

# C++ Debugging in VSCode

PyTorch has good support for C++ debugging. This guide walks you through how to utilize it and integrate it with VSCode.

## Why would you need this?

This guide is for contributors to PyTorch or PyTorch/XLA who need to touch the C++ internals to build features like [custom C++ operations](#).

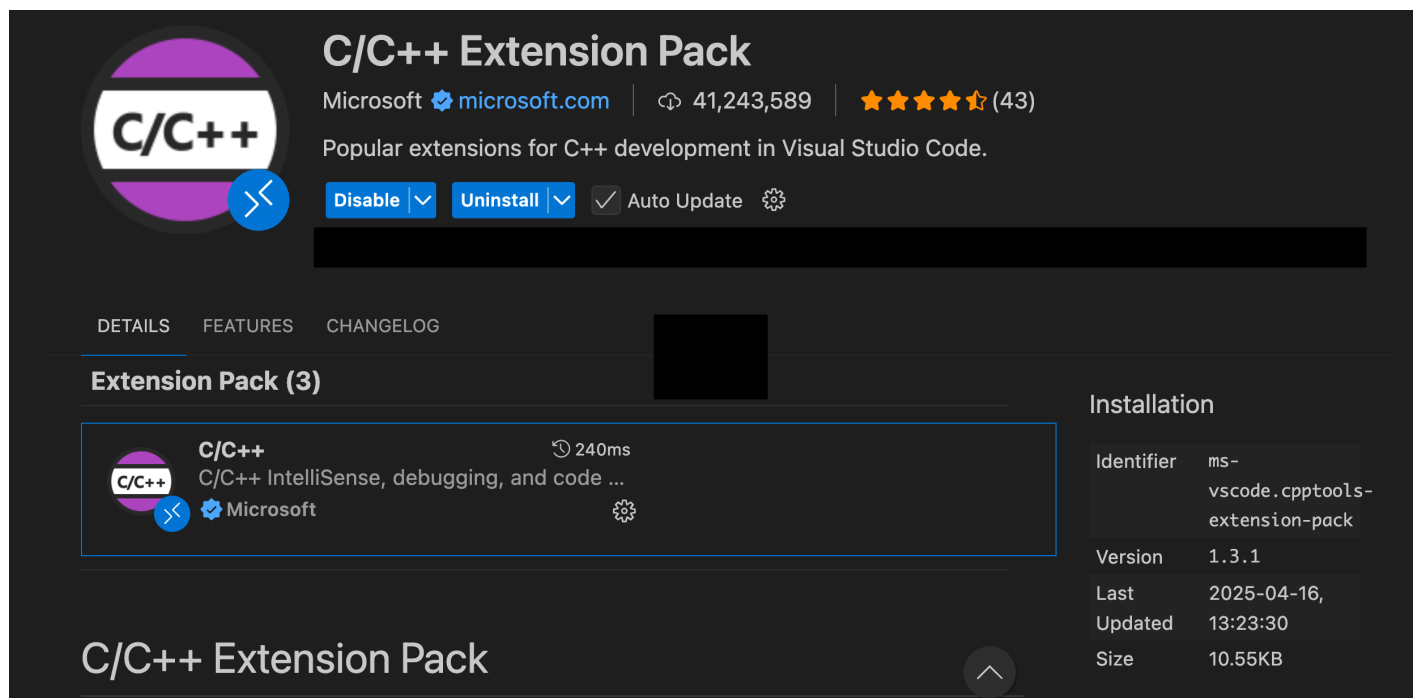
## Install GDB: The GNU Project Debugger

The official instructions are [here](#) but if you are in a conda environment, you can install it as

```
conda install -c conda-forge gdb
```

## VSCode Configuration

Integrating with VSCode will require installing the C/C++ Extension Pack, published by Microsoft. Search in the Extensions tab of VSCode with the identifier `ms-vscode.cpptools-extension-pack`.



