accelerate-main/LICENSE

accelerate-main/setup.cfg

```
[isort]
default_section = FIRSTPARTY
ensure_newline_before_comments = True
force_grid_wrap = 0
include_trailing_comma = True
known_first_party = accelerate
line_length = 119
lines_after_imports = 2
multi_line_output = 3
use_parentheses = True

[flake8]
ignore = E203, E722, E501, E741, W503, W605
max-line-length = 119
```

accelerate-main/pyproject.toml

```
[tool.black]
line-length = 119
target-version = ['py37']

[tool.ruff]
# Never enforce `E501` (line length violations).
ignore = ["E501", "E741", "W605"]
select = ["E", "F", "I", "W"]
line-length = 119

# Ignore import violations in all `__init__.py` files.
[tool.ruff.per-file-ignores]
"__init__.py" = ["E402", "F401", "F403", "F811"]

[tool.ruff.isort]
lines-after-imports = 2
known-first-party = ["accelerate"]
```

accelerate-main/docs/Makefile

```
# Minimal makefile for Sphinx documentation
#

# You can set these variables from the command line.
SPHINXOPTS    =
SPHINXBUILD   = sphinx-build
SOURCEDIR     = source
BUILDDIR      = _build

# Put it first so that "make" without argument is like "make help".
help:
■@$(SPHINXBUILD) -M help "$(SOURCEDIR)" "$(BUILDDIR)" $(SPHINXOPTS) $(O)

.PHONY: help Makefile

# Catch-all target: route all unknown targets to Sphinx using the new
# "make mode" option.  $(O) is meant as a shortcut for $(SPHINXOPTS).
%: Makefile
■@$(SPHINXBUILD) -M $@ "$(SOURCEDIR)" "$(BUILDDIR)" $(SPHINXOPTS) $(O)
```

accelerate-main/CONTRIBUTING.md

```
# How to contribute to ■ Accelerate?

Everyone is welcome to contribute, and we value everybody's contribution. Code
is thus not the only way to help the community. Answering questions, helping
others, reaching out and improving the documentations are immensely valuable to
the community.

It also helps us if you spread the word: reference the library from blog posts
on the awesome projects it made possible, shout out on Twitter every time it has
helped you, or simply star the repo to say "thank you".

Whichever way you choose to contribute, please be mindful to respect our
[code of conduct](https://github.com/huggingface/accelerate/blob/main/CODE_OF_CONDUCT.md).

## You can contribute in so many ways!

Some of the ways you can contribute to Accelerate:
* Fixing outstanding issues with the existing code;
* Contributing to the examples or to the documentation;
* Submitting issues related to bugs or desired new features.

## Submitting a new issue or feature request

Do your best to follow these guidelines when submitting an issue or a feature
request. It will make it easier for us to come back to you quickly and with good
feedback.

### Did you find a bug?

The ■ Accelerate library is robust and reliable thanks to the users who notify us of
the problems they encounter. So thank you for reporting an issue.

First, we would really appreciate it if you could **make sure the bug was not
already reported** (use the search bar on Github under Issues).

Did not find it? :( So we can act quickly on it, please follow these steps:

* Include your **OS type and version**, the versions of **Python** and **PyTorch**.
* A short, self-contained, code snippet that allows us to reproduce the bug in
  less than 30s;
* Provide the with your Accelerate configuration (located by default in `~/.cache/huggingface/accelerate

### Do you want a new feature?

A good feature request addresses the following points:

1. Motivation first:
* Is it related to a problem/frustration with the library? If so, please explain
  why. Providing a code snippet that demonstrates the problem is best.
* Is it related to something you would need for a project? We'd love to hear
  about it!
* Is it something you worked on and think could benefit the community?
```

  Awesome! Tell us what problem it solved for you.
2. Write a *full paragraph* describing the feature;
3. Provide a **code snippet** that demonstrates its future use;
4. In case this is related to a paper, please attach a link;
5. Attach any additional information (drawings, screenshots, etc.) you think may help.

If your issue is well written we're already 80% of the way there by the time you
post it.

## Submitting a pull request (PR)

Before writing code, we strongly advise you to search through the existing PRs or
issues to make sure that nobody is already working on the same thing. If you are
unsure, it is always a good idea to open an issue to get some feedback.

You will need basic `git` proficiency to be able to contribute to
■ Accelerate. `git` is not the easiest tool to use but it has the greatest
manual. Type `git --help` in a shell and enjoy. If you prefer books, [Pro
Git](https://git-scm.com/book/en/v2) is a very good reference.

Follow these steps to start contributing:

1. Fork the [repository](https://github.com/huggingface/accelerate) by
   clicking on the 'Fork' button on the repository's page. This creates a copy of the code
   under your GitHub user account.

2. Clone your fork to your local disk, and add the base repository as a remote. The following command
   assumes you have your public SSH key uploaded to GitHub. See the following guide for more
   [information](https://docs.github.com/en/repositories/creating-and-managing-repositories/cloning-a-re

   ```bash
   $ git clone git@github.com:<your Github handle>/accelerate.git
   $ cd accelerate
   $ git remote add upstream https://github.com/huggingface/accelerate.git
   ```

3. Create a new branch to hold your development changes, and do this for every new PR you work on.

   Start by synchronizing your `main` branch with the `upstream/main` branch (ore details in the [GitHub

   ```bash
   $ git checkout main
   $ git fetch upstream
   $ git merge upstream/main
   ```

   Once your `main` branch is synchronized, create a new branch from it:

   ```bash
   $ git checkout -b a-descriptive-name-for-my-changes
   ```

   Do not work on the `main` branch.

4. Set up a development environment by running the following command in a conda or a virtual environment

   ```bash
   $ pip install -e ".[quality]"
   ```

   (If accelerate was already installed in the virtual environment, remove
   it with `pip uninstall accelerate` before reinstalling it in editable
   mode with the `-e` flag.)

   Alternatively, if you are using [Visual Studio Code](https://code.visualstudio.com/Download), the fas
   the provided Dev Container. Documentation on how to get started with dev containers is available [her

5. Develop the features on your branch.

   As you work on the features, you should make sure that the test suite
   passes. You should run the tests impacted by your changes like this (see
   below an explanation regarding the environment variable):

```bash
   $ pytest tests/<TEST_TO_RUN>.py
   ```

   > For the following commands leveraging the `make` utility, we recommend using the WSL system when ru
   > Windows. More information [here](https://docs.microsoft.com/en-us/windows/wsl/about).

   You can also run the full suite with the following command.

   ```bash
   $ make test
   ```

   `accelerate` relies on `black` and `ruff` to format its source code
   consistently. After you make changes, apply automatic style corrections and code verifications
   that can't be automated in one go with:

   This target is also optimized to only work with files modified by the PR you're working on.

   If you prefer to run the checks one after the other, the following command apply the
   style corrections:

   ```bash
   $ make style
   ```

   `accelerate` also uses a few custom scripts to check for coding mistakes. Quality
   control runs in CI, however you can also run the same checks with:

   ```bash
   $ make quality
   ```

   Once you're happy with your changes, add changed files using `git add` and
   make a commit with `git commit` to record your changes locally:

   ```bash
   $ git add modified_file.py
   $ git commit
   ```

   Please write [good commit messages](https://chris.beams.io/posts/git-commit/).

   It is a good idea to sync your copy of the code with the original
   repository regularly. This way you can quickly account for changes:

   ```bash
   $ git fetch upstream
   $ git rebase upstream/main
   ```

   Push the changes to your account using:

   ```bash
   $ git push -u origin a-descriptive-name-for-my-changes
   ```

6. Once you are satisfied (**and the checklist below is happy too**), go to the
   webpage of your fork on GitHub. Click on 'Pull request' to send your changes
   to the project maintainers for review.

7. It's ok if maintainers ask you for changes. It happens to core contributors
   too! So everyone can see the changes in the Pull request, work in your local
   branch and push the changes to your fork. They will automatically appear in
   the pull request.

### Checklist

1. The title of your pull request should be a summary of its contribution;
2. If your pull request addresses an issue, please mention the issue number in
   the pull request description to make sure they are linked (and people

   consulting the issue know you are working on it);
3. To indicate a work in progress please prefix the title with `[WIP]`, or mark
   the PR as a draft PR. These are useful to avoid duplicated work, and to differentiate
   it from PRs ready to be merged;
4. Make sure existing tests pass;
5. Add high-coverage tests. No quality testing = no merge.

See an example of a good PR here: https://github.com/huggingface/accelerate/pull/255

### Tests

An extensive test suite is included to test the library behavior and several examples. Library tests can
the [tests folder](https://github.com/huggingface/accelerate/tree/main/tests).

We use `pytest` in order to run the tests. From the root of the
repository, here's how to run tests with `pytest` for the library:

```bash
$ python -m pytest -sv ./tests
```

In fact, that's how `make test` is implemented (sans the `pip install` line)!

You can specify a smaller set of tests in order to test only the feature
you're working on.

accelerate-main/setup.cfg

```
[isort]
default_section = FIRSTPARTY
ensure_newline_before_comments = True
force_grid_wrap = 0
include_trailing_comma = True
known_first_party = accelerate
line_length = 119
lines_after_imports = 2
multi_line_output = 3
use_parentheses = True

[flake8]
ignore = E203, E722, E501, E741, W503, W605
max-line-length = 119
```

accelerate-main/CODE_OF_CONDUCT.md

```
# Contributor Covenant Code of Conduct

## Our Pledge

We as members, contributors, and leaders pledge to make participation in our
community a harassment-free experience for everyone, regardless of age, body
size, visible or invisible disability, ethnicity, sex characteristics, gender
identity and expression, level of experience, education, socio-economic status,
nationality, personal appearance, race, religion, or sexual identity
and orientation.

We pledge to act and interact in ways that contribute to an open, welcoming,
diverse, inclusive, and healthy community.

## Our Standards

Examples of behavior that contributes to a positive environment for our
community include:

* Demonstrating empathy and kindness toward other people
* Being respectful of differing opinions, viewpoints, and experiences
* Giving and gracefully accepting constructive feedback
* Accepting responsibility and apologizing to those affected by our mistakes,
  and learning from the experience
* Focusing on what is best not just for us as individuals, but for the
  overall community

Examples of unacceptable behavior include:

* The use of sexualized language or imagery, and sexual attention or
  advances of any kind
* Trolling, insulting or derogatory comments, and personal or political attacks
* Public or private harassment
* Publishing others' private information, such as a physical or email
  address, without their explicit permission
* Other conduct which could reasonably be considered inappropriate in a
  professional setting

## Enforcement Responsibilities

Community leaders are responsible for clarifying and enforcing our standards of
acceptable behavior and will take appropriate and fair corrective action in
response to any behavior that they deem inappropriate, threatening, offensive,
or harmful.

Community leaders have the right and responsibility to remove, edit, or reject
comments, commits, code, wiki edits, issues, and other contributions that are
not aligned to this Code of Conduct, and will communicate reasons for moderation
decisions when appropriate.

## Scope

This Code of Conduct applies within all community spaces, and also applies when
an individual is officially representing the community in public spaces.
Examples of representing our community include using an official e-mail address,
posting via an official social media account, or acting as an appointed
representative at an online or offline event.

## Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be
reported to the community leaders responsible for enforcement at
feedback@huggingface.co.
All complaints will be reviewed and investigated promptly and fairly.

All community leaders are obligated to respect the privacy and security of the
reporter of any incident.
## Enforcement Guidelines
```

Community leaders will follow these Community Impact Guidelines in determining
the consequences for any action they deem in violation of this Code of Conduct:

### 1. Correction

Community Impact: Use of inappropriate language or other behavior deemed
unprofessional or unwelcome in the community.

Consequence: A private, written warning from community leaders, providing
clarity around the nature of the violation and an explanation of why the
behavior was inappropriate. A public apology may be requested.

### 2. Warning

Community Impact: A violation through a single incident or series
of actions.

Consequence: A warning with consequences for continued behavior. No
interaction with the people involved, including unsolicited interaction with
those enforcing the Code of Conduct, for a specified period of time. This
includes avoiding interactions in community spaces as well as external channels
like social media. Violating these terms may lead to a temporary or
permanent ban.

### 3. Temporary Ban

Community Impact: A serious violation of community standards, including
sustained inappropriate behavior.

Consequence: A temporary ban from any sort of interaction or public
communication with the community for a specified period of time. No public or
private interaction with the people involved, including unsolicited interaction
with those enforcing the Code of Conduct, is allowed during this period.
Violating these terms may lead to a permanent ban.

### 4. Permanent Ban

Community Impact: Demonstrating a pattern of violation of community
standards, including sustained inappropriate behavior, harassment of an
individual, or aggression toward or disparagement of classes of individuals.

Consequence: A permanent ban from any sort of public interaction within
the community.

## Attribution

This Code of Conduct is adapted from the [Contributor Covenant][homepage],
version 2.0, available at
https://www.contributor-covenant.org/version/2/0/code_of_conduct.html.

Community Impact Guidelines were inspired by [Mozilla's code of conduct
enforcement ladder](https://github.com/mozilla/diversity).

[homepage]: https://www.contributor-covenant.org

For answers to common questions about this code of conduct, see the FAQ at
https://www.contributor-covenant.org/faq. Translations are available at
https://www.contributor-covenant.org/translations.

accelerate-main/.gitignore

```
# Byte-compiled / optimized / DLL files
__pycache__/
*.py[cod]
*$py.class

# C extensions
*.so

# Distribution / packaging
.Python
build/
develop-eggs/
dist/
downloads/
eggs/
.eggs/
lib/
lib64/
parts/
sdist/
var/
wheels/
pip-wheel-metadata/
share/python-wheels/
*.egg-info/
.installed.cfg
*.egg
MANIFEST

# PyInstaller
#  Usually these files are written by a python script from a template
#  before PyInstaller builds the exe, so as to inject date/other infos into it.
*.manifest
*.spec

# Installer logs
pip-log.txt
pip-delete-this-directory.txt

# Unit test / coverage reports
htmlcov/
.tox/
.nox/
.coverage
.coverage.*
.cache
nosetests.xml
coverage.xml
*.cover
*.py,cover
.hypothesis/
.pytest_cache/

# Translations
*.mo
*.pot

# Django stuff:
*.log
local_settings.py
db.sqlite3
db.sqlite3-journal

# Flask stuff:
instance/
.webassets-cache

# Scrapy stuff:
```

```
.scrapy

# Sphinx documentation
docs/_build/

# PyBuilder
target/

# Jupyter Notebook
.ipynb_checkpoints

# IPython
profile_default/
ipython_config.py

# pyenv
.python-version

# pipenv
#   According to pypa/pipenv#598, it is recommended to include Pipfile.lock in version control.
#   However, in case of collaboration, if having platform-specific dependencies or dependencies
#   having no cross-platform support, pipenv may install dependencies that don't work, or not
#   install all needed dependencies.
#Pipfile.lock

# PEP 582; used by e.g. github.com/David-OConnor/pyflow
__pypackages__/

# Celery stuff
celerybeat-schedule
celerybeat.pid

# SageMath parsed files
*.sage.py

# Environments
.env
.venv
env/
venv/
ENV/
env.bak/
venv.bak/

# Spyder project settings
.spyderproject
.spyproject

# Rope project settings
.ropeproject

# mkdocs documentation
/site

# mypy
.mypy_cache/
.dmypy.json
dmypy.json

# Pyre type checker
.pyre/

# VSCode
.vscode

# IntelliJ
.idea

# Mac .DS_Store
.DS_Store

# More test things
```

```
wandb

# ruff
.ruff_cache
```

accelerate-main/examples/by_feature/README.md

~~~
# What are these scripts?

All scripts in this folder originate from the `nlp_example.py` file, as it is a very simplistic NLP trai

From there, each further script adds in just **one** feature of Accelerate, showing how you can quickly

A full example with all of these parts integrated together can be found in the `complete_nlp_example.py`

Adjustments to each script from the base `nlp_example.py` file can be found quickly by searching for "#

## Example Scripts by Feature and their Arguments

### Base Example (`../nlp_example.py`)

- Shows how to use `Accelerator` in an extremely simplistic PyTorch training loop
- Arguments available:
  - `mixed_precision`, whether to use mixed precision. ("no", "fp16", or "bf16")
  - `cpu`, whether to train using only the CPU. (yes/no/1/0)

All following scripts also accept these arguments in addition to their added ones.

These arguments should be added at the end of any method for starting the python script (such as `python

```bash
accelerate launch ../nlp_example.py --mixed_precision fp16 --cpu 0
```

### Checkpointing and Resuming Training (`checkpointing.py`)

- Shows how to use `Accelerator.save_state` and `Accelerator.load_state` to save or continue training
- **It is assumed you are continuing off the same training script**
- Arguments available:
  - `checkpointing_steps`, after how many steps the various states should be saved. ("epoch", 1, 2, ...)
  - `output_dir`, where saved state folders should be saved to, default is current working directory
  - `resume_from_checkpoint`, what checkpoint folder to resume from. ("epoch_0", "step_22", ...)

These arguments should be added at the end of any method for starting the python script (such as `python

(Note, `resume_from_checkpoint` assumes that we've ran the script for one epoch with the `--checkpointin

```bash
accelerate launch ./checkpointing.py --checkpointing_steps epoch output_dir "checkpointing_tutorial" --r
```

### Cross Validation (`cross_validation.py`)

- Shows how to use `Accelerator.free_memory` and run cross validation efficiently with `datasets`.
- Arguments available:
  - `num_folds`, the number of folds the training dataset should be split into.

These arguments should be added at the end of any method for starting the python script (such as `python

```bash
accelerate launch ./cross_validation.py --num_folds 2
```

### Experiment Tracking (`tracking.py`)

- Shows how to use `Accelerate.init_trackers` and `Accelerator.log`
- Can be used with Weights and Biases, TensorBoard, or CometML.
- Arguments available:
  - `with_tracking`, whether to load in all available experiment trackers from the environment.

These arguments should be added at the end of any method for starting the python script (such as `python

```bash
accelerate launch ./tracking.py --with_tracking
```
~~~

### Gradient Accumulation (`gradient_accumulation.py`)

- Shows how to use `Accelerator.no_sync` to prevent gradient averaging in a distributed setup.
- Arguments available:
  - `gradient_accumulation_steps`, the number of steps to perform before the gradients are accumulated a

These arguments should be added at the end of any method for starting the python script (such as `python

```bash
accelerate launch ./gradient_accumulation.py --gradient_accumulation_steps 5
```

accelerate-main/benchmarks/big_model_inference.py

```python
# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import argparse
import time

import torch
import transformers
from measures_util import end_measure, log_measures, start_measure
from transformers import AutoConfig, AutoModelForCausalLM, AutoModelForSeq2SeqLM, AutoTokenizer

from accelerate.utils import compute_module_sizes


DEFAULT_MODELS = {
    "gpt-j-6b": {"is_causal": True, "model": "sgugger/sharded-gpt-j-6B", "tokenizer": "EleutherAI/gpt-j-
    "gpt-neox": {"is_causal": True, "model": "EleutherAI/gpt-neox-20b"},
    "opt": {"is_causal": True, "model": "facebook/opt-30b"},
    "T0pp": {"is_causal": False, "model": "bigscience/T0pp", "model_revision": "sharded"},
}

PROMPTS = [
    "Hello, my name is",
    "Are unicorns real? Unicorns are",
    "For the first time in several years,",
    "My name is Julien and I am",
    "The goal of life is",
    "Whenever I'm sad, I like to",
]


def parse_args():
    parser = argparse.ArgumentParser(description="Run and time generations on a big model using Accelera
    parser.add_argument("model_name", type=str, default=None, help="The name of the model to try.")
    parser.add_argument(
        "--tokenizer_name", type=str, default=None, help="The name of the tokenizer (if different from t
    )
    parser.add_argument("--is_causal", type=bool, default=None, help="Whether or not the model is causal
    parser.add_argument(
        "--model_revision", type=str, default=None, help="The revision to use for the model checkpoint."
    )
    parser.add_argument("--torch_dtype", type=str, default=None, help="The dtype for the model.")
    parser.add_argument("--disk_offload", action="store_true")

    args = parser.parse_args()

    # Sanitize args
    if args.model_name in DEFAULT_MODELS:
        defaults = DEFAULT_MODELS[args.model_name]
        args.model_name = defaults["model"]
        if args.tokenizer_name is None:
            args.tokenizer_name = defaults.get("tokenizer", args.model_name)
        if args.is_causal is None:
            args.is_causal = defaults["is_causal"]
        if args.model_revision is None:
            args.model_revision = defaults.get("model_revision", "main")
    if args.is_causal is None:
        raise ValueError("Could not infer the default for `--is_causal`, pass either True or False for i
    if args.tokenizer_name is None:
        args.tokenizer_name = args.model_name
    if args.model_revision is None:
        args.model_revision = "main"

    return args


def main():
    transformers.utils.logging.set_verbosity_error()
    args = parse_args()

    if args.torch_dtype is None:
        config = AutoConfig.from_pretrained(args.model_name)
        torch_dtype = getattr(config, "torch_dtype", torch.float32)
    else:
        torch_dtype = getattr(torch, args.torch_dtype)
    model_cls = AutoModelForCausalLM if args.is_causal else AutoModelForSeq2SeqLM
    kwargs = {
        "torch_dtype": torch_dtype,
        "revision": args.model_revision,
    }
    if args.disk_offload:
        kwargs["offload_folder"] = "tmp_offload"
        kwargs["offload_state_dict"] = True

    start_measures = start_measure()
    model = model_cls.from_pretrained(args.model_name, device_map="auto", **kwargs)
    end_measures = end_measure(start_measures)
    log_measures(end_measures, "Model loading")

    module_sizes = compute_module_sizes(model)
    device_size = {v: 0 for v in model.hf_device_map.values()}
    for module, device in model.hf_device_map.items():
        device_size[device] += module_sizes[module]
    message = "\n".join([f"- {device}: {size // 2**20}MiB" for device, size in device_size.items()])
    print(f"\nTheoretical use:\n{message}")

    tokenizer = AutoTokenizer.from_pretrained(args.tokenizer_name)

    start_measures = start_measure()
    generation_times = []
    gen_tokens = []
    texts_outs = []
    for prompt in PROMPTS:
        inputs = tokenizer(prompt, return_tensors="pt").to(0)
        tokens = inputs["input_ids"][0].tolist()
        before_generate = time.time()
        outputs = model.generate(inputs["input_ids"])
        after_generate = time.time()
        outputs = outputs[0].tolist()
        num_gen_tokens = len(outputs) if outputs[: len(tokens)] != tokens else len(outputs) - len(tokens
        generation_time = after_generate - before_generate

        text_out = tokenizer.decode(outputs, skip_special_tokens=True)
        texts_outs.append(text_out)
        generation_times.append(generation_time)
        gen_tokens.append(num_gen_tokens)
        print(f"Prompt: {prompt}\nGeneration {text_out}\nIn {generation_time:.2f}s for {num_gen_tokens}

    end_measures = end_measure(start_measures)
    log_measures(end_measures, "Model generation")

    generation_times_per_token = [gen / tok for gen, tok in zip(generation_times, gen_tokens)]
    avg_gen = sum(generation_times_per_token) / len(generation_times)
    print(f"Average time of generation per token: {avg_gen:.2f}s")
    print(f"First generation (avg time per token): {generation_times_per_token[0]:.2f}s")
    avg_gen = sum(generation_times_per_token[1:]) / (len(generation_times_per_token) - 1)
    print(f"Average time of generation per token (excluding the first): {avg_gen:.2f}s")
if __name__ == "__main__":
    main()
```

accelerate-main/benchmarks/measures_util.py

```
import gc
import threading
import time

import psutil
import torch


class PeakCPUMemory:
    def __init__(self):
        self.process = psutil.Process()
        self.peak_monitoring = False

    def peak_monitor(self):
        self.cpu_memory_peak = -1

        while True:
            self.cpu_memory_peak = max(self.process.memory_info().rss, self.cpu_memory_peak)

            # can't sleep or will not catch the peak right (this comment is here on purpose)
            if not self.peak_monitoring:
                break

    def start(self):
        self.peak_monitoring = True
        self.thread = threading.Thread(target=self.peak_monitor)
        self.thread.daemon = True
        self.thread.start()

    def stop(self):
        self.peak_monitoring = False
        self.thread.join()
        return self.cpu_memory_peak


cpu_peak_tracker = PeakCPUMemory()


def start_measure():
    # Time
    measures = {"time": time.time()}

    gc.collect()
    torch.cuda.empty_cache()

    # CPU mem
    measures["cpu"] = psutil.Process().memory_info().rss
    cpu_peak_tracker.start()

    # GPU mem
    for i in range(torch.cuda.device_count()):
        measures[str(i)] = torch.cuda.memory_allocated(i)
    torch.cuda.reset_peak_memory_stats()

    return measures


def end_measure(start_measures):
    # Time
    measures = {"time": time.time() - start_measures["time"]}

    gc.collect()
    torch.cuda.empty_cache()

    # CPU mem
    measures["cpu"] = (psutil.Process().memory_info().rss - start_measures["cpu"]) / 2**20
    measures["cpu-peak"] = (cpu_peak_tracker.stop() - start_measures["cpu"]) / 2**20
    # GPU mem
    for i in range(torch.cuda.device_count()):
        measures[str(i)] = (torch.cuda.memory_allocated(i) - start_measures[str(i)]) / 2**20
        measures[f"{i}-peak"] = (torch.cuda.max_memory_allocated(i) - start_measures[str(i)]) / 2**20

    return measures


def log_measures(measures, description):
    print(f"{description}:")
    print(f"- Time: {measures['time']:.2f}s")
    for i in range(torch.cuda.device_count()):
        print(f"- GPU {i} allocated: {measures[str(i)]:.2f}MiB")
        peak = measures[f"{i}-peak"]
        print(f"- GPU {i} peak: {peak:.2f}MiB")
    print(f"- CPU RAM allocated: {measures['cpu']:.2f}MiB")
    print(f"- CPU RAM peak: {measures['cpu-peak']:.2f}MiB")
```

accelerate-main/examples/by_feature/README.md

~~~
# What are these scripts?

All scripts in this folder originate from the `nlp_example.py` file, as it is a very simplistic NLP trai

From there, each further script adds in just **one** feature of Accelerate, showing how you can quickly

A full example with all of these parts integrated together can be found in the `complete_nlp_example.py`

Adjustments to each script from the base `nlp_example.py` file can be found quickly by searching for "#

## Example Scripts by Feature and their Arguments

### Base Example (`../nlp_example.py`)

- Shows how to use `Accelerator` in an extremely simplistic PyTorch training loop
- Arguments available:
  - `mixed_precision`, whether to use mixed precision. ("no", "fp16", or "bf16")
  - `cpu`, whether to train using only the CPU. (yes/no/1/0)

All following scripts also accept these arguments in addition to their added ones.

These arguments should be added at the end of any method for starting the python script (such as `python

```bash
accelerate launch ../nlp_example.py --mixed_precision fp16 --cpu 0
```

### Checkpointing and Resuming Training (`checkpointing.py`)

- Shows how to use `Accelerator.save_state` and `Accelerator.load_state` to save or continue training
- **It is assumed you are continuing off the same training script**
- Arguments available:
  - `checkpointing_steps`, after how many steps the various states should be saved. ("epoch", 1, 2, ...)
  - `output_dir`, where saved state folders should be saved to, default is current working directory
  - `resume_from_checkpoint`, what checkpoint folder to resume from. ("epoch_0", "step_22", ...)

These arguments should be added at the end of any method for starting the python script (such as `python

(Note, `resume_from_checkpoint` assumes that we've ran the script for one epoch with the `--checkpointin

```bash
accelerate launch ./checkpointing.py --checkpointing_steps epoch output_dir "checkpointing_tutorial" --r
```

### Cross Validation (`cross_validation.py`)

- Shows how to use `Accelerator.free_memory` and run cross validation efficiently with `datasets`.
- Arguments available:
  - `num_folds`, the number of folds the training dataset should be split into.

These arguments should be added at the end of any method for starting the python script (such as `python

```bash
accelerate launch ./cross_validation.py --num_folds 2
```

### Experiment Tracking (`tracking.py`)

- Shows how to use `Accelerate.init_trackers` and `Accelerator.log`
- Can be used with Weights and Biases, TensorBoard, or CometML.
- Arguments available:
  - `with_tracking`, whether to load in all available experiment trackers from the environment.

These arguments should be added at the end of any method for starting the python script (such as `python

```bash
accelerate launch ./tracking.py --with_tracking
```
~~~

### Gradient Accumulation (`gradient_accumulation.py`)

- Shows how to use `Accelerator.no_sync` to prevent gradient averaging in a distributed setup.
- Arguments available:
  - `gradient_accumulation_steps`, the number of steps to perform before the gradients are accumulated a

These arguments should be added at the end of any method for starting the python script (such as `python

```bash
accelerate launch ./gradient_accumulation.py --gradient_accumulation_steps 5
```

accelerate-main/.github/ISSUE_TEMPLATE/bug-report.yml

```
name: "\U0001F41B Bug Report"
description: Submit a bug report to help us improve Accelerate
body:
  - type: textarea
    id: system-info
    attributes:
      label: System Info
      description: Please share your accelerate configuration with us. You can run the command `accelera
      render: Shell
      placeholder: accelerate version, OS, python version, numpy version, torch version, and accelerate'
    validations:
      required: true

  - type: checkboxes
    id: information-scripts-examples
    attributes:
      label: Information
      description: 'The problem arises when using:'
      options:
        - label: "The official example scripts"
        - label: "My own modified scripts"

  - type: checkboxes
    id: information-tasks
    attributes:
      label: Tasks
      description: "The tasks I am working on are:"
      options:
        - label: "One of the scripts in the examples/ folder of Accelerate or an officially supported `n
        - label: "My own task or dataset (give details below)"

  - type: textarea
    id: reproduction
    validations:
      required: true
    attributes:
      label: Reproduction
      description: |
        Please provide a code sample that reproduces the problem you ran into. It can be a Colab link or
        If you have code snippets, error messages, stack traces please provide them here as well.
        Important! Use code tags to correctly format your code. See https://help.github.com/en/github/wr
        Do not use screenshots, as they are hard to read and (more importantly) don't allow others to co

      placeholder: |
        Steps to reproduce the behavior:

          1.
          2.
          3.

  - type: textarea
    id: expected-behavior
    validations:
      required: true
    attributes:
      label: Expected behavior
      description: "A clear and concise description of what you would expect to happen."
      render: Shell
```

accelerate-main/.github/workflows/nightly.yml

```yaml
name: Self-hosted runner with slow tests (scheduled)

on:
  workflow_dispatch:
  schedule:
    - cron: "0 2 * * *"

env:
  RUN_SLOW: "yes"
  IS_GITHUB_CI: "1"
  SLACK_API_TOKEN: ${{ secrets.SLACK_API_TOKEN }}


jobs:
  run_all_tests_single_gpu:
    runs-on: [self-hosted, docker-gpu, multi-gpu]
    env:
      CUDA_VISIBLE_DEVICES: "0"
      TEST_TYPE: "single_gpu"
    container:
      image: huggingface/accelerate-gpu:latest
      options: --gpus all --shm-size "16gb"
    defaults:
      run:
        working-directory: accelerate/
        shell: bash
    steps:
      - name: Update clone & pip install
        run: |
          source activate accelerate
          git config --global --add safe.directory '*'
          git fetch && git checkout ${{ github.sha }}
          pip install -e . --no-deps
          pip install pytest-reportlog

      - name: Run test on GPUs
        run: |
          source activate accelerate
          make test
      - name: Run examples on GPUs
        run: |
          source activate accelerate
          pip uninstall comet_ml -y
          make test_examples

      - name: Generate Report
        if: always()
        run: |
          pip install slack_sdk
          python utils/log_reports.py >> $GITHUB_STEP_SUMMARY

  run_all_tests_multi_gpu:
    runs-on: [self-hosted, docker-gpu, multi-gpu]
    env:
      CUDA_VISIBLE_DEVICES: "0,1"
      TEST_TYPE: "multi_gpu"
    container:
      image: huggingface/accelerate-gpu:latest
      options: --gpus all --shm-size "16gb"
    defaults:
      run:
        working-directory: accelerate/
        shell: bash
    steps:
      - name: Update clone
        run: |
          source activate accelerate
          git config --global --add safe.directory '*'
          git fetch && git checkout ${{ github.sha }}
          pip install -e . --no-deps
          pip install pytest-reportlog

      - name: Run core and big modeling tests on GPUs
        run: |
          source activate accelerate
          make test_big_modeling
          make test_core

      - name: Run Integration tests on GPUs
        run: |
          source activate accelerate
          make test_integrations

      - name: Run examples on GPUs
        run: |
          source activate accelerate
          pip uninstall comet_ml -y
          make test_examples

      - name: Generate Report
        if: always()
        run: |
          pip install slack_sdk
          python utils/log_reports.py >> $GITHUB_STEP_SUMMARY
```

accelerate-main/src/accelerate/commands/test.py

```
#!/usr/bin/env python

# Copyright 2021 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import argparse
import os

from accelerate.test_utils import execute_subprocess_async


def test_command_parser(subparsers=None):
    if subparsers is not None:
        parser = subparsers.add_parser("test")
    else:
        parser = argparse.ArgumentParser("Accelerate test command")

    parser.add_argument(
        "--config_file",
        default=None,
        help=(
            "The path to use to store the config file. Will default to a file named default_config.yaml
            "location, which is the content of the environment `HF_HOME` suffixed with 'accelerate', or
            "such an environment variable, your cache directory ('~/.cache' or the content of `XDG_CACHE
            "with 'huggingface'."
        ),
    )

    if subparsers is not None:
        parser.set_defaults(func=test_command)
    return parser


def test_command(args):
    script_name = os.path.sep.join(__file__.split(os.path.sep)[:-2] + ["test_utils", "scripts", "test_sc

    if args.config_file is None:
        test_args = script_name
    else:
        test_args = f"--config_file={args.config_file} {script_name}"

    cmd = ["accelerate-launch"] + test_args.split()
    result = execute_subprocess_async(cmd, env=os.environ.copy())
    if result.returncode == 0:
        print("Test is a success! You are ready for your distributed training!")


def main():
    parser = test_command_parser()
    args = parser.parse_args()
    test_command(args)


if __name__ == "__main__":
    main()
```

accelerate-main/.github/workflows/build_docker_images.yml

```yaml
name: Build Docker images (scheduled)

on:
  workflow_dispatch:
  workflow_call:
  schedule:
    - cron: "0 1 * * *"

concurrency:
  group: docker-image-builds
  cancel-in-progress: false

jobs:
  latest-cpu:
    name: "Latest Accelerate CPU [dev]"
    runs-on: ubuntu-latest
    steps:
      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v1
      - name: Check out code
        uses: actions/checkout@v2
      - name: Login to DockerHub
        uses: docker/login-action@v1
        with:
          username: ${{ secrets.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_PASSWORD }}

      - name: Build and Push CPU
        uses: docker/build-push-action@v2
        with:
          context: ./docker/accelerate-cpu
          push: true
          tags: huggingface/accelerate-cpu

  latest-cuda:
    name: "Latest Accelerate GPU [dev]"
    runs-on: ubuntu-latest
    steps:
      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v1
      - name: Check out code
        uses: actions/checkout@v2
      - name: Login to DockerHub
        uses: docker/login-action@v1
        with:
          username: ${{ secrets.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_PASSWORD }}

      - name: Build and Push GPU
        uses: docker/build-push-action@v2
        with:
          context: ./docker/accelerate-gpu
          push: true
          tags: huggingface/accelerate-gpu
```

accelerate-main/.github/workflows/run_merge_tests.yml

```
name: Self-hosted runner tests (push to "main")

on:
  workflow_call:
  workflow_dispatch:

env:
  TESTING_MOCKED_DATALOADERS: "1"
  IS_GITHUB_CI: "1"

jobs:
  run_all_tests_single_gpu:
    runs-on: [self-hosted, docker-gpu, multi-gpu]
    env:
      CUDA_VISIBLE_DEVICES: "0"
    container:
      image: huggingface/accelerate-gpu:latest
      options: --gpus all --shm-size "16gb"
    defaults:
      run:
        working-directory: accelerate/
        shell: bash
    steps:
      - name: Update clone & pip install
        run: |
          source activate accelerate
          git config --global --add safe.directory '*'
          git fetch && git checkout ${{ github.sha }}
          pip install -e .[testing,test_trackers]
          pip install pytest-reportlog

      - name: Run CLI tests
        run: |
          source activate accelerate
          make test_cli

      - name: Run test on GPUs
        run: |
          source activate accelerate
          make test
      - name: Run examples on GPUs
        run: |
          source activate accelerate
          pip uninstall comet_ml -y
          make test_examples

      - name: Generate Report
        if: always()
        run: |
          python utils/log_reports.py >> $GITHUB_STEP_SUMMARY

  run_all_tests_multi_gpu:
    runs-on: [self-hosted, docker-gpu, multi-gpu]
    container:
      image: huggingface/accelerate-gpu:latest
      options: --gpus all --shm-size "16gb"
    defaults:
      run:
        working-directory: accelerate/
        shell: bash
    steps:
      - name: Update clone
        run: |
          source activate accelerate
          git config --global --add safe.directory '*'
          git fetch && git checkout ${{ github.sha }}
          pip install -e .[testing,test_trackers]
          pip install pytest-reportlog
```

```
- name: Run CLI tests
  run: |
    source activate accelerate
    make test_cli

- name: Run test on GPUs
  run: |
    source activate accelerate
    make test

- name: Run examples on GPUs
  run: |
    source activate accelerate
    pip uninstall comet_ml -y
    make test_examples

- name: Generate Report
  if: always()
  run: |
    python utils/log_reports.py >> $GITHUB_STEP_SUMMARY
```

accelerate-main/.github/workflows/quality.yml

```
name: Quality Check

on: [pull_request]

jobs:
  quality:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v2
    - name: Set up Python 3.7
      uses: actions/setup-python@v3
      with:
        python-version: 3.7
    - name: Install Python dependencies
      run: pip install -e .[quality]
    - name: Run Quality check
      run: make quality
```

accelerate-main/.github/workflows/build_and_run_tests.yml

```yaml
name: Trigger docker images and run tests

on:
  push:
    branches:
      - main
  workflow_dispatch:

env:
  GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}

jobs:
  check-for-source:
    runs-on: ubuntu-latest
    name: Check if setup was changed
    outputs:
      changed: ${{ steps.was_changed.outputs.changed }}
    steps:
      - uses: actions/checkout@v3.1.0
        with:
          fetch-depth: "2"

      - name: Get changed files
        id: changed-files
        uses: tj-actions/changed-files@v22.2

      - name: Was setup changed
        id: was_changed
        run: |
          for file in ${{ steps.changed-files.outputs.all_changed_files }}; do
            if [ `basename "${file}"` == "setup.py" ]; then
              echo "changed=1" >> $GITHUB_OUTPUT
            fi
          done

  build-docker-containers:
    needs: check-for-source
    if: (github.event_name == 'push') && (needs.check-for-source.outputs.changed == '1')
    uses: ./.github/workflows/build_docker_images.yml
    secrets: inherit

  run-merge-tests:
    needs: build-docker-containers
    if: always()
    uses: ./.github/workflows/run_merge_tests.yml
```

accelerate-main/utils/stale.py

```
# Copyright 2022 The HuggingFace Team, the AllenNLP library authors. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
"""
Script to close stale issue. Taken in part from the AllenNLP repository.
https://github.com/allenai/allennlp.
"""
from datetime import datetime as dt
import os

from github import Github


LABELS_TO_EXEMPT = [
    "good first issue",
    "feature request",
    "wip",
]


def main():
    g = Github(os.environ["GITHUB_TOKEN"])
    repo = g.get_repo("huggingface/accelerate")
    open_issues = repo.get_issues(state="open")

    for issue in open_issues:
        comments = sorted([comment for comment in issue.get_comments()], key=lambda i: i.created_at, rev
        last_comment = comments[0] if len(comments) > 0 else None
        current_time = dt.utcnow()
        days_since_updated = (current_time - issue.updated_at).days
        days_since_creation = (current_time - issue.created_at).days
        if (
            last_comment is not None and last_comment.user.login == "github-actions[bot]"
            and days_since_updated > 7
            and days_since_creation >= 30
            and not any(label.name.lower() in LABELS_TO_EXEMPT for label in issue.get_labels())
        ):
            # Close issue since it has been 7 days of inactivity since bot mention.
            issue.edit(state="closed")
        elif (
            days_since_updated > 23
            and days_since_creation >= 30
            and not any(label.name.lower() in LABELS_TO_EXEMPT for label in issue.get_labels())
        ):
            # Add stale comment
            issue.create_comment(
                "This issue has been automatically marked as stale because it has not had "
                "recent activity. If you think this still needs to be addressed "
                "please comment on this thread.\n\nPlease note that issues that do not follow the "
                "[contributing guidelines](https://github.com/huggingface/accelerate/blob/main/CONTRIBUT
                "are likely to be ignored."
            )


if __name__ == "__main__":
    main()
```

accelerate-main/.github/workflows/build_documentation.yml

```
name: Build documentation

on:
  push:
    branches:
      - main
      - doc-builder*
      - v*-release

jobs:
   build:
    uses: huggingface/doc-builder/.github/workflows/build_main_documentation.yml@main
    with:
      commit_sha: ${{ github.sha }}
      package: accelerate
    secrets:
      token: ${{ secrets.HUGGINGFACE_PUSH }}
```

accelerate-main/.github/workflows/delete_doc_comment.yml

```yaml
name: Delete dev documentation

on:
  pull_request:
    types: [ closed ]


jobs:
  delete:
    uses: huggingface/doc-builder/.github/workflows/delete_doc_comment.yml@main
    with:
      pr_number: ${{ github.event.number }}
      package: accelerate
```

accelerate-main/.github/workflows/build_pr_documentation.yml

```yaml
name: Build PR Documentation

on:
  pull_request:

concurrency:
  group: ${{ github.workflow }}-${{ github.head_ref || github.run_id }}
  cancel-in-progress: true

jobs:
  build:
    uses: huggingface/doc-builder/.github/workflows/build_pr_documentation.yml@main
    with:
      commit_sha: ${{ github.event.pull_request.head.sha }}
      pr_number: ${{ github.event.number }}
      package: accelerate
```

accelerate-main/.github/workflows/build-docker-images-release.yml

```yaml
name: Build Docker images (releases)

on:
  workflow_dispatch:
  release:
    types: [published]

concurrency:
  group: docker-image-builds
  cancel-in-progress: false

jobs:
  get-version:
    runs-on: ubuntu-latest
    outputs:
      version: ${{ steps.step1.outputs.version }}
    steps:
      - uses: actions/checkout@v3
      - id: step1
        run: echo "version=$(python setup.py --version)" >> $GITHUB_OUTPUT

  version-cpu:
    name: "Latest Accelerate CPU [version]"
    runs-on: ubuntu-latest
    needs: get-version
    steps:
      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v1
      - name: Check out code
        uses: actions/checkout@v2
      - name: Login to DockerHub
        uses: docker/login-action@v1
        with:
          username: ${{ secrets.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_PASSWORD }}

      - name: Build and Push CPU
        uses: docker/build-push-action@v2
        with:
          context: ./docker/accelerate-cpu
          push: true
          tags: huggingface/accelerate-cpu:${{needs.get-version.outputs.version}}

  version-cuda:
    name: "Latest Accelerate GPU [version]"
    runs-on: ubuntu-latest
    needs: get-version
    steps:
      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v1
      - name: Check out code
        uses: actions/checkout@v2
      - name: Login to DockerHub
        uses: docker/login-action@v1
        with:
          username: ${{ secrets.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_PASSWORD }}

      - name: Build and Push GPU
        uses: docker/build-push-action@v2
        with:
          context: ./docker/accelerate-gpu
          push: true
          tags: huggingface/accelerate-gpu:${{needs.get-version.outputs.version}}
```

accelerate-main/.devcontainer/devcontainer.json

```
// File only needed for VSCode users to have proper Docker based interpreters
{
    "name": "accelerate_dev_environment",
    "build": {
        // ACTION NEEDED: comment/uncomment the relevant line depending on whether you are in a CPU/GPU
         "dockerfile": "../docker/accelerate-cpu/Dockerfile"
//         "dockerfile": "../docker/accelerate-gpu/Dockerfile"
    },
    "runArgs": [
        // ACTION NEEDED: uncomment the next line if your local machine has GPUs available
//         "--gpus", "all",
        // Enable the docker container to access system resources
        "--ipc", "host"
    ],
    "remoteEnv": {
        "PYTHONPATH": "${containerEnv:PATH}:${containerWorkspaceFolder}"
    },
    "extensions": [
        // Ensure we have IntelliSense in VSCode when running inside container
        "ms-python.python"
    ],
    "workspaceFolder": "/workspaces/accelerate",
    // Need git for VSCode to color code modifications. Only runs when building environment.
    "onCreateCommand": "apt-get update && apt-get install -y git && pip install -e '.[dev]'"
}
```

accelerate-main/src/accelerate/memory_utils.py

```
# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import warnings


warnings.warn(
    "memory_utils has been reorganized to utils.memory. Import `find_executable_batchsize` from the main
    "`from accelerate import find_executable_batch_size` to avoid this warning.",
    FutureWarning,
)
```

accelerate-main/docs/source/package_reference/state.mdx

```
# Stateful Classes

Below are variations of a [singleton class](https://en.wikipedia.org/wiki/Singleton_pattern) in the sens
instances share the same state, which is initialized on the first instantiation.

These classes are immutable and store information about certain configurations or
states.

[[autodoc]] state.PartialState

[[autodoc]] state.AcceleratorState

[[autodoc]] state.GradientState
```

accelerate-main/src/accelerate/data_loader.py

```
# Copyright 2021 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import math
from contextlib import suppress
from typing import List, Optional, Union

import torch
from torch.utils.data import BatchSampler, DataLoader, IterableDataset

from .logging import get_logger
from .state import AcceleratorState, DistributedType, GradientState, is_tpu_available
from .utils import (
    RNGType,
    broadcast,
    broadcast_object_list,
    concatenate,
    find_batch_size,
    get_data_structure,
    initialize_tensors,
    is_torch_version,
    send_to_device,
    slice_tensors,
    synchronize_rng_states,
)


if is_tpu_available(check_device=False):
    import torch_xla.distributed.parallel_loader as xpl

    class MpDeviceLoaderWrapper(xpl.MpDeviceLoader):
        """
        Wrapper for the xpl.MpDeviceLoader class that knows the total batch size.

        **Available attributes:**

        - **total_batch_size** (`int`) -- Total batch size of the dataloader across all processes.
            Equal to the original batch size when `split_batches=True`; otherwise the original batch siz
            number of processes

        - **total_dataset_length** (`int`) -- Total length of the inner dataset across all processes.
        """

        @property
        def total_batch_size(self):
            return self._loader.total_batch_size

        @property
        def total_dataset_length(self):
            return self._loader.total_dataset_length


logger = get_logger(__name__)

# kwargs of the DataLoader in min version 1.4.0.
_PYTORCH_DATALOADER_KWARGS = {
    "batch_size": 1,
    "shuffle": False,
    "sampler": None,
    "batch_sampler": None,
    "num_workers": 0,
    "collate_fn": None,
    "pin_memory": False,
    "drop_last": False,
    "timeout": 0,
    "worker_init_fn": None,
    "multiprocessing_context": None,
    "generator": None,
}

# kwargs added after by version
_PYTORCH_DATALOADER_ADDITIONAL_KWARGS = {
    "1.7.0": {"prefetch_factor": 2, "persistent_workers": False},
}

for v, additional_kwargs in _PYTORCH_DATALOADER_ADDITIONAL_KWARGS.items():
    if is_torch_version(">=", v):
        _PYTORCH_DATALOADER_KWARGS.update(additional_kwargs)


class BatchSamplerShard(BatchSampler):
    """
    Wraps a PyTorch `BatchSampler` to generate batches for one of the processes only. Instances of this
    always yield a number of batches that is a round multiple of `num_processes` and that all have the s
    Depending on the value of the `drop_last` attribute of the batch sampler passed, it will either stop
    at the first batch that would be too small / not present on all processes or loop with indices from

    Args:
        batch_sampler (`torch.utils.data.sampler.BatchSampler`):
            The batch sampler to split in several shards.
        num_processes (`int`, *optional*, defaults to 1):
            The number of processes running concurrently.
        process_index (`int`, *optional*, defaults to 0):
            The index of the current process.
        split_batches (`bool`, *optional*, defaults to `False`):
            Whether the shards should be created by splitting a batch to give a piece of it on each proc
            yielding different full batches on each process.

            On two processes with a sampler of `[[0, 1, 2, 3], [4, 5, 6, 7]]`, this will result in:

            - the sampler on process 0 to yield `[0, 1, 2, 3]` and the sampler on process 1 to yield `[4
              this argument is set to `False`.
            - the sampler on process 0 to yield `[0, 1]` then `[4, 5]` and the sampler on process 1 to y
              then `[6, 7]` if this argument is set to `True`.
        even_batches (`bool`, *optional*, defaults to `True`):
            Whether or not to loop back at the beginning of the sampler when the number of samples is no
            multiple of (original batch size / number of processes).

    <Tip warning={true}>

    `BatchSampler`s with varying batch sizes are not enabled by default. To enable this behaviour, set `
    equal to `False`

    </Tip>"""

    def __init__(
        self,
        batch_sampler: BatchSampler,
        num_processes: int = 1,
        process_index: int = 0,
        split_batches: bool = False,
        even_batches: bool = True,
    ):
        if split_batches and batch_sampler.batch_size % num_processes != 0:
            raise ValueError(
                f"To use `BatchSamplerShard` in `split_batches` mode, the batch size ({batch_sampler.bat
                f"needs to be a round multiple of the number of processes ({num_processes})."
            )
        self.batch_sampler = batch_sampler
        self.num_processes = num_processes
        self.process_index = process_index
        self.split_batches = split_batches
        self.even_batches = even_batches
        self.batch_size = getattr(batch_sampler, "batch_size", None)
        self.drop_last = getattr(batch_sampler, "drop_last", False)
        if self.batch_size is None and self.even_batches:
            raise ValueError("You need to use `even_batches=False` when the batch sampler has no batch s

    @property
    def total_length(self):
        return len(self.batch_sampler)

    def __len__(self):
        if self.split_batches:
            # Split batches does not change the length of the batch sampler
            return len(self.batch_sampler)
        if len(self.batch_sampler) % self.num_processes == 0:
            # If the length is a round multiple of the number of processes, it's easy.
            return len(self.batch_sampler) // self.num_processes
        length = len(self.batch_sampler) // self.num_processes
        if self.drop_last:
            # Same if we drop the remainder.
            return length
        elif self.even_batches:
            # When we even batches we always get +1
            return length + 1
        else:
            # Otherwise it depends on the process index.
            return length + 1 if self.process_index < len(self.batch_sampler) % self.num_processes else

    def __iter__(self):
        return self._iter_with_split() if self.split_batches else self._iter_with_no_split()

    def _iter_with_split(self):
        initial_data = []
        batch_length = self.batch_sampler.batch_size // self.num_processes
        for idx, batch in enumerate(self.batch_sampler):
            if idx == 0:
                initial_data = batch
            if len(batch) == self.batch_size:
                # If the batch is full, we yield the part of it this process is responsible of.
                yield batch[batch_length * self.process_index : batch_length * (self.process_index + 1)]

        # If drop_last is True of the last batch was full, iteration is over, otherwise...
        if not self.drop_last and len(initial_data) > 0 and len(batch) < self.batch_size:
            if not self.even_batches:
                if len(batch) > batch_length * self.process_index:
                    yield batch[batch_length * self.process_index : batch_length * (self.process_index +
            else:
                # For degenerate cases where the dataset has less than num_process * batch_size samples
                while len(initial_data) < self.batch_size:
                    initial_data += initial_data
                batch = batch + initial_data
                yield batch[batch_length * self.process_index : batch_length * (self.process_index + 1)]

    def _iter_with_no_split(self):
        initial_data = []
        batch_to_yield = []
        for idx, batch in enumerate(self.batch_sampler):
            # We gather the initial indices in case we need to circle back at the end.
            if not self.drop_last and idx < self.num_processes:
                initial_data += batch
            # We identify the batch to yield but wait until we ar sure every process gets a full batch b
            # yielding it.
            if idx % self.num_processes == self.process_index:
                batch_to_yield = batch
            if idx % self.num_processes == self.num_processes - 1 and (
                self.batch_size is None or len(batch) == self.batch_size
            ):
                yield batch_to_yield
                batch_to_yield = []
        # If drop_last is True, iteration is over, otherwise...
        if not self.drop_last and len(initial_data) > 0:
            if not self.even_batches:
                if len(batch_to_yield) > 0:
                    yield batch_to_yield
            else:
                # ... we yield the complete batch we had saved before if it has the proper length
                if len(batch_to_yield) == self.batch_size:
                    yield batch_to_yield

                # For degenerate cases where the dataset has less than num_process * batch_size samples
                while len(initial_data) < self.num_processes * self.batch_size:
                    initial_data += initial_data

                # If the last batch seen was of the proper size, it has been yielded by its process so w
                if len(batch) == self.batch_size:
                    batch = []
                    idx += 1

                # Make sure we yield a multiple of self.num_processes batches
                cycle_index = 0
                while idx % self.num_processes != 0 or len(batch) > 0:
                    end_index = cycle_index + self.batch_size - len(batch)
                    batch += initial_data[cycle_index:end_index]
                    if idx % self.num_processes == self.process_index:
                        yield batch
                    cycle_index = end_index
                    batch = []
                    idx += 1


class IterableDatasetShard(IterableDataset):
    """
    Wraps a PyTorch `IterableDataset` to generate samples for one of the processes only. Instances of th
    always yield a number of samples that is a round multiple of the actual batch size (depending of the
    `split_batches`, this is either `batch_size` or `batch_size x num_processes`). Depending on the valu
    `drop_last` attribute of the batch sampler passed, it will either stop the iteration at the first ba
    be too small or loop with indices from the beginning.

    Args:
        dataset (`torch.utils.data.dataset.IterableDataset`):
            The batch sampler to split in several shards.
        batch_size (`int`, *optional*, defaults to 1):
            The size of the batches per shard (if `split_batches=False`) or the size of the batches (if
            `split_batches=True`).
        drop_last (`bool`, *optional*, defaults to `False`):
            Whether or not to drop the last incomplete batch or complete the last batches by using the s
            beginning.
        num_processes (`int`, *optional*, defaults to 1):
            The number of processes running concurrently.
        process_index (`int`, *optional*, defaults to 0):
            The index of the current process.
        split_batches (`bool`, *optional*, defaults to `False`):
            Whether the shards should be created by splitting a batch to give a piece of it on each proc
            yielding different full batches on each process.

            On two processes with an iterable dataset yielding of `[0, 1, 2, 3, 4, 5, 6, 7]`, this will

            - the shard on process 0 to yield `[0, 1, 2, 3]` and the shard on process 1 to yield `[4, 5,
              argument is set to `False`.
            - the shard on process 0 to yield `[0, 1, 4, 5]` and the sampler on process 1 to yield `[2,
              this argument is set to `True`.
    """

    def __init__(
        self,
        dataset: IterableDataset,
        batch_size: int = 1,
        drop_last: bool = False,
        num_processes: int = 1,
        process_index: int = 0,
        split_batches: bool = False,
    ):
        if split_batches and batch_size > 1 and batch_size % num_processes != 0:
            raise ValueError(
                f"To use `IterableDatasetShard` in `split_batches` mode, the batch size ({batch_size}) "
                f"needs to be a round multiple of the number of processes ({num_processes})."
            )
        self.dataset = dataset
        self.batch_size = batch_size
        self.drop_last = drop_last
        self.num_processes = num_processes
        self.process_index = process_index
        self.split_batches = split_batches

    def __iter__(self):
        real_batch_size = self.batch_size if self.split_batches else (self.batch_size * self.num_process
        process_batch_size = (self.batch_size // self.num_processes) if self.split_batches else self.bat
        process_slice = range(self.process_index * process_batch_size, (self.process_index + 1) * proces

        first_batch = None
        current_batch = []
        for element in self.dataset:
            current_batch.append(element)
            # Wait to have a full batch before yielding elements.
            if len(current_batch) == real_batch_size:
                for i in process_slice:
                    yield current_batch[i]
                if first_batch is None:
                    first_batch = current_batch.copy()
                current_batch = []

        # Finished if drop_last is True, otherwise complete the last batch with elements from the beginn
        if not self.drop_last and len(current_batch) > 0:
            if first_batch is None:
                first_batch = current_batch.copy()
            while len(current_batch) < real_batch_size:
                current_batch += first_batch
            for i in process_slice:
                yield current_batch[i]


class DataLoaderShard(DataLoader):
    """
    Subclass of a PyTorch `DataLoader` that will deal with device placement and current distributed setu

    Args:
        dataset (`torch.utils.data.dataset.Dataset`):
            The dataset to use to build this datalaoder.
        device (`torch.device`, *optional*):
            If passed, the device to put all batches on.
        rng_types (list of `str` or [`~utils.RNGType`]):
            The list of random number generators to synchronize at the beginning of each iteration. Shou
            several of:

            - `"torch"`: the base torch random number generator
            - `"cuda"`: the CUDA random number generator (GPU only)
            - `"xla"`: the XLA random number generator (TPU only)
            - `"generator"`: an optional `torch.Generator`
        synchronized_generator (`torch.Generator`, *optional*):
            A random number generator to keep synchronized across processes.
        split_batches (`int`, *optional*, defaults to 0):
            The number of batches to skip at the beginning.
        kwargs:
            All other keyword arguments to pass to the regular `DataLoader` initialization.

    **Available attributes:**

        - **total_batch_size** (`int`) -- Total batch size of the dataloader across all processes.
            Equal to the original batch size when `split_batches=True`; otherwise the original batch siz
            number of processes

        - **total_dataset_length** (`int`) -- Total length of the inner dataset across all processes.
    """
    def __init__(self, dataset, device=None, rng_types=None, synchronized_generator=None, skip_batches=0
        super().__init__(dataset, **kwargs)
        self.device = device
        self.rng_types = rng_types
        self.synchronized_generator = synchronized_generator
        self.skip_batches = skip_batches
        self.gradient_state = GradientState()

    def __iter__(self):
        if self.rng_types is not None:
            synchronize_rng_states(self.rng_types, self.synchronized_generator)
        self.gradient_state._add_dataloader(self)
        # We can safely pass because the default is -1
        with suppress(Exception):
            length = getattr(self.dataset, "total_dataset_length", len(self.dataset))
            self.gradient_state._set_remainder(length % self.total_batch_size)
        dataloader_iter = super().__iter__()
        # We iterate one batch ahead to check when we are at the end
        try:
            current_batch = next(dataloader_iter)
        except StopIteration:
            yield

        batch_index = 0
        while True:
            try:
                # But we still move it to the device so it is done before `StopIteration` is reached
                if self.device is not None:
                    current_batch = send_to_device(current_batch, self.device)
                next_batch = next(dataloader_iter)
                if batch_index >= self.skip_batches:
                    yield current_batch
                batch_index += 1
                current_batch = next_batch
            except StopIteration:
                self.gradient_state._remove_dataloader(self)
                if batch_index >= self.skip_batches:
                    yield current_batch
                break

    @property
    def total_batch_size(self):
        batch_sampler = self.sampler if isinstance(self.sampler, BatchSampler) else self.batch_sampler
        return (
            batch_sampler.batch_size
            if batch_sampler.split_batches
            else (batch_sampler.batch_size * batch_sampler.num_processes)
        )

    @property
    def total_dataset_length(self):
        if hasattr("total_length", self.dataset):
            return self.dataset.total_length
        else:
            return len(self.dataset)


class DataLoaderDispatcher(DataLoader):
    """
    Subclass of a PyTorch `DataLoader` that will iterate and preprocess on process 0 only, then dispatch
    process their part of the batch.

    Args:
        split_batches (`bool`, *optional*, defaults to `False`):
            Whether the resulting `DataLoader` should split the batches of the original data loader acro
            yield full batches (in which case it will yield batches starting at the `process_index`-th a
            `num_processes` batches at each iteration). Another way to see this is that the observed bat
            the same as the initial `dataloader` if this option is set to `True`, the batch size of the
            `dataloader` multiplied by `num_processes` otherwise. Setting this option to `True` requires
            size of the `dataloader` is a round multiple of `batch_size`.
        skip_batches (`int`, *optional*, defaults to 0):
            The number of batches to skip at the beginning of an iteration.
```

```
**Available attributes:**

    - **total_batch_size** (`int`) -- Total batch size of the dataloader across all processes.
        Equal to the original batch size when `split_batches=True`; otherwise the original batch siz
        number of processes

    - **total_dataset_length** (`int`) -- Total length of the inner dataset across all processes.
"""

def __init__(self, dataset, split_batches: bool = False, skip_batches=0, _drop_last: bool = False, *
    shuffle = False
    if is_torch_version(">=", "1.11.0"):
        from torch.utils.data.datapipes.iter.combinatorics import ShufflerIterDataPipe

        # We need to save the shuffling state of the DataPipe
        if isinstance(dataset, ShufflerIterDataPipe):
            shuffle = dataset._shuffle_enabled
    super().__init__(dataset, **kwargs)
    self.split_batches = split_batches
    if is_torch_version("<", "1.8.0"):
        raise ImportError(
            f"Using `DataLoaderDispatcher` requires PyTorch 1.8.0 minimum. You have {torch.__version
        )
    if shuffle:
        torch.utils.data.graph_settings.apply_shuffle_settings(dataset, shuffle=shuffle)

    self.gradient_state = GradientState()
    self.state = AcceleratorState()
    self._drop_last = _drop_last
    self.skip_batches = skip_batches
    # We can safely pass because the default is -1
    with suppress(Exception):
        length = getattr(self.dataset, "total_dataset_length", len(self.dataset))
        self.gradient_state._set_remainder(length % self.total_batch_size)

def _fetch_batches(self, iterator):
    batches, batch = None, None
    # On process 0, we gather the batch to dispatch.
    if self.state.process_index == 0:
        try:
            if self.split_batches:
                # One batch of the main iterator is dispatched and split.
                batch = next(iterator)
            else:
                # num_processes batches of the main iterator are concatenated then dispatched and sp
                # We add the batches one by one so we have the remainder available when drop_last=Fa
                batches = []
                for _ in range(self.state.num_processes):
                    batches.append(next(iterator))
                batch = concatenate(batches, dim=0)
            # In both cases, we need to get the structure of the batch that we will broadcast on oth
            # processes to initialize the tensors with the right shape.
            # data_structure, stop_iteration
            batch_info = [get_data_structure(batch), False]
        except StopIteration:
            batch_info = [None, True]
    else:
        batch_info = [None, self._stop_iteration]
    # This is inplace, so after this instruction, every process has the same `batch_info` as process
    broadcast_object_list(batch_info)
    self._stop_iteration = batch_info[1]
    if self._stop_iteration:
        # If drop_last is False and split_batches is False, we may have a remainder to take care of.
        if not self.split_batches and not self._drop_last:
            if self.state.process_index == 0 and len(batches) > 0:
                batch = concatenate(batches, dim=0)
                batch_info = [get_data_structure(batch), False]
            else:
                batch_info = [None, True]
            broadcast_object_list(batch_info)
    return batch, batch_info
def __iter__(self):
        self.gradient_state._add_dataloader(self)
        main_iterator = None
        if self.state.process_index == 0:
            # We only iterate through the DataLoader on process 0.
            main_iterator = super().__iter__()
        stop_iteration = False
        self._stop_iteration = False
        first_batch = None
        next_batch, next_batch_info = self._fetch_batches(main_iterator)
        batch_index = 0
        while not stop_iteration:
            batch, batch_info = next_batch, next_batch_info

            if self.state.process_index != 0:
                # Initialize tensors on other processes than process 0.
                batch = initialize_tensors(batch_info[0])
            batch = send_to_device(batch, self.state.device)
            # Broadcast the batch before splitting it.
            batch = broadcast(batch, from_process=0)

            if not self._drop_last and first_batch is None:
                # We keep at least num processes elements of the first batch to be able to complete the
                first_batch = slice_tensors(batch, slice(0, self.state.num_processes))

            observed_batch_size = find_batch_size(batch)
            batch_size = observed_batch_size // self.state.num_processes

            stop_iteration = self._stop_iteration
            if not stop_iteration:
                # We may still be at the end of the dataloader without knowing it yet: if there is nothi
                # the dataloader since the number of batches is a round multiple of the number of proces
                next_batch, next_batch_info = self._fetch_batches(main_iterator)
                # next_batch_info[0] is None when there are no more batches, otherwise we still need to
                if self._stop_iteration and next_batch_info[0] is None:
                    stop_iteration = True

            if not self._drop_last and stop_iteration and observed_batch_size % self.state.num_processes
                # If the last batch is not complete, let's add the first batch to it.
                batch = concatenate([batch, first_batch], dim=0)
                # Batch size computation above is wrong, it's off by 1 so we fix it.
                batch_size += 1

            data_slice = slice(self.state.process_index * batch_size, (self.state.process_index + 1) * b
            batch = slice_tensors(batch, data_slice)

            if stop_iteration:
                self.gradient_state._remove_dataloader(self)
                self.gradient_state._set_remainder(observed_batch_size)
            if batch_index >= self.skip_batches:
                yield batch
            batch_index += 1

    def __len__(self):
        whole_length = super().__len__()
        if self.split_batches:
            return whole_length
        elif self._drop_last:
            return whole_length // self.state.num_processes
        else:
            return math.ceil(whole_length / self.state.num_processes)

    @property
    def total_batch_size(self):
        return (
            self.dataset.batch_size if self.split_batches else (self.dataset.batch_size * self.dataset.n
        )

    @property
    def total_dataset_length(self):
        return len(self.dataset)
def prepare_data_loader(
    dataloader: DataLoader,
    device: Optional[torch.device] = None,
    num_processes: Optional[int] = None,
    process_index: Optional[int] = None,
    split_batches: bool = False,
    put_on_device: bool = False,
    rng_types: Optional[List[Union[str, RNGType]]] = None,
    dispatch_batches: Optional[bool] = None,
    even_batches: bool = True,
) -> DataLoader:
    """
    Wraps a PyTorch `DataLoader` to generate batches for one of the processes only.

    Depending on the value of the `drop_last` attribute of the `dataloader` passed, it will either stop
    at the first batch that would be too small / not present on all processes or loop with indices from

    Args:
        dataloader (`torch.utils.data.dataloader.DataLoader`):
            The data loader to split across several devices.
        device (`torch.device`):
            The target device for the returned `DataLoader`.
        num_processes (`int`, *optional*):
            The number of processes running concurrently. Will default to the value given by
            [`~state.AcceleratorState`].
        process_index (`int`, *optional*):
            The index of the current process. Will default to the value given by [`~state.AcceleratorSta
        split_batches (`bool`, *optional*, defaults to `False`):
            Whether the resulting `DataLoader` should split the batches of the original data loader acro
            yield full batches (in which case it will yield batches starting at the `process_index`-th a
            `num_processes` batches at each iteration).

            Another way to see this is that the observed batch size will be the same as the initial `dat
            this option is set to `True`, the batch size of the initial `dataloader` multiplied by `num_
            otherwise.

            Setting this option to `True` requires that the batch size of the `dataloader` is a round mu
            `batch_size`.
        put_on_device (`bool`, *optional*, defaults to `False`):
            Whether or not to put the batches on `device` (only works if the batches are nested list, tu
            dictionaries of tensors).
        rng_types (list of `str` or [`~utils.RNGType`]):
            The list of random number generators to synchronize at the beginning of each iteration. Shou
            several of:

            - `"torch"`: the base torch random number generator
            - `"cuda"`: the CUDA random number generator (GPU only)
            - `"xla"`: the XLA random number generator (TPU only)
            - `"generator"`: the `torch.Generator` of the sampler (or batch sampler if there is no sampl
              dataloader) or of the iterable dataset (if it exists) if the underlying dataset is of that

        dispatch_batches (`bool`, *optional*):
            If set to `True`, the datalaoder prepared is only iterated through on the main process and t
            are split and broadcast to each process. Will default to `True` when the underlying dataset
            `IterableDataset`, `False` otherwise.
        even_batches (`bool`, *optional*, defaults to `True`):
            If set to `True`, in cases where the total batch size across all processes does not exactly
            dataset, samples at the start of the dataset will be duplicated so the batch can be divided
            all workers.

    Returns:
        `torch.utils.data.dataloader.DataLoader`: A new data loader that will yield the portion of the b

    <Tip warning={true}>

    `BatchSampler`s with varying batch sizes are not enabled by default. To enable this behaviour, set `
    equal to `False`

    </Tip>
    """
    if dispatch_batches is None:
        if is_torch_version("<", "1.8.0") or not put_on_device:
            dispatch_batches = False
        else:
            dispatch_batches = isinstance(dataloader.dataset, IterableDataset)

    if dispatch_batches and not put_on_device:
        raise ValueError("Using `dispatch_batches=True` requires `put_on_device=True`.")
    # Grab defaults from AcceleratorState
    state = AcceleratorState()
    if num_processes is None:
        num_processes = state.num_processes
    if process_index is None:
        process_index = state.process_index

    # Sanity check
    if split_batches and dataloader.batch_size > 1 and dataloader.batch_size % num_processes != 0:
        raise ValueError(
            f"To use a `DataLoader` in `split_batches` mode, the batch size ({dataloader.batch_size}) "
            f"needs to be a round multiple of the number of processes ({num_processes})."
        )

    new_dataset = dataloader.dataset
    # Iterable dataset doesn't like batch_sampler, but data_loader creates a default one for it
    new_batch_sampler = dataloader.batch_sampler if not isinstance(new_dataset, IterableDataset) else No
    sampler_is_batch_sampler = False
    synchronized_generator = None
    # No change if no multiprocess
    if (num_processes != 1 or state.distributed_type == DistributedType.MEGATRON_LM) and not dispatch_ba
        if isinstance(new_dataset, IterableDataset):
            if getattr(dataloader.dataset, "generator", None) is not None:
                synchronized_generator = dataloader.dataset.generator
            new_dataset = IterableDatasetShard(
                new_dataset,
                batch_size=dataloader.batch_size,
                drop_last=dataloader.drop_last,
                num_processes=num_processes,
                process_index=process_index,
                split_batches=split_batches,
            )
        else:
            # New batch sampler for the current process.
            sampler_is_batch_sampler = isinstance(dataloader.sampler, BatchSampler)
            if sampler_is_batch_sampler:
                sampler = dataloader.sampler.sampler
            else:
                sampler = dataloader.batch_sampler.sampler
            if hasattr(sampler, "generator"):
                if sampler.generator is None:
                    sampler.generator = torch.Generator()
                synchronized_generator = sampler.generator

            batch_sampler = dataloader.sampler if sampler_is_batch_sampler else dataloader.batch_sampler
            new_batch_sampler = BatchSamplerShard(
                batch_sampler,
                num_processes=num_processes,
                process_index=process_index,
                split_batches=split_batches,
                even_batches=even_batches,
            )

    # We ignore all of those since they are all dealt with by our new_batch_sampler
    ignore_kwargs = [
        "batch_size",
        "shuffle",
        "sampler",
        "batch_sampler",
        "drop_last",
    ]

    if rng_types is not None and synchronized_generator is None and "generator" in rng_types:
        rng_types.remove("generator")

    kwargs = {
        k: getattr(dataloader, k, _PYTORCH_DATALOADER_KWARGS[k])
        for k in _PYTORCH_DATALOADER_KWARGS
```

```
            if k not in ignore_kwargs
        }

        # Need to provide batch_size as batch_sampler is None for Iterable dataset
        if new_batch_sampler is None:
            kwargs["drop_last"] = dataloader.drop_last
            kwargs["batch_size"] = (
                dataloader.batch_size // num_processes if split_batches and not dispatch_batches else datalo
            )

        if dispatch_batches:
            kwargs.pop("generator")
            dataloader = DataLoaderDispatcher(
                new_dataset,
                split_batches=split_batches,
                batch_sampler=new_batch_sampler,
                _drop_last=dataloader.drop_last,
                **kwargs,
            )
        elif sampler_is_batch_sampler:
            dataloader = DataLoaderShard(
                new_dataset,
                device=device if put_on_device and state.distributed_type != DistributedType.TPU else None,
                sampler=new_batch_sampler,
                batch_size=dataloader.batch_size,
                rng_types=rng_types,
                synchronized_generator=synchronized_generator,
                **kwargs,
            )
        else:
            dataloader = DataLoaderShard(
                new_dataset,
                device=device if put_on_device and state.distributed_type != DistributedType.TPU else None,
                batch_sampler=new_batch_sampler,
                rng_types=rng_types,
                synchronized_generator=synchronized_generator,
                **kwargs,
            )

        if state.distributed_type == DistributedType.TPU:
            return MpDeviceLoaderWrapper(dataloader, device)
        return dataloader


    class SkipBatchSampler(BatchSampler):
        """
        A `torch.utils.data.BatchSampler` that skips the first `n` batches of another `torch.utils.data.Batc
        """

        def __init__(self, batch_sampler, skip_batches=0):
            self.batch_sampler = batch_sampler
            self.skip_batches = skip_batches

        def __iter__(self):
            for index, samples in enumerate(self.batch_sampler):
                if index >= self.skip_batches:
                    yield samples

        @property
        def total_length(self):
            return len(self.batch_sampler)

        def __len__(self):
            return len(self.batch_sampler) - self.skip_batches


    class SkipDataLoader(DataLoader):
        """
        Subclass of a PyTorch `DataLoader` that will skip the first batches.

        Args:
            dataset (`torch.utils.data.dataset.Dataset`):
```

```
            The dataset to use to build this datalaoder.
        skip_batches (`int`, *optional*, defaults to 0):
            The number of batches to skip at the beginning.
        kwargs:
            All other keyword arguments to pass to the regular `DataLoader` initialization.
    """

    def __init__(self, dataset, skip_batches=0, **kwargs):
        super().__init__(dataset, **kwargs)
        self.skip_batches = skip_batches

    def __iter__(self):
        for index, batch in enumerate(super().__iter__()):
            if index >= self.skip_batches:
                yield batch


def skip_first_batches(dataloader, num_batches=0):
    """
    Creates a `torch.utils.data.DataLoader` that will efficiently skip the first `num_batches`.
    """
    dataset = dataloader.dataset
    sampler_is_batch_sampler = False
    if isinstance(dataset, IterableDataset):
        new_batch_sampler = None
    else:
        sampler_is_batch_sampler = isinstance(dataloader.sampler, BatchSampler)
        batch_sampler = dataloader.sampler if sampler_is_batch_sampler else dataloader.batch_sampler
        new_batch_sampler = SkipBatchSampler(batch_sampler, skip_batches=num_batches)

    # We ignore all of those since they are all dealt with by our new_batch_sampler
    ignore_kwargs = [
        "batch_size",
        "shuffle",
        "sampler",
        "batch_sampler",
        "drop_last",
    ]

    kwargs = {
        k: getattr(dataloader, k, _PYTORCH_DATALOADER_KWARGS[k])
        for k in _PYTORCH_DATALOADER_KWARGS
        if k not in ignore_kwargs
    }

    # Need to provide batch_size as batch_sampler is None for Iterable dataset
    if new_batch_sampler is None:
        kwargs["drop_last"] = dataloader.drop_last
        kwargs["batch_size"] = dataloader.batch_size

    if isinstance(dataloader, DataLoaderDispatcher):
        if new_batch_sampler is None:
            # Need to manually skip batches in the dataloader
            kwargs["skip_batches"] = num_batches
        dataloader = DataLoaderDispatcher(
            dataset,
            split_batches=dataloader.split_batches,
            batch_sampler=new_batch_sampler,
            _drop_last=dataloader._drop_last,
            **kwargs,
        )
    elif isinstance(dataloader, DataLoaderShard):
        if new_batch_sampler is None:
            # Need to manually skip batches in the dataloader
            kwargs["skip_batches"] = num_batches
        elif sampler_is_batch_sampler:
            kwargs["sampler"] = new_batch_sampler
            kwargs["batch_size"] = dataloader.batch_size
        else:
            kwargs["batch_sampler"] = new_batch_sampler
        dataloader = DataLoaderShard(
            dataset,
            device=dataloader.device,
            rng_types=dataloader.rng_types,
            synchronized_generator=dataloader.synchronized_generator,
            **kwargs,
        )
    else:
        if new_batch_sampler is None:
            # Need to manually skip batches in the dataloader
            dataloader = SkipDataLoader(dataset, skip_batches=num_batches, **kwargs)
        else:
            dataloader = DataLoader(dataset, batch_sampler=new_batch_sampler, **kwargs)

    return dataloader
```

accelerate-main/src/accelerate/commands/config/__init__.py

```
#!/usr/bin/env python

# Copyright 2021 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import argparse

from .config import config_command_parser
from .config_args import default_config_file, load_config_from_file  # noqa: F401
from .default import default_command_parser
from .update import update_command_parser


def get_config_parser(subparsers=None):
    parent_parser = argparse.ArgumentParser(add_help=False, allow_abbrev=False)
    # The main config parser
    config_parser = config_command_parser(subparsers)
    # The subparser to add commands to
    subcommands = config_parser.add_subparsers(title="subcommands", dest="subcommand")

    # Then add other parsers with the parent parser
    default_command_parser(subcommands, parents=[parent_parser])
    update_command_parser(subcommands, parents=[parent_parser])

    return config_parser


def main():
    config_parser = get_config_parser()
    args = config_parser.parse_args()

    if not hasattr(args, "func"):
        config_parser.print_help()
        exit(1)

    # Run
    args.func(args)


if __name__ == "__main__":
    main()
```

accelerate-main/docs/source/package_reference/accelerator.mdx

# Accelerator

The [`Accelerator`] is the main class provided by ■ Accelerate.
It serves at the main entrypoint for the API.

## Quick adaptation of your code

To quickly adapt your script to work on any kind of setup with ■ Accelerate just:

1. Initialize an [`Accelerator`] object (that we will call `accelerator` throughout this page) as early
2. Pass your dataloader(s), model(s), optimizer(s), and scheduler(s) to the [`~Accelerator.prepare`] met
3. Remove all the `.cuda()` or `.to(device)` from your code and let the `accelerator` handle the device

<Tip>

    Step three is optional, but considered a best practice.

</Tip>

4. Replace `loss.backward()` in your code with `accelerator.backward(loss)`
5. Gather your predictions and labels before storing them or using them for metric computation using [`~

<Tip warning={true}>

    Step five is mandatory when using distributed evaluation

</Tip>

In most cases this is all that is needed. The next section lists a few more advanced use cases and nice
you should search for and replace by the corresponding methods of your `accelerator`:

## Advanced recommendations

### Printing

`print` statements should be replaced by [`~Accelerator.print`] to be printed once per process

```diff
- print("My thing I want to print!")
+ accelerator.print("My thing I want to print!")
```

### Executing processes

#### Once on a single server

For statements that should be executed once per server, use [`~Accelerator.is_local_main_process`]:

```python
if accelerator.is_local_main_process:
    do_thing_once_per_server()
```

A function can be wrapped using the [`~Accelerator.on_local_main_process`] function to achieve the same
behavior on a function's execution:
```python
```

~~~
@accelerator.on_local_main_process
def do_my_thing():
    "Something done once per server"
    do_thing_once_per_server()
```

#### Only ever once across all servers

For statements that should only ever be executed once, use [`~Accelerator.is_main_process`]:

```python
if accelerator.is_main_process:
    do_thing_once()
```

A function can be wrapped using the [`~Accelerator.on_main_process`] function to achieve the same
behavior on a function's execution:

```python
@accelerator.on_main_process
def do_my_thing():
    "Something done once per server"
    do_thing_once()
```

#### On specific processes

If a function should be ran on a specific overall or local process index, there are similar decorators
to achieve this:

```python
@accelerator.on_local_process(local_process_idx=0)
def do_my_thing():
    "Something done on process index 0 on each server"
    do_thing_on_index_zero_on_each_server()
```

```python
@accelerator.on_process(process_index=0)
def do_my_thing():
    "Something done on process index 0"
    do_thing_on_index_zero()
```

### Synchronicity control

Use [`~Accelerator.wait_for_everyone`] to make sure all processes join that point before continuing. (Us

### Saving and loading

Use [`~Accelerator.unwrap_model`] before saving to remove all special model wrappers added during the di

```python
model = MyModel()
model = accelerator.prepare(model)
# Unwrap
model = accelerator.unwrap_model(model)
```

Use [`~Accelerator.save`] instead of `torch.save`:

```diff
  state_dict = model.state_dict()
- torch.save(state_dict, "my_state.pkl")
+ accelerator.save(state_dict, "my_state.pkl")
```

### Operations

Use [`~Accelerator.clip_grad_norm_`] instead of ``torch.nn.utils.clip_grad_norm_`` and [`~Accelerator.cl

### Gradient Accumulation
~~~

To perform gradient accumulation use [`~Accelerator.accumulate`] and specify a gradient_accumulation_ste
This will also automatically ensure the gradients are synced or unsynced when on
multi-device training, check if the step should actually be performed, and auto-scale the loss:

```diff
- accelerator = Accelerator()
+ accelerator = Accelerator(gradient_accumulation_steps=2)

  for (input, label) in training_dataloader:
+     with accelerator.accumulate(model):
          predictions = model(input)
          loss = loss_function(predictions, labels)
          accelerator.backward(loss)
          optimizer.step()
          scheduler.step()
          optimizer.zero_grad()
```

## Overall API documentation:

[[autodoc]] Accelerator

accelerate-main/examples/by_feature/tracking.py

```
# coding=utf-8
# Copyright 2021 The HuggingFace Inc. team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
import argparse
import os

import evaluate
import torch
from datasets import load_dataset
from torch.optim import AdamW
from torch.utils.data import DataLoader
from transformers import AutoModelForSequenceClassification, AutoTokenizer, get_linear_schedule_with_war

from accelerate import Accelerator, DistributedType


########################################################################
# This is a fully working simple example to use Accelerate,
# specifically showcasing the experiment tracking capability,
# and builds off the `nlp_example.py` script.
#
# This example trains a Bert base model on GLUE MRPC
# in any of the following settings (with the same script):
#   - single CPU or single GPU
#   - multi GPUS (using PyTorch distributed mode)
#   - (multi) TPUs
#   - fp16 (mixed-precision) or fp32 (normal precision)
#
# To help focus on the differences in the code, building `DataLoaders`
# was refactored into its own function.
# New additions from the base script can be found quickly by
# looking for the # New Code # tags
#
# To run it in each of these various modes, follow the instructions
# in the readme for examples:
# https://github.com/huggingface/accelerate/tree/main/examples
#
########################################################################

MAX_GPU_BATCH_SIZE = 16
EVAL_BATCH_SIZE = 32


def get_dataloaders(accelerator: Accelerator, batch_size: int = 16):
    """
    Creates a set of `DataLoader`s for the `glue` dataset,
    using "bert-base-cased" as the tokenizer.

    Args:
        accelerator (`Accelerator`):
            An `Accelerator` object
        batch_size (`int`, *optional*):
            The batch size for the train and validation DataLoaders.
    """
    tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
    datasets = load_dataset("glue", "mrpc")
    def tokenize_function(examples):
        # max_length=None => use the model max length (it's actually the default)
        outputs = tokenizer(examples["sentence1"], examples["sentence2"], truncation=True, max_length=No
        return outputs

    # Apply the method we just defined to all the examples in all the splits of the dataset
    # starting with the main process first:
    with accelerator.main_process_first():
        tokenized_datasets = datasets.map(
            tokenize_function,
            batched=True,
            remove_columns=["idx", "sentence1", "sentence2"],
        )

    # We also rename the 'label' column to 'labels' which is the expected name for labels by the models
    # transformers library
    tokenized_datasets = tokenized_datasets.rename_column("label", "labels")

    def collate_fn(examples):
        # On TPU it's best to pad everything to the same length or training will be very slow.
        max_length = 128 if accelerator.distributed_type == DistributedType.TPU else None
        # When using mixed precision we want round multiples of 8/16
        if accelerator.mixed_precision == "fp8":
            pad_to_multiple_of = 16
        elif accelerator.mixed_precision != "no":
            pad_to_multiple_of = 8
        else:
            pad_to_multiple_of = None

        return tokenizer.pad(
            examples,
            padding="longest",
            max_length=max_length,
            pad_to_multiple_of=pad_to_multiple_of,
            return_tensors="pt",
        )

    # Instantiate dataloaders.
    train_dataloader = DataLoader(
        tokenized_datasets["train"], shuffle=True, collate_fn=collate_fn, batch_size=batch_size
    )
    eval_dataloader = DataLoader(
        tokenized_datasets["validation"], shuffle=False, collate_fn=collate_fn, batch_size=EVAL_BATCH_SI
    )

    return train_dataloader, eval_dataloader


# For testing only
if os.environ.get("TESTING_MOCKED_DATALOADERS", None) == "1":
    from accelerate.test_utils.training import mocked_dataloaders

    get_dataloaders = mocked_dataloaders  # noqa: F811


def training_function(config, args):
    # For testing only
    if os.environ.get("TESTING_MOCKED_DATALOADERS", None) == "1":
        config["num_epochs"] = 2
    # Initialize Accelerator

    # New Code #
    # We pass in "all" to `log_with` to grab all available trackers in the environment
    # Note: If using a custom `Tracker` class, should be passed in here such as:
    # >>> log_with = ["all", MyCustomTrackerClassInstance()]
    if args.with_tracking:
        accelerator = Accelerator(
            cpu=args.cpu, mixed_precision=args.mixed_precision, log_with="all", logging_dir=args.logging
        )
    else:
        accelerator = Accelerator(cpu=args.cpu, mixed_precision=args.mixed_precision)
    # Sample hyper-parameters for learning rate, batch size, seed and a few other HPs
    lr = config["lr"]
```

```
num_epochs = int(config["num_epochs"])
seed = int(config["seed"])
batch_size = int(config["batch_size"])
set_seed(seed)

train_dataloader, eval_dataloader = get_dataloaders(accelerator, batch_size)
metric = evaluate.load("glue", "mrpc")

# If the batch size is too big we use gradient accumulation
gradient_accumulation_steps = 1
if batch_size > MAX_GPU_BATCH_SIZE and accelerator.distributed_type != DistributedType.TPU:
    gradient_accumulation_steps = batch_size // MAX_GPU_BATCH_SIZE
    batch_size = MAX_GPU_BATCH_SIZE

# Instantiate the model (we build the model here so that the seed also control new weights initializ
model = AutoModelForSequenceClassification.from_pretrained("bert-base-cased", return_dict=True)

# We could avoid this line since the accelerator is set with `device_placement=True` (default value)
# Note that if you are placing tensors on devices manually, this line absolutely needs to be before
# creation otherwise training will not work on TPU (`accelerate` will kindly throw an error to make
model = model.to(accelerator.device)

# Instantiate optimizer
optimizer = AdamW(params=model.parameters(), lr=lr)

# Instantiate scheduler
lr_scheduler = get_linear_schedule_with_warmup(
    optimizer=optimizer,
    num_warmup_steps=100,
    num_training_steps=(len(train_dataloader) * num_epochs) // gradient_accumulation_steps,
)

# Prepare everything
# There is no specific order to remember, we just need to unpack the objects in the same order we ga
# prepare method.
model, optimizer, train_dataloader, eval_dataloader, lr_scheduler = accelerator.prepare(
    model, optimizer, train_dataloader, eval_dataloader, lr_scheduler
)

# New Code #
# We need to initialize the trackers we use. Overall configurations can also be stored
if args.with_tracking:
    run = os.path.split(__file__)[-1].split(".")[0]
    accelerator.init_trackers(run, config)

# Now we train the model
for epoch in range(num_epochs):
    model.train()
    # New Code #
    # For our tracking example, we will log the total loss of each epoch
    if args.with_tracking:
        total_loss = 0
    for step, batch in enumerate(train_dataloader):
        # We could avoid this line since we set the accelerator with `device_placement=True`.
        batch.to(accelerator.device)
        outputs = model(**batch)
        loss = outputs.loss
        # New Code #
        if args.with_tracking:
            total_loss += loss.detach().float()
        loss = loss / gradient_accumulation_steps
        accelerator.backward(loss)
        if step % gradient_accumulation_steps == 0:
            optimizer.step()
            lr_scheduler.step()
            optimizer.zero_grad()

    model.eval()
    for step, batch in enumerate(eval_dataloader):
        # We could avoid this line since we set the accelerator with `device_placement=True` (the de
        batch.to(accelerator.device)
        with torch.no_grad():
                outputs = model(**batch)
            predictions = outputs.logits.argmax(dim=-1)
            predictions, references = accelerator.gather_for_metrics((predictions, batch["labels"]))
            metric.add_batch(
                predictions=predictions,
                references=references,
            )

        eval_metric = metric.compute()
        # Use accelerator.print to print only on the main process.
        accelerator.print(f"epoch {epoch}:", eval_metric)

        # New Code #
        # To actually log, we call `Accelerator.log`
        # The values passed can be of `str`, `int`, `float` or `dict` of `str` to `float`/`int`
        if args.with_tracking:
            accelerator.log(
                {
                    "accuracy": eval_metric["accuracy"],
                    "f1": eval_metric["f1"],
                    "train_loss": total_loss.item() / len(train_dataloader),
                    "epoch": epoch,
                },
                step=epoch,
            )

    # New Code #
    # When a run is finished, you should call `accelerator.end_training()`
    # to close all of the open trackers
    if args.with_tracking:
        accelerator.end_training()


def main():
    parser = argparse.ArgumentParser(description="Simple example of training script.")
    parser.add_argument(
        "--mixed_precision",
        type=str,
        default=None,
        choices=["no", "fp16", "bf16", "fp8"],
        help="Whether to use mixed precision. Choose"
        "between fp16 and bf16 (bfloat16). Bf16 requires PyTorch >= 1.10."
        "and an Nvidia Ampere GPU.",
    )
    parser.add_argument("--cpu", action="store_true", help="If passed, will train on the CPU.")
    parser.add_argument(
        "--with_tracking",
        action="store_true",
        help="Whether to load in all available experiment trackers from the environment and use them for
    )
    parser.add_argument(
        "--logging_dir",
        type=str,
        default="logs",
        help="Location on where to store experiment tracking logs`",
    )
    args = parser.parse_args()
    config = {"lr": 2e-5, "num_epochs": 3, "seed": 42, "batch_size": 16}
    training_function(config, args)


if __name__ == "__main__":
    main()
```

accelerate-main/examples/by_feature/checkpointing.py

```
# coding=utf-8
# Copyright 2021 The HuggingFace Inc. team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
import argparse
import os

import evaluate
import torch
from datasets import load_dataset
from torch.optim import AdamW
from torch.utils.data import DataLoader
from transformers import AutoModelForSequenceClassification, AutoTokenizer, get_linear_schedule_with_war

from accelerate import Accelerator, DistributedType


########################################################################
# This is a fully working simple example to use Accelerate,
# specifically showcasing the checkpointing capability,
# and builds off the `nlp_example.py` script.
#
# This example trains a Bert base model on GLUE MRPC
# in any of the following settings (with the same script):
#   - single CPU or single GPU
#   - multi GPUS (using PyTorch distributed mode)
#   - (multi) TPUs
#   - fp16 (mixed-precision) or fp32 (normal precision)
#
# To help focus on the differences in the code, building `DataLoaders`
# was refactored into its own function.
# New additions from the base script can be found quickly by
# looking for the # New Code # tags
#
# To run it in each of these various modes, follow the instructions
# in the readme for examples:
# https://github.com/huggingface/accelerate/tree/main/examples
#
########################################################################

MAX_GPU_BATCH_SIZE = 16
EVAL_BATCH_SIZE = 32


def get_dataloaders(accelerator: Accelerator, batch_size: int = 16):
    """
    Creates a set of `DataLoader`s for the `glue` dataset,
    using "bert-base-cased" as the tokenizer.

    Args:
        accelerator (`Accelerator`):
            An `Accelerator` object
        batch_size (`int`, *optional*):
            The batch size for the train and validation DataLoaders.
    """
    tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
    datasets = load_dataset("glue", "mrpc")
    def tokenize_function(examples):
        # max_length=None => use the model max length (it's actually the default)
        outputs = tokenizer(examples["sentence1"], examples["sentence2"], truncation=True, max_length=No
        return outputs

    # Apply the method we just defined to all the examples in all the splits of the dataset
    # starting with the main process first:
    with accelerator.main_process_first():
        tokenized_datasets = datasets.map(
            tokenize_function,
            batched=True,
            remove_columns=["idx", "sentence1", "sentence2"],
        )

    # We also rename the 'label' column to 'labels' which is the expected name for labels by the models
    # transformers library
    tokenized_datasets = tokenized_datasets.rename_column("label", "labels")

    def collate_fn(examples):
        # On TPU it's best to pad everything to the same length or training will be very slow.
        max_length = 128 if accelerator.distributed_type == DistributedType.TPU else None
        # When using mixed precision we want round multiples of 8/16
        if accelerator.mixed_precision == "fp8":
            pad_to_multiple_of = 16
        elif accelerator.mixed_precision != "no":
            pad_to_multiple_of = 8
        else:
            pad_to_multiple_of = None

        return tokenizer.pad(
            examples,
            padding="longest",
            max_length=max_length,
            pad_to_multiple_of=pad_to_multiple_of,
            return_tensors="pt",
        )

    # Instantiate dataloaders.
    train_dataloader = DataLoader(
        tokenized_datasets["train"], shuffle=True, collate_fn=collate_fn, batch_size=batch_size
    )
    eval_dataloader = DataLoader(
        tokenized_datasets["validation"], shuffle=False, collate_fn=collate_fn, batch_size=EVAL_BATCH_SI
    )

    return train_dataloader, eval_dataloader


# For testing only
if os.environ.get("TESTING_MOCKED_DATALOADERS", None) == "1":
    from accelerate.test_utils.training import mocked_dataloaders

    get_dataloaders = mocked_dataloaders  # noqa: F811


def training_function(config, args):
    # For testing only
    if os.environ.get("TESTING_MOCKED_DATALOADERS", None) == "1":
        config["num_epochs"] = 2
    # Initialize accelerator
    accelerator = Accelerator(cpu=args.cpu, mixed_precision=args.mixed_precision)
    # Sample hyper-parameters for learning rate, batch size, seed and a few other HPs
    lr = config["lr"]
    num_epochs = int(config["num_epochs"])
    seed = int(config["seed"])
    batch_size = int(config["batch_size"])

    # New Code #
    # Parse out whether we are saving every epoch or after a certain number of batches
    if hasattr(args.checkpointing_steps, "isdigit"):
        if args.checkpointing_steps == "epoch":
            checkpointing_steps = args.checkpointing_steps
        elif args.checkpointing_steps.isdigit():
```

```
        checkpointing_steps = int(args.checkpointing_steps)
    else:
        raise ValueError(
            f"Argument `checkpointing_steps` must be either a number or `epoch`. `{args.checkpointin
        )
else:
    checkpointing_steps = None

set_seed(seed)

train_dataloader, eval_dataloader = get_dataloaders(accelerator, batch_size)
metric = evaluate.load("glue", "mrpc")

# If the batch size is too big we use gradient accumulation
gradient_accumulation_steps = 1
if batch_size > MAX_GPU_BATCH_SIZE and accelerator.distributed_type != DistributedType.TPU:
    gradient_accumulation_steps = batch_size // MAX_GPU_BATCH_SIZE
    batch_size = MAX_GPU_BATCH_SIZE

# Instantiate the model (we build the model here so that the seed also control new weights initializ
model = AutoModelForSequenceClassification.from_pretrained("bert-base-cased", return_dict=True)

# We could avoid this line since the accelerator is set with `device_placement=True` (default value)
# Note that if you are placing tensors on devices manually, this line absolutely needs to be before
# creation otherwise training will not work on TPU (`accelerate` will kindly throw an error to make
model = model.to(accelerator.device)

# Instantiate optimizer
optimizer = AdamW(params=model.parameters(), lr=lr)

# Instantiate scheduler
lr_scheduler = get_linear_schedule_with_warmup(
    optimizer=optimizer,
    num_warmup_steps=100,
    num_training_steps=(len(train_dataloader) * num_epochs) // gradient_accumulation_steps,
)

# Prepare everything
# There is no specific order to remember, we just need to unpack the objects in the same order we ga
# prepare method.
model, optimizer, train_dataloader, eval_dataloader, lr_scheduler = accelerator.prepare(
    model, optimizer, train_dataloader, eval_dataloader, lr_scheduler
)

# New Code #
# We need to keep track of how many total steps we have iterated over
overall_step = 0
# We also need to keep track of the stating epoch so files are named properly
starting_epoch = 0

# We need to load the checkpoint back in before training here with `load_state`
# The total number of epochs is adjusted based on where the state is being loaded from,
# as we assume continuation of the same training script
if args.resume_from_checkpoint:
    if args.resume_from_checkpoint is not None or args.resume_from_checkpoint != "":
        accelerator.print(f"Resumed from checkpoint: {args.resume_from_checkpoint}")
        accelerator.load_state(args.resume_from_checkpoint)
        path = os.path.basename(args.resume_from_checkpoint)
    else:
        # Get the most recent checkpoint
        dirs = [f.name for f in os.scandir(os.getcwd()) if f.is_dir()]
        dirs.sort(key=os.path.getctime)
        path = dirs[-1]  # Sorts folders by date modified, most recent checkpoint is the last
    # Extract `epoch_{i}` or `step_{i}`
    training_difference = os.path.splitext(path)[0]

    if "epoch" in training_difference:
        starting_epoch = int(training_difference.replace("epoch_", "")) + 1
        resume_step = None
    else:
        resume_step = int(training_difference.replace("step_", ""))
        starting_epoch = resume_step // len(train_dataloader)
```

```
            resume_step -= starting_epoch * len(train_dataloader)

    # Now we train the model
    for epoch in range(starting_epoch, num_epochs):
        model.train()
        # New Code #
        if args.resume_from_checkpoint and epoch == starting_epoch and resume_step is not None:
            # We need to skip steps until we reach the resumed step
            train_dataloader = accelerator.skip_first_batches(train_dataloader, resume_step)
            overall_step += resume_step
        for step, batch in enumerate(train_dataloader):
            # We could avoid this line since we set the accelerator with `device_placement=True`.
            batch.to(accelerator.device)
            outputs = model(**batch)
            loss = outputs.loss
            loss = loss / gradient_accumulation_steps
            accelerator.backward(loss)
            if step % gradient_accumulation_steps == 0:
                optimizer.step()
                lr_scheduler.step()
                optimizer.zero_grad()
            # New Code #
            overall_step += 1

            # New Code #
            # We save the model, optimizer, lr_scheduler, and seed states by calling `save_state`
            # These are saved to folders named `step_{overall_step}`
            # Will contain files: "pytorch_model.bin", "optimizer.bin", "scheduler.bin", and "random_sta
            # If mixed precision was used, will also save a "scalar.bin" file
            if isinstance(checkpointing_steps, int):
                output_dir = f"step_{overall_step}"
                if overall_step % checkpointing_steps == 0:
                    if args.output_dir is not None:
                        output_dir = os.path.join(args.output_dir, output_dir)
                    accelerator.save_state(output_dir)

        model.eval()
        for step, batch in enumerate(eval_dataloader):
            # We could avoid this line since we set the accelerator with `device_placement=True` (the de
            batch.to(accelerator.device)
            with torch.no_grad():
                outputs = model(**batch)
            predictions = outputs.logits.argmax(dim=-1)
            predictions, references = accelerator.gather_for_metrics((predictions, batch["labels"]))
            metric.add_batch(
                predictions=predictions,
                references=references,
            )

        eval_metric = metric.compute()
        # Use accelerator.print to print only on the main process.
        accelerator.print(f"epoch {epoch}:", eval_metric)

        # New Code #
        # We save the model, optimizer, lr_scheduler, and seed states by calling `save_state`
        # These are saved to folders named `epoch_{epoch}`
        # Will contain files: "pytorch_model.bin", "optimizer.bin", "scheduler.bin", and "random_states.
        # If mixed precision was used, will also save a "scalar.bin" file
        if checkpointing_steps == "epoch":
            output_dir = f"epoch_{epoch}"
            if args.output_dir is not None:
                output_dir = os.path.join(args.output_dir, output_dir)
            accelerator.save_state(output_dir)


def main():
    parser = argparse.ArgumentParser(description="Simple example of training script.")
    parser.add_argument(
        "--mixed_precision",
        type=str,
        default=None,
        choices=["no", "fp16", "bf16", "fp8"],
        help="Whether to use mixed precision. Choose"
        "between fp16 and bf16 (bfloat16). Bf16 requires PyTorch >= 1.10."
        "and an Nvidia Ampere GPU.",
    )
    parser.add_argument("--cpu", action="store_true", help="If passed, will train on the CPU.")
    parser.add_argument(
        "--checkpointing_steps",
        type=str,
        default=None,
        help="Whether the various states should be saved at the end of every n steps, or 'epoch' for eac
    )
    parser.add_argument(
        "--output_dir",
        type=str,
        default=".",
        help="Optional save directory where all checkpoint folders will be stored. Default is the curren
    )
    parser.add_argument(
        "--resume_from_checkpoint",
        type=str,
        default=None,
        help="If the training should continue from a checkpoint folder.",
    )
    args = parser.parse_args()
    config = {"lr": 2e-5, "num_epochs": 3, "seed": 42, "batch_size": 16}
    training_function(config, args)


if __name__ == "__main__":
    main()
```

accelerate-main/src/accelerate/scheduler.py

```
# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

# We ignore warnings about stepping the scheduler since we step it ourselves during gradient accumulatio

import warnings

from .state import AcceleratorState, GradientState


warnings.filterwarnings("ignore", category=UserWarning, module="torch.optim.lr_scheduler")


class AcceleratedScheduler:
    """
    A wrapper around a learning rate scheduler that will only step when the optimizer(s) have a training
    to avoid making a scheduler step too fast when gradients went overflow and there was no training ste
    precision training)

    When performing gradient accumulation scheduler lengths should not be changed accordingly, Accelerat
    step the scheduler to account for it.

    Args:
        scheduler (`torch.optim.lr_scheduler._LRScheduler`):
            The scheduler to wrap.
        optimizers (one or a list of `torch.optim.Optimizer`):
            The optimizers used.
        step_with_optimizer (`bool`, *optional*, defaults to `True`):
            Whether or not the scheduler should be stepped at each optimizer step.
        split_batches (`bool`, *optional*, defaults to `False`):
            Whether or not the dataloaders split one batch across the different processes (so batch size
            regardless of the number of processes) or create batches on each process (so batch size is t
            batch size multiplied by the number of processes).
    """

    def __init__(self, scheduler, optimizers, step_with_optimizer: bool = True, split_batches: bool = Fa
        self.scheduler = scheduler
        self.optimizers = optimizers if isinstance(optimizers, (list, tuple)) else [optimizers]
        self.split_batches = split_batches
        self.step_with_optimizer = step_with_optimizer
        self.gradient_state = GradientState()

    def step(self, *args, **kwargs):
        if not self.step_with_optimizer:
            # No link between scheduler and optimizer -> just step
            self.scheduler.step(*args, **kwargs)
            return

        # Otherwise, first make sure the optimizer was stepped.
        if not self.gradient_state.sync_gradients:
            if self.gradient_state.adjust_scheduler:
                self.scheduler._step_count += 1
            return

        for opt in self.optimizers:
            if opt.step_was_skipped:
                return
        if self.split_batches:
            # Split batches -> the training dataloader batch size is not changed so one step per trainin
            self.scheduler.step(*args, **kwargs)
        else:
            # Otherwise the training dataloader batch size was multiplied by `num_processes`, so we need
            # num_processes steps per training step
            num_processes = AcceleratorState().num_processes
            for _ in range(num_processes):
                # Special case when using OneCycle and `drop_last` was not used
                if hasattr(self.scheduler, "total_steps"):
                    if self.scheduler._step_count <= self.scheduler.total_steps:
                        self.scheduler.step(*args, **kwargs)
                else:
                    self.scheduler.step(*args, **kwargs)

    # Passthroughs
    def get_last_lr(self):
        return self.scheduler.get_last_lr()

    def state_dict(self):
        return self.scheduler.state_dict()

    def load_state_dict(self, state_dict):
        self.scheduler.load_state_dict(state_dict)

    def get_lr(self):
        return self.scheduler.get_lr()

    def print_lr(self, *args, **kwargs):
        return self.scheduler.print_lr(*args, **kwargs)
```

accelerate-main/docs/source/package_reference/launchers.mdx

```
# Launchers

Functions for launching training on distributed processes.


[[autodoc]] accelerate.notebook_launcher
[[autodoc]] accelerate.debug_launcher
```

accelerate-main/docs/source/package_reference/logging.mdx

~~~
<!--Copyright 2021 The HuggingFace Team. All rights reserved.

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in comp
the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is dis
an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the Licen
specific language governing permissions and limitations under the License.
-->

# Logging with Accelerate

Accelerate has its own logging utility to handle logging while in a distributed system.
To utilize this replace cases of `logging` with `accelerate.logging`:
```diff
- import logging
+ from accelerate.logging import get_logger
- logger = logging.getLogger(__name__)
+ logger = get_logger(__name__)
```

## Setting the log level

The log level can be set with the `ACCELERATE_LOG_LEVEL` environment variable or by passing
`log_level` to `get_logger`:
```python
from accelerate.logging import get_logger

logger = get_logger(__name__, log_level="INFO")
```

[[autodoc]] logging.get_logger
~~~

accelerate-main/src/accelerate/hooks.py

```
# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import functools
from typing import Dict, List, Mapping, Optional, Union

import torch
import torch.nn as nn

from .utils import (
    PrefixedDataset,
    find_device,
    is_mps_available,
    named_module_tensors,
    send_to_device,
    set_module_tensor_to_device,
)


class ModelHook:
    """
    A hook that contains callbacks to be executed just before and after the forward method of a model. T
    with PyTorch existing hooks is that they get passed along the kwargs.

    Class attribute:
    - **no_grad** (`bool`, *optional*, defaults to `False`) -- Whether or not to execute the actual forw
      the `torch.no_grad()` context manager.
    """

    no_grad = False

    def init_hook(self, module):
        """
        To be executed when the hook is attached to the module.

        Args:
            module (`torch.nn.Module`): The module attached to this hook.
        """
        return module

    def pre_forward(self, module, *args, **kwargs):
        """
        To be executed just before the forward method of the model.

        Args:
            module (`torch.nn.Module`): The module whose forward pass will be executed just after this e
            args (`Tuple[Any]`): The positional arguments passed to the module.
            kwargs (`Dict[Str, Any]`): The keyword arguments passed to the module.

        Returns:
            `Tuple[Tuple[Any], Dict[Str, Any]]`: A tuple with the treated `args` and `kwargs`.
        """
        return args, kwargs

    def post_forward(self, module, output):
        """
        To be executed just after the forward method of the model.
        Args:
            module (`torch.nn.Module`): The module whose forward pass been executed just before this eve
            output (`Any`): The output of the module.

        Returns:
            `Any`: The processed `output`.
        """
        return output

    def detach_hook(self, module):
        """
        To be executed when the hook is detached from a module.

        Args:
            module (`torch.nn.Module`): The module detached from this hook.
        """
        return module


class SequentialHook(ModelHook):
    """
    A hook that can contain several hooks and iterates through them at each event.
    """

    def __init__(self, *hooks):
        self.hooks = hooks

    def init_hook(self, module):
        for hook in self.hooks:
            module = hook.init_hook(module)
        return module

    def pre_forward(self, module, *args, **kwargs):
        for hook in self.hooks:
            args, kwargs = hook.pre_forward(module, *args, **kwargs)
        return args, kwargs

    def post_forward(self, module, output):
        for hook in self.hooks:
            output = hook.post_forward(module, output)
        return output

    def detach_hook(self, module):
        for hook in self.hooks:
            module = hook.detach_hook(module)
        return module


def add_hook_to_module(module: nn.Module, hook: ModelHook, append: bool = False):
    """
    Adds a hook to a given module. This will rewrite the `forward` method of the module to include the h
    this behavior and restore the original `forward` method, use `remove_hook_from_module`.

    <Tip warning={true}>

    If the module already contains a hook, this will replace it with the new hook passed by default. To
    together, pass `append=True`, so it chains the current and new hook into an instance of the `Sequent

    </Tip>

    Args:
        module (`torch.nn.Module`):
            The module to attach a hook to.
        hook (`ModelHook`):
            The hook to attach.
        append (`bool`, *optional*, defaults to `False`):
            Whether the hook should be chained with an existing one (if module already contains a hook)

    Returns:
        `torch.nn.Module`: The same module, with the hook attached (the module is modified in place, so
        be discarded).
    """
    if append and (getattr(module, "_hf_hook", None) is not None):
        old_hook = module._hf_hook
        remove_hook_from_module(module)
        hook = SequentialHook(old_hook, hook)

    if hasattr(module, "_hf_hook") and hasattr(module, "_old_forward"):
        # If we already put some hook on this module, we replace it with the new one.
        old_forward = module._old_forward
    else:
        old_forward = module.forward
        module._old_forward = old_forward

    module = hook.init_hook(module)
    module._hf_hook = hook

    @functools.wraps(old_forward)
    def new_forward(*args, **kwargs):
        args, kwargs = module._hf_hook.pre_forward(module, *args, **kwargs)
        if module._hf_hook.no_grad:
            with torch.no_grad():
                output = old_forward(*args, **kwargs)
        else:
            output = old_forward(*args, **kwargs)
        return module._hf_hook.post_forward(module, output)

    module.forward = new_forward
    return module


def remove_hook_from_module(module: nn.Module, recurse=False):
    """
    Removes any hook attached to a module via `add_hook_to_module`.

    Args:
        module (`torch.nn.Module`): The module to attach a hook to.
        recurse (`bool`, **optional**): Whether to remove the hooks recursively

    Returns:
        `torch.nn.Module`: The same module, with the hook detached (the module is modified in place, so
        be discarded).
    """

    if hasattr(module, "_hf_hook"):
        module._hf_hook.detach_hook(module)
        delattr(module, "_hf_hook")

    if hasattr(module, "_old_forward"):
        module.forward = module._old_forward
        delattr(module, "_old_forward")

    if recurse:
        for child in module.children():
            remove_hook_from_module(child, recurse)

    return module


class AlignDevicesHook(ModelHook):
    """
    A generic `ModelHook` that ensures inputs and model weights are on the same device for the forward p
    associated module, potentially offloading the weights after the forward pass.

