategy
- Dos APIs diferentes:
  - tf.distribute.Strategy.experimental_distribute_dataset(dataset)
    - Más sencillo de utilizar si tenemos un dataset convencional
  - tf.distribute.Strategy.distribute_datasets_from_function(dataset_fn)
    - Más difícil de utilizar pero más flexible

Fuente: https://www.tensorflow.org/api_docs/python/tf/distribute/DistributedDataset

Distribute.strategy de TF: DistributedDataSet

- Concepto más amplio: Dataset sharding
  - Distribución del conjunto de datos entre varios nodos

Distributed training en TF

- El uso de hardware *en paralelo* puede reducir el tiempo de entrenamiento
- La paralelización del entrenamiento requiere esfuerzo por parte del programador
  - El uso de una GPU o una CPU sí que no requiere ese esfuerzo
- La paralelización requiere que TF sepa cómo coordinar el trabajo de varios trabajadores (*workers*)

Fuente: https://www.youtube.com/watch?v=S1tN9a4Proc

Distribute.Strategy de TF

- Tf.distribute.Strategy es una API de TF para distribuir el entrenamiento entre múltiples GPUs, máquinas o TPUs
- Permite ejecutar los entrenamientos en paralelo ávidamente (*eagerly)* o siguiendo una estrategia de grafo (usando tf.function)
  - Ávidamente -> depuración
  - Tf.function -> Recomendado
- Fuente: Distributed training with TensorFlow | TensorFlow Core

Distributed training en TF

- Categorías de algoritmos paralelos de TF
  - Paralelismo de datos (data paralellism)
  - Paralelismo de modelo (model paralellism)

Distributed training en TF: Data parallelism

- Paralelismo de datos (data paralellism)

  model.fit(x,y,batch_size=32)

  Dimensiones del entrenamiento:
  - Epoch (procesado de todo el *dataset*). Una pasada completa del *dataset*
  - En cada *step* procesamos *batch_size* elementos del data_set a la vez
    - Incrementar el *batch_size* está limitado por la memoria de una GPU
      - Incrementarlo mejora el rendimiento -> Podemos hacer más cosas en paralelo
    - Usando varios *workers*
      - Podemos seguir aumentando el *batch_size*
        - Se divide efectivamente entre varias GPUs
      - Y acortar el entrenamiento
- Usando varios *workers*
  - Cada uno procesa un *step* del entrenamiento de forma independiente calculando sus propios gradientes
  - Estos gradientes son *reducidos* (promediados) entre todos los trabajadores y usados en la actualización de los pesos

Distribute.Strategy de TF

- Estrategias disponibles en TF:
  - Síncrono
    - OneDeviceStrategy -> https://www.tensorflow.org/api_docs/python/tf/distribute/OneDeviceStrategy
    - MirroredStrategy
    - TPUStrategy
    - MultiWorkerMirroredStrategy
  - Asíncrono
    - ParameterServerStrategy
    - CentralStorageStrategy
- https://www.tensorflow.org/guide/distributed_training
- https://www.tensorflow.org/api_docs/python/tf/distribute/Strategy

Distribute.StrategyExtended

- API adicional para algoritmos que necesitan ser *distribution-aware*

- https://www.tensorflow.org/api_docs/python/tf/distribute/StrategyExtended

Distribute.Strategy de TF

Grado de soporte de estrategias en TF en diversos escenarios

Training API	MirroredStrategy	TPUStrategy	MultiWorkerMirroredStrategy	CentralStorageStrategy	ParameterServerStrategy
Keras Model.fit	Supported	Supported	Supported	Experimental support	Experimental support
Custom training loop	Supported	Supported	Supported	Experimental support	Experimental support
Estimator API	Limited Support	Not supported	Limited Support	Limited Support	Limited Support