The screenshot shows the VS Code Extensions Marketplace page for the 'C/C++ Extension Pack' by Microsoft. The pack is highlighted with a blue box. The pack includes the C/C++ IntelliSense, debugging, and code navigation extensions. The installation details are as follows:

Installation	
Identifier	ms-vscode.cpptools-extension-pack
Version	1.3.1
Last Updated	2025-04-16, 13:23:30
Size	10.55KB

Next, you'll need to create a `launch.json` in the `pytorch/.vscode` folder. Here is a sample file to start from. You'll need to adjust the file paths to match your specific installation.

```

{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "GDB a python interpreter",
      "type": "cppdbg",
      "request": "launch",
      "program": "/home/USERNAME/miniconda3/envs/torch310/bin/python",
      // Replace with your executable's path
      "args": [],
      "cwd": "${workspaceFolder}",
      "environment": [],
      "externalConsole": false,
      "MIMode": "gdb",
      "setupCommands": [
        {
          "description": "Enable pretty-printing for gdb",
          "text": "-enable-pretty-printing",
          "ignoreFailures": true
        },
        {
          "description": "Set Disassembly flavor to Intel",
          "text": "-gdb-set disassembly-flavor intel",
          "ignoreFailures": true
        }
      ],
      // "preLaunchTask": "C/C++: g++ build active file",
      "miDebuggerPath":
      "/home/USERNAME/miniconda3/envs/torch310/bin/gdb" // Replace with your gdb
      location
    }
  ]
}

```

## Building with Debugging Symbols

Looking into `setup.py`, we see

```

# Environment variables you are probably interested in:
#
#   DEBUG
#     build with -O0 and -g (debug symbols)
#
#   REL_WITH_DEB_INFO
#     build with optimizations and -g (debug symbols)
#

```

```
# USE_CUSTOM_DEBINFO="path/to/file1.cpp;path/to/file2.cpp"
#   build with debug info only for specified files
```

## source

Setting `DEBUG` will result in a very slow binary, since every single file will be unoptimized (`-O0`) and built with debugging symbols.

In practice, you'll want to set the `USE_CUSTOM_DEBINFO` instead.

The challenge becomes knowing which files to build with debugging symbols. A common issue will be debugging a file, then realizing you want to look at data structures or another function call in a file that's not built with debugging symbols.

Your basic flow as you get started is:

1. Identify the source file you want to debug.
2. Build that file with debugger symbols
3. Start a debugger session with a breakpoint. You'll discover additional files you want to debug.
4. Add those files to the `USE_CUSTOM_DEBINFO`
5. Rebuild those files
6. Rebuild with a command similar to
 

```
USE_CUSTOM_DEBINFO="aten/src/ATen/native/Linear.cpp;newfile.cpp" python
setup.py develop
```
7. Start your debugger session again.

Again, the key line environment variable to set is `USE_CUSTOM_DEBINFO`.

## Ensuring your file is built

In our experience, you'll need to do a full clean and rebuild each time.

