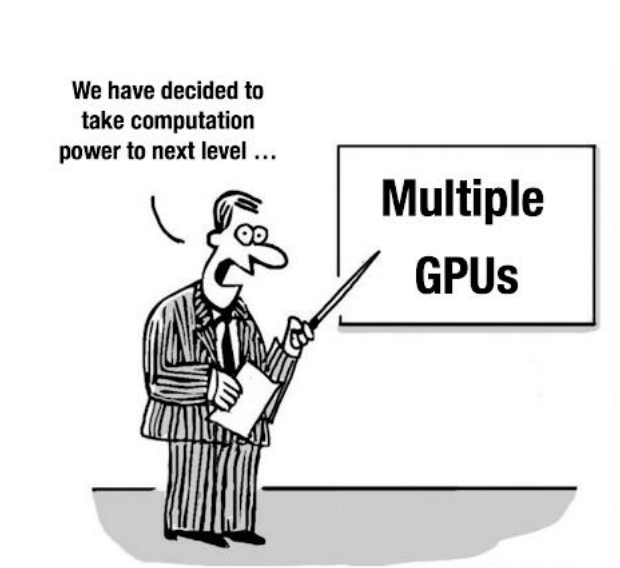
Entrenamiento distribuido basado en RPC

Diego Andrade Canosa

Roberto López Castro

Índice

- Distributed Data Parallel vs. RPC
- Backends PyTorch
- Funcionalidades principales RPC
- Ejemplos de uso



citic

Data Parallel

El conjunto de datos es muy grande, por lo que tiene que procesarse en paralelo

Distributed Data Parallel (DDP)

Model Parallel

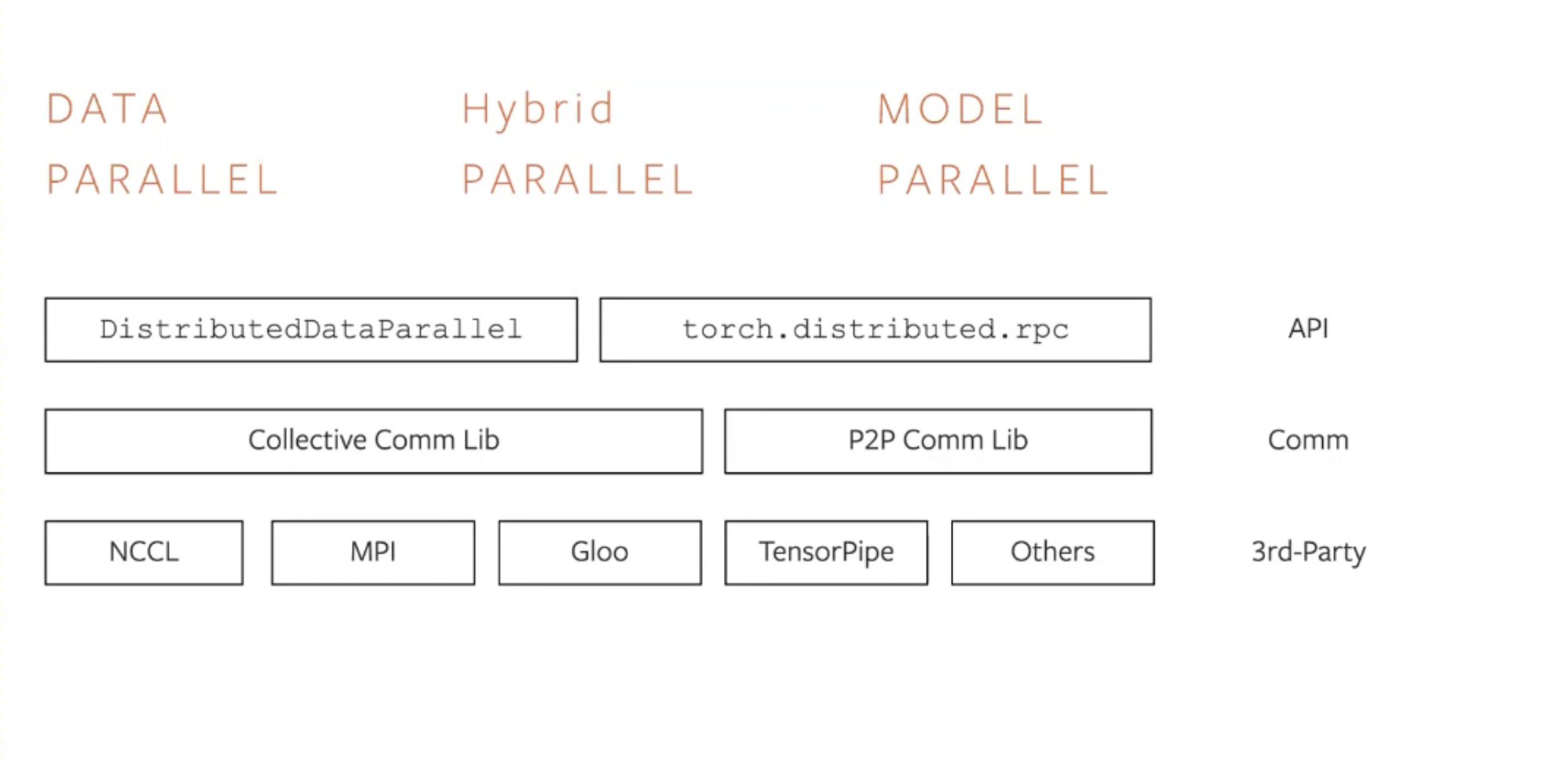
El modelo es muy grande, por lo que no coge en un único dispositivo o máquina

Remote Procedure Call (RPC)

Remote Procedure Call

- El framework de RPC distribuido facilita la ejecución remota de funciones, admite la referencia de objetos remotos sin copiar los datos reales y proporciona APIs de autograd y optimizer para ejecutar de manera transparente la propagación hacia atrás y actualizar parámetros a través de los límites de RPC. Estas características se pueden categorizar en cuatro conjuntos de APIs.

DDP vs. RPC



citic

Backends

Backend	`gloo`		`mpi`		`nccl`	
Device	CPU	GPU	CPU	GPU	CPU	GPU
send	✓	✘	✓	?	✘	✓
recv	✓	✘	✓	?	✘	✓
broadcast	✓	✓	✓	?	✘	✓
all_reduce	✓	✓	✓	?	✘	✓
reduce	✓	✘	✓	?	✘	✓
all_gather	✓	✘	✓	?	✘	✓
gather	✓	✘	✓	?	✘	✓

Backends. ¿Qué backend debería usar?

Regla general

- Usa el backend NCCL para entrenamiento distribuido en GPU.
- Usa el backend Gloo para entrenamiento distribuido en CPU.

Hosts de GPU con interconexión InfiniBand

- Usa NCCL, ya que es el único backend que actualmente admite InfiniBand y GPUDirect.

