Troubleshooting

Table of Contents

Initial Installation

Correctly configured NVIDIA drivers

Please skip this section if your system doesn't have an NVIDIA GPU.

This section's main purpose is to help diagnose and solve the problem of getting `False` when running:

```
import torch
print(torch.cuda.is_available())
```

despite having `nvidia-smi` working just fine. Which means that `pytorch` can't find the NVIDIA drivers that match the currently active kernel modules.

note: `pytorch` installs itself as `torch`. So we refer to the project and its packages as `pytorch`, but inside python we use it as `torch`.

First, starting with `pytorch-1.0.x` it doesn't matter which CUDA version you have installed on your system, always try first to install the latest `pytorch` - it has all the required libraries built into the package. However, note, that you most likely will **need 396.xx+ driver for `pytorch` built with `cuda92`** . For older drivers you will probably need to install `pytorch` with `cuda90` or ever earlier.

The only thing you to need to ensure is that you have a correctly configured NVIDIA driver, which usually you can test by running: `nvidia-smi` in your console.

If you have `nvidia-smi` working and `pytorch` still can't recognize your NVIDIA CPU, most likely your system has **more than one version of NVIDIA driver installed** (e.g. one via `apt` and another from source). If that's the case, wipe any remnants of NVIDIA drivers on your system, and install one NVIDIA driver of your choice. The following are just the guidelines, you will need to find your platform-specific commands to do it right.

1. Purge `nvidia-drivers` :

   ```
   sudo apt-get purge nvidia-*
   ```

   Note, not remove, but purge! purge in addition to removing the package, also removes package-specific configuration and any other files that were created by it.

2. Once you uninstalled the old drivers, make sure you don't have any orphaned NVIDIA drivers on your system remaining from manual installs. Usually, it's enough to run:

   ```
   find /usr/ | grep libcuda.so
   ```

   or a more sweeping one (if your system updates the `mlocate db` regularly)

   ```
   locate -e libcuda.so
   ```

   If you get any listing as a result of running of these commands, that means you still have some orphaned nvidia driver on your system.

3. Next, make sure you clear out any remaining dkms nvidia modules:

   ```
   dkms status | grep nvidia
   ```

   If after uninstalling all NVIDIA drivers, you still get the command listing dkms nvidia modules, uninstall those too.

   For example if the output is:

```
nvidia-396, 396.44, 4.15.0-34-generic, x86_64: installed
nvidia-396, 396.44, 4.15.0-36-generic, x86_64: installed
```

then run:

```
dkms remove nvidia/396.44 --all
```

Lookup `dkms` manual for the complete details if need be.

4. Install NVIDIA drivers according to your platform's way (**need 396.xx+ driver for `pytorch` built with `cuda92`**)
5. reboot

   Check that your `nvidia-smi` works, and then check that

```
import torch
print(torch.cuda.is_available())
```

   now returns `True` .

Also note that `pytorch` will **silently fallback to CPU** if it reports `torch.cuda.is_available()` as `False` , so the only indicator of something being wrong will be that your notebooks will be running very slowly and you will hear your CPU revving up (if you are using a local system). Run:

```
python -c 'import fastai.utils; fastai.utils.show_install(1)'
```

to detect such issues. If you have this problem it'll say that your torch cuda is not available.

If you're not sure which nvidia driver to install here is a reference table:

CUDA Toolkit	Linux x86_64	Windows x86_64
CUDA 10.0.130	>= 410.48	>= 411.31
CUDA 9.2	>= 396.26	>= 397.44
CUDA 9.0	>= 384.81	>= 385.54
CUDA 8.0	>= 367.48	>= 369.30

You can find a complete table with extra variations here (https://docs.nvidia.com/cuda/cuda-toolkit-release-notes/index.html).

libcuda.so.1: cannot open shared object file

This section is only relevant if you build `pytorch` from source - `pytorch` conda and pip packages link statically to `libcuda` and therefore `libcuda.so.1` is not required to be installed.

If you get an error:

```
ImportError: libcuda.so.1: cannot open shared object file: No such file or directory
```

that means you're missing `libcuda.so.1` from your system.