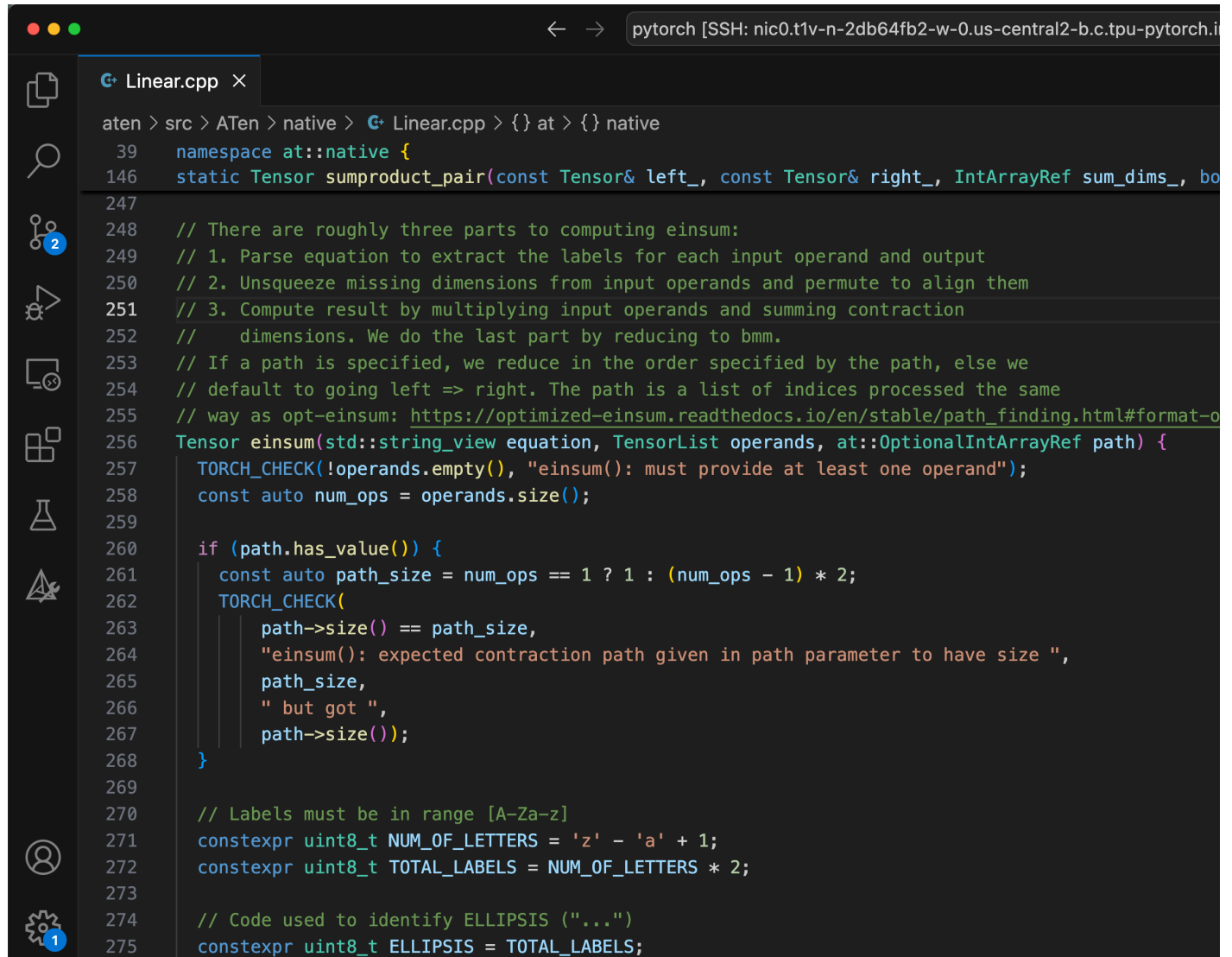
In the output of the `python setup.py develop` command, look for a line that starts with `Source files with custom debug infos` to make sure your file is correctly being built with debugger symbols. For example:

```
-- Public CUDA Deps.      :
-- Private CUDA Deps.    :
-- USE_COREML_DELEGATE    : OFF
-- BUILD_LAZY_TS_BACKEND  : ON
-- USE_ROCM_KERNEL_ASSERT : OFF
-- Source files with custom debug infos: torch/nn/functional.py
-- Configuring done (11.2s)
-- Generating done (1.4s)
-- Build files have been written to: /home/yho_google_com/pytorch/build
[6/34] Building CXX object
caffe2/CMakeFiles/torch_cpu.dir/__/aten/src/ATen/native/mkldnn/...
```

It may also be helpful to ensure that your previous debugger session is shut down.

