Features for large-scale deployments

- Fleet-wide operator profiling
- API usage logging
- Attaching metadata to saved TorchScript models
- Build environment considerations
- Common extension points

This note talks about several extension points and tricks that might be useful when running PyTorch within a larger system or operating multiple systems using PyTorch in a larger organization.

It doesn't cover topics of deploying models to production. Check :mod:`torch.jit` or one of the corresponding tutorials.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]large_scale_deployments.rst, line 10); *backlink*
>
> Unknown interpreted text role "mod".

The note assumes that you either build PyTorch from source in your organization or have an ability to statically link additional code to be loaded when PyTorch is used. Therefore, many of the hooks are exposed as C++ APIs that can be triggered once in a centralized place, e.g. in static initialization code.

Fleet-wide operator profiling

PyTorch comes with :mod:`torch.autograd.profiler` capable of measuring time taken by individual operators on demand. One can use the same mechanism to do "always ON" measurements for any process running PyTorch. It might be useful for gathering information about PyTorch workloads running in a given process or across the entire set of machines.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]large_scale_deployments.rst, line 22); *backlink*
>
> Unknown interpreted text role "mod".

New callbacks for any operator invocation can be added with `torch::addGlobalCallback`. Hooks will be called with `torch::RecordFunction` struct that describes invocation context (e.g. *name*). If enabled, `RecordFunction::inputs()` contains arguments of the function represented as `torch::IValue` variant type. Note, that inputs logging is relatively expensive and thus has to be enabled explicitly.

The operator callbacks also have access to `c10::ThreadLocalDebugInfo::get()` interface that returns a pointer to the struct holding the debug information. This debug information can be set earlier by using `at::DebugInfoGuard` object. Debug information is propagated through the forward (including async `fork` tasks) and backward passes and can be useful for passing some extra information about execution environment (e.g. model id) from the higher layers of the application down to the operator callbacks.

Invoking callbacks adds some overhead, so usually it's useful to just randomly sample operator invocations. This can be enabled on per-callback basis with an optional sampling rate passed into `torch::addGlobalCallback`.

Note, that `addGlobalCallback` is not thread-safe and can be called only when no PyTorch operator is running. Usually, it's a good idea to call them once during initialization.

Here's an example:

```
// Called somewhere in the program beginning
void init() {
    // Sample one in a hundred operator runs randomly
    addGlobalCallback(
      RecordFunctionCallback(
        &onFunctionEnter,
        &onFunctionExit)
      .needsInputs(true)
      .samplingProb(0.01)
    );
    // Note, to enable observers in the model calling thread,
    // call enableRecordFunction() in the thread before running a model
}

void onFunctionEnter(const RecordFunction& fn) {
    std::cerr << "Before function " << fn.name()
              << " with " << fn.inputs().size() << " inputs" << std::endl;
}
```

```
void onFunctionExit(const RecordFunction& fn) {
    std::cerr << "After function " << fn.name();
}
```

API usage logging

When running in a broader ecosystem, for example in managed job scheduler, it's often useful to track which binaries invoke particular PyTorch APIs. There exists simple instrumentation injected at several important API points that triggers a given callback. Because usually PyTorch is invoked in one-off python scripts, the callback fires only once for a given process for each of the APIs.

`c10::SetAPIUsageHandler` can be used to register API usage instrumentation handler. Passed argument is going to be an "api key" identifying used point, for example `python.import` for PyTorch extension import or `torch.script.compile` if TorchScript compilation was triggered.

```
SetAPIUsageLogger([](const std::string& event_name) {
    std::cerr << "API was used: " << event_name << std::endl;
});
```

Note for developers: new API trigger points can be added in code with `C10_LOG_API_USAGE_ONCE("my_api")` in C++ or `torch._C._log_api_usage_once("my.api")` in Python.

Attaching metadata to saved TorchScript models

TorchScript modules can be saved as an archive file that bundles serialized parameters and module code as TorchScript (see :meth:`torch.jit.save`). It's often convenient to bundle additional information together with the model, for example, description of model producer or auxiliary artifacts.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]large_scale_deployments.rst, line 105); *backlink***
>
> Unknown interpreted text role "meth".

It can be achieved by passing the `_extra_files` argument to :meth:`torch.jit.save` and `torch::jit::load` to store and retrieve arbitrary binary blobs during saving process. Since TorchScript files are regular ZIP archives, extra information gets stored as regular files inside archive's `extra/` directory.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]large_scale_deployments.rst, line 110); *backlink***
>
> Unknown interpreted text role "meth".

There's also a global hook allowing to attach extra files to any TorchScript archive produced in the current process. It might be useful to tag models with producer metadata, akin to JPEG metadata produced by digital cameras. Example usage might look like:

```
SetExportModuleExtraFilesHook([](const Module&) {
    ExtraFilesMap files;
    files["producer_info.json"] = "{\"user\": \"" + getenv("USER") + "\"}";
    return files;
});
```

Build environment considerations

TorchScript's compilation needs to have access to the original python files as it uses python's `inspect.getsource` call. In certain production environments it might require explicitly deploying `.py` files along with precompiled `.pyc`.

Common extension points

PyTorch APIs are generally loosely coupled and it's easy to replace a component with specialized version. Common extension points include:

- Custom operators implemented in C++ - see tutorial for more details.
- Custom data reading can be often integrated directly by invoking corresponding python library. Existing functionality of :mod:`torch.utils.data` can be utilized by extending :class:`~torch.utils.data.Dataset` or :class:`~torch.utils.data.IterableDataset`.

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]large_scale_deployments.rst, **line 145**); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]large_scale_deployments.rst, **line 145**); *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]large_scale_deployments.rst, **line 145**); *backlink*

Unknown interpreted text role "class".