    Args:
        execution_device (`torch.device`, *optional*):
            The device on which inputs and model weights should be placed before the forward pass.
        offload (`bool`, *optional*, defaults to `False`):
            Whether or not the weights should be offloaded after the forward pass.
        io_same_device (`bool`, *optional*, defaults to `False`):
            Whether or not the output should be placed on the same device as the input was.
        weights_map (`Mapping[str, torch.Tensor]`, *optional*):
            When the model weights are offloaded, a (potentially lazy) map from param names to the tenso
        offload_buffers (`bool`, *optional*, defaults to `False`):
            Whether or not to include the associated module's buffers when offloading.
        place_submodules (`bool`, *optional*, defaults to `False`):
            Whether to place the submodules on `execution_device` during the `init_hook` event.
    """

    def __init__(
        self,
        execution_device: Optional[Union[int, str, torch.device]] = None,
        offload: bool = False,
        io_same_device: bool = False,
        weights_map: Optional[Mapping] = None,
        offload_buffers: bool = False,
        place_submodules: bool = False,
    ):
        self.execution_device = execution_device
        self.offload = offload
        self.io_same_device = io_same_device
        self.weights_map = weights_map
        self.offload_buffers = offload_buffers
        self.place_submodules = place_submodules

        # Will contain the input device when `io_same_device=True`.
        self.input_device = None
        self.param_original_devices = {}
        self.buffer_original_devices = {}

    def __repr__(self):
        return (
            f"AlignDeviceHook(execution_device={self.execution_device}, offload={self.offload}, "
            f"io_same_device={self.io_same_device}, offload_buffers={self.offload_buffers}, "
            f"place_submodules={self.place_submodules})"
        )

    def init_hook(self, module):
        if not self.offload and self.execution_device is not None:
            for name, _ in named_module_tensors(module, recurse=self.place_submodules):
                set_module_tensor_to_device(module, name, self.execution_device)
        elif self.offload:
            self.original_devices = {
                name: param.device for name, param in named_module_tensors(module, recurse=self.place_su
            }
            if self.weights_map is None:
                self.weights_map = {
                    name: param.to("cpu")
                    for name, param in named_module_tensors(
                        module, include_buffers=self.offload_buffers, recurse=self.place_submodules
                    )
                }

            for name, _ in named_module_tensors(
                module, include_buffers=self.offload_buffers, recurse=self.place_submodules
            ):
                set_module_tensor_to_device(module, name, "meta")
            if not self.offload_buffers and self.execution_device is not None:
                for name, _ in module.named_buffers(recurse=self.place_submodules):
                    set_module_tensor_to_device(module, name, self.execution_device)
        return module

    def pre_forward(self, module, *args, **kwargs):
        if self.io_same_device:
            self.input_device = find_device([args, kwargs])
        if self.offload:
            for name, _ in named_module_tensors(
                module, include_buffers=self.offload_buffers, recurse=self.place_submodules
            ):
                set_module_tensor_to_device(module, name, self.execution_device, value=self.weights_map[

        return send_to_device(args, self.execution_device), send_to_device(kwargs, self.execution_device

    def post_forward(self, module, output):
        if self.offload:
            for name, _ in named_module_tensors(
                module, include_buffers=self.offload_buffers, recurse=self.place_submodules
            ):
                set_module_tensor_to_device(module, name, "meta")

        if self.io_same_device and self.input_device is not None:
            output = send_to_device(output, self.input_device)

        return output

    def detach_hook(self, module):
        if self.offload:
            for name, device in self.original_devices.items():
                if device != torch.device("meta"):
                    set_module_tensor_to_device(module, name, device, value=self.weights_map.get(name, N


def attach_execution_device_hook(
    module: torch.nn.Module,
    execution_device: Union[int, str, torch.device],
    preload_module_classes: Optional[List[str]] = None,
):
    """
    Recursively attaches `AlignDevicesHook` to all submodules of a given model to make sure they have th
    execution device

    Args:
        module (`torch.nn.Module`):
            The module where we want to attach the hooks.
        execution_device (`int`, `str` or `torch.device`):
            The device on which inputs and model weights should be placed before the forward pass.
        preload_module_classes (`List[str]`, *optional*):
            A list of classes whose instances should load all their weights (even in the submodules) at
            of the forward. This should only be used for classes that have submodules which are register
            called directly during the forward, for instance if a `dense` linear layer is registered, bu
            `dense.weight` and `dense.bias` are used in some operations instead of calling `dense` direc
    """
    if not hasattr(module, "_hf_hook") and len(module.state_dict()) > 0:
        add_hook_to_module(module, AlignDevicesHook(execution_device))

    # Break the recursion if we get to a preload module.
    if preload_module_classes is not None and module.__class__.__name__ in preload_module_classes:
        return

    for child in module.children():
        attach_execution_device_hook(child, execution_device)


def attach_align_device_hook(
    module: torch.nn.Module,
    execution_device: Optional[torch.device] = None,
    offload: bool = False,
    weights_map: Optional[Mapping] = None,
    offload_buffers: bool = False,
    module_name: str = "",
    preload_module_classes: Optional[List[str]] = None,
):
    """
    Recursively attaches `AlignDevicesHook` to all submodules of a given model that have direct paramete
    buffers.

    Args:
        module (`torch.nn.Module`):
            The module where we want to attach the hooks.
        execution_device (`torch.device`, *optional*):
            The device on which inputs and model weights should be placed before the forward pass.
        offload (`bool`, *optional*, defaults to `False`):
            Whether or not the weights should be offloaded after the forward pass.
        weights_map (`Mapping[str, torch.Tensor]`, *optional*):
            When the model weights are offloaded, a (potentially lazy) map from param names to the tenso
        offload_buffers (`bool`, *optional*, defaults to `False`):
            Whether or not to include the associated module's buffers when offloading.
        module_name (`str`, *optional*, defaults to `""`):
            The name of the module.
        preload_module_classes (`List[str]`, *optional*):
            A list of classes whose instances should load all their weights (even in the submodules) at
            of the forward. This should only be used for classes that have submodules which are register
            called directly during the forward, for instance if a `dense` linear layer is registered, bu
            `dense.weight` and `dense.bias` are used in some operations instead of calling `dense` direc
    """
    # Attach the hook on this module if it has any direct tensor.
    directs = named_module_tensors(module)
    full_offload = (
        offload and preload_module_classes is not None and module.__class__.__name__ in preload_module_c
    )

    if len(list(directs)) > 0 or full_offload:
        if weights_map is not None:
            prefix = f"{module_name}." if len(module_name) > 0 else ""
            prefixed_weights_map = PrefixedDataset(weights_map, prefix)
        else:
            prefixed_weights_map = None
        hook = AlignDevicesHook(
            execution_device=execution_device,
            offload=offload,
            weights_map=prefixed_weights_map,
            offload_buffers=offload_buffers,
            place_submodules=full_offload,
        )
        add_hook_to_module(module, hook, append=True)

    # We stop the recursion in case we hit the full offload.
    if full_offload:
        return

    # Recurse on all children of the module.
    for child_name, child in module.named_children():
        child_name = f"{module_name}.{child_name}" if len(module_name) > 0 else child_name
        attach_align_device_hook(
            child,
            execution_device=execution_device,
            offload=offload,
            weights_map=weights_map,
            offload_buffers=offload_buffers,
            module_name=child_name,
            preload_module_classes=preload_module_classes,
        )


def remove_hook_from_submodules(module: nn.Module):
    """
    Recursively removes all hooks attached on the submodules of a given model.

    Args:
        module (`torch.nn.Module`): The module on which to remove all hooks.
    """
    remove_hook_from_module(module)
    for child in module.children():
        remove_hook_from_submodules(child)


def attach_align_device_hook_on_blocks(
    module: nn.Module,
    execution_device: Optional[Union[torch.device, Dict[str, torch.device]]] = None,
    offload: Union[bool, Dict[str, bool]] = False,
    weights_map: Mapping = None,
    offload_buffers: bool = False,
    module_name: str = "",
    preload_module_classes: Optional[List[str]] = None,
):
    """
    Attaches `AlignDevicesHook` to all blocks of a given model as needed.

    Args:
        module (`torch.nn.Module`):
            The module where we want to attach the hooks.
        execution_device (`torch.device` or `Dict[str, torch.device]`, *optional*):
            The device on which inputs and model weights should be placed before the forward pass. It ca
            for the whole module, or a dictionary mapping module name to device.
        offload (`bool`, *optional*, defaults to `False`):
            Whether or not the weights should be offloaded after the forward pass. It can be one boolean
            module, or a dictionary mapping module name to boolean.
        weights_map (`Mapping[str, torch.Tensor]`, *optional*):
            When the model weights are offloaded, a (potentially lazy) map from param names to the tenso
        offload_buffers (`bool`, *optional*, defaults to `False`):
            Whether or not to include the associated module's buffers when offloading.
        module_name (`str`, *optional*, defaults to `""`):
            The name of the module.
        preload_module_classes (`List[str]`, *optional*):
            A list of classes whose instances should load all their weights (even in the submodules) at
            of the forward. This should only be used for classes that have submodules which are register
            called directly during the forward, for instance if a `dense` linear layer is registered, bu
            `dense.weight` and `dense.bias` are used in some operations instead of calling `dense` direc
    """
    # If one device and one offload, we've got one hook.
    if not isinstance(execution_device, Mapping) and not isinstance(offload, dict):
        if not offload:
            hook = AlignDevicesHook(execution_device=execution_device, io_same_device=True, place_submod
            add_hook_to_module(module, hook)
        else:
            attach_align_device_hook(
                module,
                execution_device=execution_device,
                offload=True,
                weights_map=weights_map,
                offload_buffers=offload_buffers,
                module_name=module_name,
            )
        return

    if not isinstance(execution_device, Mapping):
        execution_device = {key: execution_device for key in offload.keys()}
    if not isinstance(offload, Mapping):
        offload = {key: offload for key in execution_device.keys()}

    if module_name in execution_device and module_name in offload and not offload[module_name]:
        hook = AlignDevicesHook(
            execution_device=execution_device[module_name],
            offload_buffers=offload_buffers,
            io_same_device=(module_name == ""),
            place_submodules=True,
        )
        add_hook_to_module(module, hook)
        attach_execution_device_hook(module, execution_device[module_name])
    elif module_name in execution_device and module_name in offload:
        attach_align_device_hook(
            module,
            execution_device=execution_device[module_name],
            offload=True,
            weights_map=weights_map,
            offload_buffers=offload_buffers,
            module_name=module_name,
            preload_module_classes=preload_module_classes,
        )
        if not hasattr(module, "_hf_hook"):
            hook = AlignDevicesHook(execution_device=execution_device[module_name], io_same_device=(modu
            add_hook_to_module(module, hook)
        attach_execution_device_hook(
            module, execution_device[module_name], preload_module_classes=preload_module_classes
        )
    elif module_name == "":
        hook = AlignDevicesHook(execution_device=execution_device.get(""), io_same_device=True)
        add_hook_to_module(module, hook)

    for child_name, child in module.named_children():
        child_name = f"{module_name}.{child_name}" if len(module_name) > 0 else child_name
        attach_align_device_hook_on_blocks(
            child,
            execution_device=execution_device,
            offload=offload,
            weights_map=weights_map,
            offload_buffers=offload_buffers,
            module_name=child_name,
            preload_module_classes=preload_module_classes,
        )


class CpuOffload(ModelHook):
    """
    Offloads a model on the CPU until its forward pass is called. The model will not be offloaded back t
    the forward, the user needs to call the `init_hook` method again for this.

    Args:
        execution_device(`str`, `int` or `torch.device`, *optional*):
            The device on which the model should be executed. Will default to the MPS device if it's ava
            GPU 0 if there is a GPU, and finally to the CPU.
        prev_module_hook (`UserCpuOffloadHook`, *optional*):
            The hook sent back by [`cpu_offload_with_hook`] for a previous model in the pipeline you are
            passed, its offload method will be called just before the forward of the model to which this
            attached.
    """

    def __init__(
        self,
        execution_device: Optional[Union[str, int, torch.device]] = None,
        prev_module_hook: Optional["UserCpuOffloadHook"] = None,
    ):
        self.prev_module_hook = prev_module_hook

        if execution_device is not None:
            self.execution_device = execution_device
        elif is_mps_available():
            self.execution_device = torch.device("mps")
        elif torch.cuda.is_available():
            self.execution_device = torch.device(0)
        else:
            self.execution_device = torch.device("cpu")

    def init_hook(self, module):
        return module.to("cpu")

    def pre_forward(self, module, *args, **kwargs):
        module.to(self.execution_device)
        if self.prev_module_hook is not None:
            self.prev_module_hook.offload()
        return send_to_device(args, self.execution_device), send_to_device(kwargs, self.execution_device


class UserCpuOffloadHook:
    """
    A simple hook grouping a model and a `ModelHook`, which provides easy APIs for to call the init meth
    or remove it entirely.
    """

    def __init__(self, model, hook):
        self.model = model
        self.hook = hook

    def offload(self):
        self.hook.init_hook(self.model)

    def remove(self):
        remove_hook_from_module(self.model)
```

accelerate-main/docs/source/package_reference/big_modeling.mdx

```
<!--Copyright 2021 The HuggingFace Team. All rights reserved.

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in comp
the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is dis
an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the Licen
specific language governing permissions and limitations under the License.
-->

# Working with large models

## Dispatching and Offloading Models

[[autodoc]] big_modeling.init_empty_weights
[[autodoc]] big_modeling.cpu_offload
[[autodoc]] big_modeling.disk_offload
[[autodoc]] big_modeling.dispatch_model
[[autodoc]] big_modeling.load_checkpoint_and_dispatch

## Model Hooks

### Hook Classes

[[autodoc]] hooks.ModelHook
[[autodoc]] hooks.AlignDevicesHook
[[autodoc]] hooks.SequentialHook

### Adding Hooks

[[autodoc]] hooks.add_hook_to_module
[[autodoc]] hooks.attach_execution_device_hook
[[autodoc]] hooks.attach_align_device_hook
[[autodoc]] hooks.attach_align_device_hook_on_blocks

### Removing Hooks

[[autodoc]] hooks.remove_hook_from_module
[[autodoc]] hooks.remove_hook_from_submodules
```

accelerate-main/src/accelerate/optimizer.py

```
# Copyright 2021 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import inspect
import warnings

import torch

from .state import AcceleratorState, GradientState
from .utils import DistributedType, honor_type, is_torch_version, is_tpu_available


if is_tpu_available(check_device=False):
    import torch_xla.core.xla_model as xm


def move_to_device(state, device):
    if isinstance(state, (list, tuple)):
        return honor_type(state, (move_to_device(t, device) for t in state))
    elif isinstance(state, dict):
        return type(state)({k: move_to_device(v, device) for k, v in state.items()})
    elif isinstance(state, torch.Tensor):
        return state.to(device)
    return state


class AcceleratedOptimizer(torch.optim.Optimizer):
    """
    Internal wrapper around a torch optimizer.

    Conditionally will perform `step` and `zero_grad` if gradients should be synchronized when performin
    accumulation.

    Args:
        optimizer (`torch.optim.optimizer.Optimizer`):
            The optimizer to wrap.
        device_placement (`bool`, *optional*, defaults to `True`):
            Whether or not the optimizer should handle device placement. If so, it will place the state
            `optimizer` on the right device.
        scaler (`torch.cuda.amp.grad_scaler.GradScaler`, *optional*):
            The scaler to use in the step function if training with mixed precision.
    """

    def __init__(self, optimizer, device_placement=True, scaler=None):
        self.optimizer = optimizer
        self.scaler = scaler
        self.accelerator_state = AcceleratorState()
        self.gradient_state = GradientState()
        self.device_placement = device_placement
        self._is_overflow = False

        # Handle device placement
        if device_placement:
            state_dict = self.optimizer.state_dict()
            if self.accelerator_state.distributed_type == DistributedType.TPU:
                xm.send_cpu_data_to_device(state_dict, self.accelerator_state.device)
            else:
                state_dict = move_to_device(state_dict, self.accelerator_state.device)
            self.optimizer.load_state_dict(state_dict)

    @property
    def state(self):
        return self.optimizer.state

    @state.setter
    def state(self, state):
        self.optimizer.state = state

    @property
    def param_groups(self):
        return self.optimizer.param_groups

    @param_groups.setter
    def param_groups(self, param_groups):
        self.optimizer.param_groups = param_groups

    @property
    def defaults(self):
        return self.optimizer.defaults

    @defaults.setter
    def defaults(self, defaults):
        self.optimizer.defaults = defaults

    def add_param_group(self, param_group):
        self.optimizer.add_param_group(param_group)

    def load_state_dict(self, state_dict):
        if self.accelerator_state.distributed_type == DistributedType.TPU and self.device_placement:
            xm.send_cpu_data_to_device(state_dict, self.accelerator_state.device)
        self.optimizer.load_state_dict(state_dict)

    def state_dict(self):
        return self.optimizer.state_dict()

    def zero_grad(self, set_to_none=None):
        if self.gradient_state.sync_gradients:
            if is_torch_version("<", "1.7.0"):
                if set_to_none is not None:
                    raise ValueError(
                        "`set_to_none` for Optimizer.zero_grad` was introduced in PyTorch 1.7.0 and can'
                        f"earlier versions (found version {torch.__version__})."
                    )
                self.optimizer.zero_grad()
            else:
                accept_arg = "set_to_none" in inspect.signature(self.optimizer.zero_grad).parameters
                if accept_arg:
                    if set_to_none is None:
                        set_to_none = False
                    self.optimizer.zero_grad(set_to_none=set_to_none)
                else:
                    if set_to_none is not None:
                        raise ValueError("`set_to_none` for Optimizer.zero_grad` is not supported by thi
                    self.optimizer.zero_grad()

    def step(self, closure=None):
        if self.gradient_state.sync_gradients:
            if self.accelerator_state.distributed_type == DistributedType.TPU:
                optimizer_args = {"closure": closure} if closure is not None else {}
                xm.optimizer_step(self.optimizer, optimizer_args=optimizer_args)
            elif self.scaler is not None:
                scale_before = self.scaler.get_scale()
                self.scaler.step(self.optimizer, closure)
                self.scaler.update()
                scale_after = self.scaler.get_scale()
                # If we reduced the loss scale, it means the optimizer step was skipped because of gradi
                self._is_overflow = scale_after < scale_before
            else:
                self.optimizer.step(closure)
```

```
def _switch_parameters(self, parameters_map):
    for param_group in self.optimizer.param_groups:
        param_group["params"] = [parameters_map.get(p, p) for p in param_group["params"]]

@property
def is_overflow(self):
    """Whether or not the optimizer step was done, or skipped because of gradient overflow."""
    warnings.warn(
        "The `is_overflow` property is deprecated and will be removed in version 1.0 of Accelerate u
        "`optimizer.step_was_skipped` instead.",
        FutureWarning,
    )
    return self._is_overflow

@property
def step_was_skipped(self):
    """Whether or not the optimizer step was skipped."""
    return self._is_overflow

def __getstate__(self):
    return self.__dict__.copy()

def __setstate__(self, state):
    self.__dict__.update(state)
```

accelerate-main/src/accelerate/test_utils/training.py

```
# Copyright 2021 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import numpy as np
import torch
from torch.utils.data import DataLoader

from accelerate.utils.dataclasses import DistributedType


class RegressionDataset:
    def __init__(self, a=2, b=3, length=64, seed=None):
        if seed is not None:
            np.random.seed(seed)
        self.length = length
        self.x = np.random.normal(size=(length,)).astype(np.float32)
        self.y = a * self.x + b + np.random.normal(scale=0.1, size=(length,)).astype(np.float32)

    def __len__(self):
        return self.length

    def __getitem__(self, i):
        return {"x": self.x[i], "y": self.y[i]}


class RegressionModel(torch.nn.Module):
    def __init__(self, a=0, b=0, double_output=False):
        super().__init__()
        self.a = torch.nn.Parameter(torch.tensor(a).float())
        self.b = torch.nn.Parameter(torch.tensor(b).float())
        self.first_batch = True

    def forward(self, x=None):
        if self.first_batch:
            print(f"Model dtype: {self.a.dtype}, {self.b.dtype}. Input dtype: {x.dtype}")
            self.first_batch = False
        return x * self.a + self.b


def mocked_dataloaders(accelerator, batch_size: int = 16):
    from datasets import load_dataset
    from transformers import AutoTokenizer

    tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
    data_files = {"train": "tests/test_samples/MRPC/train.csv", "validation": "tests/test_samples/MRPC/d
    datasets = load_dataset("csv", data_files=data_files)
    label_list = datasets["train"].unique("label")

    label_to_id = {v: i for i, v in enumerate(label_list)}

    def tokenize_function(examples):
        # max_length=None => use the model max length (it's actually the default)
        outputs = tokenizer(
            examples["sentence1"], examples["sentence2"], truncation=True, max_length=None, padding="max
        )
        if "label" in examples:
            outputs["labels"] = [label_to_id[l] for l in examples["label"]]
        return outputs

    # Apply the method we just defined to all the examples in all the splits of the dataset
    tokenized_datasets = datasets.map(
        tokenize_function,
        batched=True,
        remove_columns=["sentence1", "sentence2", "label"],
    )

    def collate_fn(examples):
        # On TPU it's best to pad everything to the same length or training will be very slow.
        if accelerator.distributed_type == DistributedType.TPU:
            return tokenizer.pad(examples, padding="max_length", max_length=128, return_tensors="pt")
        return tokenizer.pad(examples, padding="longest", return_tensors="pt")

    # Instantiate dataloaders.
    train_dataloader = DataLoader(tokenized_datasets["train"], shuffle=True, collate_fn=collate_fn, batc
    eval_dataloader = DataLoader(tokenized_datasets["validation"], shuffle=False, collate_fn=collate_fn,

    return train_dataloader, eval_dataloader
```

accelerate-main/src/accelerate/commands/config/__init__.py

```
#!/usr/bin/env python

# Copyright 2021 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import argparse

from .config import config_command_parser
from .config_args import default_config_file, load_config_from_file  # noqa: F401
from .default import default_command_parser
from .update import update_command_parser


def get_config_parser(subparsers=None):
    parent_parser = argparse.ArgumentParser(add_help=False, allow_abbrev=False)
    # The main config parser
    config_parser = config_command_parser(subparsers)
    # The subparser to add commands to
    subcommands = config_parser.add_subparsers(title="subcommands", dest="subcommand")

    # Then add other parsers with the parent parser
    default_command_parser(subcommands, parents=[parent_parser])
    update_command_parser(subcommands, parents=[parent_parser])

    return config_parser


def main():
    config_parser = get_config_parser()
    args = config_parser.parse_args()

    if not hasattr(args, "func"):
        config_parser.print_help()
        exit(1)

    # Run
    args.func(args)


if __name__ == "__main__":
    main()
```

accelerate-main/src/accelerate/test_utils/testing.py

```
# Copyright 2021 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import asyncio
import os
import shutil
import subprocess
import sys
import tempfile
import unittest
from distutils.util import strtobool
from functools import partial
from pathlib import Path
from typing import List, Union
from unittest import mock

import torch

from ..state import AcceleratorState, PartialState
from ..utils import (
    gather,
    is_comet_ml_available,
    is_datasets_available,
    is_deepspeed_available,
    is_safetensors_available,
    is_tensorboard_available,
    is_torch_version,
    is_tpu_available,
    is_transformers_available,
    is_wandb_available,
)


def parse_flag_from_env(key, default=False):
    try:
        value = os.environ[key]
    except KeyError:
        # KEY isn't set, default to `default`.
        _value = default
    else:
        # KEY is set, convert it to True or False.
        try:
            _value = strtobool(value)
        except ValueError:
            # More values are supported, but let's keep the message simple.
            raise ValueError(f"If set, {key} must be yes or no.")
    return _value


_run_slow_tests = parse_flag_from_env("RUN_SLOW", default=False)


def skip(test_case):
    "Decorator that skips a test unconditionally"
    return unittest.skip("Test was skipped")(test_case)
def slow(test_case):
    """
    Decorator marking a test as slow. Slow tests are skipped by default. Set the RUN_SLOW environment va
    truthy value to run them.
    """
    return unittest.skipUnless(_run_slow_tests, "test is slow")(test_case)


def require_cpu(test_case):
    """
    Decorator marking a test that must be only ran on the CPU. These tests are skipped when a GPU is ava
    """
    return unittest.skipUnless(not torch.cuda.is_available(), "test requires only a CPU")(test_case)


def require_cuda(test_case):
    """
    Decorator marking a test that requires CUDA. These tests are skipped when there are no GPU available
    """
    return unittest.skipUnless(torch.cuda.is_available(), "test requires a GPU")(test_case)


def require_mps(test_case):
    """
    Decorator marking a test that requires MPS backend. These tests are skipped when torch doesn't suppo
    backend.
    """
    is_mps_supported = hasattr(torch.backends, "mps") and torch.backends.mps.is_available()
    return unittest.skipUnless(is_mps_supported, "test requires a `mps` backend support in `torch`")(tes


def require_huggingface_suite(test_case):
    """
    Decorator marking a test that requires transformers and datasets. These tests are skipped when they
    """
    return unittest.skipUnless(
        is_transformers_available() and is_datasets_available(), "test requires the Hugging Face suite"
    )(test_case)


def require_tpu(test_case):
    """
    Decorator marking a test that requires TPUs. These tests are skipped when there are no TPUs availabl
    """
    return unittest.skipUnless(is_tpu_available(), "test requires TPU")(test_case)


def require_single_gpu(test_case):
    """
    Decorator marking a test that requires CUDA on a single GPU. These tests are skipped when there are
    available or number of GPUs is more than one.
    """
    return unittest.skipUnless(torch.cuda.device_count() == 1, "test requires a GPU")(test_case)


def require_multi_gpu(test_case):
    """
    Decorator marking a test that requires a multi-GPU setup. These tests are skipped on a machine witho
    GPUs.
    """
    return unittest.skipUnless(torch.cuda.device_count() > 1, "test requires multiple GPUs")(test_case)


def require_safetensors(test_case):
    """
    Decorator marking a test that requires safetensors installed. These tests are skipped when safetenso
    installed
    """
    return unittest.skipUnless(is_safetensors_available(), "test requires safetensors")(test_case)


def require_deepspeed(test_case):
    """
    Decorator marking a test that requires DeepSpeed installed. These tests are skipped when DeepSpeed i
    """
    return unittest.skipUnless(is_deepspeed_available(), "test requires DeepSpeed")(test_case)


def require_fsdp(test_case):
    """
    Decorator marking a test that requires FSDP installed. These tests are skipped when FSDP isn't insta
    """
    return unittest.skipUnless(is_torch_version(">=", "1.12.0"), "test requires torch version >= 1.12.0"


def require_torch_min_version(test_case=None, version=None):
    """
    Decorator marking that a test requires a particular torch version to be tested. These tests are skip
    installed torch version is less than the required one.
    """
    if test_case is None:
        return partial(require_torch_min_version, version=version)
    return unittest.skipUnless(is_torch_version(">=", version), f"test requires torch version >= {versio


def require_tensorboard(test_case):
    """
    Decorator marking a test that requires tensorboard installed. These tests are skipped when tensorboa
    installed
    """
    return unittest.skipUnless(is_tensorboard_available(), "test requires Tensorboard")(test_case)


def require_wandb(test_case):
    """
    Decorator marking a test that requires wandb installed. These tests are skipped when wandb isn't ins
    """
    return unittest.skipUnless(is_wandb_available(), "test requires wandb")(test_case)


def require_comet_ml(test_case):
    """
    Decorator marking a test that requires comet_ml installed. These tests are skipped when comet_ml isn
    """
    return unittest.skipUnless(is_comet_ml_available(), "test requires comet_ml")(test_case)


_atleast_one_tracker_available = (
    any([is_wandb_available(), is_tensorboard_available()]) and not is_comet_ml_available()
)


def require_trackers(test_case):
    """
    Decorator marking that a test requires at least one tracking library installed. These tests are skip
    are installed
    """
    return unittest.skipUnless(
        _atleast_one_tracker_available,
        "test requires at least one tracker to be available and for `comet_ml` to not be installed",
    )(test_case)


class TempDirTestCase(unittest.TestCase):
    """
    A TestCase class that keeps a single `tempfile.TemporaryDirectory` open for the duration of the clas
    data at the start of a test, and then destroyes it at the end of the TestCase.

    Useful for when a class or API requires a single constant folder throughout it's use, such as Weight

    The temporary directory location will be stored in `self.tmpdir`
    """

    clear_on_setup = True

    @classmethod
```

~~~
    def setUpClass(cls):
        "Creates a `tempfile.TemporaryDirectory` and stores it in `cls.tmpdir`"
        cls.tmpdir = tempfile.mkdtemp()

    @classmethod
    def tearDownClass(cls):
        "Remove `cls.tmpdir` after test suite has finished"
        if os.path.exists(cls.tmpdir):
            shutil.rmtree(cls.tmpdir)

    def setUp(self):
        "Destroy all contents in `self.tmpdir`, but not `self.tmpdir`"
        if self.clear_on_setup:
            for path in Path(self.tmpdir).glob("**/*"):
                if path.is_file():
                    path.unlink()
                elif path.is_dir():
                    shutil.rmtree(path)


class AccelerateTestCase(unittest.TestCase):
    """
    A TestCase class that will reset the accelerator state at the end of every test. Every test that che
    the `AcceleratorState` class should inherit from this to avoid silent failures due to state being sh
    tests.
    """

    def tearDown(self):
        super().tearDown()
        # Reset the state of the AcceleratorState singleton.
        AcceleratorState._reset_state()
        PartialState._reset_state()


class MockingTestCase(unittest.TestCase):
    """
    A TestCase class designed to dynamically add various mockers that should be used in every test, mimi
    behavior of a class-wide mock when defining one normally will not do.

    Useful when a mock requires specific information available only initialized after `TestCase.setUpCla
    setting an environment variable with that information.

    The `add_mocks` function should be ran at the end of a `TestCase`'s `setUp` function, after a call t
    `super().setUp()` such as:
    ```python
    def setUp(self):
        super().setUp()
        mocks = mock.patch.dict(os.environ, {"SOME_ENV_VAR", "SOME_VALUE"})
        self.add_mocks(mocks)
    ```
    """

    def add_mocks(self, mocks: Union[mock.Mock, List[mock.Mock]]):
        """
        Add custom mocks for tests that should be repeated on each test. Should be called during
        `MockingTestCase.setUp`, after `super().setUp()`.

        Args:
            mocks (`mock.Mock` or list of `mock.Mock`):
                Mocks that should be added to the `TestCase` after `TestCase.setUpClass` has been run
        """
        self.mocks = mocks if isinstance(mocks, (tuple, list)) else [mocks]
        for m in self.mocks:
            m.start()
            self.addCleanup(m.stop)


def are_the_same_tensors(tensor):
    state = AcceleratorState()
    tensor = tensor[None].clone().to(state.device)
    tensors = gather(tensor).cpu()
    tensor = tensor[0].cpu()
~~~

```
    for i in range(tensors.shape[0]):
        if not torch.equal(tensors[i], tensor):
            return False
    return True


class _RunOutput:
    def __init__(self, returncode, stdout, stderr):
        self.returncode = returncode
        self.stdout = stdout
        self.stderr = stderr


async def _read_stream(stream, callback):
    while True:
        line = await stream.readline()
        if line:
            callback(line)
        else:
            break


async def _stream_subprocess(cmd, env=None, stdin=None, timeout=None, quiet=False, echo=False) -> _RunOu
    if echo:
        print("\nRunning: ", " ".join(cmd))

    p = await asyncio.create_subprocess_exec(
        cmd[0],
        *cmd[1:],
        stdin=stdin,
        stdout=asyncio.subprocess.PIPE,
        stderr=asyncio.subprocess.PIPE,
        env=env,
    )

    # note: there is a warning for a possible deadlock when using `wait` with huge amounts of data in th
    # https://docs.python.org/3/library/asyncio-subprocess.html#asyncio.asyncio.subprocess.Process.wait
    #
    # If it starts hanging, will need to switch to the following code. The problem is that no data
    # will be seen until it's done and if it hangs for example there will be no debug info.
    # out, err = await p.communicate()
    # return _RunOutput(p.returncode, out, err)

    out = []
    err = []

    def tee(line, sink, pipe, label=""):
        line = line.decode("utf-8").rstrip()
        sink.append(line)
        if not quiet:
            print(label, line, file=pipe)

    # XXX: the timeout doesn't seem to make any difference here
    await asyncio.wait(
        [
            asyncio.create_task(_read_stream(p.stdout, lambda l: tee(l, out, sys.stdout, label="stdout:"
            asyncio.create_task(_read_stream(p.stderr, lambda l: tee(l, err, sys.stderr, label="stderr:"
        ],
        timeout=timeout,
    )
    return _RunOutput(await p.wait(), out, err)


def execute_subprocess_async(cmd, env=None, stdin=None, timeout=180, quiet=False, echo=True) -> _RunOutp
    loop = asyncio.get_event_loop()
    result = loop.run_until_complete(
        _stream_subprocess(cmd, env=env, stdin=stdin, timeout=timeout, quiet=quiet, echo=echo)
    )

    cmd_str = " ".join(cmd)
    if result.returncode > 0:
        stderr = "\n".join(result.stderr)
```

```
            raise RuntimeError(
                f"'{cmd_str}' failed with returncode {result.returncode}\n\n"
                f"The combined stderr from workers follows:\n{stderr}"
            )

    return result


class SubprocessCallException(Exception):
    pass


def run_command(command: List[str], return_stdout=False):
    """
    Runs `command` with `subprocess.check_output` and will potentially return the `stdout`. Will also pr
    if an error occured while running `command`
    """
    try:
        output = subprocess.check_output(command, stderr=subprocess.STDOUT)
        if return_stdout:
            if hasattr(output, "decode"):
                output = output.decode("utf-8")
            return output
    except subprocess.CalledProcessError as e:
        raise SubprocessCallException(
            f"Command `{' '.join(command)}` failed with the following error:\n\n{e.output.decode()}"
        ) from e
```

accelerate-main/src/accelerate/test_utils/examples.py

```
#!/usr/bin/env python

# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
"""
A collection of utilities for comparing `examples/complete_*_example.py` scripts with the capabilities i
`examples/by_feature` example. `compare_against_test` is the main function that should be used when test
others are used to either get the code that matters, or to preprocess them (such as stripping comments)
"""

import os
from typing import List


def get_function_contents_by_name(lines: List[str], name: str):
    """
    Extracts a function from `lines` of segmented source code with the name `name`.

    Args:
        lines (`List[str]`):
            Source code of a script seperated by line.
        name (`str`):
            The name of the function to extract. Should be either `training_function` or `main`
    """
    if name != "training_function" and name != "main":
        raise ValueError(f"Incorrect function name passed: {name}, choose either 'main' or 'training_fun
    good_lines, found_start = [], False
    for line in lines:
        if not found_start and f"def {name}" in line:
            found_start = True
            good_lines.append(line)
            continue
        if found_start:
            if name == "training_function" and "def main" in line:
                return good_lines
            if name == "main" and "if __name__" in line:
                return good_lines
            good_lines.append(line)


def clean_lines(lines: List[str]):
    """
    Filters `lines` and removes any entries that start with a comment ('#') or is just a newline ('\n')

    Args:
        lines (`List[str]`):
            Source code of a script seperated by line.
    """
    return [line for line in lines if not line.lstrip().startswith("#") and line != "\n"]


def compare_against_test(base_filename: str, feature_filename: str, parser_only: bool, secondary_filenam
    """
    Tests whether the additional code inside of `feature_filename` was implemented in `base_filename`. T
    used when testing to see if `complete_*_.py` examples have all of the implementations from each of t
    `examples/by_feature/*` scripts.
    It utilizes `nlp_example.py` to extract out all of the repeated training code, so that only the new
    is examined and checked. If something *other* than `nlp_example.py` should be used, such as `cv_exam
    `complete_cv_example.py` script, it should be passed in for the `secondary_filename` parameter.

    Args:
        base_filename (`str` or `os.PathLike`):
            The filepath of a single "complete" example script to test, such as `examples/complete_cv_ex
        feature_filename (`str` or `os.PathLike`):
            The filepath of a single feature example script. The contents of this script are checked to
            exist in `base_filename`
        parser_only (`bool`):
            Whether to compare only the `main()` sections in both files, or to compare the contents of
            `training_loop()`
        secondary_filename (`str`, *optional*):
            A potential secondary filepath that should be included in the check. This function extracts
            functionalities off of "examples/nlp_example.py", so if `base_filename` is a script other th
            `complete_nlp_example.py`, the template script should be included here. Such as `examples/cv
    """
    with open(base_filename, "r") as f:
        base_file_contents = f.readlines()
    with open(os.path.abspath(os.path.join("examples", "nlp_example.py")), "r") as f:
        full_file_contents = f.readlines()
    with open(feature_filename, "r") as f:
        feature_file_contents = f.readlines()
    if secondary_filename is not None:
        with open(secondary_filename, "r") as f:
            secondary_file_contents = f.readlines()

    # This is our base, we remove all the code from here in our `full_filename` and `feature_filename` t
    if parser_only:
        base_file_func = clean_lines(get_function_contents_by_name(base_file_contents, "main"))
        full_file_func = clean_lines(get_function_contents_by_name(full_file_contents, "main"))
        feature_file_func = clean_lines(get_function_contents_by_name(feature_file_contents, "main"))
        if secondary_filename is not None:
            secondary_file_func = clean_lines(get_function_contents_by_name(secondary_file_contents, "ma
    else:
        base_file_func = clean_lines(get_function_contents_by_name(base_file_contents, "training_functio
        full_file_func = clean_lines(get_function_contents_by_name(full_file_contents, "training_functio
        feature_file_func = clean_lines(get_function_contents_by_name(feature_file_contents, "training_f
        if secondary_filename is not None:
            secondary_file_func = clean_lines(
                get_function_contents_by_name(secondary_file_contents, "training_function")
            )

    _dl_line = "train_dataloader, eval_dataloader = get_dataloaders(accelerator, batch_size)\n"

    # Specific code in our script that differs from the full version, aka what is new
    new_feature_code = []
    passed_idxs = []  # We keep track of the idxs just in case it's a repeated statement
    it = iter(feature_file_func)
    for i in range(len(feature_file_func) - 1):
        if i not in passed_idxs:
            line = next(it)
            if (line not in full_file_func) and (line.lstrip() != _dl_line):
                if "TESTING_MOCKED_DATALOADERS" not in line:
                    new_feature_code.append(line)
                    passed_idxs.append(i)
                else:
                    # Skip over the `config['num_epochs'] = 2` statement
                    _ = next(it)

    # Extract out just the new parts from the full_file_training_func
    new_full_example_parts = []
    passed_idxs = []  # We keep track of the idxs just in case it's a repeated statement
    for i, line in enumerate(base_file_func):
        if i not in passed_idxs:
            if (line not in full_file_func) and (line.lstrip() != _dl_line):
                if "TESTING_MOCKED_DATALOADERS" not in line:
                    new_full_example_parts.append(line)
                    passed_idxs.append(i)

    # Finally, get the overall diff
    diff_from_example = [line for line in new_feature_code if line not in new_full_example_parts]
```

```
if secondary_filename is not None:
    diff_from_two = [line for line in full_file_contents if line not in secondary_file_func]
    diff_from_example = [line for line in diff_from_example if line not in diff_from_two]

return diff_from_example
```

accelerate-main/src/accelerate/commands/config/__init__.py

```python
#!/usr/bin/env python

# Copyright 2021 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import argparse

from .config import config_command_parser
from .config_args import default_config_file, load_config_from_file  # noqa: F401
from .default import default_command_parser
from .update import update_command_parser


def get_config_parser(subparsers=None):
    parent_parser = argparse.ArgumentParser(add_help=False, allow_abbrev=False)
    # The main config parser
    config_parser = config_command_parser(subparsers)
    # The subparser to add commands to
    subcommands = config_parser.add_subparsers(title="subcommands", dest="subcommand")

    # Then add other parsers with the parent parser
    default_command_parser(subcommands, parents=[parent_parser])
    update_command_parser(subcommands, parents=[parent_parser])

    return config_parser


def main():
    config_parser = get_config_parser()
    args = config_parser.parse_args()

    if not hasattr(args, "func"):
        config_parser.print_help()
        exit(1)

    # Run
    args.func(args)


if __name__ == "__main__":
    main()
```

accelerate-main/src/accelerate/test_utils/scripts/test_script.py

```
#!/usr/bin/env python

# Copyright 2021 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import contextlib
import io

import torch
from torch.utils.data import DataLoader

from accelerate import Accelerator
from accelerate.data_loader import prepare_data_loader
from accelerate.state import AcceleratorState
from accelerate.test_utils import RegressionDataset, RegressionModel, are_the_same_tensors
from accelerate.utils import (
    DistributedType,
    gather,
    is_bf16_available,
    is_torch_version,
    set_seed,
    synchronize_rng_states,
)


def print_main(state):
    print(f"Printing from the main process {state.process_index}")


def print_local_main(state):
    print(f"Printing from the local main process {state.local_process_index}")


def print_last(state):
    print(f"Printing from the last process {state.process_index}")


def print_on(state, process_idx):
    print(f"Printing from process {process_idx}: {state.process_index}")


def process_execution_check():
    accelerator = Accelerator()
    num_processes = accelerator.num_processes
    with accelerator.main_process_first():
        idx = torch.tensor(accelerator.process_index).to(accelerator.device)
    idxs = accelerator.gather(idx)
    if num_processes > 1:
        assert idxs[0] == 0, "Main process was not first."

    # Test the decorators
    f = io.StringIO()
    with contextlib.redirect_stdout(f):
        accelerator.on_main_process(print_main)(accelerator.state)
    result = f.getvalue().rstrip()
    if accelerator.is_main_process:
        assert result == "Printing from the main process 0", f"{result} != Printing from the main proces
    else:
        assert f.getvalue().rstrip() == "", f'{result} != ""'
    f.truncate(0)
    f.seek(0)

    with contextlib.redirect_stdout(f):
        accelerator.on_local_main_process(print_local_main)(accelerator.state)
    if accelerator.is_local_main_process:
        assert f.getvalue().rstrip() == "Printing from the local main process 0"
    else:
        assert f.getvalue().rstrip() == ""
    f.truncate(0)
    f.seek(0)

    with contextlib.redirect_stdout(f):
        accelerator.on_last_process(print_last)(accelerator.state)
    if accelerator.is_last_process:
        assert f.getvalue().rstrip() == f"Printing from the last process {accelerator.state.num_processe
    else:
        assert f.getvalue().rstrip() == ""
    f.truncate(0)
    f.seek(0)

    for process_idx in range(num_processes):
        with contextlib.redirect_stdout(f):
            accelerator.on_process(print_on, process_index=process_idx)(accelerator.state, process_idx)
        if accelerator.process_index == process_idx:
            assert f.getvalue().rstrip() == f"Printing from process {process_idx}: {accelerator.process_
        else:
            assert f.getvalue().rstrip() == ""
        f.truncate(0)
        f.seek(0)


def init_state_check():
    # Test we can instantiate this twice in a row.
    state = AcceleratorState()
    if state.local_process_index == 0:
        print("Testing, testing. 1, 2, 3.")
    print(state)


def rng_sync_check():
    state = AcceleratorState()
    synchronize_rng_states(["torch"])
    assert are_the_same_tensors(torch.get_rng_state()), "RNG states improperly synchronized on CPU."
    if state.distributed_type == DistributedType.MULTI_GPU:
        synchronize_rng_states(["cuda"])
        assert are_the_same_tensors(torch.cuda.get_rng_state()), "RNG states improperly synchronized on
    generator = torch.Generator()
    synchronize_rng_states(["generator"], generator=generator)
    assert are_the_same_tensors(generator.get_state()), "RNG states improperly synchronized in generator

    if state.local_process_index == 0:
        print("All rng are properly synched.")


def dl_preparation_check():
    state = AcceleratorState()
    length = 32 * state.num_processes

    dl = DataLoader(range(length), batch_size=8)
    dl = prepare_data_loader(dl, state.device, state.num_processes, state.process_index, put_on_device=T
    result = []
    for batch in dl:
        result.append(gather(batch))
    result = torch.cat(result)

    print(state.process_index, result, type(dl))
    assert torch.equal(result.cpu(), torch.arange(0, length).long()), "Wrong non-shuffled dataloader res

    dl = DataLoader(range(length), batch_size=8)
    dl = prepare_data_loader(
        dl,
        state.device,
        state.num_processes,
        state.process_index,
        put_on_device=True,
        split_batches=True,
    )
    result = []
    for batch in dl:
        result.append(gather(batch))
    result = torch.cat(result)
    assert torch.equal(result.cpu(), torch.arange(0, length).long()), "Wrong non-shuffled dataloader res

    if state.process_index == 0:
        print("Non-shuffled dataloader passing.")

    dl = DataLoader(range(length), batch_size=8, shuffle=True)
    dl = prepare_data_loader(dl, state.device, state.num_processes, state.process_index, put_on_device=T
    result = []
    for batch in dl:
        result.append(gather(batch))
    result = torch.cat(result).tolist()
    result.sort()
    assert result == list(range(length)), "Wrong shuffled dataloader result."

    dl = DataLoader(range(length), batch_size=8, shuffle=True)
    dl = prepare_data_loader(
        dl,
        state.device,
        state.num_processes,
        state.process_index,
        put_on_device=True,
        split_batches=True,
    )
    result = []
    for batch in dl:
        result.append(gather(batch))
    result = torch.cat(result).tolist()
    result.sort()
    assert result == list(range(length)), "Wrong shuffled dataloader result."

    if state.local_process_index == 0:
        print("Shuffled dataloader passing.")


def central_dl_preparation_check():
    state = AcceleratorState()
    length = 32 * state.num_processes

    dl = DataLoader(range(length), batch_size=8)
    dl = prepare_data_loader(
        dl, state.device, state.num_processes, state.process_index, put_on_device=True, dispatch_batches
    )
    result = []
    for batch in dl:
        result.append(gather(batch))
    result = torch.cat(result)
    assert torch.equal(result.cpu(), torch.arange(0, length).long()), "Wrong non-shuffled dataloader res

    dl = DataLoader(range(length), batch_size=8)
    dl = prepare_data_loader(
        dl,
        state.device,
        state.num_processes,
        state.process_index,
        put_on_device=True,
        split_batches=True,
        dispatch_batches=True,
    )
    result = []
    for batch in dl:
        result.append(gather(batch))
    result = torch.cat(result)
    assert torch.equal(result.cpu(), torch.arange(0, length).long()), "Wrong non-shuffled dataloader res

    if state.process_index == 0:
        print("Non-shuffled central dataloader passing.")

    dl = DataLoader(range(length), batch_size=8, shuffle=True)
    dl = prepare_data_loader(
        dl, state.device, state.num_processes, state.process_index, put_on_device=True, dispatch_batches
    )
    result = []
    for batch in dl:
        result.append(gather(batch))
    result = torch.cat(result).tolist()
    result.sort()
    assert result == list(range(length)), "Wrong shuffled dataloader result."

    dl = DataLoader(range(length), batch_size=8, shuffle=True)
    dl = prepare_data_loader(
        dl,
        state.device,
        state.num_processes,
        state.process_index,
        put_on_device=True,
        split_batches=True,
        dispatch_batches=True,
    )
    result = []
    for batch in dl:
        result.append(gather(batch))
    result = torch.cat(result).tolist()
    result.sort()
    assert result == list(range(length)), "Wrong shuffled dataloader result."

    if state.local_process_index == 0:
        print("Shuffled central dataloader passing.")


def mock_training(length, batch_size, generator):
    set_seed(42)
    generator.manual_seed(42)
    train_set = RegressionDataset(length=length)
    train_dl = DataLoader(train_set, batch_size=batch_size, shuffle=True, generator=generator)
    model = RegressionModel()
    optimizer = torch.optim.SGD(model.parameters(), lr=0.1)
    for epoch in range(3):
        for batch in train_dl:
            model.zero_grad()
            output = model(batch["x"])
            loss = torch.nn.functional.mse_loss(output, batch["y"])
            loss.backward()
            optimizer.step()
    return train_set, model


def training_check():
    state = AcceleratorState()
    generator = torch.Generator()
    batch_size = 8
    length = batch_size * 4 * state.num_processes

    train_set, old_model = mock_training(length, batch_size * state.num_processes, generator)
    assert are_the_same_tensors(old_model.a), "Did not obtain the same model on both processes."
    assert are_the_same_tensors(old_model.b), "Did not obtain the same model on both processes."

    accelerator = Accelerator()
    train_dl = DataLoader(train_set, batch_size=batch_size, shuffle=True, generator=generator)
    model = RegressionModel()
    optimizer = torch.optim.SGD(model.parameters(), lr=0.1)

    train_dl, model, optimizer = accelerator.prepare(train_dl, model, optimizer)
```

```
set_seed(42)
generator.manual_seed(42)
for epoch in range(3):
    for batch in train_dl:
        model.zero_grad()
        output = model(batch["x"])
        loss = torch.nn.functional.mse_loss(output, batch["y"])
        accelerator.backward(loss)
        optimizer.step()

model = accelerator.unwrap_model(model).cpu()
assert torch.allclose(old_model.a, model.a), "Did not obtain the same model on CPU or distributed tr
assert torch.allclose(old_model.b, model.b), "Did not obtain the same model on CPU or distributed tr

accelerator.print("Training yielded the same results on one CPU or distributed setup with no batch s

accelerator = Accelerator(split_batches=True)
train_dl = DataLoader(train_set, batch_size=batch_size * state.num_processes, shuffle=True, generato
model = RegressionModel()
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)

train_dl, model, optimizer = accelerator.prepare(train_dl, model, optimizer)
set_seed(42)
generator.manual_seed(42)
for _ in range(3):
    for batch in train_dl:
        model.zero_grad()
        output = model(batch["x"])
        loss = torch.nn.functional.mse_loss(output, batch["y"])
        accelerator.backward(loss)
        optimizer.step()

model = accelerator.unwrap_model(model).cpu()
assert torch.allclose(old_model.a, model.a), "Did not obtain the same model on CPU or distributed tr
assert torch.allclose(old_model.b, model.b), "Did not obtain the same model on CPU or distributed tr

accelerator.print("Training yielded the same results on one CPU or distributes setup with batch spli

if torch.cuda.is_available():
    # Mostly a test that FP16 doesn't crash as the operation inside the model is not converted to FP
    print("FP16 training check.")
    AcceleratorState._reset_state()
    accelerator = Accelerator(mixed_precision="fp16")
    train_dl = DataLoader(train_set, batch_size=batch_size, shuffle=True, generator=generator)
    model = RegressionModel()
    optimizer = torch.optim.SGD(model.parameters(), lr=0.1)

    train_dl, model, optimizer = accelerator.prepare(train_dl, model, optimizer)
    set_seed(42)
    generator.manual_seed(42)
    for _ in range(3):
        for batch in train_dl:
            model.zero_grad()
            output = model(batch["x"])
            loss = torch.nn.functional.mse_loss(output, batch["y"])
            accelerator.backward(loss)
            optimizer.step()

    model = accelerator.unwrap_model(model).cpu()
    assert torch.allclose(old_model.a, model.a), "Did not obtain the same model on CPU or distribute
    assert torch.allclose(old_model.b, model.b), "Did not obtain the same model on CPU or distribute

# BF16 support is only for CPU + TPU, and some GPU
if is_bf16_available():
    # Mostly a test that BF16 doesn't crash as the operation inside the model is not converted to BF
    print("BF16 training check.")
    AcceleratorState._reset_state()
    accelerator = Accelerator(mixed_precision="bf16")
    train_dl = DataLoader(train_set, batch_size=batch_size, shuffle=True, generator=generator)
    model = RegressionModel()
    optimizer = torch.optim.SGD(model.parameters(), lr=0.1)
    train_dl, model, optimizer = accelerator.prepare(train_dl, model, optimizer)
```

```
        set_seed(42)
        generator.manual_seed(42)
        for _ in range(3):
            for batch in train_dl:
                model.zero_grad()
                output = model(batch["x"])
                loss = torch.nn.functional.mse_loss(output, batch["y"])
                accelerator.backward(loss)
                optimizer.step()

        model = accelerator.unwrap_model(model).cpu()
        assert torch.allclose(old_model.a, model.a), "Did not obtain the same model on CPU or distribute
        assert torch.allclose(old_model.b, model.b), "Did not obtain the same model on CPU or distribute


def main():
    accelerator = Accelerator()
    state = accelerator.state
    if state.local_process_index == 0:
        print("**Initialization**")
    init_state_check()
    if state.local_process_index == 0:
        print("\n**Test process execution**")
    process_execution_check()

    if state.local_process_index == 0:
        print("\n**Test random number generator synchronization**")
    rng_sync_check()

    if state.local_process_index == 0:
        print("\n**DataLoader integration test**")
    dl_preparation_check()
    if state.distributed_type != DistributedType.TPU and is_torch_version(">=", "1.8.0"):
        central_dl_preparation_check()

    # Trainings are not exactly the same in DeepSpeed and CPU mode
    if state.distributed_type == DistributedType.DEEPSPEED:
        return

    if state.local_process_index == 0:
        print("\n**Training integration test**")
    training_check()


def _mp_fn(index):
    # For xla_spawn (TPUs)
    main()


if __name__ == "__main__":
    main()
```

accelerate-main/src/accelerate/test_utils/scripts/test_distributed_data_loop.py

```
#!/usr/bin/env python

# Copyright 2021 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.


import warnings
from typing import List
from unittest.mock import Mock

import torch
from torch.utils.data import DataLoader, IterableDataset, TensorDataset

from accelerate.accelerator import Accelerator
from accelerate.utils.dataclasses import DistributedType


class DummyIterableDataset(IterableDataset):
    def __init__(self, data):
        self.data = data

    def __iter__(self):
        for element in self.data:
            yield element


def create_accelerator(even_batches=True):
    accelerator = Accelerator(even_batches=even_batches)
    assert accelerator.num_processes == 2, "this script expects that two GPUs are available"
    return accelerator


def create_dataloader(accelerator: Accelerator, dataset_size: int, batch_size: int, iterable: bool = Fal
    """
    Create a simple DataLoader to use during the test cases
    """
    if iterable:
        dataset = DummyIterableDataset(torch.as_tensor(range(dataset_size)))
    else:
        dataset = TensorDataset(torch.as_tensor(range(dataset_size)))

    dl = DataLoader(dataset, batch_size=batch_size)
    dl = accelerator.prepare(dl)

    return dl


def verify_dataloader_batch_sizes(
    accelerator: Accelerator,
    dataset_size: int,
    batch_size: int,
    process_0_expected_batch_sizes: List[int],
    process_1_expected_batch_sizes: List[int],
):
    """
    A helper function for verifying the batch sizes coming from a prepared dataloader in each process
    """
    dl = create_dataloader(accelerator=accelerator, dataset_size=dataset_size, batch_size=batch_size)

    batch_sizes = [len(batch[0]) for batch in dl]

    if accelerator.process_index == 0:
        assert batch_sizes == process_0_expected_batch_sizes
    elif accelerator.process_index == 1:
        assert batch_sizes == process_1_expected_batch_sizes


def test_default_ensures_even_batch_sizes():
    accelerator = create_accelerator()

    # without padding, we would expect a different number of batches
    verify_dataloader_batch_sizes(
        accelerator,
        dataset_size=3,
        batch_size=1,
        process_0_expected_batch_sizes=[1, 1],
        process_1_expected_batch_sizes=[1, 1],
    )

    # without padding, we would expect the same number of batches, but different sizes
    verify_dataloader_batch_sizes(
        accelerator,
        dataset_size=7,
        batch_size=2,
        process_0_expected_batch_sizes=[2, 2],
        process_1_expected_batch_sizes=[2, 2],
    )


def test_can_disable_even_batches():
    accelerator = create_accelerator(even_batches=False)

    verify_dataloader_batch_sizes(
        accelerator,
        dataset_size=3,
        batch_size=1,
        process_0_expected_batch_sizes=[1, 1],
        process_1_expected_batch_sizes=[1],
    )

    verify_dataloader_batch_sizes(
        accelerator,
        dataset_size=7,
        batch_size=2,
        process_0_expected_batch_sizes=[2, 2],
        process_1_expected_batch_sizes=[2, 1],
    )


def test_can_join_uneven_inputs():
    accelerator = create_accelerator(even_batches=False)

    model = torch.nn.Linear(1, 1)
    ddp_model = accelerator.prepare(model)

    dl = create_dataloader(accelerator, dataset_size=3, batch_size=1)

    batch_idxs = []
    with accelerator.join_uneven_inputs([ddp_model]):
        for batch_idx, batch in enumerate(dl):
            output = ddp_model(batch[0].float())
            loss = output.sum()
            loss.backward()
            batch_idxs.append(batch_idx)

    accelerator.wait_for_everyone()
    if accelerator.process_index == 0:
        assert batch_idxs == [0, 1]
    elif accelerator.process_index == 1:
        assert batch_idxs == [0]


def test_join_raises_warning_for_non_ddp_distributed(accelerator):
    with warnings.catch_warnings(record=True) as w:
        with accelerator.join_uneven_inputs([Mock()]):
            pass

        assert issubclass(w[-1].category, UserWarning)
        assert "only supported for multi-GPU" in str(w[-1].message)


def test_join_can_override_even_batches():
    default_even_batches = True
    overridden_even_batches = False
    accelerator = create_accelerator(even_batches=default_even_batches)
    model = torch.nn.Linear(1, 1)
    ddp_model = accelerator.prepare(model)
    train_dl = create_dataloader(accelerator, dataset_size=3, batch_size=1)
    valid_dl = create_dataloader(accelerator, dataset_size=3, batch_size=1)

    with accelerator.join_uneven_inputs([ddp_model], even_batches=overridden_even_batches):
        train_dl_overridden_value = train_dl.batch_sampler.even_batches
        valid_dl_overridden_value = valid_dl.batch_sampler.even_batches

    assert train_dl_overridden_value == overridden_even_batches
    assert valid_dl_overridden_value == overridden_even_batches
    assert train_dl.batch_sampler.even_batches == default_even_batches
    assert valid_dl.batch_sampler.even_batches == default_even_batches


def test_join_can_override_for_mixed_type_dataloaders():
    default_even_batches = True
    overridden_even_batches = False
    accelerator = create_accelerator(even_batches=default_even_batches)
    model = torch.nn.Linear(1, 1)
    ddp_model = accelerator.prepare(model)
    create_dataloader(accelerator, dataset_size=3, batch_size=1, iterable=True)
    batch_dl = create_dataloader(accelerator, dataset_size=3, batch_size=1)

    with warnings.catch_warnings():
        warnings.filterwarnings("ignore")
        try:
            with accelerator.join_uneven_inputs([ddp_model], even_batches=overridden_even_batches):
                batch_dl_overridden_value = batch_dl.batch_sampler.even_batches
        except AttributeError:
            # ensure attribute error is not raised when processing iterable dl
            raise AssertionError

    assert batch_dl_overridden_value == overridden_even_batches
    assert batch_dl.batch_sampler.even_batches == default_even_batches


def test_join_raises_warning_for_iterable_when_overriding_even_batches():
    accelerator = create_accelerator()
    model = torch.nn.Linear(1, 1)
    ddp_model = accelerator.prepare(model)
    create_dataloader(accelerator, dataset_size=3, batch_size=1, iterable=True)

    with warnings.catch_warnings(record=True) as w:
        with accelerator.join_uneven_inputs([ddp_model], even_batches=False):
            pass

        assert issubclass(w[-1].category, UserWarning)
        assert "only supported for map-style datasets" in str(w[-1].message)


def main():
    accelerator = create_accelerator()
    accelerator.print("Test that even_batches variable ensures uniform batches across processes")
```

```python
    test_default_ensures_even_batch_sizes()

    accelerator.print("Run tests with even_batches disabled")
    test_can_disable_even_batches()

    accelerator.print("Test joining uneven inputs")
    test_can_join_uneven_inputs()

    accelerator.print("Test overriding even_batches when joining uneven inputs")
    test_join_can_override_even_batches()

    accelerator.print("Test overriding even_batches for mixed dataloader types")
    test_join_can_override_for_mixed_type_dataloaders()

    accelerator.print("Test overriding even_batches raises a warning for iterable dataloaders")
    test_join_raises_warning_for_iterable_when_overriding_even_batches()

    accelerator.print("Test join with non DDP distributed raises warning")
    original_state = accelerator.state.distributed_type
    accelerator.state.distributed_type = DistributedType.FSDP
    test_join_raises_warning_for_non_ddp_distributed(accelerator)
    accelerator.state.distributed_type = original_state


if __name__ == "__main__":
    main()
```

accelerate-main/src/accelerate/test_utils/scripts/test_sync.py

```
# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

from copy import deepcopy

import torch
import torch.nn.functional as F
from torch.optim import AdamW
from torch.optim.lr_scheduler import LambdaLR
from torch.utils.data import DataLoader

from accelerate.accelerator import Accelerator
from accelerate.state import GradientState
from accelerate.test_utils import RegressionDataset, RegressionModel
from accelerate.utils import DistributedType, set_seed


def check_model_parameters(model_a, model_b, did_step, iteration):
    for param, grad_param in zip(model_a.parameters(), model_b.parameters()):
        if not param.requires_grad:
            continue
        if not did_step:
            # Grads should not be in sync
            assert (
                torch.allclose(param.grad, grad_param.grad) is False
            ), f"Gradients in sync when they should not be at iteration {iteration}:\nmodel_a grad ({par
        else:
            # Grads should be in sync
            assert (
                torch.allclose(param.grad, grad_param.grad) is True
            ), f"Gradients not in sync when they should be at iteration {iteration}:\nmodel_a grad ({par


def step_model(model, input, target, accelerator, do_backward=True):
    model.train()
    output = model(input)
    loss = F.mse_loss(output, target.to(output.device))
    if not do_backward:
        loss /= accelerator.gradient_accumulation_steps
        loss.backward()
    else:
        accelerator.backward(loss)


def get_training_setup(accelerator, sched=False):
    "Returns everything needed to perform basic training"
    set_seed(42)
    model = RegressionModel()
    ddp_model = deepcopy(model)
    dset = RegressionDataset(length=80)
    dataloader = DataLoader(dset, batch_size=16)
    model.to(accelerator.device)
    if sched:
        opt = AdamW(params=model.parameters(), lr=1e-3)
        ddp_opt = AdamW(params=ddp_model.parameters(), lr=1e-3)
        sched = LambdaLR(opt, lr_lambda=lambda epoch: epoch**0.65)
        ddp_sched = LambdaLR(ddp_opt, lr_lambda=lambda epoch: epoch**0.65)
    # Make a copy of `model`
    if sched:
        ddp_model, ddp_opt, ddp_sched, dataloader = accelerator.prepare(ddp_model, ddp_opt, ddp_sched, d
    else:
        ddp_model, dataloader = accelerator.prepare(ddp_model, dataloader)
    if sched:
        return (model, opt, sched, dataloader, ddp_model, ddp_opt, ddp_sched)
    return model, ddp_model, dataloader


def test_noop_sync(accelerator):
    # Test when on a single CPU or GPU that the context manager does nothing
    model, ddp_model, dataloader = get_training_setup(accelerator)
    # Use a single batch
    ddp_input, ddp_target = next(iter(dataloader)).values()
    for iteration in range(3):
        # Gather the distributed inputs and targs for the base model
        input, target = accelerator.gather((ddp_input, ddp_target))
        input, target = input.to(accelerator.device), target.to(accelerator.device)
        # Perform our initial ground truth step in non "DDP"
        step_model(model, input, target, accelerator)
        # Do "gradient accumulation" (noop)
        if iteration % 2 == 0:
            # Accumulate grads locally
            with accelerator.no_sync(ddp_model):
                step_model(ddp_model, ddp_input, ddp_target, accelerator)
        else:
            # Sync grads
            step_model(ddp_model, ddp_input, ddp_target, accelerator)

        # Since `no_sync` is a noop, `ddp_model` and `model` grads should always be in sync
        check_model_parameters(model, ddp_model, True, iteration)
        for param, ddp_param in zip(model.parameters(), ddp_model.parameters()):
            if not param.requires_grad:
                continue
            assert torch.allclose(
                param.grad, ddp_param.grad
            ), f"Gradients not in sync when they should be:\nModel grad ({param.grad}) != DDP grad ({ddp

        # Shuffle ddp_input on each iteration
        torch.manual_seed(1337 + iteration)
        ddp_input = ddp_input[torch.randperm(len(ddp_input))]


def test_distributed_sync(accelerator):
    # Test on distributed setup that context manager behaves properly
    model, ddp_model, dataloader = get_training_setup(accelerator)
    # Use a single batch
    ddp_input, ddp_target = next(iter(dataloader)).values()
    for iteration in range(3):
        # Gather the distributed inputs and targs for the base model
        input, target = accelerator.gather((ddp_input, ddp_target))
        input, target = input.to(accelerator.device), target.to(accelerator.device)
        # Perform our initial ground truth step in non "DDP"
        step_model(model, input, target, accelerator)
        # Do "gradient accumulation" (noop)
        if iteration % 2 == 0:
            # Accumulate grads locally
            with accelerator.no_sync(ddp_model):
                step_model(ddp_model, ddp_input, ddp_target, accelerator)
        else:
            # Sync grads
            step_model(ddp_model, ddp_input, ddp_target, accelerator)

        # DDP model and model should only be in sync when not (iteration % 2 == 0)
        for param, ddp_param in zip(model.parameters(), ddp_model.parameters()):
            if not param.requires_grad:
                continue
            if iteration % 2 == 0:
                # Grads should not be in sync
                assert (
                    torch.allclose(param.grad, ddp_param.grad) is False
                ), f"Gradients in sync when they should not be:\nModel grad ({param.grad}) == DDP grad (
            else:
                # Grads should be in sync
                assert (
                    torch.allclose(param.grad, ddp_param.grad) is True
                ), f"Gradients not in sync when they should be:\nModel grad ({param.grad}) != DDP grad (

        # Shuffle ddp_input on each iteration
        torch.manual_seed(1337 + iteration)
        ddp_input = ddp_input[torch.randperm(len(ddp_input))]


def test_gradient_accumulation(split_batches=False, dispatch_batches=False):
    accelerator = Accelerator(
        split_batches=split_batches, dispatch_batches=dispatch_batches, gradient_accumulation_steps=2
    )
    # Test that context manager behaves properly
    model, ddp_model, dataloader = get_training_setup(accelerator)
    for iteration, batch in enumerate(dataloader):
        ddp_input, ddp_target = batch.values()
        # Gather the distributed inputs and targs for the base model
        input, target = accelerator.gather((ddp_input, ddp_target))
        input, target = input.to(accelerator.device), target.to(accelerator.device)
        # Perform our initial ground truth step in non "DDP"
        step_model(model, input, target, accelerator, False)
        # Do "gradient accumulation" (noop)
        with accelerator.accumulate(ddp_model):
            step_model(ddp_model, ddp_input, ddp_target, accelerator)

        # DDP model and model should only be in sync when not (iteration % 2 == 0)
        for param, ddp_param in zip(model.parameters(), ddp_model.parameters()):
            if not param.requires_grad:
                continue
            if ((iteration + 1) % 2 == 0) or (iteration == len(dataloader) - 1):
                # Grads should be in sync
                assert (
                    torch.allclose(param.grad, ddp_param.grad) is True
                ), f"Gradients not in sync when they should be at iteration {iteration}:\nModel grad ({p
            else:
                # Grads should not be in sync
                assert (
                    torch.allclose(param.grad, ddp_param.grad) is False
                ), f"Gradients in sync when they should not be at iteration {iteration}:\nModel grad ({p

        # Shuffle ddp_input on each iteration
        torch.manual_seed(1337 + iteration)
        ddp_input = ddp_input[torch.randperm(len(ddp_input))]
    GradientState._reset_state()


def test_gradient_accumulation_with_opt_and_scheduler(split_batches=False, dispatch_batches=False):
    accelerator = Accelerator(
        split_batches=split_batches, dispatch_batches=dispatch_batches, gradient_accumulation_steps=2
    )
    # Test that context manager behaves properly
    model, opt, sched, dataloader, ddp_model, ddp_opt, ddp_sched = get_training_setup(accelerator, True)
    for iteration, batch in enumerate(dataloader):
        ddp_input, ddp_target = batch.values()
        # Gather the distributed inputs and targs for the base model
        input, target = accelerator.gather((ddp_input, ddp_target))
        input, target = input.to(accelerator.device), target.to(accelerator.device)
        # Perform our initial ground truth step in non "DDP"
        model.train()
        ddp_model.train()
        step_model(model, input, target, accelerator, False)
        opt.step()

        if ((iteration + 1) % 2 == 0) or ((iteration + 1) == len(dataloader)):
            if split_batches:
                sched.step()
            else:
                for _ in range(accelerator.num_processes):
```

```
                sched.step()
        opt.zero_grad()
        # Perform gradient accumulation under wrapper
        with accelerator.accumulate(ddp_model):
            step_model(ddp_model, ddp_input, ddp_target, accelerator)
            ddp_opt.step()
            ddp_sched.step()
            ddp_opt.zero_grad()

        # Learning rates should be the same
        assert (
            opt.param_groups[0]["lr"] == ddp_opt.param_groups[0]["lr"]
        ), f'Learning rates found in each optimizer did not align\nopt: {opt.param_groups[0]["lr"]}\nDDP
        did_step = (((iteration + 1) % 2) == 0) or ((iteration + 1) == len(dataloader))
        if accelerator.num_processes > 1:
            check_model_parameters(model, ddp_model, did_step, iteration)
        # Shuffle ddp_input on each iteration
        torch.manual_seed(1337 + iteration)
    GradientState._reset_state()


def test_dataloader_break():
    accelerator = Accelerator()

    first_dset = RegressionDataset(length=80)
    first_dataloader = DataLoader(first_dset, batch_size=16)
    second_dset = RegressionDataset(length=96)
    second_dataloader = DataLoader(second_dset, batch_size=16)
    first_dataloader, second_dataloader = accelerator.prepare(first_dataloader, second_dataloader)
    for iteration, _ in enumerate(first_dataloader):
        # Will be True except if we are on the last batch
        if iteration < len(first_dataloader) - 1:
            assert id(accelerator.gradient_state.active_dataloader) == id(first_dataloader)
            if iteration == 1:
                for batch_num, _ in enumerate(second_dataloader):
                    if batch_num < len(second_dataloader) - 1:
                        assert id(accelerator.gradient_state.active_dataloader) == id(
                            second_dataloader
                        ), f"First dataloader: {id(first_dataloader)}\nSecond dataloader: {id(second_dat
                    else:
                        assert id(accelerator.gradient_state.active_dataloader) == id(
                            first_dataloader
                        ), f"First dataloader: {id(first_dataloader)}\nSecond dataloader: {id(second_dat
        else:
            assert accelerator.gradient_state.active_dataloader is None


def main():
    accelerator = Accelerator()
    state = accelerator.state
    if state.local_process_index == 0:
        print("**Test `accumulate` gradient accumulation with dataloader break**")
    test_dataloader_break()
    if state.distributed_type == DistributedType.NO:
        if state.local_process_index == 0:
            print("**Test NOOP `no_sync` context manager**")
        test_noop_sync(accelerator)
    if state.distributed_type in (DistributedType.MULTI_GPU, DistributedType.MULTI_CPU):
        if state.local_process_index == 0:
            print("**Test Distributed `no_sync` context manager**")
        test_distributed_sync(accelerator)
    if state.distributed_type == DistributedType.MULTI_GPU:
        for split_batch in [True, False]:
            for dispatch_batches in [True, False]:
                if state.local_process_index == 0:
                    print(
                        "**Test `accumulate` gradient accumulation, ",
                        f"`split_batches={split_batch}` and `dispatch_batches={dispatch_batches}`**",
                    )
                test_gradient_accumulation(split_batch, dispatch_batches)
    if state.local_process_index == 0:
        print(
            "**Test `accumulate` gradient accumulation with optimizer and scheduler, ",
            "`split_batches=False`, `dispatch_batches=False`**",
        )
    test_gradient_accumulation_with_opt_and_scheduler()
    if state.distributed_type == DistributedType.MULTI_GPU:
        for split_batch in [True, False]:
            for dispatch_batches in [True, False]:
                if not split_batch and not dispatch_batches:
                    continue
                if state.local_process_index == 0:
                    print(
                        "**Test `accumulate` gradient accumulation with optimizer and scheduler, ",
                        f"`split_batches={split_batch}` and `dispatch_batches={dispatch_batches}`**",
                    )
                test_gradient_accumulation_with_opt_and_scheduler(split_batch, dispatch_batches)


def _mp_fn(index):
    # For xla_spawn (TPUs)
    main()


if __name__ == "__main__":
    main()
```

accelerate-main/src/accelerate/test_utils/scripts/test_ops.py

```
#!/usr/bin/env python

# Copyright 2023 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import torch

from accelerate import PartialState
from accelerate.utils.operations import broadcast, gather, pad_across_processes, reduce


def create_tensor(state):
    return (torch.arange(state.num_processes) + 1.0 + (state.num_processes * state.process_index)).to(st


def test_gather(state):
    tensor = create_tensor(state)
    gathered_tensor = gather(tensor)
    assert gathered_tensor.tolist() == list(range(1, state.num_processes**2 + 1))


def test_broadcast(state):
    tensor = create_tensor(state)
    broadcasted_tensor = broadcast(tensor)
    assert broadcasted_tensor.shape == torch.Size([state.num_processes])
    assert broadcasted_tensor.tolist() == list(range(1, state.num_processes + 1))


def test_pad_across_processes(state):
    # We need to pad the tensor with one more element if we are the main process
    # to ensure that we can pad
    if state.is_main_process:
        tensor = torch.arange(state.num_processes + 1).to(state.device)
    else:
        tensor = torch.arange(state.num_processes).to(state.device)
    padded_tensor = pad_across_processes(tensor)
    assert padded_tensor.shape == torch.Size([state.num_processes + 1])
    if not state.is_main_process:
        assert padded_tensor.tolist() == list(range(0, state.num_processes)) + [0]


def test_reduce_sum(state):
    # For now runs on only two processes
    if state.num_processes != 2:
        return
    tensor = create_tensor(state)
    reduced_tensor = reduce(tensor, "sum")
    truth_tensor = torch.tensor([4.0, 6]).to(state.device)
    assert torch.allclose(reduced_tensor, truth_tensor), f"{reduced_tensor} != {truth_tensor}"


def test_reduce_mean(state):
    # For now runs on only two processes
    if state.num_processes != 2:
        return
    tensor = create_tensor(state)
    reduced_tensor = reduce(tensor, "mean")
    truth_tensor = torch.tensor([2.0, 3]).to(state.device)
    assert torch.allclose(reduced_tensor, truth_tensor), f"{reduced_tensor} != {truth_tensor}"


def _mp_fn(index):
    # For xla_spawn (TPUs)
    main()


def main():
    state = PartialState()
    state.print("testing gather")
    test_gather(state)
    state.print("testing broadcast")
    test_broadcast(state)
    state.print("testing pad_across_processes")
    test_pad_across_processes(state)
    state.print("testing reduce_sum")
    test_reduce_sum(state)
    state.print("testing reduce_mean")
    test_reduce_mean(state)


if __name__ == "__main__":
    main()
```

accelerate-main/tests/test_cli.py

```python
# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import inspect
import os
import unittest
from pathlib import Path

import torch

import accelerate
from accelerate.test_utils import execute_subprocess_async
from accelerate.test_utils.testing import run_command


class AccelerateLauncherTester(unittest.TestCase):
    """
    Test case for verifying the `accelerate launch` CLI operates correctly.
    If a `default_config.yaml` file is located in the cache it will temporarily move it
    for the duration of the tests.
    """

    mod_file = inspect.getfile(accelerate.test_utils)
    test_file_path = os.path.sep.join(mod_file.split(os.path.sep)[:-1] + ["scripts", "test_cli.py"])

    base_cmd = ["accelerate", "launch"]
    config_folder = Path.home() / ".cache/huggingface/accelerate"
    config_file = "default_config.yaml"
    config_path = config_folder / config_file
    changed_path = config_folder / "_default_config.yaml"

    test_config_path = Path("tests/test_configs")

    @classmethod
    def setUpClass(cls):
        if cls.config_path.is_file():
            cls.config_path.rename(cls.changed_path)

    @classmethod
    def tearDownClass(cls):
        if cls.changed_path.is_file():
            cls.changed_path.rename(cls.config_path)

    def test_no_config(self):
        cmd = self.base_cmd
        if torch.cuda.is_available() and (torch.cuda.device_count() > 1):
            cmd += ["--multi_gpu"]
        execute_subprocess_async(cmd + [self.test_file_path], env=os.environ.copy())

    def test_config_compatibility(self):
        for config in sorted(self.test_config_path.glob("**/*.yaml")):
            with self.subTest(config_file=config):
                execute_subprocess_async(
                    self.base_cmd + ["--config_file", str(config), self.test_file_path], env=os.environ.
                )

    def test_accelerate_test(self):
        execute_subprocess_async(["accelerate", "test"], env=os.environ.copy())


class TpuConfigTester(unittest.TestCase):
    """
    Test case for verifying the `accelerate tpu-config` CLI passes the right `gcloud` command.
    """

    tpu_name = "test-tpu"
    tpu_zone = "us-central1-a"
    command = "ls"
    cmd = ["accelerate", "tpu-config"]
    base_output = "cd /usr/share"
    command_file = "tests/test_samples/test_command_file.sh"
    gcloud = "Running gcloud compute tpus tpu-vm ssh"

    @staticmethod
    def clean_output(output):
        return "".join(output).rstrip()

    def test_base(self):
        output = run_command(
            self.cmd
            + ["--command", self.command, "--tpu_zone", self.tpu_zone, "--tpu_name", self.tpu_name, "--d
            return_stdout=True,
        )
        self.assertEqual(
            self.clean_output(output),
            f"{self.gcloud} test-tpu --zone us-central1-a --command {self.base_output}; ls --worker all"
        )

    def test_base_backward_compatibility(self):
        output = run_command(
            self.cmd
            + [
                "--config_file",
                "tests/test_configs/0_12_0.yaml",
                "--command",
                self.command,
                "--tpu_zone",
                self.tpu_zone,
                "--tpu_name",
                self.tpu_name,
                "--debug",
            ],
            return_stdout=True,
        )
        self.assertEqual(
            self.clean_output(output),
            f"{self.gcloud} test-tpu --zone us-central1-a --command {self.base_output}; ls --worker all"
        )

    def test_with_config_file(self):
        output = run_command(
            self.cmd + ["--config_file", "tests/test_configs/latest.yaml", "--debug"], return_stdout=Tru
        )
        self.assertEqual(
            self.clean_output(output),
            f'{self.gcloud} test-tpu --zone us-central1-a --command {self.base_output}; echo "hello worl
        )

    def test_with_config_file_and_command(self):
        output = run_command(
            self.cmd + ["--config_file", "tests/test_configs/latest.yaml", "--command", self.command, "-
            return_stdout=True,
        )
        self.assertEqual(
            self.clean_output(output),
            f"{self.gcloud} test-tpu --zone us-central1-a --command {self.base_output}; ls --worker all"
        )

    def test_with_config_file_and_multiple_command(self):
        output = run_command(
            self.cmd
            + [
                "--config_file",
                "tests/test_configs/latest.yaml",
                "--command",
                self.command,
                "--command",
                'echo "Hello World"',
                "--debug",
            ],
            return_stdout=True,
        )
        self.assertEqual(
            self.clean_output(output),
            f'{self.gcloud} test-tpu --zone us-central1-a --command {self.base_output}; ls; echo "Hello
        )

    def test_with_config_file_and_command_file(self):
        output = run_command(
            self.cmd
            + ["--config_file", "tests/test_configs/latest.yaml", "--command_file", self.command_file, "
            return_stdout=True,
        )
        self.assertEqual(
            self.clean_output(output),
            f'{self.gcloud} test-tpu --zone us-central1-a --command {self.base_output}; echo "hello worl
        )

    def test_with_config_file_and_command_file_backward_compatibility(self):
        output = run_command(
            self.cmd
            + [
                "--config_file",
                "tests/test_configs/0_12_0.yaml",
                "--command_file",
                self.command_file,
                "--tpu_zone",
                self.tpu_zone,
                "--tpu_name",
                self.tpu_name,
                "--debug",
            ],
            return_stdout=True,
        )
        self.assertEqual(
            self.clean_output(output),
            f'{self.gcloud} test-tpu --zone us-central1-a --command {self.base_output}; echo "hello worl
        )

    def test_accelerate_install(self):
        output = run_command(
            self.cmd + ["--config_file", "tests/test_configs/latest.yaml", "--install_accelerate", "--de
            return_stdout=True,
        )
        self.assertEqual(
            self.clean_output(output),
            f'{self.gcloud} test-tpu --zone us-central1-a --command {self.base_output}; pip install acce
        )

    def test_accelerate_install_version(self):
        output = run_command(
            self.cmd
            + [
                "--config_file",
                "tests/test_configs/latest.yaml",
                "--install_accelerate",
                "--accelerate_version",
                "12.0.0",
                "--debug",
            ],
            return_stdout=True,
        )
        self.assertEqual(
            self.clean_output(output),
            f'{self.gcloud} test-tpu --zone us-central1-a --command {self.base_output}; pip install acce
        )
```

accelerate-main/src/accelerate/commands/config/__init__.py

```python
#!/usr/bin/env python

# Copyright 2021 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import argparse

from .config import config_command_parser
from .config_args import default_config_file, load_config_from_file  # noqa: F401
from .default import default_command_parser
from .update import update_command_parser


def get_config_parser(subparsers=None):
    parent_parser = argparse.ArgumentParser(add_help=False, allow_abbrev=False)
    # The main config parser
    config_parser = config_command_parser(subparsers)
    # The subparser to add commands to
    subcommands = config_parser.add_subparsers(title="subcommands", dest="subcommand")

    # Then add other parsers with the parent parser
    default_command_parser(subcommands, parents=[parent_parser])
    update_command_parser(subcommands, parents=[parent_parser])

    return config_parser


def main():
    config_parser = get_config_parser()
    args = config_parser.parse_args()

    if not hasattr(args, "func"):
        config_parser.print_help()
        exit(1)

    # Run
    args.func(args)


if __name__ == "__main__":
    main()
```

accelerate-main/src/accelerate/test_utils/scripts/external_deps/test_checkpointing.py

```
# coding=utf-8
# Copyright 2022 The HuggingFace Inc. team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
import argparse
import json
import os

import evaluate
import torch
from datasets import load_dataset
from torch.optim import AdamW
from torch.utils.data import DataLoader
from transformers import AutoModelForSequenceClassification, AutoTokenizer, get_linear_schedule_with_war

from accelerate import Accelerator, DistributedType
from accelerate.utils.deepspeed import DummyOptim, DummyScheduler


MAX_GPU_BATCH_SIZE = 16
EVAL_BATCH_SIZE = 32


def get_dataloaders(accelerator: Accelerator, batch_size: int = 16, model_name: str = "bert-base-cased")
    """
    Creates a set of `DataLoader`s for the `glue` dataset.

    Args:
        accelerator (`Accelerator`):
            An `Accelerator` object
        batch_size (`int`, *optional*):
            The batch size for the train and validation DataLoaders.
        model_name (`str`, *optional*):
    """
    tokenizer = AutoTokenizer.from_pretrained(model_name)
    datasets = load_dataset("glue", "mrpc")

    def tokenize_function(examples):
        # max_length=None => use the model max length (it's actually the default)
        outputs = tokenizer(examples["sentence1"], examples["sentence2"], truncation=True, max_length=No
        return outputs

    # Apply the method we just defined to all the examples in all the splits of the dataset
    tokenized_datasets = datasets.map(
        tokenize_function, batched=True, remove_columns=["idx", "sentence1", "sentence2"], load_from_cac
    )

    # We also rename the 'label' column to 'labels' which is the expected name for labels by the models
    # transformers library
    tokenized_datasets = tokenized_datasets.rename_column("label", "labels")

    def collate_fn(examples):
        # On TPU it's best to pad everything to the same length or training will be very slow.
        if accelerator.distributed_type == DistributedType.TPU:
            return tokenizer.pad(examples, padding="max_length", max_length=128, return_tensors="pt")
        return tokenizer.pad(examples, padding="longest", return_tensors="pt")
    # Instantiate dataloaders.
    train_dataloader = DataLoader(
        tokenized_datasets["train"], shuffle=True, collate_fn=collate_fn, batch_size=batch_size
    )
    eval_dataloader = DataLoader(
        tokenized_datasets["validation"], shuffle=False, collate_fn=collate_fn, batch_size=EVAL_BATCH_SI
    )

    return train_dataloader, eval_dataloader


def evaluation_loop(accelerator, model, eval_dataloader, metric):
    model.eval()
    samples_seen = 0
    for step, batch in enumerate(eval_dataloader):
        # We could avoid this line since we set the accelerator with `device_placement=True`.
        batch.to(accelerator.device)
        with torch.no_grad():
            outputs = model(**batch)
        predictions = outputs.logits.argmax(dim=-1)
        # It is slightly faster to call this once, than multiple times
        predictions, references = accelerator.gather(
            (predictions, batch["labels"])
        )  # If we are in a multiprocess environment, the last batch has duplicates
        if accelerator.use_distributed:
            if step == len(eval_dataloader) - 1:
                predictions = predictions[: len(eval_dataloader.dataset) - samples_seen]
                references = references[: len(eval_dataloader.dataset) - samples_seen]
            else:
                samples_seen += references.shape[0]
        metric.add_batch(
            predictions=predictions,
            references=references,
        )

    eval_metric = metric.compute()
    return eval_metric["accuracy"]


def training_function(config, args):
    # Initialize accelerator
    accelerator = Accelerator()

    # Sample hyper-parameters for learning rate, batch size, seed and a few other HPs
    lr = config["lr"]
    num_epochs = int(config["num_epochs"])
    seed = int(config["seed"])
    batch_size = int(config["batch_size"])
    model_name = args.model_name_or_path

    set_seed(seed)
    train_dataloader, eval_dataloader = get_dataloaders(accelerator, batch_size, model_name)

    # Instantiate the model (we build the model here so that the seed also control new weights initializ
    model = AutoModelForSequenceClassification.from_pretrained(model_name, return_dict=True)

    # Instantiate optimizer
    optimizer_cls = (
        AdamW
        if accelerator.state.deepspeed_plugin is None
        or "optimizer" not in accelerator.state.deepspeed_plugin.deepspeed_config
        else DummyOptim
    )
    optimizer = optimizer_cls(params=model.parameters(), lr=lr)

    if accelerator.state.deepspeed_plugin is not None:
        gradient_accumulation_steps = accelerator.state.deepspeed_plugin.deepspeed_config[
            "gradient_accumulation_steps"
        ]
    else:
        gradient_accumulation_steps = 1
    max_training_steps = (len(train_dataloader) * num_epochs) // gradient_accumulation_steps
```

```
# Instantiate scheduler
if (
    accelerator.state.deepspeed_plugin is None
    or "scheduler" not in accelerator.state.deepspeed_plugin.deepspeed_config
):
    lr_scheduler = get_linear_schedule_with_warmup(
        optimizer=optimizer,
        num_warmup_steps=0,
        num_training_steps=max_training_steps,
    )
else:
    lr_scheduler = DummyScheduler(optimizer, total_num_steps=max_training_steps, warmup_num_steps=0)

# Prepare everything
# There is no specific order to remember, we just need to unpack the objects in the same order we ga
# prepare method.
model, optimizer, train_dataloader, eval_dataloader, lr_scheduler = accelerator.prepare(
    model, optimizer, train_dataloader, eval_dataloader, lr_scheduler
)

# We need to keep track of how many total steps we have iterated over
overall_step = 0
# We also need to keep track of the stating epoch so files are named properly
starting_epoch = 0
metric = evaluate.load("glue", "mrpc")
ending_epoch = num_epochs

if args.partial_train_epoch is not None:
    ending_epoch = args.partial_train_epoch

if args.resume_from_checkpoint:
    accelerator.load_state(args.resume_from_checkpoint)
    epoch_string = args.resume_from_checkpoint.split("epoch_")[1]
    state_epoch_num = ""
    for char in epoch_string:
        if char.isdigit():
            state_epoch_num += char
        else:
            break
    starting_epoch = int(state_epoch_num) + 1
    accuracy = evaluation_loop(accelerator, model, eval_dataloader, metric)
    accelerator.print("resumed checkpoint performance:", accuracy)
    accelerator.print("resumed checkpoint's scheduler's lr:", lr_scheduler.get_lr()[0])
    accelerator.print("resumed optimizers's lr:", optimizer.param_groups[0]["lr"])
    with open(os.path.join(args.output_dir, f"state_{starting_epoch-1}.json"), "r") as f:
        resumed_state = json.load(f)
        assert resumed_state["accuracy"] == accuracy, "Accuracy mismatch, loading from checkpoint fa
        assert (
            resumed_state["lr"] == lr_scheduler.get_lr()[0]
        ), "Scheduler learning rate mismatch, loading from checkpoint failed"
        assert (
            resumed_state["optimizer_lr"] == optimizer.param_groups[0]["lr"]
        ), "Optimizer learning rate mismatch, loading from checkpoint failed"
        assert resumed_state["epoch"] == starting_epoch - 1, "Epoch mismatch, loading from checkpoin
        return

# Now we train the model
state = {}
for epoch in range(starting_epoch, ending_epoch):
    model.train()
    for step, batch in enumerate(train_dataloader):
        outputs = model(**batch)
        loss = outputs.loss
        loss = loss / gradient_accumulation_steps
        accelerator.backward(loss)
        if step % gradient_accumulation_steps == 0:
            optimizer.step()
            lr_scheduler.step()
            optimizer.zero_grad()

        overall_step += 1
    output_dir = f"epoch_{epoch}"
```

```
        output_dir = os.path.join(args.output_dir, output_dir)
        accelerator.save_state(output_dir)
        accuracy = evaluation_loop(accelerator, model, eval_dataloader, metric)
        state["accuracy"] = accuracy
        state["lr"] = lr_scheduler.get_lr()[0]
        state["optimizer_lr"] = optimizer.param_groups[0]["lr"]
        state["epoch"] = epoch
        state["step"] = overall_step
        accelerator.print(f"epoch {epoch}:", state)

        accelerator.wait_for_everyone()
        if accelerator.is_main_process:
            with open(os.path.join(args.output_dir, f"state_{epoch}.json"), "w") as f:
                json.dump(state, f)


def main():
    parser = argparse.ArgumentParser(description="Simple example of training script tracking peak GPU me
    parser.add_argument(
        "--model_name_or_path",
        type=str,
        default="bert-base-cased",
        help="Path to pretrained model or model identifier from huggingface.co/models.",
        required=False,
    )
    parser.add_argument(
        "--output_dir",
        type=str,
        default=".",
        help="Optional save directory where all checkpoint folders will be stored. Default is the curren
    )
    parser.add_argument(
        "--resume_from_checkpoint",
        type=str,
        default=None,
        help="If the training should continue from a checkpoint folder.",
    )
    parser.add_argument(
        "--partial_train_epoch",
        type=int,
        default=None,
        help="If passed, the training will stop after this number of epochs.",
    )
    parser.add_argument(
        "--num_epochs",
        type=int,
        default=2,
        help="Number of train epochs.",
    )
    args = parser.parse_args()
    config = {"lr": 2e-5, "num_epochs": args.num_epochs, "seed": 42, "batch_size": 16}

    training_function(config, args)


if __name__ == "__main__":
    main()
```

accelerate-main/src/accelerate/test_utils/scripts/external_deps/test_performance.py

```
# coding=utf-8
# Copyright 2022 The HuggingFace Inc. team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
import argparse
import json
import os

import evaluate
import torch
from datasets import load_dataset
from torch.optim import AdamW
from torch.utils.data import DataLoader
from transformers import AutoModelForSequenceClassification, AutoTokenizer, get_linear_schedule_with_war

from accelerate import Accelerator, DistributedType
from accelerate.utils.deepspeed import DummyOptim, DummyScheduler


MAX_GPU_BATCH_SIZE = 16
EVAL_BATCH_SIZE = 32


def get_dataloaders(accelerator: Accelerator, batch_size: int = 16, model_name: str = "bert-base-cased")
    """
    Creates a set of `DataLoader`s for the `glue` dataset.

    Args:
        accelerator (`Accelerator`):
            An `Accelerator` object
        batch_size (`int`, *optional*):
            The batch size for the train and validation DataLoaders.
        model_name (`str`, *optional*):
    """
    tokenizer = AutoTokenizer.from_pretrained(model_name)
    datasets = load_dataset("glue", "mrpc")

    def tokenize_function(examples):
        # max_length=None => use the model max length (it's actually the default)
        outputs = tokenizer(examples["sentence1"], examples["sentence2"], truncation=True, max_length=No
        return outputs

    # Apply the method we just defined to all the examples in all the splits of the dataset
    tokenized_datasets = datasets.map(
        tokenize_function, batched=True, remove_columns=["idx", "sentence1", "sentence2"], load_from_cac
    )

    # We also rename the 'label' column to 'labels' which is the expected name for labels by the models
    # transformers library
    tokenized_datasets = tokenized_datasets.rename_column("label", "labels")

    def collate_fn(examples):
        # On TPU it's best to pad everything to the same length or training will be very slow.
        if accelerator.distributed_type == DistributedType.TPU:
            return tokenizer.pad(examples, padding="max_length", max_length=128, return_tensors="pt")
        return tokenizer.pad(examples, padding="longest", return_tensors="pt")
    # Instantiate dataloaders.
    train_dataloader = DataLoader(
        tokenized_datasets["train"], shuffle=True, collate_fn=collate_fn, batch_size=batch_size
    )
    eval_dataloader = DataLoader(
        tokenized_datasets["validation"], shuffle=False, collate_fn=collate_fn, batch_size=EVAL_BATCH_SI
    )

    return train_dataloader, eval_dataloader


def training_function(config, args):
    # Initialize accelerator
    accelerator = Accelerator()

    # Sample hyper-parameters for learning rate, batch size, seed and a few other HPs
    lr = config["lr"]
    num_epochs = int(config["num_epochs"])
    seed = int(config["seed"])
    batch_size = int(config["batch_size"])
    model_name = args.model_name_or_path

    set_seed(seed)
    train_dataloader, eval_dataloader = get_dataloaders(accelerator, batch_size, model_name)

    # Instantiate the model (we build the model here so that the seed also control new weights initializ
    model = AutoModelForSequenceClassification.from_pretrained(model_name, return_dict=True)

    # Instantiate optimizer
    optimizer_cls = (
        AdamW
        if accelerator.state.deepspeed_plugin is None
        or "optimizer" not in accelerator.state.deepspeed_plugin.deepspeed_config
        else DummyOptim
    )
    optimizer = optimizer_cls(params=model.parameters(), lr=lr)

    if accelerator.state.deepspeed_plugin is not None:
        gradient_accumulation_steps = accelerator.state.deepspeed_plugin.deepspeed_config[
            "gradient_accumulation_steps"
        ]
    else:
        gradient_accumulation_steps = 1
    max_training_steps = (len(train_dataloader) * num_epochs) // gradient_accumulation_steps

    # Instantiate scheduler
    if (
        accelerator.state.deepspeed_plugin is None
        or "scheduler" not in accelerator.state.deepspeed_plugin.deepspeed_config
    ):
        lr_scheduler = get_linear_schedule_with_warmup(
            optimizer=optimizer,
            num_warmup_steps=0,
            num_training_steps=max_training_steps,
        )
    else:
        lr_scheduler = DummyScheduler(optimizer, total_num_steps=max_training_steps, warmup_num_steps=0)

    # Prepare everything
    # There is no specific order to remember, we just need to unpack the objects in the same order we ga
    # prepare method.
    model, optimizer, train_dataloader, eval_dataloader, lr_scheduler = accelerator.prepare(
        model, optimizer, train_dataloader, eval_dataloader, lr_scheduler
    )

    # We need to keep track of how many total steps we have iterated over
    overall_step = 0
    # We also need to keep track of the stating epoch so files are named properly
    starting_epoch = 0

    # Now we train the model
    metric = evaluate.load("glue", "mrpc")
    best_performance = 0
    performance_metric = {}
    for epoch in range(starting_epoch, num_epochs):
        model.train()
        for step, batch in enumerate(train_dataloader):
            outputs = model(**batch)
            loss = outputs.loss
            loss = loss / gradient_accumulation_steps
            accelerator.backward(loss)
            if step % gradient_accumulation_steps == 0:
                optimizer.step()
                lr_scheduler.step()
                optimizer.zero_grad()

            overall_step += 1

        model.eval()
        samples_seen = 0
        for step, batch in enumerate(eval_dataloader):
            # We could avoid this line since we set the accelerator with `device_placement=True`.
            batch.to(accelerator.device)
            with torch.no_grad():
                outputs = model(**batch)
            predictions = outputs.logits.argmax(dim=-1)
            # It is slightly faster to call this once, than multiple times
            predictions, references = accelerator.gather(
                (predictions, batch["labels"])
            )  # If we are in a multiprocess environment, the last batch has duplicates
            if accelerator.use_distributed:
                if step == len(eval_dataloader) - 1:
                    predictions = predictions[: len(eval_dataloader.dataset) - samples_seen]
                    references = references[: len(eval_dataloader.dataset) - samples_seen]
                else:
                    samples_seen += references.shape[0]
            metric.add_batch(
                predictions=predictions,
                references=references,
            )

        eval_metric = metric.compute()
        # Use accelerator.print to print only on the main process.
        accelerator.print(f"epoch {epoch}:", eval_metric)
        performance_metric[f"epoch-{epoch}"] = eval_metric["accuracy"]

        if best_performance < eval_metric["accuracy"]:
            best_performance = eval_metric["accuracy"]

    if args.performance_lower_bound is not None:
        assert (
            args.performance_lower_bound <= best_performance
        ), f"Best performance metric {best_performance} is lower than the lower bound {args.performance_

    accelerator.wait_for_everyone()
    if accelerator.is_main_process:
        with open(os.path.join(args.output_dir, "all_results.json"), "w") as f:
            json.dump(performance_metric, f)


def main():
    parser = argparse.ArgumentParser(description="Simple example of training script tracking peak GPU me
    parser.add_argument(
        "--model_name_or_path",
        type=str,
        default="bert-base-cased",
        help="Path to pretrained model or model identifier from huggingface.co/models.",
        required=False,
    )
    parser.add_argument(
        "--output_dir",
        type=str,
        default=".",
        help="Optional save directory where all checkpoint folders will be stored. Default is the curren
    )
    parser.add_argument(
        "--performance_lower_bound",
        type=float,
        default=None,
        help="Optional lower bound for the performance metric. If set, the training will throw error whe
    )
    parser.add_argument(
        "--num_epochs",
        type=int,
        default=3,
        help="Number of train epochs.",
    )
    args = parser.parse_args()
    config = {"lr": 2e-5, "num_epochs": args.num_epochs, "seed": 42, "batch_size": 16}
    training_function(config, args)


if __name__ == "__main__":
    main()
```

accelerate-main/src/accelerate/test_utils/scripts/external_deps/test_peak_memory_usage.py

```
# coding=utf-8
# Copyright 2022 The HuggingFace Inc. team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
import argparse
import gc
import json
import os

import torch
from datasets import load_dataset
from torch.optim import AdamW
from torch.utils.data import DataLoader
from transformers import AutoModelForSequenceClassification, AutoTokenizer, get_linear_schedule_with_war

from accelerate import Accelerator, DistributedType
from accelerate.utils.deepspeed import DummyOptim, DummyScheduler


MAX_GPU_BATCH_SIZE = 16
EVAL_BATCH_SIZE = 32


# Converting Bytes to Megabytes
def b2mb(x):
    return int(x / 2**20)


# This context manager is used to track the peak memory usage of the process
class TorchTracemalloc:
    def __enter__(self):
        gc.collect()
        torch.cuda.empty_cache()
        torch.cuda.reset_max_memory_allocated()  # reset the peak gauge to zero
        self.begin = torch.cuda.memory_allocated()
        return self

    def __exit__(self, *exc):
        gc.collect()
        torch.cuda.empty_cache()
        self.end = torch.cuda.memory_allocated()
        self.peak = torch.cuda.max_memory_allocated()
        self.used = b2mb(self.end - self.begin)
        self.peaked = b2mb(self.peak - self.begin)
        # print(f"delta used/peak {self.used:4d}/{self.peaked:4d}")


def get_dataloaders(
    accelerator: Accelerator,
    batch_size: int = 16,
    model_name: str = "bert-base-cased",
    n_train: int = 320,
    n_val: int = 160,
):
    """
    Creates a set of `DataLoader`s for the `glue` dataset.
    Args:
        accelerator (`Accelerator`):
            An `Accelerator` object
        batch_size (`int`, *optional*):
            The batch size for the train and validation DataLoaders.
        model_name (`str`, *optional*):
            The name of the model to use.
        n_train (`int`, *optional*):
            The number of training examples to use.
        n_val (`int`, *optional*):
            The number of validation examples to use.
    """
    tokenizer = AutoTokenizer.from_pretrained(model_name)
    datasets = load_dataset(
        "glue", "mrpc", split={"train": f"train[:{n_train}]", "validation": f"validation[:{n_val}]"}
    )

    def tokenize_function(examples):
        # max_length=None => use the model max length (it's actually the default)
        outputs = tokenizer(examples["sentence1"], examples["sentence2"], truncation=True, max_length=No
        return outputs

    # Apply the method we just defined to all the examples in all the splits of the dataset
    tokenized_datasets = datasets.map(
        tokenize_function, batched=True, remove_columns=["idx", "sentence1", "sentence2"], load_from_cac
    )

    # We also rename the 'label' column to 'labels' which is the expected name for labels by the models
    # transformers library
    tokenized_datasets = tokenized_datasets.rename_column("label", "labels")

    def collate_fn(examples):
        # On TPU it's best to pad everything to the same length or training will be very slow.
        if accelerator.distributed_type == DistributedType.TPU:
            return tokenizer.pad(examples, padding="max_length", max_length=128, return_tensors="pt")
        return tokenizer.pad(examples, padding="longest", return_tensors="pt")

    # Instantiate dataloaders.
    train_dataloader = DataLoader(
        tokenized_datasets["train"], shuffle=True, collate_fn=collate_fn, batch_size=batch_size
    )
    eval_dataloader = DataLoader(
        tokenized_datasets["validation"], shuffle=False, collate_fn=collate_fn, batch_size=EVAL_BATCH_SI
    )

    return train_dataloader, eval_dataloader


def training_function(config, args):
    # Initialize accelerator
    accelerator = Accelerator()

    # Sample hyper-parameters for learning rate, batch size, seed and a few other HPs
    lr = config["lr"]
    num_epochs = int(config["num_epochs"])
    seed = int(config["seed"])
    batch_size = int(config["batch_size"])
    model_name = args.model_name_or_path

    set_seed(seed)
    train_dataloader, eval_dataloader = get_dataloaders(accelerator, batch_size, model_name, args.n_trai

    # Instantiate the model (we build the model here so that the seed also control new weights initializ
    model = AutoModelForSequenceClassification.from_pretrained(model_name, return_dict=True)

    # Instantiate optimizer
    optimizer_cls = (
        AdamW
        if accelerator.state.deepspeed_plugin is None
        or "optimizer" not in accelerator.state.deepspeed_plugin.deepspeed_config
        else DummyOptim
    )
    optimizer = optimizer_cls(params=model.parameters(), lr=lr)

    if accelerator.state.deepspeed_plugin is not None:
        gradient_accumulation_steps = accelerator.state.deepspeed_plugin.deepspeed_config[
            "gradient_accumulation_steps"
        ]
    else:
        gradient_accumulation_steps = 1
    max_training_steps = (len(train_dataloader) * num_epochs) // gradient_accumulation_steps

    # Instantiate scheduler
    if (
        accelerator.state.deepspeed_plugin is None
        or "scheduler" not in accelerator.state.deepspeed_plugin.deepspeed_config
    ):
        lr_scheduler = get_linear_schedule_with_warmup(
            optimizer=optimizer,
            num_warmup_steps=0,
            num_training_steps=max_training_steps,
        )
    else:
        lr_scheduler = DummyScheduler(optimizer, total_num_steps=max_training_steps, warmup_num_steps=0)

    # Prepare everything
    # There is no specific order to remember, we just need to unpack the objects in the same order we ga
    # prepare method.
    model, optimizer, train_dataloader, eval_dataloader, lr_scheduler = accelerator.prepare(
        model, optimizer, train_dataloader, eval_dataloader, lr_scheduler
    )

    # We need to keep track of how many total steps we have iterated over
    overall_step = 0
    # We also need to keep track of the stating epoch so files are named properly
    starting_epoch = 0

    # Now we train the model
    train_total_peak_memory = {}
    for epoch in range(starting_epoch, num_epochs):
        with TorchTracemalloc() as tracemalloc:
            model.train()
            for step, batch in enumerate(train_dataloader):
                outputs = model(**batch)
                loss = outputs.loss
                loss = loss / gradient_accumulation_steps
                accelerator.backward(loss)
                if step % gradient_accumulation_steps == 0:
                    optimizer.step()
                    lr_scheduler.step()
                    optimizer.zero_grad()

                overall_step += 1

        # Printing the GPU memory usage details such as allocated memory, peak memory, and total memory
        accelerator.print("Memory before entering the train : {}".format(b2mb(tracemalloc.begin)))
        accelerator.print("Memory consumed at the end of the train (end-begin): {}".format(tracemalloc.u
        accelerator.print("Peak Memory consumed during the train (max-begin): {}".format(tracemalloc.pea
        accelerator.print(
            "Total Peak Memory consumed during the train (max): {}".format(
                tracemalloc.peaked + b2mb(tracemalloc.begin)
            )
        )
        train_total_peak_memory[f"epoch-{epoch}"] = tracemalloc.peaked + b2mb(tracemalloc.begin)
        if args.peak_memory_upper_bound is not None:
            assert (
                train_total_peak_memory[f"epoch-{epoch}"] <= args.peak_memory_upper_bound
            ), "Peak memory usage exceeded the upper bound"

    accelerator.wait_for_everyone()
    if accelerator.is_main_process:
        with open(os.path.join(args.output_dir, "peak_memory_utilization.json"), "w") as f:
            json.dump(train_total_peak_memory, f)
def main():
    parser = argparse.ArgumentParser(description="Simple example of training script tracking peak GPU me
    parser.add_argument(
        "--model_name_or_path",
        type=str,
        default="bert-base-cased",
        help="Path to pretrained model or model identifier from huggingface.co/models.",
        required=False,
    )
    parser.add_argument(
        "--output_dir",
        type=str,
        default=".",
        help="Optional save directory where all checkpoint folders will be stored. Default is the curren
    )
    parser.add_argument(
        "--peak_memory_upper_bound",
        type=float,
        default=None,
        help="The upper bound of peak memory usage in MB. If set, the training will throw an error if th
    )
    parser.add_argument(
        "--n_train",
        type=int,
        default=320,
        help="Number of training examples to use.",
    )
    parser.add_argument(
        "--n_val",
        type=int,
        default=160,
        help="Number of validation examples to use.",
    )
    parser.add_argument(
        "--num_epochs",
        type=int,
        default=1,
        help="Number of train epochs.",
    )
    args = parser.parse_args()
    config = {"lr": 2e-5, "num_epochs": args.num_epochs, "seed": 42, "batch_size": 16}
    training_function(config, args)


if __name__ == "__main__":
    main()
```

accelerate-main/tests/test_metrics.py

```python
# Copyright 2021 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import inspect
import os
import unittest

import torch

import accelerate
from accelerate import debug_launcher
from accelerate.test_utils import (
    execute_subprocess_async,
    require_cpu,
    require_huggingface_suite,
    require_multi_gpu,
    require_single_gpu,
    require_torch_min_version,
)
from accelerate.utils import get_launch_prefix, patch_environment


@require_huggingface_suite
@require_torch_min_version(version="1.8.0")
class MetricTester(unittest.TestCase):
    def setUp(self):
        mod_file = inspect.getfile(accelerate.test_utils)
        self.test_file_path = os.path.sep.join(
            mod_file.split(os.path.sep)[:-1] + ["scripts", "external_deps", "test_metrics.py"]
        )

        from accelerate.test_utils.scripts.external_deps import test_metrics  # noqa: F401

        self.test_metrics = test_metrics

    @require_cpu
    def test_metric_cpu_noop(self):
        debug_launcher(self.test_metrics.main, num_processes=1)

    @require_cpu
    def test_metric_cpu_multi(self):
        debug_launcher(self.test_metrics.main)

    @require_single_gpu
    def test_metric_gpu(self):
        self.test_metrics.main()

    @require_multi_gpu
    def test_metric_gpu_multi(self):
        print(f"Found {torch.cuda.device_count()} devices.")
        cmd = get_launch_prefix() + [f"--nproc_per_node={torch.cuda.device_count()}", self.test_file_pat
        with patch_environment(omp_num_threads=1):
            execute_subprocess_async(cmd, env=os.environ.copy())
```

accelerate-main/src/accelerate/utils/torch_xla.py

~~~python
# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import subprocess
import sys

import pkg_resources


def install_xla(upgrade: bool = False):
    """
    Helper function to install appropriate xla wheels based on the `torch` version in Google Colaborator

    Args:
        upgrade (`bool`, *optional*, defaults to `False`):
            Whether to upgrade `torch` and install the latest `torch_xla` wheels.