## Using VSCode's Visual Debugger

First, open the file you want to debug.



```
aten > src > ATen > native > Linear.cpp > {} at > {} native
39 namespace at::native {
146 static Tensor sumproduct_pair(const Tensor& left_, const Tensor& right_, IntArrayRef sum_dims_, bo
247
248 // There are roughly three parts to computing einsum:
249 // 1. Parse equation to extract the labels for each input operand and output
250 // 2. Unsqueeze missing dimensions from input operands and permute to align them
251 // 3. Compute result by multiplying input operands and summing contraction
252 // dimensions. We do the last part by reducing to bmm.
253 // If a path is specified, we reduce in the order specified by the path, else we
254 // default to going left => right. The path is a list of indices processed the same
255 // way as opt-einsum: https://optimized-einsum.readthedocs.io/en/stable/path\_finding.html#format-o
256 Tensor einsum(std::string_view equation, TensorList operands, at::OptionalIntArrayRef path) {
257     TORCH_CHECK(!operands.empty(), "einsum(): must provide at least one operand");
258     const auto num_ops = operands.size();
259
260     if (path.has_value()) {
261         const auto path_size = num_ops == 1 ? 1 : (num_ops - 1) * 2;
262         TORCH_CHECK(
263             path->size() == path_size,
264             "einsum(): expected contraction path given in path parameter to have size ",
265             path_size,
266             " but got ",
267             path->size());
268     }
269
270     // Labels must be in range [A-Za-z]
271     constexpr uint8_t NUM_OF_LETTERS = 'z' - 'a' + 1;
272     constexpr uint8_t TOTAL_LABELS = NUM_OF_LETTERS * 2;
273
274     // Code used to identify ELLIPSIS ("...")
275     constexpr uint8_t ELLIPSIS = TOTAL_LABELS;
```

Second, click to the left of the line number you want to set a breakpoint. You'll see a red dot there, and, in the lower left of the debugger tab.

The screenshot shows a debugger interface with a sidebar on the left and a code editor on the right. The sidebar contains several sections:

- VARIABLES**: A section for viewing current variables.
- WATCH**: A section for adding watchpoints.
- CALL STACK**: A section for viewing the current call stack.
- BREAKPOINTS**: A section for managing breakpoints. It shows a checkbox for "All C++ Exceptions" (unchecked) and a checked breakpoint for "Linear.cpp aten/src/AT..." at line 256.