Hosts de GPU con interconexión Ethernet

- Usa NCCL, ya que actualmente ofrece el mejor rendimiento de entrenamiento distribuido en GPU, especialmente para entrenamiento distribuido en un solo nodo con múltiples procesos o en varios nodos. Si encuentras algún problema con NCCL, utiliza Gloo como opción de respaldo.

Backends. ¿Qué backend debería usar?

Hosts de CPU con interconexión InfiniBand

- Si tu InfiniBand ha habilitado IP sobre IB, utiliza Gloo; de lo contrario, utiliza MPI en su lugar.

Hosts de CPU con interconexión Ethernet

- Utiliza Gloo, a menos que tengas razones específicas para utilizar MPI.

TensorPipe

- Por debajo, PyTorch RPC se basa en TensorPipe como backend de comunicación. PyTorch RPC extrae todos los Tensores de cada solicitud o respuesta en una lista y empaqueta todo lo demás en una carga binaria. Luego, TensorPipe elegirá automáticamente un canal de comunicación para cada Tensor según el tipo de dispositivo del Tensor y la disponibilidad del canal tanto en el llamador como en el receptor. Los canales existentes de TensorPipe incluyen NVLink, InfiniBand, SHM, CMA, TCP, etc.

- Lo veremos a continuación...

Distributed RPC Framework

WARNING

CUDA support was introduced in PyTorch 1.9 and is still a **beta** feature. Not all features of the RPC package are yet compatible with CUDA support and thus their use is discouraged. These unsupported features include: RRefs, JIT compatibility, dist autograd and dist optimizer, and profiling. These shortcomings will be addressed in future releases.

RPC - Funcionalidades

- **Remote Execution** – Ejecutar funciones definidas por el usuario o módulos de forma remota
- **Remote Reference** – Acceso y referencia a objetos de datos remotos eficientes
- **Distributed Autograd** – Extensión del autograd de Pytorch más allá de los límites locales

Remote Execution

API – Remote Execution

Existen tres principales APIs en RPC:

- rpc_sync() (síncrono),
- rpc_async() (asíncrono), y
- remote() (asíncrono y devuelve una referencia al valor remoto de salida)

Glosario

- **Caller**: proceso o nodo que realiza la llamada remota a través de la comunicación RPC. En otras palabras, es el punto de origen que inicia la comunicación y realiza una solicitud a otro proceso o nodo en la red para ejecutar una función o acceder a un objeto remoto
- **Trainer**: componente o entidad que se encarga de realizar el entrenamiento de un modelo de aprendizaje automático
- Las "**embedding tables**" (tablas de embedding) son una técnica utilizada en el procesamiento de lenguaje natural (NLP) y en sistemas de recomendación para representar datos categóricos o discretos en forma de vectores densos de números reales (embeddings). Estas tablas se utilizan comúnmente para transformar palabras, ítems, o cualquier otra entidad discreta en una representación vectorial continua, que es más adecuada para su procesamiento en algoritmos de aprendizaje automático.

Glosario

- **Parameter server**: El framework del servidor de parámetros es un paradigma en el que un conjunto de servidores almacena parámetros, como "embedding tables" grandes, y varios “trainers” consultan los servidores de parámetros para recuperar los parámetros más actualizados. Estos ”trainers” pueden ejecutar un bucle de entrenamiento localmente y sincronizarse ocasionalmente con el servidor de parámetros para obtener los últimos parámetros.

API – Remote Execution

rpc_sync() se utiliza si el usuario no puede continuar sin el valor devuelto (dependencia)

En caso de que esa dependencia no exista, el usuario puede hacer uso de rpc_async(), obtener un objeto “future”, y esperar en dicho objeto cuando el valor sea necesario.

El método remote() es útil cuando se necesita crear algo de forma remota, pero nunca se necesitará un fetch desde el proceso creador.

Ejemplo: un proceso maestro (driver) configura un parameter server y un trainer. El maestro puede crear una embedding table en el parameter server y luego compartir la referencia a dicha tabla con el trainer, pero el maestro nunca va a usar de nuevo la tabla localmente. En este ejemplo rpc_sync() and rpc_async() no son métodos apropiados ya que siempre implican que el valor retornado va a ser devuelto al maestro inmediatamente o en el futuro.

API – Remote Execution

```
# Por defecto utiliza TensorPipe para la comunicacion
rpc_init("w0", rank=0, world_size=2)

#sincrono, devuelve el resultado
x = torch.zeros(2)
ret = rpc_init("w1", torch.add, args=(x, 1))

def my_add(x,y):
   return x + y

#asincrono, devuelve "future"
fut = rpc_async("w1", my_add, args=(x ,1))

#asincrono, devuelve referencia al resultado
@torch.jit.script
def script_add(x,y):
   return x+y

rref = remote("w1", script_add, args=(x, 1))

shutdown()
```

citic

API – Remote Execution

```
# Por defecto utiliza TensorPipe para la comunicacion
rpc_init("w0", rank=0, world_size=2)

#sincrono, devuelve el resultado
x = torch.zeros(2)
ret = rpc_init("w1", torch.add, args=(x, 1))

def my_add(x,y):
   return x + y

#asincrono, devuelve "future"
fut = rpc_async("w1", my_add, args=(x,1))

#asincrono, devuelve referencia al resultado
@torch.jit.script
def script_add(x,y):
   return x+y

rref = remote("w1", script_add, args=(x, 1))

shutdown()
```

rpc_init():
"w0:" nombre del proceso actual (worker)
rank_id: identificador del proceso
word-size: número total de procesos
Esta función crea un agente corriendo en background. Además, cuando termina asegura que todos los extremos (peers) están listos para comunicarse entre sí.

rpc_sync():
"w1": nombre del proceso destino.
Esta función devuelve el resultado de forma síncrona

rpc_async():
Es la versión asíncrona del método anterior. En este caso se devuelve un objeto "future" y podemos esperar en dicho objeto.
ret = fut.wait()

remote():
También devuelve inmediatamente, pero el objeto devuelto por la función "script_add" es una referencia y su resultado no será buscado por el maestro. Por el contrario, con rpc_async el resultado va a retornar al maestro en algún momento futuro.

API – Remote Execution

```
# Por defecto utiliza TensorPipe para la comunicacion
rpc_init("w0", rank=0, world_size=2)

#sincrono, devuelve el resultado
x = torch.zeros(2)
ret = rpc_init("w1", torch.add, args=(x, 1))

def my_add(x,y):
   return x + y

#asincrono, devuelve "future"
fut = rpc_async("w1", my_add, args=(x,1))

#asincrono, devuelve referencia al resultado
@torch.jit.script
def script_add(x,y):
   return x+y

rref = remote("w1", script_add, args=(x, 1))

shutdown()
```

TensorPipe entiende los Tensores: si intentas combinar un entrenamiento local de PyTorch con bibliotecas de terceros, esas bibliotecas esperan pasar un tipo de datos JSON o definido por el usuario, y en algún momento esos resultados se serializarán. Con RPC, puedes enviar Tensores como argumentos grandes y la comunicación no será un cuello de botella.