    Example:

    ```python
    >>> from accelerate.utils import install_xla

    >>> install_xla(upgrade=True)
    ~~~
    """
    in_colab = False
    if "IPython" in sys.modules:
        in_colab = "google.colab" in str(sys.modules["IPython"].get_ipython())

    if in_colab:
        if upgrade:
            torch_install_cmd = ["pip", "install", "-U", "torch"]
            subprocess.run(torch_install_cmd, check=True)
        # get the current version of torch
        torch_version = pkg_resources.get_distribution("torch").version
        torch_version_trunc = torch_version[: torch_version.rindex(".")]
        xla_wheel = f"https://storage.googleapis.com/tpu-pytorch/wheels/colab/torch_xla-{torch_version_t
        xla_install_cmd = ["pip", "install", xla_wheel]
        subprocess.run(xla_install_cmd, check=True)
    else:
        raise RuntimeError("`install_xla` utility works only on google colab.")
```
```

accelerate-main/src/accelerate/utils/environment.py

```
# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import os
from distutils.util import strtobool


def get_int_from_env(env_keys, default):
    """Returns the first positive env value found in the `env_keys` list or the default."""
    for e in env_keys:
        val = int(os.environ.get(e, -1))
        if val >= 0:
            return val
    return default


def parse_flag_from_env(key, default=False):
    """Returns truthy value for `key` from the env if available else the default."""
    value = os.environ.get(key, str(default))
    return strtobool(value) == 1  # As its name indicates `strtobool` actually returns an int...


def parse_choice_from_env(key, default="no"):
    value = os.environ.get(key, str(default))
    return value
```

accelerate-main/src/accelerate/commands/config/__init__.py

```
#!/usr/bin/env python

# Copyright 2021 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import argparse

from .config import config_command_parser
from .config_args import default_config_file, load_config_from_file  # noqa: F401
from .default import default_command_parser
from .update import update_command_parser


def get_config_parser(subparsers=None):
    parent_parser = argparse.ArgumentParser(add_help=False, allow_abbrev=False)
    # The main config parser
    config_parser = config_command_parser(subparsers)
    # The subparser to add commands to
    subcommands = config_parser.add_subparsers(title="subcommands", dest="subcommand")

    # Then add other parsers with the parent parser
    default_command_parser(subcommands, parents=[parent_parser])
    update_command_parser(subcommands, parents=[parent_parser])

    return config_parser


def main():
    config_parser = get_config_parser()
    args = config_parser.parse_args()

    if not hasattr(args, "func"):
        config_parser.print_help()
        exit(1)

    # Run
    args.func(args)


if __name__ == "__main__":
    main()
```

accelerate-main/src/accelerate/utils/offload.py

```
# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import json
import os
from collections.abc import Mapping
from typing import Dict, List, Optional, Union

import numpy as np
import torch

from ..logging import get_logger
from .imports import is_safetensors_available


logger = get_logger(__name__)


def offload_weight(weight, weight_name, offload_folder, index=None):
    dtype = None
    # Check the string instead of the dtype to be compatible with versions of PyTorch that don't have bf
    if str(weight.dtype) == "torch.bfloat16":
        # Need to reinterpret the underlined data as int16 since NumPy does not handle bfloat16s.
        weight = weight.view(torch.int16)
        dtype = "bfloat16"
    array = weight.cpu().numpy()
    tensor_file = os.path.join(offload_folder, f"{weight_name}.dat")
    if index is not None:
        if dtype is None:
            dtype = str(array.dtype)
        index[weight_name] = {"dtype": dtype, "shape": list(array.shape)}
    if array.ndim == 0:
        array = array[None]
    file_array = np.memmap(tensor_file, dtype=array.dtype, mode="w+", shape=array.shape)
    file_array[:] = array[:]
    file_array.flush()
    return index


def load_offloaded_weight(weight_file, weight_info):
    shape = tuple(weight_info["shape"])
    if shape == ():
        # NumPy memory-mapped arrays can't have 0 dims so it was saved as 1d tensor
        shape = (1,)

    dtype = weight_info["dtype"]
    if dtype == "bfloat16":
        # NumPy does not support bfloat16 so this was saved as a int16
        dtype = "int16"

    weight = np.memmap(weight_file, dtype=dtype, shape=shape, mode="r")

    if len(weight_info["shape"]) == 0:
        weight = weight[0]
    weight = torch.tensor(weight)
    if weight_info["dtype"] == "bfloat16":
        weight = weight.view(torch.bfloat16)
    return weight


def save_offload_index(index, offload_folder):
    if index is None or len(index) == 0:
        # Nothing to save
        return

    offload_index_file = os.path.join(offload_folder, "index.json")
    if os.path.isfile(offload_index_file):
        with open(offload_index_file, "r", encoding="utf-8") as f:
            current_index = json.load(f)
    else:
        current_index = {}
    current_index.update(index)

    with open(offload_index_file, "w", encoding="utf-8") as f:
        json.dump(current_index, f, indent=2)


def offload_state_dict(save_dir: Union[str, os.PathLike], state_dict: Dict[str, torch.Tensor]):
    """
    Offload a state dict in a given folder.

    Args:
        save_dir (`str` or `os.PathLike`):
            The directory in which to offload the state dict.
        state_dict (`Dict[str, torch.Tensor]`):
            The dictionary of tensors to offload.
    """
    os.makedirs(save_dir, exist_ok=True)
    index = {}
    for name, parameter in state_dict.items():
        index = offload_weight(parameter, name, save_dir, index=index)

    # Update index
    save_offload_index(index, save_dir)


class PrefixedDataset(Mapping):
    """
    Will access keys in a given dataset by adding a prefix.

    Args:
        dataset (`Mapping`): Any map with string keys.
        prefix (`str`): A prefix to add when trying to access any element in the underlying dataset.
    """

    def __init__(self, dataset: Mapping, prefix: str):
        self.dataset = dataset
        self.prefix = prefix

    def __getitem__(self, key):
        return self.dataset[f"{self.prefix}{key}"]

    def __iter__(self):
        return iter([key for key in self.dataset if key.startswith(self.prefix)])

    def __len__(self):
        return len(self.dataset)


class OffloadedWeightsLoader(Mapping):
    """
    A collection that loads weights stored in a given state dict or memory-mapped on disk.

    Args:
        state_dict (`Dict[str, torch.Tensor]`, *optional*):
            A dictionary parameter name to tensor.
        save_folder (`str` or `os.PathLike`, *optional*):
            The directory in which the weights are stored (by `offload_state_dict` for instance).
        index (`Dict`, *optional*):
            A dictionary from weight name to their information (`dtype`/ `shape` or safetensors filename
            to the index saved in `save_folder`.
    """

    def __init__(
        self,
        state_dict: Dict[str, torch.Tensor] = None,
        save_folder: Optional[Union[str, os.PathLike]] = None,
        index: Mapping = None,
        device=None,
    ):
        if state_dict is None and save_folder is None:
            raise ValueError("Need either a `state_dict` or a `save_folder` containing offloaded weights

        self.state_dict = {} if state_dict is None else state_dict
        self.save_folder = save_folder
        if index is None and save_folder is not None:
            with open(os.path.join(save_folder, "index.json")) as f:
                index = json.load(f)
        self.index = {} if index is None else index
        self.all_keys = list(self.state_dict.keys())
        self.all_keys.extend([key for key in self.index if key not in self.all_keys])
        self.device = device

    def __getitem__(self, key: str):
        # State dict gets priority
        if key in self.state_dict:
            return self.state_dict[key]
        weight_info = self.index[key]
        if weight_info.get("safetensors_file") is not None:
            if not is_safetensors_available():
                raise ImportError("These offloaded weights require the use of safetensors: `pip install

            if "SAFETENSORS_FAST_GPU" not in os.environ:
                logger.info("Enabling fast loading with safetensors by setting `SAFETENSORS_FAST_GPU` to
                os.environ["SAFETENSORS_FAST_GPU"] = "1"

            from safetensors import safe_open

            device = "cpu" if self.device is None else self.device
            with safe_open(weight_info["safetensors_file"], framework="pt", device=device) as f:
                tensor = f.get_tensor(weight_info.get("weight_name", key))

            if "dtype" in weight_info:
                return tensor.to(getattr(torch, weight_info["dtype"]))
            else:
                return tensor

        weight_file = os.path.join(self.save_folder, f"{key}.dat")
        return load_offloaded_weight(weight_file, weight_info)

    def __iter__(self):
        return iter(self.all_keys)

    def __len__(self):
        return len(self.all_keys)


def extract_submodules_state_dict(state_dict: Dict[str, torch.Tensor], submodule_names: List[str]):
    """
    Extract the sub state-dict corresponding to a list of given submodules.

    Args:
        state_dict (`Dict[str, torch.Tensor]`): The state dict to extract from.
        submodule_names (`List[str]`): The list of submodule names we want to extract.
    """
    result = {}
    for module_name in submodule_names:
        # We want to catch module_name parameter (module_name.xxx) or potentially module_name, but not a
        # submodules that could being like module_name (transformers.h.1 and transformers.h.10 for insta
        result.update(
            {
```

```
            key: param
            for key, param in state_dict.items()
            if key == module_name or key.startswith(module_name + ".")
        }
    )
return result
```

accelerate-main/src/accelerate/utils/other.py

```
# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import os
from contextlib import contextmanager

import torch

from ..commands.config.default import write_basic_config  # noqa: F401
from ..state import PartialState
from .dataclasses import DistributedType
from .imports import is_deepspeed_available, is_tpu_available
from .transformer_engine import convert_model


if is_deepspeed_available():
    from deepspeed import DeepSpeedEngine

if is_tpu_available(check_device=False):
    import torch_xla.core.xla_model as xm


def extract_model_from_parallel(model, keep_fp32_wrapper: bool = True):
    """
    Extract a model from its distributed containers.

    Args:
        model (`torch.nn.Module`):
            The model to extract.
        keep_fp32_wrapper (`bool`, *optional*):
            Whether to remove mixed precision hooks from the model.

    Returns:
        `torch.nn.Module`: The extracted model.
    """
    options = (torch.nn.parallel.DistributedDataParallel, torch.nn.DataParallel)
    if is_deepspeed_available():
        options += (DeepSpeedEngine,)

    while isinstance(model, options):
        model = model.module

    if not keep_fp32_wrapper:
        forward = getattr(model, "forward")
        original_forward = model.__dict__.pop("_original_forward", None)
        if original_forward is not None:
            while hasattr(forward, "__wrapped__"):
                forward = forward.__wrapped__
                if forward == original_forward:
                    break
            model.forward = forward
        if getattr(model, "_converted_to_transformer_engine", False):
            convert_model(model, to_transformer_engine=False)
    return model


def wait_for_everyone():
```

~~~
    """
    Introduces a blocking point in the script, making sure all processes have reached this point before

    <Tip warning={true}>

    Make sure all processes will reach this instruction otherwise one of your processes will hang foreve

    </Tip>
    """
    PartialState().wait_for_everyone()


def save(obj, f):
    """
    Save the data to disk. Use in place of `torch.save()`.

    Args:
        obj: The data to save
        f: The file (or file-like object) to use to save the data
    """
    if PartialState().distributed_type == DistributedType.TPU:
        xm.save(obj, f)
    elif PartialState().local_process_index == 0:
        torch.save(obj, f)


@contextmanager
def patch_environment(**kwargs):
    """
    A context manager that will add each keyword argument passed to `os.environ` and remove them when ex

    Will convert the values in `kwargs` to strings and upper-case all the keys.

    Example:

    ```python
    >>> import os
    >>> from accelerate.utils import patch_environment

    >>> with patch_environment(FOO="bar"):
    ...     print(os.environ["FOO"])  # prints "bar"
    >>> print(os.environ["FOO"])  # raises KeyError
    ```
    """
    for key, value in kwargs.items():
        os.environ[key.upper()] = str(value)

    yield

    for key in kwargs:
        del os.environ[key.upper()]


def get_pretty_name(obj):
    """
    Gets a pretty name from `obj`.
    """
    if not hasattr(obj, "__qualname__") and not hasattr(obj, "__name__"):
        obj = getattr(obj, "__class__", obj)
    if hasattr(obj, "__qualname__"):
        return obj.__qualname__
    if hasattr(obj, "__name__"):
        return obj.__name__
    return str(obj)


def merge_dicts(source, destination):
    """
    Recursively merges two dictionaries.

    Args:
        source (`dict`): The dictionary to merge into `destination`.
~~~

```
    destination (`dict`): The dictionary to merge `source` into.
"""
for key, value in source.items():
    if isinstance(value, dict):
        node = destination.setdefault(key, {})
        merge_dicts(value, node)
    else:
        destination[key] = value

return destination
```

accelerate-main/src/accelerate/utils/rich.py

```
# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

from .imports import is_rich_available


if is_rich_available():
    from rich.traceback import install

    install(show_locals=False)

else:
    raise ModuleNotFoundError("To use the rich extension, install rich with `pip install rich`")
```

accelerate-main/src/accelerate/utils/operations.py

```
# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

"""
A set of basic tensor ops compatible with tpu, gpu, and multigpu
"""

import pickle
from functools import update_wrapper
from typing import Any, Mapping

import torch

from ..state import PartialState
from .constants import CUDA_DISTRIBUTED_TYPES
from .dataclasses import DistributedType, TensorInformation
from .imports import is_torch_distributed_available, is_tpu_available
from .versions import is_torch_version


if is_tpu_available(check_device=False):
    import torch_xla.core.xla_model as xm


if is_torch_distributed_available():
    from torch.distributed import ReduceOp


def is_torch_tensor(tensor):
    return isinstance(tensor, torch.Tensor)


def is_tensor_information(tensor_info):
    return isinstance(tensor_info, TensorInformation)


def honor_type(obj, generator):
    """
    Cast a generator to the same type as obj (list, tuple, or namedtuple)
    """
    try:
        return type(obj)(generator)
    except TypeError:
        # Some objects may not be able to instantiate from a generator directly
        return type(obj)(*list(generator))


def recursively_apply(func, data, *args, test_type=is_torch_tensor, error_on_other_type=False, **kwargs)
    """
    Recursively apply a function on a data structure that is a nested list/tuple/dictionary of a given b

    Args:
        func (`callable`):
            The function to recursively apply.
        data (nested list/tuple/dictionary of `main_type`):
            The data on which to apply `func`
        *args:
            Positional arguments that will be passed to `func` when applied on the unpacked data.
        main_type (`type`, *optional*, defaults to `torch.Tensor`):
            The base type of the objects to which apply `func`.
        error_on_other_type (`bool`, *optional*, defaults to `False`):
            Whether to return an error or not if after unpacking `data`, we get on an object that is not
            `main_type`. If `False`, the function will leave objects of types different than `main_type`
        **kwargs:
            Keyword arguments that will be passed to `func` when applied on the unpacked data.

    Returns:
        The same data structure as `data` with `func` applied to every object of type `main_type`.
    """
    if isinstance(data, (tuple, list)):
        return honor_type(
            data,
            (
                recursively_apply(
                    func, o, *args, test_type=test_type, error_on_other_type=error_on_other_type, **kwar
                )
                for o in data
            ),
        )
    elif isinstance(data, Mapping):
        return type(data)(
            {
                k: recursively_apply(
                    func, v, *args, test_type=test_type, error_on_other_type=error_on_other_type, **kwar
                )
                for k, v in data.items()
            }
        )
    elif test_type(data):
        return func(data, *args, **kwargs)
    elif error_on_other_type:
        raise TypeError(
            f"Can't apply {func.__name__} on object of type {type(data)}, only of nested list/tuple/dict
            f"that satisfy {test_type.__name__}."
        )
    return data


def send_to_device(tensor, device, non_blocking=False):
    """
    Recursively sends the elements in a nested list/tuple/dictionary of tensors to a given device.

    Args:
        tensor (nested list/tuple/dictionary of `torch.Tensor`):
            The data to send to a given device.
        device (`torch.device`):
            The device to send the data to.

    Returns:
        The same data structure as `tensor` with all tensors sent to the proper device.
    """

    def _send_to_device(t, device, non_blocking):
        try:
            return t.to(device, non_blocking=non_blocking)
        except TypeError:  # .to() doesn't accept non_blocking as kwarg
            return t.to(device)

    def _has_to_method(t):
        return hasattr(t, "to")

    return recursively_apply(_send_to_device, tensor, device, non_blocking, test_type=_has_to_method)


def get_data_structure(data):
    """
    Recursively gathers the information needed to rebuild a nested list/tuple/dictionary of tensors.

    Args:
        data (nested list/tuple/dictionary of `torch.Tensor`):
            The data to send to analyze.

    Returns:
        The same data structure as `data` with [`~utils.TensorInformation`] instead of tensors.
    """

    def _get_data_structure(tensor):
        return TensorInformation(shape=tensor.shape, dtype=tensor.dtype)

    return recursively_apply(_get_data_structure, data)


def initialize_tensors(data_structure):
    """
    Recursively initializes tensors from a nested list/tuple/dictionary of [`~utils.TensorInformation`].

    Returns:
        The same data structure as `data` with tensors instead of [`~utils.TensorInformation`].
    """

    def _initialize_tensor(tensor_info):
        return torch.empty(*tensor_info.shape, dtype=tensor_info.dtype)

    return recursively_apply(_initialize_tensor, data_structure, test_type=is_tensor_information)


def find_batch_size(data):
    """
    Recursively finds the batch size in a nested list/tuple/dictionary of lists of tensors.

    Args:
        data (nested list/tuple/dictionary of `torch.Tensor`): The data from which to find the batch siz

    Returns:
        `int`: The batch size.
    """
    if isinstance(data, (tuple, list)):
        return find_batch_size(data[0])
    elif isinstance(data, Mapping):
        for k in data.keys():
            return find_batch_size(data[k])
    elif not isinstance(data, torch.Tensor):
        raise TypeError(f"Can only find the batch size of tensors but got {type(data)}.")
    return data.shape[0]


def _tpu_gather(tensor, name="gather tensor"):
    if isinstance(tensor, (list, tuple)):
        return honor_type(tensor, (_tpu_gather(t, name=f"{name}_{i}") for i, t in enumerate(tensor)))
    elif isinstance(tensor, Mapping):
        return type(tensor)({k: _tpu_gather(v, name=f"{name}_{k}") for k, v in tensor.items()})
    elif not isinstance(tensor, torch.Tensor):
        raise TypeError(f"Can't gather the values of type {type(tensor)}, only of nested list/tuple/dict
    if tensor.ndim == 0:
        tensor = tensor.clone()[None]
    return xm.mesh_reduce(name, tensor, torch.cat)


def _gpu_gather(tensor):
    def _gpu_gather_one(tensor):
        if tensor.ndim == 0:
            tensor = tensor.clone()[None]
        output_tensors = [tensor.clone() for _ in range(torch.distributed.get_world_size())]
        torch.distributed.all_gather(output_tensors, tensor)
        return torch.cat(output_tensors, dim=0)

    return recursively_apply(_gpu_gather_one, tensor, error_on_other_type=True)


_cpu_gather = _gpu_gather
def gather(tensor):
    """
    Recursively gather tensor in a nested list/tuple/dictionary of tensors from all devices.

    Args:
        tensor (nested list/tuple/dictionary of `torch.Tensor`):
            The data to gather.

    Returns:
        The same data structure as `tensor` with all tensors sent to the proper device.
    """
    if PartialState().distributed_type == DistributedType.TPU:
        return _tpu_gather(tensor, name="accelerate.utils.gather")
    elif PartialState().distributed_type in CUDA_DISTRIBUTED_TYPES:
        return _gpu_gather(tensor)
    elif PartialState().distributed_type == DistributedType.MULTI_CPU:
        return _cpu_gather(tensor)
    else:
        return tensor


def _gpu_gather_object(object: Any):
    def _gpu_gather_object_one(object: Any):
        output_objects = [None for _ in range(PartialState().num_processes)]
        torch.distributed.all_gather_object(output_objects, object)
        return output_objects

    return recursively_apply(_gpu_gather_object_one, object)


_cpu_gather_object = _gpu_gather_object


def gather_object(object: Any):
    """
    Recursively gather object in a nested list/tuple/dictionary of objects from all devices.

    Args:
        object (nested list/tuple/dictionary of picklable object):
            The data to gather.

    Returns:
        The same data structure as `object` with all the objects sent to every device.
    """
    if PartialState().distributed_type == DistributedType.TPU:
        raise NotImplementedError("gather objects in TPU is not supported")
    elif PartialState().distributed_type in CUDA_DISTRIBUTED_TYPES:
        return _gpu_gather_object(object)
    elif PartialState().distributed_type == DistributedType.MULTI_CPU:
        return _cpu_gather_object(object)
    else:
        return object


def _gpu_broadcast(data, src=0):
    def _gpu_broadcast_one(tensor, src=0):
        torch.distributed.broadcast(tensor, src=src)
        return tensor

    return recursively_apply(_gpu_broadcast_one, data, error_on_other_type=True, src=src)


def _tpu_broadcast(tensor, src=0, name="broadcast tensor"):
    if isinstance(tensor, (list, tuple)):
        return honor_type(tensor, (_tpu_broadcast(t, name=f"{name}_{i}") for i, t in enumerate(tensor)))
    elif isinstance(tensor, Mapping):
        return type(tensor)({k: _tpu_broadcast(v, name=f"{name}_{k}") for k, v in tensor.items()})
    return xm.mesh_reduce(name, tensor, lambda x: x[src])


def broadcast(tensor, from_process: int = 0):
    """
    Recursively broadcast tensor in a nested list/tuple/dictionary of tensors to all devices.
    Args:
        tensor (nested list/tuple/dictionary of `torch.Tensor`):
            The data to gather.
        from_process (`int`, *optional*, defaults to 0):
            The process from which to send the data

    Returns:
        The same data structure as `tensor` with all tensors broadcasted to the proper device.
    """
    if PartialState().distributed_type == DistributedType.TPU:
        return _tpu_broadcast(tensor, src=from_process, name="accelerate.utils.broadcast")
    elif PartialState().distributed_type in CUDA_DISTRIBUTED_TYPES:
        return _gpu_broadcast(tensor, src=from_process)
    elif PartialState().distributed_type == DistributedType.MULTI_CPU:
        return _gpu_broadcast(tensor, src=from_process)
    else:
        return tensor


def broadcast_object_list(object_list, from_process: int = 0):
    """
    Broadcast a list of picklable objects form one process to the others.

    Args:
        object_list (list of picklable objects):
            The list of objects to broadcast. This list will be modified inplace.
        from_process (`int`, *optional*, defaults to 0):
            The process from which to send the data.

    Returns:
        The same list containing the objects from process 0.
    """
    if PartialState().distributed_type == DistributedType.TPU:
        for i, obj in enumerate(object_list):
            object_list[i] = xm.mesh_reduce("accelerate.utils.broadcast_object_list", obj, lambda x: x[f
    elif PartialState().distributed_type in CUDA_DISTRIBUTED_TYPES:
        torch.distributed.broadcast_object_list(object_list, src=from_process)
    elif PartialState().distributed_type == DistributedType.MULTI_CPU:
        torch.distributed.broadcast_object_list(object_list, src=from_process)
    return object_list


def slice_tensors(data, tensor_slice):
    """
    Recursively takes a slice in a nested list/tuple/dictionary of tensors.

    Args:
        data (nested list/tuple/dictionary of `torch.Tensor`):
            The data to slice.
        tensor_slice (`slice`):
            The slice to take.

    Returns:
        The same data structure as `data` with all the tensors slices.
    """

    def _slice_tensor(tensor, tensor_slice):
        return tensor[tensor_slice]

    return recursively_apply(_slice_tensor, data, tensor_slice)


def concatenate(data, dim=0):
    """
    Recursively concatenate the tensors in a nested list/tuple/dictionary of lists of tensors with the s

    Args:
        data (nested list/tuple/dictionary of lists of tensors `torch.Tensor`):
            The data to concatenate.
        dim (`int`, *optional*, defaults to 0):
            The dimension on which to concatenate.
    Returns:
        The same data structure as `data` with all the tensors concatenated.
    """
    if isinstance(data[0], (tuple, list)):
        return honor_type(data[0], (concatenate([d[i] for d in data], dim=dim) for i in range(len(data[0
    elif isinstance(data[0], Mapping):
        return type(data[0])({k: concatenate([d[k] for d in data], dim=dim) for k in data[0].keys()})
    elif not isinstance(data[0], torch.Tensor):
        raise TypeError(f"Can only concatenate tensors but got {type(data[0])}")
    return torch.cat(data, dim=dim)


def pad_across_processes(tensor, dim=0, pad_index=0, pad_first=False):
    """
    Recursively pad the tensors in a nested list/tuple/dictionary of tensors from all devices to the sam
    can safely be gathered.

    Args:
        tensor (nested list/tuple/dictionary of `torch.Tensor`):
            The data to gather.
        dim (`int`, *optional*, defaults to 0):
            The dimension on which to pad.
        pad_index (`int`, *optional*, defaults to 0):
            The value with which to pad.
        pad_first (`bool`, *optional*, defaults to `False`):
            Whether to pad at the beginning or the end.
    """

    def _pad_across_processes(tensor, dim=0, pad_index=0, pad_first=False):
        if dim >= len(tensor.shape):
            return tensor

        # Gather all sizes
        size = torch.tensor(tensor.shape, device=tensor.device)[None]
        sizes = gather(size).cpu()
        # Then pad to the maximum size
        max_size = max(s[dim] for s in sizes)
        if max_size == tensor.shape[dim]:
            return tensor

        old_size = tensor.shape
        new_size = list(old_size)
        new_size[dim] = max_size
        new_tensor = tensor.new_zeros(tuple(new_size)) + pad_index
        if pad_first:
            indices = tuple(
                slice(max_size - old_size[dim], max_size) if i == dim else slice(None) for i in range(le
            )
        else:
            indices = tuple(slice(0, old_size[dim]) if i == dim else slice(None) for i in range(len(new_
        new_tensor[indices] = tensor
        return new_tensor

    return recursively_apply(
        _pad_across_processes, tensor, error_on_other_type=True, dim=dim, pad_index=pad_index, pad_first
    )


def reduce(tensor, reduction="mean"):
    """
    Recursively reduce the tensors in a nested list/tuple/dictionary of lists of tensors across all proc
    mean of a given operation.

    Args:
        tensor (nested list/tuple/dictionary of `torch.Tensor`):
            The data to reduce.
        reduction (`str`, *optional*, defaults to `"mean"`):
            A reduction method. Can be of "mean", "sum", or "none"

    Returns:
        The same data structure as `data` with all the tensors reduced.
    """
    def _reduce_across_processes(tensor, reduction="mean"):
        state = PartialState()
        cloned_tensor = tensor.clone()
        if state.distributed_type == DistributedType.NO:
            if reduction == "sum":
                return cloned_tensor.sum()
            else:
                return cloned_tensor.mean()
        if state.distributed_type == DistributedType.TPU:
            xm.all_reduce("sum", cloned_tensor)
        elif state.distributed_type.value in CUDA_DISTRIBUTED_TYPES:
            torch.distributed.all_reduce(cloned_tensor, ReduceOp.SUM)
        elif state.distributed_type == DistributedType.MULTI_CPU:
            torch.distributed.all_reduce(cloned_tensor, ReduceOp.SUM)
        if reduction == "mean":
            cloned_tensor /= state.num_processes
        return cloned_tensor

    return recursively_apply(_reduce_across_processes, tensor, error_on_other_type=True, reduction=reduc


def convert_to_fp32(tensor):
    """
    Recursively converts the elements nested list/tuple/dictionary of tensors in FP16/BF16 precision to

    Args:
        tensor (nested list/tuple/dictionary of `torch.Tensor`):
            The data to convert from FP16/BF16 to FP32.

    Returns:
        The same data structure as `tensor` with all tensors that were in FP16/BF16 precision converted
    """

    def _convert_to_fp32(tensor):
        return tensor.float()

    def _is_fp16_bf16_tensor(tensor):
        return hasattr(tensor, "dtype") and (
            tensor.dtype == torch.float16 or (is_torch_version(">=", "1.10") and tensor.dtype == torch.b
        )

    return recursively_apply(_convert_to_fp32, tensor, test_type=_is_fp16_bf16_tensor)


class ConvertOutputsToFp32:
    """
    Decorator to apply to a function outputing tensors (like a model forward pass) that ensures the outp
    precision will be convert back to FP32.

    Args:
        model_forward (`Callable`):
            The function which outputs we want to treat.

    Returns:
        The same function as `model_forward` but with converted outputs.
    """

    def __init__(self, model_forward):
        self.model_forward = model_forward
        update_wrapper(self, model_forward)

    def __call__(self, *args, **kwargs):
        return convert_to_fp32(self.model_forward(*args, **kwargs))

    def __getstate__(self):
        raise pickle.PicklingError(
            "Cannot pickle a prepared model with automatic mixed precision, please unwrap the model with
        )


convert_outputs_to_fp32 = ConvertOutputsToFp32
def find_device(data):
    """
```

```
Finds the device on which a nested dict/list/tuple of tensors lies (assuming they are all on the sam

Args:
    (nested list/tuple/dictionary of `torch.Tensor`): The data we want to know the device of.
"""
if isinstance(data, Mapping):
    for obj in data.values():
        device = find_device(obj)
        if device is not None:
            return device
elif isinstance(data, (tuple, list)):
    for obj in data:
        device = find_device(obj)
        if device is not None:
            return device
elif isinstance(data, torch.Tensor):
    return data.device
```

accelerate-main/examples/by_feature/memory.py

```python
# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
import argparse
import os

# New Code #
import evaluate
import torch
from datasets import load_dataset
from torch.optim import AdamW
from torch.utils.data import DataLoader
from transformers import AutoModelForSequenceClassification, AutoTokenizer, get_linear_schedule_with_war

from accelerate import Accelerator, DistributedType
from accelerate.utils import find_executable_batch_size


########################################################################
# This is a fully working simple example to use Accelerate,
# specifically showcasing how to ensure out-of-memory errors never
# interrupt training, and builds off the `nlp_example.py` script.
#
# This example trains a Bert base model on GLUE MRPC
# in any of the following settings (with the same script):
#   - single CPU or single GPU
#   - multi GPUS (using PyTorch distributed mode)
#   - (multi) TPUs
#   - fp16 (mixed-precision) or fp32 (normal precision)
#
# New additions from the base script can be found quickly by
# looking for the # New Code # tags
#
# To run it in each of these various modes, follow the instructions
# in the readme for examples:
# https://github.com/huggingface/accelerate/tree/main/examples
#
########################################################################


MAX_GPU_BATCH_SIZE = 16
EVAL_BATCH_SIZE = 32


def get_dataloaders(accelerator: Accelerator, batch_size: int = 16):
    """
    Creates a set of `DataLoader`s for the `glue` dataset,
    using "bert-base-cased" as the tokenizer.

    Args:
        accelerator (`Accelerator`):
            An `Accelerator` object
        batch_size (`int`, *optional*):
            The batch size for the train and validation DataLoaders.
    """
    tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
    datasets = load_dataset("glue", "mrpc")
    def tokenize_function(examples):
        # max_length=None => use the model max length (it's actually the default)
        outputs = tokenizer(examples["sentence1"], examples["sentence2"], truncation=True, max_length=No
        return outputs

    # Apply the method we just defined to all the examples in all the splits of the dataset
    # starting with the main process first:
    with accelerator.main_process_first():
        tokenized_datasets = datasets.map(
            tokenize_function,
            batched=True,
            remove_columns=["idx", "sentence1", "sentence2"],
        )

    # We also rename the 'label' column to 'labels' which is the expected name for labels by the models
    # transformers library
    tokenized_datasets = tokenized_datasets.rename_column("label", "labels")

    def collate_fn(examples):
        # On TPU it's best to pad everything to the same length or training will be very slow.
        max_length = 128 if accelerator.distributed_type == DistributedType.TPU else None
        # When using mixed precision we want round multiples of 8/16
        if accelerator.mixed_precision == "fp8":
            pad_to_multiple_of = 16
        elif accelerator.mixed_precision != "no":
            pad_to_multiple_of = 8
        else:
            pad_to_multiple_of = None

        return tokenizer.pad(
            examples,
            padding="longest",
            max_length=max_length,
            pad_to_multiple_of=pad_to_multiple_of,
            return_tensors="pt",
        )

    # Instantiate dataloaders.
    train_dataloader = DataLoader(
        tokenized_datasets["train"], shuffle=True, collate_fn=collate_fn, batch_size=batch_size
    )
    eval_dataloader = DataLoader(
        tokenized_datasets["validation"], shuffle=False, collate_fn=collate_fn, batch_size=EVAL_BATCH_SI
    )

    return train_dataloader, eval_dataloader


# For testing only
if os.environ.get("TESTING_MOCKED_DATALOADERS", None) == "1":
    from accelerate.test_utils.training import mocked_dataloaders

    get_dataloaders = mocked_dataloaders  # noqa: F811


def training_function(config, args):
    # For testing only
    if os.environ.get("TESTING_MOCKED_DATALOADERS", None) == "1":
        config["num_epochs"] = 2
    # Initialize accelerator
    accelerator = Accelerator(cpu=args.cpu, mixed_precision=args.mixed_precision)
    # Sample hyper-parameters for learning rate, batch size, seed and a few other HPs
    lr = config["lr"]
    num_epochs = int(config["num_epochs"])
    seed = int(config["seed"])
    batch_size = int(config["batch_size"])

    metric = evaluate.load("glue", "mrpc")

    # New Code #
    # We now can define an inner training loop function. It should take a batch size as the only paramet
    # and build the dataloaders in there.
    # It also gets our decorator
    @find_executable_batch_size(starting_batch_size=batch_size)
    def inner_training_loop(batch_size):
        # And now just move everything below under this function
        # We need to bring in the Accelerator object from earlier
        nonlocal accelerator
        # And reset all of its attributes that could hold onto any memory:
        accelerator.free_memory()

        # Then we can declare the model, optimizer, and everything else:
        set_seed(seed)

        # Instantiate the model (we build the model here so that the seed also control new weights initi
        model = AutoModelForSequenceClassification.from_pretrained("bert-base-cased", return_dict=True)

        # We could avoid this line since the accelerator is set with `device_placement=True` (default va
        # Note that if you are placing tensors on devices manually, this line absolutely needs to be bef
        # creation otherwise training will not work on TPU (`accelerate` will kindly throw an error to m
        model = model.to(accelerator.device)

        # Instantiate optimizer
        optimizer = AdamW(params=model.parameters(), lr=lr)
        train_dataloader, eval_dataloader = get_dataloaders(accelerator, batch_size)

        # Instantiate scheduler
        lr_scheduler = get_linear_schedule_with_warmup(
            optimizer=optimizer,
            num_warmup_steps=100,
            num_training_steps=(len(train_dataloader) * num_epochs),
        )

        # Prepare everything
        # There is no specific order to remember, we just need to unpack the objects in the same order w
        # prepare method.
        model, optimizer, train_dataloader, eval_dataloader, lr_scheduler = accelerator.prepare(
            model, optimizer, train_dataloader, eval_dataloader, lr_scheduler
        )

        # Now we train the model
        for epoch in range(num_epochs):
            model.train()
            for step, batch in enumerate(train_dataloader):
                # We could avoid this line since we set the accelerator with `device_placement=True`.
                batch.to(accelerator.device)
                outputs = model(**batch)
                loss = outputs.loss
                accelerator.backward(loss)
                optimizer.step()
                lr_scheduler.step()
                optimizer.zero_grad()

            model.eval()
            for step, batch in enumerate(eval_dataloader):
                # We could avoid this line since we set the accelerator with `device_placement=True`.
                batch.to(accelerator.device)
                with torch.no_grad():
                    outputs = model(**batch)
                predictions = outputs.logits.argmax(dim=-1)
                predictions, references = accelerator.gather_for_metrics((predictions, batch["labels"]))
                metric.add_batch(
                    predictions=predictions,
                    references=references,
                )

            eval_metric = metric.compute()
            # Use accelerator.print to print only on the main process.
            accelerator.print(f"epoch {epoch}:", eval_metric)

    # New Code #
    # And call it at the end with no arguments
    # Note: You could also refactor this outside of your training loop function
    inner_training_loop()
def main():
    parser = argparse.ArgumentParser(description="Simple example of training script.")
    parser.add_argument(
        "--mixed_precision",
        type=str,
        default=None,
        choices=["no", "fp16", "bf16", "fp8"],
        help="Whether to use mixed precision. Choose"
        "between fp16 and bf16 (bfloat16). Bf16 requires PyTorch >= 1.10."
        "and an Nvidia Ampere GPU.",
    )
    parser.add_argument("--cpu", action="store_true", help="If passed, will train on the CPU.")
    args = parser.parse_args()
    config = {"lr": 2e-5, "num_epochs": 3, "seed": 42, "batch_size": 16}
    training_function(config, args)


if __name__ == "__main__":
    main()
```

accelerate-main/docs/source/package_reference/deepspeed.mdx

```
<!--Copyright 2021 The HuggingFace Team. All rights reserved.

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in comp
the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is dis
an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the Licen
specific language governing permissions and limitations under the License.
-->

# Utilities for DeepSpeed

[[autodoc]] utils.DeepSpeedPlugin

[[autodoc]] utils.DummyOptim

[[autodoc]] utils.DummyScheduler

[[autodoc]] utils.DeepSpeedEngineWrapper

[[autodoc]] utils.DeepSpeedOptimizerWrapper

[[autodoc]] utils.DeepSpeedSchedulerWrapper
```

accelerate-main/src/accelerate/utils/tqdm.py

```
# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

from .imports import is_tqdm_available


if is_tqdm_available():
    import tqdm.auto as _tqdm

from ..state import PartialState


def tqdm(main_process_only: bool = True, *args, **kwargs):
    """
    Wrapper around `tqdm.tqdm` that optionally displays only on the main process.

    Args:
        main_process_only (`bool`, *optional*):
            Whether to display the progress bar only on the main process
    """
    if not is_tqdm_available():
        raise ImportError("Accelerate's `tqdm` module requires `tqdm` to be installed. Please run `pip i
    disable = False
    if main_process_only:
        disable = PartialState().local_process_index == 0
    return _tqdm(*args, **kwargs, disable=disable)
```

accelerate-main/src/accelerate/utils/modeling.py

~~~python
# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import gc
import json
import logging
import os
import re
import shutil
import tempfile
from collections import defaultdict
from typing import Dict, List, Optional, Tuple, Union

import torch
import torch.nn as nn

from .imports import is_safetensors_available
from .offload import load_offloaded_weight, offload_weight, save_offload_index


if is_safetensors_available():
    from safetensors import safe_open
    from safetensors.torch import load_file as safe_load_file

WEIGHTS_INDEX_NAME = "pytorch_model.bin.index.json"


logger = logging.getLogger(__name__)


def convert_file_size_to_int(size: Union[int, str]):
    """
    Converts a size expressed as a string with digits an unit (like `"5MB"`) to an integer (in bytes).

    Args:
        size (`int` or `str`): The size to convert. Will be directly returned if an `int`.

    Example:

    ```py
    >>> convert_file_size_to_int("1MiB")
    1048576
    ~~~
    """
    if isinstance(size, int):
        return size
    if size.upper().endswith("GIB"):
        return int(size[:-3]) * (2**30)
    if size.upper().endswith("MIB"):
        return int(size[:-3]) * (2**20)
    if size.upper().endswith("KIB"):
        return int(size[:-3]) * (2**10)
    if size.upper().endswith("GB"):
        int_size = int(size[:-2]) * (10**9)
        return int_size // 8 if size.endswith("b") else int_size
    if size.upper().endswith("MB"):
        int_size = int(size[:-2]) * (10**6)
```
```

~~~
        return int_size // 8 if size.endswith("b") else int_size
    if size.upper().endswith("KB"):
        int_size = int(size[:-2]) * (10**3)
        return int_size // 8 if size.endswith("b") else int_size
    raise ValueError("`size` is not in a valid format. Use an integer followed by the unit, e.g., '5GB'.


def dtype_byte_size(dtype: torch.dtype):
    """
    Returns the size (in bytes) occupied by one parameter of type `dtype`.

    Example:

    ```py
    >>> dtype_byte_size(torch.float32)
    4
    ~~~
    """
    if dtype == torch.bool:
        return 1 / 8
    bit_search = re.search(r"[^\d](\d+)$", str(dtype))
    if bit_search is None:
        raise ValueError(f"`dtype` is not a valid dtype: {dtype}.")
    bit_size = int(bit_search.groups()[0])
    return bit_size // 8

def set_module_tensor_to_device(
    module: nn.Module,
    tensor_name: str,
    device: Union[int, str, torch.device],
    value: Optional[torch.Tensor] = None,
    dtype: Optional[Union[str, torch.dtype]] = None,
):
    """
    A helper function to set a given tensor (parameter of buffer) of a module on a specific device (note
    `param.to(device)` creates a new tensor not linked to the parameter, which is why we need this funct

    Args:
        module (`torch.nn.Module`):
            The module in which the tensor we want to move lives.
        param_name (`str`):
            The full name of the parameter/buffer.
        device (`int`, `str` or `torch.device`):
            The device on which to set the tensor.
        value (`torch.Tensor`, *optional*):
            The value of the tensor (useful when going from the meta device to any other device).
        dtype (`torch.dtype`, *optional*):
            If passed along the value of the parameter will be cast to this `dtype`. Otherwise, `value`
            the dtype of the existing parameter in the model.
    """
    # Recurse if needed
    if "." in tensor_name:
        splits = tensor_name.split(".")
        for split in splits[:-1]:
            new_module = getattr(module, split)
            if new_module is None:
                raise ValueError(f"{module} has no attribute {split}.")
            module = new_module
        tensor_name = splits[-1]

    if tensor_name not in module._parameters and tensor_name not in module._buffers:
        raise ValueError(f"{module} does not have a parameter or a buffer named {tensor_name}.")
    is_buffer = tensor_name in module._buffers
    old_value = getattr(module, tensor_name)

    if old_value.device == torch.device("meta") and device not in ["meta", torch.device("meta")] and val
        raise ValueError(f"{tensor_name} is on the meta device, we need a `value` to put in on {device}.

    if value is not None:
        if dtype is None:
            # For compatibility with PyTorch load_state_dict which converts state dict dtype to existing
```
```

~~~
            value = value.to(old_value.dtype)
        elif not str(value.dtype).startswith(("torch.uint", "torch.int", "torch.bool")):
            value = value.to(dtype)

    with torch.no_grad():
        if value is None:
            new_value = old_value.to(device)
        elif isinstance(value, torch.Tensor):
            new_value = value.to(device)
        else:
            new_value = torch.tensor(value, device=device)

        if is_buffer:
            module._buffers[tensor_name] = new_value
        elif value is not None or torch.device(device) != module._parameters[tensor_name].device:
            param_cls = type(module._parameters[tensor_name])
            kwargs = module._parameters[tensor_name].__dict__
            if param_cls.__name__ == "Int8Params":
                new_value = param_cls(new_value, requires_grad=old_value.requires_grad, **kwargs).to(dev
            else:
                new_value = param_cls(new_value, requires_grad=old_value.requires_grad).to(device)
            module._parameters[tensor_name] = new_value


def named_module_tensors(module: nn.Module, include_buffers: bool = True, recurse: bool = False):
    """
    A helper function that gathers all the tensors (parameters + buffers) of a given module. If `include
    it's the same as doing `module.named_parameters(recurse=recurse) + module.named_buffers(recurse=recu

    Args:
        module (`torch.nn.Module`):
            The module we want the tensors on.
        include_buffer (`bool`, *optional*, defaults to `True`):
            Whether or not to include the buffers in the result.
        recurse (`bool`, *optional`, defaults to `False`):
            Whether or not to go look in every submodule or just return the direct parameters and buffer
    """
    for named_parameter in module.named_parameters(recurse=recurse):
        yield named_parameter

    if include_buffers:
        for named_buffer in module.named_buffers(recurse=recurse):
            yield named_buffer


def find_tied_parameters(model: nn.Module, **kwargs):
    """
    Find the tied parameters in a given model.

    <Tip warning={true}>

    The signature accepts keyword arguments, but they are for the recursive part of this function and yo
    them.

    </Tip>

    Args:
        model (`torch.nn.Module`): The model to inspect.

    Returns:
        Dict[str, str]: A dictionary mapping tied parameter names to the name of the parameter they are

    Example:

    ```py
    >>> from collections import OrderedDict
    >>> import torch.nn as nn

    >>> model = nn.Sequential(OrderedDict([("linear1", nn.Linear(4, 4)), ("linear2", nn.Linear(4, 4))]))
    >>> model.linear2.weight = test_model.linear1.weight
    >>> find_tied_parameters(test_model)
    {'linear1.weight': 'linear2.weight'}
~~~

```
    ```
    """
    # Initialize result and named_parameters before recursing.
    named_parameters = kwargs.get("named_parameters", None)
    prefix = kwargs.get("prefix", "")
    result = kwargs.get("result", {})

    if named_parameters is None:
        named_parameters = {n: p for n, p in model.named_parameters()}
    else:
        # A tied parameter will not be in the full `named_parameters` seen above but will be in the `nam
        # of the submodule it belongs to. So while recursing we track the names that are not in the init
        # `named_parameters`.
        for name, parameter in model.named_parameters():
            full_name = name if prefix == "" else f"{prefix}.{name}"
            if full_name not in named_parameters:
                # When we find one, it has to be one of the existing parameters.
                for new_name, new_param in named_parameters.items():
                    if new_param is parameter:
                        result[new_name] = full_name

    # Once we have treated direct parameters, we move to the child modules.
    for name, child in model.named_children():
        child_name = name if prefix == "" else f"{prefix}.{name}"
        find_tied_parameters(child, named_parameters=named_parameters, prefix=child_name, result=result)

    return result

def retie_parameters(model, tied_params):
    """
    Reties tied parameters in a given model if the link was broken (for instance when adding hooks).

    Args:
        model (`torch.nn.Module`):
            The model in which to retie parameters.
        tied_params (`Dict[str, str]`):
            A mapping parameter name to tied parameter name as obtained by `find_tied_parameters`.
    """
    for param_name, tied_param_name in tied_params.items():
        param = model
        for split in param_name.split("."):
            param = getattr(param, split)
        tied_module = model
        for split in tied_param_name.split(".")[:-1]:
            tied_module = getattr(tied_module, split)
        setattr(tied_module, tied_param_name.split(".")[-1], param)

def _get_proper_dtype(dtype: Union[str, torch.device]) -> torch.dtype:
    """
    Just does torch.dtype(dtype) if necessary.
    """
    if isinstance(dtype, str):
        # We accept "torch.float16" or just "float16"
        dtype = dtype.replace("torch.", "")
        dtype = getattr(torch, dtype)
    return dtype

def compute_module_sizes(
    model: nn.Module,
    dtype: Optional[Union[str, torch.device]] = None,
    special_dtypes: Optional[Dict[str, Union[str, torch.device]]] = None,
):
    """
    Compute the size of each submodule of a given model.
    """
    if dtype is not None:
        dtype = _get_proper_dtype(dtype)
        dtype_size = dtype_byte_size(dtype)
    if special_dtypes is not None:
```
```

```
        special_dtypes = {key: _get_proper_dtype(dtyp) for key, dtyp in special_dtypes.items()}
        special_dtypes_size = {key: dtype_byte_size(dtyp) for key, dtyp in special_dtypes.items()}
    module_sizes = defaultdict(int)
    for name, tensor in named_module_tensors(model, recurse=True):
        if special_dtypes is not None and name in special_dtypes:
            size = tensor.numel() * special_dtypes_size[name]
        elif dtype is None:
            size = tensor.numel() * dtype_byte_size(tensor.dtype)
        else:
            size = tensor.numel() * min(dtype_size, dtype_byte_size(tensor.dtype))
        name_parts = name.split(".")
        for idx in range(len(name_parts) + 1):
            module_sizes[".".join(name_parts[:idx])] += size

    return module_sizes


def get_max_layer_size(
    modules: List[Tuple[str, torch.nn.Module]], module_sizes: Dict[str, int], no_split_module_classes: L
):
    """
    Utility function that will scan a list of named modules and return the maximum size used by one full
    definition of a layer being:
    - a module with no direct children (just parameters and buffers)
    - a module whose class name is in the list `no_split_module_classes`

    Args:
        modules (`List[Tuple[str, torch.nn.Module]]`):
            The list of named modules where we want to determine the maximum layer size.
        module_sizes (`Dict[str, int]`):
            A dictionary mapping each layer name to its size (as generated by `compute_module_sizes`).
        no_split_module_classes (`List[str]`):
            A list of class names for layers we don't want to be split.

    Returns:
        `Tuple[int, List[str]]`: The maximum size of a layer with the list of layer names realizing that
    """
    max_size = 0
    layer_names = []
    modules_to_treat = modules.copy()
    while len(modules_to_treat) > 0:
        module_name, module = modules_to_treat.pop(0)
        modules_children = list(module.named_children()) if isinstance(module, torch.nn.Module) else []
        if len(modules_children) == 0 or module.__class__.__name__ in no_split_module_classes:
            # No splitting this one so we compare to the max_size
            size = module_sizes[module_name]
            if size > max_size:
                max_size = size
                layer_names = [module_name]
            elif size == max_size:
                layer_names.append(module_name)
        else:
            modules_to_treat = [(f"{module_name}.{n}", v) for n, v in modules_children] + modules_to_tre
    return max_size, layer_names


def get_max_memory(max_memory: Optional[Dict[Union[int, str], Union[int, str]]] = None):
    """
    Get the maximum memory available if nothing is passed, converts string to int otherwise.
    """
    import psutil

    if max_memory is None:
        if not torch.cuda.is_available():
            max_memory = {}
        else:
            # Make sure CUDA is initialized on each GPU to have the right memory info.
            for i in range(torch.cuda.device_count()):
                _ = torch.tensor([0], device=i)
            max_memory = {i: torch.cuda.mem_get_info(i)[0] for i in range(torch.cuda.device_count())}
        max_memory["cpu"] = psutil.virtual_memory().available
        return max_memory
    for key in max_memory:
        if isinstance(max_memory[key], str):
            max_memory[key] = convert_file_size_to_int(max_memory[key])
    return max_memory


def clean_device_map(device_map: Dict[str, Union[int, str, torch.device]], module_name: str = ""):
    """
    Cleans a device_map by grouping all submodules that go on the same device together.
    """
    # Get the value of the current module and if there is only one split across several keys, regroup it
    prefix = "" if module_name == "" else f"{module_name}."
    values = [v for k, v in device_map.items() if k.startswith(prefix)]
    if len(set(values)) == 1 and len(values) > 1:
        for k in [k for k in device_map if k.startswith(prefix)]:
            del device_map[k]
        device_map[module_name] = values[0]

    # Recurse over the children
    children_modules = [k for k in device_map.keys() if k.startswith(module_name) and len(k) > len(modul
    idx = len(module_name.split(".")) + 1 if len(module_name) > 0 else 1
    children_modules = set(".".join(k.split(".")[:idx]) for k in children_modules)
    for child in children_modules:
        clean_device_map(device_map, module_name=child)

    return device_map


def load_offloaded_weights(model, index, offload_folder):
    """
    Loads the weights from the offload folder into the model.

    Args:
        model (`torch.nn.Module`):
            The model to load the weights into.
        index (`dict`):
            A dictionary containing the parameter name and its metadata for each parameter that was offl
            model.
        offload_folder (`str`):
            The folder where the offloaded weights are stored.
    """
    if index is None or len(index) == 0:
        # Nothing to do
        return

    for param_name, metadata in index.items():
        tensor_file = os.path.join(offload_folder, f"{param_name}.dat")
        weight = load_offloaded_weight(tensor_file, metadata)
        set_module_tensor_to_device(model, param_name, "cpu", value=weight)


def get_balanced_memory(
    model: nn.Module,
    max_memory: Optional[Dict[Union[int, str], Union[int, str]]] = None,
    no_split_module_classes: Optional[List[str]] = None,
    dtype: Optional[Union[str, torch.dtype]] = None,
    special_dtypes: Optional[Dict[str, Union[str, torch.device]]] = None,
    low_zero: bool = False,
):
    """
    Compute a `max_memory` dictionary for [`infer_auto_device_map`] that will balance the use of each av

    <Tip>

    All computation is done analyzing sizes and dtypes of the model parameters. As a result, the model c
    meta device (as it would if initialized within the `init_empty_weights` context manager).

    </Tip>

    Args:
        model (`torch.nn.Module`):
            The model to analyze.
        max_memory (`Dict`, *optional*):
            A dictionary device identifier to maximum memory. Will default to the maximum memory availab
        no_split_module_classes (`List[str]`, *optional*):
            A list of layer class names that should never be split across device (for instance any layer
            residual connection).
        dtype (`str` or `torch.dtype`, *optional*):
            If provided, the weights will be converted to that type when loaded.
        special_dtypes (`Dict[str, Union[str, torch.device]]`, *optional*):
            If provided, special dtypes to consider for some specific weights (will override dtype used
            all weights).
        low_zero (`bool`, *optional*):
            Minimizes the number of weights on GPU 0, which is convenient when it's used for other opera
            Transformers generate function).
    """
    # Get default / clean up max_memory
    max_memory = get_max_memory(max_memory)

    if not torch.cuda.is_available():
        return max_memory

    num_devices = len([d for d in max_memory if torch.device(d).type == "cuda" and max_memory[d] > 0])
    module_sizes = compute_module_sizes(model, dtype=dtype, special_dtypes=special_dtypes)
    per_gpu = module_sizes[""] // (num_devices - 1 if low_zero else num_devices)

    # We can't just set the memory to model_size // num_devices as it will end being too small: each GPU
    # slightly less layers and some layers will end up offload at the end. So this function computes a b
    # add which is the biggest of:
    # - the size of no split block (if applicable)
    # - the mean of the layer sizes
    if no_split_module_classes is None:
        no_split_module_classes = []
    elif not isinstance(no_split_module_classes, (list, tuple)):
        no_split_module_classes = [no_split_module_classes]

    # Identify the size of the no_split_block modules
    if len(no_split_module_classes) > 0:
        no_split_children = {}
        for name, size in module_sizes.items():
            if name == "":
                continue
            submodule = model
            for submodule_name in name.split("."):
                submodule = getattr(submodule, submodule_name)
            class_name = submodule.__class__.__name__
            if class_name in no_split_module_classes and class_name not in no_split_children:
                no_split_children[class_name] = size

            if set(no_split_children.keys()) == set(no_split_module_classes):
                break
        buffer = max(no_split_children.values()) if len(no_split_children) > 0 else 0
    else:
        buffer = 0

    # Compute mean of final modules. In the first dict of module sizes, leaves are the parameters
    leaves = [n for n in module_sizes if len([p for p in module_sizes if p.startswith(n) and len(p) > le
    module_sizes = {n: v for n, v in module_sizes.items() if n not in leaves}
    # Once removed, leaves are the final modules.
    leaves = [n for n in module_sizes if len([p for p in module_sizes if p.startswith(n) and len(p) > le
    mean_leaves = int(sum([module_sizes[n] for n in leaves]) / len(leaves))
    buffer = int(1.25 * max(buffer, mean_leaves))
    per_gpu += buffer

    max_memory = get_max_memory(max_memory)
    last_gpu = max(i for i in max_memory if isinstance(i, int) and max_memory[i] > 0)
    # The last device is left with max_memory just in case the buffer is not enough.
    for i in range(last_gpu):
        max_memory[i] = min(0 if low_zero and i == 0 else per_gpu, max_memory[i])

    if low_zero:
        min_zero = max(0, module_sizes[""] - sum([max_memory[i] for i in range(1, num_devices)]))
        max_memory[0] = min(min_zero, max_memory[0])
    return max_memory
```

```
def infer_auto_device_map(
    model: nn.Module,
    max_memory: Optional[Dict[Union[int, str], Union[int, str]]] = None,
    no_split_module_classes: Optional[List[str]] = None,
    dtype: Optional[Union[str, torch.dtype]] = None,
    special_dtypes: Optional[Dict[str, Union[str, torch.dtype]]] = None,
):
    """
    Compute a device map for a given model giving priority to GPUs, then offload on CPU and finally offl
    such that:
    - we don't exceed the memory available of any of the GPU.
    - if offload to the CPU is needed, there is always room left on GPU 0 to put back the layer offloade
      has the largest size.
    - if offload to the CPU is needed,we don't exceed the RAM available on the CPU.
    - if offload to the disk is needed, there is always room left on the CPU to put back the layer offlo
      that has the largest size.

    <Tip>

    All computation is done analyzing sizes and dtypes of the model parameters. As a result, the model c
    meta device (as it would if initialized within the `init_empty_weights` context manager).

    </Tip>

    Args:
        model (`torch.nn.Module`):
            The model to analyze.
        max_memory (`Dict`, *optional*):
            A dictionary device identifier to maximum memory. Will default to the maximum memory availab
        no_split_module_classes (`List[str]`, *optional*):
            A list of layer class names that should never be split across device (for instance any layer
            residual connection).
        dtype (`str` or `torch.dtype`, *optional*):
            If provided, the weights will be converted to that type when loaded.
        special_dtypes (`Dict[str, Union[str, torch.device]]`, *optional*):
            If provided, special dtypes to consider for some specific weights (will override dtype used
            all weights).
    """
    # Get default / clean up max_memory
    max_memory = get_max_memory(max_memory)
    if no_split_module_classes is None:
        no_split_module_classes = []
    elif not isinstance(no_split_module_classes, (list, tuple)):
        no_split_module_classes = [no_split_module_classes]

    devices = list(max_memory.keys())
    gpus = [device for device in devices if device != "cpu"]
    if "disk" not in devices:
        devices.append("disk")

    # Devices that need to keep space for a potential offloaded layer.
    main_devices = [gpus[0], "cpu"] if len(gpus) > 0 else ["cpu"]

    module_sizes = compute_module_sizes(model, dtype=dtype, special_dtypes=special_dtypes)
    tied_parameters = find_tied_parameters(model)

    device_map = {}
    current_device = 0
    current_memory_used = 0

    # Direct submodules and parameters
    modules_to_treat = (
        list(model.named_parameters(recurse=False))
        + list(model.named_children())
        + list(model.named_buffers(recurse=False))
    )
    # Initialize maximum largest layer, to know which space to keep in memory
    max_layer_size, max_layer_names = get_max_layer_size(modules_to_treat, module_sizes, no_split_module

    # Ready ? This is going to be a bit messy.
    while len(modules_to_treat) > 0:
        name, module = modules_to_treat.pop(0)
```

```
# Max size in the remaining layers may have changed since we took one, so we maybe update it.
max_layer_names = [n for n in max_layer_names if not n.startswith(name)]
if len(max_layer_names) == 0:
    max_layer_size, max_layer_names = get_max_layer_size(
        [(n, m) for n, m in modules_to_treat if isinstance(m, torch.nn.Module)],
        module_sizes,
        no_split_module_classes,
    )
# Assess size needed
module_size = module_sizes[name]
# We keep relevant tied parameters only: once of the tied parameters is inside the current modul
# is not.
tied_params = [v for k, v in tied_parameters.items() if name in k and name not in v]
# We ignore parameters that are tied when they're tied to > 1 one
tied_param = tied_params[0] if len(tied_params) == 1 else None

device = devices[current_device]
current_max_size = max_memory[device] if device != "disk" else None
# Reduce max size available by the largest layer.
if devices[current_device] in main_devices:
    current_max_size = current_max_size - max_layer_size
# Case 1 -> We're too big!
if current_max_size is not None and current_memory_used + module_size > current_max_size:
    # Split or not split?
    modules_children = list(module.named_children())
    if len(modules_children) == 0 or module.__class__.__name__ in no_split_module_classes:
        # -> no split, we go to the next device
        current_device += 1
        modules_to_treat = [(name, module)] + modules_to_treat
        current_memory_used = 0
    else:
        # -> split, we replace the module studied by its children + parameters
        modules_children = list(module.named_parameters(recurse=False)) + modules_children
        modules_to_treat = [(f"{name}.{n}", v) for n, v in modules_children] + modules_to_treat
        # Update the max layer size.
        max_layer_size, max_layer_names = get_max_layer_size(
            [(n, m) for n, m in modules_to_treat if isinstance(m, torch.nn.Module)],
            module_sizes,
            no_split_module_classes,
        )

# Case 2, it fits! We're not entirely out of the wood though, because we may have some tied para
elif tied_param is not None:
    # Determine the sized occupied by this module + the module containing the tied parameter
    tied_module_size = module_size
    tied_module_index = [i for i, (n, _) in enumerate(modules_to_treat) if n in tied_param][0]
    tied_module_name, tied_module = modules_to_treat[tied_module_index]
    tied_module_size += module_sizes[tied_module_name] - module_sizes[tied_param]
    if current_max_size is not None and current_memory_used + tied_module_size > current_max_siz
        # Split or not split?
        tied_module_children = list(tied_module.named_children())
        if len(tied_module_children) == 0 or tied_module.__class__.__name__ in no_split_module_c
            # If the tied module is not split, we go to the next device
            current_device += 1
            modules_to_treat = [(name, module)] + modules_to_treat
            current_memory_used = 0
        else:
            # Otherwise, we replace the tied module by its children.
            tied_module_children = list(tied_module.named_parameters(recurse=False)) + tied_modu
            tied_module_children = [(f"{tied_module_name}.{n}", v) for n, v in tied_module_child
            modules_to_treat = (
                [(name, module)]
                + modules_to_treat[:tied_module_index]
                + tied_module_children
                + modules_to_treat[tied_module_index + 1 :]
            )
            # Update the max layer size.
            max_layer_size, max_layer_names = get_max_layer_size(
                [(n, m) for n, m in modules_to_treat if isinstance(m, torch.nn.Module)],
                module_sizes,
                no_split_module_classes,
            )
```

```
                else:
                    # We really really fit!
                    current_memory_used += tied_module_size
                    device_map[name] = devices[current_device]
                    modules_to_treat.pop(tied_module_index)
                    device_map[tied_module_name] = devices[current_device]
            else:
                current_memory_used += module_size
                device_map[name] = devices[current_device]

        return clean_device_map(device_map)


    def check_device_map(model: nn.Module, device_map: Dict[str, Union[int, str, torch.device]]):
        """
        Checks a device map covers everything in a given model.

        Args:
            model (`torch.nn.Module`): The model to check the device map against.
            device_map (`Dict[str, Union[int, str, torch.device]]`): The device map to check.
        """
        all_model_tensors = [name for name, _ in model.state_dict().items()]
        for module_name in device_map.keys():
            all_model_tensors = [name for name in all_model_tensors if not name.startswith(module_name)]
        if len(all_model_tensors) > 0:
            non_covered_params = ", ".join(all_model_tensors)
            raise ValueError(
                f"The device_map provided does not give any device for the following parameters: {non_covere
            )


    def load_state_dict(checkpoint_file, device_map=None):
        """
        Load a checkpoint from a given file. If the checkpoint is in the safetensors format and a device map
        weights can be fast-loaded directly on the GPU.

        Args:
            checkpoint_file (`str`): The path to the checkpoint to load.
            device_map (`Dict[str, Union[int, str, torch.device]]`, *optional*):
                A map that specifies where each submodule should go. It doesn't need to be refined to each p
                name, once a given module name is inside, every submodule of it will be sent to the same dev
        """
        if checkpoint_file.endswith(".safetensors"):
            if not is_safetensors_available():
                raise ImportError(
                    f"To load {checkpoint_file}, the `safetensors` library is necessary `pip install safeten
                )
            with safe_open(checkpoint_file, framework="pt") as f:
                metadata = f.metadata()
                weight_names = f.keys()

            if metadata is None:
                logger.warn(
                    f"The safetensors archive passed at {checkpoint_file} does not contain metadata. "
                    "Make sure to save your model with the `save_pretrained` method. Defaulting to 'pt' meta
                )
                metadata = {"format": "pt"}

            if metadata.get("format") not in ["pt", "tf", "flax"]:
                raise OSError(
                    f"The safetensors archive passed at {checkpoint_file} does not contain the valid metadat
                    "you save your model with the `save_pretrained` method."
                )
            elif metadata["format"] != "pt":
                raise ValueError(f"The checkpoint passed was saved with {metadata['format']}, we need a the
            if device_map is None:
                return safe_load_file(checkpoint_file)
            else:
                devices = [device for device in device_map.values() if device not in ["disk"]]

                # if we only have one device we can load everything directly
                if len(devices) == 1:
```

```
                return safe_load_file(checkpoint_file, device=devices[0])

            # cpu device should always exist as fallback option
            if "cpu" not in devices:
                devices.append("cpu")

            # For each device, get the weights that go there
            device_weights = {device: [] for device in devices}
            for module_name, device in device_map.items():
                if device in devices:
                    device_weights[device].extend([k for k in weight_names if k.startswith(module_name)]

            # all weights that haven't defined a device should be loaded on CPU
            device_weights["cpu"].extend([k for k in weight_names if k not in sum(device_weights.values(
            tensors = {}
            for device in devices:
                with safe_open(checkpoint_file, framework="pt", device=device) as f:
                    for key in device_weights[device]:
                        tensors[key] = f.get_tensor(key)

            return tensors
    else:
        return torch.load(checkpoint_file)


def load_checkpoint_in_model(
    model: nn.Module,
    checkpoint: Union[str, os.PathLike],
    device_map: Optional[Dict[str, Union[int, str, torch.device]]] = None,
    offload_folder: Optional[Union[str, os.PathLike]] = None,
    dtype: Optional[Union[str, torch.dtype]] = None,
    offload_state_dict: bool = False,
    offload_buffers: bool = False,
):
    """
    Loads a (potentially sharded) checkpoint inside a model, potentially sending weights to a given devi
    loaded.

    <Tip warning={true}>

    Once loaded across devices, you still need to call [`dispatch_model`] on your model to make it able
    group the checkpoint loading and dispatch in one single call, use [`load_checkpoint_and_dispatch`].

    </Tip>

    Args:
        model (`torch.nn.Module`):
            The model in which we want to load a checkpoint.
        checkpoint (`str` or `os.PathLike`):
            The folder checkpoint to load. It can be:
            - a path to a file containing a whole model state dict
            - a path to a `.json` file containing the index to a sharded checkpoint
            - a path to a folder containing a unique `.index.json` file and the shards of a checkpoint.
        device_map (`Dict[str, Union[int, str, torch.device]]`, *optional*):
            A map that specifies where each submodule should go. It doesn't need to be refined to each p
            name, once a given module name is inside, every submodule of it will be sent to the same dev
        offload_folder (`str` or `os.PathLike`, *optional*):
            If the `device_map` contains any value `"disk"`, the folder where we will offload weights.
        dtype (`str` or `torch.dtype`, *optional*):
            If provided, the weights will be converted to that type when loaded.
        offload_state_dict (`bool`, *optional*, defaults to `False`):
            If `True`, will temporarily offload the CPU state dict on the hard drive to avoid getting ou
            the weight of the CPU state dict + the biggest shard does not fit.
        offload_buffers (`bool`, *optional*, defaults to `False`):
            Whether or not to include the buffers in the weights offloaded to disk.
    """
    if offload_folder is None and device_map is not None and "disk" in device_map.values():
        raise ValueError(
            "At least one of the model submodule will be offloaded to disk, please pass along an `offloa
        )
    elif offload_folder is not None and device_map is not None and "disk" in device_map.values():
        os.makedirs(offload_folder, exist_ok=True)
```

```
if isinstance(dtype, str):
    # We accept "torch.float16" or just "float16"
    dtype = dtype.replace("torch.", "")
    dtype = getattr(torch, dtype)

checkpoint_files = None
index_filename = None
if os.path.isfile(checkpoint):
    if str(checkpoint).endswith(".json"):
        index_filename = checkpoint
    else:
        checkpoint_files = [checkpoint]
elif os.path.isdir(checkpoint):
    potential_index = [f for f in os.listdir(checkpoint) if f.endswith(".index.json")]
    if len(potential_index) == 0:
        raise ValueError(f"{checkpoint} is not a folder containing a `.index.json` file.")
    elif len(potential_index) == 1:
        index_filename = os.path.join(checkpoint, potential_index[0])
    else:
        raise ValueError(f"{checkpoint} containing more than one `.index.json` file, delete the irre
else:
    raise ValueError(
        "`checkpoint` should be the path to a file containing a whole state dict, or the index of a
        f"checkpoint, or a folder containing a sharded checkpoint, but got {checkpoint}."
    )

if index_filename is not None:
    checkpoint_folder = os.path.split(index_filename)[0]
    with open(index_filename, "r") as f:
        index = json.loads(f.read())

    if "weight_map" in index:
        index = index["weight_map"]
    checkpoint_files = sorted(list(set(index.values())))
    checkpoint_files = [os.path.join(checkpoint_folder, f) for f in checkpoint_files]

# Logic for missing/unexepected keys goes here.

offload_index = {}
if offload_state_dict:
    state_dict_folder = tempfile.mkdtemp()
    state_dict_index = {}

buffer_names = [name for name, _ in model.named_buffers()]

for checkpoint_file in checkpoint_files:
    checkpoint = load_state_dict(checkpoint_file, device_map=device_map)
    if device_map is None:
        model.load_state_dict(checkpoint, strict=False)
    else:
        for param_name, param in checkpoint.items():
            module_name = param_name

            while len(module_name) > 0 and module_name not in device_map:
                module_name = ".".join(module_name.split(".")[:-1])
            if module_name == "" and "" not in device_map:
                # TODO: group all errors and raise at the end.
                raise ValueError(f"{param_name} doesn't have any device set.")
            param_device = device_map[module_name]

            if param_device == "disk":
                if offload_buffers or param_name not in buffer_names:
                    set_module_tensor_to_device(model, param_name, "meta")
                offload_weight(param, param_name, offload_folder, index=offload_index)
            elif param_device == "cpu" and offload_state_dict:
                set_module_tensor_to_device(model, param_name, "meta")
                offload_weight(param, param_name, state_dict_folder, index=state_dict_index)
            else:
                set_module_tensor_to_device(model, param_name, param_device, value=param, dtype=dtyp

    # Force Python to clean up.
    del checkpoint
    gc.collect()

save_offload_index(offload_index, offload_folder)

# Load back offloaded state dict on CPU
if offload_state_dict:
    load_offloaded_weights(model, state_dict_index, state_dict_folder)
    shutil.rmtree(state_dict_folder)
```

accelerate-main/docs/source/package_reference/megatron_lm.mdx

```
<!--Copyright 2021 The HuggingFace Team. All rights reserved.

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in comp
the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is dis
an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the Licen
specific language governing permissions and limitations under the License.
-->

# Utilities for Megatron-LM

[[autodoc]] utils.MegatronLMPlugin

[[autodoc]] utils.MegatronLMDummyScheduler

[[autodoc]] utils.MegatronLMDummyDataLoader

[[autodoc]] utils.AbstractTrainStep

[[autodoc]] utils.GPTTrainStep

[[autodoc]] utils.BertTrainStep

[[autodoc]] utils.T5TrainStep

[[autodoc]] utils.avg_losses_across_data_parallel_group
```

accelerate-main/src/accelerate/utils/transformer_engine.py

```python
# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import torch.nn as nn

from .imports import is_fp8_available


if is_fp8_available():
    import transformer_engine.pytorch as te


def convert_model(model, to_transformer_engine=True, _convert_linear=True, _convert_ln=True):
    """
    Recursively converts the linear and layernorm layers of a model to their `transformers_engine` count
    """
    if not is_fp8_available():
        raise ImportError("Using `convert_model` requires transformer_engine to be installed.")
    for name, module in model.named_children():
        if isinstance(module, nn.Linear) and to_transformer_engine and _convert_linear:
            # Return early if the linear layer weights are not multiples of 16
            if any(p % 16 != 0 for p in module.weight.shape):
                return
            has_bias = module.bias is not None
            te_module = te.Linear(module.in_features, module.out_features, bias=has_bias)
            te_module.weight.data = module.weight.data.clone()
            if has_bias:
                te_module.bias.data = module.bias.data.clone()

            setattr(model, name, te_module)
        elif isinstance(module, nn.LayerNorm) and to_transformer_engine and _convert_ln:
            te_module = te.LayerNorm(module.normalized_shape[0], eps=module.eps)
            te_module.weight.data = module.weight.data.clone()
            te_module.bias.data = module.bias.data.clone()

            setattr(model, name, te_module)
        elif isinstance(module, te.Linear) and not to_transformer_engine and _convert_linear:
            has_bias = module.bias is not None
            new_module = nn.Linear(module.in_features, module.out_features, bias=has_bias)
            new_module.weight.data = module.weight.data.clone()
            if has_bias:
                new_module.bias.data = module.bias.data.clone()

            setattr(model, name, new_module)
        elif isinstance(module, te.LayerNorm) and not to_transformer_engine and _convert_ln:
            new_module = nn.LayerNorm(module.normalized_shape[0], eps=module.eps)
            new_module.weight.data = module.weight.data.clone()
            new_module.bias.data = module.bias.data.clone()

            setattr(model, name, new_module)
        else:
            convert_model(
                module,
                to_transformer_engine=to_transformer_engine,
                _convert_linear=_convert_linear,
                _convert_ln=_convert_ln,
            )
```

```
def has_transformer_engine_layers(model):
    """
    Returns whether a given model has some `transformer_engine` layer or not.
    """
    for m in model.modules():
        if isinstance(m, (te.LayerNorm, te.Linear)):
            return True
    return False
```

accelerate-main/docs/source/basic_tutorials/launch.mdx

~~~
# Launching your ■ Accelerate scripts

In the previous tutorial, you were introduced to how to modify your current training script to use ■ Acc
The final version of that code is shown below:

```python
from accelerate import Accelerator

accelerator = Accelerator()

model, optimizer, training_dataloader, scheduler = accelerator.prepare(
    model, optimizer, training_dataloader, scheduler
)

for batch in training_dataloader:
    optimizer.zero_grad()
    inputs, targets = batch
    outputs = model(inputs)
    loss = loss_function(outputs, targets)
    accelerator.backward(loss)
    optimizer.step()
    scheduler.step()
~~~

But how do you run this code and have it utilize the special hardware available to it?

First you should rewrite the above code into a function, and make it callable as a script. For example:

```diff
  from accelerate import Accelerator

+ def main():
      accelerator = Accelerator()

      model, optimizer, training_dataloader, scheduler = accelerator.prepare(
          model, optimizer, training_dataloader, scheduler
      )

      for batch in training_dataloader:
          optimizer.zero_grad()
          inputs, targets = batch
          outputs = model(inputs)
          loss = loss_function(outputs, targets)
          accelerator.backward(loss)
          optimizer.step()
          scheduler.step()

+ if __name__ == "__main__":
+     main()
```

Next you need to launch it with `accelerate launch`.

<Tip warning={true}>

  It's recommended you run `accelerate config` before using `accelerate launch` to configure your enviro
```
```

  Otherwise ■ Accelerate will use very basic defaults depending on your system setup.

</Tip>

## Using accelerate launch

■ Accelerate has a special CLI command to help you launch your code in your system through `accelerate l
This command wraps around all of the different commands needed to launch your script on various platform

<Tip>

  If you are familiar with launching scripts in PyTorch yourself such as with `torchrun`, you can still

</Tip>

You can launch your script quickly by using:

```bash
accelerate launch {script_name.py} --arg1 --arg2 ...
```

Just put `accelerate launch` at the start of your command, and pass in additional arguments and paramete

Since this runs the various torch spawn methods, all of the expected environment variables can be modifi
For example, here is how to use `accelerate launch` with a single GPU:

```bash
CUDA_VISIBLE_DEVICES="0" accelerate launch {script_name.py} --arg1 --arg2 ...
```

You can also use `accelerate launch` without performing `accelerate config` first, but you may need to m
In this case, ■ Accelerate will make some hyperparameter decisions for you, e.g., if GPUs are available,
Here is how you would use all GPUs and train with mixed precision disabled:

```bash
accelerate launch --multi_gpu {script_name.py} {--arg1} {--arg2} ...
```

To get more specific you should pass in the needed parameters yourself. For instance, here is how you
would also launch that same script on two GPUs using mixed precision while avoiding all of the warnings:

```bash
accelerate launch --multi_gpu --mixed_precision=fp16 --num_processes=2 {script_name.py} {--arg1} {--arg2
```

For a complete list of parameters you can pass in, run:

```bash
accelerate launch -h
```

<Tip>

  Even if you are not using ■ Accelerate in your code, you can still use the launcher for starting your

</Tip>

For a visualization of this difference, that earlier `accelerate launch` on multi-gpu would look somethi

```bash
MIXED_PRECISION="fp16" torchrun --nproc_per_node=2 --num_machines=1 {script_name.py} {--arg1} {--arg2} .
```

## Why you should always use `accelerate config`

Why is it useful to the point you should **always** run `accelerate config`?

Remember that earlier call to `accelerate launch` as well as `torchrun`?
Post configuration, to run that script with the needed parts you just need to use `accelerate launch` ou

```bash
```

```
accelerate launch {script_name.py} {--arg1} {--arg2} ...
```

## Custom Configurations

As briefly mentioned earlier, `accelerate launch` should be mostly used through combining set configurat
made with the `accelerate config` command. These configs are saved to a `default_config.yaml` file in yo
This cache folder is located at (with decreasing order of priority):

- The content of your environment variable `HF_HOME` suffixed with `accelerate`.
- If it does not exist, the content of your environment variable `XDG_CACHE_HOME` suffixed with
  `huggingface/accelerate`.
- If this does not exist either, the folder `~/.cache/huggingface/accelerate`.

To have multiple configurations, the flag `--config_file` can be passed to the `accelerate launch` comma
with the location of the custom yaml.

An example yaml may look something like the following for two GPUs on a single machine using `fp16` for
```yaml
compute_environment: LOCAL_MACHINE
deepspeed_config: {}
distributed_type: MULTI_GPU
fsdp_config: {}
machine_rank: 0
main_process_ip: null
main_process_port: null
main_training_function: main
mixed_precision: fp16
num_machines: 1
num_processes: 2
use_cpu: false
```

Launching a script from the location of that custom yaml file looks like the following:
```bash
accelerate launch --config_file {path/to/config/my_config_file.yaml} {script_name.py} {--arg1} {--arg2}
```

accelerate-main/src/accelerate/utils/imports.py

```python
# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import importlib
import os
import sys
import warnings
from distutils.util import strtobool
from functools import lru_cache

import torch
from packaging.version import parse

from .environment import parse_flag_from_env
from .versions import compare_versions, is_torch_version


# The package importlib_metadata is in a different place, depending on the Python version.
if sys.version_info < (3, 8):
    import importlib_metadata
else:
    import importlib.metadata as importlib_metadata


try:
    import torch_xla.core.xla_model as xm  # noqa: F401

    _tpu_available = True
except ImportError:
    _tpu_available = False


# Cache this result has it's a C FFI call which can be pretty time-consuming
_torch_distributed_available = torch.distributed.is_available()


def is_torch_distributed_available() -> bool:
    return _torch_distributed_available


def is_ccl_available():
    return (
        importlib.util.find_spec("torch_ccl") is not None
        or importlib.util.find_spec("oneccl_bindings_for_pytorch") is not None
    )


def get_ccl_version():
    return importlib_metadata.version("oneccl_bind_pt")


def is_apex_available():
    return importlib.util.find_spec("apex") is not None


def is_fp8_available():
    return importlib.util.find_spec("transformer_engine") is not None
```

```
@lru_cache()
def is_tpu_available(check_device=True):
    "Checks if `torch_xla` is installed and potentially if a TPU is in the environment"
    if _tpu_available and check_device:
        try:
            # Will raise a RuntimeError if no XLA configuration is found
            _ = xm.xla_device()
            return True
        except RuntimeError:
            return False
    return _tpu_available


def is_deepspeed_available():
    package_exists = importlib.util.find_spec("deepspeed") is not None
    # Check we're not importing a "deepspeed" directory somewhere but the actual library by trying to gr
    # AND checking it has an author field in the metadata that is HuggingFace.
    if package_exists:
        try:
            _ = importlib_metadata.metadata("deepspeed")
            return True
        except importlib_metadata.PackageNotFoundError:
            return False


def is_bf16_available(ignore_tpu=False):
    "Checks if bf16 is supported, optionally ignoring the TPU"
    if is_tpu_available():
        return not ignore_tpu
    if is_torch_version(">=", "1.10"):
        if torch.cuda.is_available():
            return torch.cuda.is_bf16_supported()
        return True
    return False


def is_megatron_lm_available():
    if strtobool(os.environ.get("ACCELERATE_USE_MEGATRON_LM", "False")) == 1:
        package_exists = importlib.util.find_spec("megatron") is not None
        if package_exists:
            megatron_version = parse(importlib_metadata.version("megatron-lm"))
            return compare_versions(megatron_version, ">=", "2.2.0")
    return False


def is_safetensors_available():
    return importlib.util.find_spec("safetensors") is not None


def is_transformers_available():
    return importlib.util.find_spec("transformers") is not None


def is_datasets_available():
    return importlib.util.find_spec("datasets") is not None


def is_aim_available():
    return importlib.util.find_spec("aim") is not None


def is_tensorboard_available():
    return importlib.util.find_spec("tensorboard") is not None or importlib.util.find_spec("tensorboardX


def is_wandb_available():
    return importlib.util.find_spec("wandb") is not None


def is_comet_ml_available():
    return importlib.util.find_spec("comet_ml") is not None
def is_boto3_available():
    return importlib.util.find_spec("boto3") is not None


def is_rich_available():
    if importlib.util.find_spec("rich") is not None:
        if parse_flag_from_env("DISABLE_RICH"):
            warnings.warn(
                "The `DISABLE_RICH` flag is deprecated and will be removed in version 0.17.0 of ■ Accele
                FutureWarning,
            )
            return not parse_flag_from_env("DISABLE_RICH")
        return not parse_flag_from_env("ACCELERATE_DISABLE_RICH")
    return False


def is_sagemaker_available():
    return importlib.util.find_spec("sagemaker") is not None


def is_tqdm_available():
    return importlib.util.find_spec("tqdm") is not None


def is_mlflow_available():
    return importlib.util.find_spec("mlflow") is not None


def is_mps_available():
    return is_torch_version(">=", "1.12") and torch.backends.mps.is_available() and torch.backends.mps.i
```

accelerate-main/src/accelerate/utils/dataclasses.py

```
# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

"""
General namespace and dataclass related classes
"""

import argparse
import copy
import enum
import functools
import os
import typing
import warnings
from contextlib import contextmanager
from dataclasses import dataclass, field
from datetime import timedelta
from distutils.util import strtobool
from typing import Any, Callable, Dict, Iterable, List, Optional, Tuple

import torch

from .constants import FSDP_AUTO_WRAP_POLICY, FSDP_BACKWARD_PREFETCH, FSDP_STATE_DICT_TYPE, MODEL_NAME,
from .versions import is_torch_version


class KwargsHandler:
    """
    Internal mixin that implements a `to_kwargs()` method for a dataclass.
    """

    def to_dict(self):
        return copy.deepcopy(self.__dict__)

    def to_kwargs(self):
        """
        Returns a dictionary containing the attributes with values different from the default of this cl
        """
        default_dict = self.__class__().to_dict()
        this_dict = self.to_dict()
        return {k: v for k, v in this_dict.items() if default_dict[k] != v}


@dataclass
class DistributedDataParallelKwargs(KwargsHandler):
    """
    Use this object in your [`Accelerator`] to customize how your model is wrapped in a
    `torch.nn.parallel.DistributedDataParallel`. Please refer to the documentation of this
    [wrapper](https://pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html)
    information on each argument.

    <Tip warning={true}>

    `gradient_as_bucket_view` is only available in PyTorch 1.7.0 and later versions.

    `static_graph` is only available in PyTorch 1.11.0 and later versions.
    </Tip>
```

~~~
        Example:

        ```python
        from accelerate import Accelerator
        from accelerate.utils import DistributedDataParallelKwargs

        kwargs = DistributedDataParallelKwargs(find_unused_parameters=True)
        accelerator = Accelerator(kwargs_handlers=[kwargs])
        ~~~
        """

        dim: int = 0
        broadcast_buffers: bool = True
        bucket_cap_mb: int = 25
        find_unused_parameters: bool = False
        check_reduction: bool = False
        gradient_as_bucket_view: bool = False
        static_graph: bool = False

    @dataclass
    class GradScalerKwargs(KwargsHandler):
        """
        Use this object in your [`Accelerator`] to customize the behavior of mixed precision, specifically h
        `torch.cuda.amp.GradScaler` used is created. Please refer to the documentation of this
        [scaler](https://pytorch.org/docs/stable/amp.html?highlight=gradscaler) for more information on each

        <Tip warning={true}>

        `GradScaler` is only available in PyTorch 1.5.0 and later versions.

        </Tip>

        Example:

        ```python
        from accelerate import Accelerator
        from accelerate.utils import GradScalerKwargs

        kwargs = GradScalerKwargs(backoff_filter=0.25)
        accelerator = Accelerator(kwargs_handlers=[kwargs])
        ```
        """

        init_scale: float = 65536.0
        growth_factor: float = 2.0
        backoff_factor: float = 0.5
        growth_interval: int = 2000
        enabled: bool = True

    @dataclass
    class InitProcessGroupKwargs(KwargsHandler):
        """
        Use this object in your [`Accelerator`] to customize the initialization of the distributed processes
        to the documentation of this
        [method](https://pytorch.org/docs/stable/distributed.html#torch.distributed.init_process_group) for
        information on each argument.

        ```python
        from datetime import timedelta
        from accelerate import Accelerator
        from accelerate.utils import InitProcessGroupKwargs

        kwargs = InitProcessGroupKwargs(timeout=timedelta(seconds=800))
        accelerator = Accelerator(kwargs_handlers=[kwargs])
        ```
        """

        init_method: Optional[str] = None
        timeout: timedelta = timedelta(seconds=1800)
    @dataclass
```
```

~~~python
class FP8RecipeKwargs(KwargsHandler):
    """
    Use this object in your [`Accelerator`] to customize the initialization of the recipe for FP8 mixed
    training. Please refer to the documentation of this
    [class](https://docs.nvidia.com/deeplearning/transformer-engine/user-guide/api/common.html#transform
    for more information on each argument.

    ```python
    from accelerate import Accelerator
    from accelerate.utils import FP8RecipeKwargs

    kwargs = FP8RecipeKwargs(fp8_format="HYBRID")
    accelerator = Accelerator(mixed_precision="fp8", kwargs_handlers=[kwargs])
    ~~~
    """

    margin: int = 0
    interval: int = 1
    fp8_format: str = "E4M3"
    amax_history_len: int = 1
    amax_compute_algo: str = "most_recent"
    override_linear_precision: Tuple[bool, bool, bool] = (False, False, False)

    def __post_init__(self):
        self.fp8_format = self.fp8_format.upper()
        if self.fp8_format not in ["E4M3", "HYBRID"]:
            raise ValueError("`fp8_format` must be 'E4M3' or 'HYBRID'.")
        if self.amax_compute_algo not in ["max", "most_recent"]:
            raise ValueError("`amax_compute_algo` must be 'max' or 'most_recent'")

class DistributedType(str, enum.Enum):
    """
    Represents a type of distributed environment.

    Values:

        - **NO** -- Not a distributed environment, just a single process.
        - **MULTI_CPU** -- Distributed on multiple CPU nodes.
        - **MULTI_GPU** -- Distributed on multiple GPUs.
        - **DEEPSPEED** -- Using DeepSpeed.
        - **TPU** -- Distributed on TPUs.
    """

    # Subclassing str as well as Enum allows the `DistributedType` to be JSON-serializable out of the bo
    NO = "NO"
    MULTI_CPU = "MULTI_CPU"
    MULTI_GPU = "MULTI_GPU"
    DEEPSPEED = "DEEPSPEED"
    FSDP = "FSDP"
    TPU = "TPU"
    MPS = "MPS" # here for backward compatibility. Remove in v0.18.0
    MEGATRON_LM = "MEGATRON_LM"

class SageMakerDistributedType(str, enum.Enum):
    """
    Represents a type of distributed environment.

    Values:

        - **NO** -- Not a distributed environment, just a single process.
        - **DATA_PARALLEL** -- using sagemaker distributed data parallelism.
        - **MODEL_PARALLEL** -- using sagemaker distributed model parallelism.
    """

    # Subclassing str as well as Enum allows the `SageMakerDistributedType` to be JSON-serializable out
    NO = "NO"
    DATA_PARALLEL = "DATA_PARALLEL"
    MODEL_PARALLEL = "MODEL_PARALLEL"
class ComputeEnvironment(str, enum.Enum):
    """
```
```

```
    Represents a type of the compute environment.

    Values:

        - **LOCAL_MACHINE** -- private/custom cluster hardware.
        - **AMAZON_SAGEMAKER** -- Amazon SageMaker as compute environment.
    """

    # Subclassing str as well as Enum allows the `ComputeEnvironment` to be JSON-serializable out of the
    LOCAL_MACHINE = "LOCAL_MACHINE"
    AMAZON_SAGEMAKER = "AMAZON_SAGEMAKER"


class DynamoBackend(str, enum.Enum):
    """
    Represents a dynamo backend (see https://github.com/pytorch/torchdynamo).

    Values:

        - **NO** -- Do not use torch dynamo.
        - **EAGER** -- Uses PyTorch to run the extracted GraphModule. This is quite useful in debugging
          issues.
        - **AOT_EAGER** -- Uses AotAutograd with no compiler, i.e, just using PyTorch eager for the AotA
          extracted forward and backward graphs. This is useful for debugging, and unlikely to give spee
        - **INDUCTOR** -- Uses TorchInductor backend with AotAutograd and cudagraphs by leveraging codeg
          kernels. [Read
          more](https://dev-discuss.pytorch.org/t/torchinductor-a-pytorch-native-compiler-with-define-by
        - **NVFUSER** -- nvFuser with TorchScript. [Read
          more](https://dev-discuss.pytorch.org/t/tracing-with-primitives-update-1-nvfuser-and-its-primi
        - **AOT_NVFUSER** -- nvFuser with AotAutograd. [Read
          more](https://dev-discuss.pytorch.org/t/tracing-with-primitives-update-1-nvfuser-and-its-primi
        - **AOT_CUDAGRAPHS** -- cudagraphs with AotAutograd. [Read
          more](https://github.com/pytorch/torchdynamo/pull/757)
        - **OFI** -- Uses Torchscript optimize_for_inference. Inference only. [Read
          more](https://pytorch.org/docs/stable/generated/torch.jit.optimize_for_inference.html)
        - **FX2TRT** -- Uses Nvidia TensorRT for inference optimizations. Inference only. [Read
          more](https://github.com/pytorch/TensorRT/blob/master/docsrc/tutorials/getting_started_with_fx
        - **ONNXRT** -- Uses ONNXRT for inference on CPU/GPU. Inference only. [Read more](https://onnxru
        - **IPEX** -- Uses IPEX for inference on CPU. Inference only. [Read
          more](https://github.com/intel/intel-extension-for-pytorch).

    """

    # Subclassing str as well as Enum allows the `SageMakerDistributedType` to be JSON-serializable out
    NO = "NO"
    EAGER = "EAGER"
    AOT_EAGER = "AOT_EAGER"
    INDUCTOR = "INDUCTOR"
    NVFUSER = "NVFUSER"
    AOT_NVFUSER = "AOT_NVFUSER"
    AOT_CUDAGRAPHS = "AOT_CUDAGRAPHS"
    OFI = "OFI"
    FX2TRT = "FX2TRT"
    ONNXRT = "ONNXRT"
    IPEX = "IPEX"


class EnumWithContains(enum.EnumMeta):
    "A metaclass that adds the ability to check if `self` contains an item with the `in` operator"

    def __contains__(cls, item):
        try:
            cls(item)
        except ValueError:
            return False
        return True


class BaseEnum(enum.Enum, metaclass=EnumWithContains):
    "An enum class that can get the value of an item with `str(Enum.key)`"

    def __str__(self):
        return self.value

    @classmethod
    def list(cls):
        "Method to list all the possible items in `cls`"
        return list(map(str, cls))


class LoggerType(BaseEnum):
    """Represents a type of supported experiment tracker

    Values:

        - **ALL** -- all available trackers in the environment that are supported
        - **TENSORBOARD** -- TensorBoard as an experiment tracker
        - **WANDB** -- wandb as an experiment tracker
        - **COMETML** -- comet_ml as an experiment tracker
    """

    ALL = "all"
    AIM = "aim"
    TENSORBOARD = "tensorboard"
    WANDB = "wandb"
    COMETML = "comet_ml"
    MLFLOW = "mlflow"


class PrecisionType(BaseEnum):
    """Represents a type of precision used on floating point values

    Values:

        - **NO** -- using full precision (FP32)
        - **FP16** -- using half precision
        - **BF16** -- using brain floating point precision
    """

    NO = "no"
    FP8 = "fp8"
    FP16 = "fp16"
    BF16 = "bf16"


class RNGType(BaseEnum):
    TORCH = "torch"
    CUDA = "cuda"
    XLA = "xla"
    GENERATOR = "generator"


# data classes


@dataclass
class TensorInformation:
    shape: torch.Size
    dtype: torch.dtype


@dataclass
class ProjectConfiguration:
    """
    Configuration for the Accelerator object based on inner-project needs.
    """

    project_dir: str = field(default=None, metadata={"help": "A path to a directory for storing data."})
    logging_dir: str = field(
        default=None,
        metadata={
            "help": "A path to a directory for storing logs of locally-compatible loggers. If None, defa
        },
    )
    automatic_checkpoint_naming: bool = field(
        default=False,
        metadata={"help": "Whether saved states should be automatically iteratively named."},
    )

    total_limit: int = field(
        default=None,
        metadata={"help": "The maximum number of total saved states to keep."},
    )

    iteration: int = field(
        default=0,
        metadata={"help": "The current save iteration."},
    )

    def __post_init__(self):
        if self.logging_dir is None:
            self.logging_dir = self.project_dir


@dataclass
class GradientAccumulationPlugin(KwargsHandler):
    """
    A plugin to configure gradient accumulation behavior.
    """

    num_steps: int = field(default=None, metadata={"help": "The number of steps to accumulate gradients
    adjust_scheduler: bool = field(
        default=True,
        metadata={
            "help": "Whether to adjust the scheduler steps to account for the number of steps being accu
        },
    )


@dataclass
class TorchDynamoPlugin(KwargsHandler):
    """
    This plugin is used to compile a model with PyTorch 2.0
    """

    backend: DynamoBackend = field(
        default=None,
        metadata={"help": f"Possible options are {[b.value.lower() for b in DynamoBackend]}"},
    )
    mode: str = field(
        default=None, metadata={"help": "Possible options are 'default', 'reduce-overhead' or 'max-autot
    )
    fullgraph: bool = field(default=None, metadata={"help": "Whether it is ok to break model into severa
    dynamic: bool = field(default=None, metadata={"help": "Whether to use dynamic shape for tracing"})
    options: Any = field(default=None, metadata={"help": "A dictionary of options to pass to the backend
    disable: bool = field(default=False, metadata={"help": "Turn torch.compile() into a no-op for testin

    def __post_init__(self):
        prefix = "ACCELERATE_DYNAMO_"
        if self.backend is None:
            self.backend = os.environ.get(prefix + "BACKEND", "no")
        self.backend = DynamoBackend(self.backend.upper())
        if self.mode is None:
            self.mode = os.environ.get(prefix + "MODE", "default")
        if self.fullgraph is None:
            self.fullgraph = strtobool(os.environ.get(prefix + "USE_FULLGRAPH", "False")) == 1
        if self.dynamic is None:
            self.dynamic = strtobool(os.environ.get(prefix + "USE_DYNAMIC", "False")) == 1

    def to_dict(self):
        dynamo_config = copy.deepcopy(self.__dict__)
        dynamo_config["backend"] = dynamo_config["backend"].value.lower()
        return dynamo_config


@dataclass
```

```
class DeepSpeedPlugin:
    """
    This plugin is used to integrate DeepSpeed.
    """

    hf_ds_config: Any = field(
        default=None,
        metadata={
            "help": "path to DeepSpeed config file or dict or an object of class `accelerate.utils.deeps
        },
    )
    gradient_accumulation_steps: int = field(
        default=None, metadata={"help": "Number of steps to accumulate gradients before updating optimiz
    )
    gradient_clipping: float = field(default=None, metadata={"help": "Enable gradient clipping with valu
    zero_stage: int = field(
        default=None,
        metadata={"help": "Possible options are 0,1,2,3; Default will be taken from environment variable
    )
    is_train_batch_min: str = field(
        default=True,
        metadata={"help": "If both train & eval dataloaders are specified, this will decide the train_ba
    )
    offload_optimizer_device: bool = field(
        default=None,
        metadata={"help": "Possible options are none|cpu|nvme. Only applicable with ZeRO Stages 2 and 3.
    )
    offload_param_device: bool = field(
        default=None,
        metadata={"help": "Possible options are none|cpu|nvme. Only applicable with ZeRO Stage 3."},
    )
    zero3_init_flag: bool = field(
        default=None,
        metadata={
            "help": "Flag to indicate whether to enable `deepspeed.zero.Init` for constructing massive m
            "Only applicable with ZeRO Stage-3."
        },
    )
    zero3_save_16bit_model: bool = field(
        default=None,
        metadata={"help": "Flag to indicate whether to save 16-bit model. Only applicable with ZeRO Stag
    )

    def __post_init__(self):
        from .deepspeed import HfDeepSpeedConfig

        if self.gradient_accumulation_steps is None:
            self.gradient_accumulation_steps = int(os.environ.get("ACCELERATE_GRADIENT_ACCUMULATION_STEP

        if self.gradient_clipping is None:
            gradient_clipping = os.environ.get("ACCELERATE_GRADIENT_CLIPPING", "none")
            if gradient_clipping != "none":
                self.gradient_clipping = float(gradient_clipping)

        if self.zero_stage is None:
            self.zero_stage = int(os.environ.get("ACCELERATE_DEEPSPEED_ZERO_STAGE", 2))

        if self.offload_optimizer_device is None:
            self.offload_optimizer_device = os.environ.get("ACCELERATE_DEEPSPEED_OFFLOAD_OPTIMIZER_DEVIC

        if self.offload_param_device is None:
            self.offload_param_device = os.environ.get("ACCELERATE_DEEPSPEED_OFFLOAD_PARAM_DEVICE", "non

        if self.zero3_save_16bit_model is None:
            self.zero3_save_16bit_model = (
                os.environ.get("ACCELERATE_DEEPSPEED_ZERO3_SAVE_16BIT_MODEL", "false") == "true"
            )

        if self.hf_ds_config is None:
            self.hf_ds_config = os.environ.get("ACCELERATE_DEEPSPEED_CONFIG_FILE", "none")
        if (
            isinstance(self.hf_ds_config, dict)
            or (isinstance(self.hf_ds_config, str) and self.hf_ds_config != "none")
            or isinstance(self.hf_ds_config, HfDeepSpeedConfig)
        ):
            if not isinstance(self.hf_ds_config, HfDeepSpeedConfig):
                self.hf_ds_config = HfDeepSpeedConfig(self.hf_ds_config)
            if "gradient_accumulation_steps" not in self.hf_ds_config.config:
                self.hf_ds_config.config["gradient_accumulation_steps"] = 1
            if "zero_optimization" not in self.hf_ds_config.config:
                raise ValueError("Please specify the ZeRO optimization config in the DeepSpeed config.")

            self._deepspeed_config_checks()
            kwargs = {
                "gradient_accumulation_steps": self.gradient_accumulation_steps,
                "gradient_clipping": self.gradient_clipping if self.gradient_clipping else 1.0,
                "zero_optimization.stage": self.zero_stage,
                "zero_optimization.offload_optimizer.device": self.offload_optimizer_device,
                "zero_optimization.offload_param.device": self.offload_param_device,
                "zero_optimization.stage3_gather_16bit_weights_on_model_save": self.zero3_save_16bit_mod
            }
            for key in kwargs.keys():
                self.fill_match(key, **kwargs, must_match=False)
            self.hf_ds_config.set_stage_and_offload()
        else:
            config = {
                "train_batch_size": "auto",
                "train_micro_batch_size_per_gpu": "auto",
                "gradient_accumulation_steps": self.gradient_accumulation_steps,
                "zero_optimization": {
                    "stage": self.zero_stage,
                    "offload_optimizer": {
                        "device": self.offload_optimizer_device,
                    },
                    "offload_param": {
                        "device": self.offload_param_device,
                    },
                    "stage3_gather_16bit_weights_on_model_save": self.zero3_save_16bit_model,
                },
            }
            if self.gradient_clipping:
                config["gradient_clipping"] = self.gradient_clipping
            self.hf_ds_config = HfDeepSpeedConfig(config)

        self.deepspeed_config = self.hf_ds_config.config
        self.deepspeed_config["steps_per_print"] = float("inf")  # this will stop deepspeed from logging
        if self.zero3_init_flag is None:
            self.zero3_init_flag = (
                strtobool(os.environ.get("ACCELERATE_DEEPSPEED_ZERO3_INIT", str(self.hf_ds_config.is_zer
            )
        if self.zero3_init_flag and not self.hf_ds_config.is_zero3():
            warnings.warn("DeepSpeed Zero3 Init flag is only applicable for ZeRO Stage 3. Setting it to
            self.zero3_init_flag = False

    def fill_match(self, ds_key_long, mismatches=None, must_match=True, **kwargs):
        mismatches = [] if mismatches is None else mismatches
        config, ds_key = self.hf_ds_config.find_config_node(ds_key_long)
        if config is None:
            return

        if config.get(ds_key) == "auto":
            if ds_key_long in kwargs:
                config[ds_key] = kwargs[ds_key_long]
                return
            else:
                raise ValueError(
                    f"`{ds_key_long}` not found in kwargs. "
                    f"Please specify `{ds_key_long}` without `auto`(set to correct value) in the DeepSpe
                    "pass it in kwargs."
                )

        if not must_match:
            return
        ds_val = config.get(ds_key)
        if ds_val is not None and ds_key_long in kwargs:
            if ds_val != kwargs[ds_key_long]:
                mismatches.append(f"- ds {ds_key_long}={ds_val} vs arg {ds_key_long}={kwargs[ds_key_long

    def deepspeed_config_process(self, prefix="", mismatches=None, config=None, must_match=True, **kwarg
        """Process the DeepSpeed config with the values from the kwargs."""
        mismatches = [] if mismatches is None else mismatches
        if config is None:
            config = self.deepspeed_config
        for key, value in config.items():
            if isinstance(value, dict):
                self.deepspeed_config_process(
                    prefix=prefix + key + ".", mismatches=mismatches, config=value, must_match=must_matc
                )
            else:
                self.fill_match(prefix + key, mismatches, must_match=must_match, **kwargs)
        if len(mismatches) > 0 and prefix == "":
            mismatches_msg = "\n".join(mismatches)
            raise ValueError(
                "Please correct the following DeepSpeed config values that mismatch kwargs "
                f" values:\n{mismatches_msg}\nThe easiest method is to set these DeepSpeed config values
            )

    def set_mixed_precision(self, mixed_precision):
        ds_config = self.deepspeed_config
        kwargs = {
            "fp16.enabled": mixed_precision == "fp16",
            "bf16.enabled": mixed_precision == "bf16",
        }
        if mixed_precision == "fp16":
            if "fp16" not in ds_config:
                ds_config["fp16"] = {"enabled": True, "auto_cast": True}
        elif mixed_precision == "bf16":
            if "bf16" not in ds_config:
                ds_config["bf16"] = {"enabled": True}

        if mixed_precision != "no":
            diff_dtype = "bf16" if mixed_precision == "fp16" else "fp16"
            if str(ds_config.get(diff_dtype, {}).get("enabled", "False")).lower() == "true":
                raise ValueError(
                    f"`--mixed_precision` arg cannot be set to `{mixed_precision}` when `{diff_dtype}` i
                )
        for dtype in ["fp16", "bf16"]:
            if dtype not in ds_config:
                ds_config[dtype] = {"enabled": False}
        self.fill_match("fp16.enabled", must_match=False, **kwargs)
        self.fill_match("bf16.enabled", must_match=False, **kwargs)

    def set_deepspeed_weakref(self):
        from .imports import is_transformers_available

        if self.zero3_init_flag:
            if not is_transformers_available():
                raise Exception(
                    "When `zero3_init_flag` is set, it requires Transformers to be installed. "
                    "Please run `pip install transformers`."
                )
            ds_config = copy.deepcopy(self.deepspeed_config)
            if "gradient_accumulation_steps" not in ds_config or ds_config["gradient_accumulation_steps"
                ds_config["gradient_accumulation_steps"] = 1
            if (
                "train_micro_batch_size_per_gpu" not in ds_config
                or ds_config["train_micro_batch_size_per_gpu"] == "auto"
            ):
                ds_config["train_micro_batch_size_per_gpu"] = 1
            if ds_config["train_batch_size"] == "auto":
                del ds_config["train_batch_size"]

            from transformers.deepspeed import HfDeepSpeedConfig

            self.dschf = HfDeepSpeedConfig(ds_config)  # keep this object alive # noqa
    def is_zero3_init_enabled(self):
        return self.zero3_init_flag

    @contextmanager
    def zero3_init_context_manager(self, enable=False):
        old = self.zero3_init_flag
        if old == enable:
            yield
        else:
            self.zero3_init_flag = enable
            self.dschf = None
            self.set_deepspeed_weakref()
            yield
            self.zero3_init_flag = old
            self.dschf = None
            self.set_deepspeed_weakref()

    def _deepspeed_config_checks(self):
        env_variable_names_to_ignore = [
            "ACCELERATE_GRADIENT_ACCUMULATION_STEPS",
            "ACCELERATE_GRADIENT_CLIPPING",
            "ACCELERATE_DEEPSPEED_ZERO_STAGE",
            "ACCELERATE_DEEPSPEED_OFFLOAD_OPTIMIZER_DEVICE",
            "ACCELERATE_DEEPSPEED_OFFLOAD_PARAM_DEVICE",
            "ACCELERATE_DEEPSPEED_ZERO3_SAVE_16BIT_MODEL",
            "ACCELERATE_MIXED_PRECISION",
        ]
        env_variable_names_to_ignore = [
            name.replace("ACCELERATE_", "").replace("DEEPSPEED_", "").lower() for name in env_variable_n
        ]

        deepspeed_fields_from_accelerate_config = os.environ.get("ACCELERATE_CONFIG_DS_FIELDS", "").spli

        if any(name in env_variable_names_to_ignore for name in deepspeed_fields_from_accelerate_config)
            raise ValueError(
                f"When using `deepspeed_config_file`, the following accelerate config variables will be
                "Please specify them appropriately in the DeepSpeed config file.\n"
                "If you are using an accelerate config file, remove others config variables mentioned in
                "The easiest method is to create a new config following the questionnaire via `accelerat
                "It will only ask for the necessary config variables when using `deepspeed_config_file`.
            )


@dataclass
class FullyShardedDataParallelPlugin:
    """
    This plugin is used to enable fully sharded data parallelism.
    """

    sharding_strategy: "typing.Any" = field(
        default=None,
        metadata={
            "help": "FSDP Sharding Strategy of type `torch.distributed.fsdp.fully_sharded_data_parallel.
        },
    )
    backward_prefetch: "typing.Any" = field(
        default=None,
        metadata={
            "help": "FSDP Backward Prefetch of type `torch.distributed.fsdp.fully_sharded_data_parallel.
        },
    )
    mixed_precision_policy: "typing.Any" = field(
        default=None,
        metadata={
            "help": "A config to enable mixed precision training with FullyShardedDataParallel. "
            "The 3 flags that are set are `param_dtype`, `reduce_dtype`, `buffer_dtype`. "
            "Each flag expects `torch.dtype` as the value. "
            "It is of type `torch.distributed.fsdp.fully_sharded_data_parallel.MixedPrecision`."
        },
    )
    auto_wrap_policy: Optional[Callable] = field(
        default=None,
        metadata={"help": "A callable specifying a policy to recursively wrap layers with FSDP"},
    )
    cpu_offload: "typing.Any" = field(
        default=None,
        metadata={
            "help": "Decides Whether to offload parameters and gradients to CPU. "
            "It is of type `torch.distributed.fsdp.fully_sharded_data_parallel.CPUOffload`."
        },
    )
    ignored_modules: Optional[Iterable[torch.nn.Module]] = field(
        default=None,
        metadata={"help": "A list of modules to ignore for FSDP."},
    )

    state_dict_type: "typing.Any" = field(
        default=None,
        metadata={
            "help": "FSDP State Dict Type of type `torch.distributed.fsdp.fully_sharded_data_parallel.St
        },
    )

    state_dict_config: "typing.Any" = field(
        default=None,
        metadata={
            "help": "FSDP State Dict Config of type `torch.distributed.fsdp.fully_sharded_data_parallel.
        },
    )

    limit_all_gathers: bool = field(
        default=False,
        metadata={
            "help": "If False, then FSDP allows the CPU thread to schedule all-gathers "
            "without any extra synchronization. If True, then FSDP explicitly synchronizes the CPU threa
            "too many in-flight all-gathers. This bool only affects the sharded strategies that schedule
            "Enabling this can help lower the number of CUDA malloc retries."
        },
    )

    use_orig_params: bool = field(
        default=False,
        metadata={"help": "If True, enables parameter-efficient fine-tuning"},
    )

    def __post_init__(self):
        from torch.distributed.fsdp.fully_sharded_data_parallel import (
            BackwardPrefetch,
            CPUOffload,
            FullStateDictConfig,
            ShardingStrategy,
            StateDictType,
        )

        if self.sharding_strategy is None:
            self.sharding_strategy = ShardingStrategy(int(os.environ.get("FSDP_SHARDING_STRATEGY", 1)))

        if self.cpu_offload is None:
            if os.environ.get("FSDP_OFFLOAD_PARAMS", "false") == "true":
                self.cpu_offload = CPUOffload(offload_params=True)
            else:
                self.cpu_offload = CPUOffload(offload_params=False)

        if self.backward_prefetch is None:
            prefetch_policy = os.environ.get("FSDP_BACKWARD_PREFETCH", "NO_PREFETCH")
            if prefetch_policy != FSDP_BACKWARD_PREFETCH[-1]:
                self.backward_prefetch = BackwardPrefetch(FSDP_BACKWARD_PREFETCH.index(prefetch_policy)

        if self.state_dict_type is None:
            state_dict_type_policy = os.environ.get("FSDP_STATE_DICT_TYPE", "FULL_STATE_DICT")
            self.state_dict_type = StateDictType(FSDP_STATE_DICT_TYPE.index(state_dict_type_policy) + 1)

            if self.state_dict_type == StateDictType.FULL_STATE_DICT and self.state_dict_config is None:
                self.state_dict_config = FullStateDictConfig(offload_to_cpu=True, rank0_only=True)
    @staticmethod
```

```
def get_module_class_from_name(module, name):
    """
    Gets a class from a module by its name.

    Args:
        module (`torch.nn.Module`): The module to get the class from.
        name (`str`): The name of the class.
    """
    modules_children = list(module.children())
    if module.__class__.__name__ == name:
        return module.__class__
    elif len(modules_children) == 0:
        return
    else:
        for child_module in modules_children:
            module_class = FullyShardedDataParallelPlugin.get_module_class_from_name(child_module, n
            if module_class is not None:
                return module_class

def set_auto_wrap_policy(self, model):
    from torch.distributed.fsdp.wrap import size_based_auto_wrap_policy, transformer_auto_wrap_polic

    if self.auto_wrap_policy is None:
        auto_wrap_policy = os.environ.get("FSDP_AUTO_WRAP_POLICY", "NO_WRAP")
        if auto_wrap_policy == FSDP_AUTO_WRAP_POLICY[0]:
            transformer_cls_names_to_wrap = os.environ.get("FSDP_TRANSFORMER_CLS_TO_WRAP", "").split
            transformer_cls_to_wrap = set()
            for layer_class in transformer_cls_names_to_wrap:
                transformer_cls = FullyShardedDataParallelPlugin.get_module_class_from_name(model, l
                if transformer_cls is None:
                    raise Exception("Could not find the transformer layer class to wrap in the model
                else:
                    transformer_cls_to_wrap.add(transformer_cls)

            self.auto_wrap_policy = functools.partial(
                transformer_auto_wrap_policy,
                # Transformer layer class to wrap
                transformer_layer_cls=transformer_cls_to_wrap,
            )
        elif auto_wrap_policy == FSDP_AUTO_WRAP_POLICY[1]:
            min_num_params = int(os.environ.get("FSDP_MIN_NUM_PARAMS", 0))
            if min_num_params > 0:
                self.auto_wrap_policy = functools.partial(
                    size_based_auto_wrap_policy, min_num_params=min_num_params
                )

def set_mixed_precision(self, mixed_precision):
    if mixed_precision == "fp16":
        dtype = torch.float16
    elif mixed_precision == "bf16":
        dtype = torch.bfloat16
    else:
        raise ValueError(f"Unknown mixed precision value: {mixed_precision}")
    from torch.distributed.fsdp.fully_sharded_data_parallel import MixedPrecision

    if self.mixed_precision_policy is None:
        self.mixed_precision_policy = MixedPrecision(param_dtype=dtype, reduce_dtype=dtype, buffer_d

def save_model(self, accelerator, model, output_dir, model_index=0):
    from torch.distributed.fsdp.fully_sharded_data_parallel import FullyShardedDataParallel as FSDP
    from torch.distributed.fsdp.fully_sharded_data_parallel import StateDictType

    if is_torch_version("<=", "1.13.5"):
        with FSDP.state_dict_type(model, self.state_dict_type, self.state_dict_config):
            state_dict = model.state_dict()
    else:
        FSDP.set_state_dict_type(model, self.state_dict_type, self.state_dict_config)
        state_dict = model.state_dict()

    if self.state_dict_type == StateDictType.FULL_STATE_DICT:
        weights_name = f"{MODEL_NAME}.bin" if model_index == 0 else f"{MODEL_NAME}_{model_index}.bin
        output_model_file = os.path.join(output_dir, weights_name)
```

```
            if accelerator.process_index == 0:
                print(f"Saving model to {output_model_file}")
                torch.save(state_dict, output_model_file)
                print(f"Model saved to {output_model_file}")
        else:
            weights_name = (
                f"{MODEL_NAME}_rank{accelerator.process_index}.bin"
                if model_index == 0
                else f"{MODEL_NAME}_{model_index}_rank{accelerator.process_index}.bin"
            )
            output_model_file = os.path.join(output_dir, weights_name)
            print(f"Saving model to {output_model_file}")
            torch.save(state_dict, output_model_file)
            print(f"Model saved to {output_model_file}")

    def load_model(self, accelerator, model, input_dir, model_index=0):
        from torch.distributed.fsdp.fully_sharded_data_parallel import FullyShardedDataParallel as FSDP
        from torch.distributed.fsdp.fully_sharded_data_parallel import StateDictType

        accelerator.wait_for_everyone()

        if self.state_dict_type == StateDictType.FULL_STATE_DICT:
            weights_name = f"{MODEL_NAME}.bin" if model_index == 0 else f"{MODEL_NAME}_{model_index}.bin
            input_model_file = os.path.join(input_dir, weights_name)
            accelerator.print(f"Loading model from {input_model_file}")
            state_dict = torch.load(input_model_file)
            accelerator.print(f"Model loaded from {input_model_file}")
        else:
            weights_name = (
                f"{MODEL_NAME}_rank{accelerator.process_index}.bin"
                if model_index == 0
                else f"{MODEL_NAME}_{model_index}_rank{accelerator.process_index}.bin"
            )
            input_model_file = os.path.join(input_dir, weights_name)
            print(f"Loading model from {input_model_file}")
            state_dict = torch.load(input_model_file)
            print(f"Model loaded from {input_model_file}")

        if is_torch_version("<=", "1.13.5"):
            with FSDP.state_dict_type(model, self.state_dict_type, self.state_dict_config):
                model.load_state_dict(state_dict)
        else:
            FSDP.set_state_dict_type(model, self.state_dict_type, self.state_dict_config)
            model.load_state_dict(state_dict)

    def save_optimizer(self, accelerator, optimizer, model, output_dir, optimizer_index=0, optim_input=N
        from torch.distributed.fsdp.fully_sharded_data_parallel import FullyShardedDataParallel as FSDP

        optim_state = FSDP.full_optim_state_dict(model, optimizer, optim_input=optim_input)
        if accelerator.process_index == 0:
            optim_state_name = (
                f"{OPTIMIZER_NAME}.bin" if optimizer_index == 0 else f"{OPTIMIZER_NAME}_{optimizer_index
            )
            output_optimizer_file = os.path.join(output_dir, optim_state_name)
            print(f"Saving Optimizer state to {output_optimizer_file}")
            torch.save(optim_state, output_optimizer_file)
            print(f"Optimizer state saved in {output_optimizer_file}")

    def load_optimizer(self, accelerator, optimizer, model, input_dir, optimizer_index=0):
        from torch.distributed.fsdp.fully_sharded_data_parallel import FullyShardedDataParallel as FSDP

        accelerator.wait_for_everyone()
        full_osd = None
        if accelerator.process_index == 0:
            optimizer_name = (
                f"{OPTIMIZER_NAME}.bin" if optimizer_index == 0 else f"{OPTIMIZER_NAME}_{optimizer_index
            )
            input_optimizer_file = os.path.join(input_dir, optimizer_name)
            print(f"Loading Optimizer state from {input_optimizer_file}")
            full_osd = torch.load(input_optimizer_file)
            print(f"Optimizer state loaded from {input_optimizer_file}")
        # called from all ranks, though only rank0 has a valid param for full_osd
        sharded_osd = FSDP.scatter_full_optim_state_dict(full_osd, model)
        optimizer.load_state_dict(sharded_osd)


@dataclass
class MegatronLMPlugin:
    """
    Plugin for Megatron-LM to enable tensor, pipeline, sequence and data parallelism. Also to enable sel
    activation recomputation and optimized fused kernels.
    """

    tp_degree: int = field(default=None, metadata={"help": "tensor parallelism degree."})
    pp_degree: int = field(default=None, metadata={"help": "pipeline parallelism degree."})
    num_micro_batches: int = field(default=None, metadata={"help": "number of micro-batches."})
    gradient_clipping: float = field(
        default=None, metadata={"help": "gradient clipping value based on global L2 Norm (0 to disable)"
    )
    sequence_parallelism: bool = field(
        default=None,
        metadata={"help": "enable sequence parallelism"},
    )
    recompute_activation: bool = field(
        default=None,
        metadata={"help": "enable selective activation recomputation"},
    )
    use_distributed_optimizer: bool = field(
        default=None,
        metadata={"help": "enable distributed optimizer"},
    )
    pipeline_model_parallel_split_rank: int = field(
        default=None, metadata={"help": "Rank where encoder and decoder should be split."}
    )
    num_layers_per_virtual_pipeline_stage: int = field(
        default=None, metadata={"help": "Number of layers per virtual pipeline stage."}
    )
    is_train_batch_min: str = field(
        default=True,
        metadata={"help": "If both train & eval dataloaders are specified, this will decide the micro_ba
    )
    train_iters: int = field(
        default=None,
        metadata={
            "help": "Total number of iterations to train over all training runs. "
            "Note that either train-iters or train-samples should be provided when using `MegatronLMDumm
        },
    )
    train_samples: int = field(
        default=None,
        metadata={
            "help": "Total number of samples to train over all training runs. "
            "Note that either train-iters or train-samples should be provided when using `MegatronLMDumm
        },
    )
    weight_decay_incr_style: str = field(
        default="constant",
        metadata={"help": 'Weight decay increment function. choices=["constant", "linear", "cosine"]. '}
    )
    start_weight_decay: float = field(
        default=None,
        metadata={"help": "Initial weight decay coefficient for L2 regularization."},
    )
    end_weight_decay: float = field(
        default=None,
        metadata={"help": "End of run weight decay coefficient for L2 regularization."},
    )
    lr_decay_style: str = field(
        default="linear",
        metadata={"help": "Learning rate decay function. choices=['constant', 'linear', 'cosine']."},
    )
    lr_decay_iters: int = field(
        default=None,
        metadata={"help": "Number of iterations for learning rate decay. If None defaults to `train_iter
    )
    lr_decay_samples: int = field(
        default=None,
        metadata={"help": "Number of samples for learning rate decay. If None defaults to `train_samples
    )
    lr_warmup_iters: int = field(
        default=None,
        metadata={"help": "number of iterations to linearly warmup learning rate over."},
    )
    lr_warmup_samples: int = field(
        default=None,
        metadata={"help": "number of samples to linearly warmup learning rate over."},
    )
    lr_warmup_fraction: float = field(
        default=None,
        metadata={"help": "fraction of lr-warmup-(iters/samples) to linearly warmup learning rate over."
    )
    min_lr: float = field(
        default=0,
        metadata={"help": "Minumum value for learning rate. The scheduler clip values below this thresho
    )
    consumed_samples: List[int] = field(
        default=None,
        metadata={
            "help": "Number of samples consumed in the same order as the dataloaders to `accelerator.pre
        },
    )
    no_wd_decay_cond: Optional[Callable] = field(default=None, metadata={"help": "Condition to disable w
    scale_lr_cond: Optional[Callable] = field(default=None, metadata={"help": "Condition to scale learni
    lr_mult: float = field(default=1.0, metadata={"help": "Learning rate multiplier."})
    megatron_dataset_flag: bool = field(
        default=False,
        metadata={"help": "Whether the format of dataset follows Megatron-LM Indexed/Cached/MemoryMapped
    )
    seq_length: int = field(
        default=None,
        metadata={"help": "Maximum sequence length to process."},
    )
    encoder_seq_length: int = field(
        default=None,
        metadata={"help": "Maximum sequence length to process for the encoder."},
    )
    decoder_seq_length: int = field(
        default=None,
        metadata={"help": "Maximum sequence length to process for the decoder."},
    )
    tensorboard_dir: str = field(
        default=None,
        metadata={"help": "Path to save tensorboard logs."},
    )
    set_all_logging_options: bool = field(
        default=False,
        metadata={"help": "Whether to set all logging options."},
    )
    eval_iters: int = field(
        default=100, metadata={"help": "Number of iterations to run for evaluation validation/test for."
    )
    eval_interval: int = field(
        default=1000, metadata={"help": "Interval between running evaluation on validation set."}
    )
    return_logits: bool = field(
        default=False,
        metadata={"help": "Whether to return logits from the model."},
    )

    # custom train step args
    custom_train_step_class: Optional[Any] = field(
        default=None,
        metadata={"help": "Custom train step class."},
    )
    custom_train_step_kwargs: Optional[Dict[str, Any]] = field(
        default=None,
        metadata={"help": "Custom train step kwargs."},
    )

    # custom model args
    custom_model_provider_function: Optional[Callable] = field(
        default=None,
        metadata={"help": "Custom model provider function."},
    )
    custom_prepare_model_function: Optional[Callable] = field(
        default=None,
        metadata={"help": "Custom prepare model function."},
    )

    # remaining args such as enabling Alibi/ROPE positional embeddings,
    # wandb logging, Multi-Query Attention, etc.
    other_megatron_args: Optional[Dict[str, Any]] = field(
        default=None,
        metadata={"help": "Other Megatron-LM arguments. Please refer Megatron-LM"},
    )

    def __post_init__(self):
        prefix = "MEGATRON_LM_"
        if self.tp_degree is None:
            self.tp_degree = int(os.environ.get(prefix + "TP_DEGREE", 1))
        if self.pp_degree is None:
            self.pp_degree = int(os.environ.get(prefix + "PP_DEGREE", 1))
        if self.num_micro_batches is None:
            self.num_micro_batches = int(os.environ.get(prefix + "NUM_MICRO_BATCHES", 1))
        if self.gradient_clipping is None:
            self.gradient_clipping = float(os.environ.get(prefix + "GRADIENT_CLIPPING", 1.0))
        if self.recompute_activation is None:
            self.recompute_activation = strtobool(os.environ.get(prefix + "RECOMPUTE_ACTIVATION", "False
        if self.use_distributed_optimizer is None:
            self.use_distributed_optimizer = (
                strtobool(os.environ.get(prefix + "USE_DISTRIBUTED_OPTIMIZER", "False")) == 1
            )
        if self.sequence_parallelism is None:
            self.sequence_parallelism = strtobool(os.environ.get(prefix + "SEQUENCE_PARALLELISM", "False

        if self.pp_degree > 1 or self.use_distributed_optimizer:
            self.DDP_impl = "local"
        else:
            self.DDP_impl = "torch"

        if self.consumed_samples is not None:
            if len(self.consumed_samples) == 1:
                self.consumed_samples.extend([0, 0])
            elif len(self.consumed_samples) == 2:
                self.consumed_samples.append(0)

        self.megatron_lm_default_args = {
            "tensor_model_parallel_size": self.tp_degree,
            "pipeline_model_parallel_size": self.pp_degree,
            "pipeline_model_parallel_split_rank": self.pipeline_model_parallel_split_rank,
            "num_layers_per_virtual_pipeline_stage": self.num_layers_per_virtual_pipeline_stage,
            "DDP_impl": self.DDP_impl,
            "use_distributed_optimizer": self.use_distributed_optimizer,
            "sequence_parallel": self.sequence_parallelism,
            "clip_grad": self.gradient_clipping,
            "num_micro_batches": self.num_micro_batches,
            "consumed_samples": self.consumed_samples,
            "no_wd_decay_cond": self.no_wd_decay_cond,
            "scale_lr_cond": self.scale_lr_cond,
            "lr_mult": self.lr_mult,
            "megatron_dataset_flag": self.megatron_dataset_flag,
            "eval_iters": self.eval_iters,
            "eval_interval": self.eval_interval,
        }
        if self.recompute_activation:
            self.megatron_lm_default_args["recompute_granularity"] = "selective"
        if self.tensorboard_dir is not None:
            self.megatron_lm_default_args["tensorboard_dir"] = self.tensorboard_dir
            if self.set_all_logging_options:
                self.set_tensorboard_logging_options()
        if self.other_megatron_args is not None:
            self.megatron_lm_default_args.update(self.other_megatron_args)

    def set_network_size_args(self, model, batch_data=None):
        # Check if the model is either BERT, GPT or T5 else raise error
        # set 'num_layers', 'hidden_size', 'num_attention_heads', 'max_position_embeddings'
        if "megatron-bert" in model.config.model_type.lower():
            model_type_name = "bert"
            num_layers = model.config.num_hidden_layers
            hidden_size = model.config.hidden_size
            num_attention_heads = model.config.num_attention_heads
            max_position_embeddings = model.config.max_position_embeddings
            num_labels = model.config.num_labels
            orig_vocab_size = model.config.vocab_size
            if "maskedlm" in model.__class__.__name__.lower():
                pretraining_flag = True
            if self.seq_length is not None:
                if self.encoder_seq_length is not None:
                    warnings.warn("Both `seq_length` and `encoder_seq_length` are set. Using `encoder_se
                self.seq_length = self.encoder_seq_length
            elif self.encoder_seq_length is not None:
                self.seq_length = self.encoder_seq_length
            elif batch_data is not None:
                self.seq_length = batch_data["input_ids"].shape[1]
            else:
                self.seq_length = max_position_embeddings
            self.megatron_lm_default_args["seq_length"] = self.seq_length
        elif "gpt2" in model.config.model_type.lower():
            model_type_name = "gpt"
            num_layers = model.config.n_layer
            hidden_size = model.config.n_embd
            num_attention_heads = model.config.n_head
            max_position_embeddings = model.config.n_positions
            orig_vocab_size = model.config.vocab_size
            pretraining_flag = True
            if self.seq_length is not None:
                if self.decoder_seq_length is not None:
                    warnings.warn("Both `seq_length` and `decoder_seq_length` are set. Using `decoder_se
                self.seq_length = self.decoder_seq_length
            elif self.decoder_seq_length is not None:
                self.seq_length = self.decoder_seq_length
            elif batch_data is not None:
                self.seq_length = batch_data["input_ids"].shape[1]
            else:
                self.seq_length = max_position_embeddings
            self.megatron_lm_default_args["seq_length"] = self.seq_length
            self.megatron_lm_default_args["return_logits"] = self.return_logits
            self.megatron_lm_default_args["tokenizer_type"] = "GPT2BPETokenizer"
        elif "t5" in model.config.model_type.lower():
            model_type_name = "t5"
            num_layers = model.config.num_layers
            hidden_size = model.config.d_model
            num_attention_heads = model.config.num_heads
            max_position_embeddings = model.config.n_positions if hasattr(model.config, "n_positions") e
            orig_vocab_size = model.config.vocab_size
            pretraining_flag = True
            if self.encoder_seq_length is None:
                if batch_data is not None:
                    self.encoder_seq_length = batch_data["input_ids"].shape[1]
                else:
                    self.encoder_seq_length = max_position_embeddings
            if self.decoder_seq_length is None:
                if batch_data is not None:
                    self.decoder_seq_length = batch_data["labels"].shape[1]
                else:
                    self.decoder_seq_length = max_position_embeddings

            self.megatron_lm_default_args["encoder_seq_length"] = self.encoder_seq_length
            self.megatron_lm_default_args["decoder_seq_length"] = self.decoder_seq_length
        else:
            raise ValueError(
                "■ Accelerate Megatron-LM integration supports only BERT, GPT and T5 model. "
                "Please check the model you are using is one of those."
            )

        self.megatron_lm_default_args["model_type_name"] = model_type_name
        self.megatron_lm_default_args["num_layers"] = num_layers
        self.megatron_lm_default_args["hidden_size"] = hidden_size
        self.megatron_lm_default_args["num_attention_heads"] = num_attention_heads
        self.megatron_lm_default_args["max_position_embeddings"] = max_position_embeddings
        self.megatron_lm_default_args["pretraining_flag"] = pretraining_flag
        self.megatron_lm_default_args["orig_vocab_size"] = orig_vocab_size
        self.megatron_lm_default_args["model_return_dict"] = model.config.return_dict
        if model_type_name == "bert":
            self.megatron_lm_default_args["num_labels"] = num_labels

    def set_mixed_precision(self, mixed_precision):
        if mixed_precision == "fp16":
            self.megatron_lm_default_args["fp16"] = True
        elif mixed_precision == "bf16":
            self.megatron_lm_default_args["bf16"] = True
            self.DDP_impl = "local"
            self.megatron_lm_default_args["DDP_impl"] = self.DDP_impl

    def set_training_args(self, micro_batch_size, dp_degree):
        self.data_parallel_size = dp_degree
        self.micro_batch_size = micro_batch_size
        self.global_batch_size = dp_degree * micro_batch_size * self.num_micro_batches
        self.megatron_lm_default_args["data_parallel_size"] = self.data_parallel_size
        self.megatron_lm_default_args["micro_batch_size"] = self.micro_batch_size
        self.megatron_lm_default_args["global_batch_size"] = self.global_batch_size

    def set_optimizer_type(self, optimizer):
        optimizer_name = optimizer.__class__.__name__.lower()
        if "adam" in optimizer_name:
            self.megatron_lm_default_args["optimizer"] = "adam"
            self.megatron_lm_default_args["adam_beta1"] = optimizer.defaults["betas"][0]
            self.megatron_lm_default_args["adam_beta2"] = optimizer.defaults["betas"][1]
            self.megatron_lm_default_args["adam_eps"] = optimizer.defaults["eps"]
        elif "sgd" in optimizer_name:
            self.megatron_lm_default_args["optimizer"] = "sgd"
            self.megatron_lm_default_args["sgd_momentum"] = optimizer.defaults["momentum"]
        else:
            raise ValueError(f"Optimizer {optimizer_name} is not supported by Megatron-LM")

        self.megatron_lm_default_args["lr"] = optimizer.defaults["lr"]
        self.megatron_lm_default_args["weight_decay"] = optimizer.defaults["weight_decay"]

    def set_scheduler_args(self, scheduler):
        if self.train_iters is None:
            self.train_iters = scheduler.total_num_steps // self.megatron_lm_default_args["data_parallel
            if self.train_samples is not None:
                self.train_samples = None
                warnings.warn(
                    "Ignoring `train_samples` as `train_iters` based on scheduler is being used for trai
                )
        if self.lr_warmup_iters is None:
            self.lr_warmup_iters = scheduler.warmup_num_steps // self.megatron_lm_default_args["data_par
            if self.lr_warmup_samples is not None:
                warnings.warn(
                    "Ignoring `lr_warmup_samples` as `lr_warmup_iters` based on scheduler is being used
                )
            self.lr_warmup_samples = 0

        self.megatron_lm_default_args["train_iters"] = self.train_iters
        self.megatron_lm_default_args["lr_warmup_iters"] = self.lr_warmup_iters
        self.megatron_lm_default_args["train_samples"] = self.train_samples
        self.megatron_lm_default_args["lr_warmup_samples"] = self.lr_warmup_samples
        self.megatron_lm_default_args["lr_decay_iters"] = self.lr_decay_iters
        self.megatron_lm_default_args["lr_decay_samples"] = self.lr_decay_samples
        self.megatron_lm_default_args["lr_warmup_fraction"] = self.lr_warmup_fraction
        self.megatron_lm_default_args["lr_decay_style"] = self.lr_decay_style
        self.megatron_lm_default_args["weight_decay_incr_style"] = self.weight_decay_incr_style
        self.megatron_lm_default_args["start_weight_decay"] = self.start_weight_decay
        self.megatron_lm_default_args["end_weight_decay"] = self.end_weight_decay
        self.megatron_lm_default_args["min_lr"] = self.min_lr

    def set_tensorboard_logging_options(self):
        from megatron.arguments import _add_logging_args

        parser = argparse.ArgumentParser()
        parser = _add_logging_args(parser)
        logging_args = parser.parse_known_args()
        self.dataset_args = vars(logging_args[0])
        for key, value in self.dataset_args.items():
            if key.startswith("log_"):
                self.megatron_lm_default_args[key] = True
            elif key.startswith("no_log_"):
                self.megatron_lm_default_args[key.replace("no_", "")] = True
```