Distribute.Strategy de TF

Grado de soporte de estrategias en TF en diversos escenarios

Training API	MirroredStrategy	TPUStrategy	MultiWorkerMirroredStrategy	CentralStorageStrategy	ParameterServerStrategy
Keras Model.fit	Supported	Supported	Supported	Experimental support	Experimental support
Custom training loop	Supported	Supported	Supported	Experimental support	Experimental support
Estimator API	Limited Support	Not supported	Limited Support	Limited Support	Limited Support

Distribute.Strategy de TF: MirroredStrategy

- La MirroredStrategy soporta entrenamiento distribuido síncrono en múltiples GPUs en un nodo
- Se crea una réplica por cada GPU
  - Juntas forman una única variable conceptual llamada MirroredVariable
  - Se mantiene la coherencia aplicando actualizaciones similares en todas
    - Como si fuese un espejo (mirror)
    - Implementaciones eficientes de algoritmos all-reduce

```
mirrored_strategy = tf.distribute.MirroredStrategy(devices=["/gpu:0", "/gpu:1"])
```

Distribute.Strategy de TF: MirroredStrategy

- Cada GPU realiza la *forward pass* en un subconjunto diferente de los datos de entrada para calcular la *loss function*
- Cada GPU calcula sus propios gradientes basándose en la *loss function* calculada localmente
- Se realiza la agregación global (promedio) de estos gradientes a través de un algoritmo *all-reduce*
- Se actualizan los pesos usando los gradientes resultantes
  - Todos los dispositivos tendrán una copia sincronizada (espejo) del modelo entrenado

Actividad: Entrenamiento 1 nodo – 2 GPUs

- https://github.com/diegoandradecanosa/CFR24/tree/main/tf_dist/003

Actividad: Entrenamiento 1 nodo – 2 GPUs

- Solución de problemas comunes
  - Si se cuelga el kernel hay que reiniciarlo
    - Kernel -> Restart kernel
    - O la combinación de teclas “0+0”
  - Si falla la ejecución por problemas de uso de memoria, entonces podemos matar los procesos que hayan quedado ejecutándose en la GPU
    - nvidia-smi Al final del comando habrá una lista de procesos
    - Eliminarlos con kill -9 *pid*

Distribute.Strategy de TF: TPUStrategy

- Específica para Google TPUs
- Similar a MirroredStrategy
- Usa una implementación específica de las operaciones all-reduce optimizada para TPUs

```
cluster_resolver = tf.distribute.cluster_resolver.TPUClusterResolver(
    tpu=tpu_address)
tf.config.experimental_connect_to_cluster(cluster_resolver)
tf.tpu.experimental.initialize_tpu_system(cluster_resolver)
tpu_strategy = tf.distribute.TPUStrategy(cluster_resolver)
```

Distribute.Strategy de TF: MultiWorkerMirroredStrategy

- MultiWorkerMirroredStrategy es similar a MirroredStrategy pero con soporte para varios nodos
  - Crea copias de todas las variables en todos los trabajadores y en todos los dispositivos

```
communication_options = tf.distribute.experimental.CommunicationOptions(
implementation=tf.distribute.experimental.CommunicationImplementation.NCCL)
strategy =
tf.distribute.MultiWorkerMirroredStrategy(communication_options=communication_options)
```

Hay 2 opciones para las Comunicaciones entre dispositivos:

.RING: Basado en RPC, válido para CPU y GPU

.NCCL: Específico para GPU, mejor rendimiento cuando se puede utilizar

.AUTO: Deja a TF elegir el mejor método disponible

Distribute.Strategy de TF: MultiWorkerMirroredStrategy

- El uso de múltiples nodos requiere configurar la variable de entorno: TF_CONFIG.
  - Tiene estructura de diccionario
  - Dos componentes:
    - La definición de un **cluster**
      - **Diccionario con listas de nodos (host:puerto) de distintos tipos:**
        - **Ps: servidores**
        - **Workers: trabajadores**
    - La definición de cada tarea (**task**)
      - Type: worker/ps
      - Index

```
os.environ["TF_CONFIG"] = json.dumps({
    "cluster": {
        "worker": ["host1:port", "host2:port", "host3:port"],
        "ps": ["host4:port", "host5:port"]
    },
   "task": {"type": "worker", "index": 1}
})
```