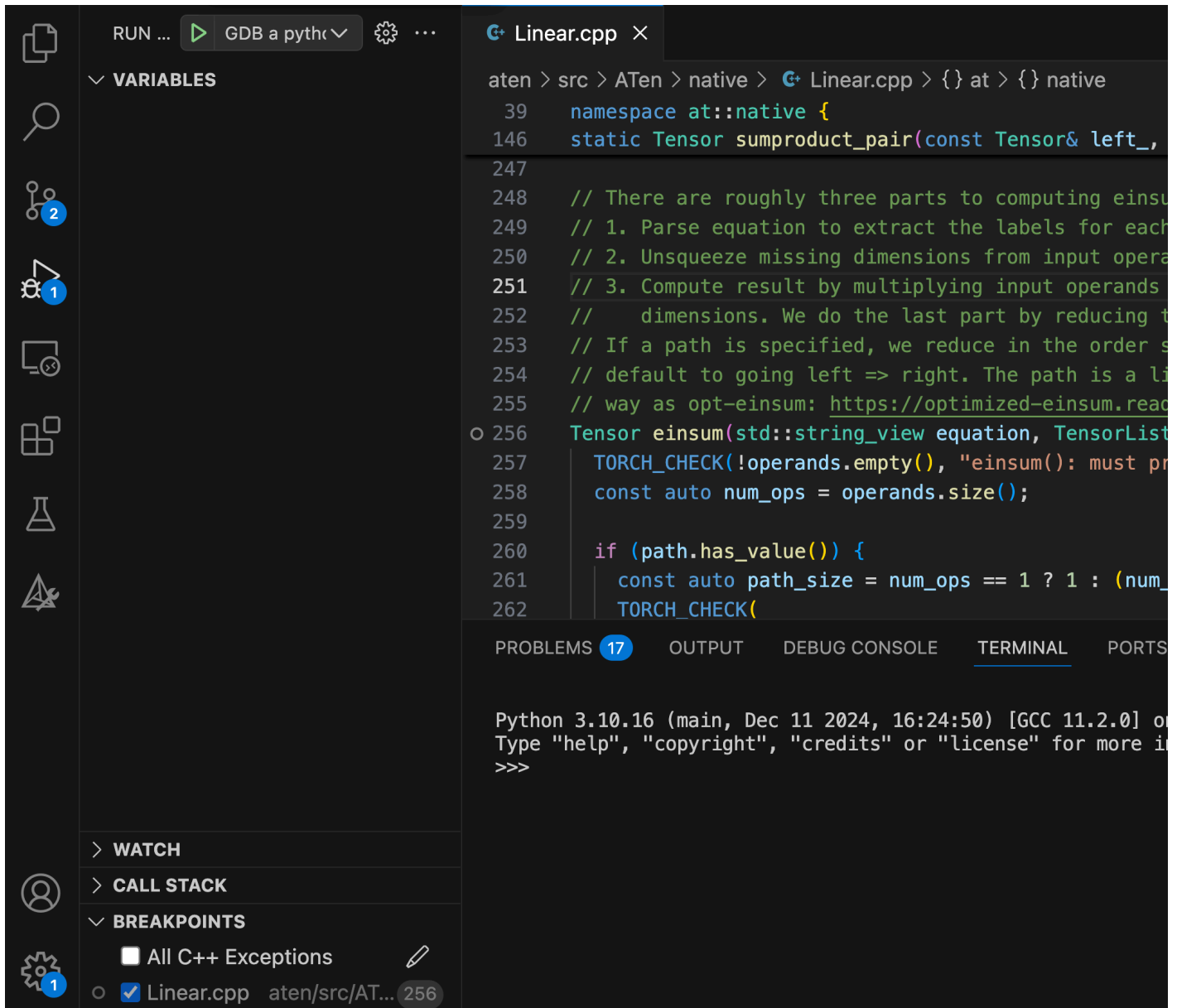
The code editor displays the following C++ code:

```

aten > src > ATen > native > C++ Linear
39 namespace at::native {
146 static Tensor sumproduct_
247
248 // There are roughly three
249 // 1. Parse equation to e
250 // 2. Unsqueeze missing d
251 // 3. Compute result by m
252 // dimensions. We do t
253 // If a path is specified
254 // default to going left
255 // way as opt-einsum: htt
256 Tensor einsum(std::string
257 TORCH_CHECK(!operands.e
258 const auto num_ops = op
259
260 if (path.has_value()) {
261     const auto path_size
262     TORCH_CHECK(
263         path->size() == p
264         "einsum(): expect
265         path_size,
266         " but got ",
267         path->size());
268 }
269
270 // Labels must be in ra
271 constexpr uint8_t NUM_0
272 constexpr uint8_t TOTAL
273
274 // Code used to identif
275 constexpr uint8_t ELLIP

```

Third, start the debugger session with the green play button. This triggers the rules we set in the launch.json file. Notice that the red button now fades to an empty white circle. That's because you have an active session, but the debugger does not know if it will actually break there. The reason is that you have not imported torch yet, so the torch library you built with debugger symbols is not yet loaded.



In the python interpreter, `import torch`. Now, the breakpoints are active because torch has loaded the underlying C++ libraries.

The screenshot displays a C++ debugger interface. On the left, a sidebar contains navigation icons and panels for **VARIABLES**, **WATCH**, **CALL STACK**, and **BREAKPOINTS**. The **BREAKPOINTS** panel shows a breakpoint set at `Linear.cpp` at line `256`. The main editor shows the source code for `Linear.cpp` with a red dot indicating the breakpoint at line 256. The code includes a namespace `at::native` and a function `sumproduct_pair`. The `einsum` function is defined, and the breakpoint is set at the start of the `if` block that checks for a path value. The bottom panel shows the terminal output of a Python shell, indicating that the breakpoint was triggered by the `torch.einsum()` command.

```

aten > src > ATen > native > Linear.cpp > {} at > {} native
39 namespace at::native {
146 static Tensor sumproduct_pair(const Tensor& le
247
248 // There are roughly three parts to computing
249 // 1. Parse equation to extract the labels for
250 // 2. Unsqueeze missing dimensions from input
251 // 3. Compute result by multiplying input oper
252 // dimensions. We do the last part by reduc
253 // If a path is specified, we reduce in the or
254 // default to going left => right. The path is
255 // way as opt-einsum: https://optimized-einsum
256 Tensor einsum(std::string_view equation, Tens
257 TORCH_CHECK(!operands.empty(), "einsum(): mu
258 const auto num_ops = operands.size();
259
260 if (path.has_value()) {
261     const auto path_size = num_ops == 1 ? 1 :
262     TORCH_CHECK(