accelerate-main/src/accelerate/utils/random.py

```
# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import random
from typing import List, Optional, Union

import numpy as np
import torch

from ..state import AcceleratorState
from .constants import CUDA_DISTRIBUTED_TYPES
from .dataclasses import DistributedType, RNGType
from .imports import is_tpu_available


if is_tpu_available(check_device=False):
    import torch_xla.core.xla_model as xm


def set_seed(seed: int, device_specific: bool = False):
    """
    Helper function for reproducible behavior to set the seed in `random`, `numpy`, `torch`.

    Args:
        seed (`int`):
            The seed to set.
        device_specific (`bool`, *optional*, defaults to `False`):
            Whether to differ the seed on each device slightly with `self.process_index`.
    """
    if device_specific:
        seed += AcceleratorState().process_index
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    # ^^ safe to call this function even if cuda is not available
    if is_tpu_available():
        xm.set_rng_state(seed)


def synchronize_rng_state(rng_type: Optional[RNGType] = None, generator: Optional[torch.Generator] = Non
    # Get the proper rng state
    if rng_type == RNGType.TORCH:
        rng_state = torch.get_rng_state()
    elif rng_type == RNGType.CUDA:
        rng_state = torch.cuda.get_rng_state()
    elif rng_type == RNGType.XLA:
        assert is_tpu_available(), "Can't synchronize XLA seeds on an environment without TPUs."
        rng_state = torch.tensor(xm.get_rng_state())
    elif rng_type == RNGType.GENERATOR:
        assert generator is not None, "Need a generator to synchronize its seed."
        rng_state = generator.get_state()

    # Broadcast the rng state from device 0 to other devices
    state = AcceleratorState()
    if state.distributed_type == DistributedType.TPU:
        rng_state = xm.mesh_reduce("random_seed", rng_state, lambda x: x[0])
    elif state.distributed_type in CUDA_DISTRIBUTED_TYPES:
        rng_state = rng_state.to(state.device)
        torch.distributed.broadcast(rng_state, 0)
        rng_state = rng_state.cpu()
    elif state.distributed_type == DistributedType.MULTI_CPU:
        torch.distributed.broadcast(rng_state, 0)

    # Set the broadcast rng state
    if rng_type == RNGType.TORCH:
        torch.set_rng_state(rng_state)
    elif rng_type == RNGType.CUDA:
        torch.cuda.set_rng_state(rng_state)
    elif rng_type == RNGType.XLA:
        xm.set_rng_state(rng_state.item())
    elif rng_type == RNGType.GENERATOR:
        generator.set_state(rng_state)


def synchronize_rng_states(rng_types: List[Union[str, RNGType]], generator: Optional[torch.Generator] =
    for rng_type in rng_types:
        synchronize_rng_state(RNGType(rng_type), generator=generator)
```

accelerate-main/src/accelerate/utils/versions.py

```
# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import sys
from typing import Union

from packaging.version import Version, parse

from .constants import STR_OPERATION_TO_FUNC


if sys.version_info < (3, 8):
    import importlib_metadata
else:
    import importlib.metadata as importlib_metadata

torch_version = parse(importlib_metadata.version("torch"))


def compare_versions(library_or_version: Union[str, Version], operation: str, requirement_version: str):
    """
    Compares a library version to some requirement using a given operation.

    Args:
        library_or_version (`str` or `packaging.version.Version`):
            A library name or a version to check.
        operation (`str`):
            A string representation of an operator, such as `">"` or `"<="`.
        requirement_version (`str`):
            The version to compare the library version against
    """
    if operation not in STR_OPERATION_TO_FUNC.keys():
        raise ValueError(f"`operation` must be one of {list(STR_OPERATION_TO_FUNC.keys())}, received {op
    operation = STR_OPERATION_TO_FUNC[operation]
    if isinstance(library_or_version, str):
        library_or_version = parse(importlib_metadata.version(library_or_version))
    return operation(library_or_version, parse(requirement_version))


def is_torch_version(operation: str, version: str):
    """
    Compares the current PyTorch version to a given reference with an operation.

    Args:
        operation (`str`):
            A string representation of an operator, such as `">"` or `"<="`
        version (`str`):
            A string version of PyTorch
    """
    return compare_versions(torch_version, operation, version)
```

accelerate-main/src/accelerate/utils/constants.py

```
# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import operator as op


SCALER_NAME = "scaler.pt"
MODEL_NAME = "pytorch_model"
RNG_STATE_NAME = "random_states"
OPTIMIZER_NAME = "optimizer"
SCHEDULER_NAME = "scheduler"
SAGEMAKER_PYTORCH_VERSION = "1.10.2"
SAGEMAKER_PYTHON_VERSION = "py38"
SAGEMAKER_TRANSFORMERS_VERSION = "4.17.0"
SAGEMAKER_PARALLEL_EC2_INSTANCES = ["ml.p3.16xlarge", "ml.p3dn.24xlarge", "ml.p4dn.24xlarge"]
FSDP_SHARDING_STRATEGY = ["FULL_SHARD", "SHARD_GRAD_OP", "NO_SHARD"]
FSDP_AUTO_WRAP_POLICY = ["TRANSFORMER_BASED_WRAP", "SIZE_BASED_WRAP", "NO_WRAP"]
FSDP_BACKWARD_PREFETCH = ["BACKWARD_PRE", "BACKWARD_POST", "NO_PREFETCH"]
FSDP_STATE_DICT_TYPE = ["FULL_STATE_DICT", "LOCAL_STATE_DICT", "SHARDED_STATE_DICT"]
DEEPSPEED_MULTINODE_LAUNCHERS = ["pdsh", "standard", "openmpi", "mvapich"]
TORCH_DYNAMO_MODES = ["default", "reduce-overhead", "max-autotune"]

STR_OPERATION_TO_FUNC = {">": op.gt, ">=": op.ge, "==": op.eq, "!=": op.ne, "<=": op.le, "<": op.lt}

# These are the args for `torch.distributed.launch` for pytorch < 1.9
TORCH_LAUNCH_PARAMS = [
    "nnodes",
    "nproc_per_node",
    "rdzv_backend",
    "rdzv_endpoint",
    "rdzv_id",
    "rdzv_conf",
    "standalone",
    "max_restarts",
    "monitor_interval",
    "start_method",
    "role",
    "module",
    "m",
    "no_python",
    "run_path",
    "log_dir",
    "r",
    "redirects",
    "t",
    "tee",
    "node_rank",
    "master_addr",
    "master_port",
]

CUDA_DISTRIBUTED_TYPES = ["DEEPSPEED", "MULTI_GPU", "FSDP", "MEGATRON_LM"]
```

accelerate-main/src/accelerate/commands/config/__init__.py

```python
#!/usr/bin/env python

# Copyright 2021 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import argparse

from .config import config_command_parser
from .config_args import default_config_file, load_config_from_file  # noqa: F401
from .default import default_command_parser
from .update import update_command_parser


def get_config_parser(subparsers=None):
    parent_parser = argparse.ArgumentParser(add_help=False, allow_abbrev=False)
    # The main config parser
    config_parser = config_command_parser(subparsers)
    # The subparser to add commands to
    subcommands = config_parser.add_subparsers(title="subcommands", dest="subcommand")

    # Then add other parsers with the parent parser
    default_command_parser(subcommands, parents=[parent_parser])
    update_command_parser(subcommands, parents=[parent_parser])

    return config_parser


def main():
    config_parser = get_config_parser()
    args = config_parser.parse_args()

    if not hasattr(args, "func"):
        config_parser.print_help()
        exit(1)

    # Run
    args.func(args)


if __name__ == "__main__":
    main()
```

accelerate-main/src/accelerate/commands/tpu.py

```
#!/usr/bin/env python

# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import argparse
import os
import subprocess

from packaging.version import Version, parse

from accelerate.commands.config.config_args import default_config_file, load_config_from_file


_description = "Run commands across TPU VMs for initial setup before running `accelerate launch`."


def tpu_command_parser(subparsers=None):
    if subparsers is not None:
        parser = subparsers.add_parser("tpu-config", description=_description)
    else:
        parser = argparse.ArgumentParser("Accelerate tpu-config command", description=_description)
    # Core arguments
    config_args = parser.add_argument_group(
        "Config Arguments", "Arguments that can be configured through `accelerate config`."
    )
    config_args.add_argument(
        "--config_file",
        type=str,
        default=None,
        help="Path to the config file to use for accelerate.",
    )
    config_args.add_argument(
        "--tpu_name",
        default=None,
        help="The name of the TPU to use. If not specified, will use the TPU specified in the config fil
    )
    config_args.add_argument(
        "--tpu_zone",
        default=None,
        help="The zone of the TPU to use. If not specified, will use the zone specified in the config fi
    )
    pod_args = parser.add_argument_group("TPU Arguments", "Arguments for options ran inside the TPU.")
    pod_args.add_argument(
        "--use_alpha",
        action="store_true",
        help="Whether to use `gcloud alpha` when running the TPU training script instead of `gcloud`.",
    )
    pod_args.add_argument(
        "--command_file",
        default=None,
        help="The path to the file containing the commands to run on the pod on startup.",
    )
    pod_args.add_argument(
        "--command",
        action="append",
        nargs="+",
        help="A command to run on the pod. Can be passed multiple times.",
    )
    pod_args.add_argument(
        "--install_accelerate",
        action="store_true",
        help="Whether to install accelerate on the pod. Defaults to False.",
    )
    pod_args.add_argument(
        "--accelerate_version",
        default="latest",
        help="The version of accelerate to install on the pod. If not specified, will use the latest pyp
    )
    pod_args.add_argument(
        "--debug", action="store_true", help="If set, will print the command that would be run instead o
    )

    if subparsers is not None:
        parser.set_defaults(func=tpu_command_launcher)
    return parser


def tpu_command_launcher(args):
    defaults = None

    # Get the default from the config file if it exists.
    if args.config_file is not None or os.path.isfile(default_config_file):
        defaults = load_config_from_file(args.config_file)
        if not args.command_file and defaults.command_file is not None and not args.command:
            args.command_file = defaults.command_file
        if not args.command and defaults.commands is not None:
            args.command = defaults.commands
        if not args.tpu_name:
            args.tpu_name = defaults.tpu_name
        if not args.tpu_zone:
            args.tpu_zone = defaults.tpu_zone
    if args.accelerate_version == "dev":
        args.accelerate_version = "git+https://github.com/huggingface/accelerate.git"
    elif args.accelerate_version == "latest":
        args.accelerate_version = "accelerate -U"
    elif isinstance(parse(args.accelerate_version), Version):
        args.accelerate_version = f"accelerate=={args.accelerate_version}"

    if not args.command_file and not args.command:
        raise ValueError("You must specify either a command file or a command to run on the pod.")

    if args.command_file:
        with open(args.command_file, "r") as f:
            args.command = [f.read().splitlines()]

    # To turn list of lists into list of strings
    if isinstance(args.command[0], list):
        args.command = [line for cmd in args.command for line in cmd]
    # Default to the shared folder and install accelerate
    new_cmd = ["cd /usr/share"]
    if args.install_accelerate:
        new_cmd += [f"pip install {args.accelerate_version}"]
    new_cmd += args.command
    args.command = "; ".join(new_cmd)

    # Then send it to gcloud
    # Eventually try to use google-api-core to do this instead of subprocess
    cmd = ["gcloud"]
    if args.use_alpha:
        cmd += ["alpha"]
    cmd += [
        "compute",
        "tpus",
        "tpu-vm",
        "ssh",
        args.tpu_name,
        "--zone",
        args.tpu_zone,
        "--command",
        args.command,
        "--worker",
        "all",
    ]
    if args.debug:
        print(f"Running {' '.join(cmd)}")
        return
    subprocess.run(cmd)
    print("Successfully setup pod.")


def main():
    parser = tpu_command_parser()
    args = parser.parse_args()

    tpu_command_launcher(args)
```

accelerate-main/src/accelerate/commands/test.py

```
#!/usr/bin/env python

# Copyright 2021 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import argparse
import os

from accelerate.test_utils import execute_subprocess_async


def test_command_parser(subparsers=None):
    if subparsers is not None:
        parser = subparsers.add_parser("test")
    else:
        parser = argparse.ArgumentParser("Accelerate test command")

    parser.add_argument(
        "--config_file",
        default=None,
        help=(
            "The path to use to store the config file. Will default to a file named default_config.yaml
            "location, which is the content of the environment `HF_HOME` suffixed with 'accelerate', or
            "such an environment variable, your cache directory ('~/.cache' or the content of `XDG_CACHE
            "with 'huggingface'."
        ),
    )

    if subparsers is not None:
        parser.set_defaults(func=test_command)
    return parser


def test_command(args):
    script_name = os.path.sep.join(__file__.split(os.path.sep)[:-2] + ["test_utils", "scripts", "test_sc

    if args.config_file is None:
        test_args = script_name
    else:
        test_args = f"--config_file={args.config_file} {script_name}"

    cmd = ["accelerate-launch"] + test_args.split()
    result = execute_subprocess_async(cmd, env=os.environ.copy())
    if result.returncode == 0:
        print("Test is a success! You are ready for your distributed training!")


def main():
    parser = test_command_parser()
    args = parser.parse_args()
    test_command(args)


if __name__ == "__main__":
    main()
```

accelerate-main/src/accelerate/commands/accelerate_cli.py

```
#!/usr/bin/env python

# Copyright 2021 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

from argparse import ArgumentParser

from accelerate.commands.config import get_config_parser
from accelerate.commands.env import env_command_parser
from accelerate.commands.launch import launch_command_parser
from accelerate.commands.test import test_command_parser
from accelerate.commands.tpu import tpu_command_parser


def main():
    parser = ArgumentParser("Accelerate CLI tool", usage="accelerate <command> [<args>]", allow_abbrev=F
    subparsers = parser.add_subparsers(help="accelerate command helpers")

    # Register commands
    get_config_parser(subparsers=subparsers)
    env_command_parser(subparsers=subparsers)
    launch_command_parser(subparsers=subparsers)
    tpu_command_parser(subparsers=subparsers)
    test_command_parser(subparsers=subparsers)

    # Let's go
    args = parser.parse_args()

    if not hasattr(args, "func"):
        parser.print_help()
        exit(1)

    # Run
    args.func(args)


if __name__ == "__main__":
    main()
```

accelerate-main/docs/source/basic_tutorials/launch.mdx

# Launching your ■ Accelerate scripts

In the previous tutorial, you were introduced to how to modify your current training script to use ■ Acc
The final version of that code is shown below:

```python
from accelerate import Accelerator

accelerator = Accelerator()

model, optimizer, training_dataloader, scheduler = accelerator.prepare(
    model, optimizer, training_dataloader, scheduler
)

for batch in training_dataloader:
    optimizer.zero_grad()
    inputs, targets = batch
    outputs = model(inputs)
    loss = loss_function(outputs, targets)
    accelerator.backward(loss)
    optimizer.step()
    scheduler.step()
```

But how do you run this code and have it utilize the special hardware available to it?

First you should rewrite the above code into a function, and make it callable as a script. For example:

```diff
  from accelerate import Accelerator

+ def main():
      accelerator = Accelerator()

      model, optimizer, training_dataloader, scheduler = accelerator.prepare(
          model, optimizer, training_dataloader, scheduler
      )

      for batch in training_dataloader:
          optimizer.zero_grad()
          inputs, targets = batch
          outputs = model(inputs)
          loss = loss_function(outputs, targets)
          accelerator.backward(loss)
          optimizer.step()
          scheduler.step()

+ if __name__ == "__main__":
+     main()
```

Next you need to launch it with `accelerate launch`.

<Tip warning={true}>

  It's recommended you run `accelerate config` before using `accelerate launch` to configure your enviro

  Otherwise ■ Accelerate will use very basic defaults depending on your system setup.

</Tip>

## Using accelerate launch

■ Accelerate has a special CLI command to help you launch your code in your system through `accelerate l
This command wraps around all of the different commands needed to launch your script on various platforn

<Tip>

  If you are familiar with launching scripts in PyTorch yourself such as with `torchrun`, you can still

</Tip>

You can launch your script quickly by using:

```bash
accelerate launch {script_name.py} --arg1 --arg2 ...
```

Just put `accelerate launch` at the start of your command, and pass in additional arguments and paramete

Since this runs the various torch spawn methods, all of the expected environment variables can be modifi
For example, here is how to use `accelerate launch` with a single GPU:

```bash
CUDA_VISIBLE_DEVICES="0" accelerate launch {script_name.py} --arg1 --arg2 ...
```

You can also use `accelerate launch` without performing `accelerate config` first, but you may need to m
In this case, ■ Accelerate will make some hyperparameter decisions for you, e.g., if GPUs are available,
Here is how you would use all GPUs and train with mixed precision disabled:

```bash
accelerate launch --multi_gpu {script_name.py} {--arg1} {--arg2} ...
```

To get more specific you should pass in the needed parameters yourself. For instance, here is how you
would also launch that same script on two GPUs using mixed precision while avoiding all of the warnings:

```bash
accelerate launch --multi_gpu --mixed_precision=fp16 --num_processes=2 {script_name.py} {--arg1} {--arg2
```

For a complete list of parameters you can pass in, run:

```bash
accelerate launch -h
```

<Tip>

  Even if you are not using ■ Accelerate in your code, you can still use the launcher for starting your

</Tip>

For a visualization of this difference, that earlier `accelerate launch` on multi-gpu would look somethi

```bash
MIXED_PRECISION="fp16" torchrun --nproc_per_node=2 --num_machines=1 {script_name.py} {--arg1} {--arg2} .
```

## Why you should always use `accelerate config`

Why is it useful to the point you should **always** run `accelerate config`?

Remember that earlier call to `accelerate launch` as well as `torchrun`?
Post configuration, to run that script with the needed parts you just need to use `accelerate launch` ou

```bash
```

```
accelerate launch {script_name.py} {--arg1} {--arg2} ...
```

## Custom Configurations

As briefly mentioned earlier, `accelerate launch` should be mostly used through combining set configurat
made with the `accelerate config` command. These configs are saved to a `default_config.yaml` file in yo
This cache folder is located at (with decreasing order of priority):

- The content of your environment variable `HF_HOME` suffixed with `accelerate`.
- If it does not exist, the content of your environment variable `XDG_CACHE_HOME` suffixed with
  `huggingface/accelerate`.
- If this does not exist either, the folder `~/.cache/huggingface/accelerate`.

To have multiple configurations, the flag `--config_file` can be passed to the `accelerate launch` comma
with the location of the custom yaml.

An example yaml may look something like the following for two GPUs on a single machine using `fp16` for
```yaml
compute_environment: LOCAL_MACHINE
deepspeed_config: {}
distributed_type: MULTI_GPU
fsdp_config: {}
machine_rank: 0
main_process_ip: null
main_process_port: null
main_training_function: main
mixed_precision: fp16
num_machines: 1
num_processes: 2
use_cpu: false
```

Launching a script from the location of that custom yaml file looks like the following:
```bash
accelerate launch --config_file {path/to/config/my_config_file.yaml} {script_name.py} {--arg1} {--arg2}
```

accelerate-main/src/accelerate/commands/env.py

```
#!/usr/bin/env python

# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import argparse
import os
import platform

import numpy as np
import torch

from accelerate import __version__ as version
from accelerate.commands.config import default_config_file, load_config_from_file


def env_command_parser(subparsers=None):
    if subparsers is not None:
        parser = subparsers.add_parser("env")
    else:
        parser = argparse.ArgumentParser("Accelerate env command")

    parser.add_argument(
        "--config_file", default=None, help="The config file to use for the default values in the launch
    )

    if subparsers is not None:
        parser.set_defaults(func=env_command)
    return parser


def env_command(args):
    pt_version = torch.__version__
    pt_cuda_available = torch.cuda.is_available()

    accelerate_config = "Not found"
    # Get the default from the config file.
    if args.config_file is not None or os.path.isfile(default_config_file):
        accelerate_config = load_config_from_file(args.config_file).to_dict()

    info = {
        "`Accelerate` version": version,
        "Platform": platform.platform(),
        "Python version": platform.python_version(),
        "Numpy version": np.__version__,
        "PyTorch version (GPU?)": f"{pt_version} ({pt_cuda_available})",
    }

    print("\nCopy-and-paste the text below in your GitHub issue\n")
    print("\n".join([f"- {prop}: {val}" for prop, val in info.items()]))

    print("- `Accelerate` default config:" if args.config_file is None else "- `Accelerate` config passe
    accelerate_config_str = (
        "\n".join([f"\t- {prop}: {val}" for prop, val in accelerate_config.items()])
        if isinstance(accelerate_config, dict)
        else f"\t{accelerate_config}"
    )
    print(accelerate_config_str)

    info["`Accelerate` configs"] = accelerate_config

    return info


def main() -> int:
    parser = env_command_parser()
    args = parser.parse_args()
    env_command(args)
    return 0


if __name__ == "__main__":
    raise SystemExit(main())
```

accelerate-main/src/accelerate/commands/menu/keymap.py

```python
# Copyright 2022 The HuggingFace Team and Brian Chao. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

"""
Utilities relating to parsing raw characters from the keyboard, based on https://github.com/bchao1/bulle
"""


import os
import string
import sys


ARROW_KEY_FLAG = 1 << 8

KEYMAP = {
    "tab": ord("\t"),
    "newline": ord("\r"),
    "esc": 27,
    "up": 65 + ARROW_KEY_FLAG,
    "down": 66 + ARROW_KEY_FLAG,
    "right": 67 + ARROW_KEY_FLAG,
    "left": 68 + ARROW_KEY_FLAG,
    "mod_int": 91,
    "undefined": sys.maxsize,
    "interrupt": 3,
    "insert": 50,
    "delete": 51,
    "pg_up": 53,
    "pg_down": 54,
}

KEYMAP["arrow_begin"] = KEYMAP["up"]
KEYMAP["arrow_end"] = KEYMAP["left"]

if sys.platform == "win32":
    WIN_CH_BUFFER = []
    WIN_KEYMAP = {
        b"\xe0H": KEYMAP["up"] - ARROW_KEY_FLAG,
        b"\x00H": KEYMAP["up"] - ARROW_KEY_FLAG,
        b"\xe0P": KEYMAP["down"] - ARROW_KEY_FLAG,
        b"\x00P": KEYMAP["down"] - ARROW_KEY_FLAG,
        b"\xe0M": KEYMAP["right"] - ARROW_KEY_FLAG,
        b"\x00M": KEYMAP["right"] - ARROW_KEY_FLAG,
        b"\xe0K": KEYMAP["left"] - ARROW_KEY_FLAG,
        b"\x00K": KEYMAP["left"] - ARROW_KEY_FLAG,
    }

for i in range(10):
    KEYMAP[str(i)] = ord(str(i))


def get_raw_chars():
    "Gets raw characters from inputs"
    if os.name == "nt":
        import msvcrt
        encoding = "mbcs"
```

```
            # Flush the keyboard buffer
            while msvcrt.kbhit():
                msvcrt.getch()
            if len(WIN_CH_BUFFER) == 0:
                # Read the keystroke
                ch = msvcrt.getch()

                # If it is a prefix char, get second part
                if ch in (b"\x00", b"\xe0"):
                    ch2 = ch + msvcrt.getch()
                    # Translate actual Win chars to bullet char types
                    try:
                        chx = chr(WIN_KEYMAP[ch2])
                        WIN_CH_BUFFER.append(chr(KEYMAP["mod_int"]))
                        WIN_CH_BUFFER.append(chx)
                        if ord(chx) in (
                            KEYMAP["insert"] - 1 << 9,
                            KEYMAP["delete"] - 1 << 9,
                            KEYMAP["pg_up"] - 1 << 9,
                            KEYMAP["pg_down"] - 1 << 9,
                        ):
                            WIN_CH_BUFFER.append(chr(126))
                        ch = chr(KEYMAP["esc"])
                    except KeyError:
                        ch = ch2[1]
                else:
                    ch = ch.decode(encoding)
            else:
                ch = WIN_CH_BUFFER.pop(0)
        elif os.name == "posix":
            import termios
            import tty

            fd = sys.stdin.fileno()
            old_settings = termios.tcgetattr(fd)
            try:
                tty.setraw(fd)
                ch = sys.stdin.read(1)
            finally:
                termios.tcsetattr(fd, termios.TCSADRAIN, old_settings)
        return ch


    def get_character():
        "Gets a character from the keyboard and returns the key code"
        char = get_raw_chars()
        if ord(char) in [KEYMAP["interrupt"], KEYMAP["newline"]]:
            return char

        elif ord(char) == KEYMAP["esc"]:
            combo = get_raw_chars()
            if ord(combo) == KEYMAP["mod_int"]:
                key = get_raw_chars()
                if ord(key) >= KEYMAP["arrow_begin"] - ARROW_KEY_FLAG and ord(key) <= KEYMAP["arrow_end"] -
                    return chr(ord(key) + ARROW_KEY_FLAG)
                else:
                    return KEYMAP["undefined"]
            else:
                return get_raw_chars()

        else:
            if char in string.printable:
                return char
            else:
                return KEYMAP["undefined"]
```

accelerate-main/src/accelerate/commands/config/__init__.py

```
#!/usr/bin/env python

# Copyright 2021 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import argparse

from .config import config_command_parser
from .config_args import default_config_file, load_config_from_file  # noqa: F401
from .default import default_command_parser
from .update import update_command_parser


def get_config_parser(subparsers=None):
    parent_parser = argparse.ArgumentParser(add_help=False, allow_abbrev=False)
    # The main config parser
    config_parser = config_command_parser(subparsers)
    # The subparser to add commands to
    subcommands = config_parser.add_subparsers(title="subcommands", dest="subcommand")

    # Then add other parsers with the parent parser
    default_command_parser(subcommands, parents=[parent_parser])
    update_command_parser(subcommands, parents=[parent_parser])

    return config_parser


def main():
    config_parser = get_config_parser()
    args = config_parser.parse_args()

    if not hasattr(args, "func"):
        config_parser.print_help()
        exit(1)

    # Run
    args.func(args)


if __name__ == "__main__":
    main()
```

accelerate-main/src/accelerate/commands/menu/input.py

```python
# Copyright 2022 The HuggingFace Team and Brian Chao. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

"""
This file contains utilities for handling input from the user and registering specific keys to specific
based on https://github.com/bchao1/bullet
"""

from typing import List

from .keymap import KEYMAP, get_character


def mark(key: str):
    """
    Mark the function with the key code so it can be handled in the register
    """

    def decorator(func):
        handle = getattr(func, "handle_key", [])
        handle += [key]
        setattr(func, "handle_key", handle)
        return func

    return decorator


def mark_multiple(*keys: List[str]):
    """
    Mark the function with the key codes so it can be handled in the register
    """

    def decorator(func):
        handle = getattr(func, "handle_key", [])
        handle += keys
        setattr(func, "handle_key", handle)
        return func

    return decorator


class KeyHandler(type):
    """
    Metaclass that adds the key handlers to the class
    """

    def __new__(cls, name, bases, attrs):
        new_cls = super().__new__(cls, name, bases, attrs)
        if not hasattr(new_cls, "key_handler"):
            setattr(new_cls, "key_handler", {})
        setattr(new_cls, "handle_input", KeyHandler.handle_input)

        for value in attrs.values():
            handled_keys = getattr(value, "handle_key", [])
            for key in handled_keys:
                new_cls.key_handler[key] = value
        return new_cls
    @staticmethod
    def handle_input(cls):
        "Finds and returns the selected character if it exists in the handler"
        char = get_character()
        if char != KEYMAP["undefined"]:
            char = ord(char)
        handler = cls.key_handler.get(char)
        if handler:
            cls.current_selection = char
            return handler(cls)
        else:
            return None


def register(cls):
    """Adds KeyHandler metaclass to the class"""
    return KeyHandler(cls.__name__, cls.__bases__, cls.__dict__.copy())
```

accelerate-main/src/accelerate/commands/menu/selection_menu.py

```
# Copyright 2022 The HuggingFace Team and Brian Chao. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

"""
Main driver for the selection menu, based on https://github.com/bchao1/bullet
"""
import sys

from . import cursor, input
from .helpers import Direction, clear_line, forceWrite, linebreak, move_cursor, reset_cursor, writeColor
from .keymap import KEYMAP


@input.register
class BulletMenu:
    """
    A CLI menu to select a choice from a list of choices using the keyboard.
    """

    def __init__(self, prompt: str = None, choices: list = []):
        self.position = 0
        self.choices = choices
        self.prompt = prompt
        if sys.platform == "win32":
            self.arrow_char = "*"
        else:
            self.arrow_char = "➔ "

    def write_choice(self, index, end: str = ""):
        if sys.platform != "win32":
            writeColor(self.choices[index], 32, end)
        else:
            forceWrite(self.choices[index], end)

    def print_choice(self, index: int):
        "Prints the choice at the given index"
        if index == self.position:
            forceWrite(f" {self.arrow_char} ")
            self.write_choice(index)
        else:
            forceWrite(f"    {self.choices[index]}")
        reset_cursor()

    def move_direction(self, direction: Direction, num_spaces: int = 1):
        "Should not be directly called, used to move a direction of either up or down"
        old_position = self.position
        if direction == Direction.DOWN:
            if self.position + 1 >= len(self.choices):
                return
            self.position += num_spaces
        else:
            if self.position - 1 < 0:
                return
            self.position -= num_spaces
        clear_line()
        self.print_choice(old_position)
        move_cursor(num_spaces, direction.name)
        self.print_choice(self.position)

    @input.mark(KEYMAP["up"])
    def move_up(self):
        self.move_direction(Direction.UP)

    @input.mark(KEYMAP["down"])
    def move_down(self):
        self.move_direction(Direction.DOWN)

    @input.mark(KEYMAP["newline"])
    def select(self):
        move_cursor(len(self.choices) - self.position, "DOWN")
        return self.position

    @input.mark(KEYMAP["interrupt"])
    def interrupt(self):
        move_cursor(len(self.choices) - self.position, "DOWN")
        raise KeyboardInterrupt

    @input.mark_multiple(*[KEYMAP[str(number)] for number in range(10)])
    def select_row(self):
        index = int(chr(self.current_selection))
        movement = index - self.position
        if index == self.position:
            return
        if index < len(self.choices):
            if self.position > index:
                self.move_direction(Direction.UP, -movement)
            elif self.position < index:
                self.move_direction(Direction.DOWN, movement)
            else:
                return
        else:
            return

    def run(self, default_choice: int = 0):
        "Start the menu and return the selected choice"
        if self.prompt:
            linebreak()
            forceWrite(self.prompt, "\n")
            forceWrite("Please select a choice using the arrow or number keys, and selecting with enter"
        self.position = default_choice
        for i in range(len(self.choices)):
            self.print_choice(i)
            forceWrite("\n")
        move_cursor(len(self.choices) - self.position, "UP")
        with cursor.hide():
            while True:
                choice = self.handle_input()
                if choice is not None:
                    reset_cursor()
                    for _ in range(len(self.choices) + 1):
                        move_cursor(1, "UP")
                        clear_line()
                    self.write_choice(choice, "\n")
                    return choice
```

accelerate-main/src/accelerate/commands/menu/helpers.py

```
# Copyright 2022 The HuggingFace Team and Brian Chao. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

"""
A variety of helper functions and constants when dealing with terminal menu choices, based on
https://github.com/bchao1/bullet
"""

import enum
import shutil
import sys


TERMINAL_WIDTH, _ = shutil.get_terminal_size()

CURSOR_TO_CHAR = {"UP": "A", "DOWN": "B", "RIGHT": "C", "LEFT": "D"}


class Direction(enum.Enum):
    UP = 0
    DOWN = 1


def forceWrite(content, end=""):
    sys.stdout.write(str(content) + end)
    sys.stdout.flush()


def writeColor(content, color, end=""):
    forceWrite(f"\u001b[{color}m{content}\u001b[0m", end)


def reset_cursor():
    forceWrite("\r")


def move_cursor(num_lines: int, direction: str):
    forceWrite(f"\033[{num_lines}{CURSOR_TO_CHAR[direction.upper()]}")


def clear_line():
    forceWrite(" " * TERMINAL_WIDTH)
    reset_cursor()


def linebreak():
    reset_cursor()
    forceWrite("-" * TERMINAL_WIDTH)
```

accelerate-main/src/accelerate/commands/menu/cursor.py

```
# Copyright 2022 The HuggingFace Team and Brian Chao. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

"""
A utility for showing and hiding the terminal cursor on Windows and Linux, based on https://github.com/b
"""

import os
import sys
from contextlib import contextmanager


# Windows only
if os.name == "nt":
    import ctypes
    import msvcrt  # noqa

    class CursorInfo(ctypes.Structure):
        # _fields is a specific attr expected by ctypes
        _fields_ = [("size", ctypes.c_int), ("visible", ctypes.c_byte)]


def hide_cursor():
    if os.name == "nt":
        ci = CursorInfo()
        handle = ctypes.windll.kernel32.GetStdHandle(-11)
        ctypes.windll.kernel32.GetConsoleCursorInfo(handle, ctypes.byref(ci))
        ci.visible = False
        ctypes.windll.kernel32.SetConsoleCursorInfo(handle, ctypes.byref(ci))
    elif os.name == "posix":
        sys.stdout.write("\033[?25l")
        sys.stdout.flush()


def show_cursor():
    if os.name == "nt":
        ci = CursorInfo()
        handle = ctypes.windll.kernel32.GetStdHandle(-11)
        ctypes.windll.kernel32.GetConsoleCursorInfo(handle, ctypes.byref(ci))
        ci.visible = True
        ctypes.windll.kernel32.SetConsoleCursorInfo(handle, ctypes.byref(ci))
    elif os.name == "posix":
        sys.stdout.write("\033[?25h")
        sys.stdout.flush()


@contextmanager
def hide():
    "Context manager to hide the terminal cursor"
    try:
        hide_cursor()
        yield
    finally:
        show_cursor()
```

accelerate-main/src/accelerate/commands/config/__init__.py

```
#!/usr/bin/env python

# Copyright 2021 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import argparse

from .config import config_command_parser
from .config_args import default_config_file, load_config_from_file  # noqa: F401
from .default import default_command_parser
from .update import update_command_parser


def get_config_parser(subparsers=None):
    parent_parser = argparse.ArgumentParser(add_help=False, allow_abbrev=False)
    # The main config parser
    config_parser = config_command_parser(subparsers)
    # The subparser to add commands to
    subcommands = config_parser.add_subparsers(title="subcommands", dest="subcommand")

    # Then add other parsers with the parent parser
    default_command_parser(subcommands, parents=[parent_parser])
    update_command_parser(subcommands, parents=[parent_parser])

    return config_parser


def main():
    config_parser = get_config_parser()
    args = config_parser.parse_args()

    if not hasattr(args, "func"):
        config_parser.print_help()
        exit(1)

    # Run
    args.func(args)


if __name__ == "__main__":
    main()
```

accelerate-main/src/accelerate/commands/config/config.py

```
#!/usr/bin/env python

# Copyright 2021 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import argparse
import os

from accelerate.utils import ComputeEnvironment

from .cluster import get_cluster_input
from .config_args import cache_dir, default_config_file, default_yaml_config_file, load_config_from_file
from .config_utils import _ask_field, _ask_options, _convert_compute_environment  # noqa: F401
from .sagemaker import get_sagemaker_input


description = "Launches a series of prompts to create and save a `default_config.yaml` configuration fil


def get_user_input():
    compute_environment = _ask_options(
        "In which compute environment are you running?",
        ["This machine", "AWS (Amazon SageMaker)"],
        _convert_compute_environment,
    )
    if compute_environment == ComputeEnvironment.AMAZON_SAGEMAKER:
        config = get_sagemaker_input()
    else:
        config = get_cluster_input()
    return config


def config_command_parser(subparsers=None):
    if subparsers is not None:
        parser = subparsers.add_parser("config", description=description)
    else:
        parser = argparse.ArgumentParser("Accelerate config command", description=description)

    parser.add_argument(
        "--config_file",
        default=None,
        help=(
            "The path to use to store the config file. Will default to a file named default_config.yaml
            "location, which is the content of the environment `HF_HOME` suffixed with 'accelerate', or
            "such an environment variable, your cache directory ('~/.cache' or the content of `XDG_CACHE
            "with 'huggingface'."
        ),
    )

    if subparsers is not None:
        parser.set_defaults(func=config_command)
    return parser


def config_command(args):
    config = get_user_input()
    if args.config_file is not None:
        config_file = args.config_file
    else:
        if not os.path.isdir(cache_dir):
            os.makedirs(cache_dir)
        config_file = default_yaml_config_file

    if config_file.endswith(".json"):
        config.to_json_file(config_file)
    else:
        config.to_yaml_file(config_file)
    print(f"accelerate configuration saved at {config_file}")


def main():
    parser = config_command_parser()
    args = parser.parse_args()
    config_command(args)


if __name__ == "__main__":
    main()
```

accelerate-main/src/accelerate/commands/config/default.py

```python
#!/usr/bin/env python

# Copyright 2021 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

from pathlib import Path

import torch

from .config_args import ClusterConfig, default_json_config_file
from .config_utils import SubcommandHelpFormatter


description = "Create a default config file for Accelerate with only a few flags set."


def write_basic_config(mixed_precision="no", save_location: str = default_json_config_file, dynamo_backe
    """
    Creates and saves a basic cluster config to be used on a local machine with potentially multiple GPU
    set CPU if it is a CPU-only machine.

    Args:
        mixed_precision (`str`, *optional*, defaults to "no"):
            Mixed Precision to use. Should be one of "no", "fp16", or "bf16"
        save_location (`str`, *optional*, defaults to `default_json_config_file`):
            Optional custom save location. Should be passed to `--config_file` when using `accelerate la
            location is inside the huggingface cache folder (`~/.cache/huggingface`) but can be override
            the `HF_HOME` environmental variable, followed by `accelerate/default_config.yaml`.
    """
    path = Path(save_location)
    path.parent.mkdir(parents=True, exist_ok=True)
    if path.exists():
        print(
            f"Configuration already exists at {save_location}, will not override. Run `accelerate config
        )
        return False
    mixed_precision = mixed_precision.lower()
    if mixed_precision not in ["no", "fp16", "bf16"]:
        raise ValueError(f"`mixed_precision` should be one of 'no', 'fp16', or 'bf16'. Received {mixed_p
    config = {
        "compute_environment": "LOCAL_MACHINE",
        "mixed_precision": mixed_precision,
    }
    if torch.cuda.is_available():
        num_gpus = torch.cuda.device_count()
        config["num_processes"] = num_gpus
        config["use_cpu"] = False
        if num_gpus > 1:
            config["distributed_type"] = "MULTI_GPU"
        else:
            config["distributed_type"] = "NO"
    else:
        num_gpus = 0
        config["use_cpu"] = True
        config["num_processes"] = 1
        config["distributed_type"] = "NO"
    config = ClusterConfig(**config)
    config.to_json_file(path)
    return path


def default_command_parser(parser, parents):
    parser = parser.add_parser("default", parents=parents, help=description, formatter_class=SubcommandH
    parser.add_argument(
        "--config_file",
        default=default_json_config_file,
        help=(
            "The path to use to store the config file. Will default to a file named default_config.yaml
            "location, which is the content of the environment `HF_HOME` suffixed with 'accelerate', or
            "such an environment variable, your cache directory ('~/.cache' or the content of `XDG_CACHE
            "with 'huggingface'."
        ),
        dest="save_location",
    )

    parser.add_argument(
        "--mixed_precision",
        choices=["no", "fp16", "bf16"],
        type=str,
        help="Whether or not to use mixed precision training. "
        "Choose between FP16 and BF16 (bfloat16) training. "
        "BF16 training is only supported on Nvidia Ampere GPUs and PyTorch 1.10 or later.",
        default="no",
    )
    parser.set_defaults(func=default_config_command)
    return parser


def default_config_command(args):
    config_file = write_basic_config(args.mixed_precision, args.save_location)
    if config_file:
        print(f"accelerate configuration saved at {config_file}")
```

accelerate-main/src/accelerate/commands/config/config_args.py

```python
#!/usr/bin/env python

# Copyright 2021 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import json
import os
from dataclasses import dataclass
from enum import Enum
from typing import List, Optional, Union

import yaml

from ...utils import ComputeEnvironment, DistributedType, SageMakerDistributedType
from ...utils.constants import SAGEMAKER_PYTHON_VERSION, SAGEMAKER_PYTORCH_VERSION, SAGEMAKER_TRANSFORME


hf_cache_home = os.path.expanduser(
    os.getenv("HF_HOME", os.path.join(os.getenv("XDG_CACHE_HOME", "~/.cache"), "huggingface"))
)
cache_dir = os.path.join(hf_cache_home, "accelerate")
default_json_config_file = os.path.join(cache_dir, "default_config.yaml")
default_yaml_config_file = os.path.join(cache_dir, "default_config.yaml")

# For backward compatibility: the default config is the json one if it's the only existing file.
if os.path.isfile(default_yaml_config_file) or not os.path.isfile(default_json_config_file):
    default_config_file = default_yaml_config_file
else:
    default_config_file = default_json_config_file


def load_config_from_file(config_file):
    if config_file is not None:
        if not os.path.isfile(config_file):
            raise FileNotFoundError(
                f"The passed configuration file `{config_file}` does not exist. "
                "Please pass an existing file to `accelerate launch`, or use the the default one "
                "created through `accelerate config` and run `accelerate launch` "
                "without the `--config_file` argument."
            )
    else:
        config_file = default_config_file
    with open(config_file, "r", encoding="utf-8") as f:
        if config_file.endswith(".json"):
            if (
                json.load(f).get("compute_environment", ComputeEnvironment.LOCAL_MACHINE)
                == ComputeEnvironment.LOCAL_MACHINE
            ):
                config_class = ClusterConfig
            else:
                config_class = SageMakerConfig
            return config_class.from_json_file(json_file=config_file)
        else:
            if (
                yaml.safe_load(f).get("compute_environment", ComputeEnvironment.LOCAL_MACHINE)
                == ComputeEnvironment.LOCAL_MACHINE
            ):
```

```
            config_class = ClusterConfig
        else:
            config_class = SageMakerConfig
        return config_class.from_yaml_file(yaml_file=config_file)


@dataclass
class BaseConfig:
    compute_environment: ComputeEnvironment
    distributed_type: Union[DistributedType, SageMakerDistributedType]
    mixed_precision: str
    use_cpu: bool

    def to_dict(self):
        result = self.__dict__
        # For serialization, it's best to convert Enums to strings (or their underlying value type).
        for key, value in result.items():
            if isinstance(value, Enum):
                result[key] = value.value
        result = {k: v for k, v in result.items() if v is not None}
        return result

    @classmethod
    def from_json_file(cls, json_file=None):
        json_file = default_json_config_file if json_file is None else json_file
        with open(json_file, "r", encoding="utf-8") as f:
            config_dict = json.load(f)
        if "compute_environment" not in config_dict:
            config_dict["compute_environment"] = ComputeEnvironment.LOCAL_MACHINE
        if "mixed_precision" not in config_dict:
            config_dict["mixed_precision"] = "fp16" if ("fp16" in config_dict and config_dict["fp16"]) e
        if "fp16" in config_dict:  # Convert the config to the new format.
            del config_dict["fp16"]
        if "dynamo_backend" in config_dict:  # Convert the config to the new format.
            dynamo_backend = config_dict.pop("dynamo_backend")
            config_dict["dynamo_config"] = {} if dynamo_backend == "NO" else {"dynamo_backend": dynamo_b
        if "use_cpu" not in config_dict:
            config_dict["use_cpu"] = False
        return cls(**config_dict)

    def to_json_file(self, json_file):
        with open(json_file, "w", encoding="utf-8") as f:
            content = json.dumps(self.to_dict(), indent=2, sort_keys=True) + "\n"
            f.write(content)

    @classmethod
    def from_yaml_file(cls, yaml_file=None):
        yaml_file = default_yaml_config_file if yaml_file is None else yaml_file
        with open(yaml_file, "r", encoding="utf-8") as f:
            config_dict = yaml.safe_load(f)
        if "compute_environment" not in config_dict:
            config_dict["compute_environment"] = ComputeEnvironment.LOCAL_MACHINE

        if "mixed_precision" not in config_dict:
            config_dict["mixed_precision"] = "fp16" if ("fp16" in config_dict and config_dict["fp16"]) e
        if "fp16" in config_dict:  # Convert the config to the new format.
            del config_dict["fp16"]
        if "dynamo_backend" in config_dict:  # Convert the config to the new format.
            dynamo_backend = config_dict.pop("dynamo_backend")
            config_dict["dynamo_config"] = {} if dynamo_backend == "NO" else {"dynamo_backend": dynamo_b
        if "use_cpu" not in config_dict:
            config_dict["use_cpu"] = False
        return cls(**config_dict)

    def to_yaml_file(self, yaml_file):
        with open(yaml_file, "w", encoding="utf-8") as f:
            yaml.safe_dump(self.to_dict(), f)

    def __post_init__(self):
        if isinstance(self.compute_environment, str):
            self.compute_environment = ComputeEnvironment(self.compute_environment)
        if isinstance(self.distributed_type, str):
            if self.compute_environment == ComputeEnvironment.AMAZON_SAGEMAKER:
                self.distributed_type = SageMakerDistributedType(self.distributed_type)
            else:
                self.distributed_type = DistributedType(self.distributed_type)
        if self.dynamo_config is None:
            self.dynamo_config = {}


@dataclass
class ClusterConfig(BaseConfig):
    num_processes: int
    machine_rank: int = 0
    num_machines: int = 1
    gpu_ids: Optional[str] = None
    main_process_ip: Optional[str] = None
    main_process_port: Optional[int] = None
    rdzv_backend: Optional[str] = "static"
    same_network: Optional[bool] = False
    main_training_function: str = "main"

    # args for deepspeed_plugin
    deepspeed_config: dict = None
    # args for fsdp
    fsdp_config: dict = None
    # args for megatron_lm
    megatron_lm_config: dict = None
    # args for TPU
    downcast_bf16: bool = False

    # args for TPU pods
    tpu_name: str = None
    tpu_zone: str = None
    tpu_use_cluster: bool = False
    tpu_use_sudo: bool = False
    command_file: str = None
    commands: List[str] = None
    tpu_vm: List[str] = None
    tpu_env: List[str] = None

    # args for dynamo
    dynamo_config: dict = None

    def __post_init__(self):
        if self.deepspeed_config is None:
            self.deepspeed_config = {}
        if self.fsdp_config is None:
            self.fsdp_config = {}
        if self.megatron_lm_config is None:
            self.megatron_lm_config = {}
        return super().__post_init__()


@dataclass
class SageMakerConfig(BaseConfig):
    ec2_instance_type: str
    iam_role_name: str
    image_uri: Optional[str] = None
    profile: Optional[str] = None
    region: str = "us-east-1"
    num_machines: int = 1
    gpu_ids: str = "all"
    base_job_name: str = f"accelerate-sagemaker-{num_machines}"
    pytorch_version: str = SAGEMAKER_PYTORCH_VERSION
    transformers_version: str = SAGEMAKER_TRANSFORMERS_VERSION
    py_version: str = SAGEMAKER_PYTHON_VERSION
    sagemaker_inputs_file: str = None
    sagemaker_metrics_file: str = None
    additional_args: dict = None
    dynamo_config: dict = None
```

accelerate-main/docs/source/usage_guides/sagemaker.mdx

# Amazon SageMaker

Hugging Face and Amazon introduced new [Hugging Face Deep Learning Containers (DLCs)](https://github.com
make it easier than ever to train Hugging Face Transformer models in [Amazon SageMaker](https://aws.amaz

## Getting Started

### Setup & Installation

Before you can run your ■ Accelerate scripts on Amazon SageMaker you need to sign up for an AWS account.
have an AWS account yet learn more [here](https://docs.aws.amazon.com/sagemaker/latest/dg/gs-set-up.html

After you have your AWS Account you need to install the `sagemaker` sdk for ■ Accelerate with:

```bash
pip install "accelerate[sagemaker]" --upgrade
```

■ Accelerate currently uses the ■ DLCs, with `transformers`, `datasets` and `tokenizers` pre-installed.
Accelerate is not in the DLC yet (will soon be added!) so to use it within Amazon SageMaker you need to
`requirements.txt` in the same directory where your training script is located and add it as dependency:

```
accelerate
```

You should also add any other dependencies you have to this `requirements.txt`.

### Configure ■ Accelerate

You can configure the launch configuration for Amazon SageMaker the same as you do for non SageMaker tra
the ■ Accelerate CLI:

```bash
accelerate config
# In which compute environment are you running? ([0] This machine, [1] AWS (Amazon SageMaker)): 1
```

■ Accelerate will go through a questionnaire about your Amazon SageMaker setup and create a config file

<Tip>

    ■ Accelerate is not saving any of your credentials.

</Tip>

### Prepare a ■ Accelerate fine-tuning script

The training script is very similar to a training script you might run outside of SageMaker, but to save
after training you need to specify either `/opt/ml/model` or use `os.environ["SM_MODEL_DIR"]` as your sa
directory. After training, artifacts in this directory are uploaded to S3:

```diff
```

- torch.save('/opt/ml/model`)
+ accelerator.save('/opt/ml/model')
```

<Tip warning={true}>

    SageMaker doesn’t support argparse actions. If you want to use, for example, boolean hyperparameters
    specify type as bool in your script and provide an explicit True or False value for this hyperparame

</Tip>

### Launch Training

You can launch your training with ■ Accelerate CLI with:

```
accelerate launch path_to_script.py --args_to_the_script

This will launch your training script using your configuration. The only thing you have to do is provide
arguments needed by your training script as named arguments.

Examples

<Tip>

    If you run one of the example scripts, don't forget to add `accelerator.save('/opt/ml/model')` to it

</Tip>

```bash
accelerate launch ./examples/sagemaker_example.py
```

Outputs:

```
Configuring Amazon SageMaker environment
Converting Arguments to Hyperparameters
Creating Estimator
2021-04-08 11:56:50 Starting - Starting the training job...
2021-04-08 11:57:13 Starting - Launching requested ML instancesProfilerReport-1617883008: InProgress
.........
2021-04-08 11:58:54 Starting - Preparing the instances for training.........
2021-04-08 12:00:24 Downloading - Downloading input data
2021-04-08 12:00:24 Training - Downloading the training image..................
2021-04-08 12:03:39 Training - Training image download completed. Training in progress..
........
epoch 0: {'accuracy': 0.7598039215686274, 'f1': 0.8178438661710037}
epoch 1: {'accuracy': 0.8357843137254902, 'f1': 0.882249560632689}
epoch 2: {'accuracy': 0.8406862745098039, 'f1': 0.8869565217391304}
........
2021-04-08 12:05:40 Uploading - Uploading generated training model
2021-04-08 12:05:40 Completed - Training job completed
Training seconds: 331
Billable seconds: 331
You can find your model data at: s3://your-bucket/accelerate-sagemaker-1-2021-04-08-11-56-47-108/output/
```

## Advanced Features

### Distributed Training: Data Parallelism

Set up the accelerate config by running `accelerate config` and answer the SageMaker questions and set i
To use SageMaker DDP, select it when asked
`What is the distributed mode? ([0] No distributed training, [1] data parallelism):`.
Example config below:
```yaml
base_job_name: accelerate-sagemaker-1
compute_environment: AMAZON_SAGEMAKER
distributed_type: DATA_PARALLEL
ec2_instance_type: ml.p3.16xlarge
```

~~~~
iam_role_name: xxxxx
image_uri: null
mixed_precision: fp16
num_machines: 1
profile: xxxxx
py_version: py38
pytorch_version: 1.10.2
region: us-east-1
transformers_version: 4.17.0
use_cpu: false
```

### Distributed Training: Model Parallelism

*currently in development, will be supported soon.*

### Python packages and dependencies

■ Accelerate currently uses the ■ DLCs, with `transformers`, `datasets` and `tokenizers` pre-installed.
want to use different/other Python packages you can do this by adding them to the `requirements.txt`. Th
will be installed before your training script is started.

### Local Training: SageMaker Local mode

The local mode in the SageMaker SDK allows you to run your training script locally inside the HuggingFac
or using your custom container image. This is useful for debugging and testing your training script insi
Local mode uses Docker compose (*Note: Docker Compose V2 is not supported yet*). The SDK will handle the
to pull the DLC to your local environment. You can emulate CPU (single and multi-instance) and GPU (sing

To use local mode, you need to set your `ec2_instance_type` to `local`.

```yaml
ec2_instance_type: local
```

### Advanced configuration

The configuration allows you to override parameters for the [Estimator](https://sagemaker.readthedocs.io
These settings have to be applied in the config file and are not part of `accelerate config`. You can co

```yaml
additional_args:
  # enable network isolation to restrict internet access for containers
  enable_network_isolation: True
```

You can find all available configuration [here](https://sagemaker.readthedocs.io/en/stable/api/training/

### Use Spot Instances

You can use Spot Instances e.g. using (see [Advanced configuration](#advanced-configuration)):
```yaml
additional_args:
  use_spot_instances: True
  max_wait: 86400
```

*Note: Spot Instances are subject to be terminated and training to be continued from a checkpoint. This

### Remote scripts: Use scripts located on Github

*undecided if feature is needed. Contact us if you would like this feature.*
~~~~

accelerate-main/src/accelerate/commands/config/update.py

```
#!/usr/bin/env python

# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

from pathlib import Path

from .config_args import default_config_file, load_config_from_file
from .config_utils import SubcommandHelpFormatter


description = "Update an existing config file with the latest defaults while maintaining the old configu


def update_config(args):
    """
    Update an existing config file with the latest defaults while maintaining the old configuration.
    """
    config_file = args.config_file
    if config_file is None and Path(default_config_file).exists():
        config_file = default_config_file
    elif not Path(config_file).exists():
        raise ValueError(f"The passed config file located at {config_file} doesn't exist.")
    config = load_config_from_file(config_file)

    if config_file.endswith(".json"):
        config.to_json_file(config_file)
    else:
        config.to_yaml_file(config_file)
    return config_file


def update_command_parser(parser, parents):
    parser = parser.add_parser("update", parents=parents, help=description, formatter_class=SubcommandHe
    parser.add_argument(
        "--config_file",
        default=None,
        help=(
            "The path to the config file to update. Will default to a file named default_config.yaml in
            "location, which is the content of the environment `HF_HOME` suffixed with 'accelerate', or
            "such an environment variable, your cache directory ('~/.cache' or the content of `XDG_CACHE
            "with 'huggingface'."
        ),
    )

    parser.set_defaults(func=update_config_command)
    return parser


def update_config_command(args):
    config_file = update_config(args)
    print(f"Sucessfully updated the configuration file at {config_file}.")
```

accelerate-main/src/accelerate/commands/config/cluster.py

```
#!/usr/bin/env python

# Copyright 2021 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import os

from ...utils import (
    ComputeEnvironment,
    DistributedType,
    is_deepspeed_available,
    is_mps_available,
    is_transformers_available,
)
from ...utils.constants import (
    DEEPSPEED_MULTINODE_LAUNCHERS,
    FSDP_AUTO_WRAP_POLICY,
    FSDP_BACKWARD_PREFETCH,
    FSDP_SHARDING_STRATEGY,
    FSDP_STATE_DICT_TYPE,
    TORCH_DYNAMO_MODES,
)
from .config_args import ClusterConfig
from .config_utils import (
    DYNAMO_BACKENDS,
    _ask_field,
    _ask_options,
    _convert_distributed_mode,
    _convert_dynamo_backend,
    _convert_mixed_precision,
    _convert_yes_no_to_bool,
)


def get_cluster_input():
    distributed_type = _ask_options(
        "Which type of machine are you using?",
        ["No distributed training", "multi-CPU", "multi-GPU", "TPU"],
        _convert_distributed_mode,
    )

    machine_rank = 0
    num_machines = 1
    num_processes = 1
    gpu_ids = None
    main_process_ip = None
    main_process_port = None
    rdzv_backend = "static"
    same_network = True

    if distributed_type in [DistributedType.MULTI_GPU, DistributedType.MULTI_CPU]:
        num_machines = _ask_field(
            "How many different machines will you use (use more than 1 for multi-node training)? [1]: ",
            int,
            default=1,
        )
        if num_machines > 1:
```

```
        machine_rank = _ask_options(
            "What is the rank of this machine?",
            list(range(num_machines)),
            int,
        )
        main_process_ip = _ask_field(
            "What is the IP address of the machine that will host the main process? ",
        )
        main_process_port = _ask_field(
            "What is the port you will use to communicate with the main process? ",
            int,
        )
        same_network = _ask_field(
            "Are all the machines on the same local network? Answer `no` if nodes are on the cloud a
            _convert_yes_no_to_bool,
            default=True,
            error_message="Please enter yes or no.",
        )
        if not same_network:
            rdzv_backend = _ask_field(
                "What rendezvous backend will you use? ('static', 'c10d', ...): ", default="static"
            )

if distributed_type == DistributedType.NO:
    use_cpu = _ask_field(
        "Do you want to run your training on CPU only (even if a GPU / Apple Silicon device is avail
        _convert_yes_no_to_bool,
        default=False,
        error_message="Please enter yes or no.",
    )
elif distributed_type == DistributedType.MULTI_CPU:
    use_cpu = True
else:
    use_cpu = False

dynamo_config = {}
use_dynamo = _ask_field(
    "Do you wish to optimize your script with torch dynamo?[yes/NO]:",
    _convert_yes_no_to_bool,
    default=False,
    error_message="Please enter yes or no.",
)
if use_dynamo:
    prefix = "dynamo_"
    dynamo_config[prefix + "backend"] = _ask_options(
        "Which dynamo backend would you like to use?",
        [x.lower() for x in DYNAMO_BACKENDS],
        _convert_dynamo_backend,
        default=2,
    )
    use_custom_options = _ask_field(
        "Do you want to customize the defaults sent to torch.compile? [yes/NO]: ",
        _convert_yes_no_to_bool,
        default=False,
        error_message="Please enter yes or no.",
    )

    if use_custom_options:
        dynamo_config[prefix + "mode"] = _ask_options(
            "Which mode do you want to use?",
            TORCH_DYNAMO_MODES,
            lambda x: TORCH_DYNAMO_MODES[int(x)],
            default=0,
        )
        dynamo_config[prefix + "use_fullgraph"] = _ask_field(
            "Do you want the fullgraph mode or it is ok to break model into several subgraphs? [yes/
            _convert_yes_no_to_bool,
            default=False,
            error_message="Please enter yes or no.",
        )
        dynamo_config[prefix + "use_dynamic"] = _ask_field(
            "Do you want to enable dynamic shape tracing? [yes/NO]: ",
            _convert_yes_no_to_bool,
            default=False,
            error_message="Please enter yes or no.",
        )

use_mps = not use_cpu and is_mps_available()
deepspeed_config = {}
if distributed_type in [DistributedType.MULTI_GPU, DistributedType.NO] and not use_mps:
    use_deepspeed = _ask_field(
        "Do you want to use DeepSpeed? [yes/NO]: ",
        _convert_yes_no_to_bool,
        default=False,
        error_message="Please enter yes or no.",
    )
    if use_deepspeed:
        distributed_type = DistributedType.DEEPSPEED
        assert (
            is_deepspeed_available()
        ), "DeepSpeed is not installed => run `pip3 install deepspeed` or build it from source"

    if distributed_type == DistributedType.DEEPSPEED:
        use_deepspeed_config = _ask_field(
            "Do you want to specify a json file to a DeepSpeed config? [yes/NO]: ",
            _convert_yes_no_to_bool,
            default=False,
            error_message="Please enter yes or no.",
        )
        if use_deepspeed_config:
            deepspeed_config["deepspeed_config_file"] = _ask_field(
                "Please enter the path to the json DeepSpeed config file: ",
                str,
                default="none",
            )
        else:
            deepspeed_config["zero_stage"] = _ask_options(
                "What should be your DeepSpeed's ZeRO optimization stage?",
                [0, 1, 2, 3],
                int,
                default=2,
            )

            deepspeed_devices = ["none", "cpu", "nvme"]
            if deepspeed_config["zero_stage"] >= 2:
                deepspeed_config["offload_optimizer_device"] = _ask_options(
                    "Where to offload optimizer states?", deepspeed_devices, lambda x: deepspeed_dev
                )
                deepspeed_config["offload_param_device"] = _ask_options(
                    "Where to offload parameters?", deepspeed_devices, lambda x: deepspeed_devices[i
                )
            deepspeed_config["gradient_accumulation_steps"] = _ask_field(
                "How many gradient accumulation steps you're passing in your script? [1]: ",
                int,
                default=1,
            )
            use_gradient_clipping = _ask_field(
                "Do you want to use gradient clipping? [yes/NO]: ",
                _convert_yes_no_to_bool,
                default=False,
                error_message="Please enter yes or no.",
            )
            if use_gradient_clipping:
                deepspeed_config["gradient_clipping"] = _ask_field(
                    "What is the gradient clipping value? [1.0]: ",
                    float,
                    default=1.0,
                )
            if deepspeed_config["zero_stage"] == 3:
                deepspeed_config["zero3_save_16bit_model"] = _ask_field(
                    "Do you want to save 16-bit model weights when using ZeRO Stage-3? [yes/NO]: ",
                    _convert_yes_no_to_bool,
                    default=False,
                    error_message="Please enter yes or no.",
                )
        deepspeed_config["zero3_init_flag"] = _ask_field(
            "Do you want to enable `deepspeed.zero.Init` when using ZeRO Stage-3 for constructing ma
            _convert_yes_no_to_bool,
            default=False,
            error_message="Please enter yes or no.",
        )
        if deepspeed_config["zero3_init_flag"]:
            if not is_transformers_available():
                raise Exception(
                    "When `zero3_init_flag` is set, it requires Transformers to be installed. "
                    "Please run `pip3 install transformers`."
                )

        if num_machines > 1:
            launcher_query = "Which Type of launcher do you want to use?"
            deepspeed_config["deepspeed_multinode_launcher"] = _ask_options(
                launcher_query,
                DEEPSPEED_MULTINODE_LAUNCHERS,
                lambda x: DEEPSPEED_MULTINODE_LAUNCHERS[int(x)],
            )

            if deepspeed_config["deepspeed_multinode_launcher"] != DEEPSPEED_MULTINODE_LAUNCHERS[1]:
                deepspeed_config["deepspeed_hostfile"] = _ask_field(
                    "DeepSpeed configures multi-node compute resources with hostfile. "
                    "Each row is of the format `hostname slots=[num_gpus]`, e.g., `localhost slots=2
                    "for more information please refer official [documentation]"
                    "(https://www.deepspeed.ai/getting-started/#resource-configuration-multi-node).
                    "Please specify the location of hostfile: ",
                    str,
                )

                is_exclusion_filter = _ask_field(
                    "Do you want to specify exclusion filter string? [yes/NO]: ",
                    _convert_yes_no_to_bool,
                    default=False,
                    error_message="Please enter yes or no.",
                )
                if is_exclusion_filter:
                    deepspeed_config["deepspeed_exclusion_filter"] = _ask_field(
                        "DeepSpeed exclusion filter string: ",
                        str,
                    )

                is_inclusion_filter = _ask_field(
                    "Do you want to specify inclusion filter string? [yes/NO]: ",
                    _convert_yes_no_to_bool,
                    default=False,
                    error_message="Please enter yes or no.",
                )
                if is_inclusion_filter:
                    deepspeed_config["deepspeed_inclusion_filter"] = _ask_field(
                        "DeepSpeed inclusion filter string: ",
                        str,
                    )

fsdp_config = {}
if distributed_type in [DistributedType.MULTI_GPU]:
    use_fsdp = _ask_field(
        "Do you want to use FullyShardedDataParallel? [yes/NO]: ",
        _convert_yes_no_to_bool,
        default=False,
        error_message="Please enter yes or no.",
    )
    if use_fsdp:
        distributed_type = DistributedType.FSDP
    if distributed_type == DistributedType.FSDP:
        sharding_strategy_query = "What should be your sharding strategy?"
        fsdp_config["fsdp_sharding_strategy"] = _ask_options(
            sharding_strategy_query,
            FSDP_SHARDING_STRATEGY,
            lambda x: int(x) + 1,
            default=1,
        )
        fsdp_config["fsdp_offload_params"] = _ask_field(
            "Do you want to offload parameters and gradients to CPU? [yes/NO]: ",
            _convert_yes_no_to_bool,
            default=False,
            error_message="Please enter yes or no.",
        )
        fsdp_wrap_query = "What should be your auto wrap policy?"
        fsdp_config["fsdp_auto_wrap_policy"] = _ask_options(
            fsdp_wrap_query,
            FSDP_AUTO_WRAP_POLICY,
            lambda x: FSDP_AUTO_WRAP_POLICY[int(x)],
        )
        if fsdp_config["fsdp_auto_wrap_policy"] == FSDP_AUTO_WRAP_POLICY[0]:
            fsdp_config["fsdp_transformer_layer_cls_to_wrap"] = _ask_field(
                "Specify the comma-separated list of transformer layer class names (case-sensitive)
                "`BertLayer`, `GPTJBlock`, `T5Block`, `BertLayer,BertEmbeddings,BertSelfOutput` ...?
                str,
            )
        elif fsdp_config["fsdp_auto_wrap_policy"] == FSDP_AUTO_WRAP_POLICY[1]:
            fsdp_config["fsdp_min_num_params"] = _ask_field(
                "What should be your FSDP's minimum number of parameters for Default Auto Wrapping P
                int,
                default=1e8,
            )
        fsdp_backward_prefetch_query = "What should be your FSDP's backward prefetch policy?"
        fsdp_config["fsdp_backward_prefetch_policy"] = _ask_options(
            fsdp_backward_prefetch_query,
            FSDP_BACKWARD_PREFETCH,
            lambda x: FSDP_BACKWARD_PREFETCH[int(x)],
        )
        fsdp_state_dict_type_query = "What should be your FSDP's state dict type?"
        fsdp_config["fsdp_state_dict_type"] = _ask_options(
            fsdp_state_dict_type_query,
            FSDP_STATE_DICT_TYPE,
            lambda x: FSDP_STATE_DICT_TYPE[int(x)],
        )

megatron_lm_config = {}
if distributed_type in [DistributedType.MULTI_GPU]:
    use_megatron_lm = _ask_field(
        "Do you want to use Megatron-LM ? [yes/NO]: ",
        _convert_yes_no_to_bool,
        default=False,
        error_message="Please enter yes or no.",
    )
    if use_megatron_lm:
        distributed_type = DistributedType.MEGATRON_LM
    if distributed_type == DistributedType.MEGATRON_LM:
        prefix = "megatron_lm_"
        megatron_lm_config[prefix + "tp_degree"] = _ask_field(
            "What is the Tensor Parallelism degree/size? [1]:",
            int,
            default=1,
            error_message="Please enter an integer.",
        )
        if megatron_lm_config[prefix + "tp_degree"] > 1:
            megatron_lm_config[prefix + "sequence_parallelism"] = _ask_field(
                "Do you want to enable Sequence Parallelism? [YES/no]: ",
                _convert_yes_no_to_bool,
                default=True,
                error_message="Please enter yes or no.",
            )

        megatron_lm_config[prefix + "pp_degree"] = _ask_field(
            "What is the Pipeline Parallelism degree/size? [1]:",
            int,
            default=1,
            error_message="Please enter an integer.",
        )
        if megatron_lm_config[prefix + "pp_degree"] > 1:
            megatron_lm_config[prefix + "num_micro_batches"] = _ask_field(
                "What is the number of micro-batches? [1]:",
                int,
                default=1,
                error_message="Please enter an integer.",
            )

        megatron_lm_config[prefix + "recompute_activations"] = _ask_field(
            "Do you want to enable selective activation recomputation? [YES/no]: ",
            _convert_yes_no_to_bool,
            default=True,
            error_message="Please enter yes or no.",
        )

        megatron_lm_config[prefix + "use_distributed_optimizer"] = _ask_field(
            "Do you want to use distributed optimizer "
            "which shards optimizer state and gradients across data pralellel ranks? [YES/no]: ",
            _convert_yes_no_to_bool,
            default=True,
            error_message="Please enter yes or no.",
        )

        megatron_lm_config[prefix + "gradient_clipping"] = _ask_field(
            "What is the gradient clipping value based on global L2 Norm (0 to disable)? [1.0]: ",
            float,
            default=1.0,
        )
# TPU specific defaults
tpu_commands = None
tpu_command_file = None
tpu_downcast_bf16 = "no"
tpu_env = []
tpu_name = None
tpu_vm = None
tpu_zone = None
tpu_use_sudo = False
tpu_use_cluster = False

if distributed_type in [DistributedType.MULTI_CPU, DistributedType.MULTI_GPU, DistributedType.TPU]:
    machine_type = str(distributed_type).split(".")[1].replace("MULTI_", "")
    if machine_type == "TPU":
        machine_type += " cores"
    else:
        machine_type += "(s)"
    num_processes = _ask_field(
        f"How many {machine_type} should be used for distributed training? [1]:",
        int,
        default=1,
        error_message="Please enter an integer.",
    )
elif distributed_type in [DistributedType.FSDP, DistributedType.DEEPSPEED, DistributedType.MEGATRON_
    num_processes = _ask_field(
        "How many GPU(s) should be used for distributed training? [1]:",
        int,
        default=1,
        error_message="Please enter an integer.",
    )
else:
    num_processes = 1

if distributed_type in [DistributedType.MULTI_GPU, DistributedType.NO] and not use_cpu and not use_m
    gpu_ids = _ask_field(
        "What GPU(s) (by id) should be used for training on this machine as a comma-seperated list? 
        default="all",
    )

if distributed_type == DistributedType.TPU:
    mixed_precision = "no"
    main_training_function = _ask_field(
        "What is the name of the function in your script that should be launched in all parallel scr
        default="main",
    )
    tpu_use_cluster = _ask_field(
        "Are you using a TPU cluster? [yes/NO]: ",
        _convert_yes_no_to_bool,
        default=False,
        error_message="Please enter yes or no.",
    )
    if tpu_use_cluster:
        tpu_name = _ask_field(
            "What is the name of your TPU cluster? ",
            default=None,
            error_message="Please enter the name of your TPU cluster.",
        )
        tpu_zone = _ask_field(
            "What is the zone of your TPU cluster? ",
            default=None,
            error_message="Please enter the zone of your TPU cluster.",
        )
        tpu_use_sudo = _ask_field(
            "To run a python script in a TPU pod, should `sudo` be used? [yes/NO]: ",
            default=False,
            error_message="Please enter yes or no.",
        )
        run_commands = _ask_field(
            "Do you have code you wish to run on startup in each pod? [yes/NO]: ",
            _convert_yes_no_to_bool,
            default=False,
            error_message="Please enter yes or no.",
        )
        if run_commands:
            use_command_file = _ask_field(
                "Is this code located in a bash script? [yes/NO]: ",
                _convert_yes_no_to_bool,
                default=False,
                error_message="Please enter yes or no.",
            )
            if use_command_file:
                tpu_command_file = _ask_field(
                    "What is the path to your bash script? ",
                    default=None,
                    error_message="Please enter the path to your bash script.",
                )
                tpu_command_file = os.path.abspath(tpu_command_file)
            else:
                print("Please enter each command seperately you wish to run on startup in each pod."
                tpu_commands = []
                another_command = True
                while another_command:
                    tpu_commands.append(
                        _ask_field(
                            "Please enter a single command to be ran ",
                            default=None,
                            error_message="Please enter the commands you wish to run on startup in e
                        )
                    )
                    another_command = _ask_field(
                        "Do you wish to add another command? [yes/NO]: ",
                        _convert_yes_no_to_bool,
                        default=False,
                        error_message="Please enter yes or no.",
                    )
        tpu_vm = _ask_field(
            "If not using an instance group, what are the names of the Compute VM instances to be us
            default="",
        ).split(",")
        tpu_env = _ask_field(
            "What environment variables do you wish to set in each pod, seperated by a comma: ",
            default="",
        ).split(",")

else:
    main_training_function = "main"
    if distributed_type == DistributedType.DEEPSPEED and use_deepspeed_config:
        mixed_precision = None
    else:
        mixed_precision = _ask_options(
            "Do you wish to use FP16 or BF16 (mixed precision)?",
            ["no", "fp16", "bf16", "fp8"],
            _convert_mixed_precision,
        )

if use_dynamo and mixed_precision == "no" and not use_cpu:
    print(
        "Torch dynamo used without mixed precision requires TF32 to be efficient. Accelerate will en
    )

if distributed_type == DistributedType.TPU and mixed_precision == "bf16":
    tpu_downcast_bf16 = _ask_field(
        "Should `torch.float` be cast as `bfloat16` and `torch.double` remain `float32` on TPUs?", d
    )

return ClusterConfig(
    compute_environment=ComputeEnvironment.LOCAL_MACHINE,
    distributed_type=distributed_type,
    num_processes=num_processes,
    gpu_ids=gpu_ids,
    mixed_precision=mixed_precision,
    downcast_bf16=tpu_downcast_bf16,
    machine_rank=machine_rank,
    num_machines=num_machines,
    main_process_ip=main_process_ip,
    main_process_port=main_process_port,
    main_training_function=main_training_function,
    deepspeed_config=deepspeed_config,
    fsdp_config=fsdp_config,
    megatron_lm_config=megatron_lm_config,
    use_cpu=use_cpu,
    rdzv_backend=rdzv_backend,
    same_network=same_network,
    commands=tpu_commands,
    command_file=tpu_command_file,
    tpu_env=tpu_env,
    tpu_name=tpu_name,
    tpu_vm=tpu_vm,
    tpu_zone=tpu_zone,
    tpu_use_sudo=tpu_use_sudo,
    tpu_use_cluster=tpu_use_cluster,
    dynamo_config=dynamo_config,
)
```

accelerate-main/src/accelerate/commands/config/config_utils.py

```python
#!/usr/bin/env python

# Copyright 2021 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import argparse

from ...utils.dataclasses import (
    ComputeEnvironment,
    DistributedType,
    DynamoBackend,
    PrecisionType,
    SageMakerDistributedType,
)
from ..menu import BulletMenu


DYNAMO_BACKENDS = [
    "EAGER",
    "AOT_EAGER",
    "INDUCTOR",
    "NVFUSER",
    "AOT_NVFUSER",
    "AOT_CUDAGRAPHS",
    "OFI",
    "FX2TRT",
    "ONNXRT",
    "IPEX",
]


def _ask_field(input_text, convert_value=None, default=None, error_message=None):
    ask_again = True
    while ask_again:
        result = input(input_text)
        try:
            if default is not None and len(result) == 0:
                return default
            return convert_value(result) if convert_value is not None else result
        except Exception:
            if error_message is not None:
                print(error_message)


def _ask_options(input_text, options=[], convert_value=None, default=0):
    menu = BulletMenu(input_text, options)
    result = menu.run(default_choice=default)
    return convert_value(result) if convert_value is not None else result


def _convert_compute_environment(value):
    value = int(value)
    return ComputeEnvironment(["LOCAL_MACHINE", "AMAZON_SAGEMAKER"][value])


def _convert_distributed_mode(value):
    value = int(value)
    return DistributedType(["NO", "MULTI_CPU", "MULTI_GPU", "TPU"][value])


def _convert_dynamo_backend(value):
    value = int(value)
    return DynamoBackend(DYNAMO_BACKENDS[value]).value


def _convert_mixed_precision(value):
    value = int(value)
    return PrecisionType(["no", "fp16", "bf16", "fp8"][value])


def _convert_sagemaker_distributed_mode(value):
    value = int(value)
    return SageMakerDistributedType(["NO", "DATA_PARALLEL", "MODEL_PARALLEL"][value])


def _convert_yes_no_to_bool(value):
    return {"yes": True, "no": False}[value.lower()]


class SubcommandHelpFormatter(argparse.RawDescriptionHelpFormatter):
    """
    A custom formatter that will remove the usage line from the help message for subcommands.
    """

    def _format_usage(self, usage, actions, groups, prefix):
        usage = super()._format_usage(usage, actions, groups, prefix)
        usage = usage.replace("<command> [<args>] ", "")
        return usage
```

accelerate-main/utils/stale.py

```
# Copyright 2022 The HuggingFace Team, the AllenNLP library authors. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
"""
Script to close stale issue. Taken in part from the AllenNLP repository.
https://github.com/allenai/allennlp.
"""
from datetime import datetime as dt
import os

from github import Github


LABELS_TO_EXEMPT = [
    "good first issue",
    "feature request",
    "wip",
]


def main():
    g = Github(os.environ["GITHUB_TOKEN"])
    repo = g.get_repo("huggingface/accelerate")
    open_issues = repo.get_issues(state="open")

    for issue in open_issues:
        comments = sorted([comment for comment in issue.get_comments()], key=lambda i: i.created_at, rev
        last_comment = comments[0] if len(comments) > 0 else None
        current_time = dt.utcnow()
        days_since_updated = (current_time - issue.updated_at).days
        days_since_creation = (current_time - issue.created_at).days
        if (
            last_comment is not None and last_comment.user.login == "github-actions[bot]"
            and days_since_updated > 7
            and days_since_creation >= 30
            and not any(label.name.lower() in LABELS_TO_EXEMPT for label in issue.get_labels())
        ):
            # Close issue since it has been 7 days of inactivity since bot mention.
            issue.edit(state="closed")
        elif (
            days_since_updated > 23
            and days_since_creation >= 30
            and not any(label.name.lower() in LABELS_TO_EXEMPT for label in issue.get_labels())
        ):
            # Add stale comment
            issue.create_comment(
                "This issue has been automatically marked as stale because it has not had "
                "recent activity. If you think this still needs to be addressed "
                "please comment on this thread.\n\nPlease note that issues that do not follow the "
                "[contributing guidelines](https://github.com/huggingface/accelerate/blob/main/CONTRIBUT
                "are likely to be ignored."
            )


if __name__ == "__main__":
    main()
```

accelerate-main/utils/log_reports.py

```python
import json, os
from pathlib import Path
from datetime import date

failed = []
passed = []

group_info = []

total_num_failed = 0
for log in Path().glob("*.log"):
    section_num_failed = 0
    with open(log, "r") as f:
        for line in f:
            line = json.loads(line)
            if line.get("nodeid", "") != "":
                test = line["nodeid"]
                if line.get("duration", None) is not None:
                    duration = f'{line["duration"]:.4f}'
                    if line.get("outcome", "") == "failed":
                        section_num_failed += 1
                        failed.append([test, duration, log.name.split('_')[0]])
                        total_num_failed += 1
                    else:
                        passed.append([test, duration, log.name.split('_')[0]])
    group_info.append([str(log), section_num_failed, failed])
    failed = []
message = ""
if total_num_failed > 0:
    for name, num_failed, failed_tests in group_info:
        if num_failed > 0:
            if num_failed == 1:
                message += f"*{name}: {num_failed} failed test*\n"
            else:
                message += f"*{name}: {num_failed} failed tests*\n"
            failed_table = '| Test Location | Test Class | Test Name | PyTorch Version |\n|---|---|---|-
            for test in failed_tests:
                failed_table += ' | '.join(test[0].split("::"))
            failed_table += f" | {test[2]} |"
            message += failed_table
    print(f'### {message}')
else:
    message = "No failed tests! ■"
    print(f'## {message}')

if os.environ.get("TEST_TYPE", "") != "":
    from slack_sdk import WebClient
    message = f'*Nightly {os.environ.get("TEST_TYPE")} test results for {date.today()}:*\n{message}'

    client = WebClient(token=os.environ['SLACK_API_TOKEN'])
    client.chat_postMessage(channel='#accelerate-ci-daily', text=message)
```

accelerate-main/tests/test_accelerator.py

```python
import json
import os
import tempfile
import unittest
from unittest.mock import patch

import torch
from torch.utils.data import DataLoader, TensorDataset

from accelerate import infer_auto_device_map, init_empty_weights
from accelerate.accelerator import Accelerator
from accelerate.state import PartialState
from accelerate.test_utils import require_multi_gpu, slow
from accelerate.test_utils.testing import AccelerateTestCase, require_cuda
from accelerate.utils import patch_environment


def create_components():
    model = torch.nn.Linear(2, 4)
    optimizer = torch.optim.AdamW(model.parameters(), lr=1.0)
    scheduler = torch.optim.lr_scheduler.OneCycleLR(optimizer, max_lr=0.01, steps_per_epoch=2, epochs=1)
    train_dl = DataLoader(TensorDataset(torch.tensor([1, 2, 3])))
    valid_dl = DataLoader(TensorDataset(torch.tensor([4, 5, 6])))

    return model, optimizer, scheduler, train_dl, valid_dl


def get_signature(model):
    return (model.weight.abs().sum() + model.bias.abs().sum()).item()


def load_random_weights(model):
    state = torch.nn.Linear(*tuple(model.weight.T.shape)).state_dict()
    model.load_state_dict(state)


class AcceleratorTester(AccelerateTestCase):
    @require_cuda
    def test_accelerator_can_be_reinstantiated(self):
        _ = Accelerator()
        assert PartialState._shared_state["_cpu"] is False
        assert PartialState._shared_state["device"].type == "cuda"
        with self.assertRaises(ValueError):
            _ = Accelerator(cpu=True)

    def test_prepared_objects_are_referenced(self):
        accelerator = Accelerator()
        model, optimizer, scheduler, train_dl, valid_dl = create_components()

        (
            prepared_model,
            prepared_optimizer,
            prepared_scheduler,
            prepared_train_dl,
            prepared_valid_dl,
        ) = accelerator.prepare(model, optimizer, scheduler, train_dl, valid_dl)

        self.assertTrue(prepared_model in accelerator._models)
        self.assertTrue(prepared_optimizer in accelerator._optimizers)
        self.assertTrue(prepared_scheduler in accelerator._schedulers)
        self.assertTrue(prepared_train_dl in accelerator._dataloaders)
        self.assertTrue(prepared_valid_dl in accelerator._dataloaders)

    def test_free_memory_dereferences_prepared_components(self):
        accelerator = Accelerator()
        model, optimizer, scheduler, train_dl, valid_dl = create_components()
        accelerator.prepare(model, optimizer, scheduler, train_dl, valid_dl)
        accelerator.free_memory()
        self.assertTrue(len(accelerator._models) == 0)
        self.assertTrue(len(accelerator._optimizers) == 0)
        self.assertTrue(len(accelerator._schedulers) == 0)
        self.assertTrue(len(accelerator._dataloaders) == 0)

    def test_env_var_device(self):
        """Tests that setting the torch device with ACCELERATE_TORCH_DEVICE overrides default device."""
        PartialState._reset_state()

        # Mock torch.cuda.set_device to avoid an exception as the device doesn't exist
        def noop(*args, **kwargs):
            pass

        with patch("torch.cuda.set_device", noop), patch_environment(ACCELERATE_TORCH_DEVICE="cuda:64"):
            accelerator = Accelerator()
            self.assertEqual(str(accelerator.state.device), "cuda:64")

    def test_save_load_model(self):
        accelerator = Accelerator()
        model, optimizer, scheduler, train_dl, valid_dl = create_components()
        accelerator.prepare(model, optimizer, scheduler, train_dl, valid_dl)

        model_signature = get_signature(model)

        with tempfile.TemporaryDirectory() as tmpdirname:
            accelerator.save_state(tmpdirname)

            # make sure random weights don't match
            load_random_weights(model)
            self.assertTrue(abs(model_signature - get_signature(model)) > 1e-3)

            # make sure loaded weights match
            accelerator.load_state(tmpdirname)
            self.assertTrue(abs(model_signature - get_signature(model)) < 1e-3)

    def test_save_load_model_with_hooks(self):
        accelerator = Accelerator()
        model, optimizer, scheduler, train_dl, valid_dl = create_components()
        accelerator.prepare(model, optimizer, scheduler, train_dl, valid_dl)

        model_signature = get_signature(model)

        # saving hook
        def save_config(models, weights, output_dir):
            config = {"class_name": models[0].__class__.__name__}

            with open(os.path.join(output_dir, "data.json"), "w") as f:
                json.dump(config, f)

        # loading hook
        def load_config(models, input_dir):
            with open(os.path.join(input_dir, "data.json"), "r") as f:
                config = json.load(f)

            models[0].class_name = config["class_name"]

        save_hook = accelerator.register_save_state_pre_hook(save_config)
        load_hook = accelerator.register_load_state_pre_hook(load_config)

        with tempfile.TemporaryDirectory() as tmpdirname:
            accelerator.save_state(tmpdirname)

            # make sure random weights don't match with hooks
            load_random_weights(model)
            self.assertTrue(abs(model_signature - get_signature(model)) > 1e-3)

            # random class name to verify correct one is loaded
            model.class_name = "random"

            # make sure loaded weights match with hooks
            accelerator.load_state(tmpdirname)
            self.assertTrue(abs(model_signature - get_signature(model)) < 1e-3)
            # mode.class_name is loaded from config
            self.assertTrue(model.class_name == model.__class__.__name__)

        # remove hooks
        save_hook.remove()
        load_hook.remove()

        with tempfile.TemporaryDirectory() as tmpdirname:
            accelerator.save_state(tmpdirname)

            # make sure random weights don't match with hooks removed
            load_random_weights(model)
            self.assertTrue(abs(model_signature - get_signature(model)) > 1e-3)

            # random class name to verify correct one is loaded
            model.class_name = "random"

            # make sure loaded weights match with hooks removed
            accelerator.load_state(tmpdirname)
            self.assertTrue(abs(model_signature - get_signature(model)) < 1e-3)

            # mode.class_name is NOT loaded from config
            self.assertTrue(model.class_name != model.__class__.__name__)

    @slow
    def test_accelerator_bnb(self):
        """Tests that the accelerator can be used with the BNB library."""
        from transformers import AutoModelForCausalLM

        model = AutoModelForCausalLM.from_pretrained(
            "EleutherAI/gpt-neo-125m",
            load_in_8bit=True,
            device_map="auto",
        )
        accelerator = Accelerator()

        # This should work
        model = accelerator.prepare(model)

    @slow
    @unittest.skip("Skip until the next `transformers` release")
    def test_accelerator_bnb_cpu_error(self):
        """Tests that the accelerator can be used with the BNB library. This should fail as we are tryin
        that is loaded between cpu and gpu"""
        from transformers import AutoModelForCausalLM

        accelerator = Accelerator()

        with init_empty_weights():
            model = AutoModelForCausalLM.from_pretrained(
                "EleutherAI/gpt-neo-125m",
            )
            device_map = infer_auto_device_map(model)
            device_map["lm_head"] = "cpu"

        model = AutoModelForCausalLM.from_pretrained(
            "EleutherAI/gpt-neo-125m", device_map=device_map, load_in_8bit=True, llm_int8_enable_fp32_cp
        )

        # This should not work and get value error
        with self.assertRaises(ValueError):
            model = accelerator.prepare(model)

    @slow
    @require_multi_gpu
    def test_accelerator_bnb_multi_gpu(self):
        """Tests that the accelerator can be used with the BNB library."""
        from transformers import AutoModelForCausalLM

        with init_empty_weights():
            model = AutoModelForCausalLM.from_pretrained(
                "EleutherAI/gpt-neo-125m",
            )
            device_map = infer_auto_device_map(model)
            device_map["lm_head"] = 1

        model = AutoModelForCausalLM.from_pretrained(
            "EleutherAI/gpt-neo-125m",
            load_in_8bit=True,
            device_map=device_map,
        )
        accelerator = Accelerator()

        # This should not work and get value error
        with self.assertRaises(ValueError):
            _ = accelerator.prepare(model)

    @require_cuda
    def test_accelerator_cpu_flag_prepare(self):
        model = torch.nn.Linear(10, 10)
        sgd = torch.optim.SGD(model.parameters(), lr=0.01)
        accelerator = Accelerator(cpu=True)
        _ = accelerator.prepare(sgd)
```

accelerate-main/tests/test_memory_utils.py

```
# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import unittest

import torch
from torch import nn

from accelerate.test_utils import require_cuda
from accelerate.utils.memory import find_executable_batch_size, release_memory


def raise_fake_out_of_memory():
    raise RuntimeError("CUDA out of memory.")


class ModelForTest(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear1 = nn.Linear(3, 4)
        self.batchnorm = nn.BatchNorm1d(4)
        self.linear2 = nn.Linear(4, 5)

    def forward(self, x):
        return self.linear2(self.batchnorm(self.linear1(x)))


class MemoryTest(unittest.TestCase):
    def test_memory_implicit(self):
        batch_sizes = []

        @find_executable_batch_size(starting_batch_size=128)
        def mock_training_loop_function(batch_size):
            nonlocal batch_sizes
            batch_sizes.append(batch_size)
            if batch_size != 8:
                raise_fake_out_of_memory()

        mock_training_loop_function()
        self.assertListEqual(batch_sizes, [128, 64, 32, 16, 8])

    def test_memory_explicit(self):
        batch_sizes = []

        @find_executable_batch_size(starting_batch_size=128)
        def mock_training_loop_function(batch_size, arg1):
            nonlocal batch_sizes
            batch_sizes.append(batch_size)
            if batch_size != 8:
                raise_fake_out_of_memory()
            return batch_size, arg1

        bs, arg1 = mock_training_loop_function("hello")
        self.assertListEqual(batch_sizes, [128, 64, 32, 16, 8])
        self.assertListEqual([bs, arg1], [8, "hello"])

    def test_start_zero(self):
        @find_executable_batch_size(starting_batch_size=0)
        def mock_training_loop_function(batch_size):
            pass

        with self.assertRaises(RuntimeError) as cm:
            mock_training_loop_function()
            self.assertIn("No executable batch size found, reached zero.", cm.exception.args[0])

    def test_approach_zero(self):
        @find_executable_batch_size(starting_batch_size=16)
        def mock_training_loop_function(batch_size):
            if batch_size > 0:
                raise_fake_out_of_memory()
            pass

        with self.assertRaises(RuntimeError) as cm:
            mock_training_loop_function()
            self.assertIn("No executable batch size found, reached zero.", cm.exception.args[0])

    def test_verbose_guard(self):
        @find_executable_batch_size(starting_batch_size=128)
        def mock_training_loop_function(batch_size, arg1, arg2):
            if batch_size != 8:
                raise raise_fake_out_of_memory()

        with self.assertRaises(TypeError) as cm:
            mock_training_loop_function(128, "hello", "world")
            self.assertIn("Batch size was passed into `f`", cm.exception.args[0])
            self.assertIn("`f(arg1='hello', arg2='world')", cm.exception.args[0])

    def test_any_other_error(self):
        @find_executable_batch_size(starting_batch_size=16)
        def mock_training_loop_function(batch_size):
            raise ValueError("Oops, we had an error!")

        with self.assertRaises(ValueError) as cm:
            mock_training_loop_function()
            self.assertIn("Oops, we had an error!", cm.exception.args[0])

    @require_cuda
    def test_release_memory(self):
        self.assertEqual(torch.cuda.memory_allocated(), 0)
        model = ModelForTest()
        model.cuda()
        self.assertGreater(torch.cuda.memory_allocated(), 0)
        model = release_memory(model)
        self.assertEqual(torch.cuda.memory_allocated(), 0)
```

accelerate-main/tests/test_big_modeling.py

```
# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import os
import unittest
from tempfile import TemporaryDirectory

import torch
import torch.nn as nn
from transformers import AutoModelForCausalLM, AutoTokenizer

from accelerate.big_modeling import (
    cpu_offload,
    cpu_offload_with_hook,
    disk_offload,
    dispatch_model,
    init_empty_weights,
    init_on_device,
    load_checkpoint_and_dispatch,
)
from accelerate.hooks import remove_hook_from_submodules
from accelerate.test_utils import require_cuda, require_mps, require_multi_gpu, require_torch_min_versio
from accelerate.utils import offload_state_dict


class ModelForTest(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear1 = nn.Linear(3, 4)
        self.batchnorm = nn.BatchNorm1d(4)
        self.linear2 = nn.Linear(4, 5)

    def forward(self, x):
        return self.linear2(self.batchnorm(self.linear1(x)))


class ModelForTestTiedWeights(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear1 = nn.Linear(4, 4)
        self.batchnorm = nn.BatchNorm1d(4)
        self.linear2 = nn.Linear(4, 4)

    def forward(self, x):
        return self.linear2(self.batchnorm(self.linear1(x)))


class BiggerModelForTest(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear1 = nn.Linear(3, 4)
        self.linear2 = nn.Linear(4, 5)
        self.batchnorm = nn.BatchNorm1d(5)
        self.linear3 = nn.Linear(5, 6)
        self.linear4 = nn.Linear(6, 5)

    def forward(self, x):
        return self.linear4(self.linear3(self.batchnorm(self.linear2(self.linear1(x)))))


# To test preload_module_classes
class ModuleWithUnusedSubModules(nn.Module):
    def __init__(self, input_dim, output_dim):
        super().__init__()
        self.linear = nn.Linear(input_dim, output_dim)

    def forward(self, x):
        return x @ self.linear.weight.t() + self.linear.bias


class ModelWithUnusedSubModulesForTest(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear1 = ModuleWithUnusedSubModules(3, 4)
        self.linear2 = ModuleWithUnusedSubModules(4, 5)
        self.batchnorm = nn.BatchNorm1d(5)
        self.linear3 = ModuleWithUnusedSubModules(5, 6)
        self.linear4 = ModuleWithUnusedSubModules(6, 5)

    def forward(self, x):
        return self.linear4(self.linear3(self.batchnorm(self.linear2(self.linear1(x)))))


@require_torch_min_version(version="1.9.0")
class BigModelingTester(unittest.TestCase):
    def test_init_empty_weights(self):
        # base use
        with init_empty_weights():
            module = nn.Linear(4, 5)
        self.assertEqual(module.weight.device, torch.device("meta"))

        # base use with buffers, they are not touched
        with init_empty_weights():
            module = nn.BatchNorm1d(4)
        self.assertEqual(module.weight.device, torch.device("meta"))
        self.assertEqual(module.running_mean.device, torch.device("cpu"))

        # Use with include_buffers=True
        with init_empty_weights(include_buffers=True):
            module = nn.BatchNorm1d(4)
        self.assertEqual(module.weight.device, torch.device("meta"))
        self.assertEqual(module.running_mean.device, torch.device("meta"))

        # Double check we didn't break PyTorch
        module = nn.BatchNorm1d(4)
        self.assertEqual(module.weight.device, torch.device("cpu"))
        self.assertEqual(module.running_mean.device, torch.device("cpu"))

    def test_init_empty_weights_very_large_model(self):
        # This is a 100 billion parameters model.
        with init_empty_weights():
            _ = nn.Sequential(*[nn.Linear(10000, 10000) for _ in range(1000)])

