Frequently Asked Questions

My model reports "cuda runtime error(2): out of memory"

As the error message suggests, you have run out of memory on your GPU. Since we often deal with large amounts of data in PyTorch, small mistakes can rapidly cause your program to use up all of your GPU; fortunately, the fixes in these cases are often simple. Here are a few common things to check:

Don't accumulate history across your training loop. By default, computations involving variables that require gradients will keep history. This means that you should avoid using such variables in computations which will live beyond your training loops, e.g., when tracking statistics. Instead, you should detach the variable or access its underlying data.

Sometimes, it can be non-obvious when differentiable variables can occur. Consider the following training loop (abridged from source):

```
total_loss = 0
for i in range(10000):
    optimizer.zero_grad()
    output = model(input)
    loss = criterion(output)
    loss.backward()
    optimizer.step()
    total_loss += loss
```

Here, `total_loss` is accumulating history across your training loop, since `loss` is a differentiable variable with autograd history. You can fix this by writing *total_loss += float(loss)* instead.

Other instances of this problem: 1.

Don't hold onto tensors and variables you don't need. If you assign a Tensor or Variable to a local, Python will not deallocate until the local goes out of scope. You can free this reference by using `del x`. Similarly, if you assign a Tensor or Variable to a member variable of an object, it will not deallocate until the object goes out of scope. You will get the best memory usage if you don't hold onto temporaries you don't need.

The scopes of locals can be larger than you expect. For example:

```
for i in range(5):
    intermediate = f(input[i])
    result += g(intermediate)
output = h(result)
return output
```

Here, `intermediate` remains live even while `h` is executing, because its scope extrudes past the end of the loop. To free it earlier, you should `del intermediate` when you are done with it.

Avoid running RNNs on sequences that are too large. The amount of memory required to backpropagate through an RNN scales linearly with the length of the RNN input; thus, you will run out of memory if you try to feed an RNN a sequence that is too long.

The technical term for this phenomenon is backpropagation through time, and there are plenty of references for how to implement truncated BPTT, including in the word language model example; truncation is handled by the `repackage` function as described in this forum post.

Don't use linear layers that are too large. A linear layer `nn.Linear(m, n)` uses $O(nm)$ memory: that is to say, the memory requirements of the weights scales quadratically with the number of features. It is very easy to blow through your memory this way (and remember that you will need at least twice the size of the weights, since you also need to store the gradients.)

Consider checkpointing. You can trade-off memory for compute by using checkpoint.

My GPU memory isn't freed properly

PyTorch uses a caching memory allocator to speed up memory allocations. As a result, the values shown in `nvidia-smi` usually don't reflect the true memory usage. See :ref:`cuda-memory-management` for more details about GPU memory management.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]faq.rst, line 90); *backlink***
>
> Unknown interpreted text role "ref".

If your GPU memory isn't freed even after Python quits, it is very likely that some Python subprocesses are still alive. You may find them via `ps -elf | grep python` and manually kill them with `kill -9 [pid]`.

My out of memory exception handler can't allocate memory

You may have some code that tries to recover from out of memory errors.

```
try:
    run_model(batch_size)
except RuntimeError: # Out of memory
    for _ in range(batch_size):
        run_model(1)
```

But find that when you do run out of memory, your recovery code can't allocate either. That's because the python exception object holds a reference to the stack frame where the error was raised. Which prevents the original tensor objects from being freed. The solution is to move you OOM recovery code outside of the `except` clause.

```
oom = False
try:
    run_model(batch_size)
except RuntimeError: # Out of memory
    oom = True

if oom:
    for _ in range(batch_size):
        run_model(1)
```

My data loader workers return identical random numbers

You are likely using other libraries to generate random numbers in the dataset and worker subprocesses are started via `fork`. See :class:`torch.utils.data.DataLoader`'s documentation for how to properly set up random seeds in workers with its :attr:`worker_init_fn` option.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]faq.rst, line 134); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]faq.rst, line 134); *backlink*

Unknown interpreted text role "attr".

My recurrent network doesn't work with data parallelism

There is a subtlety in using the `pack sequence -> recurrent network -> unpack sequence` pattern in a :class:`~torch.nn.Module` with :class:`~torch.nn.DataParallel` or :func:`~torch.nn.parallel.data_parallel`. Input to each the :meth:`forward` on each device will only be part of the entire input. Because the unpack operation :func:`torch.nn.utils.rnn.pad_packed_sequence` by default only pads up to the longest input it sees, i.e., the longest on that particular device, size mismatches will happen when results are gathered together. Therefore, you can instead take advantage of the :attr:`total_length` argument of :func:`~torch.nn.utils.rnn.pad_packed_sequence` to make sure that the :meth:`forward` calls return sequences of same length. For example, you can write:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]faq.rst, line 143); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]faq.rst, line 143); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]faq.rst, line 143); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]faq.rst, line 143); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]faq.rst, line 143); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]faq.rst, line 143); *backlink*

Unknown interpreted text role "attr".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]faq.rst, line 143); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]faq.rst, line 143); *backlink*

Unknown interpreted text role "meth".

```
from torch.nn.utils.rnn import pack_padded_sequence, pad_packed_sequence

class MyModule(nn.Module):
    # ... __init__, other methods, etc.

    # padded_input is of shape [B x T x *] (batch_first mode) and contains
    # the sequences sorted by lengths
    #   B is the batch size
    #   T is max sequence length
    def forward(self, padded_input, input_lengths):
        total_length = padded_input.size(1)  # get the max sequence length
        packed_input = pack_padded_sequence(padded_input, input_lengths,
                                            batch_first=True)
        packed_output, _ = self.my_lstm(packed_input)
        output, _ = pad_packed_sequence(packed_output, batch_first=True,
                                        total_length=total_length)
        return output


m = MyModule().cuda()
dp_m = nn.DataParallel(m)
```

Additionally, extra care needs to be taken when batch dimension is dim `1` (i.e., `batch_first=False`) with data parallelism. In this case, the first argument of pack_padded_sequence `padding_input` will be of shape `[T x B x *]` and should be scattered along dim `1`, but the second argument `input_lengths` will be of shape `[B]` and should be scattered along dim `0`. Extra code to manipulate the tensor shapes will be needed.