TensorPipe != MPI -> TensorPipe intentará seleccionar el mejor canal para ti (TCP si no son servidores diferentes, o NVLINK en caso contrario, por ejemplo) -> Cuando llamas a rpc_init, también puedes especificar las conexiones disponibles (ver https://pytorch.org/docs/stable/rpc.html#backends)

Por lo tanto, **TensorPipe permite la comunicación directa de GPU a GPU, pero no de MPI.**

x puede ser un nn.Linear, por lo que los pesos y el bias se pueden comunicar como un paquete de dos Tensores.

Datos como un objeto Python.

API – Remote Execution

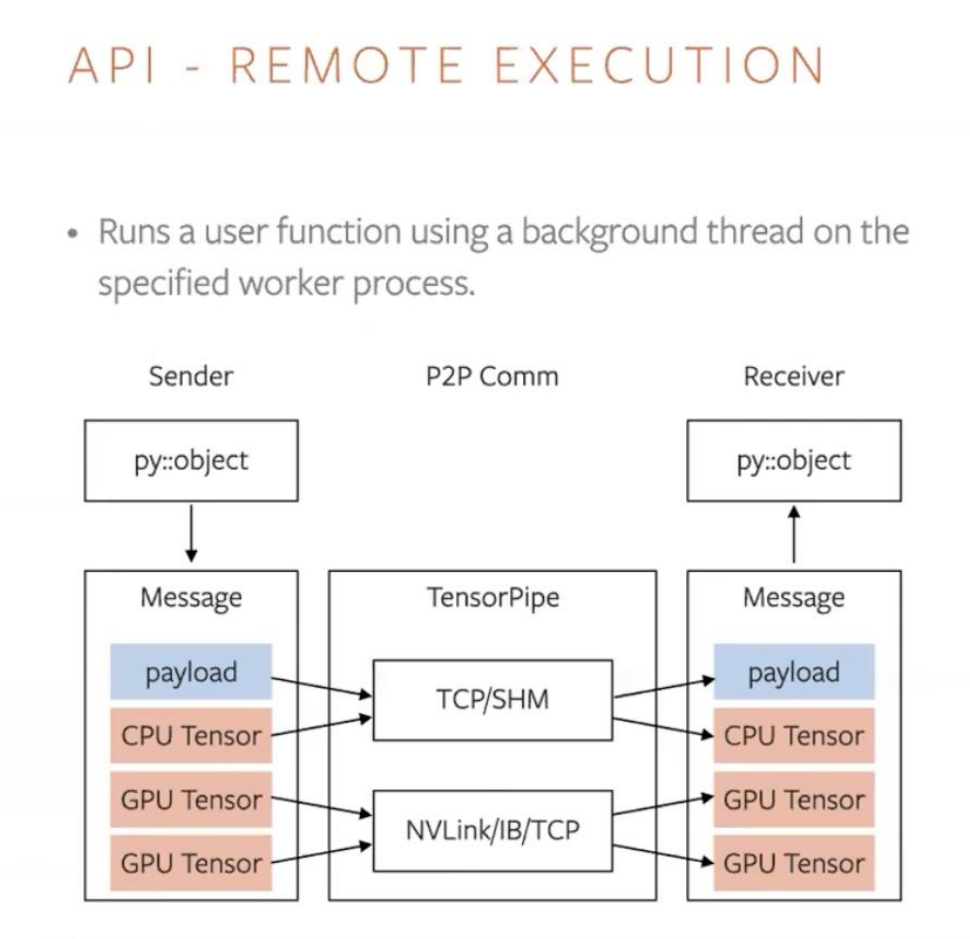
TensorPipe entiende los Tensores: si intentas combinar un entrenamiento local de PyTorch con bibliotecas de terceros, esas bibliotecas esperan pasar un tipo de datos JSON o definido por el usuario, y en algún momento esos resultados se serializarán. Con RPC, puedes enviar Tensores como argumentos grandes y la comunicación no será un cuello de botella.

TensorPipe != MPI -> TensorPipe intentará seleccionar el mejor canal para ti (TCP si no son servidores diferentes, o NVLINK en caso contrario, por ejemplo) -> Cuando llamas a rpc_init, también puedes especificar las conexiones disponibles (ver https://pytorch.org/docs/stable/rpc.html#backends)

Por lo tanto, **TensorPipe permite la comunicación directa de GPU a GPU, pero no de MPI.**

x puede ser un nn.Linear, por lo que los pesos y el bias se pueden comunicar como un paquete de dos Tensores.

Datos como un objeto Python.



citic

API – Remote Execution

```
>>> import os
>>> from torch.distributed import rpc
>>> os.environ['MASTER_ADDR'] = 'localhost'
>>> os.environ['MASTER_PORT'] = '29500'
>>>
>>> rpc.init_rpc(
>>>     "worker1",
>>>     rank=0,
>>>     world_size=2,
>>>     rpc_backend_options=rpc.TensorPipeRpcBackendOptions(
>>>         num_worker_threads=8,
>>>         rpc_timeout=20 # 20 second timeout
>>>     )
>>> )
>>>
>>> # omitting init_rpc invocation on worker2
```