    @require_cuda
    def test_init_on_device_cuda(self):
        device = torch.device("cuda:0")
        with init_on_device(device):
            model = nn.Linear(10, 10)
        self.assertEqual(model.weight.device, device)
        self.assertEqual(model.weight.device, device)

    @require_mps
    def test_init_on_device_mps(self):
        device = torch.device("mps:0")
        with init_on_device(device):
            model = nn.Linear(10, 10)
        self.assertEqual(model.weight.device, device)
        self.assertEqual(model.weight.device, device)
    def test_cpu_offload(self):
        model = ModelForTest()
        x = torch.randn(2, 3)
        expected = model(x)

        device = torch.device(0 if torch.cuda.is_available() else "cpu")

        cpu_offload(model, execution_device=device)
        output = model(x)
        self.assertTrue(
            torch.allclose(expected, output.cpu(), 1e-4, 1e-5), msg=f"Expected: {expected}\nActual: {out
        )

        # Clean up for next test.
        remove_hook_from_submodules(model)

        cpu_offload(model, execution_device=device, offload_buffers=True)
        output = model(x)
        self.assertTrue(
            torch.allclose(expected, output.cpu(), 1e-4, 1e-5), msg=f"Expected: {expected}\nActual: {out
        )

    def test_cpu_offload_with_unused_submodules(self):
        model = ModelWithUnusedSubModulesForTest()
        x = torch.randn(2, 3)
        expected = model(x)

        device = torch.device(0 if torch.cuda.is_available() else "cpu")

        cpu_offload(model, execution_device=device, preload_module_classes=["ModuleWithUnusedSubModules"
        output = model(x)
        self.assertTrue(
            torch.allclose(expected, output.cpu(), 1e-4, 1e-5), msg=f"Expected: {expected}\nActual: {out
        )

        # Clean up for next test.
        remove_hook_from_submodules(model)

        cpu_offload(
            model,
            execution_device=device,
            offload_buffers=True,
            preload_module_classes=["ModuleWithUnusedSubModules"],
        )
        output = model(x)
        self.assertTrue(
            torch.allclose(expected, output.cpu(), 1e-4, 1e-5), msg=f"Expected: {expected}\nActual: {out
        )

    @slow
    @require_cuda
    def test_cpu_offload_gpt2(self):
        tokenizer = AutoTokenizer.from_pretrained("gpt2")
        inputs = tokenizer("Hello world! My name is", return_tensors="pt").to(0)

        gpt2 = AutoModelForCausalLM.from_pretrained("gpt2")
        cpu_offload(gpt2, execution_device=0)
        outputs = gpt2.generate(inputs["input_ids"])
        self.assertEqual(
            tokenizer.decode(outputs[0].tolist()),
            "Hello world! My name is Kiyoshi, and I'm a student at the University of Tokyo",
        )

    def test_disk_offload(self):
        model = ModelForTest()
        x = torch.randn(2, 3)
        expected = model(x)

        device = torch.device(0 if torch.cuda.is_available() else "cpu")

        with TemporaryDirectory() as tmp_dir:
            disk_offload(model, tmp_dir, execution_device=device)
            output = model(x)
            self.assertTrue(
                torch.allclose(expected, output.cpu(), 1e-4, 1e-5), msg=f"Expected: {expected}\nActual:
            )

            # Clean up for next test.
            remove_hook_from_submodules(model)

        with TemporaryDirectory() as tmp_dir:
            disk_offload(model, tmp_dir, execution_device=device, offload_buffers=True)
            output = model(x)
            self.assertTrue(
                torch.allclose(expected, output.cpu(), 1e-4, 1e-5), msg=f"Expected: {expected}\nActual:
            )

    def test_disk_offload_with_unused_submodules(self):
        model = ModelWithUnusedSubModulesForTest()
        x = torch.randn(2, 3)
        expected = model(x)

        device = torch.device(0 if torch.cuda.is_available() else "cpu")

        with TemporaryDirectory() as tmp_dir:
            disk_offload(
                model, tmp_dir, execution_device=device, preload_module_classes=["ModuleWithUnusedSubMod
            )
            output = model(x)
            self.assertTrue(
                torch.allclose(expected, output.cpu(), 1e-4, 1e-5), msg=f"Expected: {expected}\nActual:
            )

            # Clean up for next test.
            remove_hook_from_submodules(model)

        with TemporaryDirectory() as tmp_dir:
            disk_offload(
                model,
                tmp_dir,
                execution_device=device,
                offload_buffers=True,
                preload_module_classes=["ModuleWithUnusedSubModules"],
            )
            output = model(x)
            self.assertTrue(
                torch.allclose(expected, output.cpu(), 1e-4, 1e-5), msg=f"Expected: {expected}\nActual:
            )

    @slow
    @require_cuda
    def test_disk_offload_gpt2(self):
        tokenizer = AutoTokenizer.from_pretrained("gpt2")
        inputs = tokenizer("Hello world! My name is", return_tensors="pt").to(0)

        gpt2 = AutoModelForCausalLM.from_pretrained("gpt2")
        with TemporaryDirectory() as tmp_dir:
            disk_offload(gpt2, tmp_dir, execution_device=0)
            outputs = gpt2.generate(inputs["input_ids"])
            self.assertEqual(
                tokenizer.decode(outputs[0].tolist()),
                "Hello world! My name is Kiyoshi, and I'm a student at the University of Tokyo",
            )

    @require_cuda
    def test_dispatch_model(self):
        model = ModelForTest()
        device_map = {"linear1": "disk", "batchnorm": "cpu", "linear2": 0}

        x = torch.randn(2, 3)
        expected = model(x)

        with TemporaryDirectory() as tmp_dir:
            dispatch_model(model, device_map, offload_dir=tmp_dir)
            output = model(x)
            self.assertTrue(torch.allclose(expected, output.cpu(), atol=1e-5))

    @require_cuda
    def test_dispatch_model_tied_weights(self):
        model = ModelForTestTiedWeights()
        model.linear1.weight = model.linear2.weight
        device_map = {"linear1": 0, "batchnorm": 0, "linear2": 0}

        dispatch_model(model, device_map)
        self.assertIs(model.linear2.weight, model.linear1.weight)

    @require_multi_gpu
    def test_dispatch_model_multi_gpu(self):
        model = BiggerModelForTest()
        device_map = {"linear1": "cpu", "linear2": "disk", "batchnorm": "cpu", "linear3": 0, "linear4":

        x = torch.randn(2, 3)
        expected = model(x)

        with TemporaryDirectory() as tmp_dir:
            dispatch_model(model, device_map, offload_dir=tmp_dir)
            output = model(x)
            self.assertTrue(torch.allclose(expected, output.cpu(), atol=1e-5))

    @slow
    @require_multi_gpu
    def test_dispatch_model_gpt2_on_two_gpus(self):
        tokenizer = AutoTokenizer.from_pretrained("gpt2")
        inputs = tokenizer("Hello world! My name is", return_tensors="pt").to(0)

        gpt2 = AutoModelForCausalLM.from_pretrained("gpt2")
        # Dispatch on GPUs 0 and 1
        device_map = {
            "transformer.wte": 0,
            "transformer.wpe": 0,
            "transformer.ln_f": 1,
            "lm_head": 0,
        }
        for i in range(12):
            device_map[f"transformer.h.{i}"] = 0 if i <= 5 else 1

        gpt2 = dispatch_model(gpt2, device_map)
        outputs = gpt2.generate(inputs["input_ids"])
        self.assertEqual(
            tokenizer.decode(outputs[0].tolist()),
            "Hello world! My name is Kiyoshi, and I'm a student at the University of Tokyo",
        )

        # Dispatch with a bit of CPU offload
        gpt2 = AutoModelForCausalLM.from_pretrained("gpt2")
        for i in range(4):
            device_map[f"transformer.h.{i}"] = "cpu"
        gpt2 = dispatch_model(gpt2, device_map)
        outputs = gpt2.generate(inputs["input_ids"])
        self.assertEqual(
            tokenizer.decode(outputs[0].tolist()),
            "Hello world! My name is Kiyoshi, and I'm a student at the University of Tokyo",
        )
        # Dispatch with a bit of CPU and disk offload
        gpt2 = AutoModelForCausalLM.from_pretrained("gpt2")
        for i in range(2):
            device_map[f"transformer.h.{i}"] = "disk"

        with TemporaryDirectory() as tmp_dir:
            state_dict = {
                k: p for k, p in gpt2.state_dict().items() if "transformer.h.0" in k or "transformer.h.1
            }
            offload_state_dict(tmp_dir, state_dict)
            gpt2 = dispatch_model(gpt2, device_map, offload_dir=tmp_dir)
            outputs = gpt2.generate(inputs["input_ids"])
            self.assertEqual(
                tokenizer.decode(outputs[0].tolist()),
                "Hello world! My name is Kiyoshi, and I'm a student at the University of Tokyo",
            )

    @require_cuda
    def test_dispatch_model_with_unused_submodules(self):
        model = ModelWithUnusedSubModulesForTest()
        device_map = {"linear1": "cpu", "linear2": "disk", "batchnorm": "cpu", "linear3": 0, "linear4":

        x = torch.randn(2, 3)
        expected = model(x)

        with TemporaryDirectory() as tmp_dir:
            dispatch_model(
                model, device_map, offload_dir=tmp_dir, preload_module_classes=["ModuleWithUnusedSubModu
            )
            output = model(x)
            self.assertTrue(torch.allclose(expected, output.cpu(), atol=1e-5))

    @require_multi_gpu
    def test_dispatch_model_with_unused_submodules_multi_gpu(self):
        model = ModelWithUnusedSubModulesForTest()
        device_map = {"linear1": "cpu", "linear2": "disk", "batchnorm": "cpu", "linear3": 0, "linear4":

        x = torch.randn(2, 3)
        expected = model(x)

        with TemporaryDirectory() as tmp_dir:
            dispatch_model(
                model, device_map, offload_dir=tmp_dir, preload_module_classes=["ModuleWithUnusedSubModu
            )
            output = model(x)
            self.assertTrue(torch.allclose(expected, output.cpu(), atol=1e-5))

    @require_cuda
    def test_load_checkpoint_and_dispatch(self):
        model = ModelForTest()
        device_map = {"linear1": "cpu", "batchnorm": "cpu", "linear2": 0}

        x = torch.randn(2, 3)
        expected = model(x)

        with TemporaryDirectory() as tmp_dir:
            checkpoint = os.path.join(tmp_dir, "pt_model.bin")
            torch.save(model.state_dict(), checkpoint)

            new_model = ModelForTest()
            new_model = load_checkpoint_and_dispatch(new_model, checkpoint, device_map=device_map)

        # CPU-offloaded weights are on the meta device while waiting for the forward pass.
        self.assertEqual(new_model.linear1.weight.device, torch.device("meta"))
        self.assertEqual(new_model.linear2.weight.device, torch.device(0))

        output = new_model(x)
        self.assertTrue(torch.allclose(expected, output.cpu(), atol=1e-5))

    @require_multi_gpu
    def test_load_checkpoint_and_dispatch_multi_gpu(self):
        model = BiggerModelForTest()
        device_map = {"linear1": "cpu", "linear2": "cpu", "batchnorm": 0, "linear3": 0, "linear4": 1}

        x = torch.randn(2, 3)
        expected = model(x)

        with TemporaryDirectory() as tmp_dir:
            checkpoint = os.path.join(tmp_dir, "pt_model.bin")
            torch.save(model.state_dict(), checkpoint)

            new_model = BiggerModelForTest()
            new_model = load_checkpoint_and_dispatch(new_model, checkpoint, device_map=device_map)

        # CPU-offloaded weights are on the meta device while waiting for the forward pass.
        self.assertEqual(new_model.linear1.weight.device, torch.device("meta"))
        self.assertEqual(new_model.linear2.weight.device, torch.device("meta"))
        self.assertEqual(new_model.linear3.weight.device, torch.device(0))
        self.assertEqual(new_model.linear4.weight.device, torch.device(1))

        output = new_model(x)
        self.assertTrue(torch.allclose(expected, output.cpu(), atol=1e-5))

    @require_cuda
    def test_load_checkpoint_and_dispatch_with_unused_submodules(self):
        model = ModelWithUnusedSubModulesForTest()
        device_map = {"linear1": "cpu", "linear2": "cpu", "batchnorm": 0, "linear3": 0, "linear4": 0}

        x = torch.randn(2, 3)
        expected = model(x)

        with TemporaryDirectory() as tmp_dir:
            checkpoint = os.path.join(tmp_dir, "pt_model.bin")
            torch.save(model.state_dict(), checkpoint)

            new_model = ModelWithUnusedSubModulesForTest()
            new_model = load_checkpoint_and_dispatch(
                new_model, checkpoint, device_map=device_map, preload_module_classes=["ModuleWithUnusedS
            )

        # CPU-offloaded weights are on the meta device while waiting for the forward pass.
        self.assertEqual(new_model.linear1.linear.weight.device, torch.device("meta"))
        self.assertEqual(new_model.linear2.linear.weight.device, torch.device("meta"))
        self.assertEqual(new_model.linear3.linear.weight.device, torch.device(0))
        self.assertEqual(new_model.linear4.linear.weight.device, torch.device(0))

        output = new_model(x)
        self.assertTrue(torch.allclose(expected, output.cpu(), atol=1e-5))

    @require_multi_gpu
    def test_load_checkpoint_and_dispatch_multi_gpu_with_unused_submodules(self):
        model = ModelWithUnusedSubModulesForTest()
        device_map = {"linear1": "cpu", "linear2": "cpu", "batchnorm": 0, "linear3": 0, "linear4": 1}

        x = torch.randn(2, 3)
        expected = model(x)

        with TemporaryDirectory() as tmp_dir:
            checkpoint = os.path.join(tmp_dir, "pt_model.bin")
            torch.save(model.state_dict(), checkpoint)

            new_model = ModelWithUnusedSubModulesForTest()
            new_model = load_checkpoint_and_dispatch(
                new_model, checkpoint, device_map=device_map, preload_module_classes=["ModuleWithUnusedS
            )

        # CPU-offloaded weights are on the meta device while waiting for the forward pass.
        self.assertEqual(new_model.linear1.linear.weight.device, torch.device("meta"))
        self.assertEqual(new_model.linear2.linear.weight.device, torch.device("meta"))
        self.assertEqual(new_model.linear3.linear.weight.device, torch.device(0))
        self.assertEqual(new_model.linear4.linear.weight.device, torch.device(1))

        output = new_model(x)
        self.assertTrue(torch.allclose(expected, output.cpu(), atol=1e-5))

    @require_cuda
    def test_cpu_offload_with_hook(self):
        model1 = torch.nn.Linear(4, 5)
        model1, hook1 = cpu_offload_with_hook(model1)
        self.assertEqual(model1.weight.device, torch.device("cpu"))

        inputs = torch.randn(3, 4)
        outputs = model1(inputs)
        self.assertEqual(outputs.device, torch.device(0))
        self.assertEqual(model1.weight.device, torch.device(0))

        hook1.offload()
        self.assertEqual(model1.weight.device, torch.device("cpu"))
```

```
model2 = torch.nn.Linear(5, 5)
model2, hook2 = cpu_offload_with_hook(model2, prev_module_hook=hook1)
self.assertEqual(model2.weight.device, torch.device("cpu"))

outputs = model1(inputs)
self.assertEqual(outputs.device, torch.device(0))
self.assertEqual(model1.weight.device, torch.device(0))

outputs = model2(outputs)
self.assertEqual(outputs.device, torch.device(0))
self.assertEqual(model1.weight.device, torch.device("cpu"))
self.assertEqual(model2.weight.device, torch.device(0))

hook2.offload()
self.assertEqual(model2.weight.device, torch.device("cpu"))
```

accelerate-main/tests/test_multigpu.py

```
# Copyright 2021 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import inspect
import os
import unittest

import torch

import accelerate
from accelerate import Accelerator
from accelerate.test_utils import execute_subprocess_async, require_multi_gpu
from accelerate.utils import get_launch_prefix, patch_environment


class MultiGPUTester(unittest.TestCase):
    def setUp(self):
        mod_file = inspect.getfile(accelerate.test_utils)
        self.test_file_path = os.path.sep.join(mod_file.split(os.path.sep)[:-1] + ["scripts", "test_scri
        self.data_loop_file_path = os.path.sep.join(
            mod_file.split(os.path.sep)[:-1] + ["scripts", "test_distributed_data_loop.py"]
        )
        self.operation_file_path = os.path.sep.join(mod_file.split(os.path.sep)[:-1] + ["scripts", "test

    @require_multi_gpu
    def test_multi_gpu(self):
        print(f"Found {torch.cuda.device_count()} devices.")
        cmd = get_launch_prefix() + [self.test_file_path]
        with patch_environment(omp_num_threads=1):
            execute_subprocess_async(cmd, env=os.environ.copy())

    @require_multi_gpu
    def test_multi_gpu_ops(self):
        print(f"Found {torch.cuda.device_count()} devices.")
        cmd = get_launch_prefix() + [self.operation_file_path]
        with patch_environment(omp_num_threads=1):
            execute_subprocess_async(cmd, env=os.environ.copy())

    @require_multi_gpu
    def test_pad_across_processes(self):
        cmd = get_launch_prefix() + [inspect.getfile(self.__class__)]
        with patch_environment(omp_num_threads=1):
            execute_subprocess_async(cmd, env=os.environ.copy())

    @require_multi_gpu
    def test_distributed_data_loop(self):
        """
        This TestCase checks the behaviour that occurs during distributed training or evaluation,
        when the batch size does not evenly divide the dataset size.
        """
        print(f"Found {torch.cuda.device_count()} devices, using 2 devices only")
        cmd = get_launch_prefix() + [f"--nproc_per_node={torch.cuda.device_count()}", self.data_loop_fil
        with patch_environment(omp_num_threads=1, cuda_visible_devices="0,1"):
            execute_subprocess_async(cmd, env=os.environ.copy())


if __name__ == "__main__":
```

```
accelerator = Accelerator()
shape = (accelerator.state.process_index + 2, 10)
tensor = torch.randint(0, 10, shape).to(accelerator.device)

error_msg = ""

tensor1 = accelerator.pad_across_processes(tensor)
if tensor1.shape[0] != accelerator.state.num_processes + 1:
    error_msg += f"Found shape {tensor1.shape} but should have {accelerator.state.num_processes + 1}
if not torch.equal(tensor1[: accelerator.state.process_index + 2], tensor):
    error_msg += "Tensors have different values."
if not torch.all(tensor1[accelerator.state.process_index + 2 :] == 0):
    error_msg += "Padding was not done with the right value (0)."

tensor2 = accelerator.pad_across_processes(tensor, pad_first=True)
if tensor2.shape[0] != accelerator.state.num_processes + 1:
    error_msg += f"Found shape {tensor2.shape} but should have {accelerator.state.num_processes + 1}
index = accelerator.state.num_processes - accelerator.state.process_index - 1
if not torch.equal(tensor2[index:], tensor):
    error_msg += "Tensors have different values."
if not torch.all(tensor2[:index] == 0):
    error_msg += "Padding was not done with the right value (0)."

# Raise error at the end to make sure we don't stop at the first failure.
if len(error_msg) > 0:
    raise ValueError(error_msg)
```

accelerate-main/tests/test_tpu.py

```
# Copyright 2021 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import inspect
import os
import sys
import unittest

import accelerate
from accelerate.test_utils import execute_subprocess_async, require_tpu


class MultiTPUTester(unittest.TestCase):
    def setUp(self):
        mod_file = inspect.getfile(accelerate.test_utils)
        self.test_file_path = os.path.sep.join(mod_file.split(os.path.sep)[:-1] + ["scripts", "test_scri
        self.test_dir = os.path.sep.join(inspect.getfile(self.__class__).split(os.path.sep)[:-1])

    @require_tpu
    def test_tpu(self):
        distributed_args = f"""
            {self.test_dir}/xla_spawn.py
            --num_cores 8
            {self.test_file_path}
        """.split()
        cmd = [sys.executable] + distributed_args
        execute_subprocess_async(cmd, env=os.environ.copy())
```

accelerate-main/tests/test_data_loader.py

```
# Copyright 2021 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import random
import unittest

from torch.utils.data import BatchSampler, DataLoader, IterableDataset

from accelerate.data_loader import (
    BatchSamplerShard,
    IterableDatasetShard,
    SkipBatchSampler,
    SkipDataLoader,
    skip_first_batches,
)


class RandomIterableDataset(IterableDataset):
    # For testing, an iterable dataset of random length
    def __init__(self, p_stop=0.01, max_length=1000):
        self.p_stop = p_stop
        self.max_length = max_length

    def __iter__(self):
        count = 0
        stop = False
        while not stop and count < self.max_length:
            yield count
            count += 1
            stop = random.random() < self.p_stop


class DataLoaderTester(unittest.TestCase):
    def check_batch_sampler_shards(self, batch_sampler, expected, split_batches=False, even_batches=True
        batch_sampler_shards = [
            BatchSamplerShard(batch_sampler, 2, i, split_batches=split_batches, even_batches=even_batche
            for i in range(2)
        ]
        batch_sampler_lists = [list(batch_sampler_shard) for batch_sampler_shard in batch_sampler_shards
        if not split_batches:
            self.assertListEqual([len(shard) for shard in batch_sampler_shards], [len(e) for e in expect
        self.assertListEqual(batch_sampler_lists, expected)

    def test_batch_sampler_shards_with_no_splits(self):
        # Check the shards when the dataset is a round multiple of total batch size.
        batch_sampler = BatchSampler(range(24), batch_size=3, drop_last=False)
        expected = [
            [[0, 1, 2], [6, 7, 8], [12, 13, 14], [18, 19, 20]],
            [[3, 4, 5], [9, 10, 11], [15, 16, 17], [21, 22, 23]],
        ]
        self.check_batch_sampler_shards(batch_sampler, expected)

        batch_sampler = BatchSampler(range(24), batch_size=3, drop_last=True)
        # Expected shouldn't change
        self.check_batch_sampler_shards(batch_sampler, expected)

        # Check the shards when the dataset is a round multiple of batch size but not total batch size.
        batch_sampler = BatchSampler(range(21), batch_size=3, drop_last=False)
        expected = [
            [[0, 1, 2], [6, 7, 8], [12, 13, 14], [18, 19, 20]],
            [[3, 4, 5], [9, 10, 11], [15, 16, 17], [0, 1, 2]],
        ]
        self.check_batch_sampler_shards(batch_sampler, expected)

        batch_sampler = BatchSampler(range(21), batch_size=3, drop_last=True)
        expected = [
            [[0, 1, 2], [6, 7, 8], [12, 13, 14]],
            [[3, 4, 5], [9, 10, 11], [15, 16, 17]],
        ]
        self.check_batch_sampler_shards(batch_sampler, expected)

        # Check the shards when the dataset is not a round multiple of batch size but has a multiple of
        # num_processes batch.
        batch_sampler = BatchSampler(range(22), batch_size=3, drop_last=False)
        expected = [
            [[0, 1, 2], [6, 7, 8], [12, 13, 14], [18, 19, 20]],
            [[3, 4, 5], [9, 10, 11], [15, 16, 17], [21, 0, 1]],
        ]
        self.check_batch_sampler_shards(batch_sampler, expected)

        batch_sampler = BatchSampler(range(22), batch_size=3, drop_last=True)
        expected = [
            [[0, 1, 2], [6, 7, 8], [12, 13, 14]],
            [[3, 4, 5], [9, 10, 11], [15, 16, 17]],
        ]
        self.check_batch_sampler_shards(batch_sampler, expected)

        # Check the shards when the dataset is not a round multiple of batch size but and has not a mult
        # num_processes batch.
        batch_sampler = BatchSampler(range(20), batch_size=3, drop_last=False)
        expected = [
            [[0, 1, 2], [6, 7, 8], [12, 13, 14], [18, 19, 0]],
            [[3, 4, 5], [9, 10, 11], [15, 16, 17], [1, 2, 3]],
        ]
        self.check_batch_sampler_shards(batch_sampler, expected)

        batch_sampler = BatchSampler(range(20), batch_size=3, drop_last=True)
        expected = [
            [[0, 1, 2], [6, 7, 8], [12, 13, 14]],
            [[3, 4, 5], [9, 10, 11], [15, 16, 17]],
        ]
        self.check_batch_sampler_shards(batch_sampler, expected)

        # Check the shards when the dataset is very small.
        batch_sampler = BatchSampler(range(2), batch_size=3, drop_last=False)
        expected = [[[0, 1, 0]], [[1, 0, 1]]]
        self.check_batch_sampler_shards(batch_sampler, expected)

        batch_sampler = BatchSampler(range(2), batch_size=3, drop_last=True)
        expected = [[], []]
        self.check_batch_sampler_shards(batch_sampler, expected)

    def test_batch_sampler_shards_with_splits(self):
        # Check the shards when the dataset is a round multiple of batch size.
        batch_sampler = BatchSampler(range(24), batch_size=4, drop_last=False)
        expected = [
            [[0, 1], [4, 5], [8, 9], [12, 13], [16, 17], [20, 21]],
            [[2, 3], [6, 7], [10, 11], [14, 15], [18, 19], [22, 23]],
        ]
        self.check_batch_sampler_shards(batch_sampler, expected, split_batches=True)

        batch_sampler = BatchSampler(range(24), batch_size=4, drop_last=True)
        # Expected shouldn't change
        self.check_batch_sampler_shards(batch_sampler, expected, split_batches=True)

        # Check the shards when the dataset is not a round multiple of batch size.
        batch_sampler = BatchSampler(range(22), batch_size=4, drop_last=False)
        expected = [
            [[0, 1], [4, 5], [8, 9], [12, 13], [16, 17], [20, 21]],
            [[2, 3], [6, 7], [10, 11], [14, 15], [18, 19], [0, 1]],
        ]
        self.check_batch_sampler_shards(batch_sampler, expected, split_batches=True)

        batch_sampler = BatchSampler(range(22), batch_size=4, drop_last=True)
        expected = [
            [[0, 1], [4, 5], [8, 9], [12, 13], [16, 17]],
            [[2, 3], [6, 7], [10, 11], [14, 15], [18, 19]],
        ]
        self.check_batch_sampler_shards(batch_sampler, expected, split_batches=True)

        # Check the shards when the dataset is not a round multiple of batch size or num_processes.
        batch_sampler = BatchSampler(range(21), batch_size=4, drop_last=False)
        expected = [
            [[0, 1], [4, 5], [8, 9], [12, 13], [16, 17], [20, 0]],
            [[2, 3], [6, 7], [10, 11], [14, 15], [18, 19], [1, 2]],
        ]
        self.check_batch_sampler_shards(batch_sampler, expected, split_batches=True)

        batch_sampler = BatchSampler(range(21), batch_size=4, drop_last=True)
        expected = [
            [[0, 1], [4, 5], [8, 9], [12, 13], [16, 17]],
            [[2, 3], [6, 7], [10, 11], [14, 15], [18, 19]],
        ]
        self.check_batch_sampler_shards(batch_sampler, expected, split_batches=True)

        # Check the shards when the dataset is very small.
        batch_sampler = BatchSampler(range(2), batch_size=4, drop_last=False)
        expected = [[[0, 1]], [[0, 1]]]
        self.check_batch_sampler_shards(batch_sampler, expected, split_batches=True)

        batch_sampler = BatchSampler(range(2), batch_size=4, drop_last=True)
        expected = [[], []]
        self.check_batch_sampler_shards(batch_sampler, expected, split_batches=True)

    def test_batch_sampler_shards_with_no_splits_no_even(self):
        # Check the shards when the dataset is a round multiple of total batch size.
        batch_sampler = BatchSampler(range(24), batch_size=3, drop_last=False)
        expected = [
            [[0, 1, 2], [6, 7, 8], [12, 13, 14], [18, 19, 20]],
            [[3, 4, 5], [9, 10, 11], [15, 16, 17], [21, 22, 23]],
        ]
        self.check_batch_sampler_shards(batch_sampler, expected, even_batches=False)

        batch_sampler = BatchSampler(range(24), batch_size=3, drop_last=True)
        # Expected shouldn't change
        self.check_batch_sampler_shards(batch_sampler, expected, even_batches=False)

        # Check the shards when the dataset is a round multiple of batch size but not total batch size.
        batch_sampler = BatchSampler(range(21), batch_size=3, drop_last=False)
        expected = [
            [[0, 1, 2], [6, 7, 8], [12, 13, 14], [18, 19, 20]],
            [[3, 4, 5], [9, 10, 11], [15, 16, 17]],
        ]
        self.check_batch_sampler_shards(batch_sampler, expected, even_batches=False)

        batch_sampler = BatchSampler(range(21), batch_size=3, drop_last=True)
        expected = [
            [[0, 1, 2], [6, 7, 8], [12, 13, 14]],
            [[3, 4, 5], [9, 10, 11], [15, 16, 17]],
        ]
        self.check_batch_sampler_shards(batch_sampler, expected, even_batches=False)

        # Check the shards when the dataset is not a round multiple of batch size but has a multiple of
        # num_processes batch.
        batch_sampler = BatchSampler(range(22), batch_size=3, drop_last=False)
        expected = [
            [[0, 1, 2], [6, 7, 8], [12, 13, 14], [18, 19, 20]],
            [[3, 4, 5], [9, 10, 11], [15, 16, 17], [21]],
        ]
        self.check_batch_sampler_shards(batch_sampler, expected, even_batches=False)
        batch_sampler = BatchSampler(range(22), batch_size=3, drop_last=True)
        expected = [
            [[0, 1, 2], [6, 7, 8], [12, 13, 14]],
            [[3, 4, 5], [9, 10, 11], [15, 16, 17]],
        ]
        self.check_batch_sampler_shards(batch_sampler, expected, even_batches=False)

        # Check the shards when the dataset is not a round multiple of batch size but and has not a mult
        # num_processes batch.
        batch_sampler = BatchSampler(range(20), batch_size=3, drop_last=False)
        expected = [
            [[0, 1, 2], [6, 7, 8], [12, 13, 14], [18, 19]],
            [[3, 4, 5], [9, 10, 11], [15, 16, 17]],
        ]
        self.check_batch_sampler_shards(batch_sampler, expected, even_batches=False)

        batch_sampler = BatchSampler(range(20), batch_size=3, drop_last=True)
        expected = [
            [[0, 1, 2], [6, 7, 8], [12, 13, 14]],
            [[3, 4, 5], [9, 10, 11], [15, 16, 17]],
        ]
        self.check_batch_sampler_shards(batch_sampler, expected, even_batches=False)

        # Check the shards when the dataset is very small.
        batch_sampler = BatchSampler(range(2), batch_size=3, drop_last=False)
        expected = [[[0, 1]], []]
        self.check_batch_sampler_shards(batch_sampler, expected, even_batches=False)

        batch_sampler = BatchSampler(range(2), batch_size=3, drop_last=True)
        expected = [[], []]
        self.check_batch_sampler_shards(batch_sampler, expected, even_batches=False)

    def test_batch_sampler_shards_with_splits_no_even(self):
        # Check the shards when the dataset is a round multiple of batch size.
        batch_sampler = BatchSampler(range(24), batch_size=4, drop_last=False)
        expected = [
            [[0, 1], [4, 5], [8, 9], [12, 13], [16, 17], [20, 21]],
            [[2, 3], [6, 7], [10, 11], [14, 15], [18, 19], [22, 23]],
        ]
        self.check_batch_sampler_shards(batch_sampler, expected, split_batches=True, even_batches=False)

        batch_sampler = BatchSampler(range(24), batch_size=4, drop_last=True)
        # Expected shouldn't change
        self.check_batch_sampler_shards(batch_sampler, expected, split_batches=True, even_batches=False)

        # Check the shards when the dataset is not a round multiple of batch size.
        batch_sampler = BatchSampler(range(22), batch_size=4, drop_last=False)
        expected = [
            [[0, 1], [4, 5], [8, 9], [12, 13], [16, 17], [20, 21]],
            [[2, 3], [6, 7], [10, 11], [14, 15], [18, 19]],
        ]
        self.check_batch_sampler_shards(batch_sampler, expected, split_batches=True, even_batches=False)

        batch_sampler = BatchSampler(range(22), batch_size=4, drop_last=True)
        expected = [
            [[0, 1], [4, 5], [8, 9], [12, 13], [16, 17]],
            [[2, 3], [6, 7], [10, 11], [14, 15], [18, 19]],
        ]
        self.check_batch_sampler_shards(batch_sampler, expected, split_batches=True, even_batches=False)

        # Check the shards when the dataset is not a round multiple of batch size or num_processes.
        batch_sampler = BatchSampler(range(21), batch_size=4, drop_last=False)
        expected = [
            [[0, 1], [4, 5], [8, 9], [12, 13], [16, 17], [20]],
            [[2, 3], [6, 7], [10, 11], [14, 15], [18, 19]],
        ]
        self.check_batch_sampler_shards(batch_sampler, expected, split_batches=True, even_batches=False)

        batch_sampler = BatchSampler(range(21), batch_size=4, drop_last=True)
        expected = [
            [[0, 1], [4, 5], [8, 9], [12, 13], [16, 17]],
            [[2, 3], [6, 7], [10, 11], [14, 15], [18, 19]],
        ]
        self.check_batch_sampler_shards(batch_sampler, expected, split_batches=True, even_batches=False)

        # Check the shards when the dataset is very small.
        batch_sampler = BatchSampler(range(2), batch_size=4, drop_last=False)
        expected = [[[0, 1]], []]
        self.check_batch_sampler_shards(batch_sampler, expected, split_batches=True, even_batches=False)

        batch_sampler = BatchSampler(range(2), batch_size=4, drop_last=True)
        expected = [[], []]
        self.check_batch_sampler_shards(batch_sampler, expected, split_batches=True, even_batches=False)

    def test_batch_sampler_with_varying_batch_size(self):
        batch_sampler = [[0, 1, 2], [3, 4], [5, 6, 7, 8], [9, 10, 11], [12, 13]]
        batch_sampler_shards = [BatchSamplerShard(batch_sampler, 2, i, even_batches=False) for i in rang

        self.assertEqual(len(batch_sampler_shards[0]), 3)
        self.assertEqual(len(batch_sampler_shards[1]), 2)

        self.assertListEqual(list(batch_sampler_shards[0]), [[0, 1, 2], [5, 6, 7, 8], [12, 13]])
        self.assertListEqual(list(batch_sampler_shards[1]), [[3, 4], [9, 10, 11]])

    def check_iterable_dataset_shards(
        self, dataset, seed, batch_size, drop_last=False, num_processes=2, split_batches=False
    ):
        random.seed(seed)
        reference = list(dataset)

        iterable_dataset_shards = [
            IterableDatasetShard(
                dataset,
                batch_size=batch_size,
                drop_last=drop_last,
                num_processes=num_processes,
                process_index=i,
                split_batches=split_batches,
            )
            for i in range(num_processes)
        ]
        iterable_dataset_lists = []
        for iterable_dataset_shard in iterable_dataset_shards:
            # Since our random iterable dataset will be... random... we need to use a seed to get reprod
            random.seed(seed)
            iterable_dataset_lists.append(list(iterable_dataset_shard))

        shard_batch_size = batch_size // num_processes if split_batches else batch_size
        # All iterable dataset shard should have the same length, a round multiple of shard_batch_size
        first_list = iterable_dataset_lists[0]
        for l in iterable_dataset_lists[1:]:
            self.assertEqual(len(l), len(first_list))
            self.assertTrue(len(l) % shard_batch_size == 0)

        observed = []
        for idx in range(0, len(first_list), shard_batch_size):
            for l in iterable_dataset_lists:
                observed += l[idx : idx + shard_batch_size]

        if not drop_last:
            while len(reference) < len(observed):
                reference += reference
        self.assertListEqual(observed, reference[: len(observed)])

    def test_iterable_dataset_shard(self):
        seed = 42
        dataset = RandomIterableDataset()

        self.check_iterable_dataset_shards(dataset, seed, batch_size=4, drop_last=False, split_batches=F
        self.check_iterable_dataset_shards(dataset, seed, batch_size=4, drop_last=True, split_batches=Fa
        self.check_iterable_dataset_shards(dataset, seed, batch_size=4, drop_last=False, split_batches=T
        self.check_iterable_dataset_shards(dataset, seed, batch_size=4, drop_last=True, split_batches=Tr

        # Edge case with a very small dataset
        dataset = RandomIterableDataset(max_length=2)
        self.check_iterable_dataset_shards(dataset, seed, batch_size=4, drop_last=False, split_batches=F
        self.check_iterable_dataset_shards(dataset, seed, batch_size=4, drop_last=True, split_batches=Fa
        self.check_iterable_dataset_shards(dataset, seed, batch_size=4, drop_last=False, split_batches=T
        self.check_iterable_dataset_shards(dataset, seed, batch_size=4, drop_last=True, split_batches=Tr

    def test_skip_batch_sampler(self):
        batch_sampler = BatchSampler(range(16), batch_size=4, drop_last=False)
        new_batch_sampler = SkipBatchSampler(batch_sampler, 2)
        self.assertListEqual(list(new_batch_sampler), [[8, 9, 10, 11], [12, 13, 14, 15]])

    def test_skip_data_loader(self):
        dataloader = SkipDataLoader(list(range(16)), batch_size=4, skip_batches=2)
        self.assertListEqual([t.tolist() for t in dataloader], [[8, 9, 10, 11], [12, 13, 14, 15]])

    def test_skip_first_batches(self):
        dataloader = DataLoader(list(range(16)), batch_size=4)
        new_dataloader = skip_first_batches(dataloader, num_batches=2)
        self.assertListEqual([t.tolist() for t in new_dataloader], [[8, 9, 10, 11], [12, 13, 14, 15]])
```

accelerate-main/tests/test_modeling_utils.py

```
# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import json
import os
import tempfile
import unittest

import torch
import torch.nn as nn

from accelerate.test_utils import require_cuda, require_multi_gpu, require_safetensors
from accelerate.test_utils.testing import require_torch_min_version
from accelerate.utils.modeling import (
    check_device_map,
    clean_device_map,
    compute_module_sizes,
    find_tied_parameters,
    get_balanced_memory,
    infer_auto_device_map,
    load_checkpoint_in_model,
    load_state_dict,
    named_module_tensors,
    set_module_tensor_to_device,
)


class ModelForTest(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear1 = nn.Linear(3, 4)
        self.batchnorm = nn.BatchNorm1d(4)
        self.linear2 = nn.Linear(4, 5)

    def forward(self, x):
        return self.linear2(self.batchnorm(self.linear1(x)))


@require_torch_min_version(version="1.9.0")
class ModelingUtilsTester(unittest.TestCase):
    def check_set_module_tensor_for_device(self, model, device1, device2):
        self.assertEqual(model.linear1.weight.device, torch.device(device1))

        with self.subTest("Access by submodule and direct name for a parameter"):
            set_module_tensor_to_device(model.linear1, "weight", device2)
            self.assertEqual(model.linear1.weight.device, torch.device(device2))

            if torch.device(device2) == torch.device("meta"):
                with self.assertRaises(ValueError):
                    # We need a `value` to set the weight back on device1
                    set_module_tensor_to_device(model.linear1, "weight", device1)

                set_module_tensor_to_device(model.linear1, "weight", device1, value=torch.randn(4, 3))
            else:
                set_module_tensor_to_device(model.linear1, "weight", device1)
            self.assertEqual(model.linear1.weight.device, torch.device(device1))
        with self.subTest("Access by module and full name for a parameter"):
            set_module_tensor_to_device(model, "linear1.weight", device2)
            self.assertEqual(model.linear1.weight.device, torch.device(device2))

            if torch.device(device2) == torch.device("meta"):
                with self.assertRaises(ValueError):
                    # We need a `value` to set the weight back on device1
                    set_module_tensor_to_device(model, "linear1.weight", device1)
                set_module_tensor_to_device(model, "linear1.weight", device1, value=torch.randn(4, 3))
            else:
                set_module_tensor_to_device(model, "linear1.weight", device1)
            self.assertEqual(model.linear1.weight.device, torch.device(device1))

        self.assertEqual(model.batchnorm.running_mean.device, torch.device(device1))

        with self.subTest("Access by submodule and direct name for a buffer"):
            set_module_tensor_to_device(model.batchnorm, "running_mean", device2)
            self.assertEqual(model.batchnorm.running_mean.device, torch.device(device2))

            if torch.device(device2) == torch.device("meta"):
                with self.assertRaises(ValueError):
                    # We need a `value` to set the weight back on device1
                    set_module_tensor_to_device(model.batchnorm, "running_mean", device1)
                set_module_tensor_to_device(model.batchnorm, "running_mean", device1, value=torch.randn(
            else:
                set_module_tensor_to_device(model.batchnorm, "running_mean", device1)
            self.assertEqual(model.batchnorm.running_mean.device, torch.device(device1))

        with self.subTest("Access by module and full name for a parameter"):
            set_module_tensor_to_device(model, "batchnorm.running_mean", device2)
            self.assertEqual(model.batchnorm.running_mean.device, torch.device(device2))

            if torch.device(device2) == torch.device("meta"):
                with self.assertRaises(ValueError):
                    # We need a `value` to set the weight back on CPU
                    set_module_tensor_to_device(model, "batchnorm.running_mean", device1)

                set_module_tensor_to_device(model, "batchnorm.running_mean", device1, value=torch.randn(
            else:
                set_module_tensor_to_device(model, "batchnorm.running_mean", device1)
            self.assertEqual(model.batchnorm.running_mean.device, torch.device(device1))

    def test_set_module_tensor_to_meta_and_cpu(self):
        model = ModelForTest()
        self.check_set_module_tensor_for_device(model, "cpu", "meta")

    @require_cuda
    def test_set_module_tensor_to_cpu_and_gpu(self):
        model = ModelForTest()
        self.check_set_module_tensor_for_device(model, "cpu", 0)

    @require_cuda
    def test_set_module_tensor_to_meta_and_gpu(self):
        model = ModelForTest().to(0)
        self.check_set_module_tensor_for_device(model, 0, "meta")

    @require_multi_gpu
    def test_set_module_tensor_between_gpus(self):
        model = ModelForTest().to(0)
        self.check_set_module_tensor_for_device(model, 0, 1)

    def test_set_module_tensor_sets_dtype(self):
        model = ModelForTest()
        set_module_tensor_to_device(model, "linear1.weight", "cpu", value=model.linear1.weight, dtype=to
        self.assertEqual(model.linear1.weight.dtype, torch.float16)

    def test_named_tensors(self):
        model = nn.BatchNorm1d(4)
        named_tensors = named_module_tensors(model)
        self.assertListEqual(
            [name for name, _ in named_tensors],
            ["weight", "bias", "running_mean", "running_var", "num_batches_tracked"],
        )
        named_tensors = named_module_tensors(model, include_buffers=False)
        self.assertListEqual([name for name, _ in named_tensors], ["weight", "bias"])

        model = ModelForTest()
        named_tensors = named_module_tensors(model)
        self.assertListEqual([name for name, _ in named_tensors], [])

        named_tensors = named_module_tensors(model, recurse=True)
        self.assertListEqual(
            [name for name, _ in named_tensors],
            [
                "linear1.weight",
                "linear1.bias",
                "batchnorm.weight",
                "batchnorm.bias",
                "linear2.weight",
                "linear2.bias",
                "batchnorm.running_mean",
                "batchnorm.running_var",
                "batchnorm.num_batches_tracked",
            ],
        )

        named_tensors = named_module_tensors(model, include_buffers=False, recurse=True)
        self.assertListEqual(
            [name for name, _ in named_tensors],
            ["linear1.weight", "linear1.bias", "batchnorm.weight", "batchnorm.bias", "linear2.weight", "
        )

    def test_find_tied_parameters(self):
        model = ModelForTest()
        self.assertDictEqual(find_tied_parameters(model), {})
        model.linear2.weight = model.linear1.weight
        self.assertDictEqual(find_tied_parameters(model), {"linear1.weight": "linear2.weight"})

    def test_compute_module_sizes(self):
        model = ModelForTest()
        expected_sizes = {"": 236, "linear1": 64, "linear1.weight": 48, "linear1.bias": 16}
        expected_sizes.update({"linear2": 100, "linear2.weight": 80, "linear2.bias": 20})
        expected_sizes.update({"batchnorm": 72, "batchnorm.weight": 16, "batchnorm.bias": 16})
        expected_sizes.update(
            {"batchnorm.running_mean": 16, "batchnorm.running_var": 16, "batchnorm.num_batches_tracked":
        )

        module_sizes = compute_module_sizes(model)
        self.assertDictEqual(module_sizes, expected_sizes)

        model.half()
        expected_sizes = {k: s // 2 for k, s in expected_sizes.items()}
        # This one is not converted to half.
        expected_sizes["batchnorm.num_batches_tracked"] = 8
        # This impacts batchnorm and total
        expected_sizes["batchnorm"] += 4
        expected_sizes[""] += 4

        module_sizes = compute_module_sizes(model)
        self.assertDictEqual(module_sizes, expected_sizes)

    def test_check_device_map(self):
        model = ModelForTest()
        check_device_map(model, {"": 0})
        with self.assertRaises(ValueError):
            check_device_map(model, {"linear1": 0, "linear2": 1})

        check_device_map(model, {"linear1": 0, "linear2": 1, "batchnorm": 1})

    def shard_test_model(self, model, tmp_dir):
        module_index = {
            "linear1": "checkpoint_part1.bin",
            "batchnorm": "checkpoint_part2.bin",
            "linear2": "checkpoint_part3.bin",
        }
        index = {}
        for name, _ in model.state_dict().items():
            module = name.split(".")[0]
            index[name] = module_index[module]

        with open(os.path.join(tmp_dir, "weight_map.index.json"), "w") as f:
            json.dump(index, f)

        for module, fname in module_index.items():
            state_dict = {k: v for k, v in model.state_dict().items() if k.startswith(module)}
            full_fname = os.path.join(tmp_dir, fname)
            torch.save(state_dict, full_fname)

    def test_load_checkpoint_in_model(self):
        # Check with whole checkpoint
        model = ModelForTest()
        with tempfile.TemporaryDirectory() as tmp_dir:
            fname = os.path.join(tmp_dir, "pt_model.bin")
            torch.save(model.state_dict(), fname)
            load_checkpoint_in_model(model, fname)

        # Check with sharded index
        model = ModelForTest()
        with tempfile.TemporaryDirectory() as tmp_dir:
            self.shard_test_model(model, tmp_dir)
            index_file = os.path.join(tmp_dir, "weight_map.index.json")
            load_checkpoint_in_model(model, index_file)

        # Check with sharded checkpoint
        model = ModelForTest()
        with tempfile.TemporaryDirectory() as tmp_dir:
            self.shard_test_model(model, tmp_dir)
            load_checkpoint_in_model(model, tmp_dir)

    @require_cuda
    def test_load_checkpoint_in_model_one_gpu(self):
        device_map = {"linear1": 0, "batchnorm": "cpu", "linear2": "cpu"}

        # Check with whole checkpoint
        model = ModelForTest()
        with tempfile.TemporaryDirectory() as tmp_dir:
            fname = os.path.join(tmp_dir, "pt_model.bin")
            torch.save(model.state_dict(), fname)
            load_checkpoint_in_model(model, fname, device_map=device_map)
        self.assertEqual(model.linear1.weight.device, torch.device(0))
        self.assertEqual(model.batchnorm.weight.device, torch.device("cpu"))
        self.assertEqual(model.linear2.weight.device, torch.device("cpu"))

        # Check with sharded index
        model = ModelForTest()
        with tempfile.TemporaryDirectory() as tmp_dir:
            self.shard_test_model(model, tmp_dir)
            index_file = os.path.join(tmp_dir, "weight_map.index.json")
            load_checkpoint_in_model(model, index_file, device_map=device_map)

        self.assertEqual(model.linear1.weight.device, torch.device(0))
        self.assertEqual(model.batchnorm.weight.device, torch.device("cpu"))
        self.assertEqual(model.linear2.weight.device, torch.device("cpu"))

        # Check with sharded checkpoint folder
        model = ModelForTest()
        with tempfile.TemporaryDirectory() as tmp_dir:
            self.shard_test_model(model, tmp_dir)
            load_checkpoint_in_model(model, tmp_dir, device_map=device_map)

        self.assertEqual(model.linear1.weight.device, torch.device(0))
        self.assertEqual(model.batchnorm.weight.device, torch.device("cpu"))
        self.assertEqual(model.linear2.weight.device, torch.device("cpu"))

    @require_cuda
    def test_load_checkpoint_in_model_disk_offload(self):
        device_map = {"linear1": "cpu", "batchnorm": "disk", "linear2": "cpu"}
        model = ModelForTest()
        with tempfile.TemporaryDirectory() as tmp_dir:
            fname = os.path.join(tmp_dir, "pt_model.bin")
            torch.save(model.state_dict(), fname)
            load_checkpoint_in_model(model, fname, device_map=device_map, offload_folder=tmp_dir)
        self.assertEqual(model.linear1.weight.device, torch.device("cpu"))
        self.assertEqual(model.batchnorm.weight.device, torch.device("meta"))
        # Buffers are not offloaded by default
        self.assertEqual(model.batchnorm.running_mean.device, torch.device("cpu"))
        self.assertEqual(model.linear2.weight.device, torch.device("cpu"))

        model = ModelForTest()
        with tempfile.TemporaryDirectory() as tmp_dir:
            fname = os.path.join(tmp_dir, "pt_model.bin")
            torch.save(model.state_dict(), fname)
            load_checkpoint_in_model(model, fname, device_map=device_map, offload_folder=tmp_dir, offloa
        self.assertEqual(model.linear1.weight.device, torch.device("cpu"))
        self.assertEqual(model.batchnorm.weight.device, torch.device("meta"))
        self.assertEqual(model.batchnorm.running_mean.device, torch.device("meta"))
        self.assertEqual(model.linear2.weight.device, torch.device("cpu"))

    @require_multi_gpu
    def test_load_checkpoint_in_model_two_gpu(self):
        device_map = {"linear1": 0, "batchnorm": "cpu", "linear2": 1}

        # Check with whole checkpoint
        model = ModelForTest()
        with tempfile.TemporaryDirectory() as tmp_dir:
            fname = os.path.join(tmp_dir, "pt_model.bin")
            torch.save(model.state_dict(), fname)
            load_checkpoint_in_model(model, fname, device_map=device_map)
        self.assertEqual(model.linear1.weight.device, torch.device(0))
        self.assertEqual(model.batchnorm.weight.device, torch.device("cpu"))
        self.assertEqual(model.linear2.weight.device, torch.device(1))

        # Check with sharded index
        model = ModelForTest()
        with tempfile.TemporaryDirectory() as tmp_dir:
            self.shard_test_model(model, tmp_dir)
            index_file = os.path.join(tmp_dir, "weight_map.index.json")
            load_checkpoint_in_model(model, index_file, device_map=device_map)

        self.assertEqual(model.linear1.weight.device, torch.device(0))
        self.assertEqual(model.batchnorm.weight.device, torch.device("cpu"))
        self.assertEqual(model.linear2.weight.device, torch.device(1))

        # Check with sharded checkpoint
        model = ModelForTest()
        with tempfile.TemporaryDirectory() as tmp_dir:
            self.shard_test_model(model, tmp_dir)
            load_checkpoint_in_model(model, tmp_dir, device_map=device_map)

        self.assertEqual(model.linear1.weight.device, torch.device(0))
        self.assertEqual(model.batchnorm.weight.device, torch.device("cpu"))
        self.assertEqual(model.linear2.weight.device, torch.device(1))

    def test_clean_device_map(self):
        # Regroup everything if all is on the same device
        self.assertDictEqual(clean_device_map({"a": 0, "b": 0, "c": 0}), {"": 0})
        # Regroups children of level 1 on the same device
        self.assertDictEqual(
            clean_device_map({"a.x": 0, "a.y": 0, "b.x": 1, "b.y": 1, "c": 1}), {"a": 0, "b": 1, "c": 1}
        )
        # Regroups children of level 2 on the same device
        self.assertDictEqual(
            clean_device_map({"a.x": 0, "a.y": 0, "b.x.0": 1, "b.x.1": 1, "b.y.0": 2, "b.y.1": 2, "c": 2
            {"a": 0, "b.x": 1, "b.y": 2, "c": 2},
        )

    def test_infer_auto_device_map(self):
        model = ModelForTest()
        # model has size 236: linear1 64, batchnorm 72, linear2 100
        device_map = infer_auto_device_map(model, max_memory={0: 200, 1: 200})
        # only linear1 fits on device 0 as we keep memory available for the maximum layer in case of off
        self.assertDictEqual(device_map, {"linear1": 0, "batchnorm": 1, "linear2": 1})

        device_map = infer_auto_device_map(model, max_memory={0: 200, 1: 172, 2: 200})
        # On device 1, we don't care about keeping size available for the max layer, so even if there is
        # size available for batchnorm + linear2, they fit here.
        self.assertDictEqual(device_map, {"linear1": 0, "batchnorm": 1, "linear2": 1})

        model.linear1.weight = model.linear2.weight
        device_map = infer_auto_device_map(model, max_memory={0: 200, 1: 200})
        # By tying weights, the whole model fits on device 0
        self.assertDictEqual(device_map, {"": 0})

        # When splitting a bigger model, the split is done at the layer level
        model = nn.Sequential(ModelForTest(), ModelForTest(), ModelForTest())
        device_map = infer_auto_device_map(model, max_memory={0: 500, 1: 500})
        self.assertDictEqual(device_map, {"0": 0, "1.linear1": 0, "1.batchnorm": 0, "1.linear2": 1, "2":

        # With no_split_module_classes, it's done at that module level
        model = nn.Sequential(ModelForTest(), ModelForTest(), ModelForTest())
        device_map = infer_auto_device_map(
            model, max_memory={0: 500, 1: 500}, no_split_module_classes=["ModelForTest"]
        )
        self.assertDictEqual(device_map, {"0": 0, "1": 1, "2": 1})

        # Now if we have weights tied inside submodules, tied weights are on the same device.
        model = nn.Sequential(ModelForTest(), ModelForTest(), ModelForTest())
        layer0 = getattr(model, "0")
        layer2 = getattr(model, "2")
        layer0.linear2.weight = layer2.linear2.weight
        device_map = infer_auto_device_map(model, max_memory={0: 400, 1: 500})
        expected = {"0": 0, "2.linear2": 0, "1": 1, "2.linear1": 1, "2.batchnorm": 1}
        self.assertDictEqual(device_map, expected)

    @require_cuda
    def test_get_balanced_memory(self):
        model = ModelForTest()
        # model has size 236: linear1 64, batchnorm 72, linear2 100
        max_memory = get_balanced_memory(model, max_memory={0: 200, 1: 200})
        self.assertDictEqual({0: 200, 1: 200}, max_memory)

        max_memory = get_balanced_memory(model, max_memory={0: 300, 1: 300})
        self.assertDictEqual({0: 215, 1: 300}, max_memory)

        # Last device always get max memory to give more buffer and avoid accidental CPU offload
        max_memory = get_balanced_memory(model, max_memory={0: 300, 1: 500})
        self.assertDictEqual({0: 215, 1: 500}, max_memory)

        # Last device always get max memory to give more buffer, even if CPU is provided
        max_memory = get_balanced_memory(model, max_memory={0: 300, "cpu": 1000})
        self.assertDictEqual({0: 300, "cpu": 1000}, max_memory)

        # If we set a device to 0, it's not counted.
        max_memory = get_balanced_memory(model, max_memory={0: 0, 1: 300, 2: 300})
        self.assertDictEqual({0: 0, 1: 215, 2: 300}, max_memory)

    @require_cuda
    @require_safetensors
    def test_load_state_dict(self):
        from safetensors.torch import save_file

        state_dict = {k: torch.randn(4, 5) for k in ["a", "b", "c"]}
        device_maps = [{"a": "cpu", "b": 0, "c": "disk"}, {"a": 0, "b": 0, "c": "disk"}, {"a": 0, "b": 0

        for device_map in device_maps:
            with tempfile.TemporaryDirectory() as tmp_dir:
                checkpoint_file = os.path.join(tmp_dir, "model.safetensors")
                save_file(state_dict, checkpoint_file, metadata={"format": "pt"})

                loaded_state_dict = load_state_dict(checkpoint_file, device_map=device_map)
            for param, device in device_map.items():
```

```
device = device if device != "disk" else "cpu"
self.assertEqual(loaded_state_dict[param].device, torch.device(device))
```

accelerate-main/tests/test_offload.py

```
# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import os
import unittest
from tempfile import TemporaryDirectory

import torch
import torch.nn as nn

from accelerate.utils import (
    OffloadedWeightsLoader,
    extract_submodules_state_dict,
    is_torch_version,
    load_offloaded_weight,
    offload_state_dict,
    offload_weight,
)


class ModelForTest(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear1 = nn.Linear(3, 4)
        self.batchnorm = nn.BatchNorm1d(4)
        self.linear2 = nn.Linear(4, 5)

    def forward(self, x):
        return self.linear2(self.batchnorm(self.linear1(x)))


class OffloadTester(unittest.TestCase):
    def test_offload_state_dict(self):
        model = ModelForTest()
        with TemporaryDirectory() as tmp_dir:
            offload_state_dict(tmp_dir, model.state_dict())
            index_file = os.path.join(tmp_dir, "index.json")
            self.assertTrue(os.path.isfile(index_file))
            # TODO: add tests on what is inside the index

            for key in ["linear1.weight", "linear1.bias", "linear2.weight", "linear2.bias"]:
                weight_file = os.path.join(tmp_dir, f"{key}.dat")
                self.assertTrue(os.path.isfile(weight_file))
                # TODO: add tests on the fact weights are properly loaded

    def test_offload_weight(self):
        dtypes = [torch.float16, torch.float32]
        if is_torch_version(">=", "1.10"):
            dtypes.append(torch.bfloat16)

        for dtype in dtypes:
            weight = torch.randn(2, 3, dtype=dtype)
            with TemporaryDirectory() as tmp_dir:
                index = offload_weight(weight, "weight", tmp_dir, {})
                weight_file = os.path.join(tmp_dir, "weight.dat")
                self.assertTrue(os.path.isfile(weight_file))
                self.assertDictEqual(index, {"weight": {"shape": [2, 3], "dtype": str(dtype).split(".")[
            new_weight = load_offloaded_weight(weight_file, index["weight"])
            self.assertTrue(torch.equal(weight, new_weight))

    def test_offload_weights_loader(self):
        model = ModelForTest()
        state_dict = model.state_dict()
        cpu_part = {k: v for k, v in state_dict.items() if "linear2" not in k}
        disk_part = {k: v for k, v in state_dict.items() if "linear2" in k}

        with TemporaryDirectory() as tmp_dir:
            offload_state_dict(tmp_dir, disk_part)
            weight_map = OffloadedWeightsLoader(state_dict=cpu_part, save_folder=tmp_dir)

            # Every key is there with the right value
            self.assertEqual(sorted(weight_map), sorted(state_dict.keys()))
            for key, param in state_dict.items():
                self.assertTrue(torch.allclose(param, weight_map[key]))

        cpu_part = {k: v for k, v in state_dict.items() if "weight" in k}
        disk_part = {k: v for k, v in state_dict.items() if "weight" not in k}

        with TemporaryDirectory() as tmp_dir:
            offload_state_dict(tmp_dir, disk_part)
            weight_map = OffloadedWeightsLoader(state_dict=cpu_part, save_folder=tmp_dir)

            # Every key is there with the right value
            self.assertEqual(sorted(weight_map), sorted(state_dict.keys()))
            for key, param in state_dict.items():
                self.assertTrue(torch.allclose(param, weight_map[key]))

        with TemporaryDirectory() as tmp_dir:
            offload_state_dict(tmp_dir, state_dict)
            # Duplicates are removed
            weight_map = OffloadedWeightsLoader(state_dict=cpu_part, save_folder=tmp_dir)

            # Every key is there with the right value
            self.assertEqual(sorted(weight_map), sorted(state_dict.keys()))
            for key, param in state_dict.items():
                self.assertTrue(torch.allclose(param, weight_map[key]))

    def test_extract_submodules_state_dict(self):
        state_dict = {"a.1": 0, "a.10": 1, "a.2": 2}
        extracted = extract_submodules_state_dict(state_dict, ["a.1", "a.2"])
        self.assertDictEqual(extracted, {"a.1": 0, "a.2": 2})

        state_dict = {"a.1.a": 0, "a.10.a": 1, "a.2.a": 2}
        extracted = extract_submodules_state_dict(state_dict, ["a.1", "a.2"])
        self.assertDictEqual(extracted, {"a.1.a": 0, "a.2.a": 2})
```

accelerate-main/tests/xla_spawn.py

```
# Copyright 2021 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

"""
A simple launcher script for TPU training

Inspired by https://github.com/pytorch/pytorch/blob/master/torch/distributed/launch.py

::
    >>> python xla_spawn.py --num_cores=NUM_CORES_YOU_HAVE
               YOUR_TRAINING_SCRIPT.py (--arg1 --arg2 --arg3 and all other
               arguments of your training script)

"""


import importlib
import sys
from argparse import REMAINDER, ArgumentParser
from pathlib import Path

import torch_xla.distributed.xla_multiprocessing as xmp


def parse_args():
    """
    Helper function parsing the command line options
    @retval ArgumentParser
    """
    parser = ArgumentParser(
        description=(
            "PyTorch TPU distributed training launch "
            "helper utility that will spawn up "
            "multiple distributed processes"
        )
    )

    # Optional arguments for the launch helper
    parser.add_argument("--num_cores", type=int, default=1, help="Number of TPU cores to use (1 or 8).")

    # positional
    parser.add_argument(
        "training_script",
        type=str,
        help=(
            "The full path to the single TPU training "
            "program/script to be launched in parallel, "
            "followed by all the arguments for the "
            "training script"
        ),
    )

    # rest from the training program
    parser.add_argument("training_script_args", nargs=REMAINDER)

    return parser.parse_args()
def main():
    args = parse_args()

    # Import training_script as a module.
    script_fpath = Path(args.training_script)
    sys.path.append(str(script_fpath.parent.resolve()))
    mod_name = script_fpath.stem
    mod = importlib.import_module(mod_name)

    # Patch sys.argv
    sys.argv = [args.training_script] + args.training_script_args + ["--tpu_num_cores", str(args.num_cor
    xmp.spawn(mod._mp_fn, args=(), nprocs=args.num_cores)


if __name__ == "__main__":
    main()
```

accelerate-main/tests/test_scheduler.py

```
# Copyright 2021 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import unittest
from functools import partial

import torch

from accelerate import Accelerator, debug_launcher
from accelerate.state import AcceleratorState, GradientState
from accelerate.test_utils import require_cpu, require_huggingface_suite
from accelerate.utils import GradientAccumulationPlugin


def one_cycle_test(num_processes=2, step_scheduler_with_optimizer=True, split_batches=False):
    accelerator = Accelerator(step_scheduler_with_optimizer=step_scheduler_with_optimizer, split_batches
    model = torch.nn.Linear(2, 4)
    optimizer = torch.optim.AdamW(model.parameters(), lr=1.0)
    scheduler = torch.optim.lr_scheduler.OneCycleLR(optimizer, max_lr=0.01, steps_per_epoch=2, epochs=1)
    model, optimizer, scheduler = accelerator.prepare(model, optimizer, scheduler)

    # Optimizer has stepped
    scheduler.step()
    if step_scheduler_with_optimizer or (num_processes == 1):
        assert (
            scheduler.scheduler.last_epoch == num_processes
        ), f"Last Epoch ({scheduler.scheduler.last_epoch}) != Num Processes ({num_processes})"
    else:
        assert (
            scheduler.scheduler.last_epoch != num_processes
        ), f"Last Epoch ({scheduler.scheduler.last_epoch}) == Num Processes ({num_processes})"


def lambda_test(num_processes=2, step_scheduler_with_optimizer=True, split_batches=False):
    accelerator = Accelerator(step_scheduler_with_optimizer=step_scheduler_with_optimizer, split_batches
    model = torch.nn.Linear(2, 4)
    optimizer = torch.optim.AdamW(model.parameters(), lr=1.0)
    scheduler = torch.optim.lr_scheduler.LambdaLR(optimizer, lr_lambda=lambda n: 1 - n / 10)
    model, optimizer, scheduler = accelerator.prepare(model, optimizer, scheduler)

    # Optimizer has stepped
    optimizer._is_overflow = False
    scheduler.step()
    expected_lr = 1 - (num_processes if (step_scheduler_with_optimizer and not split_batches) else 1) /
    assert (
        scheduler.get_last_lr()[0] == expected_lr
    ), f"Wrong lr found at first step, expected {expected_lr}, got {scheduler.get_last_lr()[0]}"

    # Optimizer has not stepped
    optimizer._is_overflow = True
    scheduler.step()
    if not step_scheduler_with_optimizer:
        expected_lr = 1 - 2 / 10
    assert (
        scheduler.get_last_lr()[0] == expected_lr
    ), f"Wrong lr found at second step, expected {expected_lr}, got {scheduler.get_last_lr()[0]}"
def accumulation_test(num_processes: int = 2):
    """
    With this test, an observed batch size of 64 should result in neglible
    differences in the scheduler after going through the correct number of steps.

    Uses single, two, and four steps to test.
    """
    from transformers import get_linear_schedule_with_warmup

    steps = [1, 2, 4]
    for num_steps in steps:
        plugin = GradientAccumulationPlugin(num_steps=num_steps, adjust_scheduler=num_steps > 1)
        accelerator = Accelerator(gradient_accumulation_plugin=plugin)
        model = torch.nn.Linear(2, 4)
        optimizer = torch.optim.AdamW(model.parameters(), lr=10.0)
        scheduler = get_linear_schedule_with_warmup(optimizer=optimizer, num_warmup_steps=0, num_trainin

        model, optimizer, scheduler = accelerator.prepare(model, optimizer, scheduler)

        for i in range(10 * num_steps):
            with accelerator.accumulate(model):
                optimizer.step()
                scheduler.step()

            if i == (10 * num_steps - 2):
                assert (
                    scheduler.get_last_lr()[0] != 0
                ), f"Wrong lr found at second-to-last step, expected non-zero, got {scheduler.get_last_l
        assert (
            scheduler.get_last_lr()[0] == 0
        ), f"Wrong lr found at last step, expected 0, got {scheduler.get_last_lr()[0]}"
        GradientState._reset_state()


@require_cpu
class SchedulerTester(unittest.TestCase):
    def test_lambda_scheduler_steps_with_optimizer_single_process(self):
        debug_launcher(partial(lambda_test, num_processes=1), num_processes=1)
        debug_launcher(partial(lambda_test, num_processes=1, split_batches=True), num_processes=1)

    def test_one_cycle_scheduler_steps_with_optimizer_single_process(self):
        debug_launcher(partial(one_cycle_test, num_processes=1), num_processes=1)
        debug_launcher(partial(one_cycle_test, num_processes=1, split_batches=True), num_processes=1)

    def test_lambda_scheduler_not_step_with_optimizer_single_process(self):
        debug_launcher(partial(lambda_test, num_processes=1, step_scheduler_with_optimizer=False), num_p

    def test_one_cycle_scheduler_not_step_with_optimizer_single_process(self):
        debug_launcher(partial(one_cycle_test, num_processes=1, step_scheduler_with_optimizer=False), nu

    def test_lambda_scheduler_steps_with_optimizer_multiprocess(self):
        AcceleratorState._reset_state(True)
        debug_launcher(lambda_test)
        debug_launcher(partial(lambda_test, num_processes=1, split_batches=True), num_processes=1)

    def test_one_cycle_scheduler_steps_with_optimizer_multiprocess(self):
        AcceleratorState._reset_state(True)
        debug_launcher(one_cycle_test)
        debug_launcher(partial(one_cycle_test, num_processes=1, split_batches=True), num_processes=1)

    def test_lambda_scheduler_not_step_with_optimizer_multiprocess(self):
        AcceleratorState._reset_state(True)
        debug_launcher(partial(lambda_test, step_scheduler_with_optimizer=False))

    def test_one_cycle_scheduler_not_step_with_optimizer_multiprocess(self):
        AcceleratorState._reset_state(True)
        debug_launcher(partial(one_cycle_test, step_scheduler_with_optimizer=False))

    @require_huggingface_suite
    def test_accumulation(self):
        AcceleratorState._reset_state(True)
        debug_launcher(partial(accumulation_test, num_processes=1))
        debug_launcher(accumulation_test)
```

accelerate-main/tests/test_sagemaker.py

```
import unittest
from dataclasses import dataclass

import pytest

from accelerate.commands.config.config_args import SageMakerConfig
from accelerate.utils import ComputeEnvironment
from accelerate.utils.launch import _convert_nargs_to_dict


@dataclass
class MockLaunchConfig(SageMakerConfig):
    compute_environment = ComputeEnvironment.AMAZON_SAGEMAKER
    fp16 = True
    ec2_instance_type = "ml.p3.2xlarge"
    iam_role_name = "accelerate_sagemaker_execution_role"
    profile = "hf-sm"
    region = "us-east-1"
    num_machines = 1
    base_job_name = "accelerate-sagemaker-1"
    pytorch_version = "1.6"
    transformers_version = "4.4"
    training_script = "train.py"
    success_training_script_args = [
        "--model_name_or_path",
        "bert",
        "--do_train",
        "False",
        "--epochs",
        "3",
        "--learning_rate",
        "5e-5",
        "--max_steps",
        "50.5",
    ]
    fail_training_script_args = [
        "--model_name_or_path",
        "bert",
        "--do_train",
        "--do_test",
        "False",
        "--do_predict",
        "--epochs",
        "3",
        "--learning_rate",
        "5e-5",
        "--max_steps",
        "50.5",
    ]


class SageMakerLaunch(unittest.TestCase):
    def test_args_convert(self):
        # If no defaults are changed, `to_kwargs` returns an empty dict.
        converted_args = _convert_nargs_to_dict(MockLaunchConfig.success_training_script_args)
        assert isinstance(converted_args["model_name_or_path"], str)
        assert isinstance(converted_args["do_train"], bool)
        assert isinstance(converted_args["epochs"], int)
        assert isinstance(converted_args["learning_rate"], float)
        assert isinstance(converted_args["max_steps"], float)

        with pytest.raises(ValueError):
            _convert_nargs_to_dict(MockLaunchConfig.fail_training_script_args)
```

accelerate-main/tests/test_grad_sync.py

```
# Copyright 2021 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import inspect
import os
import unittest

import torch

import accelerate
from accelerate import debug_launcher
from accelerate.test_utils import (
    execute_subprocess_async,
    require_cpu,
    require_multi_gpu,
    require_single_gpu,
    test_sync,
)
from accelerate.utils import get_launch_prefix, patch_environment


class SyncScheduler(unittest.TestCase):
    def setUp(self):
        mod_file = inspect.getfile(accelerate.test_utils)
        self.test_file_path = os.path.sep.join(mod_file.split(os.path.sep)[:-1] + ["scripts", "test_sync

    @require_cpu
    def test_gradient_sync_cpu_noop(self):
        debug_launcher(test_sync.main, num_processes=1)

    @require_cpu
    def test_gradient_sync_cpu_multi(self):
        debug_launcher(test_sync.main)

    @require_single_gpu
    def test_gradient_sync_gpu(self):
        test_sync.main()

    @require_multi_gpu
    def test_gradient_sync_gpu_multi(self):
        print(f"Found {torch.cuda.device_count()} devices.")
        cmd = get_launch_prefix() + [f"--nproc_per_node={torch.cuda.device_count()}", self.test_file_pat
        with patch_environment(omp_num_threads=1):
            execute_subprocess_async(cmd, env=os.environ.copy())
```

accelerate-main/tests/test_examples.py

```
# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import ast
import os
import re
import shutil
import tempfile
import unittest
from unittest import mock

import torch

from accelerate.test_utils.examples import compare_against_test
from accelerate.test_utils.testing import TempDirTestCase, require_trackers, run_command, slow
from accelerate.utils import write_basic_config


# DataLoaders built from `test_samples/MRPC` for quick testing
# Should mock `{script_name}.get_dataloaders` via:
# @mock.patch("{script_name}.get_dataloaders", mocked_dataloaders)

EXCLUDE_EXAMPLES = [
    "cross_validation.py",
    "gradient_accumulation.py",
    "multi_process_metrics.py",
    "memory.py",
    "automatic_gradient_accumulation.py",
    "fsdp_with_peak_mem_tracking.py",
    "deepspeed_with_config_support.py",
    "megatron_lm_gpt_pretraining.py",
]


class ExampleDifferenceTests(unittest.TestCase):
    """
    This TestCase checks that all of the `complete_*` scripts contain all of the
    information found in the `by_feature` scripts, line for line. If one fails,
    then a complete example does not contain all of the features in the features
    scripts, and should be updated.

    Each example script should be a single test (such as `test_nlp_example`),
    and should run `one_complete_example` twice: once with `parser_only=True`,
    and the other with `parser_only=False`. This is so that when the test
    failures are returned to the user, they understand if the discrepancy lies in
    the `main` function, or the `training_loop` function. Otherwise it will be
    unclear.

    Also, if there are any expected differences between the base script used and
    `complete_nlp_example.py` (the canonical base script), these should be included in
    `special_strings`. These would be differences in how something is logged, print statements,
    etc (such as calls to `Accelerate.log()`)
    """

    def one_complete_example(
        self, complete_file_name: str, parser_only: bool, secondary_filename: str = None, special_string
    ):
        """
        Tests a single `complete` example against all of the implemented `by_feature` scripts

        Args:
            complete_file_name (`str`):
                The filename of a complete example
            parser_only (`bool`):
                Whether to look at the main training function, or the argument parser
            secondary_filename (`str`, *optional*):
                A potential secondary base file to strip all script information not relevant for checkin
                such as "cv_example.py" when testing "complete_cv_example.py"
            special_strings (`list`, *optional*):
                A list of strings to potentially remove before checking no differences are left. These s
                diffs that are file specific, such as different logging variations between files.
        """
        self.maxDiff = None
        by_feature_path = os.path.abspath(os.path.join("examples", "by_feature"))
        examples_path = os.path.abspath("examples")
        for item in os.listdir(by_feature_path):
            if item not in EXCLUDE_EXAMPLES:
                item_path = os.path.join(by_feature_path, item)
                if os.path.isfile(item_path) and ".py" in item_path:
                    with self.subTest(
                        tested_script=complete_file_name,
                        feature_script=item,
                        tested_section="main()" if parser_only else "training_function()",
                    ):
                        diff = compare_against_test(
                            os.path.join(examples_path, complete_file_name), item_path, parser_only, sec
                        )
                        diff = "\n".join(diff)
                        if special_strings is not None:
                            for string in special_strings:
                                diff = diff.replace(string, "")
                        self.assertEqual(diff, "")

    def test_nlp_examples(self):
        self.one_complete_example("complete_nlp_example.py", True)
        self.one_complete_example("complete_nlp_example.py", False)

    def test_cv_examples(self):
        cv_path = os.path.abspath(os.path.join("examples", "cv_example.py"))
        special_strings = [
            " " * 16 + "{\n\n",
            " " * 20 + '"accuracy": eval_metric["accuracy"],\n\n',
            " " * 20 + '"f1": eval_metric["f1"],\n\n',
            " " * 20 + '"train_loss": total_loss.item() / len(train_dataloader),\n\n',
            " " * 20 + '"epoch": epoch,\n\n',
            " " * 16 + "},\n\n",
            " " * 16 + "step=epoch,\n",
            " " * 12,
        ]
        self.one_complete_example("complete_cv_example.py", True, cv_path, special_strings)
        self.one_complete_example("complete_cv_example.py", False, cv_path, special_strings)


@mock.patch.dict(os.environ, {"TESTING_MOCKED_DATALOADERS": "1"})
class FeatureExamplesTests(TempDirTestCase):
    clear_on_setup = False

    @classmethod
    def setUpClass(cls):
        super().setUpClass()
        cls._tmpdir = tempfile.mkdtemp()
        cls.configPath = os.path.join(cls._tmpdir, "default_config.yml")

        write_basic_config(save_location=cls.configPath)
        cls._launch_args = ["accelerate", "launch", "--config_file", cls.configPath]

    @classmethod
    def tearDownClass(cls):
        super().tearDownClass()
        shutil.rmtree(cls._tmpdir)

    def test_checkpointing_by_epoch(self):
        testargs = f"""
        examples/by_feature/checkpointing.py
        --checkpointing_steps epoch
        --output_dir {self.tmpdir}
        """.split()
        run_command(self._launch_args + testargs)
        self.assertTrue(os.path.exists(os.path.join(self.tmpdir, "epoch_0")))

    def test_checkpointing_by_steps(self):
        testargs = f"""
        examples/by_feature/checkpointing.py
        --checkpointing_steps 1
        --output_dir {self.tmpdir}
        """.split()
        _ = run_command(self._launch_args + testargs)
        self.assertTrue(os.path.exists(os.path.join(self.tmpdir, "step_2")))

    def test_load_states_by_epoch(self):
        testargs = f"""
        examples/by_feature/checkpointing.py
        --resume_from_checkpoint {os.path.join(self.tmpdir, "epoch_0")}
        """.split()
        output = run_command(self._launch_args + testargs, return_stdout=True)
        self.assertNotIn("epoch 0:", output)
        self.assertIn("epoch 1:", output)

    def test_load_states_by_steps(self):
        testargs = f"""
        examples/by_feature/checkpointing.py
        --resume_from_checkpoint {os.path.join(self.tmpdir, "step_2")}
        """.split()
        output = run_command(self._launch_args + testargs, return_stdout=True)
        if torch.cuda.is_available():
            num_processes = torch.cuda.device_count()
        else:
            num_processes = 1
        if num_processes > 1:
            self.assertNotIn("epoch 0:", output)
            self.assertIn("epoch 1:", output)
        else:
            self.assertIn("epoch 0:", output)
            self.assertIn("epoch 1:", output)

    @slow
    def test_cross_validation(self):
        testargs = """
        examples/by_feature/cross_validation.py
        --num_folds 2
        """.split()
        with mock.patch.dict(os.environ, {"TESTING_MOCKED_DATALOADERS": "0"}):
            output = run_command(self._launch_args + testargs, return_stdout=True)
            results = ast.literal_eval(re.findall("({.+})", output)[-1])
            self.assertGreaterEqual(results["accuracy"], 0.75)

    def test_multi_process_metrics(self):
        testargs = ["examples/by_feature/multi_process_metrics.py"]
        run_command(self._launch_args + testargs)

    @require_trackers
    @mock.patch.dict(os.environ, {"WANDB_MODE": "offline"})
    def test_tracking(self):
        with tempfile.TemporaryDirectory() as tmpdir:
            testargs = f"""
            examples/by_feature/tracking.py
            --with_tracking
            --logging_dir {tmpdir}
            """.split()
            run_command(self._launch_args + testargs)
            self.assertTrue(os.path.exists(os.path.join(tmpdir, "tracking")))
```

```
def test_gradient_accumulation(self):
    testargs = ["examples/by_feature/gradient_accumulation.py"]
    run_command(self._launch_args + testargs)
```

accelerate-main/docs/source/basic_tutorials/launch.mdx

# Launching your ■ Accelerate scripts

In the previous tutorial, you were introduced to how to modify your current training script to use ■ Acc
The final version of that code is shown below:

```python
from accelerate import Accelerator

accelerator = Accelerator()

model, optimizer, training_dataloader, scheduler = accelerator.prepare(
    model, optimizer, training_dataloader, scheduler
)

for batch in training_dataloader:
    optimizer.zero_grad()
    inputs, targets = batch
    outputs = model(inputs)
    loss = loss_function(outputs, targets)
    accelerator.backward(loss)
    optimizer.step()
    scheduler.step()
```

But how do you run this code and have it utilize the special hardware available to it?

First you should rewrite the above code into a function, and make it callable as a script. For example:

```diff
  from accelerate import Accelerator

+ def main():
      accelerator = Accelerator()

      model, optimizer, training_dataloader, scheduler = accelerator.prepare(
          model, optimizer, training_dataloader, scheduler
      )

      for batch in training_dataloader:
          optimizer.zero_grad()
          inputs, targets = batch
          outputs = model(inputs)
          loss = loss_function(outputs, targets)
          accelerator.backward(loss)
          optimizer.step()
          scheduler.step()

+ if __name__ == "__main__":
+     main()
```

Next you need to launch it with `accelerate launch`.

<Tip warning={true}>

  It's recommended you run `accelerate config` before using `accelerate launch` to configure your enviro

  Otherwise ■ Accelerate will use very basic defaults depending on your system setup.

</Tip>

## Using accelerate launch

■ Accelerate has a special CLI command to help you launch your code in your system through `accelerate l
This command wraps around all of the different commands needed to launch your script on various platform

<Tip>

  If you are familiar with launching scripts in PyTorch yourself such as with `torchrun`, you can still

</Tip>

You can launch your script quickly by using:

```bash
accelerate launch {script_name.py} --arg1 --arg2 ...
```

Just put `accelerate launch` at the start of your command, and pass in additional arguments and paramete

Since this runs the various torch spawn methods, all of the expected environment variables can be modifi
For example, here is how to use `accelerate launch` with a single GPU:

```bash
CUDA_VISIBLE_DEVICES="0" accelerate launch {script_name.py} --arg1 --arg2 ...
```

You can also use `accelerate launch` without performing `accelerate config` first, but you may need to m
In this case, ■ Accelerate will make some hyperparameter decisions for you, e.g., if GPUs are available,
Here is how you would use all GPUs and train with mixed precision disabled:

```bash
accelerate launch --multi_gpu {script_name.py} {--arg1} {--arg2} ...
```

To get more specific you should pass in the needed parameters yourself. For instance, here is how you
would also launch that same script on two GPUs using mixed precision while avoiding all of the warnings:

```bash
accelerate launch --multi_gpu --mixed_precision=fp16 --num_processes=2 {script_name.py} {--arg1} {--arg2
```

For a complete list of parameters you can pass in, run:

```bash
accelerate launch -h
```

<Tip>

  Even if you are not using ■ Accelerate in your code, you can still use the launcher for starting your

</Tip>

For a visualization of this difference, that earlier `accelerate launch` on multi-gpu would look somethi

```bash
MIXED_PRECISION="fp16" torchrun --nproc_per_node=2 --num_machines=1 {script_name.py} {--arg1} {--arg2} .
```

## Why you should always use `accelerate config`

Why is it useful to the point you should **always** run `accelerate config`?

Remember that earlier call to `accelerate launch` as well as `torchrun`?
Post configuration, to run that script with the needed parts you just need to use `accelerate launch` ou

```bash
```

accelerate launch {script_name.py} {--arg1} {--arg2} ...

## Custom Configurations

As briefly mentioned earlier, `accelerate launch` should be mostly used through combining set configurat
made with the `accelerate config` command. These configs are saved to a `default_config.yaml` file in yo
This cache folder is located at (with decreasing order of priority):

- The content of your environment variable `HF_HOME` suffixed with `accelerate`.
- If it does not exist, the content of your environment variable `XDG_CACHE_HOME` suffixed with
  `huggingface/accelerate`.
- If this does not exist either, the folder `~/.cache/huggingface/accelerate`.

To have multiple configurations, the flag `--config_file` can be passed to the `accelerate launch` comma
with the location of the custom yaml.

An example yaml may look something like the following for two GPUs on a single machine using `fp16` for
```yaml
compute_environment: LOCAL_MACHINE
deepspeed_config: {}
distributed_type: MULTI_GPU
fsdp_config: {}
machine_rank: 0
main_process_ip: null
main_process_port: null
main_training_function: main
mixed_precision: fp16
num_machines: 1
num_processes: 2
use_cpu: false
```

Launching a script from the location of that custom yaml file looks like the following:
```bash
accelerate launch --config_file {path/to/config/my_config_file.yaml} {script_name.py} {--arg1} {--arg2}
```

```
    def test_with_scheduler(self):
        with tempfile.TemporaryDirectory() as tmpdir:
            set_seed(42)
            model = DummyModel()
            optimizer = torch.optim.Adam(params=model.parameters(), lr=1e-3)
            scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=1, gamma=0.99)
            train_dataloader, valid_dataloader = dummy_dataloaders()
            project_config = ProjectConfiguration(automatic_checkpoint_naming=True)
            # Train baseline
            accelerator = Accelerator(project_dir=tmpdir, project_config=project_config)
            model, optimizer, train_dataloader, valid_dataloader, scheduler = accelerator.prepare(
                model, optimizer, train_dataloader, valid_dataloader, scheduler
            )
            # Save initial
            accelerator.save_state()
            scheduler_state = scheduler.state_dict()
            train(3, model, train_dataloader, optimizer, accelerator, scheduler)
            self.assertNotEqual(scheduler_state, scheduler.state_dict())

            # Load everything back in and make sure all states work
            accelerator.load_state(os.path.join(tmpdir, "checkpoints", "checkpoint_0"))
            self.assertEqual(scheduler_state, scheduler.state_dict())

    def test_checkpoint_deletion(self):
        with tempfile.TemporaryDirectory() as tmpdir:
            set_seed(42)
            model = DummyModel()
            project_config = ProjectConfiguration(automatic_checkpoint_naming=True, total_limit=2)
            # Train baseline
            accelerator = Accelerator(project_dir=tmpdir, project_config=project_config)
            model = accelerator.prepare(model)
            # Save 3 states:
            for _ in range(11):
                accelerator.save_state()
            self.assertTrue(not os.path.exists(os.path.join(tmpdir, "checkpoints", "checkpoint_0")))
            self.assertTrue(os.path.exists(os.path.join(tmpdir, "checkpoints", "checkpoint_9")))
            self.assertTrue(os.path.exists(os.path.join(tmpdir, "checkpoints", "checkpoint_10")))
```

accelerate-main/tests/test_cli.py

```python
# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import inspect
import os
import unittest
from pathlib import Path

import torch

import accelerate
from accelerate.test_utils import execute_subprocess_async
from accelerate.test_utils.testing import run_command


class AccelerateLauncherTester(unittest.TestCase):
    """
    Test case for verifying the `accelerate launch` CLI operates correctly.
    If a `default_config.yaml` file is located in the cache it will temporarily move it
    for the duration of the tests.
    """

    mod_file = inspect.getfile(accelerate.test_utils)
    test_file_path = os.path.sep.join(mod_file.split(os.path.sep)[:-1] + ["scripts", "test_cli.py"])

    base_cmd = ["accelerate", "launch"]
    config_folder = Path.home() / ".cache/huggingface/accelerate"
    config_file = "default_config.yaml"
    config_path = config_folder / config_file
    changed_path = config_folder / "_default_config.yaml"

    test_config_path = Path("tests/test_configs")

    @classmethod
    def setUpClass(cls):
        if cls.config_path.is_file():
            cls.config_path.rename(cls.changed_path)

    @classmethod
    def tearDownClass(cls):
        if cls.changed_path.is_file():
            cls.changed_path.rename(cls.config_path)

    def test_no_config(self):
        cmd = self.base_cmd
        if torch.cuda.is_available() and (torch.cuda.device_count() > 1):
            cmd += ["--multi_gpu"]
        execute_subprocess_async(cmd + [self.test_file_path], env=os.environ.copy())

    def test_config_compatibility(self):
        for config in sorted(self.test_config_path.glob("**/*.yaml")):
            with self.subTest(config_file=config):
                execute_subprocess_async(
                    self.base_cmd + ["--config_file", str(config), self.test_file_path], env=os.environ.
                )

    def test_accelerate_test(self):
        execute_subprocess_async(["accelerate", "test"], env=os.environ.copy())


class TpuConfigTester(unittest.TestCase):
    """
    Test case for verifying the `accelerate tpu-config` CLI passes the right `gcloud` command.
    """

    tpu_name = "test-tpu"
    tpu_zone = "us-central1-a"
    command = "ls"
    cmd = ["accelerate", "tpu-config"]
    base_output = "cd /usr/share"
    command_file = "tests/test_samples/test_command_file.sh"
    gcloud = "Running gcloud compute tpus tpu-vm ssh"

    @staticmethod
    def clean_output(output):
        return "".join(output).rstrip()

    def test_base(self):
        output = run_command(
            self.cmd
            + ["--command", self.command, "--tpu_zone", self.tpu_zone, "--tpu_name", self.tpu_name, "--d
            return_stdout=True,
        )
        self.assertEqual(
            self.clean_output(output),
            f"{self.gcloud} test-tpu --zone us-central1-a --command {self.base_output}; ls --worker all"
        )

    def test_base_backward_compatibility(self):
        output = run_command(
            self.cmd
            + [
                "--config_file",
                "tests/test_configs/0_12_0.yaml",
                "--command",
                self.command,
                "--tpu_zone",
                self.tpu_zone,
                "--tpu_name",
                self.tpu_name,
                "--debug",
            ],
            return_stdout=True,
        )
        self.assertEqual(
            self.clean_output(output),
            f"{self.gcloud} test-tpu --zone us-central1-a --command {self.base_output}; ls --worker all"
        )

    def test_with_config_file(self):
        output = run_command(
            self.cmd + ["--config_file", "tests/test_configs/latest.yaml", "--debug"], return_stdout=Tru
        )
        self.assertEqual(
            self.clean_output(output),
            f'{self.gcloud} test-tpu --zone us-central1-a --command {self.base_output}; echo "hello worl
        )

    def test_with_config_file_and_command(self):
        output = run_command(
            self.cmd + ["--config_file", "tests/test_configs/latest.yaml", "--command", self.command, "-
            return_stdout=True,
        )
        self.assertEqual(
            self.clean_output(output),
            f"{self.gcloud} test-tpu --zone us-central1-a --command {self.base_output}; ls --worker all"
        )

    def test_with_config_file_and_multiple_command(self):
        output = run_command(
            self.cmd
            + [
                "--config_file",
                "tests/test_configs/latest.yaml",
                "--command",
                self.command,
                "--command",
                'echo "Hello World"',
                "--debug",
            ],
            return_stdout=True,
        )
        self.assertEqual(
            self.clean_output(output),
            f'{self.gcloud} test-tpu --zone us-central1-a --command {self.base_output}; ls; echo "Hello
        )

    def test_with_config_file_and_command_file(self):
        output = run_command(
            self.cmd
            + ["--config_file", "tests/test_configs/latest.yaml", "--command_file", self.command_file, "
            return_stdout=True,
        )
        self.assertEqual(
            self.clean_output(output),
            f'{self.gcloud} test-tpu --zone us-central1-a --command {self.base_output}; echo "hello worl
        )

    def test_with_config_file_and_command_file_backward_compatibility(self):
        output = run_command(
            self.cmd
            + [
                "--config_file",
                "tests/test_configs/0_12_0.yaml",
                "--command_file",
                self.command_file,
                "--tpu_zone",
                self.tpu_zone,
                "--tpu_name",
                self.tpu_name,
                "--debug",
            ],
            return_stdout=True,
        )
        self.assertEqual(
            self.clean_output(output),
            f'{self.gcloud} test-tpu --zone us-central1-a --command {self.base_output}; echo "hello worl
        )

    def test_accelerate_install(self):
        output = run_command(
            self.cmd + ["--config_file", "tests/test_configs/latest.yaml", "--install_accelerate", "--de
            return_stdout=True,
        )
        self.assertEqual(
            self.clean_output(output),
            f'{self.gcloud} test-tpu --zone us-central1-a --command {self.base_output}; pip install acce
        )

    def test_accelerate_install_version(self):
        output = run_command(
            self.cmd
            + [
                "--config_file",
                "tests/test_configs/latest.yaml",
                "--install_accelerate",
                "--accelerate_version",
                "12.0.0",
                "--debug",
            ],
            return_stdout=True,
        )
        self.assertEqual(
            self.clean_output(output),
            f'{self.gcloud} test-tpu --zone us-central1-a --command {self.base_output}; pip install acce
        )
```

accelerate-main/tests/test_utils.py

```python
# Copyright 2021 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import os
import pickle
import unittest
from collections import UserDict, namedtuple

import torch

from accelerate.test_utils.testing import require_cuda
from accelerate.test_utils.training import RegressionModel
from accelerate.utils import (
    convert_outputs_to_fp32,
    extract_model_from_parallel,
    find_device,
    patch_environment,
    send_to_device,
)


ExampleNamedTuple = namedtuple("ExampleNamedTuple", "a b c")


class UtilsTester(unittest.TestCase):
    def test_send_to_device(self):
        tensor = torch.randn(5, 2)
        device = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")

        result1 = send_to_device(tensor, device)
        self.assertTrue(torch.equal(result1.cpu(), tensor))

        result2 = send_to_device((tensor, [tensor, tensor], 1), device)
        self.assertIsInstance(result2, tuple)
        self.assertTrue(torch.equal(result2[0].cpu(), tensor))
        self.assertIsInstance(result2[1], list)
        self.assertTrue(torch.equal(result2[1][0].cpu(), tensor))
        self.assertTrue(torch.equal(result2[1][1].cpu(), tensor))
        self.assertEqual(result2[2], 1)

        result2 = send_to_device({"a": tensor, "b": [tensor, tensor], "c": 1}, device)
        self.assertIsInstance(result2, dict)
        self.assertTrue(torch.equal(result2["a"].cpu(), tensor))
        self.assertIsInstance(result2["b"], list)
        self.assertTrue(torch.equal(result2["b"][0].cpu(), tensor))
        self.assertTrue(torch.equal(result2["b"][1].cpu(), tensor))
        self.assertEqual(result2["c"], 1)

        result3 = send_to_device(ExampleNamedTuple(a=tensor, b=[tensor, tensor], c=1), device)
        self.assertIsInstance(result3, ExampleNamedTuple)
        self.assertTrue(torch.equal(result3.a.cpu(), tensor))
        self.assertIsInstance(result3.b, list)
        self.assertTrue(torch.equal(result3.b[0].cpu(), tensor))
        self.assertTrue(torch.equal(result3.b[1].cpu(), tensor))
        self.assertEqual(result3.c, 1)

        result4 = send_to_device(UserDict({"a": tensor, "b": [tensor, tensor], "c": 1}), device)
        self.assertIsInstance(result4, UserDict)
        self.assertTrue(torch.equal(result4["a"].cpu(), tensor))
        self.assertIsInstance(result4["b"], list)
        self.assertTrue(torch.equal(result4["b"][0].cpu(), tensor))
        self.assertTrue(torch.equal(result4["b"][1].cpu(), tensor))
        self.assertEqual(result4["c"], 1)

    def test_patch_environment(self):
        with patch_environment(aa=1, BB=2):
            self.assertEqual(os.environ.get("AA"), "1")
            self.assertEqual(os.environ.get("BB"), "2")

        self.assertNotIn("AA", os.environ)
        self.assertNotIn("BB", os.environ)

    def test_can_undo_convert_outputs(self):
        model = RegressionModel()
        model._original_forward = model.forward
        model.forward = convert_outputs_to_fp32(model.forward)
        model = extract_model_from_parallel(model, keep_fp32_wrapper=False)
        _ = pickle.dumps(model)

    @require_cuda
    def test_can_undo_fp16_conversion(self):
        model = RegressionModel()
        model._original_forward = model.forward
        model.forward = torch.cuda.amp.autocast(dtype=torch.float16)(model.forward)
        model.forward = convert_outputs_to_fp32(model.forward)
        model = extract_model_from_parallel(model, keep_fp32_wrapper=False)
        _ = pickle.dumps(model)

    def test_find_device(self):
        self.assertEqual(find_device([1, "a", torch.tensor([1, 2, 3])]), torch.device("cpu"))
        self.assertEqual(find_device({"a": 1, "b": torch.tensor([1, 2, 3])}), torch.device("cpu"))
        self.assertIsNone(find_device([1, "a"]))
```

accelerate-main/tests/test_hooks.py

```
# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import inspect
import unittest

import torch
import torch.nn as nn

from accelerate.hooks import (
    AlignDevicesHook,
    ModelHook,
    SequentialHook,
    add_hook_to_module,
    attach_align_device_hook,
    remove_hook_from_module,
    remove_hook_from_submodules,
)
from accelerate.test_utils import require_multi_gpu, require_torch_min_version


class ModelForTest(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear1 = nn.Linear(3, 4)
        self.batchnorm = nn.BatchNorm1d(4)
        self.linear2 = nn.Linear(4, 5)

    def forward(self, x):
        return self.linear2(self.batchnorm(self.linear1(x)))


class PreForwardHook(ModelHook):
    def pre_forward(self, module, *args, **kwargs):
        return (args[0] + 1,) + args[1:], kwargs


class PostForwardHook(ModelHook):
    def post_forward(self, module, output):
        return output + 1


@require_torch_min_version(version="1.9.0")
class HooksModelTester(unittest.TestCase):
    def test_add_and_remove_hooks(self):
        test_model = ModelForTest()
        test_hook = ModelHook()

        add_hook_to_module(test_model, test_hook)
        self.assertEqual(test_model._hf_hook, test_hook)
        self.assertTrue(hasattr(test_model, "_old_forward"))

        # Check adding the hook did not change the name or the signature
        self.assertEqual(test_model.forward.__name__, "forward")
        self.assertListEqual(list(inspect.signature(test_model.forward).parameters), ["x"])

        remove_hook_from_module(test_model)
        self.assertFalse(hasattr(test_model, "_hf_hook"))
        self.assertFalse(hasattr(test_model, "_old_forward"))

    def test_append_and_remove_hooks(self):
        test_model = ModelForTest()
        test_hook = ModelHook()

        add_hook_to_module(test_model, test_hook)
        add_hook_to_module(test_model, test_hook, append=True)

        self.assertEqual(isinstance(test_model._hf_hook, SequentialHook), True)
        self.assertEqual(len(test_model._hf_hook.hooks), 2)
        self.assertTrue(hasattr(test_model, "_old_forward"))

        # Check adding the hook did not change the name or the signature
        self.assertEqual(test_model.forward.__name__, "forward")
        self.assertListEqual(list(inspect.signature(test_model.forward).parameters), ["x"])

        remove_hook_from_module(test_model)
        self.assertFalse(hasattr(test_model, "_hf_hook"))
        self.assertFalse(hasattr(test_model, "_old_forward"))

    def test_pre_forward_hook_is_executed(self):
        test_model = ModelForTest()
        x = torch.randn(2, 3)
        expected = test_model(x + 1)
        expected2 = test_model(x + 2)

        test_hook = PreForwardHook()
        add_hook_to_module(test_model, test_hook)
        output1 = test_model(x)
        self.assertTrue(torch.allclose(output1, expected, atol=1e-5))

        # Attaching a hook to a model when it already has one replaces, does not chain
        test_hook = PreForwardHook()
        add_hook_to_module(test_model, test_hook)
        output1 = test_model(x)
        self.assertTrue(torch.allclose(output1, expected, atol=1e-5))

        # You need to use the sequential hook to chain two or more hooks
        test_hook = SequentialHook(PreForwardHook(), PreForwardHook())
        add_hook_to_module(test_model, test_hook)

        output2 = test_model(x)
        assert torch.allclose(output2, expected2, atol=1e-5)

    def test_post_forward_hook_is_executed(self):
        test_model = ModelForTest()
        x = torch.randn(2, 3)
        output = test_model(x)

        test_hook = PostForwardHook()
        add_hook_to_module(test_model, test_hook)
        output1 = test_model(x)
        self.assertTrue(torch.allclose(output1, output + 1, atol=1e-5))

        # Attaching a hook to a model when it already has one replaces, does not chain
        test_hook = PostForwardHook()
        add_hook_to_module(test_model, test_hook)
        output1 = test_model(x)
        self.assertTrue(torch.allclose(output1, output + 1, atol=1e-5))

        # You need to use the sequential hook to chain two or more hooks
        test_hook = SequentialHook(PostForwardHook(), PostForwardHook())
        add_hook_to_module(test_model, test_hook)

        output2 = test_model(x)
        assert torch.allclose(output2, output + 2, atol=1e-5)

    def test_no_grad_in_hook(self):
        test_model = ModelForTest()
        x = torch.randn(2, 3)
        output = test_model(x)

        test_hook = PostForwardHook()
        add_hook_to_module(test_model, test_hook)
        output1 = test_model(x)
        self.assertTrue(torch.allclose(output1, output + 1))
        self.assertTrue(output1.requires_grad)

        test_hook.no_grad = True
        output1 = test_model(x)
        self.assertFalse(output1.requires_grad)

    @require_multi_gpu
    def test_align_devices_as_model_parallelism(self):
        model = ModelForTest()
        # Everything is on CPU
        self.assertEqual(model.linear1.weight.device, torch.device("cpu"))
        self.assertEqual(model.batchnorm.weight.device, torch.device("cpu"))
        self.assertEqual(model.linear2.weight.device, torch.device("cpu"))

        # This will move each submodule on different devices
        add_hook_to_module(model.linear1, AlignDevicesHook(execution_device=0))
        add_hook_to_module(model.batchnorm, AlignDevicesHook(execution_device=0))
        add_hook_to_module(model.linear2, AlignDevicesHook(execution_device=1))

        self.assertEqual(model.linear1.weight.device, torch.device(0))
        self.assertEqual(model.batchnorm.weight.device, torch.device(0))
        self.assertEqual(model.batchnorm.running_mean.device, torch.device(0))
        self.assertEqual(model.linear2.weight.device, torch.device(1))

        # We can still make a forward pass. The input does not need to be on any particular device
        x = torch.randn(2, 3)
        output = model(x)
        self.assertEqual(output.device, torch.device(1))

        # We can add a general hook to put back output on same device as input.
        add_hook_to_module(model, AlignDevicesHook(io_same_device=True))
        x = torch.randn(2, 3).to(0)
        output = model(x)
        self.assertEqual(output.device, torch.device(0))

    def test_align_devices_as_cpu_offload(self):
        model = ModelForTest()

        # Everything is on CPU
        self.assertEqual(model.linear1.weight.device, torch.device("cpu"))
        self.assertEqual(model.batchnorm.weight.device, torch.device("cpu"))
        self.assertEqual(model.linear2.weight.device, torch.device("cpu"))

        # This will move each submodule on different devices
        hook_kwargs = {"execution_device": 0 if torch.cuda.is_available() else "cpu", "offload": True}

        add_hook_to_module(model.linear1, AlignDevicesHook(**hook_kwargs))
        add_hook_to_module(model.batchnorm, AlignDevicesHook(**hook_kwargs))
        add_hook_to_module(model.linear2, AlignDevicesHook(**hook_kwargs))

        # Parameters have been offloaded, so on the meta device
        self.assertEqual(model.linear1.weight.device, torch.device("meta"))
        self.assertEqual(model.batchnorm.weight.device, torch.device("meta"))
        self.assertEqual(model.linear2.weight.device, torch.device("meta"))
        # Buffers are not included in the offload by default, so are on the execution device
        device = torch.device(hook_kwargs["execution_device"])
        self.assertEqual(model.batchnorm.running_mean.device, device)

        x = torch.randn(2, 3)
        output = model(x)
        self.assertEqual(output.device, device)

        # Removing hooks loads back the weights in the model.
        remove_hook_from_module(model.linear1)
        remove_hook_from_module(model.batchnorm)
        remove_hook_from_module(model.linear2)
        self.assertEqual(model.linear1.weight.device, torch.device("cpu"))
        self.assertEqual(model.batchnorm.weight.device, torch.device("cpu"))
        self.assertEqual(model.linear2.weight.device, torch.device("cpu"))

        # Now test with buffers included in the offload
        hook_kwargs = {
            "execution_device": 0 if torch.cuda.is_available() else "cpu",
            "offload": True,
            "offload_buffers": True,
        }

        add_hook_to_module(model.linear1, AlignDevicesHook(**hook_kwargs))
        add_hook_to_module(model.batchnorm, AlignDevicesHook(**hook_kwargs))
        add_hook_to_module(model.linear2, AlignDevicesHook(**hook_kwargs))

        # Parameters have been offloaded, so on the meta device, buffers included
        self.assertEqual(model.linear1.weight.device, torch.device("meta"))
        self.assertEqual(model.batchnorm.weight.device, torch.device("meta"))
        self.assertEqual(model.linear2.weight.device, torch.device("meta"))
        self.assertEqual(model.batchnorm.running_mean.device, torch.device("meta"))

        x = torch.randn(2, 3)
        output = model(x)
        self.assertEqual(output.device, device)

        # Removing hooks loads back the weights in the model.
        remove_hook_from_module(model.linear1)
        remove_hook_from_module(model.batchnorm)
        remove_hook_from_module(model.linear2)
        self.assertEqual(model.linear1.weight.device, torch.device("cpu"))
        self.assertEqual(model.batchnorm.weight.device, torch.device("cpu"))
        self.assertEqual(model.linear2.weight.device, torch.device("cpu"))

    def test_attach_align_device_hook_as_cpu_offload(self):
        model = ModelForTest()

        # Everything is on CPU
        self.assertEqual(model.linear1.weight.device, torch.device("cpu"))
        self.assertEqual(model.batchnorm.weight.device, torch.device("cpu"))
        self.assertEqual(model.linear2.weight.device, torch.device("cpu"))

        # This will move each submodule on different devices
        execution_device = 0 if torch.cuda.is_available() else "cpu"
        attach_align_device_hook(model, execution_device=execution_device, offload=True)

        # Parameters have been offloaded, so on the meta device
        self.assertEqual(model.linear1.weight.device, torch.device("meta"))
        self.assertEqual(model.batchnorm.weight.device, torch.device("meta"))
        self.assertEqual(model.linear2.weight.device, torch.device("meta"))
        # Buffers are not included in the offload by default, so are on the execution device
        device = torch.device(execution_device)
        self.assertEqual(model.batchnorm.running_mean.device, device)

        x = torch.randn(2, 3)
        output = model(x)
        self.assertEqual(output.device, device)

        # Removing hooks loads back the weights in the model.
        remove_hook_from_submodules(model)
        self.assertEqual(model.linear1.weight.device, torch.device("cpu"))
        self.assertEqual(model.batchnorm.weight.device, torch.device("cpu"))
        self.assertEqual(model.linear2.weight.device, torch.device("cpu"))

        # Now test with buffers included in the offload
        attach_align_device_hook(model, execution_device=execution_device, offload=True, offload_buffers

        # Parameters have been offloaded, so on the meta device, buffers included
        self.assertEqual(model.linear1.weight.device, torch.device("meta"))
        self.assertEqual(model.batchnorm.weight.device, torch.device("meta"))
        self.assertEqual(model.linear2.weight.device, torch.device("meta"))
        self.assertEqual(model.batchnorm.running_mean.device, torch.device("meta"))
        x = torch.randn(2, 3)
        output = model(x)
        self.assertEqual(output.device, device)

        # Removing hooks loads back the weights in the model.
        remove_hook_from_submodules(model)
        self.assertEqual(model.linear1.weight.device, torch.device("cpu"))
        self.assertEqual(model.batchnorm.weight.device, torch.device("cpu"))
        self.assertEqual(model.linear2.weight.device, torch.device("cpu"))

    def test_attach_align_device_hook_as_cpu_offload_with_weight_map(self):
        model = ModelForTest()

        # Everything is on CPU
        self.assertEqual(model.linear1.weight.device, torch.device("cpu"))
        self.assertEqual(model.batchnorm.weight.device, torch.device("cpu"))
        self.assertEqual(model.linear2.weight.device, torch.device("cpu"))

        # This will move each submodule on different devices
        execution_device = 0 if torch.cuda.is_available() else "cpu"
        attach_align_device_hook(
            model, execution_device=execution_device, offload=True, weights_map=model.state_dict()
        )

        # Parameters have been offloaded, so on the meta device
        self.assertEqual(model.linear1.weight.device, torch.device("meta"))
        self.assertEqual(model.batchnorm.weight.device, torch.device("meta"))
        self.assertEqual(model.linear2.weight.device, torch.device("meta"))
        # Buffers are not included in the offload by default, so are on the execution device
        device = torch.device(execution_device)
        self.assertEqual(model.batchnorm.running_mean.device, device)

        x = torch.randn(2, 3)
        output = model(x)
        self.assertEqual(output.device, device)

        # Removing hooks loads back the weights in the model.
        remove_hook_from_submodules(model)
        self.assertEqual(model.linear1.weight.device, torch.device("cpu"))
        self.assertEqual(model.batchnorm.weight.device, torch.device("cpu"))
        self.assertEqual(model.linear2.weight.device, torch.device("cpu"))

        # Now test with buffers included in the offload
        attach_align_device_hook(
            model,
            execution_device=execution_device,
            offload=True,
            weights_map=model.state_dict(),
            offload_buffers=True,
        )

        # Parameters have been offloaded, so on the meta device, buffers included
        self.assertEqual(model.linear1.weight.device, torch.device("meta"))
        self.assertEqual(model.batchnorm.weight.device, torch.device("meta"))
        self.assertEqual(model.linear2.weight.device, torch.device("meta"))
        self.assertEqual(model.batchnorm.running_mean.device, torch.device("meta"))

        x = torch.randn(2, 3)
        output = model(x)
        self.assertEqual(output.device, device)

        # Removing hooks loads back the weights in the model.
        remove_hook_from_submodules(model)
        self.assertEqual(model.linear1.weight.device, torch.device("cpu"))
        self.assertEqual(model.batchnorm.weight.device, torch.device("cpu"))
        self.assertEqual(model.linear2.weight.device, torch.device("cpu"))
```

accelerate-main/tests/test_cpu.py

```
# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import unittest

from accelerate import debug_launcher
from accelerate.test_utils import require_cpu, test_ops, test_script


@require_cpu
class MultiCPUTester(unittest.TestCase):
    def test_cpu(self):
        debug_launcher(test_script.main)

    def test_ops(self):
        debug_launcher(test_ops.main)
```

accelerate-main/tests/test_tracking.py

```
# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import csv
import json
import logging
import os
import re
import subprocess
import tempfile
import unittest
import zipfile
from pathlib import Path
from typing import Optional
from unittest import mock

# We use TF to parse the logs
from accelerate import Accelerator
from accelerate.test_utils.testing import (
    MockingTestCase,
    TempDirTestCase,
    require_comet_ml,
    require_tensorboard,
    require_wandb,
    skip,
)
from accelerate.tracking import CometMLTracker, GeneralTracker
from accelerate.utils import is_comet_ml_available


if is_comet_ml_available():
    from comet_ml import OfflineExperiment

logger = logging.getLogger(__name__)


@require_tensorboard
class TensorBoardTrackingTest(unittest.TestCase):
    def test_init_trackers(self):
        project_name = "test_project_with_config"
        with tempfile.TemporaryDirectory() as dirpath:
            accelerator = Accelerator(log_with="tensorboard", logging_dir=dirpath)
            config = {"num_iterations": 12, "learning_rate": 1e-2, "some_boolean": False, "some_string":
            accelerator.init_trackers(project_name, config)
            accelerator.end_training()
            for child in Path(f"{dirpath}/{project_name}").glob("*/**"):
                log = list(filter(lambda x: x.is_file(), child.iterdir()))[0]
            self.assertNotEqual(str(log), "")

    def test_log(self):
        project_name = "test_project_with_log"
        with tempfile.TemporaryDirectory() as dirpath:
            accelerator = Accelerator(log_with="tensorboard", project_dir=dirpath)
            accelerator.init_trackers(project_name)
            values = {"total_loss": 0.1, "iteration": 1, "my_text": "some_value"}
            accelerator.log(values, step=0)
            accelerator.end_training()
            # Logged values are stored in the outermost-tfevents file and can be read in as a TFRecord
            # Names are randomly generated each time
            log = list(filter(lambda x: x.is_file(), Path(f"{dirpath}/{project_name}").iterdir()))[0]
            self.assertNotEqual(str(log), "")

    def test_project_dir(self):
        with self.assertRaisesRegex(ValueError, "Logging with `tensorboard` requires a `logging_dir`"):
            _ = Accelerator(log_with="tensorboard")
        with tempfile.TemporaryDirectory() as dirpath:
            _ = Accelerator(log_with="tensorboard", project_dir=dirpath)
        with tempfile.TemporaryDirectory() as dirpath:
            _ = Accelerator(log_with="tensorboard", logging_dir=dirpath)


@require_wandb
@mock.patch.dict(os.environ, {"WANDB_MODE": "offline"})
class WandBTrackingTest(TempDirTestCase, MockingTestCase):
    def setUp(self):
        super().setUp()
        # wandb let's us override where logs are stored to via the WANDB_DIR env var
        self.add_mocks(mock.patch.dict(os.environ, {"WANDB_DIR": self.tmpdir}))

    @staticmethod
    def parse_log(log: str, section: str, record: bool = True):
        """
        Parses wandb log for `section` and returns a dictionary of
        all items in that section. Section names are based on the
        output of `wandb sync --view --verbose` and items starting
        with "Record" in that result
        """
        # Big thanks to the W&B team for helping us parse their logs
        pattern = rf"{section} ([\S\s]*?)\n\n"
        if record:
            pattern = rf"Record: {pattern}"
        cleaned_record = re.findall(pattern, log)[0]
        # A config
        if section == "config" or section == "history":
            cleaned_record = re.findall(r'"([a-zA-Z0-9_.,]+)', cleaned_record)
            return {key: val for key, val in zip(cleaned_record[0::2], cleaned_record[1::2])}
        # Everything else
        else:
            return dict(re.findall(r'(\w+): "([^\s]+)"', cleaned_record))

    @skip
    def test_wandb(self):
        project_name = "test_project_with_config"
        accelerator = Accelerator(log_with="wandb")
        config = {"num_iterations": 12, "learning_rate": 1e-2, "some_boolean": False, "some_string": "so
        kwargs = {"wandb": {"tags": ["my_tag"]}}
        accelerator.init_trackers(project_name, config, kwargs)
        values = {"total_loss": 0.1, "iteration": 1, "my_text": "some_value"}
        accelerator.log(values, step=0)
        accelerator.end_training()
        # The latest offline log is stored at wandb/latest-run/*.wandb
        for child in Path(f"{self.tmpdir}/wandb/latest-run").glob("*"):
            if child.is_file() and child.suffix == ".wandb":
                content = subprocess.check_output(
                    ["wandb", "sync", "--view", "--verbose", str(child)], env=os.environ.copy()
                ).decode("utf8", "ignore")
                break

        # Check HPS through careful parsing and cleaning
        logged_items = self.parse_log(content, "config")
        self.assertEqual(logged_items["num_iterations"], "12")
        self.assertEqual(logged_items["learning_rate"], "0.01")
        self.assertEqual(logged_items["some_boolean"], "false")
        self.assertEqual(logged_items["some_string"], "some_value")
        self.assertEqual(logged_items["some_string"], "some_value")

        # Run tags
        logged_items = self.parse_log(content, "run", False)
        self.assertEqual(logged_items["tags"], "my_tag")
        # Actual logging
        logged_items = self.parse_log(content, "history")
        self.assertEqual(logged_items["total_loss"], "0.1")
        self.assertEqual(logged_items["iteration"], "1")
        self.assertEqual(logged_items["my_text"], "some_value")
        self.assertEqual(logged_items["_step"], "0")


# Comet has a special `OfflineExperiment` we need to use for testing
def offline_init(self, run_name: str, tmpdir: str):
    self.run_name = run_name
    self.writer = OfflineExperiment(project_name=run_name, offline_directory=tmpdir)
    logger.info(f"Initialized offline CometML project {self.run_name}")
    logger.info("Make sure to log any initial configurations with `self.store_init_configuration` before


@require_comet_ml
@mock.patch.object(CometMLTracker, "__init__", offline_init)
class CometMLTest(unittest.TestCase):
    @staticmethod
    def get_value_from_key(log_list, key: str, is_param: bool = False):
        "Extracts `key` from Comet `log`"
        for log in log_list:
            j = json.loads(log)["payload"]
            if is_param and "param" in j.keys():
                if j["param"]["paramName"] == key:
                    return j["param"]["paramValue"]
            if "log_other" in j.keys():
                if j["log_other"]["key"] == key:
                    return j["log_other"]["val"]
            if "metric" in j.keys():
                if j["metric"]["metricName"] == key:
                    return j["metric"]["metricValue"]

    def test_init_trackers(self):
        with tempfile.TemporaryDirectory() as d:
            tracker = CometMLTracker("test_project_with_config", d)
            accelerator = Accelerator(log_with=tracker)
            config = {"num_iterations": 12, "learning_rate": 1e-2, "some_boolean": False, "some_string":
            accelerator.init_trackers(None, config)
            accelerator.end_training()
            log = os.listdir(d)[0]  # Comet is nice, it's just a zip file here
            # We parse the raw logs
            p = os.path.join(d, log)
            archive = zipfile.ZipFile(p, "r")
            log = archive.open("messages.json").read().decode("utf-8")
        list_of_json = log.split("\n")[:-1]
        self.assertEqual(self.get_value_from_key(list_of_json, "num_iterations", True), 12)
        self.assertEqual(self.get_value_from_key(list_of_json, "learning_rate", True), 0.01)
        self.assertEqual(self.get_value_from_key(list_of_json, "some_boolean", True), False)
        self.assertEqual(self.get_value_from_key(list_of_json, "some_string", True), "some_value")

    def test_log(self):
        with tempfile.TemporaryDirectory() as d:
            tracker = CometMLTracker("test_project_with_config", d)
            accelerator = Accelerator(log_with=tracker)
            accelerator.init_trackers(None)
            values = {"total_loss": 0.1, "iteration": 1, "my_text": "some_value"}
            accelerator.log(values, step=0)
            accelerator.end_training()
            log = os.listdir(d)[0]  # Comet is nice, it's just a zip file here
            # We parse the raw logs
            p = os.path.join(d, log)
            archive = zipfile.ZipFile(p, "r")
            log = archive.open("messages.json").read().decode("utf-8")
        list_of_json = log.split("\n")[:-1]
        self.assertEqual(self.get_value_from_key(list_of_json, "curr_step", True), 0)
        self.assertEqual(self.get_value_from_key(list_of_json, "total_loss"), 0.1)
        self.assertEqual(self.get_value_from_key(list_of_json, "iteration"), 1)
        self.assertEqual(self.get_value_from_key(list_of_json, "my_text"), "some_value")
class MyCustomTracker(GeneralTracker):
    "Basic tracker that writes to a csv for testing"
    _col_names = [
        "total_loss",
        "iteration",
        "my_text",
        "learning_rate",
        "num_iterations",
        "some_boolean",
        "some_string",
    ]

    name = "my_custom_tracker"
    requires_logging_directory = False

    def __init__(self, dir: str):
        self.f = open(f"{dir}/log.csv", "w+")
        self.writer = csv.DictWriter(self.f, fieldnames=self._col_names)
        self.writer.writeheader()

    @property
    def tracker(self):
        return self.writer

    def store_init_configuration(self, values: dict):
        logger.info("Call init")
        self.writer.writerow(values)

    def log(self, values: dict, step: Optional[int]):
        logger.info("Call log")
        self.writer.writerow(values)

    def finish(self):
        self.f.close()


class CustomTrackerTestCase(unittest.TestCase):
    def test_init_trackers(self):
        with tempfile.TemporaryDirectory() as d:
            tracker = MyCustomTracker(d)
            accelerator = Accelerator(log_with=tracker)
            config = {"num_iterations": 12, "learning_rate": 1e-2, "some_boolean": False, "some_string":
            accelerator.init_trackers("Some name", config)
            accelerator.end_training()
            with open(f"{d}/log.csv", "r") as f:
                data = csv.DictReader(f)
                data = next(data)
                truth = {
                    "total_loss": "",
                    "iteration": "",
                    "my_text": "",
                    "learning_rate": "0.01",
                    "num_iterations": "12",
                    "some_boolean": "False",
                    "some_string": "some_value",
                }
                self.assertDictEqual(data, truth)

    def test_log(self):
        with tempfile.TemporaryDirectory() as d:
            tracker = MyCustomTracker(d)
            accelerator = Accelerator(log_with=tracker)
            accelerator.init_trackers("Some name")
            values = {"total_loss": 0.1, "iteration": 1, "my_text": "some_value"}
            accelerator.log(values, step=0)
            accelerator.end_training()
            with open(f"{d}/log.csv", "r") as f:
                data = csv.DictReader(f)
                data = next(data)
                truth = {
                    "total_loss": "0.1",
                    "iteration": "1",
                    "my_text": "some_value",
                    "learning_rate": "",
                    "num_iterations": "",
                    "some_boolean": "",
                    "some_string": "",
                }
                self.assertDictEqual(data, truth)
```

accelerate-main/tests/test_metrics.py

```
# Copyright 2021 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import inspect
import os
import unittest

import torch

import accelerate
from accelerate import debug_launcher
from accelerate.test_utils import (
    execute_subprocess_async,
    require_cpu,
    require_huggingface_suite,
    require_multi_gpu,
    require_single_gpu,
    require_torch_min_version,
)
from accelerate.utils import get_launch_prefix, patch_environment


@require_huggingface_suite
@require_torch_min_version(version="1.8.0")
class MetricTester(unittest.TestCase):
    def setUp(self):
        mod_file = inspect.getfile(accelerate.test_utils)
        self.test_file_path = os.path.sep.join(
            mod_file.split(os.path.sep)[:-1] + ["scripts", "external_deps", "test_metrics.py"]
        )

        from accelerate.test_utils.scripts.external_deps import test_metrics  # noqa: F401

        self.test_metrics = test_metrics

    @require_cpu
    def test_metric_cpu_noop(self):
        debug_launcher(self.test_metrics.main, num_processes=1)

    @require_cpu
    def test_metric_cpu_multi(self):
        debug_launcher(self.test_metrics.main)

    @require_single_gpu
    def test_metric_gpu(self):
        self.test_metrics.main()

    @require_multi_gpu
    def test_metric_gpu_multi(self):
        print(f"Found {torch.cuda.device_count()} devices.")
        cmd = get_launch_prefix() + [f"--nproc_per_node={torch.cuda.device_count()}", self.test_file_pat
        with patch_environment(omp_num_threads=1):
            execute_subprocess_async(cmd, env=os.environ.copy())
```

accelerate-main/tests/test_optimizer.py

```
# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import pickle
import unittest

import torch

from accelerate import Accelerator
from accelerate.state import AcceleratorState
from accelerate.test_utils import require_cpu


@require_cpu
class OptimizerTester(unittest.TestCase):
    def test_accelerated_optimizer_pickling(self):
        model = torch.nn.Linear(10, 10)
        optimizer = torch.optim.SGD(model.parameters(), 0.1)
        accelerator = Accelerator()
        optimizer = accelerator.prepare(optimizer)
        try:
            pickle.loads(pickle.dumps(optimizer))
        except Exception as e:
            self.fail(f"Accelerated optimizer pickling failed with {e}")
        AcceleratorState._reset_state()
```

accelerate-main/tests/test_kwargs_handlers.py

```
# Copyright 2021 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import inspect
import os
import unittest
from dataclasses import dataclass

import torch

from accelerate import Accelerator, DistributedDataParallelKwargs, GradScalerKwargs
from accelerate.state import AcceleratorState
from accelerate.test_utils import execute_subprocess_async, require_cuda, require_multi_gpu
from accelerate.utils import KwargsHandler, get_launch_prefix


@dataclass
class MockClass(KwargsHandler):
    a: int = 0
    b: bool = False
    c: float = 3.0


class DataLoaderTester(unittest.TestCase):
    def test_kwargs_handler(self):
        # If no defaults are changed, `to_kwargs` returns an empty dict.
        self.assertDictEqual(MockClass().to_kwargs(), {})
        self.assertDictEqual(MockClass(a=2).to_kwargs(), {"a": 2})
        self.assertDictEqual(MockClass(a=2, b=True).to_kwargs(), {"a": 2, "b": True})
        self.assertDictEqual(MockClass(a=2, c=2.25).to_kwargs(), {"a": 2, "c": 2.25})

    @require_cuda
    def test_grad_scaler_kwargs(self):
        # If no defaults are changed, `to_kwargs` returns an empty dict.
        scaler_handler = GradScalerKwargs(init_scale=1024, growth_factor=2)
        AcceleratorState._reset_state()
        accelerator = Accelerator(mixed_precision="fp16", kwargs_handlers=[scaler_handler])
        print(accelerator.use_fp16)
        scaler = accelerator.scaler

        # Check the kwargs have been applied
        self.assertEqual(scaler._init_scale, 1024.0)
        self.assertEqual(scaler._growth_factor, 2.0)

        # Check the other values are at the default
        self.assertEqual(scaler._backoff_factor, 0.5)
        self.assertEqual(scaler._growth_interval, 2000)
        self.assertEqual(scaler._enabled, True)

    @require_multi_gpu
    def test_ddp_kwargs(self):
        cmd = get_launch_prefix()
        cmd += [f"--nproc_per_node={torch.cuda.device_count()}", inspect.getfile(self.__class__)]
        execute_subprocess_async(cmd, env=os.environ.copy())


if __name__ == "__main__":
```

```
ddp_scaler = DistributedDataParallelKwargs(bucket_cap_mb=15, find_unused_parameters=True)
accelerator = Accelerator(kwargs_handlers=[ddp_scaler])
model = torch.nn.Linear(100, 200)
model = accelerator.prepare(model)

# Check the values changed in kwargs
error_msg = ""
observed_bucket_cap_map = model.bucket_bytes_cap // (1024 * 1024)
if observed_bucket_cap_map != 15:
    error_msg += f"Kwargs badly passed, should have `15` but found {observed_bucket_cap_map}.\n"
if model.find_unused_parameters is not True:
    error_msg += f"Kwargs badly passed, should have `True` but found {model.find_unused_parameters}.