On most Linux systems the `libcuda` package is optional/recommended and doesn't get installed unless explicitly instructed to do so. You need to install the specific version matching the `nvidia` driver. For the sake of this example, let's assume that you installed `nvidia-396` debian package.

1. find the apt package the file belongs to - `libcuda1-396` :

```
$ dpkg -S  libcuda.so.1
libcuda1-396: /usr/lib/i386-linux-gnu/libcuda.so.1
libcuda1-396: /usr/lib/x86_64-linux-gnu/libcuda.so.1
```

2. check whether that package is installed - no, it is not installed:

```
$ apt list libcuda1-396
Listing... Done
```

3. let's install it:

```
$ apt install libcuda1-396
```

4. check whether that package is installed - yes, it is installed:

```
$ apt list libcuda1-396
Listing... Done
libcuda1-396/unknown,now 396.44-0ubuntu1 amd64 [installed]
```

Now, you shouldn't have this error anymore when you load `pytorch` .

To check which `nvidia` driver the installed package depends on, run:

```
  $ apt-cache rdepends libcuda1-396 | grep nvidia
    nvidia-396
```

To check that `nvidia-396` is installed:

```
$ apt list nvidia-396
  nvidia-396/unknown,now 396.44-0ubuntu1 amd64 [installed,automatic]
```

In your situation, change `396` to whatever version of the driver you're using.

This should be more or less similar on the recent Ubuntu Linux versions, but could vary on other systems. If it's different on your system, find out which package contains `libcuda.so.1` and install that package.

Do not mix conda-forge packages

`fastai` depends on a few packages that have a complex dependency tree, and `fastai` has to manage those very carefully, so in the conda-land we rely on the anaconda main channel and test everything against that.

If you install some packages from the `conda-forge` channel you may have problems with `fastai` installation since you may have wrong versions of packages installed.

So see which packages need to come from the main conda channel do:

```
conda search --info -c fastai "fastai=1.0.7"
```

Replace `1.0.7` with the version you're trying to install. (Without version it'll show you all versions of `fastai` and its dependencies)

Can't install the latest fastai conda package

If you execute:

```
conda install -c fastai fastai
```

and conda installs not the latest `fastai` version, but an older one, that means your conda environment has a conflict of dependencies with another previously installed package, that pinned one of its dependencies to a fixed version and only `fastai` older version's dependencies agree with that fixed version number. Unfortunately, conda is not user-friendly enough to tell you that. You may have to add the option `-v`, `-vv`, or `-vvv` after `conda install` and look through the verbose output to find out which package causes the conflict.

```
conda install -v -c fastai fastai
```

Here is a little example to understand the `conda` package dependency conflict:

Let's assume anaconda.org has 3 packages: `A`, `B` and `P`, and some of them have multiple release versions:

```
package A==1.0.17 depends on package P==1.0.5
package B==1.0.06 depends on package P==1.0.5
package B==1.0.29 depends on package P==1.0.6
package P==1.0.5
package P==1.0.6
```

If you installed `A==1.0.17` via conda, and are now trying to install `conda install B`, conda will install an older `B==1.0.6`, rather than the latest `B==1.0.29`, because the latter needs a higher version of `P`. conda can't install `B==1.0.29` because then it'll break the dependency requirements of the previously installed package `A`, which needs `P==1.0.5`. However, if conda installs `B==1.0.6` there is no conflict, as both `A==1.0.17` and `B==1.0.6` agree on dependency `P==1.0.5`.

It'd have been nice for `conda` to just tell us that: there is a newer package `B` available, but it can't install it because it conflicts on dependency with package `A` which needs package `P` of such and such version. But, alas, this is not the case.

One solution to this problem is to ask conda to install a specific version of the package (e.g. `fastai 1.0.29`):

```
conda install -c fastai fastai==1.0.29
```

It will usually then tell you if it needs to downgrade/upgrade/remove some other packages, that prevent it from installing normally.