Más ejemplos

- https://pytorch.org/docs/stable/rpc.html#rpc

API – Remote Reference

API – Remote Reference

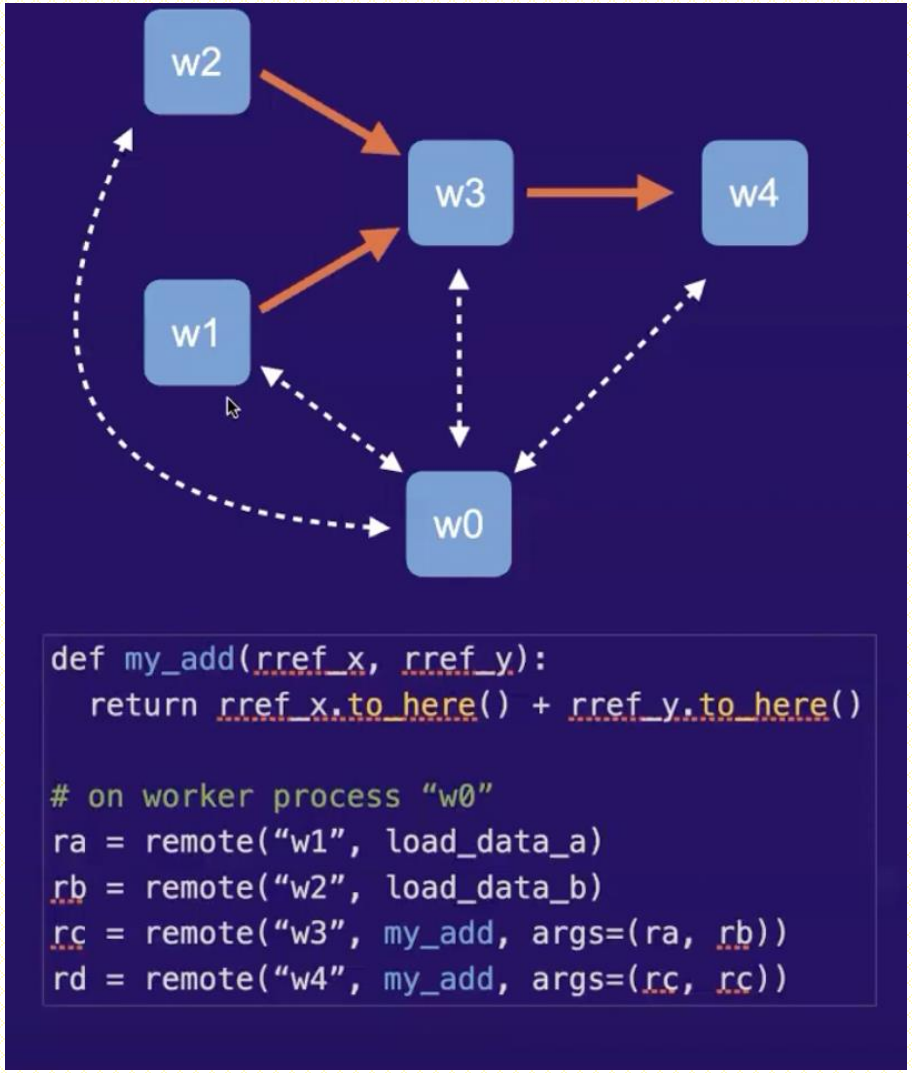
WARNING

RRefs are not currently supported when using CUDA tensors

API – Remote Reference

- Un RRef (Remote REFerence) es una referencia a un valor de algún tipo T (por ejemplo, un Tensor) en un worker remoto. Este identificador mantiene vivo el valor remoto referenciado en el propietario, pero no implica que el valor se transferirá al worker local en el futuro.
- Los RRefs se pueden utilizar en entrenamientos multi-máquina al mantener referencias a nn.Modules que existen en otros workers y llamar a las funciones apropiadas para recuperar o modificar sus parámetros durante el entrenamiento.

API – Remote Reference



¿Cuándo puede ser útil el Remote Reference de RPC?

Durante el entrenamiento es común tener un nodo maestro - w0 en este ejemplo

w0 se encarga de distribuir los datos y recibir los resultados.

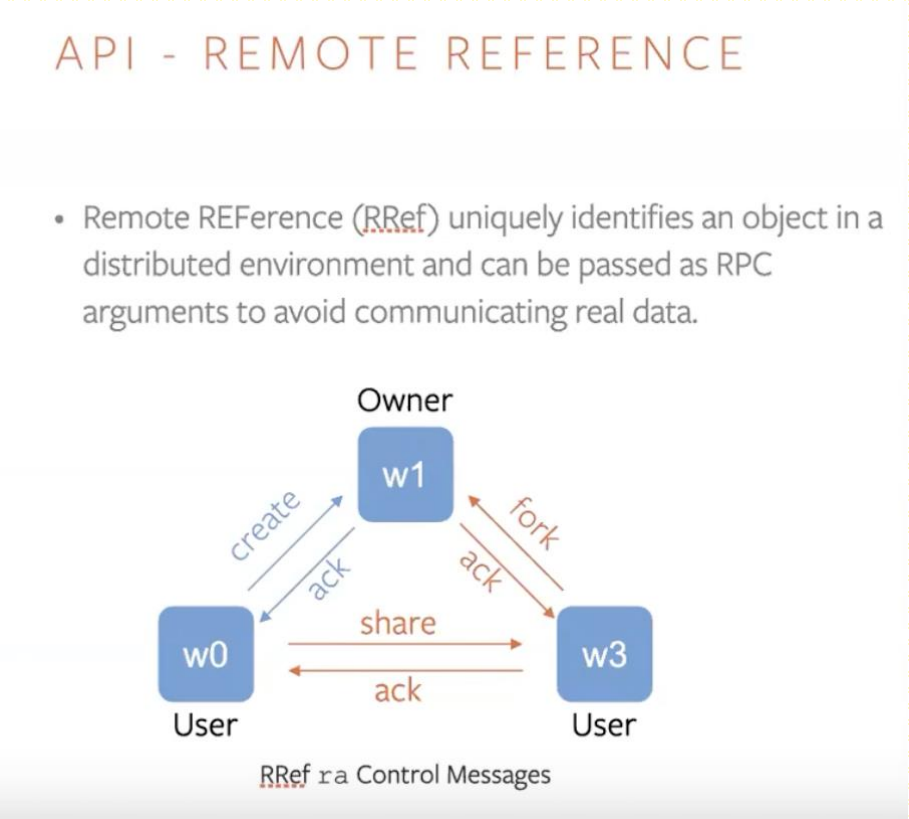
En este ejemplo, w0 envía los datos a w1 y w2

Luego, obtiene las referencias a esos dos procesos y se los envía a w3(rca, rb) y a w4 (rc, rc). Nótese que en el segundo se repite la referencia porque solo hay un proceso origen.

w0 define las dependencias, pero las comunicaciones no pasan a traves de w0, van directas entre w1-w4

Puedes definir internamente las operaciones de cada worker, y luego establecer la forma en como quieres que esos workers se comuniquen entre sí

API – Remote Reference



¿Cuándo puede ser útil el Remote Reference de RPC?

Durante el entrenamiento es común tener un nodo maestro - w0 en este ejemplo

w0 se encarga de distribuir los datos y recibir los resultados.

En este ejemplo, w0 envía los datos a w1 y w2

Luego, obtiene las referencias a esos dos procesos y se los envía a w3(rca, rb) y a w4 (rc, rc). Nótese que en el segundo se repite la referencia porque solo hay un proceso origen.

w0 define las dependencias, pero las comunicaciones no pasan a traves de w0, van directas entre w1-w4

Puedes definir internamente las operaciones de cada worker, y luego establecer la forma en como quieres que esos workers se comuniquen entre sí

API – Remote Reference

- La Referencia Remota (RRef) sirve como un puntero compartido distribuido a un objeto local o remoto. Puede ser compartido con otros workers y el recuento de referencias se manejará de manera transparente.
- Cada RRef solo tiene un propietario y el objeto solo existe en ese propietario.
- Los workers que no son propietarios y tienen RRefs pueden obtener copias del objeto del propietario solicitándolo explícitamente. Esto es útil cuando un worker necesita acceder a un objeto de datos, pero no es el creador (quien realiza la llamada a remote()) ni el propietario del objeto. El optimizador distribuido, como se discutirá a continuación, es un ejemplo de casos de uso de este tipo

API – Remote Reference

El protocolo RRef está diseñado con las siguientes suposiciones:

- **Fallas transitorias de red**: El diseño de RRef maneja las fallas transitorias de red mediante la retransmisión de mensajes. No puede manejar caídas de nodos o particiones permanentes de red. Cuando ocurren esos incidentes, la aplicación debe detener todos los trabajadores, volver al punto de control anterior y reanudar el entrenamiento.
- **UDF no idempotentes**: Suponemos que las funciones de usuario (UDF) proporcionadas a rpc_sync(), rpc_async() o remote() no son idempotentes y, por lo tanto, no se pueden reintentar. Sin embargo, los mensajes de control internos de RRef son idempotentes y se reintentan en caso de fallo del mensaje.
- **Entrega desordenada de mensajes:** No asumimos un orden de entrega de mensajes entre cualquier par de nodos, debido a que tanto el remitente como el receptor están utilizando múltiples hilos. No hay garantía de cuál mensaje será procesado primero.