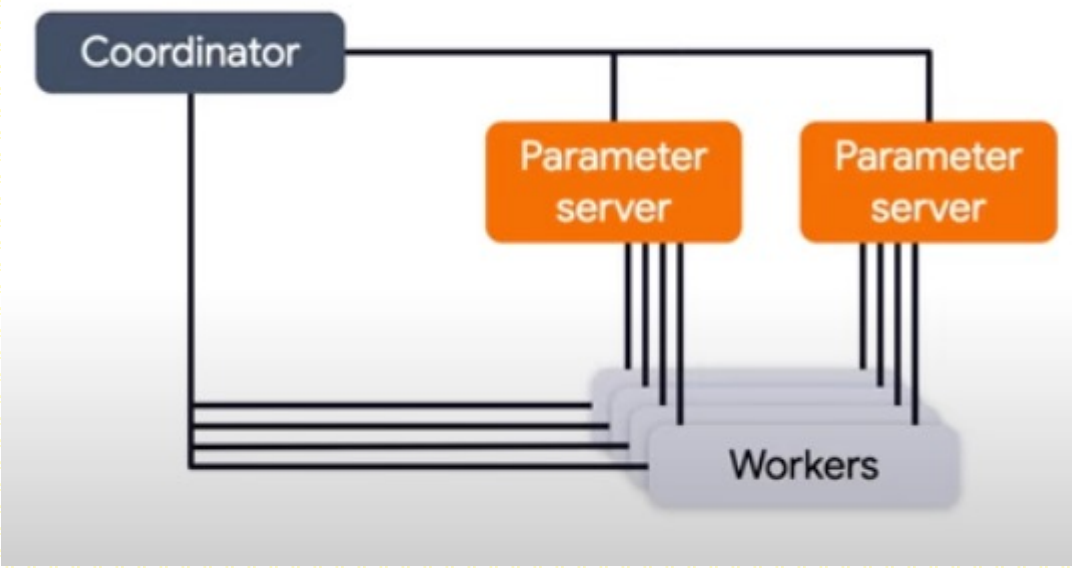
Fuente: Distributed training with TensorFlow | TensorFlow Core

Actividad: Entrenamiento 2 nodos – 2 GPUs

- https://github.com/diegoandradecanosa/CFR24/tree/main/tf_dist/004

Distribute.Strategy de TF: ParameterSeverStrategy

- Es un tipo de entrenamiento multimodo asíncrono
  - Reduce el cuello de botella del all-reduce en las estrategias síncronas
  - Recomendable para usar un nodo alto de trabajadores
- Los nodos implicados se dividen en:
  - *Workers (tf.distribute.Server)*
  - *Parameter servers (tf.distribute.Server)*
  - Un *coordinator (tf.distribute.experimental.coordinator.ClusterCoordinator)*
    - *Usa la ParameterServerStrategy para definir el paso de entrenamiento y usar un ClusterCoordinator que envía pasos de entrenamiento a los trabajadores*