In general it is the best to create a new dedicated conda environment for `fastai` and not install anything there, other than `fastai` and its requirements, and keep it that way. If you do that, you will not have any such conflicts in the future. Of course, you can install other packages into that environment, but keep track of what you install so that you could revert them in the future if such conflict arises again.

Conflicts between BLAS libraries

If you use `numpy` and `pytorch` that are linked against different Basic Linear Algebra Subprograms (BLAS) libraries you may experience segfaults if the two libraries conflict with each other. Ideally all the modules that you use (`scipy` too) should be linked against the same BLAS implementation. Currently the main implementations are OpenBLAS, MKL, ATLAS.

To check what library the packages are linked against use:

```
python -c 'import numpy; numpy.__config__.show()'
python -c 'import scipy; scipy.__config__.show()'
python -c 'import sklearn._build_utils; print(sklearn._build_utils.get_blas_info())'
```

XXX: pytorch?

Dedicated environment

`fastai` has a relatively complex set of python dependencies, and it's the best not to install those system-wide, but to use a virtual environment instead (conda (https://conda.io/docs/user-guide/tasks/manage-environments.html) or others). A lot of problems disappear when a fresh dedicated to `fastai` virtual environment is created.

The following example is for using a conda environment.

First you need to install miniconda (https://conda.io/docs/install/quick.html) or anaconda (https://docs.anaconda.com/anaconda/install/). The former comes with bare minimum of packages preinstalled, the latter has hundreds more. If you haven't changed the default configuration, miniconda usually ends up under `~/miniconda3/`, and anaconda under `~/anaconda3/`.

Once you have the software installed, here is a quick way to set up a dedicated environment for just `fastai` with `python-3.6` (of course feel free to name it the way you want it to):

```
conda update conda
conda create -y python=3.6 --name fastai-3.6
conda activate fastai-3.6
conda install -y conda
conda install -y pip setuptools
```

Now you can install `fastai` prerequisites and itself (https://github.com/fastai/fastai/blob/master/README.md#conda-install) using `conda`.

The only thing you need to remember when you start using a virtual environment is that you must activate it before using it. So for example when you open a new console and want to start `jupyter`, instead of doing:

```
jupyter notebook
```

you'd change your script to:

```
conda activate fastai-3.6
jupyter notebook
```

sometimes when you're outside of conda the above doesn't work and you need to do:

```
source ~/anaconda3/bin/activate fastai-3.6
jupyter notebook
```

(of course adjust the path to your conda installation if need to).

Virtual environments provide a lot of conveniences - for example if you want to have a stable env and an experimental one you can clone them in one command:

```
conda create --name fastai-3.6-experimental --clone fastai-3.6
```

or say you want to see how the well-working python-3.6 env will work with python-3.7:

```
conda create --name fastai-3.7 --clone fastai-3.6
conda install -n fastai-3.7 python=3.7
conda update -n fastai-3.7 --all
```

If you use advanced bash prompt functionality, like with git-prompt (https://github.com/magicmonty/bash-git-prompt), it'll now tell you automatically which environment has been activated, no matter where you're on your system. e.g. on my setup it shows:

```
             /fastai:[master|✚1…4⚑3] > conda activate
(base)       /fastai:[master|✚1…4⚑3] > conda activate fastai-3.6
(fastai-3.6) /fastai:[master|✚1…4⚑3] > conda deactivate
             /fastai:[master|✚1…4⚑3] >
```

I tweaked the prompt output for this example by adding whitespace to align the entries to make it easy to see the differences. That leading white space is not there normally. Besides the virtual env, it also shows me which git branch I'm on, and various git status information.

So now you don't need to guess and you know exactly which environment has been activated if any before you execute any code.

Am I using my GPU(s)?

It's possible that your system is misconfigured and while you think you're using your GPU you could be running on your CPU only.

You can check that by checking the output of `import torch; print(torch.cuda.is_available())` - it should return `True` if `pytorch` sees your GPU(s). You can also see the state of your setup with:

```
python -c 'import fastai.utils; fastai.utils.show_install(1)'
```

which will include that check in its report.