API – Remote Reference

Más info: https://pytorch.org/docs/stable/rpc/rref.html

API – Distributed Autograd
Distributed Optimizer

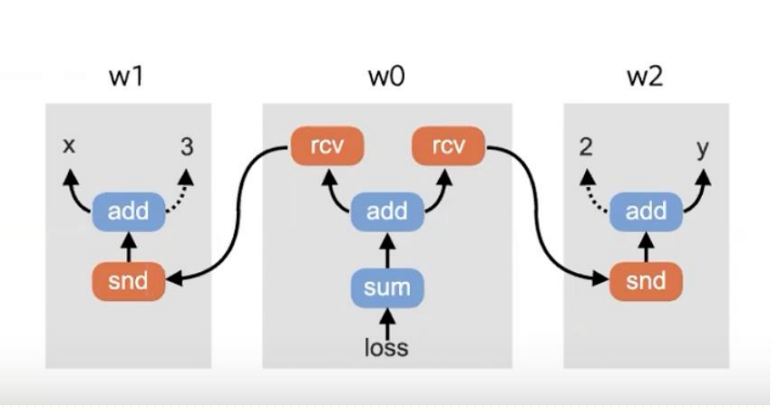
API – Distributed Autograd Distributed Optimizer

- El autograd distribuido une los motores de autograd locales en todos los workers involucrados en el pase hacia adelante y los alcanza automáticamente durante el pase hacia atrás para calcular los gradientes.
- Esto es especialmente útil si el pase hacia adelante necesita abarcar varias máquinas al realizar, por ejemplo, entrenamiento distribuido con paralelismo de modelos, entrenamiento con servidor de parámetros, etc.
- Con esta función, el código del usuario ya no necesita preocuparse por cómo enviar gradientes a través de los límites de RPC y en qué orden se deben lanzar los motores de autograd locales, lo cual puede volverse bastante complicado cuando hay llamadas RPC anidadas e interdependientes en el pase hacia adelante.

API – Distributed Autograd Distributed Optimizer

- El constructor de Distributed Optimizer: (1) recibe un optimizer() (por ejemplo, SGD(), Adagrad(), etc.) y una lista de RRefs de parámetros, (2) crea una instancia de Optimizer() en cada propietario de RRef distinto y (3) actualiza los parámetros correspondientemente al ejecutar step().
- Cuando tienes pases hacia adelante y hacia atrás distribuidos, los parámetros y gradientes se dispersarán en varios workers, por lo que se requiere un optimizador en cada uno de los workers involucrados.
- Distributed Optimizer envuelve todos esos optimizadores locales en uno solo y proporciona un constructor y una API de step() concisos.

API – Distributed Autograd Distributed Optimizer




```python
from torch.distributed import autograd, optim, rpc

with autograd.context() as ctx:
  # prepare parameters
  rx = rpc.remote("w1", create_requires_grad)
  ry = rpc.remote("w2", create_requires_grad)

  # prepare optimizer
  opt = optim.DistributedOptimizer(
    torch.optim.SGD,
    [rx, ry],
    lr=0.05
  )

  ra = rpc.remote("w1", my_add, args=(rx, 3))
  rb = rpc.remote("w2", my_add, args=(ry, 2))

  # forward - backward - optimizer step
  loss = (ra.to_here() + rb.to_here()).sum()
  autograd.backward(ctx, [loss])
  opt.step(ctx)
```

Cómo extendel el motor del autograd local?

rpc.remote() se encarga de preparar los parámetros.

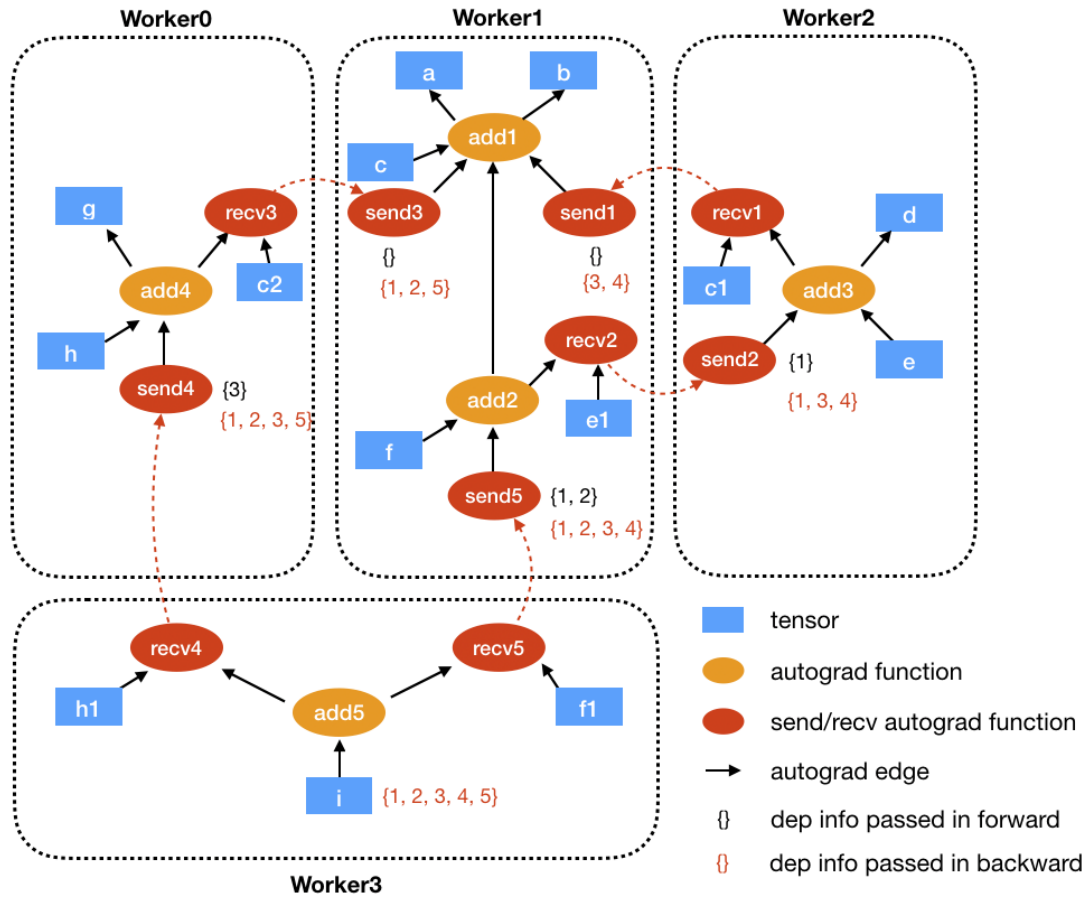
Luego creamos un optimizador distribuido. El optimizador solo busca parámetros locales y, si usamos una versión distribuida, obligamos a los usuarios a llamar a esos optimizadores locales en cada proceso. Nótese que esta función toma las referencias remotas de dos workers y crea un optimizador local en esos propietarios.