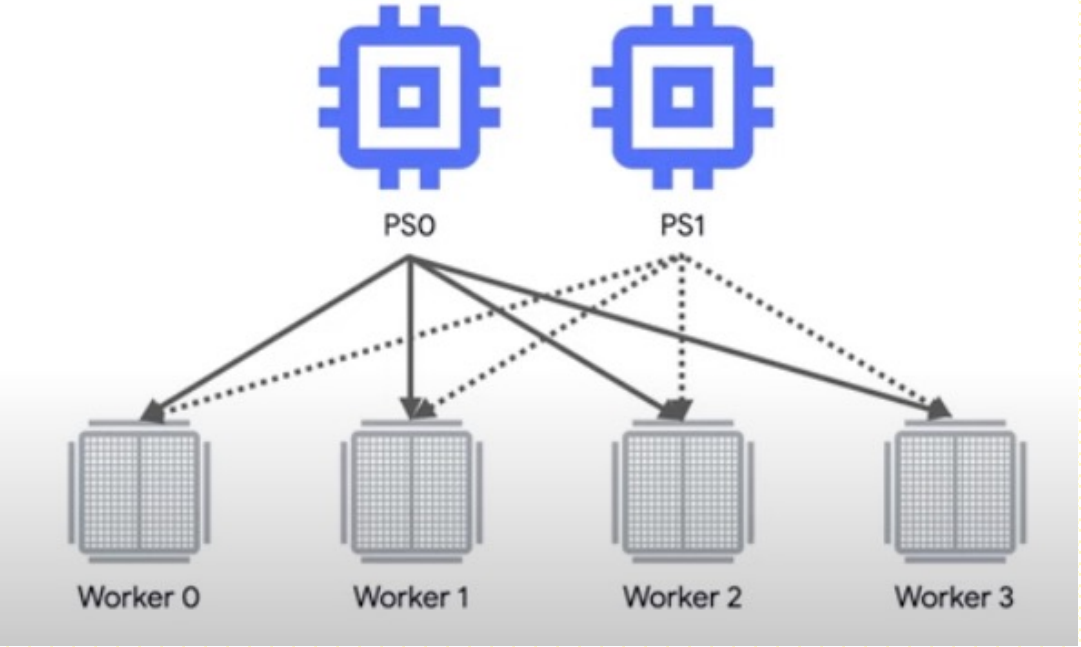


Distribute.Strategy de TF: ParameterSeverStrategy

- Soporta dos modos de entrenamiento
  - Keras Model.fit
  - Bucles de entrenamiento definidos por el usuario
- Abstracciones de Model.fit
  - Cluster, Jobs y Tasks
- Con PS, también tenemos:
  - Un Coordinator job (called *chief*)
  - Varios Worker jobs (llamados *workers*)
  - Varios PS Jobs (llamados *ps*)

Distribute.Strategy de TF: ParameterSeverStrategy

- Cada *worker* pide la última copia de los parámetros a cada uno de los *parameter servers*
  - Los parámetros están distribuidos entre varios servidores
- Cada *worker* calcula los gradientes de acuerdo a un subconjunto del *dataset*
- Los *workers* envían los parámetros de vuelta a los *parameter servers* donde se integran (reducen)



citic

Preparación del Clúster

- Components: 1 Coordinator (type *chief*), N PS *(ps),* N Workers *(worker*), y puede que una tarea Evaluator
- La tarea de coordinación necesita conocer las direcciones y puertos de todas las tareas Server (PS y Workers), pero no del Evaluator
- Las tareas Server deben saber en qué puerto escuchar
- La tarea Evaluator no tiene por qué conocer la configuración del Clúster
- La estrategia PS usará todas las GPUs disponibles en cada nodo
  - Todos deben tener el mismo número de GPUs

PS con Keras model.fit(): Esqueleto

```
variable_partitioner = (
    tf.distribute.experimental.partitioners.MinSizePartitioner(
        min_shard_bytes=(256 << 10),
        max_shards=NUM_PS))

strategy = tf.distribute.experimental.ParameterServerStrategy(
    tf.distribute.cluster_resolver.TFConfigClusterResolver(),
    variable_partitioner=variable_partitioner)
coordinator = tf.distribute.experimental.coordinator.ClusterCoordinator(
    strategy)

with strategy.scope():
            //model definition
            model.compile(…)

model.fit(…)
```

Concepto relacionado: Variable sharding

- Consiste en dividir una variable en variables más pequeñas llamadas *shards*
- Útil para:
  - Reducir consumo de red
  - Distribuir la carga de computación y almacenamiento de una variable
    - Útil, por ejemplo, para *embeddings* muy grandes que no caben en la memoria de un dispositivo
- Cómo hacerlo: Pasando un *variable_partitioner* al construir un objeto *ParameterServerStrategy*
- El particionador entonces se llamará cada vez que se cree una variable, y devuelve un nuevo de shards particionando en cada dimensión de la variable
- Varios particionadores disponibles: [Min/Max/Fixed]SizePartitioner