But the simplest direct check is to observe the output of `nvidia-smi` while you run your code. If you don't see the process show up when you run the code, then you aren't using your GPU. The easiest way to watch the updated status is through:

```
   watch -n 1 nvidia-smi
```

If you're on a local system, another way to tell you're not using your GPU would be an increased noise from your CPU fan.

Installation Updates

Please skip this section unless you have post- successful install update issues.

Normally you'd update `fastai` by running `pip install -U fastai` or `conda update fastai`, using the same package manager you used to install it in first place (but in reality, either will work).

If you use the developer setup (https://github.com/fastai/fastai/blob/master/README.md#developer-install), then you need to do a simple:

```
cd path/to/your/fastai/clone
git pull
pip install -e ".[dev]"
```

Sometimes jupyter notebooks get messed up, and `git pull` might fail with an error like:

```
error: Your local changes to the following files would be overwritten by merge:
examples/cifar.ipynb
Please, commit your changes or stash them before you can merge.
Aborting
```

then either make a new `fastai` clone (the simplest), or resolve it: disable the nbstripout filter, clean up your checkout and re-enable the filter.

```
tools/trust-origin-git-config -d
git stash
git pull
tools/trust-origin-git-config
```

Of course `git stash pop` if you made some local changes and you want them back.

You can also overwrite the folder if you have no changes you want to keep with `git checkout examples` in this case, or do a reset - but you have to do it **after** you disabled the filters, and then remember to **re-enable** them back. The instructions above do it in a non-destructive way.

The stripout filter allows us to collaborate on the notebooks w/o having conflicts with different execution counts, locally installed extensions, etc., keeping under git only the essentials. Ideally, even the `outputs` should be stripped, but that's a problem if one uses the notebooks for demo, as it is the case with `examples` notebooks.

Managing Multiple Installations

It's possible to have multiple `fastai` installs - usually in different conda environments. And when you do that it's easy to get lost in which environment of `fastai` you currently use.

Other than `conda activate wanted-env` here are some tips to find your way around:

Tell me which environment modules are imported from:

```
import sys
print(sys.path)
```

Tell me which `fastai` library got loaded (we want to know the exact location)

```
import sys, fastai
print(sys.modules['fastai'])
```

At times a quick hack can be used to get your first notebook working and then sorting out the setup. Say you checked out `fastai` to `/tmp/` :

```
cd /tmp/
git clone https://github.com/fastai/fastai
cd fastai
```

So now you know that your *uninstalled* `fastai` is located under `/tmp/fastai/` . Next, put the following on the very top of your notebook:

```
import sys
sys.path.append("/tmp/fastai")
import fastai
```

and it should just work. Now, go and sort out the rest of the installation, so that you don't need to do it for every notebook.

ModuleNotFoundError: No module named 'fastai.vision'

If you have multiple environments, it's very possible that you installed `fastai` into one environment, but then are trying to use it from another, where it's not installed. Even more confusing, the situation where different environments have different versions of `fastai` installed, so its modules are found, but they don't work as you'd expect them to.

If you use jupyter notebook, always make sure you activated the environment you installed `fastai` into before starting the `notebook` .

There is an easy way to check whether you're in the right environment by either running from jupyter cell or in your code:

```
import sys
print(sys.path)
```

and checking whether it shows the correct paths. That is compare these paths with the paths you installed `fastai` into.

Alternatively, you can use the `fastai` helper that will show you that and other important details about your environment:

```
from fastai.utils import *
show_install()
```

or the same from the command line:

```
python -m fastai.utils.show_install
```

Incidentally, we want you to include its output in any bug reports you may submit in the future.

One more situation this may happen is where you accidentally try to run fastai-1.0-based code from under `courses/*/` in the git repo, which includes a symlink to fastai-0.7 code base and then all the hell breaks loose. Just move your notebook away from those folders and all will be good.

If `import fastai` works, but not `import fastai.vision` , that may caused by the old version(maybe 0.7) of fastai you've installed. Try `pip list` in the terminal to see the version if you installed fastai by pip previously. To solve the problem, upgrade the fastai version to 1.0 or higher: `pip install --upgrade fastai` .