Cuando luego llamamos a la función step(), también se llegará a cada optimizador local individualmente.

Se definen funciones específicas para cada worker. Entonces ra.to_here()... va a conectar w1 y w0, y w2 y w0 (rcv).

La backward se aplica a w1 y w2 desde w0.
Los pases fwd y bwd para cada proceso son particulares e independientes.

API – Distributed Autograd Distributed Optimizer



https://pytorch.org/docs/stable/rpc/distributed_autograd.html

Ejemplos de uso

EXAMPLE USE CASES

PARAMETER SERVER

Communicate parameters and data across trainers and servers.

https://pytorch.org/tutorials/intermediate/rpc_param_server_tutorial.html

MODEL PARALLEL

Scale large models cross multiple machines.

https://pytorch.org/tutorials/intermediate/rpc_tutorial.html

PIPELINE PARALLEL

Overlap computation across multiple micro-batches.

https://pytorch.org/tutorials/intermediate/dist_pipeline_parallel_tutorial.html

citic

Ejemplos. Paralelismo de modelo

- Aprendizaje por refuerzo: generar múltiples observadores que se ejecutan en paralelo y comparten un único agente. RPC y RRef permiten enviar datos de un trabajador a otro haciendo referencia fácilmente a objetos de datos remotos.
- El modelo es demasiado grande para entrar en las GPUs de una sola máquina. El autograd distribuido y el optimizador distribuido permiten ejecutar el paso hacia atrás (backward pass) y el paso del optimizador como si fuera un entrenamiento local.

Ejemplo: Distributed RNN using Distributed Autograd and Distributed Optimizer

```
class EmbeddingTable(nn.Module):
    r"""
    Encoding layers of the RNNModel
    """
    def __init__(self, ntoken, ninp, dropout):
        super(EmbeddingTable, self).__init__()
        self.drop = nn.Dropout(dropout)
        self.encoder = nn.Embedding(ntoken, ninp).cuda()
        self.encoder.weight.data.uniform_(-0.1, 0.1)

    def forward(self, input):
        return self.drop(self.encoder(input.cuda()).cpu()


class Decoder(nn.Module):
    def __init__(self, ntoken, nhid, dropout):
        super(Decoder, self).__init__()
        self.drop = nn.Dropout(dropout)
        self.decoder = nn.Linear(nhid, ntoken)
        self.decoder.bias.data.zero_()
        self.decoder.weight.data.uniform_(-0.1, 0.1)

    def forward(self, output):
        return self.decoder(self.drop(output))
```

- Los tres componentes principales de una RNNs son: (1) una embedding table, (2) una capa LSTM, y (3) el decoder
- En este ejemplo vamos a dividir dos componentes típicos de una RNN en dos trabajadores distintos.
- Para ello empaquetamos la embeddingTable y el decoder en submodulos distintos con su respectivo constructor que puede ser enviado al API RPC.
- El encoder de la EmbeddingTable ha sido movido a la GPU intencionadamente. Lo veremos a continuación.

Ejemplo: Distributed RNN using Distributed Autograd and Distributed Optimizer

- En el código siguiente, 'ps' representa un servidor de parámetros que aloja los parámetros de la embedding table y el decodificador.
- El constructor utiliza la API remota para crear un objeto 'EmbeddingTable' y un objeto 'Decoder' en el servidor de parámetros, y crea localmente el submódulo LSTM.

```
class RNNModel(nn.Module):
    def __init__(self, ps, ntoken, ninp, nhid, nlayers, dropout=0.5):
        super(RNNModel, self).__init__()

        # setup embedding table remotely
        self.emb_table_rref = rpc.remote(ps, EmbeddingTable, args=(ntoken, ninp, dropout))
        # setup LSTM locally
        self.rnn = nn.LSTM(ninp, nhid, nlayers, dropout=dropout)
        # setup decoder remotely
        self.decoder_rref = rpc.remote(ps, Decoder, args=(ntoken, nhid, dropout))

    def forward(self, input, hidden):
        # pass input to the remote embedding table and fetch emb tensor back
        emb = _remote_method(EmbeddingTable.forward, self.emb_table_rref, input)
        output, hidden = self.rnn(emb, hidden)
        # pass output to the rremote decoder and get the decoded output back
        decoded = _remote_method(Decoder.forward, self.decoder_rref, output)
        return decoded, hidden
```

Ejemplo: Distributed RNN using Distributed Autograd and Distributed Optimizer

- Durante el pase hacia adelante (forward pass), el entrenador utiliza 'EmbeddingTable RRef' para encontrar el submódulo remoto y pasa los datos de entrada a 'EmbeddingTable' utilizando RPC y obtiene los resultados de búsqueda.
- Luego, ejecuta el embedding a través de la capa LSTM local y finalmente utiliza otra RPC para enviar la salida al submódulo 'Decoder'. Esto se asemeja mucho al entrenamiento paralelo de modelos en una sola máquina. La diferencia principal es reemplazar 'Tensor.to(device)' con funciones RPC.

```
class RNNModel(nn.Module):
    def __init__(self, ps, ntoken, ninp, nhid, nlayers, dropout=0.5):
        super(RNNModel, self).__init__()

        # setup embedding table remotely
        self.emb_table_rref = rpc.remote(ps, EmbeddingTable, args=(ntoken, ninp, dropout))
        # setup LSTM locally
        self.rnn = nn.LSTM(ninp, nhid, nlayers, dropout=dropout)
        # setup decoder remotely
        self.decoder_rref = rpc.remote(ps, Decoder, args=(ntoken, nhid, dropout))

    def forward(self, input, hidden):
        # pass input to the remote embedding table and fetch emb tensor back
        emb = _remote_method(EmbeddingTable.forward, self.emb_table_rref, input)
        output, hidden = self.rnn(emb, hidden)
        # pass output to the rremote decoder and get the decoded output back
        decoded = _remote_method(Decoder.forward, self.decoder_rref, output)
        return decoded, hidden
```

Ejemplo: Distributed RNN using Distributed Autograd and Distributed Optimizer

- Agreguemos una función auxiliar para generar una lista de RRefs de los parámetros del modelo, que serán utilizados por el optimizador distribuido. En el entrenamiento local, las aplicaciones podían llamar a m*odule.parameters()* para obtener referencias a todos los tensores de parámetros y pasarlo al optimizador local para actualizaciones posteriores.
- Sin embargo, la misma API no funciona en escenarios de entrenamiento distribuido, ya que algunos parámetros residen en máquinas remotas. Por lo tanto, en lugar de tomar una lista de tensores de parámetros, el optimizador distribuido toma una lista de RRefs, un RRef por cada parámetro del modelo, tanto para los parámetros locales como para los parámetros remotos.

```
def _parameter_rrefs(module):
    param_rrefs = []
    for param in module.parameters():
        param_rrefs.append(RRef(param))
    return param_rrefs
```

