

Managing GPU memory when using Tensorflow and Pytorch

Modern machine learning frameworks take advantage of GPUs to accelerate training/evaluation. Typically, the major platforms use nvidia CUDA to map deep learning graphs to operations which are then run on the GPU. CUDA requires the program to explicitly manage memory on the GPU and there are multiple strategies to do this. Unfortunately Tensorflow does not release memory until the end of the program, and while PyTorch can release memory, it is difficult to ensure that it can and does. This can be side-stepped by using Process Isolation which is applicable for both frameworks. We include a separate section for Process Isolation.

Tensorflow

By default, Tensorflow tries to allocate as much memory as it can on the GPU. The theory is if the memory is allocated in one large block, subsequent creation of variables will be closer in memory and improve performance. This behavior can be tuned in tensorflow using the `tf.config` API. We'll point out a couple of functions here:

`tf.config.list_physical_devices('GPU')` : Lists the GPUs currently usable by the python process
`tf.config.set_visible_devices(devices, device_type=None)`: Sets the specific devices Tensorflow will use.
`tf.config.experimental.set_memory_growth(gpu, True)` : Sets the gpu object to use memory growth mode. In this mode, tensorflow will only allocate the memory it needs, and grow it over time.
`tf.device('/device:GPU:2')` : in a python with context block will restrict all tensors to being allocated only on the specified device.
`tf.config.experimental.get_memory_usage(device)` : Get the memory usage of the device.
`tf.config.LogicalDeviceConfiguration` : An object which allows the user to set special requirements on a particular device. This can be used to restrict the amount of memory Tensorflow will use.
`tf.config.set_logical_device_configuration(device, logical_devices)` : A method to apply a `LogicalDeviceConfiguration` to a device.

We'll show two strategies for controlling GPU utilization with Tensorflow.

Restricting which GPU Tensorflow can use

If tensorflow can use multiple GPUs, we can restrict which one it uses in the following way:

```
# Get a list of GPU devices
gpus = tf.config.list_physical_devices('GPU')
# Restrict Tensorflow to only use the first.
tf.config.set_visible_devices(gpus[:1], device_type='GPU')
```

Restricting how much memory Tensorflow can allocate on a GPU.

We can create a logical device with the maximum amount of memory we wish Tensorflow to allocate

```
# First, Get a list of GPU devices
gpus = tf.config.list_physical_devices('GPU')
# Restrict to only the first GPU.
tf.config.set_visible_devices(gpus[:1], device_type='GPU')
# Create a LogicalDevice with the appropriate memory limit
log_dev_conf = tf.config.LogicalDeviceConfiguration(
    memory_limit=2*1024 # 2 GB
)
# Apply the logical device configuration to the first GPU
tf.config.set_logical_device_configuration(
    gpus[0],
    [log_dev_conf])
```

PyTorch

Currently, PyTorch has no mechanism to limit direct memory consumption, however pytorch does have some mechanisms for monitoring memory consumption and clearing gpu memory cache. If you are careful in deleting all python variables referencing cuda memory, PyTorch will garbage collect the memory eventually. We review these methods here.

`torch.cuda.memory_allocated(device=None)` : Specify the amount of cuda memory currently allocated on a given device.
`torch.cuda.empty_cache()` : Releases all unoccupied cached memory currently held by the caching allocator.

See more at the following discussion:

<https://discuss.pytorch.org/t/how-can-i-release-the-unused-gpu-memory/81919/9>

Process Isolation

To ensure you can clean up any GPU memory when you're finished, you can also try process isolation. This requires you to define a pickle-able python method which you can then send to a separate python process with multiprocessing. Upon completion, the other process will terminate and clean up its memory ensuring you don't leave any un-needed variables behind. This is the strategy employed by dryml, which provides a function decorator to manage the process creation and retrieval of results. We'll present a simple example here showing how you might do it.

```
# The following will not work in an interactive python shell. It must be run as a standalone script.

# Define the function in which we want to allocate memory.
def memory_consumer():
    import tensorflow as tf
    test_array = tf.random.uniform((5,5))
    av = tf.reduce_mean(test_array)
    print(av)

if __name__ == "__main__":
    import multiprocessing as mp
    # Set the multiprocessing start method as 'spawn' since we're interested in only consuming memory for the
    # problem at hand. 'fork' copies all current variables (which may be a lot)
    mp.set_start_method('spawn')
    # Call that function using a separate process
    p = mp.Process(target=memory_consumer)
    # Start the process
    p.start()
    # Wait for the process to complete by joining.
    p.join()
```

Additional methods exist such as multiprocessing pools which handle variable creation and return value management for you.

See more about how dryml does process isolation here:

<https://github.com/ncsa/DRYML>