Conda environments not showing up in Jupyter Notebook

While normally you shouldn't have this problem, and all the required things should get installed automatically, some users report that their jupyter notebook does not recognize newly created environments at times. To fix that, perform:

```
conda activate fastai
conda install jupyter nb_conda nb_conda_kernels ipykernel
python -m ipykernel install --user --name fastai --display-name "Python (fastai)"
```

Replace `fastai` with the name of your conda environment if it's different.

See also Kernels for different environments (https://ipython.readthedocs.io/en/stable/install/kernel_install.html#kernels-for-different-environments).

CUDA Errors

CUDA out of memory exception

When this error is encountered, that means the software cannot allocate the memory it needs to continue. Therefore you need to change your code to consume less memory. Most of the time in the training loops it requires either reducing the batch size and other hyper-parameters, using a smaller model or smaller items (images, etc.).

There is a particular issue with this error is that under ipython/jupyter notebook this error may lead to an unrecoverable state, where the only way out is to restart the jupyter kernel. This problem is easily solved. Please see Memory Leakage On Exception.

device-side assert triggered

CUDA's default environment allows sending commands to GPU in asynchronous mode - i.e. without waiting to check whether they were successful, thus tremendously speeding up the execution. The side effect is that if anything goes wrong, the context is gone and it's impossible to tell what the error was. That's when you get this generic error, which means that something went wrong on the GPU, but the program can't tell what.

Moreover, the only way to recover from it is to restart the kernel. Other programs and kernels will still be able to use the card, so it only affects the kernel/program the error happened in.

To debug this issue, the non-blocking CUDA mode needs to be turned off, which will slow everything down, but you will get the proper error message, albeit, it will still be unrecoverable. You can accomplish that using several approaches:

- create a cell at the very top of the notebook.

  ```
  import os
  os.environ['CUDA_LAUNCH_BLOCKING'] = "1"
  ```

  Then restart the kernel and run the notebook as usual. Now you should get a meaningful error.

- or alternatively set it globally for all notebooks by restarting `jupyter notebook` as:

  ```
  CUDA_LAUNCH_BLOCKING=1 jupyter notebook
  ```

  except this will affect all notebooks. The error messages will go into the notebook's log.

- or run the program on CPU by either removing `cuda()/to(device)` calls, or by using the following first cell of your notebook and restarting the kernel:

  ```
  import os
  os.environ['CUDA_VISIBLE_DEVICES']=''
  ```

but this can be very very slow, and it's possible that it won't be even possible if the error only happens when run on GPU.

Of course, if you're not using `jupyter notebook` then you can just set the env vars in your bash:

```
CUDA_LAUNCH_BLOCKING=1 my_pytorch_script.py
```

cuda runtime error (11) : invalid argument

If you get an error:

```
RuntimeError: cuda runtime error (11) : invalid argument at .../src/THC/THCGeneral.cpp
```

it's possible that your pytorch build doesn't support the NVIDIA Driver you have installed.

For example, you may have a newer NVIDIA driver with an older pytorch CUDA build, which most of the time should work, as it should be backward compatible, but that is not always the case. So make sure that if you run a recent NVIDIA driver you install pytorch that is built against the latest CUDA version. Follow the instructions here (https://pytorch.org/get-started/locally/).

You will find the table of different CUDA versions and their NVDIA driver counterparts here (https://github.com/fastai/fastai/blob/master/README.md#is-my-system-supported).

Memory Leakage On Exception

This section applies to both general and GPU RAM.

If an exception occurs in a jupyter notebook (or ipython shell) it stores the traceback of the exception so that it can be accessed by `%debug` and `%pdb` magic. The trouble is that this feature prevents variables involved in the exception (`locals()` in each frame involved) from being released and memory reclaimed by `gc.collect()`, when the exception is reported. And so all those variables get stuck and memory is leaked. In particular when CUDA out of memory exception is encountered you might not be able to continue using the card, until the kernel is reset, since the leaked memory will leave no free RAM to proceed with.