# Check the values of the defaults
if model.dim != 0:
    error_msg += f"Default value not respected, should have `0` but found {model.dim}.\n"
if model.broadcast_buffers is not True:
    error_msg += f"Default value not respected, should have `True` but found {model.broadcast_buffer
if model.gradient_as_bucket_view is not False:
    error_msg += f"Default value not respected, should have `False` but found {model.gradient_as_buc

# Raise error at the end to make sure we don't stop at the first failure.
if len(error_msg) > 0:
    raise ValueError(error_msg)
```

accelerate-main/tests/test_samples/test_command_file.sh

```
echo "hello world"
echo "this is a second command"
```

accelerate-main/tests/test_samples/MRPC/dev.csv

```
label,sentence1,sentence2
equivalent,He said the foodservice pie business doesn 't fit the company 's long-term growth strategy .,
not_equivalent,Magnarelli said Racicot hated the Iraqi regime and looked forward to using his long years
not_equivalent,"The dollar was at 116.92 yen against the yen , flat on the session , and at 1.2891 agair
equivalent,The AFL-CIO is waiting until October to decide if it will endorse a candidate .,The AFL-CIO a
not_equivalent,No dates have been set for the civil or the criminal trial .,"No dates have been set for
equivalent,Wal-Mart said it would check all of its million-plus domestic workers to ensure they were leg
```

accelerate-main/tests/test_samples/MRPC/train.csv

```
label,sentence1,sentence2
equivalent,He said the foodservice pie business doesn 't fit the company 's long-term growth strategy .,
not_equivalent,Magnarelli said Racicot hated the Iraqi regime and looked forward to using his long years
not_equivalent,"The dollar was at 116.92 yen against the yen , flat on the session , and at 1.2891 agair
equivalent,The AFL-CIO is waiting until October to decide if it will endorse a candidate .,The AFL-CIO a
not_equivalent,No dates have been set for the civil or the criminal trial .,"No dates have been set for
equivalent,Wal-Mart said it would check all of its million-plus domestic workers to ensure they were leg
```

accelerate-main/tests/test_configs/latest.yaml

```yaml
compute_environment: LOCAL_MACHINE
deepspeed_config: {}
distributed_type: 'NO'
downcast_bf16: 'no'
fsdp_config: {}
gpu_ids: all
machine_rank: 0
main_process_ip: null
main_process_port: null
main_training_function: main
megatron_lm_config: {}
mixed_precision: 'no'
num_machines: 1
num_processes: 1
rdzv_backend: static
same_network: true
use_cpu: false
tpu_name: 'test-tpu'
tpu_zone: 'us-central1-a'
commands: null
command_file: tests/test_samples/test_command_file.sh
```

accelerate-main/tests/test_configs/0_12_0.yaml

```
compute_environment: LOCAL_MACHINE
deepspeed_config: {}
distributed_type: 'NO'
downcast_bf16: 'no'
fsdp_config: {}
machine_rank: 0
main_process_ip: null
main_process_port: null
main_training_function: main
mixed_precision: 'no'
num_machines: 1
num_processes: 1
use_cpu: false
```

accelerate-main/tests/test_configs/0_11_0.yaml

```yaml
compute_environment: LOCAL_MACHINE
deepspeed_config: {}
distributed_type: 'NO'
fsdp_config: {}
machine_rank: 0
main_process_ip: null
main_process_port: null
main_training_function: main
mixed_precision: 'no'
num_machines: 1
num_processes: 1
use_cpu: false
```

accelerate-main/examples/by_feature/README.md

~~~
# What are these scripts?

All scripts in this folder originate from the `nlp_example.py` file, as it is a very simplistic NLP trai

From there, each further script adds in just **one** feature of Accelerate, showing how you can quickly

A full example with all of these parts integrated together can be found in the `complete_nlp_example.py`

Adjustments to each script from the base `nlp_example.py` file can be found quickly by searching for "#

## Example Scripts by Feature and their Arguments

### Base Example (`../nlp_example.py`)

- Shows how to use `Accelerator` in an extremely simplistic PyTorch training loop
- Arguments available:
  - `mixed_precision`, whether to use mixed precision. ("no", "fp16", or "bf16")
  - `cpu`, whether to train using only the CPU. (yes/no/1/0)

All following scripts also accept these arguments in addition to their added ones.

These arguments should be added at the end of any method for starting the python script (such as `python

```bash
accelerate launch ../nlp_example.py --mixed_precision fp16 --cpu 0
~~~

### Checkpointing and Resuming Training (`checkpointing.py`)

- Shows how to use `Accelerator.save_state` and `Accelerator.load_state` to save or continue training
- **It is assumed you are continuing off the same training script**
- Arguments available:
  - `checkpointing_steps`, after how many steps the various states should be saved. ("epoch", 1, 2, ...)
  - `output_dir`, where saved state folders should be saved to, default is current working directory
  - `resume_from_checkpoint`, what checkpoint folder to resume from. ("epoch_0", "step_22", ...)

These arguments should be added at the end of any method for starting the python script (such as `python

(Note, `resume_from_checkpoint` assumes that we've ran the script for one epoch with the `--checkpointin

```bash
accelerate launch ./checkpointing.py --checkpointing_steps epoch output_dir "checkpointing_tutorial" --r
```

### Cross Validation (`cross_validation.py`)

- Shows how to use `Accelerator.free_memory` and run cross validation efficiently with `datasets`.
- Arguments available:
  - `num_folds`, the number of folds the training dataset should be split into.

These arguments should be added at the end of any method for starting the python script (such as `python

```bash
accelerate launch ./cross_validation.py --num_folds 2
```

### Experiment Tracking (`tracking.py`)

- Shows how to use `Accelerate.init_trackers` and `Accelerator.log`
- Can be used with Weights and Biases, TensorBoard, or CometML.
- Arguments available:
  - `with_tracking`, whether to load in all available experiment trackers from the environment.

These arguments should be added at the end of any method for starting the python script (such as `python

```bash
accelerate launch ./tracking.py --with_tracking
```
```
```

### Gradient Accumulation (`gradient_accumulation.py`)

- Shows how to use `Accelerator.no_sync` to prevent gradient averaging in a distributed setup.
- Arguments available:
  - `gradient_accumulation_steps`, the number of steps to perform before the gradients are accumulated a

These arguments should be added at the end of any method for starting the python script (such as `python

```bash
accelerate launch ./gradient_accumulation.py --gradient_accumulation_steps 5
```

accelerate-main/tests/deepspeed/test_deepspeed.py

```
# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import inspect
import io
import itertools
import json
import os
import tempfile
from copy import deepcopy
from pathlib import Path

import torch
from parameterized import parameterized
from torch.utils.data import DataLoader
from transformers import AutoModel, AutoModelForCausalLM, get_scheduler
from transformers.testing_utils import mockenv_context
from transformers.trainer_utils import set_seed
from transformers.utils import is_torch_bf16_available

import accelerate
from accelerate.accelerator import Accelerator
from accelerate.scheduler import AcceleratedScheduler
from accelerate.state import AcceleratorState
from accelerate.test_utils.testing import (
    AccelerateTestCase,
    TempDirTestCase,
    execute_subprocess_async,
    require_cuda,
    require_deepspeed,
    require_multi_gpu,
    slow,
)
from accelerate.test_utils.training import RegressionDataset
from accelerate.utils.dataclasses import DeepSpeedPlugin
from accelerate.utils.deepspeed import (
    DeepSpeedEngineWrapper,
    DeepSpeedOptimizerWrapper,
    DeepSpeedSchedulerWrapper,
    DummyOptim,
    DummyScheduler,
)
from accelerate.utils.other import patch_environment


set_seed(42)

T5_SMALL = "t5-small"
T5_TINY = "patrickvonplaten/t5-tiny-random"
GPT2_TINY = "sshleifer/tiny-gpt2"

ZERO2 = "zero2"
ZERO3 = "zero3"

FP16 = "fp16"
BF16 = "bf16"
CUSTOM_OPTIMIZER = "custom_optimizer"
```

```
CUSTOM_SCHEDULER = "custom_scheduler"
DS_OPTIMIZER = "deepspeed_optimizer"
DS_SCHEDULER = "deepspeed_scheduler"

stages = [ZERO2, ZERO3]
optims = [CUSTOM_OPTIMIZER, DS_OPTIMIZER]
schedulers = [CUSTOM_SCHEDULER, DS_SCHEDULER]
if is_torch_bf16_available():
    dtypes = [FP16, BF16]
else:
    dtypes = [FP16]


def parameterized_custom_name_func(func, param_num, param):
    # customize the test name generator function as we want both params to appear in the sub-test
    # name, as by default it shows only the first param
    param_based_name = parameterized.to_safe_name("_".join(str(x) for x in param.args))
    return f"{func.__name__}_{param_based_name}"


# Cartesian-product of zero stages with models to test
params = list(itertools.product(stages, dtypes))
optim_scheduler_params = list(itertools.product(optims, schedulers))


@require_deepspeed
@require_cuda
class DeepSpeedConfigIntegration(AccelerateTestCase):
    def setUp(self):
        super().setUp()

        self._test_file_path = inspect.getfile(self.__class__)
        path = Path(self._test_file_path).resolve()
        self.test_file_dir_str = str(path.parents[0])

        self.ds_config_file = dict(
            zero2=f"{self.test_file_dir_str}/ds_config_zero2.json",
            zero3=f"{self.test_file_dir_str}/ds_config_zero3.json",
        )

        # use self.get_config_dict(stage) to use these to ensure the original is not modified
        with io.open(self.ds_config_file[ZERO2], "r", encoding="utf-8") as f:
            config_zero2 = json.load(f)
        with io.open(self.ds_config_file[ZERO3], "r", encoding="utf-8") as f:
            config_zero3 = json.load(f)
            # The following setting slows things down, so don't enable it by default unless needed by a
            # It's in the file as a demo for users since we want everything to work out of the box even
            config_zero3["zero_optimization"]["stage3_gather_16bit_weights_on_model_save"] = False

        self.ds_config_dict = dict(zero2=config_zero2, zero3=config_zero3)

        self.dist_env = dict(
            ACCELERATE_USE_DEEPSPEED="true",
            MASTER_ADDR="localhost",
            MASTER_PORT="10999",
            RANK="0",
            LOCAL_RANK="0",
            WORLD_SIZE="1",
        )

    def get_config_dict(self, stage):
        # As some tests modify the dict, always make a copy
        return deepcopy(self.ds_config_dict[stage])

    @parameterized.expand(stages, name_func=parameterized_custom_name_func)
    def test_deepspeed_plugin(self, stage):
        # Test zero3_init_flag will be set to False when ZeRO stage != 3
        deepspeed_plugin = DeepSpeedPlugin(
            gradient_accumulation_steps=1,
            gradient_clipping=1.0,
            zero_stage=2,
            offload_optimizer_device="cpu",
            offload_param_device="cpu",
            zero3_save_16bit_model=True,
            zero3_init_flag=True,
        )
        self.assertFalse(deepspeed_plugin.zero3_init_flag)
        deepspeed_plugin.deepspeed_config = None

        # Test zero3_init_flag will be set to True only when ZeRO stage == 3
        deepspeed_plugin = DeepSpeedPlugin(
            gradient_accumulation_steps=1,
            gradient_clipping=1.0,
            zero_stage=3,
            offload_optimizer_device="cpu",
            offload_param_device="cpu",
            zero3_save_16bit_model=True,
            zero3_init_flag=True,
        )
        self.assertTrue(deepspeed_plugin.zero3_init_flag)
        deepspeed_plugin.deepspeed_config = None

        # Test config files are loaded correctly
        deepspeed_plugin = DeepSpeedPlugin(hf_ds_config=self.ds_config_file[stage], zero3_init_flag=True
        if stage == ZERO2:
            self.assertFalse(deepspeed_plugin.zero3_init_flag)
        elif stage == ZERO3:
            self.assertTrue(deepspeed_plugin.zero3_init_flag)

        # Test `gradient_accumulation_steps` is set to 1 if unavailable in config file
        with tempfile.TemporaryDirectory() as dirpath:
            ds_config = self.get_config_dict(stage)
            del ds_config["gradient_accumulation_steps"]
            with open(os.path.join(dirpath, "ds_config.json"), "w") as out_file:
                json.dump(ds_config, out_file)
            deepspeed_plugin = DeepSpeedPlugin(hf_ds_config=os.path.join(dirpath, "ds_config.json"))
            self.assertEqual(deepspeed_plugin.deepspeed_config["gradient_accumulation_steps"], 1)
            deepspeed_plugin.deepspeed_config = None

        # Test `ValueError` is raised if `zero_optimization` is unavailable in config file
        with tempfile.TemporaryDirectory() as dirpath:
            ds_config = self.get_config_dict(stage)
            del ds_config["zero_optimization"]
            with open(os.path.join(dirpath, "ds_config.json"), "w") as out_file:
                json.dump(ds_config, out_file)
            with self.assertRaises(ValueError) as cm:
                deepspeed_plugin = DeepSpeedPlugin(hf_ds_config=os.path.join(dirpath, "ds_config.json"))
            self.assertTrue(
                "Please specify the ZeRO optimization config in the DeepSpeed config." in str(cm.excepti
            )
            deepspeed_plugin.deepspeed_config = None

        # Test `deepspeed_config_process`
        deepspeed_plugin = DeepSpeedPlugin(hf_ds_config=self.ds_config_file[stage])
        kwargs = {
            "fp16.enabled": True,
            "bf16.enabled": False,
            "optimizer.params.lr": 5e-5,
            "optimizer.params.weight_decay": 0.0,
            "scheduler.params.warmup_min_lr": 0.0,
            "scheduler.params.warmup_max_lr": 5e-5,
            "scheduler.params.warmup_num_steps": 0,
            "train_micro_batch_size_per_gpu": 16,
            "gradient_clipping": 1.0,
            "train_batch_size": 16,
            "zero_optimization.reduce_bucket_size": 5e5,
            "zero_optimization.stage3_prefetch_bucket_size": 5e5,
            "zero_optimization.stage3_param_persistence_threshold": 5e5,
            "zero_optimization.stage3_gather_16bit_weights_on_model_save": False,
        }
        deepspeed_plugin.deepspeed_config_process(**kwargs)
        for ds_key_long, value in kwargs.items():
            config, ds_key = deepspeed_plugin.hf_ds_config.find_config_node(ds_key_long)
            if config.get(ds_key) is not None:
                self.assertEqual(config.get(ds_key), value)

        # Test mismatches
        mismatches = {
            "optimizer.params.lr": 1e-5,
            "optimizer.params.weight_decay": 1e-5,
            "gradient_accumulation_steps": 2,
        }
        with self.assertRaises(ValueError) as cm:
            new_kwargs = deepcopy(kwargs)
            new_kwargs.update(mismatches)
            deepspeed_plugin.deepspeed_config_process(**new_kwargs)
        for key in mismatches.keys():
            self.assertTrue(
                key in str(cm.exception),
                f"{key} is not in the exception message:\n{cm.exception}",
            )

        # Test `ValueError` is raised if some config file fields with `auto` value is missing in `kwargs
        deepspeed_plugin.deepspeed_config["optimizer"]["params"]["lr"] = "auto"
        with self.assertRaises(ValueError) as cm:
            del kwargs["optimizer.params.lr"]
            deepspeed_plugin.deepspeed_config_process(**kwargs)
        self.assertTrue("`optimizer.params.lr` not found in kwargs." in str(cm.exception))

    @parameterized.expand([FP16, BF16], name_func=parameterized_custom_name_func)
    def test_accelerate_state_deepspeed(self, dtype):
        AcceleratorState._reset_state(True)
        deepspeed_plugin = DeepSpeedPlugin(
            gradient_accumulation_steps=1,
            gradient_clipping=1.0,
            zero_stage=ZERO2,
            offload_optimizer_device="cpu",
            offload_param_device="cpu",
            zero3_save_16bit_model=True,
            zero3_init_flag=True,
        )
        with mockenv_context(**self.dist_env):
            state = Accelerator(mixed_precision=dtype, deepspeed_plugin=deepspeed_plugin).state
            self.assertTrue(state.deepspeed_plugin.deepspeed_config[dtype]["enabled"])

    def test_init_zero3(self):
        deepspeed_plugin = DeepSpeedPlugin(
            gradient_accumulation_steps=1,
            gradient_clipping=1.0,
            zero_stage=3,
            offload_optimizer_device="cpu",
            offload_param_device="cpu",
            zero3_save_16bit_model=True,
            zero3_init_flag=True,
        )

        with mockenv_context(**self.dist_env):
            accelerator = Accelerator(deepspeed_plugin=deepspeed_plugin)  # noqa: F841
            from transformers.deepspeed import is_deepspeed_zero3_enabled

            self.assertTrue(is_deepspeed_zero3_enabled())

    @parameterized.expand(optim_scheduler_params, name_func=parameterized_custom_name_func)
    def test_prepare_deepspeed(self, optim_type, scheduler_type):
        # 1. Testing with one of the ZeRO Stages is enough to test the `_prepare_deepspeed` function.
        # Here we test using ZeRO Stage 2 with FP16 enabled.
        from deepspeed.runtime.engine import DeepSpeedEngine

        kwargs = {
            "optimizer.params.lr": 5e-5,
            "optimizer.params.weight_decay": 0.0,
            "scheduler.params.warmup_min_lr": 0.0,
            "scheduler.params.warmup_max_lr": 5e-5,
            "scheduler.params.warmup_num_steps": 0,
            "train_micro_batch_size_per_gpu": 16,
            "gradient_clipping": 1.0,
            "train_batch_size": 16,
            "zero_optimization.reduce_bucket_size": 5e5,
            "zero_optimization.stage3_prefetch_bucket_size": 5e5,
            "zero_optimization.stage3_param_persistence_threshold": 5e5,
            "zero_optimization.stage3_gather_16bit_weights_on_model_save": False,
        }

        if optim_type == CUSTOM_OPTIMIZER and scheduler_type == CUSTOM_SCHEDULER:
            # Test custom optimizer + custom scheduler
            deepspeed_plugin = DeepSpeedPlugin(
                gradient_accumulation_steps=1,
                gradient_clipping=1.0,
                zero_stage=2,
                offload_optimizer_device="cpu",
                offload_param_device="cpu",
                zero3_save_16bit_model=False,
                zero3_init_flag=False,
            )
            with mockenv_context(**self.dist_env):
                accelerator = Accelerator(mixed_precision="fp16", deepspeed_plugin=deepspeed_plugin)

                train_set = RegressionDataset(length=80)
                eval_set = RegressionDataset(length=20)
                train_dataloader = DataLoader(train_set, batch_size=16, shuffle=True)
                eval_dataloader = DataLoader(eval_set, batch_size=32, shuffle=False)
                model = AutoModel.from_pretrained(GPT2_TINY)
                optimizer = torch.optim.AdamW(model.parameters(), lr=5e-5)
                lr_scheduler = get_scheduler(
                    name="linear",
                    optimizer=optimizer,
                    num_warmup_steps=0,
                    num_training_steps=1000,
                )
                dummy_optimizer = DummyOptim(params=model.parameters())
                dummy_lr_scheduler = DummyScheduler(dummy_optimizer)

                with self.assertRaises(ValueError) as cm:
                    model, optimizer, train_dataloader, eval_dataloader, lr_scheduler = accelerator.prep
                        model, dummy_optimizer, train_dataloader, eval_dataloader, lr_scheduler
                    )
                self.assertTrue(
                    "You cannot create a `DummyOptim` without specifying an optimizer in the config file
                    in str(cm.exception)
                )
                with self.assertRaises(ValueError) as cm:
                    model, optimizer, train_dataloader, eval_dataloader, lr_scheduler = accelerator.prep
                        model, optimizer, train_dataloader, eval_dataloader, dummy_lr_scheduler
                    )
                self.assertTrue(
                    "You cannot create a `DummyScheduler` without specifying a scheduler in the config f
                    in str(cm.exception)
                )

                with self.assertRaises(ValueError) as cm:
                    model, optimizer, lr_scheduler = accelerator.prepare(model, optimizer, lr_scheduler)
                self.assertTrue(
                    "When using DeepSpeed `accelerate.prepare()` requires you to pass at least one of tr
                    "or alternatively set an integer value in `train_micro_batch_size_per_gpu` in the de
                    "or assign integer value to `AcceleratorState().deepspeed_plugin.deepspeed_config['t
                    in str(cm.exception)
                )

                model, optimizer, train_dataloader, eval_dataloader, lr_scheduler = accelerator.prepare(
                    model, optimizer, train_dataloader, eval_dataloader, lr_scheduler
                )
                self.assertTrue(accelerator.deepspeed_config["zero_allow_untested_optimizer"])
                self.assertTrue(accelerator.deepspeed_config["train_batch_size"], 16)
                self.assertEqual(type(model), DeepSpeedEngine)
                self.assertEqual(type(optimizer), DeepSpeedOptimizerWrapper)
                self.assertEqual(type(lr_scheduler), AcceleratedScheduler)
                self.assertEqual(type(accelerator.deepspeed_engine_wrapped), DeepSpeedEngineWrapper)
        elif optim_type == DS_OPTIMIZER and scheduler_type == DS_SCHEDULER:
            # Test DeepSpeed optimizer + DeepSpeed scheduler
            deepspeed_plugin = DeepSpeedPlugin(hf_ds_config=self.ds_config_file[ZERO2])
            with mockenv_context(**self.dist_env):
                accelerator = Accelerator(deepspeed_plugin=deepspeed_plugin, mixed_precision="fp16")
                train_set = RegressionDataset(length=80)
                eval_set = RegressionDataset(length=20)
                train_dataloader = DataLoader(train_set, batch_size=10, shuffle=True)
                eval_dataloader = DataLoader(eval_set, batch_size=5, shuffle=False)
                model = AutoModel.from_pretrained(GPT2_TINY)
                optimizer = torch.optim.AdamW(model.parameters(), lr=5e-5)
                lr_scheduler = get_scheduler(
                    name="linear",
                    optimizer=optimizer,
                    num_warmup_steps=0,
                    num_training_steps=1000,
                )
                dummy_optimizer = DummyOptim(params=model.parameters())
                dummy_lr_scheduler = DummyScheduler(dummy_optimizer)
                kwargs["train_batch_size"] = (
                    kwargs["train_micro_batch_size_per_gpu"]
                    * deepspeed_plugin.deepspeed_config["gradient_accumulation_steps"]
                    * accelerator.num_processes
                )
                accelerator.state.deepspeed_plugin.deepspeed_config_process(**kwargs)
                with self.assertRaises(ValueError) as cm:
                    model, optimizer, train_dataloader, eval_dataloader, lr_scheduler = accelerator.prep
                        model, optimizer, train_dataloader, eval_dataloader, dummy_lr_scheduler
                    )
                self.assertTrue(
                    "You cannot specify an optimizer in the config file and in the code at the same time
                    in str(cm.exception)
                )

                with self.assertRaises(ValueError) as cm:
                    model, optimizer, train_dataloader, eval_dataloader, lr_scheduler = accelerator.prep
                        model, dummy_optimizer, train_dataloader, eval_dataloader, lr_scheduler
                    )
                self.assertTrue(
                    "You cannot specify a scheduler in the config file and in the code at the same time"
                    in str(cm.exception)
                )

                with self.assertRaises(ValueError) as cm:
                    model, optimizer, train_dataloader, eval_dataloader, lr_scheduler = accelerator.prep
                        model, dummy_optimizer, train_dataloader, eval_dataloader, lr_scheduler
                    )
                self.assertTrue(
                    "You cannot specify a scheduler in the config file and in the code at the same time"
                    in str(cm.exception)
                )

                model, optimizer, train_dataloader, eval_dataloader, lr_scheduler = accelerator.prepare(
                    model, dummy_optimizer, train_dataloader, eval_dataloader, dummy_lr_scheduler
                )
                self.assertTrue(type(model) == DeepSpeedEngine)
                self.assertTrue(type(optimizer) == DeepSpeedOptimizerWrapper)
                self.assertTrue(type(lr_scheduler) == DeepSpeedSchedulerWrapper)
                self.assertTrue(type(accelerator.deepspeed_engine_wrapped) == DeepSpeedEngineWrapper)

        elif optim_type == CUSTOM_OPTIMIZER and scheduler_type == DS_SCHEDULER:
            # Test custom optimizer + DeepSpeed scheduler
            deepspeed_plugin = DeepSpeedPlugin(hf_ds_config=self.ds_config_file[ZERO2])
            with mockenv_context(**self.dist_env):
                accelerator = Accelerator(deepspeed_plugin=deepspeed_plugin, mixed_precision="fp16")
                train_set = RegressionDataset(length=80)
                eval_set = RegressionDataset(length=20)
                train_dataloader = DataLoader(train_set, batch_size=10, shuffle=True)
                eval_dataloader = DataLoader(eval_set, batch_size=5, shuffle=False)
                model = AutoModel.from_pretrained(GPT2_TINY)
                optimizer = torch.optim.AdamW(model.parameters(), lr=5e-5)
                lr_scheduler = get_scheduler(
                    name="linear",
                    optimizer=optimizer,
                    num_warmup_steps=0,
                    num_training_steps=1000,
                )
                dummy_optimizer = DummyOptim(params=model.parameters())
                dummy_lr_scheduler = DummyScheduler(dummy_optimizer)
                kwargs["train_batch_size"] = (
                    kwargs["train_micro_batch_size_per_gpu"]
                    * deepspeed_plugin.deepspeed_config["gradient_accumulation_steps"]
                    * accelerator.num_processes
                )
                accelerator.state.deepspeed_plugin.deepspeed_config_process(**kwargs)
                del accelerator.state.deepspeed_plugin.deepspeed_config["optimizer"]
                model, optimizer, train_dataloader, eval_dataloader, lr_scheduler = accelerator.prepare(
                    model, optimizer, train_dataloader, eval_dataloader, dummy_lr_scheduler
                )
                self.assertTrue(type(model) == DeepSpeedEngine)
                self.assertTrue(type(optimizer) == DeepSpeedOptimizerWrapper)
                self.assertTrue(type(lr_scheduler) == DeepSpeedSchedulerWrapper)
                self.assertTrue(type(accelerator.deepspeed_engine_wrapped) == DeepSpeedEngineWrapper)
        elif optim_type == DS_OPTIMIZER and scheduler_type == CUSTOM_SCHEDULER:
            # Test deepspeed optimizer + custom scheduler
            deepspeed_plugin = DeepSpeedPlugin(hf_ds_config=self.ds_config_file[ZERO2])
            with mockenv_context(**self.dist_env):
                accelerator = Accelerator(deepspeed_plugin=deepspeed_plugin, mixed_precision="fp16")
                train_set = RegressionDataset(length=80)
                eval_set = RegressionDataset(length=20)
                train_dataloader = DataLoader(train_set, batch_size=10, shuffle=True)
                eval_dataloader = DataLoader(eval_set, batch_size=5, shuffle=False)
                model = AutoModel.from_pretrained(GPT2_TINY)
                optimizer = torch.optim.AdamW(model.parameters(), lr=5e-5)
                lr_scheduler = get_scheduler(
                    name="linear",
                    optimizer=optimizer,
                    num_warmup_steps=0,
                    num_training_steps=1000,
                )
                dummy_optimizer = DummyOptim(params=model.parameters())
                dummy_lr_scheduler = DummyScheduler(dummy_optimizer)
                kwargs["train_batch_size"] = (
                    kwargs["train_micro_batch_size_per_gpu"]
                    * deepspeed_plugin.deepspeed_config["gradient_accumulation_steps"]
                    * accelerator.num_processes
                )
                accelerator.state.deepspeed_plugin.deepspeed_config_process(**kwargs)
                del accelerator.state.deepspeed_plugin.deepspeed_config["scheduler"]
                with self.assertRaises(ValueError) as cm:
                    model, optimizer, train_dataloader, eval_dataloader, lr_scheduler = accelerator.prep
                        model, dummy_optimizer, train_dataloader, eval_dataloader, lr_scheduler
                    )
                self.assertTrue(
                    "You can only specify `accelerate.utils.DummyScheduler` in the code when using `acce
                    in str(cm.exception)
                )

    def test_save_checkpoints(self):
        deepspeed_plugin = DeepSpeedPlugin(
            hf_ds_config=self.ds_config_file[ZERO3],
            zero3_init_flag=True,
        )
        del deepspeed_plugin.deepspeed_config["bf16"]
        kwargs = {
            "optimizer.params.lr": 5e-5,
            "optimizer.params.weight_decay": 0.0,
            "scheduler.params.warmup_min_lr": 0.0,
            "scheduler.params.warmup_max_lr": 5e-5,
            "scheduler.params.warmup_num_steps": 0,
            "train_micro_batch_size_per_gpu": 16,
            "gradient_clipping": 1.0,
            "train_batch_size": 16,
            "zero_optimization.reduce_bucket_size": 5e5,
            "zero_optimization.stage3_prefetch_bucket_size": 5e5,
            "zero_optimization.stage3_param_persistence_threshold": 5e5,
            "zero_optimization.stage3_gather_16bit_weights_on_model_save": False,
        }

        with mockenv_context(**self.dist_env):
            accelerator = Accelerator(deepspeed_plugin=deepspeed_plugin, mixed_precision="fp16")
            kwargs["train_batch_size"] = (
                kwargs["train_micro_batch_size_per_gpu"]
                * deepspeed_plugin.deepspeed_config["gradient_accumulation_steps"]
                * accelerator.num_processes
            )
            accelerator.state.deepspeed_plugin.deepspeed_config_process(**kwargs)

            train_set = RegressionDataset(length=80)
            eval_set = RegressionDataset(length=20)
            train_dataloader = DataLoader(train_set, batch_size=16, shuffle=True)
            eval_dataloader = DataLoader(eval_set, batch_size=32, shuffle=False)
            model = AutoModelForCausalLM.from_pretrained("gpt2")
            dummy_optimizer = DummyOptim(params=model.parameters())
            dummy_lr_scheduler = DummyScheduler(dummy_optimizer)

            model, _, train_dataloader, eval_dataloader, _ = accelerator.prepare(
                model, dummy_optimizer, train_dataloader, eval_dataloader, dummy_lr_scheduler
            )
            with self.assertRaises(ValueError) as cm:
                accelerator.get_state_dict(model)
            msg = (
                "Cannot get 16bit model weights because `stage3_gather_16bit_weights_on_model_save` in D
                "To save the model weights in 16bit, set `stage3_gather_16bit_weights_on_model_save` to 
                "set `zero3_save_16bit_model` to True when using `accelerate config`. "
                "To save the full checkpoint, run `model.save_checkpoint(save_dir)` and use `zero_to_fp3
            )
            self.assertTrue(msg in str(cm.exception))

    def test_autofill_dsconfig(self):
        deepspeed_plugin = DeepSpeedPlugin(
            hf_ds_config=self.ds_config_file[ZERO3],
            zero3_init_flag=True,
        )
        del deepspeed_plugin.deepspeed_config["bf16"]
        del deepspeed_plugin.deepspeed_config["fp16"]

        with mockenv_context(**self.dist_env):
            accelerator = Accelerator(deepspeed_plugin=deepspeed_plugin)
            train_set = RegressionDataset(length=80)
            eval_set = RegressionDataset(length=20)
            train_dataloader = DataLoader(train_set, batch_size=16, shuffle=True)
            eval_dataloader = DataLoader(eval_set, batch_size=32, shuffle=False)
            model = AutoModelForCausalLM.from_pretrained("gpt2")
            dummy_optimizer = DummyOptim(params=model.parameters(), lr=5e-5, weight_decay=1e-4)
            dummy_lr_scheduler = DummyScheduler(dummy_optimizer, warmup_num_steps=10, total_num_steps=10
            hidden_size = model.config.hidden_size
            model, _, train_dataloader, eval_dataloader, _ = accelerator.prepare(
                model, dummy_optimizer, train_dataloader, eval_dataloader, dummy_lr_scheduler
            )
            self.assertEqual(accelerator.deepspeed_config["train_micro_batch_size_per_gpu"], 16)
            self.assertEqual(accelerator.deepspeed_config["train_batch_size"], 16)

            self.assertEqual(accelerator.deepspeed_config["optimizer"]["params"]["lr"], 5e-5)
            self.assertEqual(accelerator.deepspeed_config["optimizer"]["params"]["weight_decay"], 1e-4)

            self.assertEqual(accelerator.deepspeed_config["scheduler"]["params"]["warmup_min_lr"], 0.0)
            self.assertEqual(accelerator.deepspeed_config["scheduler"]["params"]["warmup_max_lr"], 5e-5)
            self.assertEqual(accelerator.deepspeed_config["scheduler"]["params"]["warmup_num_steps"], 10

            self.assertEqual(accelerator.deepspeed_config["gradient_clipping"], 1.0)
            self.assertEqual(
                accelerator.deepspeed_config["zero_optimization"]["reduce_bucket_size"], hidden_size * h
            )
            self.assertEqual(
                accelerator.deepspeed_config["zero_optimization"]["stage3_prefetch_bucket_size"],
                0.9 * hidden_size * hidden_size,
        )
        self.assertEqual(
            accelerator.deepspeed_config["zero_optimization"]["stage3_param_persistence_threshold"],
            10 * hidden_size,
        )
        self.assertFalse(
            accelerator.deepspeed_config["zero_optimization"]["stage3_gather_16bit_weights_on_model_
        )

    @parameterized.expand([FP16, BF16], name_func=parameterized_custom_name_func)
    def test_autofill_dsconfig_from_ds_plugin(self, dtype):
        ds_config = self.ds_config_dict["zero3"]
        if dtype == BF16:
            del ds_config["fp16"]
        else:
            del ds_config["bf16"]
        ds_config[dtype]["enabled"] = "auto"
        ds_config["zero_optimization"]["stage"] = "auto"
        ds_config["zero_optimization"]["stage3_gather_16bit_weights_on_model_save"] = "auto"
        ds_config["zero_optimization"]["offload_optimizer"]["device"] = "auto"
        ds_config["zero_optimization"]["offload_param"]["device"] = "auto"
        ds_config["gradient_accumulation_steps"] = "auto"
        ds_config["gradient_clipping"] = "auto"

        deepspeed_plugin = DeepSpeedPlugin(
            hf_ds_config=ds_config,
            zero3_init_flag=True,
            gradient_accumulation_steps=1,
            gradient_clipping=1.0,
            zero_stage=2,
            offload_optimizer_device="cpu",
            offload_param_device="cpu",
            zero3_save_16bit_model=True,
        )

        with mockenv_context(**self.dist_env):
            accelerator = Accelerator(deepspeed_plugin=deepspeed_plugin, mixed_precision=dtype)
            deepspeed_plugin = accelerator.state.deepspeed_plugin
            self.assertEqual(deepspeed_plugin.deepspeed_config["gradient_clipping"], 1.0)
            self.assertEqual(deepspeed_plugin.deepspeed_config["gradient_accumulation_steps"], 1)
            self.assertEqual(deepspeed_plugin.deepspeed_config["zero_optimization"]["stage"], 2)
            self.assertEqual(
                deepspeed_plugin.deepspeed_config["zero_optimization"]["offload_optimizer"]["device"], "
            )
            self.assertEqual(deepspeed_plugin.deepspeed_config["zero_optimization"]["offload_param"]["de
            self.assertTrue(
                deepspeed_plugin.deepspeed_config["zero_optimization"]["stage3_gather_16bit_weights_on_m
            )
            self.assertTrue(deepspeed_plugin.deepspeed_config[dtype]["enabled"])

        AcceleratorState._reset_state(True)
        diff_dtype = "bf16" if dtype == "fp16" else "fp16"
        with mockenv_context(**self.dist_env):
            with self.assertRaises(ValueError) as cm:
                accelerator = Accelerator(deepspeed_plugin=deepspeed_plugin, mixed_precision=diff_dtype)
            self.assertTrue(
                f"`--mixed_precision` arg cannot be set to `{diff_dtype}` when `{dtype}` is set in the D
                in str(cm.exception)
            )

    def test_ds_config_assertions(self):
        ambiguous_env = self.dist_env.copy()
        ambiguous_env[
            "ACCELERATE_CONFIG_DS_FIELDS"
        ] = "gradient_accumulation_steps,gradient_clipping,zero_stage,offload_optimizer_device,offload_p

        with mockenv_context(**ambiguous_env):
            with self.assertRaises(ValueError) as cm:
                deepspeed_plugin = DeepSpeedPlugin(
                    hf_ds_config=self.ds_config_file[ZERO3],
                    zero3_init_flag=True,
                    gradient_accumulation_steps=1,
                    gradient_clipping=1.0,
                    zero_stage=ZERO2,
                    offload_optimizer_device="cpu",
                    offload_param_device="cpu",
                    zero3_save_16bit_model=True,
                )
                _ = Accelerator(deepspeed_plugin=deepspeed_plugin, mixed_precision=FP16)
            self.assertTrue(
                "If you are using an accelerate config file, remove others config variables mentioned in
                in str(cm.exception)
            )

    def test_basic_run(self):
        mod_file = inspect.getfile(accelerate.test_utils)
        test_file_path = os.path.sep.join(
            mod_file.split(os.path.sep)[:-1] + ["scripts", "external_deps", "test_performance.py"]
        )
        with tempfile.TemporaryDirectory() as dirpath:
            cmd = [
                "accelerate",
                "launch",
                "--num_processes=1",
                "--num_machines=1",
                "--machine_rank=0",
                "--mixed_precision=fp16",
                "--use_deepspeed",
                "--gradient_accumulation_steps=1",
                "--zero_stage=2",
                "--offload_optimizer_device=none",
                "--offload_param_device=none",
                test_file_path,
                "--model_name_or_path=distilbert-base-uncased",
                "--num_epochs=1",
                f"--output_dir={dirpath}",
            ]
            with patch_environment(omp_num_threads=1):
                execute_subprocess_async(cmd, env=os.environ.copy())


@require_deepspeed
@require_multi_gpu
@slow
class DeepSpeedIntegrationTest(TempDirTestCase):
    def setUp(self):
        super().setUp()
        self._test_file_path = inspect.getfile(self.__class__)
        path = Path(self._test_file_path).resolve()
        self.test_file_dir_str = str(path.parents[0])

        self.ds_config_file = dict(
            zero2=f"{self.test_file_dir_str}/ds_config_zero2.json",
            zero3=f"{self.test_file_dir_str}/ds_config_zero3.json",
        )

        self.stages = [1, 2, 3]
        self.zero3_offload_config = False
        self.performance_lower_bound = 0.82
        self.peak_memory_usage_upper_bound = {
            "multi_gpu_fp16": 3200,
            "deepspeed_stage_1_fp16": 1600,
            "deepspeed_stage_2_fp16": 2500,
            "deepspeed_stage_3_zero_init_fp16": 2800,
            # Disabling below test as it overwhelms the RAM memory usage
            # on CI self-hosted runner leading to tests getting killed.
            # "deepspeed_stage_3_cpu_offload_fp16": 1900,
        }
        self.n_train = 160
        self.n_val = 160

        mod_file = inspect.getfile(accelerate.test_utils)
        self.test_scripts_folder = os.path.sep.join(mod_file.split(os.path.sep)[:-1] + ["scripts", "exte
    def test_performance(self):
        self.test_file_path = os.path.join(self.test_scripts_folder, "test_performance.py")
        cmd = [
            "accelerate",
            "launch",
            "--num_processes=2",
            "--num_machines=1",
            "--machine_rank=0",
            "--mixed_precision=fp16",
            "--use_deepspeed",
            "--gradient_accumulation_steps=1",
            "--gradient_clipping=1",
            "--zero3_init_flag=True",
            "--zero3_save_16bit_model=True",
        ]
        for stage in self.stages:
            if stage == 1:
                continue
            cmd_stage = cmd.copy()
            cmd_stage.extend([f"--zero_stage={stage}"])
            cmd_stage.extend(["--offload_optimizer_device=none", "--offload_param_device=none"])
            if self.zero3_offload_config:
                with io.open(self.ds_config_file[ZERO3], "r", encoding="utf-8") as f:
                    ds_config = json.load(f)
                    del ds_config["bf16"]
                    del ds_config["optimizer"]["params"]["torch_adam"]
                    del ds_config["optimizer"]["params"]["adam_w_mode"]
                    ds_config["fp16"]["enabled"] = True
                    ds_config_path = os.path.join(self.tmpdir, "ds_config.json")
                    with open(ds_config_path, "w") as out_file:
                        json.dump(ds_config, out_file)

                cmd_stage.extend([f"--deepspeed_config_file={ds_config_path}"])

            cmd_stage.extend(
                [
                    self.test_file_path,
                    f"--output_dir={self.tmpdir}",
                    f"--performance_lower_bound={self.performance_lower_bound}",
                ]
            )
            with patch_environment(omp_num_threads=1):
                execute_subprocess_async(cmd_stage, env=os.environ.copy())

    def test_checkpointing(self):
        self.test_file_path = os.path.join(self.test_scripts_folder, "test_checkpointing.py")
        cmd = [
            "accelerate",
            "launch",
            "--num_processes=2",
            "--num_machines=1",
            "--machine_rank=0",
            "--mixed_precision=fp16",
            "--use_deepspeed",
            "--gradient_accumulation_steps=1",
            "--gradient_clipping=1",
            "--zero3_init_flag=True",
            "--zero3_save_16bit_model=True",
        ]
        for stage in self.stages:
            if stage == 1:
                continue
            cmd_stage = cmd.copy()
            cmd_stage.extend([f"--zero_stage={stage}"])
            cmd_stage.extend(["--offload_optimizer_device=none", "--offload_param_device=none"])
            if self.zero3_offload_config:
                with io.open(self.ds_config_file[ZERO3], "r", encoding="utf-8") as f:
                    ds_config = json.load(f)
                    del ds_config["bf16"]
                    del ds_config["optimizer"]["params"]["torch_adam"]
                    del ds_config["optimizer"]["params"]["adam_w_mode"]
                    ds_config["fp16"]["enabled"] = True
                    ds_config_path = os.path.join(self.tmpdir, "ds_config.json")
                    with open(ds_config_path, "w") as out_file:
                        json.dump(ds_config, out_file)

                cmd_stage.extend([f"--deepspeed_config_file={ds_config_path}"])

            cmd_stage.extend(
                [
                    self.test_file_path,
                    f"--output_dir={self.tmpdir}",
                    "--partial_train_epoch=1",
                ]
            )
            with patch_environment(omp_num_threads=1):
                execute_subprocess_async(cmd_stage, env=os.environ.copy())

            cmd_stage = cmd_stage[:-1]
            resume_from_checkpoint = os.path.join(self.tmpdir, "epoch_0")
            cmd_stage.extend(
                [
                    f"--resume_from_checkpoint={resume_from_checkpoint}",
                ]
            )
            with patch_environment(omp_num_threads=1):
                execute_subprocess_async(cmd_stage, env=os.environ.copy())

    def test_peak_memory_usage(self):
        self.test_file_path = os.path.join(self.test_scripts_folder, "test_peak_memory_usage.py")
        cmd = [
            "accelerate",
            "launch",
            "--num_processes=2",
            "--num_machines=1",
            "--machine_rank=0",
        ]
        for spec, peak_mem_upper_bound in self.peak_memory_usage_upper_bound.items():
            cmd_stage = cmd.copy()
            if "fp16" in spec:
                cmd_stage.extend(["--mixed_precision=fp16"])

            if "multi_gpu" in spec:
                continue
            else:
                cmd_stage.extend(
                    [
                        "--use_deepspeed",
                        "--gradient_accumulation_steps=1",
                        "--gradient_clipping=1",
                        "--zero3_init_flag=True",
                        "--zero3_save_16bit_model=True",
                    ]
                )
                for i in range(3):
                    if f"stage_{i+1}" in spec:
                        cmd_stage.extend([f"--zero_stage={i+1}"])
                        break
                cmd_stage.extend(["--offload_optimizer_device=none", "--offload_param_device=none"])
                if "cpu_offload" in spec:
                    with io.open(self.ds_config_file[ZERO3], "r", encoding="utf-8") as f:
                        ds_config = json.load(f)
                        del ds_config["bf16"]
                        del ds_config["fp16"]
                        del ds_config["optimizer"]["params"]["torch_adam"]
                        del ds_config["optimizer"]["params"]["adam_w_mode"]
                        ds_config_path = os.path.join(self.tmpdir, "ds_config.json")
                        with open(ds_config_path, "w") as out_file:
                            json.dump(ds_config, out_file)

                    cmd_stage.extend([f"--deepspeed_config_file={ds_config_path}"])

            cmd_stage.extend(
                [
                    self.test_file_path,
                    f"--output_dir={self.tmpdir}",
                    f"--peak_memory_upper_bound={peak_mem_upper_bound}",
                    f"--n_train={self.n_train}",
                    f"--n_val={self.n_val}",
                ]
            )
            with patch_environment(omp_num_threads=1):
                execute_subprocess_async(cmd_stage, env=os.environ.copy())
```

accelerate-main/tests/deepspeed/ds_config_zero3.json

```
{
    "fp16": {
        "enabled": "auto",
        "loss_scale": 0,
        "loss_scale_window": 1000,
        "initial_scale_power": 16,
        "hysteresis": 2,
        "min_loss_scale": 1
    },
    "bf16": {
        "enabled": "auto"
    },
    "optimizer": {
        "type": "AdamW",
        "params": {
            "lr": "auto",
            "weight_decay": "auto",
            "torch_adam": true,
            "adam_w_mode": true
        }
    },
    "scheduler": {
        "type": "WarmupLR",
        "params": {
            "warmup_min_lr": "auto",
            "warmup_max_lr": "auto",
            "warmup_num_steps": "auto"
        }
    },
    "zero_optimization": {
        "stage": 3,
        "offload_optimizer": {
            "device": "cpu",
            "pin_memory": true
        },
        "offload_param": {
            "device": "cpu",
            "pin_memory": true
        },
        "overlap_comm": true,
        "contiguous_gradients": true,
        "sub_group_size": 1e9,
        "reduce_bucket_size": "auto",
        "stage3_prefetch_bucket_size": "auto",
        "stage3_param_persistence_threshold": "auto",
        "stage3_max_live_parameters": 1e9,
        "stage3_max_reuse_distance": 1e9,
        "stage3_gather_16bit_weights_on_model_save": "auto"
    },
    "gradient_accumulation_steps": 1,
    "gradient_clipping": "auto",
    "steps_per_print": 2000,
    "train_batch_size": "auto",
    "train_micro_batch_size_per_gpu": "auto",
    "wall_clock_breakdown": false
}
```

accelerate-main/tests/deepspeed/ds_config_zero2.json

```json
{
    "fp16": {
        "enabled": "auto",
        "loss_scale": 0,
        "loss_scale_window": 1000,
        "initial_scale_power": 16,
        "hysteresis": 2,
        "min_loss_scale": 1
    },
    "bf16": {
        "enabled": "auto"
    },
    "optimizer": {
        "type": "AdamW",
        "params": {
            "lr": "auto",
            "weight_decay": "auto",
            "torch_adam": true,
            "adam_w_mode": true
        }
    },
    "scheduler": {
        "type": "WarmupLR",
        "params": {
            "warmup_min_lr": "auto",
            "warmup_max_lr": "auto",
            "warmup_num_steps": "auto"
        }
    },
    "zero_optimization": {
        "stage": 2,
        "offload_optimizer": {
            "device": "cpu",
            "pin_memory": true
        },
        "allgather_partitions": true,
        "allgather_bucket_size": 2e8,
        "overlap_comm": true,
        "reduce_scatter": true,
        "reduce_bucket_size": "auto",
        "contiguous_gradients": true
    },
    "gradient_accumulation_steps": 1,
    "gradient_clipping": "auto",
    "steps_per_print": 2000,
    "train_batch_size": "auto",
    "train_micro_batch_size_per_gpu": "auto",
    "wall_clock_breakdown": false
}
```

accelerate-main/tests/fsdp/test_fsdp.py

```
# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.


import inspect
import os

import torch
from transformers import AutoModel
from transformers.testing_utils import mockenv_context
from transformers.trainer_utils import set_seed

import accelerate
from accelerate.accelerator import Accelerator
from accelerate.state import AcceleratorState
from accelerate.test_utils.testing import (
    AccelerateTestCase,
    TempDirTestCase,
    execute_subprocess_async,
    require_cuda,
    require_fsdp,
    require_multi_gpu,
    slow,
)
from accelerate.utils.constants import (
    FSDP_AUTO_WRAP_POLICY,
    FSDP_BACKWARD_PREFETCH,
    FSDP_SHARDING_STRATEGY,
    FSDP_STATE_DICT_TYPE,
)
from accelerate.utils.dataclasses import FullyShardedDataParallelPlugin
from accelerate.utils.other import patch_environment


set_seed(42)

BERT_BASE_CASED = "bert-base-cased"
FP16 = "fp16"
BF16 = "bf16"
dtypes = [FP16, BF16]


@require_fsdp
@require_cuda
class FSDPPluginIntegration(AccelerateTestCase):
    def setUp(self):
        super().setUp()

        self.dist_env = dict(
            ACCELERATE_USE_FSDP="true",
            MASTER_ADDR="localhost",
            MASTER_PORT="10999",
            RANK="0",
            LOCAL_RANK="0",
            WORLD_SIZE="1",
        )
    def test_sharding_strategy(self):
        from torch.distributed.fsdp.fully_sharded_data_parallel import ShardingStrategy

        for i, strategy in enumerate(FSDP_SHARDING_STRATEGY):
            env = self.dist_env.copy()
            env["FSDP_SHARDING_STRATEGY"] = f"{i + 1}"
            env["FSDP_SHARDING_STRATEGY_NAME"] = strategy
            with mockenv_context(**env):
                fsdp_plugin = FullyShardedDataParallelPlugin()
                self.assertEqual(fsdp_plugin.sharding_strategy, ShardingStrategy(i + 1))

    def test_backward_prefetch(self):
        from torch.distributed.fsdp.fully_sharded_data_parallel import BackwardPrefetch

        for i, prefetch_policy in enumerate(FSDP_BACKWARD_PREFETCH):
            env = self.dist_env.copy()
            env["FSDP_BACKWARD_PREFETCH"] = prefetch_policy
            with mockenv_context(**env):
                fsdp_plugin = FullyShardedDataParallelPlugin()
                if prefetch_policy == "NO_PREFETCH":
                    self.assertIsNone(fsdp_plugin.backward_prefetch)
                else:
                    self.assertEqual(fsdp_plugin.backward_prefetch, BackwardPrefetch(i + 1))

    def test_state_dict_type(self):
        from torch.distributed.fsdp.fully_sharded_data_parallel import StateDictType

        for i, state_dict_type in enumerate(FSDP_STATE_DICT_TYPE):
            env = self.dist_env.copy()
            env["FSDP_STATE_DICT_TYPE"] = state_dict_type
            with mockenv_context(**env):
                fsdp_plugin = FullyShardedDataParallelPlugin()
                self.assertEqual(fsdp_plugin.state_dict_type, StateDictType(i + 1))
                if state_dict_type == "FULL_STATE_DICT":
                    self.assertTrue(fsdp_plugin.state_dict_config.offload_to_cpu)
                    self.assertTrue(fsdp_plugin.state_dict_config.rank0_only)

    def test_auto_wrap_policy(self):
        model = AutoModel.from_pretrained(BERT_BASE_CASED)
        for policy in FSDP_AUTO_WRAP_POLICY:
            env = self.dist_env.copy()
            env["FSDP_AUTO_WRAP_POLICY"] = policy
            if policy == "TRANSFORMER_BASED_WRAP":
                env["FSDP_TRANSFORMER_CLS_TO_WRAP"] = "BertLayer"
            elif policy == "SIZE_BASED_WRAP":
                env["FSDP_MIN_NUM_PARAMS"] = "2000"
            with mockenv_context(**env):
                fsdp_plugin = FullyShardedDataParallelPlugin()
                fsdp_plugin.set_auto_wrap_policy(model)
                if policy == "NO_WRAP":
                    self.assertIsNone(fsdp_plugin.auto_wrap_policy)
                else:
                    self.assertIsNotNone(fsdp_plugin.auto_wrap_policy)

        env = self.dist_env.copy()
        env["FSDP_AUTO_WRAP_POLICY"] = "TRANSFORMER_BASED_WRAP"
        env["FSDP_TRANSFORMER_CLS_TO_WRAP"] = "T5Layer"
        with mockenv_context(**env):
            fsdp_plugin = FullyShardedDataParallelPlugin()
            with self.assertRaises(Exception) as cm:
                fsdp_plugin.set_auto_wrap_policy(model)
            self.assertTrue("Could not find the transformer layer class to wrap in the model." in str(cm

        env = self.dist_env.copy()
        env["FSDP_AUTO_WRAP_POLICY"] = "SIZE_BASED_WRAP"
        env["FSDP_MIN_NUM_PARAMS"] = "0"
        with mockenv_context(**env):
            fsdp_plugin = FullyShardedDataParallelPlugin()
            fsdp_plugin.set_auto_wrap_policy(model)
            self.assertIsNone(fsdp_plugin.auto_wrap_policy)

    def test_mixed_precision(self):
        from torch.distributed.fsdp.fully_sharded_data_parallel import MixedPrecision
        from torch.distributed.fsdp.sharded_grad_scaler import ShardedGradScaler

        for mp_dtype in dtypes:
            env = self.dist_env.copy()
            env["ACCELERATE_MIXED_PRECISION"] = mp_dtype
            with mockenv_context(**env):
                accelerator = Accelerator()
                if mp_dtype == "fp16":
                    dtype = torch.float16
                elif mp_dtype == "bf16":
                    dtype = torch.bfloat16
                mp_policy = MixedPrecision(param_dtype=dtype, reduce_dtype=dtype, buffer_dtype=dtype)
                self.assertEqual(accelerator.state.fsdp_plugin.mixed_precision_policy, mp_policy)
                if mp_dtype == FP16:
                    self.assertTrue(isinstance(accelerator.scaler, ShardedGradScaler))
                elif mp_dtype == BF16:
                    self.assertIsNone(accelerator.scaler)
                AcceleratorState._reset_state(True)

    def test_cpu_offload(self):
        from torch.distributed.fsdp.fully_sharded_data_parallel import CPUOffload

        for flag in [True, False]:
            env = self.dist_env.copy()
            env["FSDP_OFFLOAD_PARAMS"] = str(flag).lower()
            with mockenv_context(**env):
                fsdp_plugin = FullyShardedDataParallelPlugin()
                self.assertEqual(fsdp_plugin.cpu_offload, CPUOffload(offload_params=flag))


@require_fsdp
@require_multi_gpu
@slow
class FSDPIntegrationTest(TempDirTestCase):
    def setUp(self):
        super().setUp()
        self.performance_lower_bound = 0.82
        self.performance_configs = [
            "fsdp_shard_grad_op_transformer_based_wrap",
            "fsdp_full_shard_transformer_based_wrap",
        ]
        self.peak_memory_usage_upper_bound = {
            "multi_gpu_fp16": 3200,
            "fsdp_shard_grad_op_transformer_based_wrap_fp16": 2000,
            "fsdp_full_shard_transformer_based_wrap_fp16": 1900,
            # Disabling below test as it overwhelms the RAM memory usage
            # on CI self-hosted runner leading to tests getting killed.
            # "fsdp_full_shard_cpu_offload_transformer_based_wrap_fp32": 1500,  # fp16 was leading to in
        }
        self.n_train = 160
        self.n_val = 160

        mod_file = inspect.getfile(accelerate.test_utils)
        self.test_scripts_folder = os.path.sep.join(mod_file.split(os.path.sep)[:-1] + ["scripts", "exte

    def test_performance(self):
        self.test_file_path = os.path.join(self.test_scripts_folder, "test_performance.py")
        cmd = ["accelerate", "launch", "--num_processes=2", "--num_machines=1", "--machine_rank=0", "--u
        for config in self.performance_configs:
            cmd_config = cmd.copy()
            for i, strategy in enumerate(FSDP_SHARDING_STRATEGY):
                if strategy.lower() in config:
                    cmd_config.append(f"--fsdp_sharding_strategy={i+1}")
                    break

            if "fp32" in config:
                cmd_config.append("--mixed_precision=no")
            else:
                cmd_config.append("--mixed_precision=fp16")

            if "cpu_offload" in config:
                cmd_config.append("--fsdp_offload_params=True")
            for policy in FSDP_AUTO_WRAP_POLICY:
                if policy.lower() in config:
                    cmd_config.append(f"--fsdp_auto_wrap_policy={policy}")
                    break

            if policy == "TRANSFORMER_BASED_WRAP":
                cmd_config.append("--fsdp_transformer_layer_cls_to_wrap=BertLayer")
            elif policy == "SIZE_BASED_WRAP":
                cmd_config.append("--fsdp_min_num_params=2000")

            cmd_config.extend(
                [
                    self.test_file_path,
                    f"--output_dir={self.tmpdir}",
                    f"--performance_lower_bound={self.performance_lower_bound}",
                ]
            )
            with patch_environment(omp_num_threads=1):
                execute_subprocess_async(cmd_config, env=os.environ.copy())

    def test_checkpointing(self):
        self.test_file_path = os.path.join(self.test_scripts_folder, "test_checkpointing.py")
        cmd = [
            "accelerate",
            "launch",
            "--num_processes=2",
            "--num_machines=1",
            "--machine_rank=0",
            "--use_fsdp",
            "--mixed_precision=fp16",
            "--fsdp_transformer_layer_cls_to_wrap=BertLayer",
        ]

        for i, strategy in enumerate(FSDP_SHARDING_STRATEGY):
            cmd_config = cmd.copy()
            cmd_config.append(f"--fsdp_sharding_strategy={i+1}")
            if strategy != "FULL_SHARD":
                continue
            state_dict_config_index = len(cmd_config)
            for state_dict_type in FSDP_STATE_DICT_TYPE:
                cmd_config = cmd_config[:state_dict_config_index]
                if state_dict_type == "SHARDED_STATE_DICT":
                    continue
                cmd_config.append(f"--fsdp_state_dict_type={state_dict_type}")
                cmd_config.extend(
                    [
                        self.test_file_path,
                        f"--output_dir={self.tmpdir}",
                        "--partial_train_epoch=1",
                    ]
                )
                with patch_environment(omp_num_threads=1):
                    execute_subprocess_async(cmd_config, env=os.environ.copy())

                cmd_config = cmd_config[:-1]
                resume_from_checkpoint = os.path.join(self.tmpdir, "epoch_0")
                cmd_config.extend(
                    [
                        f"--resume_from_checkpoint={resume_from_checkpoint}",
                    ]
                )
                with patch_environment(omp_num_threads=1):
                    execute_subprocess_async(cmd_config, env=os.environ.copy())

    def test_peak_memory_usage(self):
        self.test_file_path = os.path.join(self.test_scripts_folder, "test_peak_memory_usage.py")
        cmd = [
            "accelerate",
            "launch",
            "--num_processes=2",
            "--num_machines=1",
            "--machine_rank=0",
        ]
        for spec, peak_mem_upper_bound in self.peak_memory_usage_upper_bound.items():
            cmd_config = cmd.copy()
            if "fp16" in spec:
                cmd_config.extend(["--mixed_precision=fp16"])
            else:
                cmd_config.extend(["--mixed_precision=no"])

            if "multi_gpu" in spec:
                continue
            else:
                cmd_config.extend(["--use_fsdp"])
                for i, strategy in enumerate(FSDP_SHARDING_STRATEGY):
                    if strategy.lower() in spec:
                        cmd_config.append(f"--fsdp_sharding_strategy={i+1}")
                        break

                if "cpu_offload" in spec:
                    cmd_config.append("--fsdp_offload_params=True")

                for policy in FSDP_AUTO_WRAP_POLICY:
                    if policy.lower() in spec:
                        cmd_config.append(f"--fsdp_auto_wrap_policy={policy}")
                        break

                if policy == "TRANSFORMER_BASED_WRAP":
                    cmd_config.append("--fsdp_transformer_layer_cls_to_wrap=BertLayer")
                elif policy == "SIZE_BASED_WRAP":
                    cmd_config.append("--fsdp_min_num_params=2000")

            cmd_config.extend(
                [
                    self.test_file_path,
                    f"--output_dir={self.tmpdir}",
                    f"--peak_memory_upper_bound={peak_mem_upper_bound}",
                    f"--n_train={self.n_train}",
                    f"--n_val={self.n_val}",
                ]
            )
            with patch_environment(omp_num_threads=1):
                execute_subprocess_async(cmd_config, env=os.environ.copy())
```

accelerate-main/docs/Makefile

```
# Minimal makefile for Sphinx documentation
#

# You can set these variables from the command line.
SPHINXOPTS    =
SPHINXBUILD   = sphinx-build
SOURCEDIR     = source
BUILDDIR      = _build

# Put it first so that "make" without argument is like "make help".
help:
■@$(SPHINXBUILD) -M help "$(SOURCEDIR)" "$(BUILDDIR)" $(SPHINXOPTS) $(O)

.PHONY: help Makefile

# Catch-all target: route all unknown targets to Sphinx using the new
# "make mode" option.  $(O) is meant as a shortcut for $(SPHINXOPTS).
%: Makefile
■@$(SPHINXBUILD) -M $@ "$(SOURCEDIR)" "$(BUILDDIR)" $(SPHINXOPTS) $(O)
```

accelerate-main/examples/by_feature/README.md

~~~
# What are these scripts?

All scripts in this folder originate from the `nlp_example.py` file, as it is a very simplistic NLP trai

From there, each further script adds in just **one** feature of Accelerate, showing how you can quickly

A full example with all of these parts integrated together can be found in the `complete_nlp_example.py`

Adjustments to each script from the base `nlp_example.py` file can be found quickly by searching for "#

## Example Scripts by Feature and their Arguments

### Base Example (`../nlp_example.py`)

- Shows how to use `Accelerator` in an extremely simplistic PyTorch training loop
- Arguments available:
  - `mixed_precision`, whether to use mixed precision. ("no", "fp16", or "bf16")
  - `cpu`, whether to train using only the CPU. (yes/no/1/0)

All following scripts also accept these arguments in addition to their added ones.

These arguments should be added at the end of any method for starting the python script (such as `python

```bash
accelerate launch ../nlp_example.py --mixed_precision fp16 --cpu 0
~~~

### Checkpointing and Resuming Training (`checkpointing.py`)

- Shows how to use `Accelerator.save_state` and `Accelerator.load_state` to save or continue training
- **It is assumed you are continuing off the same training script**
- Arguments available:
  - `checkpointing_steps`, after how many steps the various states should be saved. ("epoch", 1, 2, ...)
  - `output_dir`, where saved state folders should be saved to, default is current working directory
  - `resume_from_checkpoint`, what checkpoint folder to resume from. ("epoch_0", "step_22", ...)

These arguments should be added at the end of any method for starting the python script (such as `python

(Note, `resume_from_checkpoint` assumes that we've ran the script for one epoch with the `--checkpointin

```bash
accelerate launch ./checkpointing.py --checkpointing_steps epoch output_dir "checkpointing_tutorial" --r
```

### Cross Validation (`cross_validation.py`)

- Shows how to use `Accelerator.free_memory` and run cross validation efficiently with `datasets`.
- Arguments available:
  - `num_folds`, the number of folds the training dataset should be split into.

These arguments should be added at the end of any method for starting the python script (such as `python

```bash
accelerate launch ./cross_validation.py --num_folds 2
```

### Experiment Tracking (`tracking.py`)

- Shows how to use `Accelerate.init_trackers` and `Accelerator.log`
- Can be used with Weights and Biases, TensorBoard, or CometML.
- Arguments available:
  - `with_tracking`, whether to load in all available experiment trackers from the environment.

These arguments should be added at the end of any method for starting the python script (such as `python

```bash
accelerate launch ./tracking.py --with_tracking
```
```
```

### Gradient Accumulation (`gradient_accumulation.py`)

- Shows how to use `Accelerator.no_sync` to prevent gradient averaging in a distributed setup.
- Arguments available:
  - `gradient_accumulation_steps`, the number of steps to perform before the gradients are accumulated a

These arguments should be added at the end of any method for starting the python script (such as `python

```bash
accelerate launch ./gradient_accumulation.py --gradient_accumulation_steps 5
```

accelerate-main/docs/source/index.mdx

~~~
<!--Copyright 2022 The HuggingFace Team. All rights reserved.

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in comp
the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is dis
an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the Licen
specific language governing permissions and limitations under the License.
-->

# Accelerate

■ Accelerate is a library that enables the same PyTorch code to be run across any distributed configurat

```diff
+ from accelerate import Accelerator
+ accelerator = Accelerator()

+ model, optimizer, training_dataloader, scheduler = accelerator.prepare(
+     model, optimizer, training_dataloader, scheduler
+ )

  for batch in training_dataloader:
      optimizer.zero_grad()
      inputs, targets = batch
      inputs = inputs.to(device)
      targets = targets.to(device)
      outputs = model(inputs)
      loss = loss_function(outputs, targets)
+     accelerator.backward(loss)
      optimizer.step()
      scheduler.step()
~~~

Built on `torch_xla` and `torch.distributed`, ■ Accelerate takes care of the heavy lifting, so you don't
Convert existing codebases to utilize [DeepSpeed](usage_guides/deepspeed), perform [fully sharded data p

<Tip>

  To get a better idea of this process, make sure to check out the [Tutorials](basic_tutorials/overview)

</Tip>

This code can then be launched on any system through Accelerate's CLI interface:
```bash
accelerate launch {my_script.py}
```

<div class="mt-10">
  <div class="w-full flex flex-col space-y-4 md:space-y-0 md:grid md:grid-cols-2 md:gap-y-4 md:gap-x-5">
    <a class="!no-underline border dark:border-gray-700 p-5 rounded-lg shadow hover:shadow-lg" href="./b
      ><div class="w-full text-center bg-gradient-to-br from-blue-400 to-blue-500 rounded-lg py-1.5 font
      <p class="text-gray-700">Learn the basics and become familiar with using ■ Accelerate. Start here
    </a>
    <a class="!no-underline border dark:border-gray-700 p-5 rounded-lg shadow hover:shadow-lg" href="./u
      ><div class="w-full text-center bg-gradient-to-br from-indigo-400 to-indigo-500 rounded-lg py-1.5
      <p class="text-gray-700">Practical guides to help you achieve a specific goal. Take a look at thes
    </a>
    <a class="!no-underline border dark:border-gray-700 p-5 rounded-lg shadow hover:shadow-lg" href="./c
      ><div class="w-full text-center bg-gradient-to-br from-pink-400 to-pink-500 rounded-lg py-1.5 font
      <p class="text-gray-700">High-level explanations for building a better understanding of important
   </a>
    <a class="!no-underline border dark:border-gray-700 p-5 rounded-lg shadow hover:shadow-lg" href="./p
      ><div class="w-full text-center bg-gradient-to-br from-purple-400 to-purple-500 rounded-lg py-1.5
      <p class="text-gray-700">Technical descriptions of how ■ Accelerate classes and methods work.</p>
```
```

```
    </a>
  </div>
</div>
```

accelerate-main/docs/source/_toctree.yml

```
- sections:
  - local: index
    title: ■ Accelerate
  - local: basic_tutorials/install
    title: Installation
  - local: quicktour
    title: Quicktour
  title: Getting started
- sections:
  - local: basic_tutorials/overview
    title: Overview
  - local: basic_tutorials/migration
    title: Migrating to ■ Accelerate
  - local: basic_tutorials/launch
    title: Launching distributed code
  - local: basic_tutorials/notebook
    title: Launching distributed training from Jupyter Notebooks
  title: Tutorials
- sections:
  - local: usage_guides/explore
    title: Start Here!
  - local: usage_guides/training_zoo
    title: Example Zoo
  - local: usage_guides/big_modeling
    title: How perform inference on large models with small resources
  - local: usage_guides/gradient_accumulation
    title: Performing gradient accumulation
  - local: usage_guides/checkpoint
    title: Saving and loading training states
  - local: usage_guides/tracking
    title: Using experiment trackers
  - local: usage_guides/memory
    title: How to avoid CUDA Out-of-Memory
  - local: usage_guides/mps
    title: How to use Apple Silicon M1 GPUs
  - local: usage_guides/deepspeed
    title: How to use DeepSpeed
  - local: usage_guides/fsdp
    title: How to use Fully Sharded Data Parallelism
  - local: usage_guides/megatron_lm
    title: How to use Megatron-LM
  - local: usage_guides/sagemaker
    title: How to use ■ Accelerate with SageMaker
  title: How-To Guides
- sections:
  - local: concept_guides/performance
    title: Comparing performance across distributed setups
  - local: concept_guides/deferring_execution
    title: Executing and deferring jobs
  - local: concept_guides/gradient_synchronization
    title: Gradient synchronization
  - local: concept_guides/training_tpu
    title: TPU best practices
  title: Concepts and fundamentals
- sections:
  - local: package_reference/accelerator
    title: Main Accelerator class
  - local: package_reference/state
    title: Stateful configuration classes
  - local: package_reference/cli
    title: The Command Line
  - local: package_reference/torch_wrappers
    title: Torch wrapper classes
  - local: package_reference/tracking
    title: Experiment trackers
  - local: package_reference/launchers
    title: Distributed launchers
  - local: package_reference/deepspeed
    title: DeepSpeed utilities
  - local: package_reference/logging
    title: Logging
  - local: package_reference/big_modeling
    title: Working with large models
  - local: package_reference/kwargs
    title: Kwargs handlers
  - local: package_reference/utilities
    title: Utility functions and classes
  - local: package_reference/megatron_lm
    title: Megatron-LM Utilities
  title: "Reference"
```

accelerate-main/docs/source/quicktour.mdx

# Quick tour

Let's have a look at the ■ Accelerate main features and traps to avoid.

## Main use

To use ■ Accelerate in your own script, you have to change four things:

1. Import the [`Accelerator`] main class and instantiate one in an `accelerator` object:

```python
from accelerate import Accelerator

accelerator = Accelerator()
```

This should happen as early as possible in your training script as it will initialize everything necessa
distributed training. You don't need to indicate the kind of environment you are in (just one machine wi
machines with several GPUs, several machines with multiple GPUs or a TPU), the library will detect this

2. Remove the call `.to(device)` or `.cuda()` for your model and input data. The `accelerator` object
will handle this for you and place all those objects on the right device for you. If you know what you'r
can leave those `.to(device)` calls but you should use the device provided by the `accelerator` object:
`accelerator.device`.

To fully deactivate the automatic device placement, pass along `device_placement=False` when initializin
[`Accelerator`].

<Tip warning={true}>

    If you place your objects manually on the proper device, be careful to create your optimizer after p
    model on `accelerator.device` or your training will fail on TPU.

</Tip>

3. Pass all objects relevant to training (optimizer, model, training dataloader, learning rate scheduler
[`~Accelerator.prepare`] method. This will make sure everything is ready for training.

```python
model, optimizer, train_dataloader, lr_scheduler = accelerator.prepare(
    model, optimizer, train_dataloader, lr_scheduler
)
```

In particular, your training dataloader will be sharded across all GPUs/TPU cores available so that each
different portion of the training dataset. Also, the random states of all processes will be synchronized
beginning of each iteration through your dataloader, to make sure the data is shuffled the same way (if
use `shuffle=True` or any kind of random sampler).

<Tip>

    The actual batch size for your training will be the number of devices used multiplied by the batch s
    your script: for instance training on 4 GPUs with a batch size of 16 set when creating the training
    train at an actual batch size of 64.

</Tip>

Alternatively, you can use the option `split_batches=True` when creating initializing your
[`Accelerator`], in which case the batch size will always stay the same, whether your run your
script on 1, 2, 4 or 64 GPUs.

You should execute this instruction as soon as all objects for training are created, before starting you
training loop.

<Tip warning={true}>

You should only pass the learning rate scheduler to [`~Accelerator.prepare`] when the scheduler need
at each optimizer step.

</Tip>

<Tip warning={true}>

Your training dataloader may change length when going through this method: if you run on X GPUs, it
length divided by X (since your actual batch size will be multiplied by X), unless you set
`split_batches=True`.

</Tip>

Any instruction using your training dataloader length (for instance if you want to log the number of tot
steps) should go after the call to [`~Accelerator.prepare`].

You can perfectly send your dataloader to [`~Accelerator.prepare`] on its own, but it's best to send the
model and optimizer to [`~Accelerator.prepare`] together.

You may or may not want to send your validation dataloader to [`~Accelerator.prepare`], depending on
whether you want to run distributed evaluation or not (see below).

4. Replace the line `loss.backward()` by `accelerator.backward(loss)`.

And you're all set! With all these changes, your script will run on your local machine as well as on mul
TPU! You can either use your favorite tool to launch the distributed training, or you can use the ■ Acce
launcher.

## Distributed evaluation

You can perform regular evaluation in your training script, if you leave your validation dataloader out
[`~Accelerator.prepare`] method. In this case, you will need to put the input data on the
`accelerator.device` manually.

To perform distributed evaluation, send along your validation dataloader to the [`~Accelerator.prepare`]
method:

```python
validation_dataloader = accelerator.prepare(validation_dataloader)
```

As for your training dataloader, it will mean that (should you run your script on multiple devices) each
only see part of the evaluation data. This means you will need to group your predictions together. This
do with the [`~Accelerator.gather_for_metrics`] method.

```python
for inputs, targets in validation_dataloader:
    predictions = model(inputs)
    # Gather all predictions and targets
    all_predictions, all_targets = accelerator.gather_for_metrics((predictions, targets))
    # Example of use with a *Datasets.Metric*
    metric.add_batch(all_predictions, all_targets)
```

<Tip warning={true}>

Similar to the training dataloader, passing your validation dataloader through
[`~Accelerator.prepare`] may change it: if you run on X GPUs, it will have its length divided by X
(since your actual batch size will be multiplied by X), unless you set `split_batches=True`.

</Tip>
Any instruction using your training dataloader length (for instance if you need the number of total trai

to create a learning rate scheduler) should go after the call to [`~Accelerator.prepare`].

Some data at the end of the dataset may be duplicated so the batch can be divided equally among all work
should be calculated through the [`~Accelerator.gather_for_metrics`] method to automatically remove the

<Tip>

If for some reason you don't wish to have this automatically done, [`~Accelerator.gather`] can be us
the data across all processes and this can manually be done instead.

</Tip>

<Tip warning={true}>

The [`~Accelerator.gather`] and [`~Accelerator.gather_for_metrics`] methods require the tensors to b
you have tensors of different sizes on each process (for instance when dynamically padding to the ma
a batch), you should use the [`~Accelerator.pad_across_processes`] method to pad you tensor to the
biggest size across processes.

</Tip>

## Launching your distributed script

You can use the regular commands to launch your distributed training (like `torch.distributed.run` for
PyTorch), they are fully compatible with ■ Accelerate.

■ Accelerate also provides a CLI tool that unifies all launchers, so you only have to remember one comma
just run:

```bash
accelerate config
```

on your machine and reply to the questions asked. This will save a *default_config.yaml* file in your ca
■ Accelerate. That cache folder is (with decreasing order of priority):

- The content of your environment variable `HF_HOME` suffixed with *accelerate*.
- If it does not exist, the content of your environment variable `XDG_CACHE_HOME` suffixed with
  huggingface/accelerate.
- If this does not exist either, the folder *~/.cache/huggingface/accelerate*

You can also specify with the flag `--config_file` the location of the file you want to save.

Once this is done, you can test everything is going well on your setup by running:

```bash
accelerate test
```

This will launch a short script that will test the distributed environment. If it runs fine, you are rea
step!

Note that if you specified a location for the config file in the previous step, you need to pass it here

```bash
accelerate test --config_file path_to_config.yaml
```

Now that this is done, you can run your script with the following command:

```bash
accelerate launch path_to_script.py --args_for_the_script
```

If you stored the config file in a non-default location, you can indicate it to the launcher like this:

```bash
accelerate launch --config_file path_to_config.yaml path_to_script.py --args_for_the_script
```

You can also override any of the arguments determined by your config file.

To see the complete list of parameters that you can pass in, run `accelerate launch -h`.

Check out the [Launch tutorial](basic_tutorials/launch) for more information about launching your script

## Launching training from a notebook

In Accelerate 0.3.0, a new [`notebook_launcher`] has been introduced to help you launch your training
function from a notebook. This launcher supports launching a training with TPUs on Colab or Kaggle, as w
on several GPUs (if the machine on which you are running your notebook has them).

Just define a function responsible for your whole training and/or evaluation in a cell of the notebook,
cell with the following code:

```python
from accelerate import notebook_launcher

notebook_launcher(training_function)
```

<Tip warning={true}>

    Your [`Accelerator`] object should only be defined inside the training function. This is because the
    initialization should be done inside the launcher only.

</Tip>

Check out the [Notebook Launcher tutorial](basic_tutorials/notebook) for more information about training

## Training on TPU

If you want to launch your script on TPUs, there are a few caveats you should be aware of. Behind the sc
will create a graph of all the operations happening in your training step (forward pass, backward pass a
step). This is why your first step of training will always be very long as building and compiling this g
optimizations takes some time.

The good news is that this compilation will be cached so the second step and all the following will be m
bad news is that it only applies if all of your steps do exactly the same operations, which implies:

- having all tensors of the same length in all your batches
- having static code (i.e., not a for loop of length that could change from step to step)

Having any of the things above change between two steps will trigger a new compilation which will, once
lot of time. In practice, that means you must take special care to have all your tensors in your inputs
shape (so no dynamic padding for instance if you are in an NLP problem) and should not use layers with f
have different lengths depending on the inputs (such as an LSTM) or the training will be excruciatingly

To introduce special behavior in your script for TPUs you can check the `distributed_type` of your
`accelerator`:

```python docstyle-ignore
from accelerate import DistributedType

if accelerator.distributed_type == DistributedType.TPU:
    # do something of static shape
else:
    # go crazy and be dynamic
```

The [NLP example](https://github.com/huggingface/accelerate/blob/main/examples/nlp_example.py) shows an
situation with dynamic padding.

One last thing to pay close attention to: if your model has tied weights (such as language models which
of the embedding matrix with the weights of the decoder), moving this model to the TPU (either yourself
passed your model to [`~Accelerator.prepare`]) will break the tying. You will need to retie the weights
after. You can find an example of this in the [run_clm_no_trainer](https://github.com/huggingface/transf
the Transformers repository.

Check out the [TPU tutorial](concept_guides/training_tpu) for more information about training on TPUs.
## Other caveats
We list here all smaller issues you could have in your script conversion and how to resolve them.

### Execute a statement only on one processes

Some of your instructions only need to run for one process on a given server: for instance a data downlo
statement. To do this, wrap the statement in a test like this:

```python docstyle-ignore
if accelerator.is_local_main_process:
    # Is executed once per server
```

Another example is progress bars: to avoid having multiple progress bars in your output, you should only
the local main process:

```python
from tqdm.auto import tqdm

progress_bar = tqdm(range(args.max_train_steps), disable=not accelerator.is_local_main_process)
```

The *local* means per machine: if you are running your training on two servers with several GPUs, the in
be executed once on each of those servers. If you need to execute something only once for all processes
machine) for instance, uploading the final model to the ■ model hub, wrap it in a test like this:

```python docstyle-ignore
if accelerator.is_main_process:
    # Is executed once only
```

For printing statements you only want executed once per machine, you can just replace the `print` functi
`accelerator.print`.

### Defer execution

When you run your usual script, instructions are executed in order. Using ■ Accelerate to deploy your sc
GPUs at the same time introduces a complication: while each process executes all instructions in order,
faster than others.

You might need to wait for all processes to have reached a certain point before executing a given instru
instance, you shouldn't save a model before being sure every process is done with training. To do this,
following line in your code:

```
accelerator.wait_for_everyone()
```

This instruction will block all the processes that arrive first until all the other processes have reach
point (if you run your script on just one GPU or CPU, this won't do anything).

### Saving/loading a model

Saving the model you trained might need a bit of adjustment: first you should wait for all processes to
point in the script as shown above, and then, you should unwrap your model before saving it. This is bec
through the [`~Accelerator.prepare`] method, your model may have been placed inside a bigger model,
which deals with the distributed training. This in turn means that saving your model state dictionary wi
any precaution will take that potential extra layer into account, and you will end up with weights you c
in your base model.

This is why it's recommended to *unwrap* your model first. Here is an example:

```
accelerator.wait_for_everyone()
unwrapped_model = accelerator.unwrap_model(model)
accelerator.save(unwrapped_model.state_dict(), filename)
```

If your script contains logic to load a checkpoint, we also recommend you load your weights in the unwra
(this is only useful if you use the load function after making your model go through
[`~Accelerator.prepare`]). Here is an example:

```
```

unwrapped_model = accelerator.unwrap_model(model)
unwrapped_model.load_state_dict(torch.load(filename))
```

Note that since all the model parameters are references to tensors, this will load your weights inside `

## Saving/loading entire states

When training your model, you may want to save the current state of the model, optimizer, random generat
You can use [`~Accelerator.save_state`] and [`~Accelerator.load_state`] respectively to do so.

To further customize where and how states saved through [`~Accelerator.save_state`] the [`~utils.Project
if `automatic_checkpoint_naming` is enabled each saved checkpoint will be located then at `Accelerator.p

If you have registered any other stateful items to be stored through [`~Accelerator.register_for_checkpo

<Tip>

    Every object passed to [`~Accelerator.register_for_checkpointing`] must have a `load_state_dict` and

</Tip>


### Gradient clipping

If you are using gradient clipping in your script, you should replace the calls to
`torch.nn.utils.clip_grad_norm_` or `torch.nn.utils.clip_grad_value_` with [`~Accelerator.clip_grad_norm
and [`~Accelerator.clip_grad_value_`] respectively.


### Mixed Precision training

If you are running your training in Mixed Precision with ■ Accelerate, you will get the best result with
computed inside your model (like in Transformer models for instance). Every computation outside of the m
executed in full precision (which is generally what you want for loss computation, especially if it invo
softmax). However you might want to put your loss computation inside the *accelerator.autocast* context

```
with accelerator.autocast():
    loss = complex_loss_function(outputs, target):
```

Another caveat with Mixed Precision training is that the gradient will skip a few updates at the beginni
sometimes during training: because of the dynamic loss scaling strategy, there are points during trainin
gradients have overflown, and the loss scaling factor is reduced to avoid this happening again at the ne

This means that you may update your learning rate scheduler when there was no update, which is fine in g
have an impact when you have very little training data, or if the first learning rate values of your sch
important. In this case, you can skip the learning rate scheduler updates when the optimizer step was no
this:

```
if not accelerator.optimizer_step_was_skipped:
    lr_scheduler.step()
~~~

### Gradient Accumulation

To perform gradient accumulation use [`~Accelerator.accumulate`] and specify a `gradient_accumulation_st
This will also automatically ensure the gradients are synced or unsynced when on multi-device training,
actually be performed, and auto-scale the loss:

```python
accelerator = Accelerator(gradient_accumulation_steps=2)
model, optimizer, training_dataloader = accelerator.prepare(model, optimizer, training_dataloader)

for input, label in training_dataloader:
    with accelerator.accumulate(model):
        predictions = model(input)
        loss = loss_function(predictions, label)
        accelerator.backward(loss)
        optimizer.step()
~~~

```
        scheduler.step()
        optimizer.zero_grad()
```

### DeepSpeed

DeepSpeed support is experimental, so the underlying API will evolve in the near future and may have som
breaking changes. In particular, ■ Accelerate does not support DeepSpeed config you have written yoursel
will be added in a next version.

<Tip warning={true}>

    The [`notebook_launcher`] does not support the DeepSpeed integration yet.

</Tip>

## Internal mechanism

Internally, the library works by first analyzing the environment in which the script is launched to dete
kind of distributed setup is used, how many different processes there are and which one the current scri
that information is stored in the [`~AcceleratorState`].

This class is initialized the first time you instantiate an [`~Accelerator`] as well as performing any
specific initialization your distributed setup needs. Its state is then uniquely shared through all inst
[`~state.AcceleratorState`].

Then, when calling [`~Accelerator.prepare`], the library:

- wraps your model(s) in the container adapted for the distributed setup,
- wraps your optimizer(s) in a [`~optimizer.AcceleratedOptimizer`],
- creates a new version of your dataloader(s) in a [`~data_loader.DataLoaderShard`].

While the model(s) and optimizer(s) are just put in simple wrappers, the dataloader(s) are re-created. T
because PyTorch does not let the user change the `batch_sampler` of a dataloader once it's been created
library handles the sharding of your data between processes by changing that `batch_sampler` to yield ev
`num_processes` batches.

The [`~data_loader.DataLoaderShard`] subclasses `DataLoader` to add the following functionality:

- it synchronizes the appropriate random number generator of all processes at each new iteration, to ens
  randomization (like shuffling) is done the exact same way across processes.
- it puts the batches on the proper device before yielding them (unless you have opted out of
  `device_placement=True`).

The random number generator synchronization will by default synchronize:

- the `generator` attribute of a given sampler (like the PyTorch `RandomSampler`) for PyTorch >= 1.6
- the main random number generator in PyTorch <=1.5.1

You can choose which random number generator(s) to synchronize with the `rng_types` argument of the main
[`Accelerator`]. In PyTorch >= 1.6, it is recommended to rely on a local `generator` to avoid
setting the same seed in the main random number generator in all processes.

<Tip warning={true}>

    Synchronization of the main torch (or CUDA or XLA) random number generator will affect any other pot
    artifacts you could have in your dataset (like random data augmentation) in the sense that all proce
    the same random numbers from the torch random modules (so will apply the same random data augmentati
    controlled by torch).

</Tip>

<Tip>

    The randomization part of your custom sampler, batch sampler or iterable dataset should be done usin
    `torch.Generator` object (in PyTorch >= 1.6), see the traditional `RandomSampler`, as an example.

</Tip>

For more details about the internals, see the [Internals page](package_reference/torch_wrappers).

accelerate-main/docs/source/imgs/course_banner.png

accelerate-main/docs/source/imgs/accelerate_logo.png

Accelerate

accelerate-main/docs/source/concept_guides/training_tpu.mdx

# Training on TPUs with ■ Accelerate

Training on TPUs can be slightly different than training on multi-gpu, even with ■ Accelerate. This guid
where you should be careful and why, as well as the best practices in general.

## Training in a Notebook

The main carepoint when training on TPUs comes from the [`notebook_launcher`]. As mentioned in the [note
restructure your training code into a function that can get passed to the [`notebook_launcher`] function

While on a TPU that last part is not as important, a critical part to understand is that when you launch
When launching from the command-line, you perform **spawning**, where a python process is not currently
utilizing a python process, you need to *fork* a new process from it to launch your code.

Where this becomes important is in regards to declaring your model. On forked TPU processes, it is recom
training function. This is different than training on GPUs where you create `n` models that have their g
model instance is shared between all the nodes and it is passed back and forth. This is important especi
on Google Colaboratory.

Below is an example of a training function passed to the [`notebook_launcher`] if training on CPUs or GP

<Tip>

    This code snippet is based off the one from the `simple_nlp_example` notebook found [here](https://g
    modifications for the sake of simplicity

</Tip>

```python
def training_function():
    # Initialize accelerator
    accelerator = Accelerator()
    model = AutoModelForSequenceClassification.from_pretrained("bert-base-cased", num_labels=2)
    train_dataloader, eval_dataloader = create_dataloaders(
        train_batch_size=hyperparameters["train_batch_size"], eval_batch_size=hyperparameters["eval_batc
    )

    # Instantiate optimizer
    optimizer = AdamW(params=model.parameters(), lr=hyperparameters["learning_rate"])

    # Prepare everything
    # There is no specific order to remember, we just need to unpack the objects in the same order we ga
    # prepare method.
    model, optimizer, train_dataloader, eval_dataloader = accelerator.prepare(
        model, optimizer, train_dataloader, eval_dataloader
    )

    num_epochs = hyperparameters["num_epochs"]
    # Now we train the model
    for epoch in range(num_epochs):
        model.train()
        for step, batch in enumerate(train_dataloader):
            outputs = model(**batch)
            loss = outputs.loss
            accelerator.backward(loss)
            optimizer.step()
```

            optimizer.zero_grad()

```python
from accelerate import notebook_launcher

notebook_launcher(training_function)
```

<Tip>

    The `notebook_launcher` will default to 8 processes if ■ Accelerate has been configured for a TPU

</Tip>

If you use this example and declare the model *inside* the training loop, then on a low-resource system
like:

```
ProcessExitedException: process 0 terminated with signal SIGSEGV
```

This error is *extremely* cryptic but the basic explanation is you ran out of system RAM. You can avoid
accept a single `model` argument, and declare it in an outside cell:

```python
# In another Jupyter cell
model = AutoModelForSequenceClassification.from_pretrained("bert-base-cased", num_labels=2)
```

```diff
+ def training_function(model):
      # Initialize accelerator
      accelerator = Accelerator()
-     model = AutoModelForSequenceClassification.from_pretrained("bert-base-cased", num_labels=2)
      train_dataloader, eval_dataloader = create_dataloaders(
          train_batch_size=hyperparameters["train_batch_size"], eval_batch_size=hyperparameters["eval_ba
      )
  ...
```

And finally calling the training function with:

```diff
  from accelerate import notebook_launcher
- notebook_launcher(training_function)
+ notebook_launcher(training_function, (model,))
```

<Tip>

    The above workaround is only needed when launching a TPU instance from a Jupyter Notebook on a low-r
    using a script or launching on a much beefier server declaring the model beforehand is not needed.

</Tip>

## Mixed Precision and Global Variables

As mentioned in the [mixed precision tutorial](../usage_guides/mixed_precision), ■ Accelerate supports f
That being said, ideally `bf16` should be utilized as it is extremely efficient to use.

There are two "layers" when using `bf16` and ■ Accelerate on TPUs, at the base level and at the operatio

At the base level, this is enabled when passing `mixed_precision="bf16"` to `Accelerator`, such as:
```python
accelerator = Accelerator(mixed_precision="bf16")
```
By default this will cast `torch.float` and `torch.double` to `bfloat16` on TPUs.
The specific configuration being set is an environmental variable of `XLA_USE_BF16` is set to `1`.

There is a further configuration you can perform which is setting the `XLA_DOWNCAST_BF16` environmental
`torch.float` is `bfloat16` and `torch.double` is `float32`.

This is performed in the `Accelerator` object when passing `downcast_bf16=True`:
```python
accelerator = Accelerator(mixed_precision="bf16", downcast_bf16=True)
```

Using downcasting instead of bf16 everywhere is good for when you are trying to calculate metrics, log v

## Training Times on TPUs

As you launch your script, you may notice that training seems exceptionally slow at first. This is becau
first run through a few batches of data to see how much memory to allocate before finally utilizing this
memory allocation extremely efficiently.

If you notice that your evaluation code to calculate the metrics of your model takes longer due to a lar
it is recommended to keep the batch size the same as the training data if it is too slow. Otherwise the
new batch size after the first few iterations.

<Tip>

    Just because the memory is allocated does not mean it will be used or that the batch size will incre

</Tip>

accelerate-main/docs/source/concept_guides/performance.mdx

## Comparing performance between different device setups

Evaluating and comparing the performance from different setups can be quite tricky if you don't know wha
For example, you cannot run the same script with the same batch size across TPU, multi-GPU, and single-G
and expect your results to line up.

But why?

There's three reasons for this that this tutorial will cover:

1. **Setting the right seeds**
2. **Observed Batch Sizes**
3. **Learning Rates**

## Setting the Seed

While this issue has not come up as much, make sure to use [`utils.set_seed`] to fully set the seed in a

```python
from accelerate.utils import set_seed

set_seed(42)
```

Why is this important? Under the hood this will set **5** different seed settings:

```python
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    # ^^ safe to call this function even if cuda is not available
    if is_tpu_available():
        xm.set_rng_state(seed)
```

The random state, numpy's state, torch, torch's cuda state, and if TPUs are available torch_xla's cuda s

## Observed Batch Sizes

When training with Accelerate, the batch size passed to the dataloader is the **batch size per GPU**. Wh
a batch size of 64 on two GPUs is truly a batch size of 128. As a result, when testing on a single GPU t
as well as similarly for TPUs.

The below table can be used as a quick reference to try out different batch sizes:

<Tip>

In this example there are two GPUs for "Multi-GPU" and a TPU pod with 8 workers

</Tip>

Single GPU Batch Size	Multi-GPU Equivalent Batch Size	TPU Equivalent Batch Size
256	128	32
128	64	16

64	32	8
32	16	4

## Learning Rates

As noted in multiple sources[[1](https://aws.amazon.com/blogs/machine-learning/scalable-multi-node-deep-
snippet shows doing so with Accelerate:

<Tip>

Since users can have their own learning rate schedulers defined, we leave this up to the user to decide
learning rate or not.

</Tip>

```python
learning_rate = 1e-3
accelerator = Accelerator()
learning_rate *= accelerator.num_processes

optimizer = AdamW(params=model.parameters(), lr=learning_rate)
```

You will also find that `accelerate` will step the learning rate based on the number of processes being
of the observed batch size noted earlier. So in a case of 2 GPUs, the learning rate will be stepped twic
to account for the batch size being twice as large (if no changes to the batch size on the single GPU in

accelerate-main/docs/source/concept_guides/gradient_synchronization.mdx

~~~
<!--Copyright 2022 The HuggingFace Team. All rights reserved.

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in comp
the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is dis
an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the Licen
specific language governing permissions and limitations under the License.
-->

# Gradient Synchronization

PyTorch's distributed module operates by communicating back and forth between all of the GPUs in your sy
This communication takes time, and ensuring all processes know the states of each other happens at parti
when using the `ddp` module.

These triggerpoints are added to the PyTorch model, specifically their `forward()` and `backward()` meth
This happens when the model is wrapped with `DistributedDataParallel`:
```python
import torch.nn as nn
from torch.nn.parallel import DistributedDataParallel

model = nn.Linear(10, 10)
ddp_model = DistributedDataParallel(model)
```
In ■ Accelerate this conversion happens automatically when calling [`~Accelerator.prepare`] and passing

```diff
+ from accelerate import Accelerator
+ accelerator = Accelerator()
  import torch.nn as nn
- from torch.nn.parallel import DistributedDataParallel

  model = nn.Linear(10,10)
+ model = accelerator.prepare(model)
```

## The slowdown in gradient accumulation

You now understand that PyTorch adds hooks to the `forward` and `backward` method of your PyTorch model
training in a distributed setup. But how does this risk slowing down your code?

In DDP (distributed data parallel), the specific order in which processes are performed and ran are expe
at specific points and these must also occur at roughly the same time before moving on.

The most direct example is when you update all of the parameters in a model through `.backward()`. All i
need to have updated their gradients, collated, and updated again before moving onto the next batch of d
gradient accumulation, you accumulate `n` losses and skip `.backward()` until `n` batches have been reac
can cause a significant slowdown since all the processes need to communicate with them more times than n
can you avoid this overhead?

## Solving the slowdown problem

Since you are skipping these batches, their gradients do not need to be synchronized until the point whe
PyTorch cannot automagically tell when you need to do this, but they do provide a tool to help through t
that is added to your model after converting it to DDP.

Under this context manager, PyTorch will skip synchronizing the gradients when `.backward()` is called,
context manager will trigger the synchronization. See an example below:
```python
ddp_model, dataloader = accelerator.prepare(model, dataloader)

for index, batch in enumerate(dataloader):
    inputs, targets = batch
~~~

~~~
    # Trigger gradient synchronization on the last batch
    if index != (len(dataloader) - 1):
        with ddp_model.no_sync():
            # Gradients only accumulate
            outputs = ddp_model(inputs)
            loss = loss_func(outputs)
            accelerator.backward(loss)
    else:
        # Gradients finally sync
        outputs = ddp_model(inputs)
        loss = loss_func(outputs)
        accelerator.backward(loss)
```

In ■ Accelerate to make this an API that can be called no matter the training device (though it may not
`ddp_model.no_sync` gets replaced with [`~Accelerator.no_sync`] and operates the same way:

```diff
  ddp_model, dataloader = accelerator.prepare(model, dataloader)

  for index, batch in enumerate(dataloader):
      inputs, targets = batch
      # Trigger gradient synchronization on the last batch
      if index != (len(dataloader)-1):
-         with ddp_model.no_sync():
+         with accelerator.no_sync(model):
              # Gradients only accumulate
              outputs = ddp_model(inputs)
              loss = loss_func(outputs, targets)
              accelerator.backward(loss)
      else:
          # Gradients finally sync
          outputs = ddp_model(inputs)
          loss = loss_func(outputs)
          accelerator.backward(loss)
```

As you may expect, the [`~Accelerator.accumulate`] function wraps around this conditional check by keepi
gradient accumulation API:

```python
ddp_model, dataloader = accelerator.prepare(model, dataloader)

for batch in dataloader:
    with accelerator.accumulate(model):
        optimizer.zero_grad()
        inputs, targets = batch
        outputs = model(inputs)
        loss = loss_function(outputs, targets)
        accelerator.backward(loss)
```

As a result, you should either use *`accelerator.accumulate` or `accelerator.no_sync`* when it comes to

## Just how much of a slowdown is there, and easy mistakes you can make

To setup a realistic example, consider the following setup:

* Two single-GPU T4 nodes and one node with two GPUs
* Each GPU is a T4, and are hosted on GCP
* The script used is a modification of the [NLP Example](https://github.com/muellerzr/timing_experiments
* Batch size per GPU is 16, and gradients are accumulated every 4 steps

All scripts are available in [this repository](https://github.com/muellerzr/timing_experiments).

If not careful about gradient synchronization and GPU communication, a *large* amount of time can be was
from when these GPUs communicate to each other during unnessisary periods.

By how much?

Reference:
- Baseline: uses no synchronization practices discussed here
~~~

- `no_sync` improperly: `no_sync` only around the `backward` call, not the `forward`
- `no_sync`: using the `no_sync` pattern properly
- `accumulate`: using [`~Accelerator.accumulate`] properly

Below are the average seconds per batch iterating over 29 batches of data for each setup on both a singl

	Baseline	`no_sync` improperly	`no_sync`	`accumulate`
Multi-Node	2±0.01s	2.13±0.08s	**0.91±0.11s**	**0.91±0.11s**
Single Node	0.50±0.01s	0.50±0.01s	**0.41±0.015s**	**0.41±0.015s**

As you can see, if you are not careful about how you setup your gradient synchronization, you can get up

If you are worried about making sure everything is done properly, we highly recommend utilizing the [`~A
`gradient_accumulation_steps` to the [`Accelerator`] object so Accelerate can handle this for you.

accelerate-main/docs/source/concept_guides/deferring_execution.mdx

~~~~
<!--Copyright 2022 The HuggingFace Team. All rights reserved.

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in comp
the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is dis
an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the Licen
specific language governing permissions and limitations under the License.
-->

# Deferring Executions

When you run your usual script, instructions are executed in order. Using ■ Accelerate to deploy your sc
GPUs at the same time introduces a complication: while each process executes all instructions in order,
faster than others.

You might need to wait for all processes to have reached a certain point before executing a given instru
instance, you shouldn't save a model before being sure every process is done with training, and you woul
continue training before all the model weights have been loaded in. To do this, just write the following

```
accelerator.wait_for_everyone()
```

This instruction will block all the processes that arrive first until all the other processes have reach
point (if you run your script on just one GPU or CPU, this won't do anything).