```

PROBLEMS 17 OUTPUT DEBUG CONSOLE TERMINAL

```

Python 3.10.16 (main, Dec 11 2024, 16:24:50) [GCC 11.2
Type "help", "copyright", "credits" or "license" for m
>>> import torch
>>>

```

> WATCH  
> CALL STACK  
 All C++ Exceptions  
 Linear.cpp aten/src/AT... 256

Run a command that will trigger the breakpoint, namely, `torch.einsum()`. Notice the yellow arrow indicating the current file location, and the variables and call stack information on the left. You are debugging!

The screenshot shows a debugger interface with the following components:

- Top Bar:** "RUN ..." button, "GDB a pythc" dropdown, and "Linear.cpp" tab.
- Left Sidebar:**
  - VARIABLES:** Locals section with `__func__ = [7]`, `num_ops = <optimized out>`, `NUM_OF_LETTERS = <optimized ...>`, and `TOTAL_LABELS = <optimized ou...>`.
  - WATCH:** Empty section.
  - CALL STACK:** Shows the current function `python...` is **PAUSED ON BREAKPOINT**. The stack includes `libtorch_cpu.so!at::native::ei`, `libtorch_cpu.so!c10::impl::wrc`, `libtorch_cpu.so!at::_ops::eins`, `libtorch_python.so!torch::autc`, `cfunction_call(PyObject * func`, `_PyObject_MakeTpCall(PyThreadS`, `_PyObject_VectorcallTstate(PyC`, `_PyObject_VectorcallTstate(PyC`, `PyObject_Vectorcall(PyObject *`, `call_function(PyObject * kwnan`, `_PyEval_EvalFrameDefault(PyThr`, `_PyEval_EvalFrame(int throwflc`, `_PyEval_Vector(PyObject * kwnc`, and `_PyFunction_Vectorcall(PyObjec`.
  - BREAKPOINTS:**  All C++ Exceptions,  Linear.cpp `aten/src/AT...` 256.
- Main Window:** Source code for `Linear.cpp` at `aten > src > ATen > native > Linear.cpp > {} at > {} native > {}`. The code shows a `namespace at::native {` block with a `static Tensor sumproduct_pair(const Tensor& left,` function. A breakpoint is set at line 256: `Tensor einsum(std::string_view equation, TensorL`. The code includes comments about parsing equations and a `TORCH_CHECK(!operands.empty(), "einsum(): must` assertion.
- Bottom Panel:**
  - PROBLEMS:** 17 items.
  - OUTPUT:** Empty.
  - DEBUG CONSOLE:** Empty.
  - TERMINAL:** Shows Python 3.10.16 (main, Dec 11 2024, 16:24:50) [GCC 11.2.0] with the following output:

```
>>> import torch
>>> torch.einsum("a->", torch.tensor([1.0]))
[]
```

## A realistic build command

As you build, you'll collect additional special flags to speed things up, like not building certain libraries or adding diagnostics. Here is one example:

```
USE_CUSTOM_DEBINFO="aten/src/ATen/native/Linear.cpp" USE_CUDA=0
LD_LIBRARY_PATH=/home/USERNAME/miniconda3/envs/torch310/lib CFLAGS="--
DHAS_TORCH_SHOW_DISPATCH_TRACE" python setup.py develop
```

## Future Work

This guide has demonstrated how to build debugging symbols for PyTorch. However, comparable flags are not available in the PyTorch/XLA build scripts yet. Future work will include adding those debugger symbol flags so contributors can debug not just PyTorch C++ files, but also PyTorch/XLA C++ files.