Ejemplo: Distributed RNN using Distributed Autograd and Distributed Optimizer

- Luego, como el modelo RNNModel contiene tres submódulos, necesitamos llamar a _parameter_rrefs tres veces y envolverlo en otra función auxiliar.

```
class RNNModel(nn.Module):
    ...
    def parameter_rrefs(self):
        remote_params = []
        # get RRefs of embedding table
        remote_params.extend(_remote_method(_parameter_rrefs, self.emb_table_rref))
        # create RRefs for local parameters
        remote_params.extend(_parameter_rrefs(self.rnn))
        # get RRefs of decoder
        remote_params.extend(_remote_method(_parameter_rrefs, self.decoder_rref))
        return remote_params
```

Ejemplo: Distributed RNN using Distributed Autograd and Distributed Optimizer

```
def run_trainer():
    batch = 5
    ntoken = 10
    ninp = 2

    nhid = 3
    nindices = 3
    nlayers = 4
    hidden = (
        torch.randn(nlayers, nindices, nhid),
        torch.randn(nlayers, nindices, nhid)
    )

    model = rnn.RNNModel('ps', ntoken, ninp, nhid, nlayers)

    # setup distributed optimizer
    opt = DistributedOptimizer(
        optim.SGD,
        model.parameter_rrefs(),
        lr=0.05,
    )

    criterion = torch.nn.CrossEntropyLoss()

    def get_next_batch():
        for _ in range(5):
            data = torch.LongTensor(batch, nindices) % ntoken
            target = torch.LongTensor(batch, ntoken) % nindices
            yield data, target
```

- Ahora estamos listos para implementar el bucle de entrenamiento. Después de inicializar los argumentos del modelo, creamos el RNNModel y el DistributedOptimizer.
- El optimizador distribuido tomará una lista de RRefs de parámetros, encontrará todos los trabajadores propietarios distintos y creará el optimizador local especificado (por ejemplo, SGD en este caso, también se pueden usar otros optimizadores locales) en cada uno de los trabajadores propietarios utilizando los argumentos dados (por ejemplo, lr=0.05).

Ejemplo: Distributed RNN using Distributed Autograd and Distributed Optimizer

```
# train for 10 iterations
for epoch in range(10):
    for data, target in get_next_batch():
        # create distributed autograd context
        with dist_autograd.context() as context_id:
            hidden[0].detach_()
            hidden[1].detach_()
            output, hidden = model(data, hidden)
            loss = criterion(output, target)
            # run distributed backward pass
            dist_autograd.backward(context_id, [loss])
            # run distributed optimizer
            opt.step(context_id)
            # not necessary to zero grads since they are
            # accumulated into the distributed autograd context
            # which is reset every iteration.
    print("Training epoch {}".format(epoch))
```

- En el bucle de entrenamiento, primero se crea un contexto de autodiferenciación distribuida, que ayudará al motor de autodiferenciación distribuida a encontrar gradientes y funciones RPC de envío/recepción involucradas.
- Luego, se inicia el pase hacia adelante como si fuera un modelo local y se ejecuta el pase hacia atrás distribuido.
- Para el pase hacia atrás distribuido, solo necesitas especificar una lista de raíces, en este caso, es el tensor de pérdida. El motor de autodiferenciación distribuida recorrerá automáticamente el gráfico distribuido y escribirá correctamente los gradientes.
- A continuación, se ejecuta la función step en el optimizador distribuido, que se comunicará con todos los optimizadores locales involucrados para actualizar los parámetros del modelo.

Ejemplo: Distributed RNN using Distributed Autograd and Distributed Optimizer

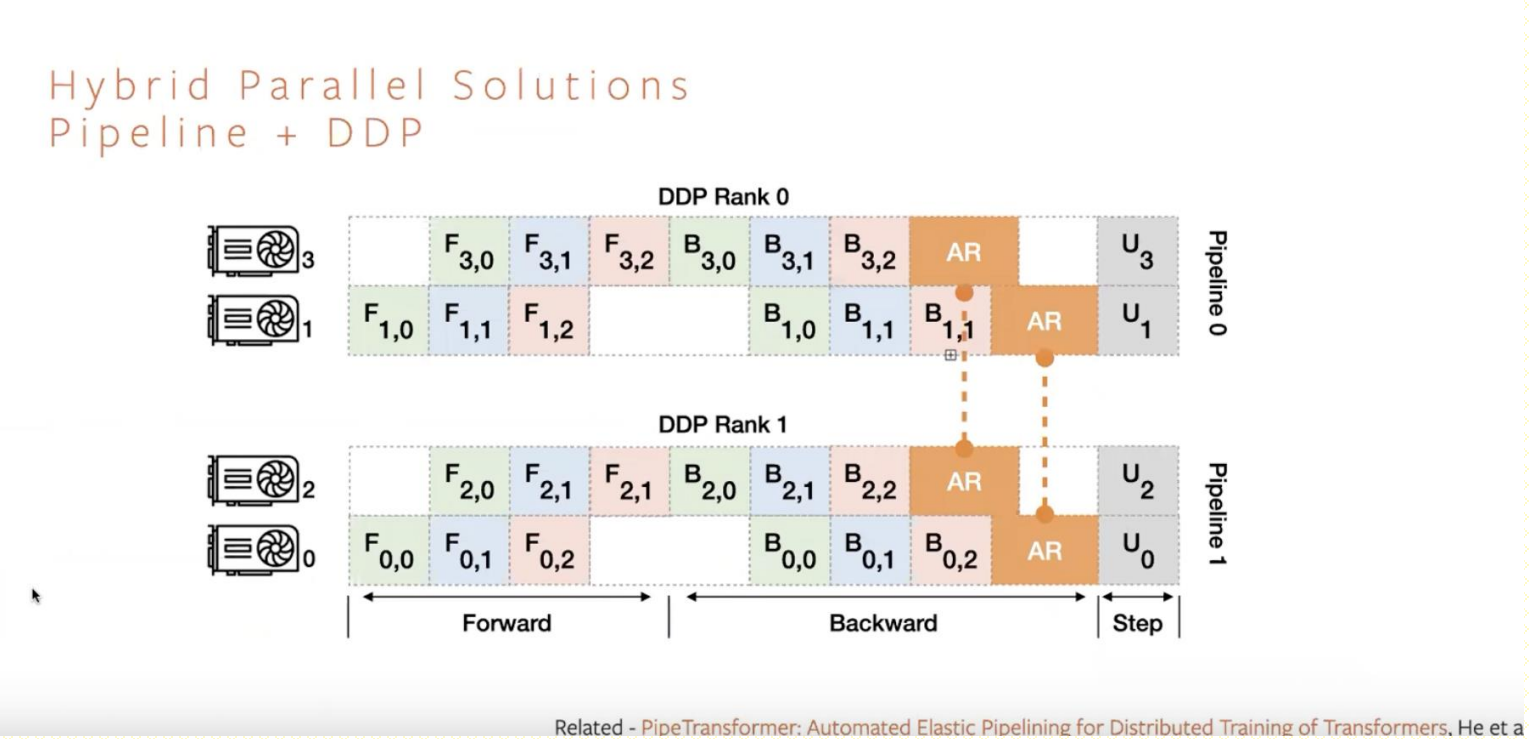
- En comparación con el entrenamiento local, una pequeña diferencia es que no es necesario ejecutar zero_grad() porque cada contexto de autodiferenciación tiene un espacio dedicado para almacenar gradientes, y como creamos un contexto por iteración, esos gradientes de diferentes iteraciones no se acumularán en el mismo conjunto de tensores.
- Por último, agreguemos algo de código adicional para iniciar los procesos del servidor de parámetros y el entrenador.

```
def run_worker(rank, world_size):
    os.environ['MASTER_ADDR'] = 'localhost'
    os.environ['MASTER_PORT'] = '29500'
    if rank == 1:
        rpc.init_rpc("trainer", rank=rank, world_size=world_size)
        _run_trainer()
    else:
        rpc.init_rpc("ps", rank=rank, world_size=world_size)
        # parameter server do nothing
        pass

    # block until all rpcs finish
    rpc.shutdown()


if __name__=="__main__":
    world_size = 2
    mp.spawn(run_worker, args=(world_size, ), nprocs=world_size, join=True)
```