PS con bucle de usuario

- Creación de una instancia de un ClusterCoordinator para enviar trabajos (normalmente *steps* de entrenamiento) para su ejecución en otros *workers*
  - Opcional trabajando con Keras Model.fit
  - Necesario con bucles de entrenamiento de usuario

Definición del *step* de entrenamiento

Wrapper tf.function

```
@tf.function
def step_fn(iterator):

    def replica_fn(batch_data, labels):
        with tf.GradientTape() as tape:
            pred = model(batch_data, training=True)
            per_example_loss = tf.keras.losses.BinaryCrossentropy(
                reduction=tf.keras.losses.Reduction.NONE)(labels, pred)
            loss = tf.nn.compute_average_loss(per_example_loss)
            model_losses = model.losses
            if model_losses:
                loss += tf.nn.scale_regularization_loss(tf.add_n(model_losses))
        gradients = tape.gradient(loss, model.trainable_variables)

        optimizer.apply_gradients(zip(gradients, model.trainable_variables))

        actual_pred = tf.cast(tf.greater(pred, 0.5), tf.int64)
        accuracy.update_state(labels, actual_pred)
        return loss

    batch_data, labels = next(iterator)
    losses = strategy.run(replica_fn, args=(batch_data, labels))
    return strategy.reduce(tf.distribute.ReduceOp.SUM, losses, axis=None)
```

Inferencia para un batch

1. Siguiente batch
2. Ejecutar step de entrenamiento para cada batch
3. Reducción de los resultados en cada trabajador

Definición del ClusterCoordinator (I)

Instancia del coordinador

Replica el conjunto de datos entre los trabajadores

```
coordinator = tf.distribute.coordinator.ClusterCoordinator(strategy)

...

@tf.function
        def per_worker_dataset_fn():
                return strategy.distribute_datasets_from_function(dataset_fn)

per_worker_dataset = coordinator.create_per_worker_dataset(per_worker_dataset_fn)
per_worker_iterator = iter(per_worker_dataset)
```

Definición del ClusterCoordinator (II)

```
num_epochs = 4
steps_per_epoch = 5
for i in range(num_epochs):
  accuracy.reset_states()
  for _ in range(steps_per_epoch):
    coordinator.schedule(step_fn, args=(per_worker_iterator,))
  # Wait at epoch boundaries.
  coordinator.join()
  print("Finished epoch %d, accuracy is %f." % (i, accuracy.result().numpy()))
...

loss = coordinator.schedule(step_fn, args=(per_worker_iterator,))
print("Final loss is %f" % loss.fetch())
```

Para cada *step*

Envío de *steps* a los trabajadores

Punto de sincronización

Central Storage Strategy

- Es una estrategia de tipo servidor de parámetros que pone todas las variables en el mismo dispositivo

```
strategy = tf.distribute.experimental.CentralStorageStrategy()
# Create a dataset
ds = tf.data.Dataset.range(5).batch(2)
# Distribute that dataset
dist_dataset = strategy.experimental_distribute_dataset(ds)

with strategy.scope():
        @tf.function
        def train_step(val):
                return val + 1

# Iterate over the distributed dataset
for x in dist_dataset:
        # process dataset elements
        strategy.run(train_step, args=(x,))
```

https://www.tensorflow.org/api_docs/python/tf/distribute/experimental/CentralStorageStrategy

Actividad: ParameterServer Ejemplo simple

- https://github.com/diegoandradecanosa/CFR24/tree/main/tf_dist/005

Actividad: Keras 3 entrenamiento distribuido

cd $STORE/CFR24/tf_dist/000

source $STORE/CFR24/scripts/interactive_1node_2gpus.sh

source $STORE/mytf/bin/activate

pip install --upgrade keras

pip install --upgrade "jax[cuda12_pip]" -f https://storage.googleapis.com/jax-releases/jax_cuda_releases.html