So now that you understand this, the quick fix solution is to just run a cell with this content:

```
1/0
```

and you should be back in the game w/o needing to restart the kernel. This fixed the problem since any new exception will free up the resources tied up by the previous exception. If you want something more instructive, use:

```
assert False, "please liberate my GPU!"
```

The leakage happens with any exception, except it's most problematic with CUDA OOM exception. For example if you tend to hit Kernel Interrupt and then re-run your training loop, you will have less RAM to run on when you re-run it.

Currently, ipython is working on a configurable solution. This section will get updated once ipython has it sorted out. You can also follow the discussion here (https://github.com/ipython/ipython/pull/11572).

If you want to understand more about the nuances of the problem of saving a traceback or an exception object, please refer to this explanation (https://stackoverflow.com/a/54295910/9201239).

The rest of this section covers a variety of solutions for this problem.

fastai Solutions

`fastai > 1.0.41` has been instrumented with the following features that will provide you a solution to this problem:

1. under non-ipython environment it doesn't do anything special
2. under ipython it strips tb by default only for the following exceptions:
    - "CUDA out of memory"
    - "device-side assert triggered" that is the `%debug` magic will work under all other exceptions, and it'll leak memory until tb is reset.
3. The env var ` FASTAI_TB_CLEAR_FRAMES` changes this behavior when run under ipython, depending on its value:

- "0": never strip tb (makes it possible to always use %debug magic, but with leaks)
- "1": always strip tb (never need to worry about leaks, but %debug won't work)

where ipython == ipython/ipython-notebook/jupyter-notebook.

At the moment we are only doing this for the fit() family of functions. If you find other fastai API needing this please let us know.

You can set `os.environ['FASTAI_TB_CLEAR_FRAMES']="0"` (or `"1"`) in your code or from the shell when you start jupyter.

Custom Solutions

If you need a solution for your own code that perhaps doesn't involve `fastai` functions, here is a decorator you can use to workaround this issue:

```
import functools, traceback
def gpu_mem_restore(func):
    "Reclaim GPU RAM if CUDA out of memory happened, or execution was interrupted"
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        try:
            return func(*args, **kwargs)
        except:
            type, val, tb = sys.exc_info()
            traceback.clear_frames(tb)
            raise type(val).with_traceback(tb) from None
    return wrapper
```

Now add it before any of your functions:

```
@gpu_mem_restore
def fit(...)
```

and OOM is now automagically recoverable! And `KeyboardInterrupt` leaks no memory!

And if you want to protect just a few lines of code, here is a context manager that does the same:

```
class gpu_mem_restore_ctx():
    " context manager to reclaim GPU RAM if CUDA out of memory happened, or execution was i
nterrupted"
    def __enter__(self): return self
    def __exit__(self, exc_type, exc_val, exc_tb):
        if not exc_val: return True
        traceback.clear_frames(exc_tb)
        raise exc_type(exc_val).with_traceback(exc_tb) from None
```

So now you can do:

```
with gpu_mem_restore_ctx():
    learn.fit_one_cycle(1, 1e-2)
```

with the same results. Except this one (fit functions) is already protected, this would be more useful for your custom code.

Note, that the trick is in running: `traceback.clear_frames(tb)` to free all `locals()` tied to the exception object.

Note that these help functions don't make any special cases and will do the clearing for any exception. Which means that you will not be able to use a debugger if you use those, since an `locals()` will be gone. You can, of course, use the more complicated versions of these functions

from fastai.utils.ipython (https://github.com/fastai/fastai/blob/master/fastai/utils/ipython.py) which have more flexibility as explained in the previous section.

If you need the same solution outside of the fastai environment, you can either copy-n-paste it from this section, or alternatively similar helper functions (a function decorator and a context manager) are available via the ipyexperiments (https://github.com/stas00/ipyexperiments) project, inside the ipyexperiments.utils.ipython (https://github.com/stas00/ipyexperiments/blob/master/docs/utils_ipython.md) module.

If after reading this section, you still have questions, please ask in this thread (https://forums.fast.ai/t/a-guide-to-recovering-from-cuda-out-of-memory-and-other-exceptions/35849).

Support

If troubleshooting wasn't successful please refer next to the support document (/support.html).