Ejemplo: Distributed RNN using Distributed Autograd and Distributed Optimizer

- https://github.com/diegoandradecanosa/Cesga2023Courses/tree/main/pytorch_dist/rpc/rnn

Ejemplo de uso 2

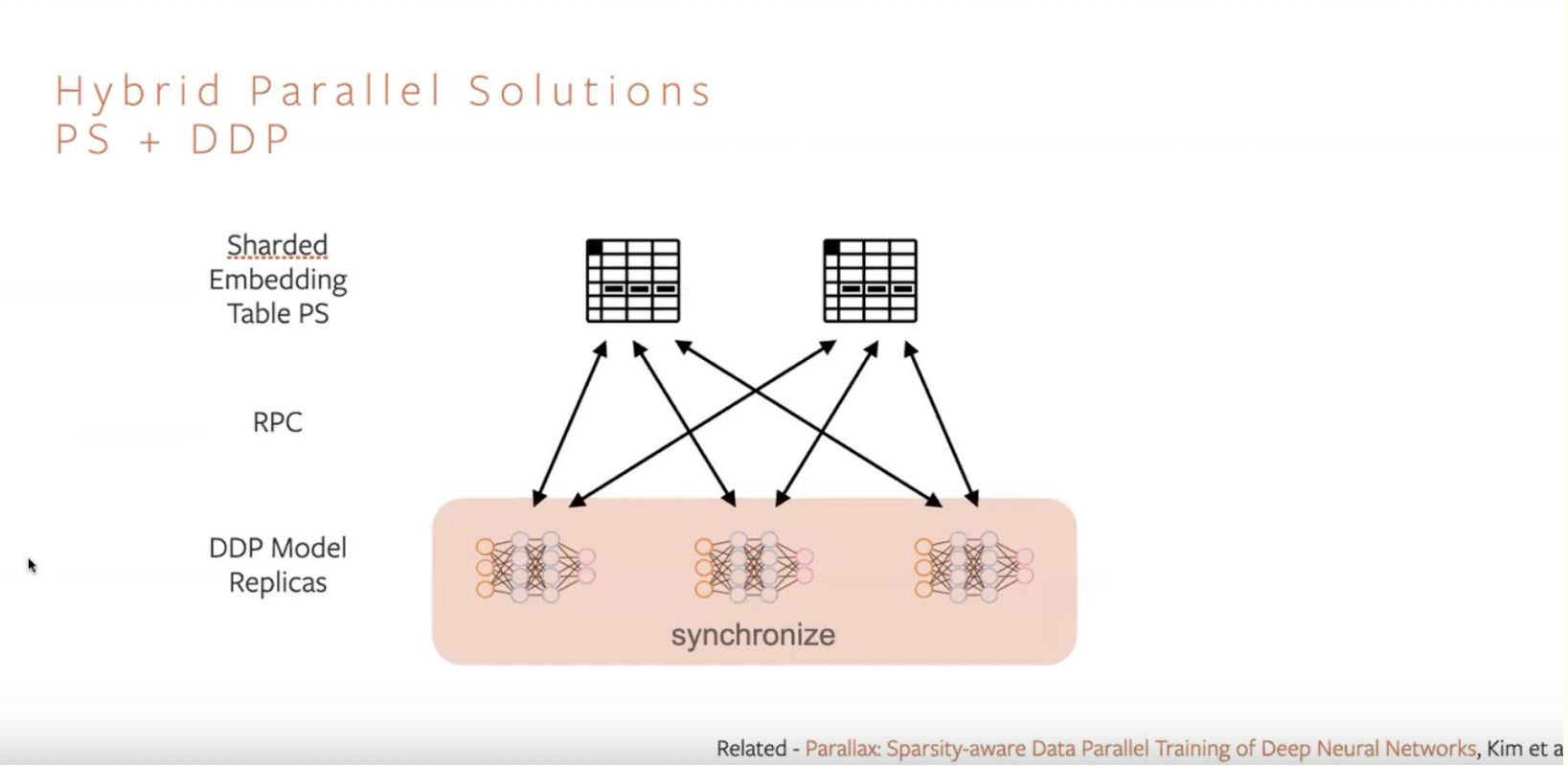


Related - PipeTransformer: Automated Elastic Pipelining for Distributed Training of Transformers. He et a

Hoy en día, PyTorch ya admite esta combinación y puede acelerar el entrenamiento sustancialmente al congelar los modelos incrementalmente y generar réplicas adicionales de modelos.

citic

Ejemplo de uso 3



Otra estructura típica se da cuando tenemos embedding tables grandes o embedding tables dispersas. Es habitual extraerlas del modelo porque de lo contrario no cabrán en la memoria de la GPU o de la máquina.

En este tipo de procesos las tablas pueden cambiar dinamicamente de tal forma que no sabemos la cantidad de no ceros en cada iteracion. Entonces en Pytorch la representacion de matrices dispersas es a traves de dos matrices densas y será necesario hacer un alltogether (allreduce/allgatherv) para conocer cuales son dichas tablas en cada iteracion, lo que representa un cuello de botella en la comunicacion.

Sin embargo si tenemos solo matrices densas, podemos ejecutar solo una instruccion collectiva de comunicacion porque el tamaño de los vectores se conoce.

DDP puede soportar este tipo de problemas de forma eficiente.

Gloo va a funcionar pero va a ser muy lento. Lo recomendado es extraer los tensores dispersos, ponerlo todo en un parameter server, y utilizar RPC para realizar la comunicacion sparse tensors - ddp para sincronizar esos tensores.

Puede haber muchas otras aplicaciones donde esto puede ser interesante, como los recomendadores.

Más ejemplos

- https://github.com/pytorch/examples/tree/main/distributed/rpc