A few example cases for when to use this utility are listed below:

<Tip>

    Some of these are utilized with the [`~Accelerator.main_process_first`] context manager, which utili
    run a particular set of code on the main process beforehand before triggering and launching the othe

</Tip>

## Downloading a Dataset

When downloading a dataset, you should download it first on the main process and then loading the cached

<Tip>

    `load_dataset` will perform a lock under the hood to stop multiple downloads from happening at once,
    not using this library you should use this method.

</Tip>

```python
with accelerator.main_process_first():
    datasets = load_dataset("glue", "mrpc")
```

Under the hood this is the same as calling:

```python
# First do something on the main process
if accelerator.is_main_process:
    datasets = load_dataset("glue", "mrpc")
else:
    accelerator.wait_for_everyone()

# And then send it to the rest of them
if not accelerator.is_main_process:
    datasets = load_dataset("glue", "mrpc")
~~~~

```
else:
    accelerator.wait_for_everyone()
```

## Saving the `state_dict`

When saving the `state_dict` of the model, since you would normally save one file on just the main proce
you should specify that:

```python
if accelerator.is_main_process:
    model = accelerator.unwrap_model(model)
    torch.save(model.state_dict(), "weights.pth")
```

## Loading in the `state_dict`

When loading in the `state_dict` to a model, optimizer, or scheduler, you should wait
for all workers to have the weights loaded in before moving on to training

```python
with accelerator.main_process_first():
    state = torch.load("weights.pth")
    model.load_state_dict(state)
```

## Applying a multi-worker CPU operation

Applying a `map()` operation on multiple workers, such as tokenizing should be done on the
main process first, and then propagated to each one.

```python
datasets = load_dataset("glue", "mrpc")

with accelerator.main_process_first():
    tokenized_datasets = datasets.map(
        tokenize_function,
        batched=True,
        remove_columns=["idx", "sentence1", "sentence2"],
    )
```

accelerate-main/examples/by_feature/tracking.py

```
# coding=utf-8
# Copyright 2021 The HuggingFace Inc. team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
import argparse
import os

import evaluate
import torch
from datasets import load_dataset
from torch.optim import AdamW
from torch.utils.data import DataLoader
from transformers import AutoModelForSequenceClassification, AutoTokenizer, get_linear_schedule_with_war

from accelerate import Accelerator, DistributedType


########################################################################
# This is a fully working simple example to use Accelerate,
# specifically showcasing the experiment tracking capability,
# and builds off the `nlp_example.py` script.
#
# This example trains a Bert base model on GLUE MRPC
# in any of the following settings (with the same script):
#   - single CPU or single GPU
#   - multi GPUS (using PyTorch distributed mode)
#   - (multi) TPUs
#   - fp16 (mixed-precision) or fp32 (normal precision)
#
# To help focus on the differences in the code, building `DataLoaders`
# was refactored into its own function.
# New additions from the base script can be found quickly by
# looking for the # New Code # tags
#
# To run it in each of these various modes, follow the instructions
# in the readme for examples:
# https://github.com/huggingface/accelerate/tree/main/examples
#
########################################################################

MAX_GPU_BATCH_SIZE = 16
EVAL_BATCH_SIZE = 32


def get_dataloaders(accelerator: Accelerator, batch_size: int = 16):
    """
    Creates a set of `DataLoader`s for the `glue` dataset,
    using "bert-base-cased" as the tokenizer.

    Args:
        accelerator (`Accelerator`):
            An `Accelerator` object
        batch_size (`int`, *optional*):
            The batch size for the train and validation DataLoaders.
    """
    tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
    datasets = load_dataset("glue", "mrpc")
    def tokenize_function(examples):
        # max_length=None => use the model max length (it's actually the default)
        outputs = tokenizer(examples["sentence1"], examples["sentence2"], truncation=True, max_length=No
        return outputs

    # Apply the method we just defined to all the examples in all the splits of the dataset
    # starting with the main process first:
    with accelerator.main_process_first():
        tokenized_datasets = datasets.map(
            tokenize_function,
            batched=True,
            remove_columns=["idx", "sentence1", "sentence2"],
        )

    # We also rename the 'label' column to 'labels' which is the expected name for labels by the models
    # transformers library
    tokenized_datasets = tokenized_datasets.rename_column("label", "labels")

    def collate_fn(examples):
        # On TPU it's best to pad everything to the same length or training will be very slow.
        max_length = 128 if accelerator.distributed_type == DistributedType.TPU else None
        # When using mixed precision we want round multiples of 8/16
        if accelerator.mixed_precision == "fp8":
            pad_to_multiple_of = 16
        elif accelerator.mixed_precision != "no":
            pad_to_multiple_of = 8
        else:
            pad_to_multiple_of = None

        return tokenizer.pad(
            examples,
            padding="longest",
            max_length=max_length,
            pad_to_multiple_of=pad_to_multiple_of,
            return_tensors="pt",
        )

    # Instantiate dataloaders.
    train_dataloader = DataLoader(
        tokenized_datasets["train"], shuffle=True, collate_fn=collate_fn, batch_size=batch_size
    )
    eval_dataloader = DataLoader(
        tokenized_datasets["validation"], shuffle=False, collate_fn=collate_fn, batch_size=EVAL_BATCH_SI
    )

    return train_dataloader, eval_dataloader


# For testing only
if os.environ.get("TESTING_MOCKED_DATALOADERS", None) == "1":
    from accelerate.test_utils.training import mocked_dataloaders

    get_dataloaders = mocked_dataloaders  # noqa: F811


def training_function(config, args):
    # For testing only
    if os.environ.get("TESTING_MOCKED_DATALOADERS", None) == "1":
        config["num_epochs"] = 2
    # Initialize Accelerator

    # New Code #
    # We pass in "all" to `log_with` to grab all available trackers in the environment
    # Note: If using a custom `Tracker` class, should be passed in here such as:
    # >>> log_with = ["all", MyCustomTrackerClassInstance()]
    if args.with_tracking:
        accelerator = Accelerator(
            cpu=args.cpu, mixed_precision=args.mixed_precision, log_with="all", logging_dir=args.logging
        )
    else:
        accelerator = Accelerator(cpu=args.cpu, mixed_precision=args.mixed_precision)
    # Sample hyper-parameters for learning rate, batch size, seed and a few other HPs
    lr = config["lr"]
```

```
num_epochs = int(config["num_epochs"])
seed = int(config["seed"])
batch_size = int(config["batch_size"])
set_seed(seed)

train_dataloader, eval_dataloader = get_dataloaders(accelerator, batch_size)
metric = evaluate.load("glue", "mrpc")

# If the batch size is too big we use gradient accumulation
gradient_accumulation_steps = 1
if batch_size > MAX_GPU_BATCH_SIZE and accelerator.distributed_type != DistributedType.TPU:
    gradient_accumulation_steps = batch_size // MAX_GPU_BATCH_SIZE
    batch_size = MAX_GPU_BATCH_SIZE

# Instantiate the model (we build the model here so that the seed also control new weights initializ
model = AutoModelForSequenceClassification.from_pretrained("bert-base-cased", return_dict=True)

# We could avoid this line since the accelerator is set with `device_placement=True` (default value)
# Note that if you are placing tensors on devices manually, this line absolutely needs to be before
# creation otherwise training will not work on TPU (`accelerate` will kindly throw an error to make
model = model.to(accelerator.device)

# Instantiate optimizer
optimizer = AdamW(params=model.parameters(), lr=lr)

# Instantiate scheduler
lr_scheduler = get_linear_schedule_with_warmup(
    optimizer=optimizer,
    num_warmup_steps=100,
    num_training_steps=(len(train_dataloader) * num_epochs) // gradient_accumulation_steps,
)

# Prepare everything
# There is no specific order to remember, we just need to unpack the objects in the same order we ga
# prepare method.
model, optimizer, train_dataloader, eval_dataloader, lr_scheduler = accelerator.prepare(
    model, optimizer, train_dataloader, eval_dataloader, lr_scheduler
)

# New Code #
# We need to initialize the trackers we use. Overall configurations can also be stored
if args.with_tracking:
    run = os.path.split(__file__)[-1].split(".")[0]
    accelerator.init_trackers(run, config)

# Now we train the model
for epoch in range(num_epochs):
    model.train()
    # New Code #
    # For our tracking example, we will log the total loss of each epoch
    if args.with_tracking:
        total_loss = 0
    for step, batch in enumerate(train_dataloader):
        # We could avoid this line since we set the accelerator with `device_placement=True`.
        batch.to(accelerator.device)
        outputs = model(**batch)
        loss = outputs.loss
        # New Code #
        if args.with_tracking:
            total_loss += loss.detach().float()
        loss = loss / gradient_accumulation_steps
        accelerator.backward(loss)
        if step % gradient_accumulation_steps == 0:
            optimizer.step()
            lr_scheduler.step()
            optimizer.zero_grad()

    model.eval()
    for step, batch in enumerate(eval_dataloader):
        # We could avoid this line since we set the accelerator with `device_placement=True` (the de
        batch.to(accelerator.device)
        with torch.no_grad():
                outputs = model(**batch)
            predictions = outputs.logits.argmax(dim=-1)
            predictions, references = accelerator.gather_for_metrics((predictions, batch["labels"]))
            metric.add_batch(
                predictions=predictions,
                references=references,
            )

        eval_metric = metric.compute()
        # Use accelerator.print to print only on the main process.
        accelerator.print(f"epoch {epoch}:", eval_metric)

        # New Code #
        # To actually log, we call `Accelerator.log`
        # The values passed can be of `str`, `int`, `float` or `dict` of `str` to `float`/`int`
        if args.with_tracking:
            accelerator.log(
                {
                    "accuracy": eval_metric["accuracy"],
                    "f1": eval_metric["f1"],
                    "train_loss": total_loss.item() / len(train_dataloader),
                    "epoch": epoch,
                },
                step=epoch,
            )

    # New Code #
    # When a run is finished, you should call `accelerator.end_training()`
    # to close all of the open trackers
    if args.with_tracking:
        accelerator.end_training()


def main():
    parser = argparse.ArgumentParser(description="Simple example of training script.")
    parser.add_argument(
        "--mixed_precision",
        type=str,
        default=None,
        choices=["no", "fp16", "bf16", "fp8"],
        help="Whether to use mixed precision. Choose"
        "between fp16 and bf16 (bfloat16). Bf16 requires PyTorch >= 1.10."
        "and an Nvidia Ampere GPU.",
    )
    parser.add_argument("--cpu", action="store_true", help="If passed, will train on the CPU.")
    parser.add_argument(
        "--with_tracking",
        action="store_true",
        help="Whether to load in all available experiment trackers from the environment and use them for
    )
    parser.add_argument(
        "--logging_dir",
        type=str,
        default="logs",
        help="Location on where to store experiment tracking logs`",
    )
    args = parser.parse_args()
    config = {"lr": 2e-5, "num_epochs": 3, "seed": 42, "batch_size": 16}
    training_function(config, args)


if __name__ == "__main__":
    main()
```

accelerate-main/docs/source/package_reference/deepspeed.mdx

```
<!--Copyright 2021 The HuggingFace Team. All rights reserved.

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in comp
the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is dis
an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the Licen
specific language governing permissions and limitations under the License.
-->

# Utilities for DeepSpeed

[[autodoc]] utils.DeepSpeedPlugin

[[autodoc]] utils.DummyOptim

[[autodoc]] utils.DummyScheduler

[[autodoc]] utils.DeepSpeedEngineWrapper

[[autodoc]] utils.DeepSpeedOptimizerWrapper

[[autodoc]] utils.DeepSpeedSchedulerWrapper
```

accelerate-main/docs/source/usage_guides/fsdp.mdx

# Fully Sharded Data Parallel

To accelerate training huge models on larger batch sizes, we can use a fully sharded data parallel model
This type of data parallel paradigm enables fitting more data and larger models by sharding the optimize
To read more about it and the benefits, check out the [Fully Sharded Data Parallel blog](https://pytorch
We have integrated the latest PyTorch's Fully Sharded Data Parallel (FSDP) training feature.
All you need to do is enable it through the config.

## How it works out of the box

On your machine(s) just run:

```bash
accelerate config
```

and answer the questions asked. This will generate a config file that will be used automatically to prop
default options when doing

```bash
accelerate launch my_script.py --args_to_my_script
```

For instance, here is how you would run the NLP example (from the root of the repo) with FSDP enabled:

```bash
compute_environment: LOCAL_MACHINE
deepspeed_config: {}
distributed_type: FSDP
downcast_bf16: 'no'
fsdp_config:
  fsdp_auto_wrap_policy: TRANSFORMER_BASED_WRAP
  fsdp_backward_prefetch_policy: BACKWARD_PRE
  fsdp_offload_params: false
  fsdp_sharding_strategy: 1
  fsdp_state_dict_type: FULL_STATE_DICT
  fsdp_transformer_layer_cls_to_wrap: GPT2Block
machine_rank: 0
main_process_ip: null
main_process_port: null
main_training_function: main
mixed_precision: 'no'
num_machines: 1
num_processes: 2
use_cpu: false
```

```bash
accelerate launch examples/nlp_example.py
```

Currently, `Accelerate` supports the following config through the CLI:

```bash
`Sharding Strategy`: [1] FULL_SHARD (shards optimizer states, gradients and parameters), [2] SHARD_GRAD_
`Offload Params`: Decides Whether to offload parameters and gradients to CPU
```

`Auto Wrap Policy`: [1] TRANSFORMER_BASED_WRAP, [2] SIZE_BASED_WRAP, [3] NO_WRAP
`Transformer Layer Class to Wrap`: When using `TRANSFORMER_BASED_WRAP`, user specifies comma-separated s
`BertLayer`, `GPTJBlock`, `T5Block`, `BertLayer,BertEmbeddings,BertSelfOutput`...
`Min Num Params`: minimum number of parameters when using `SIZE_BASED_WRAP`
`Backward Prefetch`: [1] BACKWARD_PRE, [2] BACKWARD_POST, [3] NO_PREFETCH
`State Dict Type`: [1] FULL_STATE_DICT, [2] LOCAL_STATE_DICT, [3] SHARDED_STATE_DICT

## Saving and loading

1. When using transformers `save_pretrained`, pass `state_dict=accelerator.get_state_dict(model)` to sav
  Below is an example:

```diff
  unwrapped_model.save_pretrained(
      args.output_dir,
      is_main_process=accelerator.is_main_process,
      save_function=accelerator.save,
+     state_dict=accelerator.get_state_dict(model),
)
```

### State Dict

`accelerator.get_state_dict` will call the underlying `model.state_dict` implementation. With a model w

To avoid this, PyTorch provides a context manager that adjusts the behavior of `state_dict`. To offload

```
from torch.distributed.fsdp import FullyShardedDataParallel as FSDP, StateDictType, FullStateDictConfig

full_state_dict_config = FullStateDictConfig(offload_to_cpu=True, rank0_only=True)
with FSDP.state_dict_type(unwrapped_model, StateDictType.FULL_STATE_DICT, full_state_dict_config):
    state = accelerator.get_state_dict(unwrapped_model)
```

You can then pass `state` into the `save_pretrained` method. There are several modes for `StateDictType

## A few caveats to be aware of

- PyTorch FSDP auto wraps sub-modules, flattens the parameters and shards the parameters in place.
  Due to this, any optimizer created before model wrapping gets broken and occupies more memory.
  Hence, it is highly recommended and efficient to prepare the model before creating the optimizer.
  `Accelerate` will automatically wrap the model and create an optimizer for you in case of single model
  > FSDP Warning: When using FSDP, it is efficient and recommended to call prepare for the model before

However, below is the recommended way to prepare model and optimizer while using FSDP:

```diff
  model = AutoModelForSequenceClassification.from_pretrained("bert-base-cased", return_dict=True)
+ model = accelerator.prepare(model)

  optimizer = torch.optim.AdamW(params=model.parameters(), lr=lr)

- model, optimizer, train_dataloader, eval_dataloader, lr_scheduler = accelerator.prepare(
-        model, optimizer, train_dataloader, eval_dataloader, lr_scheduler
-    )

+ optimizer, train_dataloader, eval_dataloader, lr_scheduler = accelerator.prepare(
+         optimizer, train_dataloader, eval_dataloader, lr_scheduler
+    )
```

- In case of a single model, if you have created the optimizer with multiple parameter groups and called
  then the parameter groups will be lost and the following warning is displayed:
  > FSDP Warning: When using FSDP, several parameter groups will be conflated into
  > a single one due to nested module wrapping and parameter flattening.

  This is because parameter groups created before wrapping will have no meaning post wrapping due to par
  For instance, below are the named parameters of an FSDP model on GPU 0 (When using 2 GPUs. Around 55M
  Here, if one has applied no weight decay for [bias, LayerNorm.weight] the named parameters of an unwra
  it can't be applied to the below FSDP wrapped model as there are no named parameters with either of th

  the parameters of those layers are concatenated with parameters of various other layers.
  ```
  {
    '_fsdp_wrapped_module.flat_param': torch.Size([494209]),
    '_fsdp_wrapped_module._fpw_module.bert.embeddings.word_embeddings._fsdp_wrapped_module.flat_param':
    '_fsdp_wrapped_module._fpw_module.bert.encoder._fsdp_wrapped_module.flat_param': torch.Size([4252723
  }
  ```

- In case of multiple models, it is necessary to prepare the models before creating optimizers or else i
Then pass the optimizers to the prepare call in the same order as corresponding models else `accelerator
- This feature is incompatible with `--predict_with_generate` in the `run_translation.py` script of ■ `T

For more control, users can leverage the `FullyShardedDataParallelPlugin`. After creating an instance of
For more information on these options, please refer to the PyTorch [FullyShardedDataParallel](https://gi

accelerate-main/docs/source/usage_guides/training_zoo.mdx

```
# Example Zoo

Below contains a non-exhuastive list of tutorials and scripts showcasing Accelerate

## Official Accelerate Examples:

### Basic Examples

These examples showcase the base features of Accelerate and are a great starting point

- [Barebones NLP example](https://github.com/huggingface/accelerate/blob/main/examples/nlp_example.py)
- [Barebones distributed NLP example in a Jupyter Notebook](https://github.com/huggingface/notebooks/blo
- [Barebones computer vision example](https://github.com/huggingface/accelerate/blob/main/examples/cv_ex
- [Barebones distributed computer vision example in a Jupyter Notebook](https://github.com/huggingface/n
- [Using Accelerate in Kaggle](https://www.kaggle.com/code/muellerzr/multi-gpu-and-accelerate)

### Feature Specific Examples

These examples showcase specific features that the Accelerate framework offers

- [Automatic memory-aware gradient accumulation](https://github.com/huggingface/accelerate/blob/main/exa
- [Checkpointing states](https://github.com/huggingface/accelerate/blob/main/examples/by_feature/checkpo
- [Cross validation](https://github.com/huggingface/accelerate/blob/main/examples/by_feature/cross_valid
- [DeepSpeed](https://github.com/huggingface/accelerate/blob/main/examples/by_feature/deepspeed_with_con
- [Fully Sharded Data Parallelism](https://github.com/huggingface/accelerate/blob/main/examples/by_featu
- [Gradient accumulation](https://github.com/huggingface/accelerate/blob/main/examples/by_feature/gradie
- [Memory-aware batch size finder](https://github.com/huggingface/accelerate/blob/main/examples/by_featu
- [Metric Computation](https://github.com/huggingface/accelerate/blob/main/examples/by_feature/multi_pro
- [Using Trackers](https://github.com/huggingface/accelerate/blob/main/examples/by_feature/tracking.py)
- [Using Megatron-LM](https://github.com/huggingface/accelerate/blob/main/examples/by_feature/megatron_l

### Full Examples

These examples showcase every feature in Accelerate at once that was shown in "Feature Specific Examples

- [Complete NLP example](https://github.com/huggingface/accelerate/blob/main/examples/complete_nlp_examp
- [Complete computer vision example](https://github.com/huggingface/accelerate/blob/main/examples/comple
- [Causal language model fine-tuning example](https://github.com/huggingface/transformers/blob/main/exam
- [Masked language model fine-tuning example](https://github.com/huggingface/transformers/blob/main/exam
- [Speech pretraining example](https://github.com/huggingface/transformers/blob/main/examples/pytorch/sp
- [Translation fine-tuning example](https://github.com/huggingface/transformers/blob/main/examples/pytor
- [Text classification fine-tuning example](https://github.com/huggingface/transformers/blob/main/exampl
- [Semantic segmentation fine-tuning example](https://github.com/huggingface/transformers/blob/main/exam
- [Question answering fine-tuning example](https://github.com/huggingface/transformers/blob/main/example
- [Beam search question answering fine-tuning example](https://github.com/huggingface/transformers/blob/
- [Multiple choice question answering fine-tuning example](https://github.com/huggingface/transformers/b
- [Named entity recognition fine-tuning example](https://github.com/huggingface/transformers/blob/main/e
- [Image classification fine-tuning example](https://github.com/huggingface/transformers/blob/main/examp
- [Summarization fine-tuning example](https://github.com/huggingface/transformers/blob/main/examples/pyt
- [End-to-end examples on how to use AWS SageMaker integration of Accelerate](https://github.com/hugging
- [Megatron-LM examples for various NLp tasks](https://github.com/pacman100/accelerate-megatron-test)

## Integration Examples

These are tutorials from libraries that integrate with ■ Accelerate:
### Catalyst
```

- [Distributed training tutorial with Catalyst](https://catalyst-team.github.io/catalyst/tutorials/ddp.h

### DALLE2-pytorch

- [Fine-tuning DALLE2](https://github.com/lucidrains/DALLE2-pytorch#usage)

### ■ diffusers

- [Performing textual inversion with diffusers](https://github.com/huggingface/diffusers/tree/main/examp
- [Training DreamBooth with diffusers](https://github.com/huggingface/diffusers/tree/main/examples/dream

### fastai

- [Distributed training from Jupyter Notebooks with fastai](https://docs.fast.ai/tutorial.distributed.ht
- [Basic distributed training examples with fastai](https://docs.fast.ai/examples/distributed_app_exampl

### GradsFlow

- [Auto Image Classification with GradsFlow](https://docs.gradsflow.com/en/latest/examples/nbs/01-ImageC

### imagen-pytorch

- [Fine-tuning Imagen](https://github.com/lucidrains/imagen-pytorch#usage)

### Kornia

- [Fine-tuning vision models with Kornia's Trainer](https://kornia.readthedocs.io/en/latest/get-started/

### PyTorch Accelerated

- [Quickstart distributed training tutorial with PyTorch Accelerated](https://pytorch-accelerated.readth

### PyTorch3D

- [Perform Deep Learning with 3D data](https://pytorch3d.org/tutorials/)

### Stable-Dreamfusion

- [Training with Stable-Dreamfusion to convert text to a 3D model](https://colab.research.google.com/dri

### Tez

- [Leaf disease detection with Tez and Accelerate](https://www.kaggle.com/code/abhishek/tez-faster-and-e

### trlx

- [How to implement a sentiment learning task with trlx](https://github.com/CarperAI/trlx#example-how-to

accelerate-main/docs/source/package_reference/big_modeling.mdx

```
<!--Copyright 2021 The HuggingFace Team. All rights reserved.

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in comp
the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is dis
an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the Licen
specific language governing permissions and limitations under the License.
-->

# Working with large models

## Dispatching and Offloading Models

[[autodoc]] big_modeling.init_empty_weights
[[autodoc]] big_modeling.cpu_offload
[[autodoc]] big_modeling.disk_offload
[[autodoc]] big_modeling.dispatch_model
[[autodoc]] big_modeling.load_checkpoint_and_dispatch

## Model Hooks

### Hook Classes

[[autodoc]] hooks.ModelHook
[[autodoc]] hooks.AlignDevicesHook
[[autodoc]] hooks.SequentialHook

### Adding Hooks

[[autodoc]] hooks.add_hook_to_module
[[autodoc]] hooks.attach_execution_device_hook
[[autodoc]] hooks.attach_align_device_hook
[[autodoc]] hooks.attach_align_device_hook_on_blocks

### Removing Hooks

[[autodoc]] hooks.remove_hook_from_module
[[autodoc]] hooks.remove_hook_from_submodules
```

accelerate-main/docs/source/usage_guides/sagemaker.mdx

# Amazon SageMaker

Hugging Face and Amazon introduced new [Hugging Face Deep Learning Containers (DLCs)](https://github.com
make it easier than ever to train Hugging Face Transformer models in [Amazon SageMaker](https://aws.amaz

## Getting Started

### Setup & Installation

Before you can run your ■ Accelerate scripts on Amazon SageMaker you need to sign up for an AWS account.
have an AWS account yet learn more [here](https://docs.aws.amazon.com/sagemaker/latest/dg/gs-set-up.html

After you have your AWS Account you need to install the `sagemaker` sdk for ■ Accelerate with:

```bash
pip install "accelerate[sagemaker]" --upgrade
```

■ Accelerate currently uses the ■ DLCs, with `transformers`, `datasets` and `tokenizers` pre-installed.
Accelerate is not in the DLC yet (will soon be added!) so to use it within Amazon SageMaker you need to
`requirements.txt` in the same directory where your training script is located and add it as dependency:

```
accelerate
```

You should also add any other dependencies you have to this `requirements.txt`.

### Configure ■ Accelerate

You can configure the launch configuration for Amazon SageMaker the same as you do for non SageMaker tra
the ■ Accelerate CLI:

```bash
accelerate config
# In which compute environment are you running? ([0] This machine, [1] AWS (Amazon SageMaker)): 1
```

■ Accelerate will go through a questionnaire about your Amazon SageMaker setup and create a config file

<Tip>

    ■ Accelerate is not saving any of your credentials.

</Tip>

### Prepare a ■ Accelerate fine-tuning script

The training script is very similar to a training script you might run outside of SageMaker, but to save
after training you need to specify either `/opt/ml/model` or use `os.environ["SM_MODEL_DIR"]` as your sa
directory. After training, artifacts in this directory are uploaded to S3:

```diff
```

- torch.save('/opt/ml/model`)
+ accelerator.save('/opt/ml/model')
```

<Tip warning={true}>

    SageMaker doesn’t support argparse actions. If you want to use, for example, boolean hyperparameters
    specify type as bool in your script and provide an explicit True or False value for this hyperparame

</Tip>

### Launch Training

You can launch your training with ■ Accelerate CLI with:

```
accelerate launch path_to_script.py --args_to_the_script

This will launch your training script using your configuration. The only thing you have to do is provide
arguments needed by your training script as named arguments.

Examples

<Tip>

    If you run one of the example scripts, don't forget to add `accelerator.save('/opt/ml/model')` to it

</Tip>

```bash
accelerate launch ./examples/sagemaker_example.py
```

Outputs:

```
Configuring Amazon SageMaker environment
Converting Arguments to Hyperparameters
Creating Estimator
2021-04-08 11:56:50 Starting - Starting the training job...
2021-04-08 11:57:13 Starting - Launching requested ML instancesProfilerReport-1617883008: InProgress
.........
2021-04-08 11:58:54 Starting - Preparing the instances for training.........
2021-04-08 12:00:24 Downloading - Downloading input data
2021-04-08 12:00:24 Training - Downloading the training image..................
2021-04-08 12:03:39 Training - Training image download completed. Training in progress..
........
epoch 0: {'accuracy': 0.7598039215686274, 'f1': 0.8178438661710037}
epoch 1: {'accuracy': 0.8357843137254902, 'f1': 0.882249560632689}
epoch 2: {'accuracy': 0.8406862745098039, 'f1': 0.8869565217391304}
........
2021-04-08 12:05:40 Uploading - Uploading generated training model
2021-04-08 12:05:40 Completed - Training job completed
Training seconds: 331
Billable seconds: 331
You can find your model data at: s3://your-bucket/accelerate-sagemaker-1-2021-04-08-11-56-47-108/output/
```

## Advanced Features

### Distributed Training: Data Parallelism

Set up the accelerate config by running `accelerate config` and answer the SageMaker questions and set i
To use SageMaker DDP, select it when asked
`What is the distributed mode? ([0] No distributed training, [1] data parallelism):`.
Example config below:
```yaml
base_job_name: accelerate-sagemaker-1
compute_environment: AMAZON_SAGEMAKER
distributed_type: DATA_PARALLEL
ec2_instance_type: ml.p3.16xlarge
```

~~~
iam_role_name: xxxxx
image_uri: null
mixed_precision: fp16
num_machines: 1
profile: xxxxx
py_version: py38
pytorch_version: 1.10.2
region: us-east-1
transformers_version: 4.17.0
use_cpu: false
```

### Distributed Training: Model Parallelism

*currently in development, will be supported soon.*

### Python packages and dependencies

■ Accelerate currently uses the ■ DLCs, with `transformers`, `datasets` and `tokenizers` pre-installed.
want to use different/other Python packages you can do this by adding them to the `requirements.txt`. Th
will be installed before your training script is started.

### Local Training: SageMaker Local mode

The local mode in the SageMaker SDK allows you to run your training script locally inside the HuggingFac
or using your custom container image. This is useful for debugging and testing your training script insi
Local mode uses Docker compose (*Note: Docker Compose V2 is not supported yet*). The SDK will handle the
to pull the DLC to your local environment. You can emulate CPU (single and multi-instance) and GPU (sing

To use local mode, you need to set your `ec2_instance_type` to `local`.

```yaml
ec2_instance_type: local
```

### Advanced configuration

The configuration allows you to override parameters for the [Estimator](https://sagemaker.readthedocs.io
These settings have to be applied in the config file and are not part of `accelerate config`. You can co

```yaml
additional_args:
  # enable network isolation to restrict internet access for containers
  enable_network_isolation: True
```

You can find all available configuration [here](https://sagemaker.readthedocs.io/en/stable/api/training/

### Use Spot Instances

You can use Spot Instances e.g. using (see [Advanced configuration](#advanced-configuration)):
```yaml
additional_args:
  use_spot_instances: True
  max_wait: 86400
```

*Note: Spot Instances are subject to be terminated and training to be continued from a checkpoint. This

### Remote scripts: Use scripts located on Github

*undecided if feature is needed. Contact us if you would like this feature.*
~~~

accelerate-main/docs/source/package_reference/megatron_lm.mdx

```
<!--Copyright 2021 The HuggingFace Team. All rights reserved.

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in comp
the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is dis
an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the Licen
specific language governing permissions and limitations under the License.
-->

# Utilities for Megatron-LM

[[autodoc]] utils.MegatronLMPlugin

[[autodoc]] utils.MegatronLMDummyScheduler

[[autodoc]] utils.MegatronLMDummyDataLoader

[[autodoc]] utils.AbstractTrainStep

[[autodoc]] utils.GPTTrainStep

[[autodoc]] utils.BertTrainStep

[[autodoc]] utils.T5TrainStep

[[autodoc]] utils.avg_losses_across_data_parallel_group
```

accelerate-main/docs/source/usage_guides/checkpoint.mdx

# Checkpointing

When training a PyTorch model with ■ Accelerate, you may often want to save and continue a state of trai
saving and loading the model, optimizer, RNG generators, and the GradScaler. Inside ■ Accelerate are two
- Use [`~Accelerator.save_state`] for saving everything mentioned above to a folder location
- Use [`~Accelerator.load_state`] for loading everything stored from an earlier `save_state`

To further customize where and how states saved through [`~Accelerator.save_state`] the [`~utils.Project
if `automatic_checkpoint_naming` is enabled each saved checkpoint will be located then at `Accelerator.p

It should be noted that the expectation is that those states come from the same training script, they sh

- By using [`~Accelerator.register_for_checkpointing`], you can register custom objects to be automatica
so long as the object has a `state_dict` **and** a `load_state_dict` functionality. This could include o

Below is a brief example using checkpointing to save and reload a state during training:

```python
from accelerate import Accelerator
import torch

accelerator = Accelerator(project_dir="my/save/path")

my_scheduler = torch.optim.lr_scheduler.StepLR(my_optimizer, step_size=1, gamma=0.99)
my_model, my_optimizer, my_training_dataloader = accelerator.prepare(my_model, my_optimizer, my_training

# Register the LR scheduler
accelerator.register_for_checkpointing(my_scheduler)

# Save the starting state
accelerator.save_state()

device = accelerator.device
my_model.to(device)

# Perform training
for epoch in range(num_epochs):
    for batch in my_training_dataloader:
        my_optimizer.zero_grad()
        inputs, targets = batch
        inputs = inputs.to(device)
        targets = targets.to(device)
        outputs = my_model(inputs)
        loss = my_loss_function(outputs, targets)
        accelerator.backward(loss)
        my_optimizer.step()
    my_scheduler.step()

# Restore previous state
accelerator.load_state("my/save/path/checkpointing/checkpoint_0")
```

## Restoring the state of the DataLoader

After resuming from a checkpoint, it may also be desireable to resume from a particular point in the act the state was saved during the middle of an epoch. You can use [`~Accelerator.skip_first_batches`] to do

```python
from accelerate import Accelerator

accelerator = Accelerator(project_dir="my/save/path")

train_dataloader = accelerator.prepare(train_dataloader)
accelerator.load_state("my_state")

# Assume the checkpoint was saved 100 steps into the epoch
accelerator.skip_first_batches(train_dataloader, 100)
```

accelerate-main/examples/by_feature/gradient_accumulation.py

```python
# coding=utf-8
# Copyright 2021 The HuggingFace Inc. team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
import argparse
import os

import evaluate
import torch
from datasets import load_dataset
from torch.optim import AdamW
from torch.utils.data import DataLoader
from transformers import AutoModelForSequenceClassification, AutoTokenizer, get_linear_schedule_with_war

from accelerate import Accelerator, DistributedType


########################################################################
# This is a fully working simple example to use Accelerate
# and perform gradient accumulation
#
# This example trains a Bert base model on GLUE MRPC
# in any of the following settings (with the same script):
#   - single CPU or single GPU
#   - multi GPUS (using PyTorch distributed mode)
#   - (multi) TPUs
#   - fp16 (mixed-precision) or fp32 (normal precision)
#
# To run it in each of these various modes, follow the instructions
# in the readme for examples:
# https://github.com/huggingface/accelerate/tree/main/examples
#
########################################################################


MAX_GPU_BATCH_SIZE = 16
EVAL_BATCH_SIZE = 32


def get_dataloaders(accelerator: Accelerator, batch_size: int = 16):
    """
    Creates a set of `DataLoader`s for the `glue` dataset,
    using "bert-base-cased" as the tokenizer.

    Args:
        accelerator (`Accelerator`):
            An `Accelerator` object
        batch_size (`int`, *optional*):
            The batch size for the train and validation DataLoaders.
    """
    tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
    datasets = load_dataset("glue", "mrpc")

    def tokenize_function(examples):
        # max_length=None => use the model max length (it's actually the default)
        outputs = tokenizer(examples["sentence1"], examples["sentence2"], truncation=True, max_length=No
        return outputs
    # Apply the method we just defined to all the examples in all the splits of the dataset
    # starting with the main process first:
    with accelerator.main_process_first():
        tokenized_datasets = datasets.map(
            tokenize_function,
            batched=True,
            remove_columns=["idx", "sentence1", "sentence2"],
        )

    # We also rename the 'label' column to 'labels' which is the expected name for labels by the models
    # transformers library
    tokenized_datasets = tokenized_datasets.rename_column("label", "labels")

    def collate_fn(examples):
        # On TPU it's best to pad everything to the same length or training will be very slow.
        max_length = 128 if accelerator.distributed_type == DistributedType.TPU else None
        # When using mixed precision we want round multiples of 8/16
        if accelerator.mixed_precision == "fp8":
            pad_to_multiple_of = 16
        elif accelerator.mixed_precision != "no":
            pad_to_multiple_of = 8
        else:
            pad_to_multiple_of = None

        return tokenizer.pad(
            examples,
            padding="longest",
            max_length=max_length,
            pad_to_multiple_of=pad_to_multiple_of,
            return_tensors="pt",
        )

    # Instantiate dataloaders.
    train_dataloader = DataLoader(
        tokenized_datasets["train"], shuffle=True, collate_fn=collate_fn, batch_size=batch_size
    )
    eval_dataloader = DataLoader(
        tokenized_datasets["validation"], shuffle=False, collate_fn=collate_fn, batch_size=EVAL_BATCH_SI
    )

    return train_dataloader, eval_dataloader


# For testing only
if os.environ.get("TESTING_MOCKED_DATALOADERS", None) == "1":
    from accelerate.test_utils.training import mocked_dataloaders

    get_dataloaders = mocked_dataloaders  # noqa: F811


def training_function(config, args):
    # For testing only
    if os.environ.get("TESTING_MOCKED_DATALOADERS", None) == "1":
        config["num_epochs"] = 2
    # New Code #
    gradient_accumulation_steps = int(args.gradient_accumulation_steps)
    # Initialize accelerator
    accelerator = Accelerator(
        cpu=args.cpu, mixed_precision=args.mixed_precision, gradient_accumulation_steps=gradient_accumul
    )
    if accelerator.distributed_type == DistributedType.TPU and gradient_accumulation_steps > 1:
        raise NotImplementedError(
            "Gradient accumulation on TPUs is currently not supported. Pass `gradient_accumulation_steps
        )
    # Sample hyper-parameters for learning rate, batch size, seed and a few other HPs
    lr = config["lr"]
    num_epochs = int(config["num_epochs"])
    seed = int(config["seed"])
    batch_size = int(config["batch_size"])

    metric = evaluate.load("glue", "mrpc")

    set_seed(seed)
    train_dataloader, eval_dataloader = get_dataloaders(accelerator, batch_size)
    # Instantiate the model (we build the model here so that the seed also control new weights initializ
    model = AutoModelForSequenceClassification.from_pretrained("bert-base-cased", return_dict=True)

    # We could avoid this line since the accelerator is set with `device_placement=True` (default value)
    # Note that if you are placing tensors on devices manually, this line absolutely needs to be before
    # creation otherwise training will not work on TPU (`accelerate` will kindly throw an error to make
    model = model.to(accelerator.device)

    # Instantiate optimizer
    optimizer = AdamW(params=model.parameters(), lr=lr)

    # Instantiate scheduler
    lr_scheduler = get_linear_schedule_with_warmup(
        optimizer=optimizer,
        num_warmup_steps=100,
        num_training_steps=(len(train_dataloader) * num_epochs),
    )

    # Prepare everything
    # There is no specific order to remember, we just need to unpack the objects in the same order we ga
    # prepare method.
    model, optimizer, train_dataloader, eval_dataloader, lr_scheduler = accelerator.prepare(
        model, optimizer, train_dataloader, eval_dataloader, lr_scheduler
    )

    # Now we train the model
    for epoch in range(num_epochs):
        model.train()
        for step, batch in enumerate(train_dataloader):
            # We could avoid this line since we set the accelerator with `device_placement=True`.
            batch.to(accelerator.device)
            # New code #
            # We use the new `accumulate` context manager to perform gradient accumulation
            # We also currently do not support TPUs nor advise it as bugs were found on the XLA side whe
            with accelerator.accumulate(model):
                output = model(**batch)
                loss = output.loss
                accelerator.backward(loss)
                optimizer.step()
                lr_scheduler.step()
                optimizer.zero_grad()

        model.eval()
        for step, batch in enumerate(eval_dataloader):
            # We could avoid this line since we set the accelerator with `device_placement=True`.
            batch.to(accelerator.device)
            with torch.no_grad():
                outputs = model(**batch)
            predictions = outputs.logits.argmax(dim=-1)
            predictions, references = accelerator.gather_for_metrics((predictions, batch["labels"]))
            metric.add_batch(
                predictions=predictions,
                references=references,
            )

        eval_metric = metric.compute()
        # Use accelerator.print to print only on the main process.
        accelerator.print(f"epoch {epoch}:", eval_metric)


def main():
    parser = argparse.ArgumentParser(description="Simple example of training script.")
    parser.add_argument(
        "--mixed_precision",
        type=str,
        default=None,
        choices=["no", "fp16", "bf16", "fp8"],
        help="Whether to use mixed precision. Choose"
        "between fp16 and bf16 (bfloat16). Bf16 requires PyTorch >= 1.10."
        "and an Nvidia Ampere GPU.",
    )
    # New Code #
    parser.add_argument(
        "--gradient_accumulation_steps",
        type=int,
        default=1,
        help="The number of minibatches to be ran before gradients are accumulated.",
    )
    parser.add_argument("--cpu", action="store_true", help="If passed, will train on the CPU.")
    args = parser.parse_args()
    config = {"lr": 2e-5, "num_epochs": 3, "seed": 42, "batch_size": 16}
    training_function(config, args)


if __name__ == "__main__":
    main()
```

accelerate-main/docs/source/usage_guides/explore.mdx

~~~
# Learning how to incorporate ■ Accelerate features quickly!

Please use the interactive tool below to help you get started with learning about a particular
feature of ■ Accelerate and how to utilize it! It will provide you with a code diff, an explaination
towards what is going on, as well as provide you with some useful links to explore more within
the documentation!

Most code examples start from the following python code before integrating ■ Accelerate in some way:

```python
for batch in dataloader:
    optimizer.zero_grad()
    inputs, targets = batch
    inputs = inputs.to(device)
    targets = targets.to(device)
    outputs = model(inputs)
    loss = loss_function(outputs, targets)
    loss.backward()
    optimizer.step()
    scheduler.step()
```

<div class="block dark:hidden">
■<iframe
        src="https://muellerzr-accelerate-examples.hf.space?__theme=light"
        width="850"
        height="1600"
    ></iframe>
</div>
<div class="hidden dark:block">
    <iframe
        src="https://muellerzr-accelerate-examples.hf.space?__theme=dark"
        width="850"
        height="1600"
    ></iframe>
</div>
~~~

accelerate-main/examples/by_feature/memory.py

```
# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
import argparse
import os

# New Code #
import evaluate
import torch
from datasets import load_dataset
from torch.optim import AdamW
from torch.utils.data import DataLoader
from transformers import AutoModelForSequenceClassification, AutoTokenizer, get_linear_schedule_with_war

from accelerate import Accelerator, DistributedType
from accelerate.utils import find_executable_batch_size


########################################################################
# This is a fully working simple example to use Accelerate,
# specifically showcasing how to ensure out-of-memory errors never
# interrupt training, and builds off the `nlp_example.py` script.
#
# This example trains a Bert base model on GLUE MRPC
# in any of the following settings (with the same script):
#   - single CPU or single GPU
#   - multi GPUS (using PyTorch distributed mode)
#   - (multi) TPUs
#   - fp16 (mixed-precision) or fp32 (normal precision)
#
# New additions from the base script can be found quickly by
# looking for the # New Code # tags
#
# To run it in each of these various modes, follow the instructions
# in the readme for examples:
# https://github.com/huggingface/accelerate/tree/main/examples
#
########################################################################


MAX_GPU_BATCH_SIZE = 16
EVAL_BATCH_SIZE = 32


def get_dataloaders(accelerator: Accelerator, batch_size: int = 16):
    """
    Creates a set of `DataLoader`s for the `glue` dataset,
    using "bert-base-cased" as the tokenizer.

    Args:
        accelerator (`Accelerator`):
            An `Accelerator` object
        batch_size (`int`, *optional*):
            The batch size for the train and validation DataLoaders.
    """
    tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
    datasets = load_dataset("glue", "mrpc")
    def tokenize_function(examples):
        # max_length=None => use the model max length (it's actually the default)
        outputs = tokenizer(examples["sentence1"], examples["sentence2"], truncation=True, max_length=No
        return outputs

    # Apply the method we just defined to all the examples in all the splits of the dataset
    # starting with the main process first:
    with accelerator.main_process_first():
        tokenized_datasets = datasets.map(
            tokenize_function,
            batched=True,
            remove_columns=["idx", "sentence1", "sentence2"],
        )

    # We also rename the 'label' column to 'labels' which is the expected name for labels by the models
    # transformers library
    tokenized_datasets = tokenized_datasets.rename_column("label", "labels")

    def collate_fn(examples):
        # On TPU it's best to pad everything to the same length or training will be very slow.
        max_length = 128 if accelerator.distributed_type == DistributedType.TPU else None
        # When using mixed precision we want round multiples of 8/16
        if accelerator.mixed_precision == "fp8":
            pad_to_multiple_of = 16
        elif accelerator.mixed_precision != "no":
            pad_to_multiple_of = 8
        else:
            pad_to_multiple_of = None

        return tokenizer.pad(
            examples,
            padding="longest",
            max_length=max_length,
            pad_to_multiple_of=pad_to_multiple_of,
            return_tensors="pt",
        )

    # Instantiate dataloaders.
    train_dataloader = DataLoader(
        tokenized_datasets["train"], shuffle=True, collate_fn=collate_fn, batch_size=batch_size
    )
    eval_dataloader = DataLoader(
        tokenized_datasets["validation"], shuffle=False, collate_fn=collate_fn, batch_size=EVAL_BATCH_SI
    )

    return train_dataloader, eval_dataloader


# For testing only
if os.environ.get("TESTING_MOCKED_DATALOADERS", None) == "1":
    from accelerate.test_utils.training import mocked_dataloaders

    get_dataloaders = mocked_dataloaders  # noqa: F811


def training_function(config, args):
    # For testing only
    if os.environ.get("TESTING_MOCKED_DATALOADERS", None) == "1":
        config["num_epochs"] = 2
    # Initialize accelerator
    accelerator = Accelerator(cpu=args.cpu, mixed_precision=args.mixed_precision)
    # Sample hyper-parameters for learning rate, batch size, seed and a few other HPs
    lr = config["lr"]
    num_epochs = int(config["num_epochs"])
    seed = int(config["seed"])
    batch_size = int(config["batch_size"])

    metric = evaluate.load("glue", "mrpc")

    # New Code #
    # We now can define an inner training loop function. It should take a batch size as the only paramet
    # and build the dataloaders in there.
    # It also gets our decorator
    @find_executable_batch_size(starting_batch_size=batch_size)
    def inner_training_loop(batch_size):
        # And now just move everything below under this function
        # We need to bring in the Accelerator object from earlier
        nonlocal accelerator
        # And reset all of its attributes that could hold onto any memory:
        accelerator.free_memory()

        # Then we can declare the model, optimizer, and everything else:
        set_seed(seed)

        # Instantiate the model (we build the model here so that the seed also control new weights initi
        model = AutoModelForSequenceClassification.from_pretrained("bert-base-cased", return_dict=True)

        # We could avoid this line since the accelerator is set with `device_placement=True` (default va
        # Note that if you are placing tensors on devices manually, this line absolutely needs to be bef
        # creation otherwise training will not work on TPU (`accelerate` will kindly throw an error to m
        model = model.to(accelerator.device)

        # Instantiate optimizer
        optimizer = AdamW(params=model.parameters(), lr=lr)
        train_dataloader, eval_dataloader = get_dataloaders(accelerator, batch_size)

        # Instantiate scheduler
        lr_scheduler = get_linear_schedule_with_warmup(
            optimizer=optimizer,
            num_warmup_steps=100,
            num_training_steps=(len(train_dataloader) * num_epochs),
        )

        # Prepare everything
        # There is no specific order to remember, we just need to unpack the objects in the same order w
        # prepare method.
        model, optimizer, train_dataloader, eval_dataloader, lr_scheduler = accelerator.prepare(
            model, optimizer, train_dataloader, eval_dataloader, lr_scheduler
        )

        # Now we train the model
        for epoch in range(num_epochs):
            model.train()
            for step, batch in enumerate(train_dataloader):
                # We could avoid this line since we set the accelerator with `device_placement=True`.
                batch.to(accelerator.device)
                outputs = model(**batch)
                loss = outputs.loss
                accelerator.backward(loss)
                optimizer.step()
                lr_scheduler.step()
                optimizer.zero_grad()

            model.eval()
            for step, batch in enumerate(eval_dataloader):
                # We could avoid this line since we set the accelerator with `device_placement=True`.
                batch.to(accelerator.device)
                with torch.no_grad():
                    outputs = model(**batch)
                predictions = outputs.logits.argmax(dim=-1)
                predictions, references = accelerator.gather_for_metrics((predictions, batch["labels"]))
                metric.add_batch(
                    predictions=predictions,
                    references=references,
                )

            eval_metric = metric.compute()
            # Use accelerator.print to print only on the main process.
            accelerator.print(f"epoch {epoch}:", eval_metric)

    # New Code #
    # And call it at the end with no arguments
    # Note: You could also refactor this outside of your training loop function
    inner_training_loop()
def main():
    parser = argparse.ArgumentParser(description="Simple example of training script.")
    parser.add_argument(
        "--mixed_precision",
        type=str,
        default=None,
        choices=["no", "fp16", "bf16", "fp8"],
        help="Whether to use mixed precision. Choose"
        "between fp16 and bf16 (bfloat16). Bf16 requires PyTorch >= 1.10."
        "and an Nvidia Ampere GPU.",
    )
    parser.add_argument("--cpu", action="store_true", help="If passed, will train on the CPU.")
    args = parser.parse_args()
    config = {"lr": 2e-5, "num_epochs": 3, "seed": 42, "batch_size": 16}
    training_function(config, args)


if __name__ == "__main__":
    main()
```

accelerate-main/docs/source/usage_guides/mps.mdx

~~~
# Accelerated PyTorch Training on Mac

With PyTorch v1.12 release, developers and researchers can take advantage of Apple silicon GPUs for sign
This unlocks the ability to perform machine learning workflows like prototyping and fine-tuning locally,
Apple's Metal Performance Shaders (MPS) as a backend for PyTorch enables this and can be used via the ne
This will map computational graphs and primitives on the MPS Graph framework and tuned kernels provided
For more information please refer official documents [Introducing Accelerated PyTorch Training on Mac](h
and [MPS BACKEND](https://pytorch.org/docs/stable/notes/mps.html).

### Benefits of Training and Inference using Apple Silicon Chips

1. Enables users to train larger networks or batch sizes locally
2. Reduces data retrieval latency and provides the GPU with direct access to the full memory store due t
Therefore, improving end-to-end performance.
3. Reduces costs associated with cloud-based development or the need for additional local GPUs.

**Pre-requisites**: To install torch with mps support,
please follow this nice medium article [GPU-Acceleration Comes to PyTorch on M1 Macs](https://medium.com


## How it works out of the box
It is enabled by default on MacOs machines with MPS enabled Apple Silicon GPUs.
To disable it, pass `--cpu` flag to `accelerate launch` command or answer the corresponding question whe

You can directly run the following script to test it out on MPS enabled Apple Silicon machines:
```bash
accelerate launch /examples/cv_example.py --data_dir images
```

## A few caveats to be aware of

1. We strongly recommend to install PyTorch >= 1.13 (nightly version at the time of writing) on your Mac
It has major fixes related to model correctness and performance improvements for transformer based model
Please refer to https://github.com/pytorch/pytorch/issues/82707 for more details.
2. Distributed setups `gloo` and `nccl` are not working with `mps` device.
This means that currently only single GPU of `mps` device type can be used.

Finally, please, remember that, ■ `Accelerate` only integrates MPS backend, therefore if you
have any problems or questions with regards to MPS backend usage, please, file an issue with [PyTorch Gi
~~~

accelerate-main/docs/source/basic_tutorials/launch.mdx

# Launching your ■ Accelerate scripts

In the previous tutorial, you were introduced to how to modify your current training script to use ■ Acc
The final version of that code is shown below:

```python
from accelerate import Accelerator

accelerator = Accelerator()

model, optimizer, training_dataloader, scheduler = accelerator.prepare(
    model, optimizer, training_dataloader, scheduler
)

for batch in training_dataloader:
    optimizer.zero_grad()
    inputs, targets = batch
    outputs = model(inputs)
    loss = loss_function(outputs, targets)
    accelerator.backward(loss)
    optimizer.step()
    scheduler.step()
```

But how do you run this code and have it utilize the special hardware available to it?

First you should rewrite the above code into a function, and make it callable as a script. For example:

```diff
  from accelerate import Accelerator

+ def main():
      accelerator = Accelerator()

      model, optimizer, training_dataloader, scheduler = accelerator.prepare(
          model, optimizer, training_dataloader, scheduler
      )

      for batch in training_dataloader:
          optimizer.zero_grad()
          inputs, targets = batch
          outputs = model(inputs)
          loss = loss_function(outputs, targets)
          accelerator.backward(loss)
          optimizer.step()
          scheduler.step()

+ if __name__ == "__main__":
+     main()
```

Next you need to launch it with `accelerate launch`.

<Tip warning={true}>

  It's recommended you run `accelerate config` before using `accelerate launch` to configure your enviro

  Otherwise ■ Accelerate will use very basic defaults depending on your system setup.

</Tip>

## Using accelerate launch

■ Accelerate has a special CLI command to help you launch your code in your system through `accelerate l
This command wraps around all of the different commands needed to launch your script on various platform

<Tip>

  If you are familiar with launching scripts in PyTorch yourself such as with `torchrun`, you can still

</Tip>

You can launch your script quickly by using:

```bash
accelerate launch {script_name.py} --arg1 --arg2 ...
```

Just put `accelerate launch` at the start of your command, and pass in additional arguments and paramete

Since this runs the various torch spawn methods, all of the expected environment variables can be modifi
For example, here is how to use `accelerate launch` with a single GPU:

```bash
CUDA_VISIBLE_DEVICES="0" accelerate launch {script_name.py} --arg1 --arg2 ...
```

You can also use `accelerate launch` without performing `accelerate config` first, but you may need to m
In this case, ■ Accelerate will make some hyperparameter decisions for you, e.g., if GPUs are available,
Here is how you would use all GPUs and train with mixed precision disabled:

```bash
accelerate launch --multi_gpu {script_name.py} {--arg1} {--arg2} ...
```

To get more specific you should pass in the needed parameters yourself. For instance, here is how you
would also launch that same script on two GPUs using mixed precision while avoiding all of the warnings:

```bash
accelerate launch --multi_gpu --mixed_precision=fp16 --num_processes=2 {script_name.py} {--arg1} {--arg2
```

For a complete list of parameters you can pass in, run:

```bash
accelerate launch -h
```

<Tip>

  Even if you are not using ■ Accelerate in your code, you can still use the launcher for starting your

</Tip>

For a visualization of this difference, that earlier `accelerate launch` on multi-gpu would look somethi

```bash
MIXED_PRECISION="fp16" torchrun --nproc_per_node=2 --num_machines=1 {script_name.py} {--arg1} {--arg2} .
```

## Why you should always use `accelerate config`

Why is it useful to the point you should **always** run `accelerate config`?

Remember that earlier call to `accelerate launch` as well as `torchrun`?
Post configuration, to run that script with the needed parts you just need to use `accelerate launch` ou

```bash
```

```
accelerate launch {script_name.py} {--arg1} {--arg2} ...
```

## Custom Configurations

As briefly mentioned earlier, `accelerate launch` should be mostly used through combining set configurat
made with the `accelerate config` command. These configs are saved to a `default_config.yaml` file in yo
This cache folder is located at (with decreasing order of priority):

- The content of your environment variable `HF_HOME` suffixed with `accelerate`.
- If it does not exist, the content of your environment variable `XDG_CACHE_HOME` suffixed with
  `huggingface/accelerate`.
- If this does not exist either, the folder `~/.cache/huggingface/accelerate`.

To have multiple configurations, the flag `--config_file` can be passed to the `accelerate launch` comma
with the location of the custom yaml.

An example yaml may look something like the following for two GPUs on a single machine using `fp16` for
```yaml
compute_environment: LOCAL_MACHINE
deepspeed_config: {}
distributed_type: MULTI_GPU
fsdp_config: {}
machine_rank: 0
main_process_ip: null
main_process_port: null
main_training_function: main
mixed_precision: fp16
num_machines: 1
num_processes: 2
use_cpu: false
```

Launching a script from the location of that custom yaml file looks like the following:
```bash
accelerate launch --config_file {path/to/config/my_config_file.yaml} {script_name.py} {--arg1} {--arg2}
```

accelerate-main/docs/source/basic_tutorials/overview.mdx

```
# Overview

Welcome to the ■ Accelerate tutorials! These introductory guides will help catch you up to speed on work
You'll learn how to modify your code to have it work with the API seamlessly, how to launch your script
and more!

These tutorials assume some basic knowledge of Python and familiarity with the PyTorch framework.

If you have any questions about ■ Accelerate, feel free to join and ask the community on our [forum](htt
```

accelerate-main/docs/source/basic_tutorials/notebook.mdx

# Launching Multi-Node Training from a Jupyter Environment

This tutorial teaches you how to fine tune a computer vision model with ■ Accelerate from a Jupyter Note
You will also learn how to setup a few requirements needed for ensuring your environment is configured p

<Tip>

    This tutorial is also available as a Jupyter Notebook [here](https://github.com/huggingface/notebook

</Tip>

## Configuring the Environment

Before any training can be performed, a ■ Accelerate config file must exist in the system. Usually this

```bash
accelerate config
```

However, if general defaults are fine and you are *not* running on a TPU, ■Accelerate has a utility to q

The following code will restart Jupyter after writing the configuration, as CUDA code was called to perf

<Tip warning={true}>

    CUDA can't be initialized more than once on a multi-node system. It's fine to debug in the notebook

</Tip>

```python
import os
from accelerate.utils import write_basic_config

write_basic_config()  # Write a config file
os._exit(00)  # Restart the notebook
```

## Preparing the Dataset and Model

Next you should prepare your dataset. As mentioned at earlier, great care should be taken when preparing

If you do, it is recommended to put that specific code into a function and call that from within the not

Make sure the dataset is downloaded based on the directions [here](https://github.com/huggingface/accele

```python
import os, re, torch, PIL
import numpy as np

from torch.optim.lr_scheduler import OneCycleLR
from torch.utils.data import DataLoader, Dataset
from torchvision.transforms import Compose, RandomResizedCrop, Resize, ToTensor

from accelerate import Accelerator
from accelerate.utils import set_seed
from timm import create_model
```

First you need to create a function to extract the class name based on a filename:

```python
import os

data_dir = "../../images"
fnames = os.listdir(data_dir)
fname = fnames[0]
print(fname)
```

```python out
beagle_32.jpg
```

In the case here, the label is `beagle`. Using regex you can extract the label from the filename:

```python
import re


def extract_label(fname):
    stem = fname.split(os.path.sep)[-1]
    return re.search(r"^(.*)_\d+\.jpg$", stem).groups()[0]
```

```python
extract_label(fname)
```

And you can see it properly returned the right name for our file:

```python out
"beagle"
```

Next a `Dataset` class should be made to handle grabbing the image and the label:

```python
class PetsDataset(Dataset):
    def __init__(self, file_names, image_transform=None, label_to_id=None):
        self.file_names = file_names
        self.image_transform = image_transform
        self.label_to_id = label_to_id

    def __len__(self):
        return len(self.file_names)

    def __getitem__(self, idx):
        fname = self.file_names[idx]
        raw_image = PIL.Image.open(fname)
        image = raw_image.convert("RGB")
        if self.image_transform is not None:
            image = self.image_transform(image)
        label = extract_label(fname)
        if self.label_to_id is not None:
            label = self.label_to_id[label]
        return {"image": image, "label": label}
```

Now to build the dataset. Outside the training function you can find and declare all the filenames and l
launched function:

```python
fnames = [os.path.join("../../images", fname) for fname in fnames if fname.endswith(".jpg")]
```

Next gather all the labels:

```python
```

~~~~
all_labels = [extract_label(fname) for fname in fnames]
id_to_label = list(set(all_labels))
id_to_label.sort()
label_to_id = {lbl: i for i, lbl in enumerate(id_to_label)}
```

Next, you should make a `get_dataloaders` function that will return your built dataloaders for you. As m
sent to the GPU or a TPU device when building your `DataLoaders`, they must be built using this method.

```python
def get_dataloaders(batch_size: int = 64):
    "Builds a set of dataloaders with a batch_size"
    random_perm = np.random.permutation(len(fnames))
    cut = int(0.8 * len(fnames))
    train_split = random_perm[:cut]
    eval_split = random_perm[cut:]

    # For training a simple RandomResizedCrop will be used
    train_tfm = Compose([RandomResizedCrop((224, 224), scale=(0.5, 1.0)), ToTensor()])
    train_dataset = PetsDataset([fnames[i] for i in train_split], image_transform=train_tfm, label_to_id

    # For evaluation a deterministic Resize will be used
    eval_tfm = Compose([Resize((224, 224)), ToTensor()])
    eval_dataset = PetsDataset([fnames[i] for i in eval_split], image_transform=eval_tfm, label_to_id=la

    # Instantiate dataloaders
    train_dataloader = DataLoader(train_dataset, shuffle=True, batch_size=batch_size, num_workers=4)
    eval_dataloader = DataLoader(eval_dataset, shuffle=False, batch_size=batch_size * 2, num_workers=4)
    return train_dataloader, eval_dataloader
```

Finally, you should import the scheduler to be used later:

```python
from torch.optim.lr_scheduler import CosineAnnealingLR
```

## Writing the Training Function

Now you can build the training loop. [`notebook_launcher`] works by passing in a function to call that w

Here is a basic training loop for the animal classification problem:

<Tip>

    The code has been split up to allow for explainations on each section. A full version that can be co

</Tip>


```python
def training_loop(mixed_precision="fp16", seed: int = 42, batch_size: int = 64):
    set_seed(seed)
    accelerator = Accelerator(mixed_precision=mixed_precision)
```

First you should set the seed and create an [`Accelerator`] object as early in the training loop as poss

<Tip warning={true}>

    If training on the TPU, your training loop should take in the model as a parameter and it should be
    outside of the training loop function. See the [TPU best practices](../concept_guides/training_tpu)
    to learn why

</Tip>

Next you should build your dataloaders and create your model:

```python
    train_dataloader, eval_dataloader = get_dataloaders(batch_size)
    model = create_model("resnet50d", pretrained=True, num_classes=len(label_to_id))
```
~~~~

<Tip>

    You build the model here so that the seed also controls the new weight initialization

</Tip>

As you are performing transfer learning in this example, the encoder of the model starts out frozen so t
trained only initially:

```python
    for param in model.parameters():
        param.requires_grad = False
    for param in model.get_classifier().parameters():
        param.requires_grad = True
```

Normalizing the batches of images will make training a little faster:

```python
    mean = torch.tensor(model.default_cfg["mean"])[None, :, None, None]
    std = torch.tensor(model.default_cfg["std"])[None, :, None, None]
```

To make these constants available on the active device, you should set it to the Accelerator's device:

```python
    mean = mean.to(accelerator.device)
    std = std.to(accelerator.device)
```

Next instantiate the rest of the PyTorch classes used for training:

```python
    optimizer = torch.optim.Adam(params=model.parameters(), lr=3e-2 / 25)
    lr_scheduler = OneCycleLR(optimizer=optimizer, max_lr=3e-2, epochs=5, steps_per_epoch=len(train_data
```

Before passing everything to [`~Accelerator.prepare`].

<Tip>

    There is no specific order to remember, you just need to unpack the objects in the same order you ga

</Tip>

```python
    model, optimizer, train_dataloader, eval_dataloader, lr_scheduler = accelerator.prepare(
        model, optimizer, train_dataloader, eval_dataloader, lr_scheduler
    )
```

Now train the model:

```python
    for epoch in range(5):
        model.train()
        for batch in train_dataloader:
            inputs = (batch["image"] - mean) / std
            outputs = model(inputs)
            loss = torch.nn.functional.cross_entropy(outputs, batch["label"])
            accelerator.backward(loss)
            optimizer.step()
            lr_scheduler.step()
            optimizer.zero_grad()
```

The evaluation loop will look slightly different compared to the training loop. The number of elements p
total accuracy of each batch will be added to two constants:

```python
        model.eval()
        accurate = 0
```

        num_elems = 0

Next you have the rest of your standard PyTorch loop:

```python
        for batch in eval_dataloader:
            inputs = (batch["image"] - mean) / std
            with torch.no_grad():
                outputs = model(inputs)
            predictions = outputs.argmax(dim=-1)
```

Before finally the last major difference.

When performing distributed evaluation, the predictions and labels need to be passed through
[`~Accelerator.gather`] so that all of the data is available on the current device and a properly calcul

```python
            accurate_preds = accelerator.gather(predictions) == accelerator.gather(batch["label"])
            num_elems += accurate_preds.shape[0]
            accurate += accurate_preds.long().sum()
```

Now you just need to calculate the actual metric for this problem, and you can print it on the main proc

```python
        eval_metric = accurate.item() / num_elems
        accelerator.print(f"epoch {epoch}: {100 * eval_metric:.2f}")
```

A full version of this training loop is available below:

```python
def training_loop(mixed_precision="fp16", seed: int = 42, batch_size: int = 64):
    set_seed(seed)
    # Initialize accelerator
    accelerator = Accelerator(mixed_precision=mixed_precision)
    # Build dataloaders
    train_dataloader, eval_dataloader = get_dataloaders(batch_size)

    # Instantiate the model (you build the model here so that the seed also controls new weight initaliz
    model = create_model("resnet50d", pretrained=True, num_classes=len(label_to_id))

    # Freeze the base model
    for param in model.parameters():
        param.requires_grad = False
    for param in model.get_classifier().parameters():
        param.requires_grad = True

    # You can normalize the batches of images to be a bit faster
    mean = torch.tensor(model.default_cfg["mean"])[None, :, None, None]
    std = torch.tensor(model.default_cfg["std"])[None, :, None, None]

    # To make these constants available on the active device, set it to the accelerator device
    mean = mean.to(accelerator.device)
    std = std.to(accelerator.device)

    # Intantiate the optimizer
    optimizer = torch.optim.Adam(params=model.parameters(), lr=3e-2 / 25)

    # Instantiate the learning rate scheduler
    lr_scheduler = OneCycleLR(optimizer=optimizer, max_lr=3e-2, epochs=5, steps_per_epoch=len(train_data

    # Prepare everything
    # There is no specific order to remember, you just need to unpack the objects in the same order you
    # prepare method.
    model, optimizer, train_dataloader, eval_dataloader, lr_scheduler = accelerator.prepare(
        model, optimizer, train_dataloader, eval_dataloader, lr_scheduler
    )

    # Now you train the model
```

~~~
    for epoch in range(5):
        model.train()
        for batch in train_dataloader:
            inputs = (batch["image"] - mean) / std
            outputs = model(inputs)
            loss = torch.nn.functional.cross_entropy(outputs, batch["label"])
            accelerator.backward(loss)
            optimizer.step()
            lr_scheduler.step()
            optimizer.zero_grad()

        model.eval()
        accurate = 0
        num_elems = 0
        for batch in eval_dataloader:
            inputs = (batch["image"] - mean) / std
            with torch.no_grad():
                outputs = model(inputs)
            predictions = outputs.argmax(dim=-1)
            accurate_preds = accelerator.gather(predictions) == accelerator.gather(batch["label"])
            num_elems += accurate_preds.shape[0]
            accurate += accurate_preds.long().sum()

        eval_metric = accurate.item() / num_elems
        # Use accelerator.print to print only on the main process.
        accelerator.print(f"epoch {epoch}: {100 * eval_metric:.2f}")
```

## Using the notebook_launcher

All that's left is to use the [`notebook_launcher`].

You pass in the function, the arguments (as a tuple), and the number of processes to train on. (See the

```python
from accelerate import notebook_launcher
```

```python
args = ("fp16", 42, 64)
notebook_launcher(training_loop, args, num_processes=2)
```

In the case of running on the TPU, it would look like so:

```python
model = create_model("resnet50d", pretrained=True, num_classes=len(label_to_id))

args = (model, "fp16", 42, 64)
notebook_launcher(training_loop, args, num_processes=8)
```

As it's running it will print the progress as well as state how many devices you ran on. This tutorial w

```python out
Launching training on 2 GPUs.
epoch 0: 88.12
epoch 1: 91.73
epoch 2: 92.58
epoch 3: 93.90
epoch 4: 94.71
```

And that's it!

## Conclusion

This notebook showed how to perform distributed training from inside of a Jupyter Notebook. Some key not

- Make sure to save any code that use CUDA (or CUDA imports) for the function passed to [`notebook_launc
- Set the `num_processes` to be the number of devices used for training (such as number of GPUs, CPUs, T
- If using the TPU, declare your model outside the training loop function
~~~

accelerate-main/docs/source/basic_tutorials/migration.mdx

~~~
<!--Copyright 2022 The HuggingFace Team. All rights reserved.

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in comp
the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is dis
an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the Licen
specific language governing permissions and limitations under the License.
-->

# Migrating your code to ■ Accelerate

This tutorial will detail how to easily convert existing PyTorch code to use ■ Accelerate!
You'll see that by just changing a few lines of code, ■ Accelerate can perform its magic and get you on
your way towards running your code on distributed systems with ease!

## The base training loop

To begin, write out a very basic PyTorch training loop.

<Tip>

    We are under the presumption that `training_dataloader`, `model`, `optimizer`, `scheduler`, and `los

</Tip>

```python
device = "cuda"
model.to(device)

for batch in training_dataloader:
    optimizer.zero_grad()
    inputs, targets = batch
    inputs = inputs.to(device)
    targets = targets.to(device)
    outputs = model(inputs)
    loss = loss_function(outputs, targets)
    loss.backward()
    optimizer.step()
    scheduler.step()
```

## Add in ■ Accelerate

To start using ■ Accelerate, first import and create an [`Accelerator`] instance:
```python
from accelerate import Accelerator

accelerator = Accelerator()
```
[`Accelerator`] is the main force behind utilizing all the possible options for distributed training!

### Setting the right device

The [`Accelerator`] class knows the right device to move any PyTorch object to at any time, so you shoul
change the definition of `device` to come from [`Accelerator`]:

```diff
- device = 'cuda'
+ device = accelerator.device
  model.to(device)
```

### Preparing your objects

Next you need to pass all of the important objects related to training into [`~Accelerator.prepare`]. ■
~~~

make sure everything is setup in the current environment for you to start training:

```
model, optimizer, training_dataloader, scheduler = accelerator.prepare(
    model, optimizer, training_dataloader, scheduler
)
```
These objects are returned in the same order they were sent in with. By default when using `device_place
If you need to work with data that isn't passed to [~Accelerator.prepare] but should be on the active de

<Tip warning={true}>

    Accelerate will only prepare objects that inherit from their respective PyTorch classes (such as `to

</Tip>

### Modifying the training loop

Finally, three lines of code need to be changed in the training loop. ■ Accelerate's DataLoader classes
and [`~Accelerator.backward`] should be used for performing the backward pass:

```diff
-   inputs = inputs.to(device)
-   targets = targets.to(device)
    outputs = model(inputs)
    loss = loss_function(outputs, targets)
-   loss.backward()
+   accelerator.backward(loss)
```

With that, your training loop is now ready to use ■ Accelerate!

## The finished code

Below is the final version of the converted code:

```python
from accelerate import Accelerator

accelerator = Accelerator()

model, optimizer, training_dataloader, scheduler = accelerator.prepare(
    model, optimizer, training_dataloader, scheduler
)

for batch in training_dataloader:
    optimizer.zero_grad()
    inputs, targets = batch
    outputs = model(inputs)
    loss = loss_function(outputs, targets)
    accelerator.backward(loss)
    optimizer.step()
    scheduler.step()
```

accelerate-main/docs/source/basic_tutorials/install.mdx

~~~~markdown
<!--Copyright 2022 The HuggingFace Team. All rights reserved.

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in comp
the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is dis
an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the Licen
specific language governing permissions and limitations under the License.
-->

# Installation and Configuration

Before you start, you will need to setup your environment, install the appropriate packages, and configu

## Installing ■ Accelerate

■ Accelerate is available on pypi and conda, as well as on GitHub. Details to install from each are belo

### pip

To install ■ Accelerate from pypi, perform:

```bash
pip install accelerate
```

### conda

■ Accelerate can also be installed with conda with:

```bash
conda install -c conda-forge accelerate
```

### Source

New features are added every day that haven't been released yet. To try them out yourself, install
from the GitHub repository:

```bash
pip install git+https://github.com/huggingface/accelerate
```

If you're working on contributing to the library or wish to play with the source code and see live
results as you run the code, an editable version can be installed from a locally-cloned version of the
repository:

```bash
git clone https://github.com/huggingface/accelerate
cd accelerate
pip install -e .
```

## Configuring ■ Accelerate

After installing, you need to configure ■ Accelerate for how the current system is setup for training.
To do so run the following and answer the questions prompted to you:

```bash
accelerate config
```

To write a barebones configuration that doesn't include options such as DeepSpeed configuration or runni

```bash
python -c "from accelerate.utils import write_basic_config; write_basic_config(mixed_precision='fp16')"
~~~~

~~~
■ Accelerate will automatically utilize the maximum number of GPUs available and set the mixed precision

To check that your configuration looks fine, run:

```bash
accelerate env
~~~

An example output is shown below, which describes two GPUs on a single machine with no mixed precision b

```bash
- `Accelerate` version: 0.11.0.dev0
- Platform: Linux-5.10.0-15-cloud-amd64-x86_64-with-debian-11.3
- Python version: 3.7.12
- Numpy version: 1.19.5
- PyTorch version (GPU?): 1.12.0+cu102 (True)
- `Accelerate` default config:
        - compute_environment: LOCAL_MACHINE
        - distributed_type: MULTI_GPU
        - mixed_precision: no
        - use_cpu: False
        - num_processes: 2
        - machine_rank: 0
        - num_machines: 1
        - main_process_ip: None
        - main_process_port: None
        - main_training_function: main
        - deepspeed_config: {}
        - fsdp_config: {}
```

accelerate-main/examples/by_feature/tracking.py

```
# coding=utf-8
# Copyright 2021 The HuggingFace Inc. team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
import argparse
import os

import evaluate
import torch
from datasets import load_dataset
from torch.optim import AdamW
from torch.utils.data import DataLoader
from transformers import AutoModelForSequenceClassification, AutoTokenizer, get_linear_schedule_with_war

from accelerate import Accelerator, DistributedType


########################################################################
# This is a fully working simple example to use Accelerate,
# specifically showcasing the experiment tracking capability,
# and builds off the `nlp_example.py` script.
#
# This example trains a Bert base model on GLUE MRPC
# in any of the following settings (with the same script):
#   - single CPU or single GPU
#   - multi GPUS (using PyTorch distributed mode)
#   - (multi) TPUs
#   - fp16 (mixed-precision) or fp32 (normal precision)
#
# To help focus on the differences in the code, building `DataLoaders`
# was refactored into its own function.
# New additions from the base script can be found quickly by
# looking for the # New Code # tags
#
# To run it in each of these various modes, follow the instructions
# in the readme for examples:
# https://github.com/huggingface/accelerate/tree/main/examples
#
########################################################################

MAX_GPU_BATCH_SIZE = 16
EVAL_BATCH_SIZE = 32


def get_dataloaders(accelerator: Accelerator, batch_size: int = 16):
    """
    Creates a set of `DataLoader`s for the `glue` dataset,
    using "bert-base-cased" as the tokenizer.

    Args:
        accelerator (`Accelerator`):
            An `Accelerator` object
        batch_size (`int`, *optional*):
            The batch size for the train and validation DataLoaders.
    """
    tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
    datasets = load_dataset("glue", "mrpc")
    def tokenize_function(examples):
        # max_length=None => use the model max length (it's actually the default)
        outputs = tokenizer(examples["sentence1"], examples["sentence2"], truncation=True, max_length=No
        return outputs

    # Apply the method we just defined to all the examples in all the splits of the dataset
    # starting with the main process first:
    with accelerator.main_process_first():
        tokenized_datasets = datasets.map(
            tokenize_function,
            batched=True,
            remove_columns=["idx", "sentence1", "sentence2"],
        )

    # We also rename the 'label' column to 'labels' which is the expected name for labels by the models
    # transformers library
    tokenized_datasets = tokenized_datasets.rename_column("label", "labels")

    def collate_fn(examples):
        # On TPU it's best to pad everything to the same length or training will be very slow.
        max_length = 128 if accelerator.distributed_type == DistributedType.TPU else None
        # When using mixed precision we want round multiples of 8/16
        if accelerator.mixed_precision == "fp8":
            pad_to_multiple_of = 16
        elif accelerator.mixed_precision != "no":
            pad_to_multiple_of = 8
        else:
            pad_to_multiple_of = None

        return tokenizer.pad(
            examples,
            padding="longest",
            max_length=max_length,
            pad_to_multiple_of=pad_to_multiple_of,
            return_tensors="pt",
        )

    # Instantiate dataloaders.
    train_dataloader = DataLoader(
        tokenized_datasets["train"], shuffle=True, collate_fn=collate_fn, batch_size=batch_size
    )
    eval_dataloader = DataLoader(
        tokenized_datasets["validation"], shuffle=False, collate_fn=collate_fn, batch_size=EVAL_BATCH_SI
    )

    return train_dataloader, eval_dataloader


# For testing only
if os.environ.get("TESTING_MOCKED_DATALOADERS", None) == "1":
    from accelerate.test_utils.training import mocked_dataloaders

    get_dataloaders = mocked_dataloaders  # noqa: F811


def training_function(config, args):
    # For testing only
    if os.environ.get("TESTING_MOCKED_DATALOADERS", None) == "1":
        config["num_epochs"] = 2
    # Initialize Accelerator

    # New Code #
    # We pass in "all" to `log_with` to grab all available trackers in the environment
    # Note: If using a custom `Tracker` class, should be passed in here such as:
    # >>> log_with = ["all", MyCustomTrackerClassInstance()]
    if args.with_tracking:
        accelerator = Accelerator(
            cpu=args.cpu, mixed_precision=args.mixed_precision, log_with="all", logging_dir=args.logging
        )
    else:
        accelerator = Accelerator(cpu=args.cpu, mixed_precision=args.mixed_precision)
    # Sample hyper-parameters for learning rate, batch size, seed and a few other HPs
    lr = config["lr"]
```

```
num_epochs = int(config["num_epochs"])
seed = int(config["seed"])
batch_size = int(config["batch_size"])
set_seed(seed)

train_dataloader, eval_dataloader = get_dataloaders(accelerator, batch_size)
metric = evaluate.load("glue", "mrpc")

# If the batch size is too big we use gradient accumulation
gradient_accumulation_steps = 1
if batch_size > MAX_GPU_BATCH_SIZE and accelerator.distributed_type != DistributedType.TPU:
    gradient_accumulation_steps = batch_size // MAX_GPU_BATCH_SIZE
    batch_size = MAX_GPU_BATCH_SIZE

# Instantiate the model (we build the model here so that the seed also control new weights initializ
model = AutoModelForSequenceClassification.from_pretrained("bert-base-cased", return_dict=True)

# We could avoid this line since the accelerator is set with `device_placement=True` (default value)
# Note that if you are placing tensors on devices manually, this line absolutely needs to be before
# creation otherwise training will not work on TPU (`accelerate` will kindly throw an error to make
model = model.to(accelerator.device)

# Instantiate optimizer
optimizer = AdamW(params=model.parameters(), lr=lr)

# Instantiate scheduler
lr_scheduler = get_linear_schedule_with_warmup(
    optimizer=optimizer,
    num_warmup_steps=100,
    num_training_steps=(len(train_dataloader) * num_epochs) // gradient_accumulation_steps,
)

# Prepare everything
# There is no specific order to remember, we just need to unpack the objects in the same order we ga
# prepare method.
model, optimizer, train_dataloader, eval_dataloader, lr_scheduler = accelerator.prepare(
    model, optimizer, train_dataloader, eval_dataloader, lr_scheduler
)

# New Code #
# We need to initialize the trackers we use. Overall configurations can also be stored
if args.with_tracking:
    run = os.path.split(__file__)[-1].split(".")[0]
    accelerator.init_trackers(run, config)

# Now we train the model
for epoch in range(num_epochs):
    model.train()
    # New Code #
    # For our tracking example, we will log the total loss of each epoch
    if args.with_tracking:
        total_loss = 0
    for step, batch in enumerate(train_dataloader):
        # We could avoid this line since we set the accelerator with `device_placement=True`.
        batch.to(accelerator.device)
        outputs = model(**batch)
        loss = outputs.loss
        # New Code #
        if args.with_tracking:
            total_loss += loss.detach().float()
        loss = loss / gradient_accumulation_steps
        accelerator.backward(loss)
        if step % gradient_accumulation_steps == 0:
            optimizer.step()
            lr_scheduler.step()
            optimizer.zero_grad()

    model.eval()
    for step, batch in enumerate(eval_dataloader):
        # We could avoid this line since we set the accelerator with `device_placement=True` (the de
        batch.to(accelerator.device)
        with torch.no_grad():
```

```
                    outputs = model(**batch)
                predictions = outputs.logits.argmax(dim=-1)
                predictions, references = accelerator.gather_for_metrics((predictions, batch["labels"]))
                metric.add_batch(
                    predictions=predictions,
                    references=references,
                )

            eval_metric = metric.compute()
            # Use accelerator.print to print only on the main process.
            accelerator.print(f"epoch {epoch}:", eval_metric)

            # New Code #
            # To actually log, we call `Accelerator.log`
            # The values passed can be of `str`, `int`, `float` or `dict` of `str` to `float`/`int`
            if args.with_tracking:
                accelerator.log(
                    {
                        "accuracy": eval_metric["accuracy"],
                        "f1": eval_metric["f1"],
                        "train_loss": total_loss.item() / len(train_dataloader),
                        "epoch": epoch,
                    },
                    step=epoch,
                )

        # New Code #
        # When a run is finished, you should call `accelerator.end_training()`
        # to close all of the open trackers
        if args.with_tracking:
            accelerator.end_training()


    def main():
        parser = argparse.ArgumentParser(description="Simple example of training script.")
        parser.add_argument(
            "--mixed_precision",
            type=str,
            default=None,
            choices=["no", "fp16", "bf16", "fp8"],
            help="Whether to use mixed precision. Choose"
            "between fp16 and bf16 (bfloat16). Bf16 requires PyTorch >= 1.10."
            "and an Nvidia Ampere GPU.",
        )
        parser.add_argument("--cpu", action="store_true", help="If passed, will train on the CPU.")
        parser.add_argument(
            "--with_tracking",
            action="store_true",
            help="Whether to load in all available experiment trackers from the environment and use them for
        )
        parser.add_argument(
            "--logging_dir",
            type=str,
            default="logs",
            help="Location on where to store experiment tracking logs`",
        )
        args = parser.parse_args()
        config = {"lr": 2e-5, "num_epochs": 3, "seed": 42, "batch_size": 16}
        training_function(config, args)


    if __name__ == "__main__":
        main()
```

accelerate-main/docs/source/package_reference/deepspeed.mdx

```
# Utilities for DeepSpeed

[[autodoc]] utils.DeepSpeedPlugin

[[autodoc]] utils.DummyOptim

[[autodoc]] utils.DummyScheduler

[[autodoc]] utils.DeepSpeedEngineWrapper

[[autodoc]] utils.DeepSpeedOptimizerWrapper

[[autodoc]] utils.DeepSpeedSchedulerWrapper
```

accelerate-main/docs/source/package_reference/state.mdx

# Stateful Classes

Below are variations of a [singleton class](https://en.wikipedia.org/wiki/Singleton_pattern) in the sens
instances share the same state, which is initialized on the first instantiation.

These classes are immutable and store information about certain configurations or
states.

[[autodoc]] state.PartialState

[[autodoc]] state.AcceleratorState

[[autodoc]] state.GradientState

accelerate-main/docs/source/package_reference/utilities.mdx

# Helpful Utilities

Below are a variety of utility functions that ■ Accelerate provides, broken down by use-case.

## Data Classes

These are basic dataclasses used throughout ■ Accelerate and they can be passed in as parameters.

[[autodoc]] utils.DistributedType

[[autodoc]] utils.LoggerType

[[autodoc]] utils.PrecisionType

[[autodoc]] utils.ProjectConfiguration

## Data Manipulation and Operations

These include data operations that mimic the same `torch` ops but can be used on distributed processes.

[[autodoc]] utils.broadcast

[[autodoc]] utils.concatenate

[[autodoc]] utils.gather

[[autodoc]] utils.pad_across_processes

[[autodoc]] utils.reduce

[[autodoc]] utils.send_to_device

## Environment Checks

These functionalities check the state of the current working environment including information about the

[[autodoc]] utils.is_bf16_available

[[autodoc]] utils.is_torch_version

[[autodoc]] utils.is_tpu_available

## Environment Configuration

[[autodoc]] utils.write_basic_config

When setting up ■ Accelerate for the first time, rather than running `accelerate config` [~utils.write_b

## Memory

[[autodoc]] utils.get_max_memory

[[autodoc]] utils.find_executable_batch_size

## Modeling
These utilities relate to interacting with PyTorch models

[[autodoc]] utils.extract_model_from_parallel

[[autodoc]] utils.get_max_layer_size

[[autodoc]] utils.offload_state_dict

## Parallel

These include general utilities that should be used when working in parallel.

[[autodoc]] utils.extract_model_from_parallel

[[autodoc]] utils.save

[[autodoc]] utils.wait_for_everyone

## Random

These utilities relate to setting and synchronizing of all the random states.

[[autodoc]] utils.set_seed

[[autodoc]] utils.synchronize_rng_state

[[autodoc]] utils.synchronize_rng_states

## PyTorch XLA

These include utilities that are useful while using PyTorch with XLA.

[[autodoc]] utils.install_xla

accelerate-main/docs/source/package_reference/torch_wrappers.mdx

```
<!--Copyright 2021 The HuggingFace Team. All rights reserved.

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in comp
the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is dis
an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the Licen
specific language governing permissions and limitations under the License.
-->

# Wrapper classes for torch Dataloaders, Optimizers, and Schedulers

The internal classes Accelerate uses to prepare objects for distributed training
when calling [`~Accelerator.prepare`].

## Datasets and DataLoaders

[[autodoc]] data_loader.prepare_data_loader

[[autodoc]] data_loader.BatchSamplerShard
[[autodoc]] data_loader.IterableDatasetShard
[[autodoc]] data_loader.DataLoaderShard
[[autodoc]] data_loader.DataLoaderDispatcher

## Optimizers

[[autodoc]] optimizer.AcceleratedOptimizer

## Schedulers

[[autodoc]] scheduler.AcceleratedScheduler
```

accelerate-main/docs/source/package_reference/accelerator.mdx

~~~
<!--Copyright 2021 The HuggingFace Team. All rights reserved.

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in comp
the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is dis
an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the Licen
specific language governing permissions and limitations under the License.
-->

# Accelerator

The [`Accelerator`] is the main class provided by ■ Accelerate.
It serves at the main entrypoint for the API.

## Quick adaptation of your code

To quickly adapt your script to work on any kind of setup with ■ Accelerate just:

1. Initialize an [`Accelerator`] object (that we will call `accelerator` throughout this page) as early
2. Pass your dataloader(s), model(s), optimizer(s), and scheduler(s) to the [`~Accelerator.prepare`] met
3. Remove all the `.cuda()` or `.to(device)` from your code and let the `accelerator` handle the device

<Tip>

    Step three is optional, but considered a best practice.

</Tip>

4. Replace `loss.backward()` in your code with `accelerator.backward(loss)`
5. Gather your predictions and labels before storing them or using them for metric computation using [`~

<Tip warning={true}>

    Step five is mandatory when using distributed evaluation

</Tip>

In most cases this is all that is needed. The next section lists a few more advanced use cases and nice
you should search for and replace by the corresponding methods of your `accelerator`:

## Advanced recommendations

### Printing

`print` statements should be replaced by [`~Accelerator.print`] to be printed once per process

```diff
- print("My thing I want to print!")
+ accelerator.print("My thing I want to print!")
~~~

### Executing processes

#### Once on a single server

For statements that should be executed once per server, use [`~Accelerator.is_local_main_process`]:

```python
if accelerator.is_local_main_process:
    do_thing_once_per_server()
```

A function can be wrapped using the [`~Accelerator.on_local_main_process`] function to achieve the same
behavior on a function's execution:
```python
```

~~~~
@accelerator.on_local_main_process
def do_my_thing():
    "Something done once per server"
    do_thing_once_per_server()
```

#### Only ever once across all servers

For statements that should only ever be executed once, use [`~Accelerator.is_main_process`]:

```python
if accelerator.is_main_process:
    do_thing_once()
```

A function can be wrapped using the [`~Accelerator.on_main_process`] function to achieve the same
behavior on a function's execution:

```python
@accelerator.on_main_process
def do_my_thing():
    "Something done once per server"
    do_thing_once()
```

#### On specific processes

If a function should be ran on a specific overall or local process index, there are similar decorators
to achieve this:

```python
@accelerator.on_local_process(local_process_idx=0)
def do_my_thing():
    "Something done on process index 0 on each server"
    do_thing_on_index_zero_on_each_server()
```

```python
@accelerator.on_process(process_index=0)
def do_my_thing():
    "Something done on process index 0"
    do_thing_on_index_zero()
```

### Synchronicity control

Use [`~Accelerator.wait_for_everyone`] to make sure all processes join that point before continuing. (Us

### Saving and loading

Use [`~Accelerator.unwrap_model`] before saving to remove all special model wrappers added during the di

```python
model = MyModel()
model = accelerator.prepare(model)
# Unwrap
model = accelerator.unwrap_model(model)
```

Use [`~Accelerator.save`] instead of `torch.save`:

```diff
  state_dict = model.state_dict()
- torch.save(state_dict, "my_state.pkl")
+ accelerator.save(state_dict, "my_state.pkl")
```

### Operations

Use [`~Accelerator.clip_grad_norm_`] instead of ``torch.nn.utils.clip_grad_norm_`` and [`~Accelerator.cl

### Gradient Accumulation
~~~~

To perform gradient accumulation use [`~Accelerator.accumulate`] and specify a gradient_accumulation_ste
This will also automatically ensure the gradients are synced or unsynced when on
multi-device training, check if the step should actually be performed, and auto-scale the loss:

```diff
- accelerator = Accelerator()
+ accelerator = Accelerator(gradient_accumulation_steps=2)

  for (input, label) in training_dataloader:
+     with accelerator.accumulate(model):
          predictions = model(input)
          loss = loss_function(predictions, labels)
          accelerator.backward(loss)
          optimizer.step()
          scheduler.step()
          optimizer.zero_grad()
```

## Overall API documentation:

[[autodoc]] Accelerator

accelerate-main/docs/source/package_reference/kwargs.mdx

```
<!--Copyright 2021 The HuggingFace Team. All rights reserved.

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in comp
the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is dis
an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the Licen
specific language governing permissions and limitations under the License.
-->

# Kwargs Handlers

The following objects can be passed to the main [`Accelerator`] to customize how some PyTorch objects
related to distributed training or mixed precision are created.


## DistributedDataParallelKwargs

[[autodoc]] DistributedDataParallelKwargs

## GradScalerKwargs

[[autodoc]] GradScalerKwargs

## InitProcessGroupKwargs

[[autodoc]] InitProcessGroupKwargs
```

accelerate-main/docs/source/package_reference/cli.mdx

~~~
<!--Copyright 2021 The HuggingFace Team. All rights reserved.

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in comp
the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is dis
an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the Licen
specific language governing permissions and limitations under the License.
-->

# The Command Line

Below is a list of all the available commands ■ Accelerate with their parameters

## accelerate config

**Command**:

`accelerate config` or `accelerate-config`

Launches a series of prompts to create and save a `default_config.yml` configuration file for your train
always be ran first on your machine.

**Usage**:

```bash
accelerate config [arguments]
~~~

Optional Arguments:
* `--config_file CONFIG_FILE` (`str`) -- The path to use to store the config file. Will default to a fil
                        of the environment `HF_HOME` suffixed with 'accelerate', or if you don't have su
                        (`~/.cache` or the content of `XDG_CACHE_HOME`) suffixed with `huggingface`.
* `-h`, `--help` (`bool`) -- Show a help message and exit

## accelerate config default

Command:

`accelerate config default` or `accelerate-config default`

Create a default config file for Accelerate with only a few flags set.

Usage:

```bash
accelerate config default [arguments]
```

Optional Arguments:
* `--config_file CONFIG_FILE` (`str`) -- The path to use to store the config file. Will default to a fil
                        of the environment `HF_HOME` suffixed with 'accelerate', or if you don't have su
                        (`~/.cache` or the content of `XDG_CACHE_HOME`) suffixed with `huggingface`.

* `-h`, `--help` (`bool`) -- Show a help message and exit
* `--mixed_precision {no,fp16,bf16}` (`str`) -- Whether or not to use mixed precision training. Choose b

## accelerate config update

Command:

`accelerate config update` or `accelerate-config update`

Update an existing config file with the latest defaults while maintaining the old configuration.

Usage:
```
```

```bash
accelerate config update [arguments]
```

Optional Arguments:
* `--config_file CONFIG_FILE` (`str`) -- The path to the config file to update. Will default to a file n
                        of the environment `HF_HOME` suffixed with 'accelerate', or if you don't have su
                        (`~/.cache` or the content of `XDG_CACHE_HOME`) suffixed with `huggingface`.

* `-h`, `--help` (`bool`) -- Show a help message and exit

## accelerate env

Command:

`accelerate env` or `accelerate-env`

Lists the contents of the passed ■ Accelerate configuration file. Should always be used when opening an

Usage:

```bash
accelerate env [arguments]
```

Optional Arguments:
* `--config_file CONFIG_FILE` (`str`) -- The path to use to store the config file. Will default to a fil
                        of the environment `HF_HOME` suffixed with 'accelerate', or if you don't have su
                        (`~/.cache` or the content of `XDG_CACHE_HOME`) suffixed with `huggingface`.
* `-h`, `--help` (`bool`) -- Show a help message and exit

## accelerate launch

Command:

`accelerate launch` or `accelerate-launch`

Launches a specified script on a distributed system with the right parameters.

Usage:

```bash
accelerate launch [arguments] {training_script} --{training_script-argument-1} --{training_script-argume
```

Positional Arguments:

- `{training_script}` -- The full path to the script to be launched in parallel
- `--{training_script-argument-1}` -- Arguments of the training script

Optional Arguments:

* `-h`, `--help` (`bool`) -- Show a help message and exit
* `--config_file CONFIG_FILE` (`str`)-- The config file to use for the default values in the launching s
* `-m`, `--module` (`bool`) -- Change each process to interpret the launch script as a Python module, ex
* `--no_python` (`bool`) -- Skip prepending the training script with 'python' - just execute it directly
* `--debug` (`bool`) -- Whether to print out the torch.distributed stack trace when something fails.
* `-q`, `--quiet` (`bool`) -- Silence subprocess errors from the launch stack trace to only show the rel

The rest of these arguments are configured through `accelerate config` and are read in from the specifie
values. They can also be passed in manually.

Hardware Selection Arguments:

* `--cpu` (`bool`) -- Whether or not to force the training on the CPU.
* `--multi_gpu` (`bool`) -- Whether or not this should launch a distributed GPU training.
* `--mps` (`bool`) -- Whether or not this should use MPS-enabled GPU device on MacOS machines.
* `--tpu` (`bool`) -- Whether or not this should launch a TPU training.

Resource Selection Arguments:

The following arguments are useful for fine-tuning how available hardware should be used

* `--mixed_precision {no,fp16,bf16}` (`str`) -- Whether or not to use mixed precision training. Choose b
* `--num_processes NUM_PROCESSES` (`int`) -- The total number of processes to be launched in parallel.
* `--num_machines NUM_MACHINES` (`int`) -- The total number of machines used in this training.
* `--num_cpu_threads_per_process NUM_CPU_THREADS_PER_PROCESS` (`int`) -- The number of CPU threads per p

Training Paradigm Arguments:

The following arguments are useful for selecting which training paradigm to use.

* `--use_deepspeed` (`bool`) -- Whether or not to use DeepSpeed for training.
* `--use_fsdp` (`bool`) -- Whether or not to use FullyShardedDataParallel for training.
* `--use_megatron_lm` (`bool`) -- Whether or not to use Megatron-LM for training.

Distributed GPU Arguments:

The following arguments are only useful when `multi_gpu` is passed or multi-gpu training is configured t

* `--gpu_ids` (`str`) -- What GPUs (by id) should be used for training on this machine as a comma-sepera
* `--same_network` (`bool`) -- Whether all machines used for multinode training exist on the same local
* `--machine_rank MACHINE_RANK` (`int`) -- The rank of the machine on which this script is launched.
* `--main_process_ip MAIN_PROCESS_IP` (`str`) -- The IP address of the machine of rank 0.
* `--main_process_port MAIN_PROCESS_PORT` (`int`) -- The port to use to communicate with the machine of
* `--rdzv_conf` (`str`) -- Additional rendezvous configuration (<key1>=<value1>,<key2>=<value2>,...).
* `--max_restarts` (`int`) -- Maximum number of worker group restarts before failing.
* `--monitor_interval` (`float`) -- Interval, in seconds, to monitor the state of workers.

TPU Arguments:

The following arguments are only useful when `tpu` is passed or TPU training is configured through `acce

* `--main_training_function MAIN_TRAINING_FUNCTION` (`str`) -- The name of the main function to be execu
* `--downcast_bf16` (`bool`) -- Whether when using bf16 precision on TPUs if both float and double tenso

DeepSpeed Arguments:

The following arguments are only useful when `use_deepspeed` is passed or `deepspeed` is configured thro

* `--deepspeed_config_file` (`str`) -- DeepSpeed config file.
* `--zero_stage` (`int`) -- DeepSpeed's ZeRO optimization stage.
* `--offload_optimizer_device` (`str`) -- Decides where (none|cpu|nvme) to offload optimizer states.
* `--offload_param_device` (`str`) -- Decides where (none|cpu|nvme) to offload parameters.
* `--gradient_accumulation_steps` (`int`) -- No of gradient_accumulation_steps used in your training scr
* `--gradient_clipping` (`float`) -- Gradient clipping value used in your training script.
* `--zero3_init_flag` (`str`) -- Decides Whether (true|false) to enable `deepspeed.zero.Init` for constr
* `--zero3_save_16bit_model` (`str`) -- Decides Whether (true|false) to save 16-bit model weights when u
* `--deepspeed_hostfile` (`str`) -- DeepSpeed hostfile for configuring multi-node compute resources.
* `--deepspeed_exclusion_filter` (`str`) -- DeepSpeed exclusion filter string when using mutli-node setu
* `--deepspeed_inclusion_filter` (`str`) -- DeepSpeed inclusion filter string when using mutli-node setu
* `--deepspeed_multinode_launcher` (`str`) -- DeepSpeed multi-node launcher to use.

Fully Sharded Data Parallelism Arguments:

The following arguments are only useful when `use_fdsp` is passed or Fully Sharded Data Parallelism is c

* `--fsdp_offload_params` (`str`) -- Decides Whether (true|false) to offload parameters and gradients to
* `--fsdp_min_num_params` (`int`) -- FSDP's minimum number of parameters for Default Auto Wrapping.
* `--fsdp_sharding_strategy` (`int`) -- FSDP's Sharding Strategy.
* `--fsdp_auto_wrap_policy` (`str`) -- FSDP's auto wrap policy.
* `--fsdp_transformer_layer_cls_to_wrap` (`str`) -- Transformer layer class name (case-sensitive) to wra
* `--fsdp_backward_prefetch_policy` (`str`) -- FSDP's backward prefetch policy.
* `--fsdp_state_dict_type` (`str`) -- FSDP's state dict type.

Megatron-LM Arguments:

The following arguments are only useful when `use_megatron_lm` is passed or Megatron-LM is configured th

* `--megatron_lm_tp_degree` (``) -- Megatron-LM's Tensor Parallelism (TP) degree.
* `--megatron_lm_pp_degree` (``) -- Megatron-LM's Pipeline Parallelism (PP) degree.
* `--megatron_lm_num_micro_batches` (``) -- Megatron-LM's number of micro batches when PP degree > 1.
* `--megatron_lm_sequence_parallelism` (``) -- Decides Whether (true|false) to enable Sequence Paralleli

* `--megatron_lm_recompute_activations` (``) -- Decides Whether (true|false) to enable Selective Activat
* `--megatron_lm_use_distributed_optimizer` (``) -- Decides Whether (true|false) to use distributed opti
* `--megatron_lm_gradient_clipping` (``) -- Megatron-LM's gradient clipping value based on global L2 Nor

AWS SageMaker Arguments:

The following arguments are only useful when training in SageMaker

* `--aws_access_key_id AWS_ACCESS_KEY_ID` (`str`) -- The AWS_ACCESS_KEY_ID used to launch the Amazon Sag
* `--aws_secret_access_key AWS_SECRET_ACCESS_KEY` (`str`) -- The AWS_SECRET_ACCESS_KEY used to launch th

## accelerate tpu-config

`accelerate tpu-config`

Usage:

```bash
accelerate tpu-config [arguments]
```

Optional Arguments:
* `-h`, `--help` (`bool`) -- Show a help message and exit

Config Arguments:

Arguments that can be configured through `accelerate config`.

* `--config_file` (`str`) -- Path to the config file to use for accelerate.
* `--tpu_name` (`str`) -- The name of the TPU to use. If not specified, will use the TPU specified in th
* `--tpu_zone` (`str`) -- The zone of the TPU to use. If not specified, will use the zone specified in t

TPU Arguments:

Arguments for options ran inside the TPU.

* `--command_file` (`str`) -- The path to the file containing the commands to run on the pod on startup.
* `--command` (`str`) -- A command to run on the pod. Can be passed multiple times.
* `--install_accelerate` (`bool`) -- Whether to install accelerate on the pod. Defaults to False.
* `--accelerate_version` (`str`) -- The version of accelerate to install on the pod. If not specified, w
* `--debug` (`bool`) -- If set, will print the command that would be run instead of running it.

## accelerate test

`accelerate test` or `accelerate-test`

Runs `accelerate/test_utils/test_script.py` to verify that ■ Accelerate has been properly configured on

Usage:

```bash
accelerate test [arguments]
```

Optional Arguments:
* `--config_file CONFIG_FILE` (`str`) -- The path to use to store the config file. Will default to a fil
                        of the environment `HF_HOME` suffixed with 'accelerate', or if you don't have su
                        (`~/.cache` or the content of `XDG_CACHE_HOME`) suffixed with `huggingface`.
* `-h`, `--help` (`bool`) -- Show a help message and exit

accelerate-main/docs/source/package_reference/big_modeling.mdx

```
<!--Copyright 2021 The HuggingFace Team. All rights reserved.

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in comp
the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is dis
an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the Licen
specific language governing permissions and limitations under the License.
-->

# Working with large models

## Dispatching and Offloading Models

[[autodoc]] big_modeling.init_empty_weights
[[autodoc]] big_modeling.cpu_offload
[[autodoc]] big_modeling.disk_offload
[[autodoc]] big_modeling.dispatch_model
[[autodoc]] big_modeling.load_checkpoint_and_dispatch

## Model Hooks

### Hook Classes

[[autodoc]] hooks.ModelHook
[[autodoc]] hooks.AlignDevicesHook
[[autodoc]] hooks.SequentialHook

### Adding Hooks

[[autodoc]] hooks.add_hook_to_module
[[autodoc]] hooks.attach_execution_device_hook
[[autodoc]] hooks.attach_align_device_hook
[[autodoc]] hooks.attach_align_device_hook_on_blocks

### Removing Hooks

[[autodoc]] hooks.remove_hook_from_module
[[autodoc]] hooks.remove_hook_from_submodules
```

accelerate-main/docs/source/package_reference/megatron_lm.mdx

```
<!--Copyright 2021 The HuggingFace Team. All rights reserved.

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in comp
the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is dis
an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the Licen
specific language governing permissions and limitations under the License.
-->

# Utilities for Megatron-LM

[[autodoc]] utils.MegatronLMPlugin

[[autodoc]] utils.MegatronLMDummyScheduler

[[autodoc]] utils.MegatronLMDummyDataLoader

[[autodoc]] utils.AbstractTrainStep

[[autodoc]] utils.GPTTrainStep

[[autodoc]] utils.BertTrainStep

[[autodoc]] utils.T5TrainStep

[[autodoc]] utils.avg_losses_across_data_parallel_group
```

accelerate-main/docs/source/package_reference/kwargs.mdx

```
<!--Copyright 2021 The HuggingFace Team. All rights reserved.

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in comp
the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is dis
an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the Licen
specific language governing permissions and limitations under the License.
-->

# Kwargs Handlers

The following objects can be passed to the main [`Accelerator`] to customize how some PyTorch objects
related to distributed training or mixed precision are created.


## DistributedDataParallelKwargs

[[autodoc]] DistributedDataParallelKwargs

## GradScalerKwargs

[[autodoc]] GradScalerKwargs

## InitProcessGroupKwargs

[[autodoc]] InitProcessGroupKwargs
```

accelerate-main/docs/source/package_reference/launchers.mdx

```
# Launchers

Functions for launching training on distributed processes.


[[autodoc]] accelerate.notebook_launcher
[[autodoc]] accelerate.debug_launcher
```

accelerate-main/docker/accelerate-cpu/Dockerfile

```
# Builds CPU-only Docker image of PyTorch
# Uses multi-staged approach to reduce size
# Stage 1
FROM python:3.7-slim as compile-image

ARG DEBIAN_FRONTEND=noninteractive

RUN apt update
RUN apt-get install -y --no-install-recommends \
    build-essential \
    git \
    gcc

# Setup virtual environment for Docker
ENV VIRTUAL_ENV=/opt/venv
RUN python3 -m venv ${VIRTUAL_ENV}
# Make sure we use the virtualenv
ENV PATH="${VIRTUAL_ENV}/bin:$PATH"
WORKDIR /workspace
# Install specific CPU torch wheel to save on space
RUN python3 -m pip install --upgrade --no-cache-dir pip
RUN python3 -m pip install --no-cache-dir \
    jupyter \
    git+https://github.com/huggingface/accelerate#egg=accelerate[testing,test_trackers] \
    --extra-index-url https://download.pytorch.org/whl/cpu

# Stage 2
FROM python:3.7-slim AS build-image
COPY --from=compile-image /opt/venv /opt/venv
RUN useradd -ms /bin/bash user
USER user

# Make sure we use the virtualenv
ENV PATH="/opt/venv/bin:$PATH"
CMD ["/bin/bash"]
```

accelerate-main/docker/accelerate-cpu/Dockerfile

```
# Builds CPU-only Docker image of PyTorch
# Uses multi-staged approach to reduce size
# Stage 1
FROM python:3.7-slim as compile-image

ARG DEBIAN_FRONTEND=noninteractive

RUN apt update
RUN apt-get install -y --no-install-recommends \
    build-essential \
    git \
    gcc

# Setup virtual environment for Docker
ENV VIRTUAL_ENV=/opt/venv
RUN python3 -m venv ${VIRTUAL_ENV}
# Make sure we use the virtualenv
ENV PATH="${VIRTUAL_ENV}/bin:$PATH"
WORKDIR /workspace
# Install specific CPU torch wheel to save on space
RUN python3 -m pip install --upgrade --no-cache-dir pip
RUN python3 -m pip install --no-cache-dir \
    jupyter \
    git+https://github.com/huggingface/accelerate#egg=accelerate[testing,test_trackers] \
    --extra-index-url https://download.pytorch.org/whl/cpu

# Stage 2
FROM python:3.7-slim AS build-image
COPY --from=compile-image /opt/venv /opt/venv
RUN useradd -ms /bin/bash user
USER user

# Make sure we use the virtualenv
ENV PATH="/opt/venv/bin:$PATH"
CMD ["/bin/bash"]
```

accelerate-main/manim_animations/big_model_inference/stage_5.py

```
# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

from manim import *

class Stage5(Scene):
    def construct(self):
        mem = Rectangle(height=0.5,width=0.5)
        fill = Rectangle(height=0.46,width=0.46).set_stroke(width=0)

        meta_mem = Rectangle(height=0.25,width=0.25)

        cpu_left_col_base = [mem.copy() for i in range(6)]
        cpu_right_col_base = [mem.copy() for i in range(6)]
        cpu_left_col = VGroup(*cpu_left_col_base).arrange(UP, buff=0)
        cpu_right_col = VGroup(*cpu_right_col_base).arrange(UP, buff=0)
        cpu_rects = VGroup(cpu_left_col,cpu_right_col).arrange(RIGHT, buff=0)
        cpu_text = Text("CPU", font_size=24)
        cpu = Group(cpu_rects,cpu_text).arrange(DOWN, buff=0.5, aligned_edge=DOWN)
        cpu.move_to([-2.5,-.5,0])
        self.add(cpu)

        gpu_base = [mem.copy() for i in range(4)]
        gpu_rect = VGroup(*gpu_base).arrange(UP,buff=0)
        gpu_text = Text("GPU", font_size=24)
        gpu = Group(gpu_rect,gpu_text).arrange(DOWN, buff=0.5, aligned_edge=DOWN)
        gpu.move_to([-1,-1,0])
        self.add(gpu)

        model_base = [mem.copy() for i in range(6)]
        model_rect = VGroup(*model_base).arrange(RIGHT,buff=0)

        model_text = Text("Model", font_size=24)
        model = Group(model_rect,model_text).arrange(DOWN, buff=0.5, aligned_edge=DOWN)
        model.move_to([3, -1., 0])
        self.add(model)

        model_arr = []
        model_cpu_arr = []

        for i,rect in enumerate(model_base):
            target = fill.copy().set_fill(BLUE, opacity=0.8)
            target.move_to(rect)
            model_arr.append(target)

            cpu_target = Rectangle(height=0.46,width=0.46).set_stroke(width=0.).set_fill(BLUE, opacity=0
            cpu_target.move_to(cpu_left_col_base[i])
            model_cpu_arr.append(cpu_target)

        self.add(*model_arr, *model_cpu_arr)

        disk_left_col_base = [meta_mem.copy() for i in range(6)]
        disk_right_col_base = [meta_mem.copy() for i in range(6)]
        disk_left_col = VGroup(*disk_left_col_base).arrange(UP, buff=0)
        disk_right_col = VGroup(*disk_right_col_base).arrange(UP, buff=0)
```

```
disk_rects = VGroup(disk_left_col,disk_right_col).arrange(RIGHT, buff=0)
disk_text = Text("Disk", font_size=24)
disk = Group(disk_rects,disk_text).arrange(DOWN, buff=0.5, aligned_edge=DOWN)
disk.move_to([-4,-1.25,0])
self.add(disk_text, disk_rects)

key = Square(side_length=2.2)
key.move_to([-5, 2, 0])

key_text = MarkupText(
    f"<b>Key:</b>\n\n<span fgcolor='{YELLOW}'>●</span> Empty Model",
    font_size=18,
)

key_text.move_to([-5, 2.4, 0])

self.add(key_text, key)

blue_text = MarkupText(
    f"<span fgcolor='{BLUE}'>●</span> Checkpoint",
    font_size=18,
)

blue_text.next_to(key_text, DOWN*2.4, aligned_edge=key_text.get_left())
self.add(blue_text)

step_6 = MarkupText(
    f'Now watch as an input is passed through the model\nand how the memory is utilized and hand
    font_size=24
)
step_6.move_to([2, 2, 0])

self.play(Write(step_6))

input = Square(0.3)
input.set_fill(RED, opacity=1.)
input.set_stroke(width=0.)
input.next_to(model_base[0], LEFT, buff=.5)

self.play(Write(input))

input.generate_target()
input.target.next_to(model_arr[0], direction=LEFT, buff=0.02)
self.play(MoveToTarget(input))

self.play(FadeOut(step_6))


a = Arrow(start=UP, end=DOWN, color=RED, buff=.5)
a.next_to(model_arr[0].get_left(), UP, buff=0.2)

model_cpu_arr[0].generate_target()
model_cpu_arr[0].target.move_to(gpu_rect[0])

step_7 = MarkupText(
    f'As the input reaches a layer, the hook triggers\nand weights are moved from the CPU\nto th
    font_size=24
)
step_7.move_to([2, 2, 0])

self.play(Write(step_7, run_time=3))

circ_kwargs = {"run_time":1, "fade_in":True, "fade_out":True, "buff":0.02}

self.play(
    Write(a),
    Circumscribe(model_arr[0], color=ORANGE, **circ_kwargs),
    Circumscribe(model_cpu_arr[0], color=ORANGE, **circ_kwargs),
    Circumscribe(gpu_rect[0], color=ORANGE, **circ_kwargs),
)
self.play(
    MoveToTarget(model_cpu_arr[0])
```

```
)

a_c = a.copy()
for i in range(6):
    a_c.next_to(model_arr[i].get_right()+0.02, UP, buff=0.2)

    input.generate_target()
    input.target.move_to(model_arr[i].get_right()+0.02)

    grp = AnimationGroup(
        FadeOut(a, run_time=.5),
        MoveToTarget(input, run_time=.5),
        FadeIn(a_c, run_time=.5),
        lag_ratio=0.2
    )

    self.play(grp)


    model_cpu_arr[i].generate_target()
    model_cpu_arr[i].target.move_to(cpu_left_col_base[i])


    if i < 5:
        model_cpu_arr[i+1].generate_target()
        model_cpu_arr[i+1].target.move_to(gpu_rect[0])
        if i >= 1:
            circ_kwargs["run_time"] = .7

        self.play(
            Circumscribe(model_arr[i], **circ_kwargs),
            Circumscribe(cpu_left_col_base[i], **circ_kwargs),
            Circumscribe(cpu_left_col_base[i+1], color=ORANGE, **circ_kwargs),
            Circumscribe(gpu_rect[0], color=ORANGE, **circ_kwargs),
            Circumscribe(model_arr[i+1], color=ORANGE, **circ_kwargs),
        )
        if i < 1:
            self.play(
                MoveToTarget(model_cpu_arr[i]),
                MoveToTarget(model_cpu_arr[i+1]),
            )
        else:
            self.play(
                MoveToTarget(model_cpu_arr[i], run_time=.7),
                MoveToTarget(model_cpu_arr[i+1], run_time=.7),
            )
    else:
        model_cpu_arr[i].generate_target()
        model_cpu_arr[i].target.move_to(cpu_left_col_base[-1])
        input.generate_target()
        input.target.next_to(model_arr[-1].get_right(), RIGHT+0.02, buff=0.2)

        self.play(
            Circumscribe(model_arr[-1], color=ORANGE, **circ_kwargs),
            Circumscribe(cpu_left_col_base[-1], color=ORANGE, **circ_kwargs),
            Circumscribe(gpu_rect[0], color=ORANGE, **circ_kwargs),
        )

        self.play(
            MoveToTarget(model_cpu_arr[i])
        )

    a = a_c
    a_c = a_c.copy()

input.generate_target()
input.target.next_to(model_base[-1], RIGHT+0.02, buff=.5)
self.play(
    FadeOut(step_7),
    FadeOut(a, run_time=.5),
)
step_8 = MarkupText(
    f'Inference on a model too large for GPU memory\nis successfully completed.', font_size=24
)
step_8.move_to([2, 2, 0])

self.play(
    Write(step_8, run_time=3),
    MoveToTarget(input)
)

self.wait()
```

accelerate-main/manim_animations/big_model_inference/stage_1.py

```
# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

from manim import *


class Stage1(Scene):
    def construct(self):
        mem = Rectangle(height=0.5,width=0.5)
        fill = Rectangle(height=0.46,width=0.46).set_stroke(width=0)

        cpu_left_col_base = [mem.copy() for i in range(6)]
        cpu_right_col_base = [mem.copy() for i in range(6)]
        cpu_left_col = VGroup(*cpu_left_col_base).arrange(UP, buff=0)
        cpu_right_col = VGroup(*cpu_right_col_base).arrange(UP, buff=0)
        cpu_rects = VGroup(cpu_left_col,cpu_right_col).arrange(RIGHT, buff=0)
        cpu_text = Text("CPU", font_size=24)
        cpu = Group(cpu_rects,cpu_text).arrange(DOWN, buff=0.5, aligned_edge=DOWN)
        cpu.move_to([-2.5,-.5,0])
        self.add(cpu)

        gpu_base = [mem.copy() for i in range(1)]
        gpu_rect = VGroup(*gpu_base).arrange(UP,buff=0)
        gpu_text = Text("GPU", font_size=24)
        gpu = Group(gpu_rect,gpu_text).arrange(DOWN, buff=0.5, aligned_edge=DOWN)
        gpu.align_to(cpu, DOWN)
        gpu.set_x(gpu.get_x() - 1)

        self.add(gpu)

        model_base = [mem.copy() for i in range(6)]
        model_rect = VGroup(*model_base).arrange(RIGHT,buff=0)

        model_text = Text("Model", font_size=24)
        model = Group(model_rect,model_text).arrange(DOWN, buff=0.5, aligned_edge=DOWN)
        model.move_to([3, -1., 0])

        self.play(
            Create(cpu_left_col, run_time=1),
            Create(cpu_right_col, run_time=1),
            Create(gpu_rect, run_time=1),
        )

        step_1 = MarkupText(
            f"First, an empty model skeleton is loaded\ninto <span fgcolor='{YELLOW}'>memory</span> with
            font_size=24
        )

        key = Square(side_length=2.2)
        key.move_to([-5, 2, 0])

        key_text = MarkupText(
            f"<b>Key:</b>\n\n<span fgcolor='{YELLOW}'>●</span> Empty Model",
            font_size=18,
        )
```

```
key_text.move_to([-5, 2.4, 0])


step_1.move_to([2, 2, 0])
self.play(
    Write(step_1, run_time=2.5),
    Write(key_text),
    Write(key)
)

self.add(model)


cpu_targs = []
first_animations = []
second_animations = []
for i,rect in enumerate(model_base):

    cpu_target = Rectangle(height=0.46,width=0.46).set_stroke(width=0.).set_fill(YELLOW, opacity
    cpu_target.move_to(rect)
    cpu_target.generate_target()
    cpu_target.target.height = 0.46/4
    cpu_target.target.width = 0.46/3

    if i == 0:
        cpu_target.target.next_to(cpu_left_col_base[0].get_corner(DOWN+LEFT), buff=0.02, directi
        cpu_target.target.set_x(cpu_target.target.get_x()+0.1)
    elif i == 3:
        cpu_target.target.next_to(cpu_targs[0].target, direction=UP, buff=0.)
    else:
        cpu_target.target.next_to(cpu_targs[i-1].target, direction=RIGHT, buff=0.)
    cpu_targs.append(cpu_target)

    first_animations.append(rect.animate(run_time=0.5).set_stroke(YELLOW))
    second_animations.append(MoveToTarget(cpu_target, run_time=1.5))

self.play(*first_animations)
self.play(*second_animations)


self.wait()
```

accelerate-main/manim_animations/big_model_inference/stage_2.py

```
# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

from manim import *

class Stage2(Scene):
    def construct(self):
        mem = Rectangle(height=0.5,width=0.5)
        fill = Rectangle(height=0.46,width=0.46).set_stroke(width=0)

        cpu_left_col_base = [mem.copy() for i in range(6)]
        cpu_right_col_base = [mem.copy() for i in range(6)]
        cpu_left_col = VGroup(*cpu_left_col_base).arrange(UP, buff=0)
        cpu_right_col = VGroup(*cpu_right_col_base).arrange(UP, buff=0)
        cpu_rects = VGroup(cpu_left_col,cpu_right_col).arrange(RIGHT, buff=0)
        cpu_text = Text("CPU", font_size=24)
        cpu = Group(cpu_rects,cpu_text).arrange(DOWN, buff=0.5, aligned_edge=DOWN)
        cpu.move_to([-2.5,-.5,0])
        self.add(cpu)

        gpu_base = [mem.copy() for i in range(4)]
        gpu_rect = VGroup(*gpu_base).arrange(UP,buff=0)
        gpu_text = Text("GPU", font_size=24)
        gpu = Group(gpu_rect,gpu_text).arrange(DOWN, buff=0.5, aligned_edge=DOWN)
        gpu.move_to([-1,-1,0])
        self.add(gpu)

        model_base = [mem.copy() for i in range(6)]
        model_rect = VGroup(*model_base).arrange(RIGHT,buff=0)

        model_text = Text("Model", font_size=24)
        model = Group(model_rect,model_text).arrange(DOWN, buff=0.5, aligned_edge=DOWN)
        model.move_to([3, -1., 0])
        self.add(model)

        cpu_targs = []
        for i,rect in enumerate(model_base):
            rect.set_stroke(YELLOW)
            # target = fill.copy().set_fill(YELLOW, opacity=0.7)
            # target.move_to(rect)
            # self.add(target)

            cpu_target = Rectangle(height=0.46/4,width=0.46/3).set_stroke(width=0.).set_fill(YELLOW, opa

            if i == 0:
                cpu_target.next_to(cpu_left_col_base[0].get_corner(DOWN+LEFT), buff=0.02, direction=UP)
                cpu_target.set_x(cpu_target.get_x()+0.1)
            elif i == 3:
                cpu_target.next_to(cpu_targs[0], direction=UP, buff=0.)
            else:
                cpu_target.next_to(cpu_targs[i-1], direction=RIGHT, buff=0.)
            self.add(cpu_target)
            cpu_targs.append(cpu_target)
        checkpoint_base = [mem.copy() for i in range(6)]
        checkpoint_rect = VGroup(*checkpoint_base).arrange(RIGHT,buff=0)
```

```
checkpoint_text = Text("Loaded Checkpoint", font_size=24)
checkpoint = Group(checkpoint_rect,checkpoint_text).arrange(DOWN, aligned_edge=DOWN, buff=0.4)
checkpoint.move_to([3, .5, 0])

key = Square(side_length=2.2)
key.move_to([-5, 2, 0])

key_text = MarkupText(
    f"<b>Key:</b>\n\n<span fgcolor='{YELLOW}'>●</span> Empty Model",
    font_size=18,
)

key_text.move_to([-5, 2.4, 0])

self.add(key_text, key)

blue_text = MarkupText(
    f"<span fgcolor='{BLUE}'>●</span> Checkpoint",
    font_size=18,
)

blue_text.next_to(key_text, DOWN*2.4, aligned_edge=key_text.get_left())

step_2 = MarkupText(
    f'Next, a <i><span fgcolor="{BLUE}">second</span></i> model is loaded into memory,\nwith the
    font_size=24
)
step_2.move_to([2, 2, 0])
self.play(
    Write(step_2),
    Write(blue_text)
)

self.play(
    Write(checkpoint_text, run_time=1),
    Create(checkpoint_rect, run_time=1)
)

first_animations = []
second_animations = []
for i,rect in enumerate(checkpoint_base):
    target = fill.copy().set_fill(BLUE, opacity=0.7)
    target.move_to(rect)
    first_animations.append(GrowFromCenter(target, run_time=1))

    cpu_target = target.copy()
    cpu_target.generate_target()
    if i < 5:
        cpu_target.target.move_to(cpu_left_col_base[i+1])
    else:
        cpu_target.target.move_to(cpu_right_col_base[i-5])
    second_animations.append(MoveToTarget(cpu_target, run_time=1.5))

self.play(*first_animations)
self.play(*second_animations)
self.wait()
```

accelerate-main/manim_animations/big_model_inference/stage_3.py

```
# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

from manim import *

class Stage3(Scene):
    def construct(self):
        mem = Rectangle(height=0.5,width=0.5)
        meta_mem = Rectangle(height=0.25,width=0.25)
        fill = Rectangle(height=0.46,width=0.46).set_stroke(width=0)

        cpu_left_col_base = [mem.copy() for i in range(6)]
        cpu_right_col_base = [mem.copy() for i in range(6)]
        cpu_left_col = VGroup(*cpu_left_col_base).arrange(UP, buff=0)
        cpu_right_col = VGroup(*cpu_right_col_base).arrange(UP, buff=0)
        cpu_rects = VGroup(cpu_left_col,cpu_right_col).arrange(RIGHT, buff=0)
        cpu_text = Text("CPU", font_size=24)
        cpu = Group(cpu_rects,cpu_text).arrange(DOWN, buff=0.5, aligned_edge=DOWN)
        cpu.move_to([-2.5,-.5,0])
        self.add(cpu)

        gpu_base = [mem.copy() for i in range(4)]
        gpu_rect = VGroup(*gpu_base).arrange(UP,buff=0)
        gpu_text = Text("GPU", font_size=24)
        gpu = Group(gpu_rect,gpu_text).arrange(DOWN, buff=0.5, aligned_edge=DOWN)
        gpu.move_to([-1,-1,0])
        self.add(gpu)

        model_base = [mem.copy() for i in range(6)]
        model_rect = VGroup(*model_base).arrange(RIGHT,buff=0)

        model_text = Text("Model", font_size=24)
        model = Group(model_rect,model_text).arrange(DOWN, buff=0.5, aligned_edge=DOWN)
        model.move_to([3, -1., 0])
        self.add(model)

        model_arr = []
        model_cpu_arr = []
        model_meta_arr = []

        for i,rect in enumerate(model_base):
            rect.set_stroke(YELLOW)

            cpu_target = Rectangle(height=0.46/4,width=0.46/3).set_stroke(width=0.).set_fill(YELLOW, opa

            if i == 0:
                cpu_target.next_to(cpu_left_col_base[0].get_corner(DOWN+LEFT), buff=0.02, direction=UP)
                cpu_target.set_x(cpu_target.get_x()+0.1)
            elif i == 3:
                cpu_target.next_to(model_cpu_arr[0], direction=UP, buff=0.)
            else:
                cpu_target.next_to(model_cpu_arr[i-1], direction=RIGHT, buff=0.)
            self.add(cpu_target)
            model_cpu_arr.append(cpu_target)
        self.add(*model_arr, *model_cpu_arr, *model_meta_arr)
```

```
checkpoint_base = [mem.copy() for i in range(6)]
checkpoint_rect = VGroup(*checkpoint_base).arrange(RIGHT,buff=0)

checkpoint_text = Text("Loaded Checkpoint", font_size=24)
checkpoint = Group(checkpoint_rect,checkpoint_text).arrange(DOWN, buff=0.5, aligned_edge=DOWN)
checkpoint.move_to([3, .5, 0])

self.add(checkpoint)

ckpt_arr = []
ckpt_cpu_arr = []

for i,rect in enumerate(checkpoint_base):
    target = fill.copy().set_fill(BLUE, opacity=0.7)
    target.move_to(rect)
    ckpt_arr.append(target)

    cpu_target = target.copy()
    if i < 5:
        cpu_target.move_to(cpu_left_col_base[i+1])
    else:
        cpu_target.move_to(cpu_right_col_base[i-5])
    ckpt_cpu_arr.append(cpu_target)
self.add(*ckpt_arr, *ckpt_cpu_arr)

key = Square(side_length=2.2)
key.move_to([-5, 2, 0])

key_text = MarkupText(
    f"<b>Key:</b>\n\n<span fgcolor='{YELLOW}'>●</span> Empty Model",
    font_size=18,
)

key_text.move_to([-5, 2.4, 0])

self.add(key_text, key)

blue_text = MarkupText(
    f"<span fgcolor='{BLUE}'>●</span> Checkpoint",
    font_size=18,
)

blue_text.next_to(key_text, DOWN*2.4, aligned_edge=key_text.get_left())
self.add(blue_text)

step_3 = MarkupText(
    f'Based on the passed in configuration, weights are stored in\na variety of np.memmaps on di
    font_size=24
)
step_3.move_to([2, 2, 0])

disk_left_col_base = [meta_mem.copy() for i in range(6)]
disk_right_col_base = [meta_mem.copy() for i in range(6)]
disk_left_col = VGroup(*disk_left_col_base).arrange(UP, buff=0)
disk_right_col = VGroup(*disk_right_col_base).arrange(UP, buff=0)
disk_rects = VGroup(disk_left_col,disk_right_col).arrange(RIGHT, buff=0)
disk_text = Text("Disk", font_size=24)
disk = Group(disk_rects,disk_text).arrange(DOWN, buff=0.5, aligned_edge=DOWN)
disk.move_to([-4.,-1.25,0])
self.play(
    Write(step_3, run_time=3),
    Write(disk_text, run_time=1),
    Create(disk_rects, run_time=1)
)

animations = []
for i,rect in enumerate(ckpt_cpu_arr):
    target = rect.copy()
    target.generate_target()
    target.target.move_to(disk_left_col_base[i]).scale(0.5)
    animations.append(MoveToTarget(target, run_time=1.5))
self.play(*animations)
self.play(FadeOut(step_3))

step_4 = MarkupText(
    f'Then, the checkpoint is removed from memory\nthrough garbage collection.',
    font_size=24
)
step_4.move_to([2, 2, 0])

self.play(
    Write(step_4, run_time=3)
)

self.play(
    FadeOut(checkpoint_rect, checkpoint_text, *ckpt_arr, *ckpt_cpu_arr),
)

self.wait()
```

accelerate-main/manim_animations/big_model_inference/stage_4.py

```
# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

from manim import *

class Stage4(Scene):
    def construct(self):
        mem = Rectangle(height=0.5,width=0.5)
        fill = Rectangle(height=0.46,width=0.46).set_stroke(width=0)
        meta_mem = Rectangle(height=0.25,width=0.25)

        cpu_left_col_base = [mem.copy() for i in range(6)]
        cpu_right_col_base = [mem.copy() for i in range(6)]
        cpu_left_col = VGroup(*cpu_left_col_base).arrange(UP, buff=0)
        cpu_right_col = VGroup(*cpu_right_col_base).arrange(UP, buff=0)
        cpu_rects = VGroup(cpu_left_col,cpu_right_col).arrange(RIGHT, buff=0)
        cpu_text = Text("CPU", font_size=24)
        cpu = Group(cpu_rects,cpu_text).arrange(DOWN, buff=0.5, aligned_edge=DOWN)
        cpu.move_to([-2.5,-.5,0])
        self.add(cpu)

        gpu_base = [mem.copy() for i in range(4)]
        gpu_rect = VGroup(*gpu_base).arrange(UP,buff=0)
        gpu_text = Text("GPU", font_size=24)
        gpu = Group(gpu_rect,gpu_text).arrange(DOWN, buff=0.5, aligned_edge=DOWN)
        gpu.move_to([-1,-1,0])
        self.add(gpu)

        model_base = [mem.copy() for i in range(6)]
        model_rect = VGroup(*model_base).arrange(RIGHT,buff=0)

        model_text = Text("Model", font_size=24)
        model = Group(model_rect,model_text).arrange(DOWN, buff=0.5, aligned_edge=DOWN)
        model.move_to([3, -1., 0])
        self.add(model)

        model_cpu_arr = []
        model_meta_arr = []

        for i,rect in enumerate(model_base):
            rect.set_stroke(YELLOW)

            cpu_target = Rectangle(height=0.46/4,width=0.46/3).set_stroke(width=0.).set_fill(YELLOW, opa

            if i == 0:
                cpu_target.next_to(cpu_left_col_base[0].get_corner(DOWN+LEFT), buff=0.02, direction=UP)
                cpu_target.set_x(cpu_target.get_x()+0.1)
            elif i == 3:
                cpu_target.next_to(model_cpu_arr[0], direction=UP, buff=0.)
            else:
                cpu_target.next_to(model_cpu_arr[i-1], direction=RIGHT, buff=0.)
            self.add(cpu_target)
            model_cpu_arr.append(cpu_target)

        self.add(*model_cpu_arr, *model_meta_arr)
```

```
disk_left_col_base = [meta_mem.copy() for i in range(6)]
disk_right_col_base = [meta_mem.copy() for i in range(6)]
disk_left_col = VGroup(*disk_left_col_base).arrange(UP, buff=0)
disk_right_col = VGroup(*disk_right_col_base).arrange(UP, buff=0)
disk_rects = VGroup(disk_left_col,disk_right_col).arrange(RIGHT, buff=0)
disk_text = Text("Disk", font_size=24)
disk = Group(disk_rects,disk_text).arrange(DOWN, buff=0.5, aligned_edge=DOWN)
disk.move_to([-4.,-1.25,0])
self.add(disk_text, disk_rects)

cpu_disk_arr = []

for i in range(6):
    target = fill.copy().set_fill(BLUE, opacity=0.8)
    target.move_to(disk_left_col_base[i]).scale(0.5)
    cpu_disk_arr.append(target)

self.add(*cpu_disk_arr)

key = Square(side_length=2.2)
key.move_to([-5, 2, 0])

key_text = MarkupText(
    f"<b>Key:</b>\n\n<span fgcolor='{YELLOW}'>●</span> Empty Model",
    font_size=18,
)

key_text.move_to([-5, 2.4, 0])

self.add(key_text, key)

blue_text = MarkupText(
    f"<span fgcolor='{BLUE}'>●</span> Checkpoint",
    font_size=18,
)

blue_text.next_to(key_text, DOWN*2.4, aligned_edge=key_text.get_left())
self.add(blue_text)

step_5 = MarkupText(
    f'The offloaded weights are all sent to the CPU.',
    font_size=24
)
step_5.move_to([2, 2, 0])

self.play(Write(step_5, run_time=3))

for i in range(6):
    rect = cpu_disk_arr[i]
    cp2 = rect.copy().set_fill(BLUE, opacity=0.8).scale(2.0)
    cp2.generate_target()
    cp2.target.move_to(model_base[i])

    if i == 0:
        rect.set_fill(BLUE, opacity=0.8)
        rect.generate_target()
        rect.target.move_to(cpu_left_col_base[0]).scale(2.0)

        self.remove(*model_meta_arr,
            *model_cpu_arr,
        )

    else:
        rect.generate_target()
        rect.target.move_to(cpu_left_col_base[i]).scale(2.0)
    self.play(
        MoveToTarget(rect),
        MoveToTarget(cp2),
        model_base[i].animate.set_stroke(WHITE)
    )
self.play(FadeOut(step_5))
step_5 = MarkupText(
    f'Finally, hooks are added to each weight in the model\nto transfer the weights from CPU to
    font_size=24
)
step_5.move_to([2, 2, 0])

self.play(Write(step_5, run_time=3))

arrows = []
animations = []
for i in range(6):
    a = Arrow(start=UP, end=DOWN, color=RED, buff=.5)
    a.next_to(model_base[i].get_left(), UP, buff=0.2)
    arrows.append(a)
    animations.append(Write(a))
self.play(*animations)
self.wait()
```

accelerate-main/examples/requirements.txt

```
accelerate # used to be installed in Amazon SageMaker environment
evaluate
datasets==2.3.2
```

accelerate-main/examples/cv_example.py

```
# coding=utf-8
# Copyright 2021 The HuggingFace Inc. team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
import argparse
import os
import re

import numpy as np
import PIL
import torch
from timm import create_model
from torch.optim.lr_scheduler import OneCycleLR
from torch.utils.data import DataLoader, Dataset
from torchvision.transforms import Compose, RandomResizedCrop, Resize, ToTensor

from accelerate import Accelerator


########################################################################
# This is a fully working simple example to use Accelerate
#
# This example trains a ResNet50 on the Oxford-IIT Pet Dataset
# in any of the following settings (with the same script):
#   - single CPU or single GPU
#   - multi GPUS (using PyTorch distributed mode)
#   - (multi) TPUs
#   - fp16 (mixed-precision) or fp32 (normal precision)
#
# To run it in each of these various modes, follow the instructions
# in the readme for examples:
# https://github.com/huggingface/accelerate/tree/main/examples
#
########################################################################


# Function to get the label from the filename
def extract_label(fname):
    stem = fname.split(os.path.sep)[-1]
    return re.search(r"^(.*)_\d+\.jpg$", stem).groups()[0]


class PetsDataset(Dataset):
    def __init__(self, file_names, image_transform=None, label_to_id=None):
        self.file_names = file_names
        self.image_transform = image_transform
        self.label_to_id = label_to_id

    def __len__(self):
        return len(self.file_names)

    def __getitem__(self, idx):
        fname = self.file_names[idx]
        raw_image = PIL.Image.open(fname)
        image = raw_image.convert("RGB")
        if self.image_transform is not None:
            image = self.image_transform(image)
        label = extract_label(fname)
        if self.label_to_id is not None:
            label = self.label_to_id[label]
        return {"image": image, "label": label}


def training_function(config, args):
    # Initialize accelerator
    accelerator = Accelerator(cpu=args.cpu, mixed_precision=args.mixed_precision)

    # Sample hyper-parameters for learning rate, batch size, seed and a few other HPs
    lr = config["lr"]
    num_epochs = int(config["num_epochs"])
    seed = int(config["seed"])
    batch_size = int(config["batch_size"])
    image_size = config["image_size"]
    if not isinstance(image_size, (list, tuple)):
        image_size = (image_size, image_size)

    # Grab all the image filenames
    file_names = [os.path.join(args.data_dir, fname) for fname in os.listdir(args.data_dir) if fname.end

    # Build the label correspondences
    all_labels = [extract_label(fname) for fname in file_names]
    id_to_label = list(set(all_labels))
    id_to_label.sort()
    label_to_id = {lbl: i for i, lbl in enumerate(id_to_label)}

    # Set the seed before splitting the data.
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)

    # Split our filenames between train and validation
    random_perm = np.random.permutation(len(file_names))
    cut = int(0.8 * len(file_names))
    train_split = random_perm[:cut]
    eval_split = random_perm[cut:]

    # For training we use a simple RandomResizedCrop
    train_tfm = Compose([RandomResizedCrop(image_size, scale=(0.5, 1.0)), ToTensor()])
    train_dataset = PetsDataset(
        [file_names[i] for i in train_split], image_transform=train_tfm, label_to_id=label_to_id
    )

    # For evaluation, we use a deterministic Resize
    eval_tfm = Compose([Resize(image_size), ToTensor()])
    eval_dataset = PetsDataset([file_names[i] for i in eval_split], image_transform=eval_tfm, label_to_i

    # Instantiate dataloaders.
    train_dataloader = DataLoader(train_dataset, shuffle=True, batch_size=batch_size, num_workers=4)
    eval_dataloader = DataLoader(eval_dataset, shuffle=False, batch_size=batch_size, num_workers=4)

    # Instantiate the model (we build the model here so that the seed also control new weights initializ
    model = create_model("resnet50d", pretrained=True, num_classes=len(label_to_id))

    # We could avoid this line since the accelerator is set with `device_placement=True` (default value)
    # Note that if you are placing tensors on devices manually, this line absolutely needs to be before
    # creation otherwise training will not work on TPU (`accelerate` will kindly throw an error to make
    model = model.to(accelerator.device)

    # Freezing the base model
    for param in model.parameters():
        param.requires_grad = False
    for param in model.get_classifier().parameters():
        param.requires_grad = True

    # We normalize the batches of images to be a bit faster.
    mean = torch.tensor(model.default_cfg["mean"])[None, :, None, None].to(accelerator.device)
    std = torch.tensor(model.default_cfg["std"])[None, :, None, None].to(accelerator.device)

    # Instantiate optimizer
    optimizer = torch.optim.Adam(params=model.parameters(), lr=lr / 25)
    # Instantiate learning rate scheduler
    lr_scheduler = OneCycleLR(optimizer=optimizer, max_lr=lr, epochs=num_epochs, steps_per_epoch=len(tra

    # Prepare everything
    # There is no specific order to remember, we just need to unpack the objects in the same order we ga
    # prepare method.
    model, optimizer, train_dataloader, eval_dataloader, lr_scheduler = accelerator.prepare(
        model, optimizer, train_dataloader, eval_dataloader, lr_scheduler
    )

    # Now we train the model
    for epoch in range(num_epochs):
        model.train()
        for step, batch in enumerate(train_dataloader):
            # We could avoid this line since we set the accelerator with `device_placement=True`.
            batch = {k: v.to(accelerator.device) for k, v in batch.items()}
            inputs = (batch["image"] - mean) / std
            outputs = model(inputs)
            loss = torch.nn.functional.cross_entropy(outputs, batch["label"])
            accelerator.backward(loss)
            optimizer.step()
            lr_scheduler.step()
            optimizer.zero_grad()

        model.eval()
        accurate = 0
        num_elems = 0
        for _, batch in enumerate(eval_dataloader):
            # We could avoid this line since we set the accelerator with `device_placement=True`.
            batch = {k: v.to(accelerator.device) for k, v in batch.items()}
            inputs = (batch["image"] - mean) / std
            with torch.no_grad():
                outputs = model(inputs)
            predictions = outputs.argmax(dim=-1)
            predictions, references = accelerator.gather_for_metrics((predictions, batch["label"]))
            accurate_preds = predictions == references
            num_elems += accurate_preds.shape[0]
            accurate += accurate_preds.long().sum()

        eval_metric = accurate.item() / num_elems
        # Use accelerator.print to print only on the main process.
        accelerator.print(f"epoch {epoch}: {100 * eval_metric:.2f}")


def main():
    parser = argparse.ArgumentParser(description="Simple example of training script.")
    parser.add_argument("--data_dir", required=True, help="The data folder on disk.")
    parser.add_argument(
        "--mixed_precision",
        type=str,
        default=None,
        choices=["no", "fp16", "bf16", "fp8"],
        help="Whether to use mixed precision. Choose"
        "between fp16 and bf16 (bfloat16). Bf16 requires PyTorch >= 1.10."
        "and an Nvidia Ampere GPU.",
    )
    parser.add_argument(
        "--checkpointing_steps",
        type=str,
        default=None,
        help="Whether the various states should be saved at the end of every n steps, or 'epoch' for eac
    )
    parser.add_argument("--cpu", action="store_true", help="If passed, will train on the CPU.")
    args = parser.parse_args()
    config = {"lr": 3e-2, "num_epochs": 3, "seed": 42, "batch_size": 64, "image_size": 224}
    training_function(config, args)


if __name__ == "__main__":
    main()
```

accelerate-main/examples/nlp_example.py

```
# coding=utf-8
# Copyright 2021 The HuggingFace Inc. team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
import argparse

import evaluate
import torch
from datasets import load_dataset
from torch.optim import AdamW
from torch.utils.data import DataLoader
from transformers import AutoModelForSequenceClassification, AutoTokenizer, get_linear_schedule_with_war

from accelerate import Accelerator, DistributedType


########################################################################
# This is a fully working simple example to use Accelerate
#
# This example trains a Bert base model on GLUE MRPC
# in any of the following settings (with the same script):
#   - single CPU or single GPU
#   - multi GPUS (using PyTorch distributed mode)
#   - (multi) TPUs
#   - fp16 (mixed-precision) or fp32 (normal precision)
#
# To run it in each of these various modes, follow the instructions
# in the readme for examples:
# https://github.com/huggingface/accelerate/tree/main/examples
#
########################################################################


MAX_GPU_BATCH_SIZE = 16
EVAL_BATCH_SIZE = 32


def get_dataloaders(accelerator: Accelerator, batch_size: int = 16):
    """
    Creates a set of `DataLoader`s for the `glue` dataset,
    using "bert-base-cased" as the tokenizer.

    Args:
        accelerator (`Accelerator`):
            An `Accelerator` object
        batch_size (`int`, *optional*):
            The batch size for the train and validation DataLoaders.
    """
    tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
    datasets = load_dataset("glue", "mrpc")

    def tokenize_function(examples):
        # max_length=None => use the model max length (it's actually the default)
        outputs = tokenizer(examples["sentence1"], examples["sentence2"], truncation=True, max_length=No
        return outputs

    # Apply the method we just defined to all the examples in all the splits of the dataset
    # starting with the main process first:
```

```
    with accelerator.main_process_first():
        tokenized_datasets = datasets.map(
            tokenize_function,
            batched=True,
            remove_columns=["idx", "sentence1", "sentence2"],
        )

    # We also rename the 'label' column to 'labels' which is the expected name for labels by the models
    # transformers library
    tokenized_datasets = tokenized_datasets.rename_column("label", "labels")

    def collate_fn(examples):
        # On TPU it's best to pad everything to the same length or training will be very slow.
        max_length = 128 if accelerator.distributed_type == DistributedType.TPU else None
        # When using mixed precision we want round multiples of 8/16
        if accelerator.mixed_precision == "fp8":
            pad_to_multiple_of = 16
        elif accelerator.mixed_precision != "no":
            pad_to_multiple_of = 8
        else:
            pad_to_multiple_of = None

        return tokenizer.pad(
            examples,
            padding="longest",
            max_length=max_length,
            pad_to_multiple_of=pad_to_multiple_of,
            return_tensors="pt",
        )

    # Instantiate dataloaders.
    train_dataloader = DataLoader(
        tokenized_datasets["train"], shuffle=True, collate_fn=collate_fn, batch_size=batch_size, drop_la
    )
    eval_dataloader = DataLoader(
        tokenized_datasets["validation"],
        shuffle=False,
        collate_fn=collate_fn,
        batch_size=EVAL_BATCH_SIZE,
        drop_last=(accelerator.mixed_precision == "fp8"),
    )

    return train_dataloader, eval_dataloader


def training_function(config, args):
    # Initialize accelerator
    accelerator = Accelerator(cpu=args.cpu, mixed_precision=args.mixed_precision)
    # Sample hyper-parameters for learning rate, batch size, seed and a few other HPs
    lr = config["lr"]
    num_epochs = int(config["num_epochs"])
    seed = int(config["seed"])
    batch_size = int(config["batch_size"])

    metric = evaluate.load("glue", "mrpc")

    # If the batch size is too big we use gradient accumulation
    gradient_accumulation_steps = 1
    if batch_size > MAX_GPU_BATCH_SIZE and accelerator.distributed_type != DistributedType.TPU:
        gradient_accumulation_steps = batch_size // MAX_GPU_BATCH_SIZE
        batch_size = MAX_GPU_BATCH_SIZE

    set_seed(seed)
    train_dataloader, eval_dataloader = get_dataloaders(accelerator, batch_size)
    # Instantiate the model (we build the model here so that the seed also control new weights initializ
    model = AutoModelForSequenceClassification.from_pretrained("bert-base-cased", return_dict=True)

    # We could avoid this line since the accelerator is set with `device_placement=True` (default value)
    # Note that if you are placing tensors on devices manually, this line absolutely needs to be before
    # creation otherwise training will not work on TPU (`accelerate` will kindly throw an error to make
    model = model.to(accelerator.device)
    # Instantiate optimizer
    optimizer = AdamW(params=model.parameters(), lr=lr)

    # Instantiate scheduler
    lr_scheduler = get_linear_schedule_with_warmup(
        optimizer=optimizer,
        num_warmup_steps=100,
        num_training_steps=(len(train_dataloader) * num_epochs) // gradient_accumulation_steps,
    )

    # Prepare everything
    # There is no specific order to remember, we just need to unpack the objects in the same order we ga
    # prepare method.

    model, optimizer, train_dataloader, eval_dataloader, lr_scheduler = accelerator.prepare(
        model, optimizer, train_dataloader, eval_dataloader, lr_scheduler
    )

    # Now we train the model
    for epoch in range(num_epochs):
        model.train()
        for step, batch in enumerate(train_dataloader):
            # We could avoid this line since we set the accelerator with `device_placement=True`.
            batch.to(accelerator.device)
            outputs = model(**batch)
            loss = outputs.loss
            loss = loss / gradient_accumulation_steps
            accelerator.backward(loss)
            if step % gradient_accumulation_steps == 0:
                optimizer.step()
                lr_scheduler.step()
                optimizer.zero_grad()

        model.eval()
        for step, batch in enumerate(eval_dataloader):
            # We could avoid this line since we set the accelerator with `device_placement=True`.
            batch.to(accelerator.device)
            with torch.no_grad():
                outputs = model(**batch)
            predictions = outputs.logits.argmax(dim=-1)
            predictions, references = accelerator.gather_for_metrics((predictions, batch["labels"]))
            metric.add_batch(
                predictions=predictions,
                references=references,
            )

        eval_metric = metric.compute()
        # Use accelerator.print to print only on the main process.
        accelerator.print(f"epoch {epoch}:", eval_metric)


def main():
    parser = argparse.ArgumentParser(description="Simple example of training script.")
    parser.add_argument(
        "--mixed_precision",
        type=str,
        default=None,
        choices=["no", "fp16", "bf16", "fp8"],
        help="Whether to use mixed precision. Choose"
        "between fp16 and bf16 (bfloat16). Bf16 requires PyTorch >= 1.10."
        "and an Nvidia Ampere GPU.",
    )
    parser.add_argument("--cpu", action="store_true", help="If passed, will train on the CPU.")
    args = parser.parse_args()
    config = {"lr": 2e-5, "num_epochs": 3, "seed": 42, "batch_size": 16}
    training_function(config, args)


if __name__ == "__main__":
    main()
```

accelerate-main/examples/complete_cv_example.py

```
# coding=utf-8
# Copyright 2021 The HuggingFace Inc. team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
import argparse
import os
import re

import numpy as np
import PIL
import torch
from timm import create_model
from torch.optim.lr_scheduler import OneCycleLR
from torch.utils.data import DataLoader, Dataset
from torchvision.transforms import Compose, RandomResizedCrop, Resize, ToTensor

from accelerate import Accelerator


########################################################################
# This is a fully working simple example to use Accelerate
#
# This example trains a ResNet50 on the Oxford-IIT Pet Dataset
# in any of the following settings (with the same script):
#   - single CPU or single GPU
#   - multi GPUS (using PyTorch distributed mode)
#   - (multi) TPUs
#   - fp16 (mixed-precision) or fp32 (normal precision)
#
# To run it in each of these various modes, follow the instructions
# in the readme for examples:
# https://github.com/huggingface/accelerate/tree/main/examples
#
########################################################################


# Function to get the label from the filename
def extract_label(fname):
    stem = fname.split(os.path.sep)[-1]
    return re.search(r"^(.*)_\d+\.jpg$", stem).groups()[0]


class PetsDataset(Dataset):
    def __init__(self, file_names, image_transform=None, label_to_id=None):
        self.file_names = file_names
        self.image_transform = image_transform
        self.label_to_id = label_to_id

    def __len__(self):
        return len(self.file_names)

    def __getitem__(self, idx):
        fname = self.file_names[idx]
        raw_image = PIL.Image.open(fname)
        image = raw_image.convert("RGB")
        if self.image_transform is not None:
            image = self.image_transform(image)
        label = extract_label(fname)
        if self.label_to_id is not None:
            label = self.label_to_id[label]
        return {"image": image, "label": label}


def training_function(config, args):
    # Initialize accelerator
    if args.with_tracking:
        accelerator = Accelerator(
            cpu=args.cpu, mixed_precision=args.mixed_precision, log_with="all", logging_dir=args.logging
        )
    else:
        accelerator = Accelerator(cpu=args.cpu, mixed_precision=args.mixed_precision)

    # Sample hyper-parameters for learning rate, batch size, seed and a few other HPs
    lr = config["lr"]
    num_epochs = int(config["num_epochs"])
    seed = int(config["seed"])
    batch_size = int(config["batch_size"])
    image_size = config["image_size"]
    if not isinstance(image_size, (list, tuple)):
        image_size = (image_size, image_size)

    # Parse out whether we are saving every epoch or after a certain number of batches
    if hasattr(args.checkpointing_steps, "isdigit"):
        if args.checkpointing_steps == "epoch":
            checkpointing_steps = args.checkpointing_steps
        elif args.checkpointing_steps.isdigit():
            checkpointing_steps = int(args.checkpointing_steps)
        else:
            raise ValueError(
                f"Argument `checkpointing_steps` must be either a number or `epoch`. `{args.checkpointin
            )
    else:
        checkpointing_steps = None

    # We need to initialize the trackers we use, and also store our configuration
    if args.with_tracking:
        run = os.path.split(__file__)[-1].split(".")[0]
        accelerator.init_trackers(run, config)

    # Grab all the image filenames
    file_names = [os.path.join(args.data_dir, fname) for fname in os.listdir(args.data_dir) if fname.end

    # Build the label correspondences
    all_labels = [extract_label(fname) for fname in file_names]
    id_to_label = list(set(all_labels))
    id_to_label.sort()
    label_to_id = {lbl: i for i, lbl in enumerate(id_to_label)}

    # Set the seed before splitting the data.
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)

    # Split our filenames between train and validation
    random_perm = np.random.permutation(len(file_names))
    cut = int(0.8 * len(file_names))
    train_split = random_perm[:cut]
    eval_split = random_perm[cut:]

    # For training we use a simple RandomResizedCrop
    train_tfm = Compose([RandomResizedCrop(image_size, scale=(0.5, 1.0)), ToTensor()])
    train_dataset = PetsDataset(
        [file_names[i] for i in train_split], image_transform=train_tfm, label_to_id=label_to_id
    )

    # For evaluation, we use a deterministic Resize
    eval_tfm = Compose([Resize(image_size), ToTensor()])
    eval_dataset = PetsDataset([file_names[i] for i in eval_split], image_transform=eval_tfm, label_to_i

    # Instantiate dataloaders.
```

```
train_dataloader = DataLoader(train_dataset, shuffle=True, batch_size=batch_size, num_workers=4)
eval_dataloader = DataLoader(eval_dataset, shuffle=False, batch_size=batch_size, num_workers=4)

# Instantiate the model (we build the model here so that the seed also control new weights initializ
model = create_model("resnet50d", pretrained=True, num_classes=len(label_to_id))

# We could avoid this line since the accelerator is set with `device_placement=True` (default value)
# Note that if you are placing tensors on devices manually, this line absolutely needs to be before
# creation otherwise training will not work on TPU (`accelerate` will kindly throw an error to make
model = model.to(accelerator.device)

# Freezing the base model
for param in model.parameters():
    param.requires_grad = False
for param in model.get_classifier().parameters():
    param.requires_grad = True

# We normalize the batches of images to be a bit faster.
mean = torch.tensor(model.default_cfg["mean"])[None, :, None, None].to(accelerator.device)
std = torch.tensor(model.default_cfg["std"])[None, :, None, None].to(accelerator.device)

# Instantiate optimizer
optimizer = torch.optim.Adam(params=model.parameters(), lr=lr / 25)

# Instantiate learning rate scheduler
lr_scheduler = OneCycleLR(optimizer=optimizer, max_lr=lr, epochs=num_epochs, steps_per_epoch=len(tra

# Prepare everything
# There is no specific order to remember, we just need to unpack the objects in the same order we ga
# prepare method.
model, optimizer, train_dataloader, eval_dataloader, lr_scheduler = accelerator.prepare(
    model, optimizer, train_dataloader, eval_dataloader, lr_scheduler
)
# We need to keep track of how many total steps we have iterated over
overall_step = 0
# We also need to keep track of the starting epoch so files are named properly
starting_epoch = 0

# Potentially load in the weights and states from a previous save
if args.resume_from_checkpoint:
    if args.resume_from_checkpoint is not None or args.resume_from_checkpoint != "":
        accelerator.print(f"Resumed from checkpoint: {args.resume_from_checkpoint}")
        accelerator.load_state(args.resume_from_checkpoint)
        path = os.path.basename(args.resume_from_checkpoint)
    else:
        # Get the most recent checkpoint
        dirs = [f.name for f in os.scandir(os.getcwd()) if f.is_dir()]
        dirs.sort(key=os.path.getctime)
        path = dirs[-1]  # Sorts folders by date modified, most recent checkpoint is the last
    # Extract `epoch_{i}` or `step_{i}`
    training_difference = os.path.splitext(path)[0]

    if "epoch" in training_difference:
        starting_epoch = int(training_difference.replace("epoch_", "")) + 1
        resume_step = None
    else:
        resume_step = int(training_difference.replace("step_", ""))
        starting_epoch = resume_step // len(train_dataloader)
        resume_step -= starting_epoch * len(train_dataloader)

# Now we train the model
for epoch in range(starting_epoch, num_epochs):
    model.train()
    if args.with_tracking:
        total_loss = 0
    if args.resume_from_checkpoint and epoch == starting_epoch and resume_step is not None:
        # We need to skip steps until we reach the resumed step
        train_dataloader = accelerator.skip_first_batches(train_dataloader, resume_step)
        overall_step += resume_step
    for batch in train_dataloader:
        # We could avoid this line since we set the accelerator with `device_placement=True`.
        batch = {k: v.to(accelerator.device) for k, v in batch.items()}
```

```
            inputs = (batch["image"] - mean) / std
            outputs = model(inputs)
            loss = torch.nn.functional.cross_entropy(outputs, batch["label"])
            # We keep track of the loss at each epoch
            if args.with_tracking:
                total_loss += loss.detach().float()
            accelerator.backward(loss)
            optimizer.step()
            lr_scheduler.step()
            optimizer.zero_grad()
            overall_step += 1
            if isinstance(checkpointing_steps, int):
                output_dir = f"step_{overall_step}"
                if overall_step % checkpointing_steps == 0:
                    if args.output_dir is not None:
                        output_dir = os.path.join(args.output_dir, output_dir)
                    accelerator.save_state(output_dir)
        model.eval()
        accurate = 0
        num_elems = 0
        for step, batch in enumerate(eval_dataloader):
            # We could avoid this line since we set the accelerator with `device_placement=True`.
            batch = {k: v.to(accelerator.device) for k, v in batch.items()}
            inputs = (batch["image"] - mean) / std
            with torch.no_grad():
                outputs = model(inputs)
            predictions = outputs.argmax(dim=-1)
            predictions, references = accelerator.gather_for_metrics((predictions, batch["label"]))
            accurate_preds = predictions == references
            num_elems += accurate_preds.shape[0]
            accurate += accurate_preds.long().sum()

        eval_metric = accurate.item() / num_elems
        # Use accelerator.print to print only on the main process.
        accelerator.print(f"epoch {epoch}: {100 * eval_metric:.2f}")
        if args.with_tracking:
            accelerator.log(
                {
                    "accuracy": 100 * eval_metric,
                    "train_loss": total_loss.item() / len(train_dataloader),
                    "epoch": epoch,
                },
                step=overall_step,
            )
        if checkpointing_steps == "epoch":
            output_dir = f"epoch_{epoch}"
            if args.output_dir is not None:
                output_dir = os.path.join(args.output_dir, output_dir)
            accelerator.save_state(output_dir)

    if args.with_tracking:
        accelerator.end_training()


def main():
    parser = argparse.ArgumentParser(description="Simple example of training script.")
    parser.add_argument("--data_dir", required=True, help="The data folder on disk.")
    parser.add_argument("--fp16", action="store_true", help="If passed, will use FP16 training.")
    parser.add_argument(
        "--mixed_precision",
        type=str,
        default=None,
        choices=["no", "fp16", "bf16", "fp8"],
        help="Whether to use mixed precision. Choose"
        "between fp16 and bf16 (bfloat16). Bf16 requires PyTorch >= 1.10."
        "and an Nvidia Ampere GPU.",
    )
    parser.add_argument("--cpu", action="store_true", help="If passed, will train on the CPU.")
    parser.add_argument(
        "--checkpointing_steps",
        type=str,
        default=None,
        help="Whether the various states should be saved at the end of every n steps, or 'epoch' for eac
    )
    parser.add_argument(
        "--output_dir",
        type=str,
        default=".",
        help="Optional save directory where all checkpoint folders will be stored. Default is the curren
    )
    parser.add_argument(
        "--resume_from_checkpoint",
        type=str,
        default=None,
        help="If the training should continue from a checkpoint folder.",
    )
    parser.add_argument(
        "--with_tracking",
        action="store_true",
        help="Whether to load in all available experiment trackers from the environment and use them for
    )
    parser.add_argument(
        "--logging_dir",
        type=str,
        default="logs",
        help="Location on where to store experiment tracking logs`",
    )
    args = parser.parse_args()
    config = {"lr": 3e-2, "num_epochs": 3, "seed": 42, "batch_size": 64, "image_size": 224}
    training_function(config, args)


if __name__ == "__main__":
    main()
```

accelerate-main/examples/complete_nlp_example.py

```
# coding=utf-8
# Copyright 2021 The HuggingFace Inc. team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
import argparse
import os

import evaluate
import torch
from datasets import load_dataset
from torch.optim import AdamW
from torch.utils.data import DataLoader
from transformers import AutoModelForSequenceClassification, AutoTokenizer, get_linear_schedule_with_war

from accelerate import Accelerator, DistributedType


########################################################################
# This is a fully working simple example to use Accelerate
#
# This example trains a Bert base model on GLUE MRPC
# in any of the following settings (with the same script):
#   - single CPU or single GPU
#   - multi GPUS (using PyTorch distributed mode)
#   - (multi) TPUs
#   - fp16 (mixed-precision) or fp32 (normal precision)
#
# This example also demonstrates the checkpointing and sharding capabilities
#
# To run it in each of these various modes, follow the instructions
# in the readme for examples:
# https://github.com/huggingface/accelerate/tree/main/examples
#
########################################################################


MAX_GPU_BATCH_SIZE = 16
EVAL_BATCH_SIZE = 32


def training_function(config, args):
    # Initialize accelerator
    if args.with_tracking:
        accelerator = Accelerator(
            cpu=args.cpu, mixed_precision=args.mixed_precision, log_with="all", logging_dir=args.logging
        )
    else:
        accelerator = Accelerator(cpu=args.cpu, mixed_precision=args.mixed_precision)

    if hasattr(args.checkpointing_steps, "isdigit"):
        if args.checkpointing_steps == "epoch":
            checkpointing_steps = args.checkpointing_steps
        elif args.checkpointing_steps.isdigit():
            checkpointing_steps = int(args.checkpointing_steps)
        else:
            raise ValueError(
                f"Argument `checkpointing_steps` must be either a number or `epoch`. `{args.checkpointin
            )
```

```
else:
    checkpointing_steps = None
# Sample hyper-parameters for learning rate, batch size, seed and a few other HPs
lr = config["lr"]
num_epochs = int(config["num_epochs"])
seed = int(config["seed"])
batch_size = int(config["batch_size"])

# We need to initialize the trackers we use, and also store our configuration
if args.with_tracking:
    run = os.path.split(__file__)[-1].split(".")[0]
    accelerator.init_trackers(run, config)

tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
datasets = load_dataset("glue", "mrpc")
metric = evaluate.load("glue", "mrpc")

def tokenize_function(examples):
    # max_length=None => use the model max length (it's actually the default)
    outputs = tokenizer(examples["sentence1"], examples["sentence2"], truncation=True, max_length=No
    return outputs

# Apply the method we just defined to all the examples in all the splits of the dataset
# starting with the main process first:
with accelerator.main_process_first():
    tokenized_datasets = datasets.map(
        tokenize_function,
        batched=True,
        remove_columns=["idx", "sentence1", "sentence2"],
    )

# We also rename the 'label' column to 'labels' which is the expected name for labels by the models
# transformers library
tokenized_datasets = tokenized_datasets.rename_column("label", "labels")

# If the batch size is too big we use gradient accumulation
gradient_accumulation_steps = 1
if batch_size > MAX_GPU_BATCH_SIZE and accelerator.distributed_type != DistributedType.TPU:
    gradient_accumulation_steps = batch_size // MAX_GPU_BATCH_SIZE
    batch_size = MAX_GPU_BATCH_SIZE

def collate_fn(examples):
    # On TPU it's best to pad everything to the same length or training will be very slow.
    max_length = 128 if accelerator.distributed_type == DistributedType.TPU else None
    # When using mixed precision we want round multiples of 8/16
    if accelerator.mixed_precision == "fp8":
        pad_to_multiple_of = 16
    elif accelerator.mixed_precision != "no":
        pad_to_multiple_of = 8
    else:
        pad_to_multiple_of = None

    return tokenizer.pad(
        examples,
        padding="longest",
        max_length=max_length,
        pad_to_multiple_of=pad_to_multiple_of,
        return_tensors="pt",
    )

# Instantiate dataloaders.
train_dataloader = DataLoader(
    tokenized_datasets["train"], shuffle=True, collate_fn=collate_fn, batch_size=batch_size
)
eval_dataloader = DataLoader(
    tokenized_datasets["validation"], shuffle=False, collate_fn=collate_fn, batch_size=EVAL_BATCH_SI
)

set_seed(seed)

# Instantiate the model (we build the model here so that the seed also control new weights initializ
model = AutoModelForSequenceClassification.from_pretrained("bert-base-cased", return_dict=True)
```

```
    # We could avoid this line since the accelerator is set with `device_placement=True` (default value)
    # Note that if you are placing tensors on devices manually, this line absolutely needs to be before
    # creation otherwise training will not work on TPU (`accelerate` will kindly throw an error to make
    model = model.to(accelerator.device)

    # Instantiate optimizer
    optimizer = AdamW(params=model.parameters(), lr=lr)

    # Instantiate scheduler
    lr_scheduler = get_linear_schedule_with_warmup(
        optimizer=optimizer,
        num_warmup_steps=100,
        num_training_steps=(len(train_dataloader) * num_epochs) // gradient_accumulation_steps,
    )

    # Prepare everything
    # There is no specific order to remember, we just need to unpack the objects in the same order we ga
    # prepare method.
    model, optimizer, train_dataloader, eval_dataloader, lr_scheduler = accelerator.prepare(
        model, optimizer, train_dataloader, eval_dataloader, lr_scheduler
    )

    # We need to keep track of how many total steps we have iterated over
    overall_step = 0
    # We also need to keep track of the stating epoch so files are named properly
    starting_epoch = 0

    # Potentially load in the weights and states from a previous save
    if args.resume_from_checkpoint:
        if args.resume_from_checkpoint is not None or args.resume_from_checkpoint != "":
            accelerator.print(f"Resumed from checkpoint: {args.resume_from_checkpoint}")
            accelerator.load_state(args.resume_from_checkpoint)
            path = os.path.basename(args.resume_from_checkpoint)
        else:
            # Get the most recent checkpoint
            dirs = [f.name for f in os.scandir(os.getcwd()) if f.is_dir()]
            dirs.sort(key=os.path.getctime)
            path = dirs[-1]  # Sorts folders by date modified, most recent checkpoint is the last
        # Extract `epoch_{i}` or `step_{i}`
        training_difference = os.path.splitext(path)[0]

        if "epoch" in training_difference:
            starting_epoch = int(training_difference.replace("epoch_", "")) + 1
            resume_step = None
        else:
            resume_step = int(training_difference.replace("step_", ""))
            starting_epoch = resume_step // len(train_dataloader)
            resume_step -= starting_epoch * len(train_dataloader)

    # Now we train the model
    for epoch in range(starting_epoch, num_epochs):
        model.train()
        if args.with_tracking:
            total_loss = 0
        if args.resume_from_checkpoint and epoch == starting_epoch and resume_step is not None:
            # We need to skip steps until we reach the resumed step
            train_dataloader = accelerator.skip_first_batches(train_dataloader, resume_step)
            overall_step += resume_step
        for step, batch in enumerate(train_dataloader):
            # We could avoid this line since we set the accelerator with `device_placement=True`.
            batch.to(accelerator.device)
            outputs = model(**batch)
            loss = outputs.loss
            loss = loss / gradient_accumulation_steps
            # We keep track of the loss at each epoch
            if args.with_tracking:
                total_loss += loss.detach().float()
            accelerator.backward(loss)
            if step % gradient_accumulation_steps == 0:
                optimizer.step()
                lr_scheduler.step()
                optimizer.zero_grad()
            overall_step += 1

            if isinstance(checkpointing_steps, int):
                output_dir = f"step_{overall_step}"
                if overall_step % checkpointing_steps == 0:
                    if args.output_dir is not None:
                        output_dir = os.path.join(args.output_dir, output_dir)
                    accelerator.save_state(output_dir)

        model.eval()
        for step, batch in enumerate(eval_dataloader):
            # We could avoid this line since we set the accelerator with `device_placement=True`.
            batch.to(accelerator.device)
            with torch.no_grad():
                outputs = model(**batch)
            predictions = outputs.logits.argmax(dim=-1)
            predictions, references = accelerator.gather_for_metrics((predictions, batch["labels"]))
            metric.add_batch(
                predictions=predictions,
                references=references,
            )

        eval_metric = metric.compute()
        # Use accelerator.print to print only on the main process.
        accelerator.print(f"epoch {epoch}:", eval_metric)
        if args.with_tracking:
            accelerator.log(
                {
                    "accuracy": eval_metric["accuracy"],
                    "f1": eval_metric["f1"],
                    "train_loss": total_loss.item() / len(train_dataloader),
                    "epoch": epoch,
                },
                step=epoch,
            )

        if checkpointing_steps == "epoch":
            output_dir = f"epoch_{epoch}"
            if args.output_dir is not None:
                output_dir = os.path.join(args.output_dir, output_dir)
            accelerator.save_state(output_dir)

    if args.with_tracking:
        accelerator.end_training()


def main():
    parser = argparse.ArgumentParser(description="Simple example of training script.")
    parser.add_argument(
        "--mixed_precision",
        type=str,
        default=None,
        choices=["no", "fp16", "bf16", "fp8"],
        help="Whether to use mixed precision. Choose"
        "between fp16 and bf16 (bfloat16). Bf16 requires PyTorch >= 1.10."
        "and an Nvidia Ampere GPU.",
    )
    parser.add_argument("--cpu", action="store_true", help="If passed, will train on the CPU.")
    parser.add_argument(
        "--checkpointing_steps",
        type=str,
        default=None,
        help="Whether the various states should be saved at the end of every n steps, or 'epoch' for eac
    )
    parser.add_argument(
        "--resume_from_checkpoint",
        type=str,
        default=None,
        help="If the training should continue from a checkpoint folder.",
    )
    parser.add_argument(
        "--with_tracking",
        action="store_true",
        help="Whether to load in all available experiment trackers from the environment and use them for
    )
    parser.add_argument(
        "--output_dir",
        type=str,
        default=".",
        help="Optional save directory where all checkpoint folders will be stored. Default is the curren
    )
    parser.add_argument(
        "--logging_dir",
        type=str,
        default="logs",
        help="Location on where to store experiment tracking logs`",
    )
    args = parser.parse_args()
    config = {"lr": 2e-5, "num_epochs": 3, "seed": 42, "batch_size": 16}
    training_function(config, args)


if __name__ == "__main__":
    main()
```

accelerate-main/examples/by_feature/README.md

~~~markdown
# What are these scripts?

All scripts in this folder originate from the `nlp_example.py` file, as it is a very simplistic NLP trai

From there, each further script adds in just **one** feature of Accelerate, showing how you can quickly

A full example with all of these parts integrated together can be found in the `complete_nlp_example.py`

Adjustments to each script from the base `nlp_example.py` file can be found quickly by searching for "#

## Example Scripts by Feature and their Arguments

### Base Example (`../nlp_example.py`)

- Shows how to use `Accelerator` in an extremely simplistic PyTorch training loop
- Arguments available:
  - `mixed_precision`, whether to use mixed precision. ("no", "fp16", or "bf16")
  - `cpu`, whether to train using only the CPU. (yes/no/1/0)

All following scripts also accept these arguments in addition to their added ones.

These arguments should be added at the end of any method for starting the python script (such as `python

```bash
accelerate launch ../nlp_example.py --mixed_precision fp16 --cpu 0
```

### Checkpointing and Resuming Training (`checkpointing.py`)

- Shows how to use `Accelerator.save_state` and `Accelerator.load_state` to save or continue training
- **It is assumed you are continuing off the same training script**
- Arguments available:
  - `checkpointing_steps`, after how many steps the various states should be saved. ("epoch", 1, 2, ...)
  - `output_dir`, where saved state folders should be saved to, default is current working directory
  - `resume_from_checkpoint`, what checkpoint folder to resume from. ("epoch_0", "step_22", ...)

These arguments should be added at the end of any method for starting the python script (such as `python

(Note, `resume_from_checkpoint` assumes that we've ran the script for one epoch with the `--checkpointin

```bash
accelerate launch ./checkpointing.py --checkpointing_steps epoch output_dir "checkpointing_tutorial" --r
```

### Cross Validation (`cross_validation.py`)

- Shows how to use `Accelerator.free_memory` and run cross validation efficiently with `datasets`.
- Arguments available:
  - `num_folds`, the number of folds the training dataset should be split into.

These arguments should be added at the end of any method for starting the python script (such as `python

```bash
accelerate launch ./cross_validation.py --num_folds 2
```

### Experiment Tracking (`tracking.py`)

- Shows how to use `Accelerate.init_trackers` and `Accelerator.log`
- Can be used with Weights and Biases, TensorBoard, or CometML.
- Arguments available:
  - `with_tracking`, whether to load in all available experiment trackers from the environment.

These arguments should be added at the end of any method for starting the python script (such as `python

```bash
accelerate launch ./tracking.py --with_tracking
```
~~~

### Gradient Accumulation (`gradient_accumulation.py`)

- Shows how to use `Accelerator.no_sync` to prevent gradient averaging in a distributed setup.
- Arguments available:
  - `gradient_accumulation_steps`, the number of steps to perform before the gradients are accumulated a

These arguments should be added at the end of any method for starting the python script (such as `python

```bash
accelerate launch ./gradient_accumulation.py --gradient_accumulation_steps 5
```

accelerate-main/examples/deepspeed_config_templates/zero_stage2_offload_config.json

```
{
    "fp16": {
        "enabled": true,
        "loss_scale": 0,
        "loss_scale_window": 1000,
        "initial_scale_power": 16,
        "hysteresis": 2,
        "min_loss_scale": 1
    },
    "optimizer": {
        "type": "AdamW",
        "params": {
            "lr": "auto",
            "weight_decay": "auto",
            "torch_adam": true,
            "adam_w_mode": true
        }
    },
    "scheduler": {
        "type": "WarmupDecayLR",
        "params": {
            "warmup_min_lr": "auto",
            "warmup_max_lr": "auto",
            "warmup_num_steps": "auto",
            "total_num_steps": "auto"
        }
    },
    "zero_optimization": {
        "stage": 2,
        "offload_optimizer": {
            "device": "cpu",
            "pin_memory": true
        },
        "allgather_partitions": true,
        "allgather_bucket_size": 2e8,
        "overlap_comm": true,
        "reduce_scatter": true,
        "reduce_bucket_size": "auto",
        "contiguous_gradients": true
    },
    "gradient_accumulation_steps": 1,
    "gradient_clipping": "auto",
    "steps_per_print": 2000,
    "train_batch_size": "auto",
    "train_micro_batch_size_per_gpu": "auto",
    "wall_clock_breakdown": false
}
```

accelerate-main/examples/deepspeed_config_templates/zero_stage2_config.json

```json
{
    "fp16": {
        "enabled": true,
        "loss_scale": 0,
        "loss_scale_window": 1000,
        "initial_scale_power": 16,
        "hysteresis": 2,
        "min_loss_scale": 1
    },
    "optimizer": {
        "type": "AdamW",
        "params": {
            "lr": "auto",
            "weight_decay": "auto",
            "torch_adam": true,
            "adam_w_mode": true
        }
    },
    "scheduler": {
        "type": "WarmupDecayLR",
        "params": {
            "warmup_min_lr": "auto",
            "warmup_max_lr": "auto",
            "warmup_num_steps": "auto",
            "total_num_steps": "auto"
        }
    },
    "zero_optimization": {
        "stage": 2,
        "allgather_partitions": true,
        "allgather_bucket_size": 2e8,
        "overlap_comm": true,
        "reduce_scatter": true,
        "reduce_bucket_size": "auto",
        "contiguous_gradients": true
    },
    "gradient_accumulation_steps": 1,
    "gradient_clipping": "auto",
    "steps_per_print": 2000,
    "train_batch_size": "auto",
    "train_micro_batch_size_per_gpu": "auto",
    "wall_clock_breakdown": false
}
```

accelerate-main/examples/deepspeed_config_templates/zero_stage3_config.json

```
{
    "fp16": {
        "enabled": true,
        "loss_scale": 0,
        "loss_scale_window": 1000,
        "initial_scale_power": 16,
        "hysteresis": 2,
        "min_loss_scale": 1
    },
    "optimizer": {
        "type": "AdamW",
        "params": {
            "lr": "auto",
            "weight_decay": "auto"
        }
    },
    "scheduler": {
        "type": "WarmupDecayLR",
        "params": {
            "warmup_min_lr": "auto",
            "warmup_max_lr": "auto",
            "warmup_num_steps": "auto",
            "total_num_steps": "auto"
        }
    },
    "zero_optimization": {
        "stage": 3,
        "overlap_comm": true,
        "contiguous_gradients": true,
        "reduce_bucket_size": "auto",
        "stage3_prefetch_bucket_size": "auto",
        "stage3_param_persistence_threshold": "auto",
        "sub_group_size": 1e9,
        "stage3_max_live_parameters": 1e9,
        "stage3_max_reuse_distance": 1e9,
        "stage3_gather_16bit_weights_on_model_save": "auto"
    },
    "gradient_accumulation_steps": 1,
    "gradient_clipping": "auto",
    "steps_per_print": 2000,
    "train_batch_size": "auto",
    "train_micro_batch_size_per_gpu": "auto",
    "wall_clock_breakdown": false
}
```

accelerate-main/examples/deepspeed_config_templates/zero_stage1_config.json

```
{
    "fp16": {
        "enabled": true,
        "loss_scale": 0,
        "loss_scale_window": 1000,
        "initial_scale_power": 16,
        "hysteresis": 2,
        "min_loss_scale": 1
    },
    "optimizer": {
        "type": "AdamW",
        "params": {
            "lr": "auto",
            "weight_decay": "auto",
            "torch_adam": true,
            "adam_w_mode": true
        }
    },
    "scheduler": {
        "type": "WarmupDecayLR",
        "params": {
            "warmup_min_lr": "auto",
            "warmup_max_lr": "auto",
            "warmup_num_steps": "auto",
            "total_num_steps": "auto"
        }
    },
    "zero_optimization": {
        "stage": 1,
        "allgather_partitions": true,
        "allgather_bucket_size": 2e8,
        "overlap_comm": true,
        "reduce_scatter": true,
        "reduce_bucket_size": "auto",
        "contiguous_gradients": true
    },
    "gradient_accumulation_steps": 1,
    "gradient_clipping": "auto",
    "steps_per_print": 2000,
    "train_batch_size": "auto",
    "train_micro_batch_size_per_gpu": "auto",
    "wall_clock_breakdown": false
}
```

accelerate-main/examples/deepspeed_config_templates/zero_stage3_offload_config.json

```json
{
    "fp16": {
        "enabled": true,
        "loss_scale": 0,
        "loss_scale_window": 1000,
        "initial_scale_power": 16,
        "hysteresis": 2,
        "min_loss_scale": 1
    },
    "optimizer": {
        "type": "AdamW",
        "params": {
            "lr": "auto",
            "weight_decay": "auto"
        }
    },
    "scheduler": {
        "type": "WarmupDecayLR",
        "params": {
            "warmup_min_lr": "auto",
            "warmup_max_lr": "auto",
            "warmup_num_steps": "auto",
            "total_num_steps": "auto"
        }
    },
    "zero_optimization": {
        "stage": 3,
        "offload_optimizer": {
            "device": "cpu",
            "pin_memory": true
        },
        "offload_param": {
            "device": "cpu",
            "pin_memory": true
        },
        "overlap_comm": true,
        "contiguous_gradients": true,
        "reduce_bucket_size": "auto",
        "stage3_prefetch_bucket_size": "auto",
        "stage3_param_persistence_threshold": "auto",
        "sub_group_size": 1e9,
        "stage3_max_live_parameters": 1e9,
        "stage3_max_reuse_distance": 1e9,
        "stage3_gather_16bit_weights_on_model_save": "auto"
    },
    "gradient_accumulation_steps": 1,
    "gradient_clipping": "auto",
    "steps_per_print": 2000,
    "train_batch_size": "auto",
    "train_micro_batch_size_per_gpu": "auto",
    "wall_clock_breakdown": false
}
```

accelerate-main/examples/by_feature/megatron_lm_gpt_pretraining.py

```
#!/usr/bin/env python
# coding=utf-8
# Copyright 2021 The HuggingFace Inc. team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
"""
Fine-tuning the library models for causal language modeling (GPT, GPT-2, CTRL, ...)
on a text file or a dataset without using HuggingFace Trainer.

Here is the full list of checkpoints on the hub that can be fine-tuned by this script:
https://huggingface.co/models?filter=text-generation
"""
# You can also adapt this script on your own causal language modeling task. Pointers for this are left a

import argparse
import json
import logging
import math
import os
import random
from itertools import chain
from pathlib import Path

import datasets
import torch
import transformers
from datasets import load_dataset
from huggingface_hub import Repository
from torch.utils.data import DataLoader
from tqdm.auto import tqdm
from transformers import (
    CONFIG_MAPPING,
    MODEL_MAPPING,
    AutoConfig,
    AutoModelForCausalLM,
    AutoTokenizer,
    SchedulerType,
    default_data_collator,
    get_scheduler,
)
from transformers.utils import check_min_version, get_full_repo_name, send_example_telemetry
from transformers.utils.versions import require_version

from accelerate import Accelerator, DistributedType
from accelerate.logging import get_logger
from accelerate.utils import MegatronLMDummyScheduler, set_seed


# Will error if the minimal version of Transformers is not installed. Remove at your own risks.
check_min_version("4.23.0.dev0")

logger = get_logger(__name__)

require_version("datasets>=1.8.0", "To fix: pip install -r examples/pytorch/language-modeling/requiremen

MODEL_CONFIG_CLASSES = list(MODEL_MAPPING.keys())
```

```
MODEL_TYPES = tuple(conf.model_type for conf in MODEL_CONFIG_CLASSES)


def parse_args():
    parser = argparse.ArgumentParser(description="Finetune a transformers model on a causal language mod
    parser.add_argument(
        "--dataset_name",
        type=str,
        default=None,
        help="The name of the dataset to use (via the datasets library).",
    )
    parser.add_argument(
        "--dataset_config_name",
        type=str,
        default=None,
        help="The configuration name of the dataset to use (via the datasets library).",
    )
    parser.add_argument(
        "--train_file", type=str, default=None, help="A csv or a json file containing the training data.
    )
    parser.add_argument(
        "--validation_file", type=str, default=None, help="A csv or a json file containing the validatio
    )
    parser.add_argument(
        "--validation_split_percentage",
        default=5,
        help="The percentage of the train set used as validation set in case there's no validation split
    )
    parser.add_argument(
        "--model_name_or_path",
        type=str,
        help="Path to pretrained model or model identifier from huggingface.co/models.",
        required=False,
    )
    parser.add_argument(
        "--config_name",
        type=str,
        default=None,
        help="Pretrained config name or path if not the same as model_name",
    )
    parser.add_argument(
        "--tokenizer_name",
        type=str,
        default=None,
        help="Pretrained tokenizer name or path if not the same as model_name",
    )
    parser.add_argument(
        "--use_slow_tokenizer",
        action="store_true",
        help="If passed, will use a slow tokenizer (not backed by the ■ Tokenizers library).",
    )
    parser.add_argument(
        "--per_device_train_batch_size",
        type=int,
        default=8,
        help="Batch size (per device) for the training dataloader.",
    )
    parser.add_argument(
        "--per_device_eval_batch_size",
        type=int,
        default=8,
        help="Batch size (per device) for the evaluation dataloader.",
    )
    parser.add_argument(
        "--learning_rate",
        type=float,
        default=5e-5,
        help="Initial learning rate (after the potential warmup period) to use.",
    )
    parser.add_argument("--weight_decay", type=float, default=0.0, help="Weight decay to use.")
    parser.add_argument("--num_train_epochs", type=int, default=3, help="Total number of training epochs
    parser.add_argument(
        "--max_train_steps",
        type=int,
        default=None,
        help="Total number of training steps to perform. If provided, overrides num_train_epochs.",
    )
    parser.add_argument(
        "--gradient_accumulation_steps",
        type=int,
        default=1,
        help="Number of updates steps to accumulate before performing a backward/update pass.",
    )
    parser.add_argument(
        "--lr_scheduler_type",
        type=SchedulerType,
        default="linear",
        help="The scheduler type to use.",
        choices=["linear", "cosine", "cosine_with_restarts", "polynomial", "constant", "constant_with_wa
    )
    parser.add_argument(
        "--num_warmup_steps", type=int, default=0, help="Number of steps for the warmup in the lr schedu
    )
    parser.add_argument("--output_dir", type=str, default=None, help="Where to store the final model.")
    parser.add_argument("--seed", type=int, default=None, help="A seed for reproducible training.")
    parser.add_argument(
        "--model_type",
        type=str,
        default=None,
        help="Model type to use if training from scratch.",
        choices=MODEL_TYPES,
    )
    parser.add_argument(
        "--block_size",
        type=int,
        default=None,
        help=(
            "Optional input sequence length after tokenization. The training dataset will be truncated i
            " this size for training. Default to the model max input length for single sentence inputs (
            " account special tokens)."
        ),
    )
    parser.add_argument(
        "--preprocessing_num_workers",
        type=int,
        default=None,
        help="The number of processes to use for the preprocessing.",
    )
    parser.add_argument(
        "--overwrite_cache", action="store_true", help="Overwrite the cached training and evaluation set
    )
    parser.add_argument(
        "--no_keep_linebreaks", action="store_true", help="Do not keep line breaks when using TXT files.
    )
    parser.add_argument("--push_to_hub", action="store_true", help="Whether or not to push the model to
    parser.add_argument(
        "--hub_model_id", type=str, help="The name of the repository to keep in sync with the local `out
    )
    parser.add_argument("--hub_token", type=str, help="The token to use to push to the Model Hub.")
    parser.add_argument(
        "--checkpointing_steps",
        type=str,
        default=None,
        help="Whether the various states should be saved at the end of every n steps, or 'epoch' for eac
    )
    parser.add_argument(
        "--resume_from_checkpoint",
        type=str,
        default=None,
        help="If the training should continue from a checkpoint folder.",
    )
    parser.add_argument(
        "--with_tracking",
        action="store_true",
        help="Whether to enable experiment trackers for logging.",
    )
    parser.add_argument(
        "--report_to",
        type=str,
        default="all",
        help=(
            'The integration to report the results and logs to. Supported platforms are `"tensorboard"`,
            ' `"wandb"` and `"comet_ml"`. Use `"all"` (default) to report to all integrations.'
            "Only applicable when `--with_tracking` is passed."
        ),
    )
    args = parser.parse_args()

    # Sanity checks
    if args.dataset_name is None and args.train_file is None and args.validation_file is None:
        raise ValueError("Need either a dataset name or a training/validation file.")
    else:
        if args.train_file is not None:
            extension = args.train_file.split(".")[-1]
            assert extension in ["csv", "json", "txt"], "`train_file` should be a csv, json or txt file.
        if args.validation_file is not None:
            extension = args.validation_file.split(".")[-1]
            assert extension in ["csv", "json", "txt"], "`validation_file` should be a csv, json or txt

    if args.push_to_hub:
        assert args.output_dir is not None, "Need an `output_dir` to create a repo when `--push_to_hub`

    return args


def main():
    args = parse_args()

    # Sending telemetry. Tracking the example usage helps us better allocate resources to maintain them.
    # information sent is the one passed as arguments along with your Python/PyTorch versions.
    send_example_telemetry("run_clm_no_trainer", args)

    # Initialize the accelerator. We will let the accelerator handle device placement for us in this exa
    # If we're using tracking, we also need to initialize it here and it will by default pick up all sup
    # in the environment
    accelerator_log_kwargs = {}

    if args.with_tracking:
        accelerator_log_kwargs["log_with"] = args.report_to
        accelerator_log_kwargs["logging_dir"] = args.output_dir

    accelerator = Accelerator(gradient_accumulation_steps=args.gradient_accumulation_steps, **accelerato

    # Make one log on every process with the configuration for debugging.
    logging.basicConfig(
        format="%(asctime)s - %(levelname)s - %(name)s - %(message)s",
        datefmt="%m/%d/%Y %H:%M:%S",
        level=logging.INFO,
    )
    logger.info(accelerator.state, main_process_only=False)
    if accelerator.is_local_main_process:
        datasets.utils.logging.set_verbosity_warning()
        transformers.utils.logging.set_verbosity_info()
    else:
        datasets.utils.logging.set_verbosity_error()
        transformers.utils.logging.set_verbosity_error()

    # If passed along, set the training seed now.
    if args.seed is not None:
        set_seed(args.seed)

    # Handle the repository creation
    if accelerator.is_main_process:
        if args.push_to_hub:
            if args.hub_model_id is None:
                repo_name = get_full_repo_name(Path(args.output_dir).name, token=args.hub_token)
            else:
                repo_name = args.hub_model_id
            repo = Repository(args.output_dir, clone_from=repo_name)

            with open(os.path.join(args.output_dir, ".gitignore"), "w+") as gitignore:
                if "step_*" not in gitignore:
                    gitignore.write("step_*\n")
                if "epoch_*" not in gitignore:
                    gitignore.write("epoch_*\n")
        elif args.output_dir is not None:
            os.makedirs(args.output_dir, exist_ok=True)
    accelerator.wait_for_everyone()

    # Get the datasets: you can either provide your own CSV/JSON/TXT training and evaluation files (see
    # or just provide the name of one of the public datasets available on the hub at https://huggingface
    # (the dataset will be downloaded automatically from the datasets Hub).
    #
    # For CSV/JSON files, this script will use the column called 'text' or the first column if no column
    # 'text' is found. You can easily tweak this behavior (see below).
    #
    # In distributed training, the load_dataset function guarantee that only one local process can concu
    # download the dataset.
    if args.dataset_name is not None:
        # Downloading and loading a dataset from the hub.
        raw_datasets = load_dataset(args.dataset_name, args.dataset_config_name)
        if "validation" not in raw_datasets.keys():
            raw_datasets["validation"] = load_dataset(
                args.dataset_name,
                args.dataset_config_name,
                split=f"train[:{args.validation_split_percentage}%]",
            )
            raw_datasets["train"] = load_dataset(
                args.dataset_name,
                args.dataset_config_name,
                split=f"train[{args.validation_split_percentage}%:]",
            )
    else:
        data_files = {}
        dataset_args = {}
        if args.train_file is not None:
            data_files["train"] = args.train_file
        if args.validation_file is not None:
            data_files["validation"] = args.validation_file
        extension = args.train_file.split(".")[-1]
        if extension == "txt":
            extension = "text"
            dataset_args["keep_linebreaks"] = not args.no_keep_linebreaks
        raw_datasets = load_dataset(extension, data_files=data_files, **dataset_args)
        # If no validation data is there, validation_split_percentage will be used to divide the dataset
        if "validation" not in raw_datasets.keys():
            raw_datasets["validation"] = load_dataset(
                extension,
                data_files=data_files,
                split=f"train[:{args.validation_split_percentage}%]",
                **dataset_args,
            )
            raw_datasets["train"] = load_dataset(
                extension,
                data_files=data_files,
                split=f"train[{args.validation_split_percentage}%:]",
                **dataset_args,
            )

    # See more about loading any type of standard or custom dataset (from files, python dict, pandas Dat
    # https://huggingface.co/docs/datasets/loading_datasets.html.

    # Load pretrained model and tokenizer
    #
    # In distributed training, the .from_pretrained methods guarantee that only one local process can co
    # download model & vocab.
    if args.config_name:
        config = AutoConfig.from_pretrained(args.config_name)
```

```
elif args.model_name_or_path:
    config = AutoConfig.from_pretrained(args.model_name_or_path)
else:
    config = CONFIG_MAPPING[args.model_type]()
    logger.warning("You are instantiating a new config instance from scratch.")

if args.tokenizer_name:
    tokenizer = AutoTokenizer.from_pretrained(args.tokenizer_name, use_fast=not args.use_slow_tokeni
elif args.model_name_or_path:
    tokenizer = AutoTokenizer.from_pretrained(args.model_name_or_path, use_fast=not args.use_slow_to
else:
    raise ValueError(
        "You are instantiating a new tokenizer from scratch. This is not supported by this script."
        "You can do it from another script, save it, and load it from here, using --tokenizer_name."
    )

if args.model_name_or_path:
    model = AutoModelForCausalLM.from_pretrained(
        args.model_name_or_path,
        from_tf=bool(".ckpt" in args.model_name_or_path),
        config=config,
    )
else:
    logger.info("Training new model from scratch")
    model = AutoModelForCausalLM.from_config(config)

model.resize_token_embeddings(len(tokenizer))

# Preprocessing the datasets.
# First we tokenize all the texts.
column_names = raw_datasets["train"].column_names
text_column_name = "text" if "text" in column_names else column_names[0]

def tokenize_function(examples):
    return tokenizer(examples[text_column_name])

with accelerator.main_process_first():
    tokenized_datasets = raw_datasets.map(
        tokenize_function,
        batched=True,
        num_proc=args.preprocessing_num_workers,
        remove_columns=column_names,
        load_from_cache_file=not args.overwrite_cache,
        desc="Running tokenizer on dataset",
    )

if args.block_size is None:
    block_size = tokenizer.model_max_length
    if block_size > 1024:
        logger.warning(
            f"The tokenizer picked seems to have a very large `model_max_length` ({tokenizer.model_m
            "Picking 1024 instead. You can change that default value by passing --block_size xxx."
        )
    block_size = 1024
else:
    if args.block_size > tokenizer.model_max_length:
        logger.warning(
            f"The block_size passed ({args.block_size}) is larger than the maximum length for the mo
            f"({tokenizer.model_max_length}). Using block_size={tokenizer.model_max_length}."
        )
    block_size = min(args.block_size, tokenizer.model_max_length)

# Main data processing function that will concatenate all texts from our dataset and generate chunks
def group_texts(examples):
    # Concatenate all texts.
    concatenated_examples = {k: list(chain(*examples[k])) for k in examples.keys()}
    total_length = len(concatenated_examples[list(examples.keys())[0]])
    # We drop the small remainder, we could add padding if the model supported it instead of this dr
    # customize this part to your needs.
    if total_length >= block_size:
        total_length = (total_length // block_size) * block_size
    # Split by chunks of max_len.
    result = {
        k: [t[i : i + block_size] for i in range(0, total_length, block_size)]
        for k, t in concatenated_examples.items()
    }
    result["labels"] = result["input_ids"].copy()
    return result

# Note that with `batched=True`, this map processes 1,000 texts together, so group_texts throws away
# for each of those groups of 1,000 texts. You can adjust that batch_size here but a higher value mi
# to preprocess.
#
# To speed up this part, we use multiprocessing. See the documentation of the map method for more in
# https://huggingface.co/docs/datasets/package_reference/main_classes.html#datasets.Dataset.map

with accelerator.main_process_first():
    lm_datasets = tokenized_datasets.map(
        group_texts,
        batched=True,
        num_proc=args.preprocessing_num_workers,
        load_from_cache_file=not args.overwrite_cache,
        desc=f"Grouping texts in chunks of {block_size}",
    )

train_dataset = lm_datasets["train"]
eval_dataset = lm_datasets["validation"]

# Log a few random samples from the training set:
for index in random.sample(range(len(train_dataset)), 3):
    logger.info(f"Sample {index} of the training set: {train_dataset[index]}.")

# DataLoaders creation:
train_dataloader = DataLoader(
    train_dataset, shuffle=True, collate_fn=default_data_collator, batch_size=args.per_device_train_
)
eval_dataloader = DataLoader(
    eval_dataset, collate_fn=default_data_collator, batch_size=args.per_device_eval_batch_size
)

# Optimizer
# Split weights in two groups, one with weight decay and the other not.
no_decay = ["bias", "layer_norm.weight"]
optimizer_grouped_parameters = [
    {
        "params": [p for n, p in model.named_parameters() if not any(nd in n for nd in no_decay)],
        "weight_decay": args.weight_decay,
    },
    {
        "params": [p for n, p in model.named_parameters() if any(nd in n for nd in no_decay)],
        "weight_decay": 0.0,
    },
]
optimizer = torch.optim.AdamW(optimizer_grouped_parameters, lr=args.learning_rate)

# Scheduler and math around the number of training steps.
overrode_max_train_steps = False
num_update_steps_per_epoch = math.ceil(len(train_dataloader) / args.gradient_accumulation_steps)
if args.max_train_steps is None:
    args.max_train_steps = args.num_train_epochs * num_update_steps_per_epoch
    overrode_max_train_steps = True

# New Code
# For Megatron-LM, we need to use `MegatronLMDummyScheduler` instead of regular schedulers
if accelerator.distributed_type == DistributedType.MEGATRON_LM:
    lr_scheduler = MegatronLMDummyScheduler(
        optimizer=optimizer,
        total_num_steps=args.max_train_steps,
        warmup_num_steps=args.num_warmup_steps,
    )
else:
    lr_scheduler = get_scheduler(
        name=args.lr_scheduler_type,
        optimizer=optimizer,
        num_warmup_steps=args.num_warmup_steps * args.gradient_accumulation_steps,
        num_training_steps=args.max_train_steps * args.gradient_accumulation_steps,
    )

# Prepare everything with our `accelerator`.
model, optimizer, train_dataloader, eval_dataloader, lr_scheduler = accelerator.prepare(
    model, optimizer, train_dataloader, eval_dataloader, lr_scheduler
)

# On TPU, the tie weights in our model have been disconnected, so we need to restore the ties.
if accelerator.distributed_type == DistributedType.TPU:
    model.tie_weights()

# We need to recalculate our total training steps as the size of the training dataloader may have ch
num_update_steps_per_epoch = math.ceil(len(train_dataloader) / args.gradient_accumulation_steps)
if overrode_max_train_steps:
    args.max_train_steps = args.num_train_epochs * num_update_steps_per_epoch
# Afterwards we recalculate our number of training epochs
args.num_train_epochs = math.ceil(args.max_train_steps / num_update_steps_per_epoch)

# Figure out how many steps we should save the Accelerator states
checkpointing_steps = args.checkpointing_steps
if checkpointing_steps is not None and checkpointing_steps.isdigit():
    checkpointing_steps = int(checkpointing_steps)

# We need to initialize the trackers we use, and also store our configuration.
# The trackers initializes automatically on the main process.
if args.with_tracking:
    experiment_config = vars(args)
    # TensorBoard cannot log Enums, need the raw value
    experiment_config["lr_scheduler_type"] = experiment_config["lr_scheduler_type"].value
    accelerator.init_trackers("clm_no_trainer", experiment_config)

# Train!
# New Code
# For Megatron-LM, we need to get `global_batch_size` from megatron_lm_plugin
# as it handles the specifics related to data parallelism, tensor model parallelism and pipeline par
if accelerator.distributed_type == DistributedType.MEGATRON_LM:
    total_batch_size = accelerator.state.megatron_lm_plugin.global_batch_size
else:
    total_batch_size = (
        args.per_device_train_batch_size * accelerator.num_processes * args.gradient_accumulation_st
    )

logger.info("***** Running training *****")
logger.info(f"  Num examples = {len(train_dataset)}")
logger.info(f"  Num Epochs = {args.num_train_epochs}")
logger.info(f"  Instantaneous batch size per device = {args.per_device_train_batch_size}")
logger.info(f"  Total train batch size (w. parallel, distributed & accumulation) = {total_batch_size
logger.info(f"  Gradient Accumulation steps = {args.gradient_accumulation_steps}")
logger.info(f"  Total optimization steps = {args.max_train_steps}")
# Only show the progress bar once on each machine.
progress_bar = tqdm(range(args.max_train_steps), disable=not accelerator.is_local_main_process)
completed_steps = 0
starting_epoch = 0

# Potentially load in the weights and states from a previous save
if args.resume_from_checkpoint:
    if args.resume_from_checkpoint is not None or args.resume_from_checkpoint != "":
        accelerator.print(f"Resumed from checkpoint: {args.resume_from_checkpoint}")
        accelerator.load_state(args.resume_from_checkpoint)
        path = os.path.basename(args.resume_from_checkpoint)
    else:
        # Get the most recent checkpoint
        dirs = [f.name for f in os.scandir(os.getcwd()) if f.is_dir()]
        dirs.sort(key=os.path.getctime)
        path = dirs[-1]  # Sorts folders by date modified, most recent checkpoint is the last
    # Extract `epoch_{i}` or `step_{i}`
    training_difference = os.path.splitext(path)[0]

    if "epoch" in training_difference:
        starting_epoch = int(training_difference.replace("epoch_", "")) + 1
        resume_step = None
    else:
        # need to multiply `gradient_accumulation_steps` to reflect real steps
        resume_step = int(training_difference.replace("step_", "")) * args.gradient_accumulation_ste
        starting_epoch = resume_step // len(train_dataloader)
        resume_step -= starting_epoch * len(train_dataloader)

# update the progress_bar if load from checkpoint
progress_bar.update(starting_epoch * num_update_steps_per_epoch)
completed_steps = starting_epoch * num_update_steps_per_epoch

for epoch in range(starting_epoch, args.num_train_epochs):
    model.train()
    if args.with_tracking:
        total_loss = 0
    for step, batch in enumerate(train_dataloader):
        # We need to skip steps until we reach the resumed step
        if args.resume_from_checkpoint and epoch == starting_epoch:
            if resume_step is not None and step < resume_step:
                if step % args.gradient_accumulation_steps == 0:
                    progress_bar.update(1)
                    completed_steps += 1
                continue

        with accelerator.accumulate(model):
            outputs = model(**batch)
            loss = outputs.loss
            # We keep track of the loss at each epoch
            if args.with_tracking:
                total_loss += loss.detach().float()
            accelerator.backward(loss)
            optimizer.step()
            lr_scheduler.step()
            optimizer.zero_grad()

        # Checks if the accelerator has performed an optimization step behind the scenes
        if accelerator.sync_gradients:
            progress_bar.update(1)
            completed_steps += 1

        if isinstance(checkpointing_steps, int):
            if completed_steps % checkpointing_steps == 0:
                output_dir = f"step_{completed_steps }"
                if args.output_dir is not None:
                    output_dir = os.path.join(args.output_dir, output_dir)
                accelerator.save_state(output_dir)
        if completed_steps >= args.max_train_steps:
            break

    model.eval()
    losses = []
    for step, batch in enumerate(eval_dataloader):
        with torch.no_grad():
            outputs = model(**batch)

        loss = outputs.loss
        # New Code
        # For Megatron-LM, the losses are already averaged across the data parallel group
        if accelerator.distributed_type == DistributedType.MEGATRON_LM:
            losses.append(loss)
        else:
            losses.append(accelerator.gather_for_metrics(loss.repeat(args.per_device_eval_batch_size
    try:
        if accelerator.distributed_type == DistributedType.MEGATRON_LM:
            losses = torch.tensor(losses)
        else:
            losses = torch.cat(losses)
        eval_loss = torch.mean(losses)
        perplexity = math.exp(eval_loss)
    except OverflowError:
        perplexity = float("inf")
    logger.info(f"epoch {epoch}: perplexity: {perplexity} eval_loss: {eval_loss}")
```

```
        if args.with_tracking:
            accelerator.log(
                {
                    "perplexity": perplexity,
                    "eval_loss": eval_loss,
                    "train_loss": total_loss.item() / len(train_dataloader),
                    "epoch": epoch,
                    "step": completed_steps,
                },
                step=completed_steps,
            )

        if args.push_to_hub and epoch < args.num_train_epochs - 1:
            accelerator.wait_for_everyone()
            unwrapped_model = accelerator.unwrap_model(model)
            unwrapped_model.save_pretrained(
                args.output_dir, is_main_process=accelerator.is_main_process, save_function=accelerator.
            )
            if accelerator.is_main_process:
                tokenizer.save_pretrained(args.output_dir)
                repo.push_to_hub(
                    commit_message=f"Training in progress epoch {epoch}", blocking=False, auto_lfs_prune
                )

        if args.checkpointing_steps == "epoch":
            output_dir = f"epoch_{epoch}"
            if args.output_dir is not None:
                output_dir = os.path.join(args.output_dir, output_dir)
            accelerator.save_state(output_dir)

    # this is causing some issue with Megatron-LM when using `wandb` at the end of the main function.
    # Everything works fine inspite of commenting this out. (wandb finishes/closes the run without error
    # if args.with_tracking:
    #     accelerator.end_training()

    if args.output_dir is not None:
        accelerator.wait_for_everyone()
        # New Code
        # For Megatron-LM, we need to save the model using `accelerator.save_state`
        if accelerator.distributed_type == DistributedType.MEGATRON_LM:
            accelerator.save_state(args.output_dir)
        else:
            unwrapped_model = accelerator.unwrap_model(model)
            unwrapped_model.save_pretrained(
                args.output_dir, is_main_process=accelerator.is_main_process, save_function=accelerator.
            )
        if accelerator.is_main_process:
            tokenizer.save_pretrained(args.output_dir)
            if args.push_to_hub:
                repo.push_to_hub(commit_message="End of training", auto_lfs_prune=True)

        with open(os.path.join(args.output_dir, "all_results.json"), "w") as f:
            json.dump({"perplexity": perplexity}, f)


if __name__ == "__main__":
    main()
```

accelerate-main/examples/by_feature/gradient_accumulation.py

```
# coding=utf-8
# Copyright 2021 The HuggingFace Inc. team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
import argparse
import os

import evaluate
import torch
from datasets import load_dataset
from torch.optim import AdamW
from torch.utils.data import DataLoader
from transformers import AutoModelForSequenceClassification, AutoTokenizer, get_linear_schedule_with_war

from accelerate import Accelerator, DistributedType


########################################################################
# This is a fully working simple example to use Accelerate
# and perform gradient accumulation
#
# This example trains a Bert base model on GLUE MRPC
# in any of the following settings (with the same script):
#   - single CPU or single GPU
#   - multi GPUS (using PyTorch distributed mode)
#   - (multi) TPUs
#   - fp16 (mixed-precision) or fp32 (normal precision)
#
# To run it in each of these various modes, follow the instructions
# in the readme for examples:
# https://github.com/huggingface/accelerate/tree/main/examples
#
########################################################################


MAX_GPU_BATCH_SIZE = 16
EVAL_BATCH_SIZE = 32


def get_dataloaders(accelerator: Accelerator, batch_size: int = 16):
    """
    Creates a set of `DataLoader`s for the `glue` dataset,
    using "bert-base-cased" as the tokenizer.

    Args:
        accelerator (`Accelerator`):
            An `Accelerator` object
        batch_size (`int`, *optional*):
            The batch size for the train and validation DataLoaders.
    """
    tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
    datasets = load_dataset("glue", "mrpc")

    def tokenize_function(examples):
        # max_length=None => use the model max length (it's actually the default)
        outputs = tokenizer(examples["sentence1"], examples["sentence2"], truncation=True, max_length=No
        return outputs
    # Apply the method we just defined to all the examples in all the splits of the dataset
    # starting with the main process first:
    with accelerator.main_process_first():
        tokenized_datasets = datasets.map(
            tokenize_function,
            batched=True,
            remove_columns=["idx", "sentence1", "sentence2"],
        )

    # We also rename the 'label' column to 'labels' which is the expected name for labels by the models
    # transformers library
    tokenized_datasets = tokenized_datasets.rename_column("label", "labels")

    def collate_fn(examples):
        # On TPU it's best to pad everything to the same length or training will be very slow.
        max_length = 128 if accelerator.distributed_type == DistributedType.TPU else None
        # When using mixed precision we want round multiples of 8/16
        if accelerator.mixed_precision == "fp8":
            pad_to_multiple_of = 16
        elif accelerator.mixed_precision != "no":
            pad_to_multiple_of = 8
        else:
            pad_to_multiple_of = None

        return tokenizer.pad(
            examples,
            padding="longest",
            max_length=max_length,
            pad_to_multiple_of=pad_to_multiple_of,
            return_tensors="pt",
        )

    # Instantiate dataloaders.
    train_dataloader = DataLoader(
        tokenized_datasets["train"], shuffle=True, collate_fn=collate_fn, batch_size=batch_size
    )
    eval_dataloader = DataLoader(
        tokenized_datasets["validation"], shuffle=False, collate_fn=collate_fn, batch_size=EVAL_BATCH_SI
    )

    return train_dataloader, eval_dataloader


# For testing only
if os.environ.get("TESTING_MOCKED_DATALOADERS", None) == "1":
    from accelerate.test_utils.training import mocked_dataloaders

    get_dataloaders = mocked_dataloaders  # noqa: F811


def training_function(config, args):
    # For testing only
    if os.environ.get("TESTING_MOCKED_DATALOADERS", None) == "1":
        config["num_epochs"] = 2
    # New Code #
    gradient_accumulation_steps = int(args.gradient_accumulation_steps)
    # Initialize accelerator
    accelerator = Accelerator(
        cpu=args.cpu, mixed_precision=args.mixed_precision, gradient_accumulation_steps=gradient_accumul
    )
    if accelerator.distributed_type == DistributedType.TPU and gradient_accumulation_steps > 1:
        raise NotImplementedError(
            "Gradient accumulation on TPUs is currently not supported. Pass `gradient_accumulation_steps
        )
    # Sample hyper-parameters for learning rate, batch size, seed and a few other HPs
    lr = config["lr"]
    num_epochs = int(config["num_epochs"])
    seed = int(config["seed"])
    batch_size = int(config["batch_size"])

    metric = evaluate.load("glue", "mrpc")

    set_seed(seed)
    train_dataloader, eval_dataloader = get_dataloaders(accelerator, batch_size)
    # Instantiate the model (we build the model here so that the seed also control new weights initializ
    model = AutoModelForSequenceClassification.from_pretrained("bert-base-cased", return_dict=True)

    # We could avoid this line since the accelerator is set with `device_placement=True` (default value)
    # Note that if you are placing tensors on devices manually, this line absolutely needs to be before
    # creation otherwise training will not work on TPU (`accelerate` will kindly throw an error to make
    model = model.to(accelerator.device)

    # Instantiate optimizer
    optimizer = AdamW(params=model.parameters(), lr=lr)

    # Instantiate scheduler
    lr_scheduler = get_linear_schedule_with_warmup(
        optimizer=optimizer,
        num_warmup_steps=100,
        num_training_steps=(len(train_dataloader) * num_epochs),
    )

    # Prepare everything
    # There is no specific order to remember, we just need to unpack the objects in the same order we ga
    # prepare method.
    model, optimizer, train_dataloader, eval_dataloader, lr_scheduler = accelerator.prepare(
        model, optimizer, train_dataloader, eval_dataloader, lr_scheduler
    )

    # Now we train the model
    for epoch in range(num_epochs):
        model.train()
        for step, batch in enumerate(train_dataloader):
            # We could avoid this line since we set the accelerator with `device_placement=True`.
            batch.to(accelerator.device)
            # New code #
            # We use the new `accumulate` context manager to perform gradient accumulation
            # We also currently do not support TPUs nor advise it as bugs were found on the XLA side whe
            with accelerator.accumulate(model):
                output = model(**batch)
                loss = output.loss
                accelerator.backward(loss)
                optimizer.step()
                lr_scheduler.step()
                optimizer.zero_grad()

        model.eval()
        for step, batch in enumerate(eval_dataloader):
            # We could avoid this line since we set the accelerator with `device_placement=True`.
            batch.to(accelerator.device)
            with torch.no_grad():
                outputs = model(**batch)
            predictions = outputs.logits.argmax(dim=-1)
            predictions, references = accelerator.gather_for_metrics((predictions, batch["labels"]))
            metric.add_batch(
                predictions=predictions,
                references=references,
            )

        eval_metric = metric.compute()
        # Use accelerator.print to print only on the main process.
        accelerator.print(f"epoch {epoch}:", eval_metric)


def main():
    parser = argparse.ArgumentParser(description="Simple example of training script.")
    parser.add_argument(
        "--mixed_precision",
        type=str,
        default=None,
        choices=["no", "fp16", "bf16", "fp8"],
        help="Whether to use mixed precision. Choose"
        "between fp16 and bf16 (bfloat16). Bf16 requires PyTorch >= 1.10."
        "and an Nvidia Ampere GPU.",
    )
    # New Code #
    parser.add_argument(
        "--gradient_accumulation_steps",
        type=int,
        default=1,
        help="The number of minibatches to be ran before gradients are accumulated.",
    )
    parser.add_argument("--cpu", action="store_true", help="If passed, will train on the CPU.")
    args = parser.parse_args()
    config = {"lr": 2e-5, "num_epochs": 3, "seed": 42, "batch_size": 16}
    training_function(config, args)


if __name__ == "__main__":
    main()
```

accelerate-main/examples/by_feature/tracking.py

```
# coding=utf-8
# Copyright 2021 The HuggingFace Inc. team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
import argparse
import os

import evaluate
import torch
from datasets import load_dataset
from torch.optim import AdamW
from torch.utils.data import DataLoader
from transformers import AutoModelForSequenceClassification, AutoTokenizer, get_linear_schedule_with_war

from accelerate import Accelerator, DistributedType


########################################################################
# This is a fully working simple example to use Accelerate,
# specifically showcasing the experiment tracking capability,
# and builds off the `nlp_example.py` script.
#
# This example trains a Bert base model on GLUE MRPC
# in any of the following settings (with the same script):
#   - single CPU or single GPU
#   - multi GPUS (using PyTorch distributed mode)
#   - (multi) TPUs
#   - fp16 (mixed-precision) or fp32 (normal precision)
#
# To help focus on the differences in the code, building `DataLoaders`
# was refactored into its own function.
# New additions from the base script can be found quickly by
# looking for the # New Code # tags
#
# To run it in each of these various modes, follow the instructions
# in the readme for examples:
# https://github.com/huggingface/accelerate/tree/main/examples
#
########################################################################

MAX_GPU_BATCH_SIZE = 16
EVAL_BATCH_SIZE = 32


def get_dataloaders(accelerator: Accelerator, batch_size: int = 16):
    """
    Creates a set of `DataLoader`s for the `glue` dataset,
    using "bert-base-cased" as the tokenizer.

    Args:
        accelerator (`Accelerator`):
            An `Accelerator` object
        batch_size (`int`, *optional*):
            The batch size for the train and validation DataLoaders.
    """
    tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
    datasets = load_dataset("glue", "mrpc")
    def tokenize_function(examples):
        # max_length=None => use the model max length (it's actually the default)
        outputs = tokenizer(examples["sentence1"], examples["sentence2"], truncation=True, max_length=No
        return outputs

    # Apply the method we just defined to all the examples in all the splits of the dataset
    # starting with the main process first:
    with accelerator.main_process_first():
        tokenized_datasets = datasets.map(
            tokenize_function,
            batched=True,
            remove_columns=["idx", "sentence1", "sentence2"],
        )

    # We also rename the 'label' column to 'labels' which is the expected name for labels by the models
    # transformers library
    tokenized_datasets = tokenized_datasets.rename_column("label", "labels")

    def collate_fn(examples):
        # On TPU it's best to pad everything to the same length or training will be very slow.
        max_length = 128 if accelerator.distributed_type == DistributedType.TPU else None
        # When using mixed precision we want round multiples of 8/16
        if accelerator.mixed_precision == "fp8":
            pad_to_multiple_of = 16
        elif accelerator.mixed_precision != "no":
            pad_to_multiple_of = 8
        else:
            pad_to_multiple_of = None

        return tokenizer.pad(
            examples,
            padding="longest",
            max_length=max_length,
            pad_to_multiple_of=pad_to_multiple_of,
            return_tensors="pt",
        )

    # Instantiate dataloaders.
    train_dataloader = DataLoader(
        tokenized_datasets["train"], shuffle=True, collate_fn=collate_fn, batch_size=batch_size
    )
    eval_dataloader = DataLoader(
        tokenized_datasets["validation"], shuffle=False, collate_fn=collate_fn, batch_size=EVAL_BATCH_SI
    )

    return train_dataloader, eval_dataloader


# For testing only
if os.environ.get("TESTING_MOCKED_DATALOADERS", None) == "1":
    from accelerate.test_utils.training import mocked_dataloaders

    get_dataloaders = mocked_dataloaders  # noqa: F811


def training_function(config, args):
    # For testing only
    if os.environ.get("TESTING_MOCKED_DATALOADERS", None) == "1":
        config["num_epochs"] = 2
    # Initialize Accelerator

    # New Code #
    # We pass in "all" to `log_with` to grab all available trackers in the environment
    # Note: If using a custom `Tracker` class, should be passed in here such as:
    # >>> log_with = ["all", MyCustomTrackerClassInstance()]
    if args.with_tracking:
        accelerator = Accelerator(
            cpu=args.cpu, mixed_precision=args.mixed_precision, log_with="all", logging_dir=args.logging
        )
    else:
        accelerator = Accelerator(cpu=args.cpu, mixed_precision=args.mixed_precision)
    # Sample hyper-parameters for learning rate, batch size, seed and a few other HPs
    lr = config["lr"]
```

```
num_epochs = int(config["num_epochs"])
seed = int(config["seed"])
batch_size = int(config["batch_size"])
set_seed(seed)

train_dataloader, eval_dataloader = get_dataloaders(accelerator, batch_size)
metric = evaluate.load("glue", "mrpc")

# If the batch size is too big we use gradient accumulation
gradient_accumulation_steps = 1
if batch_size > MAX_GPU_BATCH_SIZE and accelerator.distributed_type != DistributedType.TPU:
    gradient_accumulation_steps = batch_size // MAX_GPU_BATCH_SIZE
    batch_size = MAX_GPU_BATCH_SIZE

# Instantiate the model (we build the model here so that the seed also control new weights initializ
model = AutoModelForSequenceClassification.from_pretrained("bert-base-cased", return_dict=True)

# We could avoid this line since the accelerator is set with `device_placement=True` (default value)
# Note that if you are placing tensors on devices manually, this line absolutely needs to be before
# creation otherwise training will not work on TPU (`accelerate` will kindly throw an error to make
model = model.to(accelerator.device)

# Instantiate optimizer
optimizer = AdamW(params=model.parameters(), lr=lr)

# Instantiate scheduler
lr_scheduler = get_linear_schedule_with_warmup(
    optimizer=optimizer,
    num_warmup_steps=100,
    num_training_steps=(len(train_dataloader) * num_epochs) // gradient_accumulation_steps,
)

# Prepare everything
# There is no specific order to remember, we just need to unpack the objects in the same order we ga
# prepare method.
model, optimizer, train_dataloader, eval_dataloader, lr_scheduler = accelerator.prepare(
    model, optimizer, train_dataloader, eval_dataloader, lr_scheduler
)

# New Code #
# We need to initialize the trackers we use. Overall configurations can also be stored
if args.with_tracking:
    run = os.path.split(__file__)[-1].split(".")[0]
    accelerator.init_trackers(run, config)

# Now we train the model
for epoch in range(num_epochs):
    model.train()
    # New Code #
    # For our tracking example, we will log the total loss of each epoch
    if args.with_tracking:
        total_loss = 0
    for step, batch in enumerate(train_dataloader):
        # We could avoid this line since we set the accelerator with `device_placement=True`.
        batch.to(accelerator.device)
        outputs = model(**batch)
        loss = outputs.loss
        # New Code #
        if args.with_tracking:
            total_loss += loss.detach().float()
        loss = loss / gradient_accumulation_steps
        accelerator.backward(loss)
        if step % gradient_accumulation_steps == 0:
            optimizer.step()
            lr_scheduler.step()
            optimizer.zero_grad()

    model.eval()
    for step, batch in enumerate(eval_dataloader):
        # We could avoid this line since we set the accelerator with `device_placement=True` (the de
        batch.to(accelerator.device)
        with torch.no_grad():
                outputs = model(**batch)
            predictions = outputs.logits.argmax(dim=-1)
            predictions, references = accelerator.gather_for_metrics((predictions, batch["labels"]))
            metric.add_batch(
                predictions=predictions,
                references=references,
            )

        eval_metric = metric.compute()
        # Use accelerator.print to print only on the main process.
        accelerator.print(f"epoch {epoch}:", eval_metric)

        # New Code #
        # To actually log, we call `Accelerator.log`
        # The values passed can be of `str`, `int`, `float` or `dict` of `str` to `float`/`int`
        if args.with_tracking:
            accelerator.log(
                {
                    "accuracy": eval_metric["accuracy"],
                    "f1": eval_metric["f1"],
                    "train_loss": total_loss.item() / len(train_dataloader),
                    "epoch": epoch,
                },
                step=epoch,
            )

    # New Code #
    # When a run is finished, you should call `accelerator.end_training()`
    # to close all of the open trackers
    if args.with_tracking:
        accelerator.end_training()


def main():
    parser = argparse.ArgumentParser(description="Simple example of training script.")
    parser.add_argument(
        "--mixed_precision",
        type=str,
        default=None,
        choices=["no", "fp16", "bf16", "fp8"],
        help="Whether to use mixed precision. Choose"
        "between fp16 and bf16 (bfloat16). Bf16 requires PyTorch >= 1.10."
        "and an Nvidia Ampere GPU.",
    )
    parser.add_argument("--cpu", action="store_true", help="If passed, will train on the CPU.")
    parser.add_argument(
        "--with_tracking",
        action="store_true",
        help="Whether to load in all available experiment trackers from the environment and use them for
    )
    parser.add_argument(
        "--logging_dir",
        type=str,
        default="logs",
        help="Location on where to store experiment tracking logs`",
    )
    args = parser.parse_args()
    config = {"lr": 2e-5, "num_epochs": 3, "seed": 42, "batch_size": 16}
    training_function(config, args)


if __name__ == "__main__":
    main()
```

accelerate-main/examples/by_feature/checkpointing.py

```python
# coding=utf-8
# Copyright 2021 The HuggingFace Inc. team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
import argparse
import os

import evaluate
import torch
from datasets import load_dataset
from torch.optim import AdamW
from torch.utils.data import DataLoader
from transformers import AutoModelForSequenceClassification, AutoTokenizer, get_linear_schedule_with_war

from accelerate import Accelerator, DistributedType


########################################################################
# This is a fully working simple example to use Accelerate,
# specifically showcasing the checkpointing capability,
# and builds off the `nlp_example.py` script.
#
# This example trains a Bert base model on GLUE MRPC
# in any of the following settings (with the same script):
#   - single CPU or single GPU
#   - multi GPUS (using PyTorch distributed mode)
#   - (multi) TPUs
#   - fp16 (mixed-precision) or fp32 (normal precision)
#
# To help focus on the differences in the code, building `DataLoaders`
# was refactored into its own function.
# New additions from the base script can be found quickly by
# looking for the # New Code # tags
#
# To run it in each of these various modes, follow the instructions
# in the readme for examples:
# https://github.com/huggingface/accelerate/tree/main/examples
#
########################################################################

MAX_GPU_BATCH_SIZE = 16
EVAL_BATCH_SIZE = 32


def get_dataloaders(accelerator: Accelerator, batch_size: int = 16):
    """
    Creates a set of `DataLoader`s for the `glue` dataset,
    using "bert-base-cased" as the tokenizer.

    Args:
        accelerator (`Accelerator`):
            An `Accelerator` object
        batch_size (`int`, *optional*):
            The batch size for the train and validation DataLoaders.
    """
    tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
    datasets = load_dataset("glue", "mrpc")
    def tokenize_function(examples):
        # max_length=None => use the model max length (it's actually the default)
        outputs = tokenizer(examples["sentence1"], examples["sentence2"], truncation=True, max_length=No
        return outputs

    # Apply the method we just defined to all the examples in all the splits of the dataset
    # starting with the main process first:
    with accelerator.main_process_first():
        tokenized_datasets = datasets.map(
            tokenize_function,
            batched=True,
            remove_columns=["idx", "sentence1", "sentence2"],
        )

    # We also rename the 'label' column to 'labels' which is the expected name for labels by the models
    # transformers library
    tokenized_datasets = tokenized_datasets.rename_column("label", "labels")

    def collate_fn(examples):
        # On TPU it's best to pad everything to the same length or training will be very slow.
        max_length = 128 if accelerator.distributed_type == DistributedType.TPU else None
        # When using mixed precision we want round multiples of 8/16
        if accelerator.mixed_precision == "fp8":
            pad_to_multiple_of = 16
        elif accelerator.mixed_precision != "no":
            pad_to_multiple_of = 8
        else:
            pad_to_multiple_of = None

        return tokenizer.pad(
            examples,
            padding="longest",
            max_length=max_length,
            pad_to_multiple_of=pad_to_multiple_of,
            return_tensors="pt",
        )

    # Instantiate dataloaders.
    train_dataloader = DataLoader(
        tokenized_datasets["train"], shuffle=True, collate_fn=collate_fn, batch_size=batch_size
    )
    eval_dataloader = DataLoader(
        tokenized_datasets["validation"], shuffle=False, collate_fn=collate_fn, batch_size=EVAL_BATCH_SI
    )

    return train_dataloader, eval_dataloader


# For testing only
if os.environ.get("TESTING_MOCKED_DATALOADERS", None) == "1":
    from accelerate.test_utils.training import mocked_dataloaders

    get_dataloaders = mocked_dataloaders  # noqa: F811


def training_function(config, args):
    # For testing only
    if os.environ.get("TESTING_MOCKED_DATALOADERS", None) == "1":
        config["num_epochs"] = 2
    # Initialize accelerator
    accelerator = Accelerator(cpu=args.cpu, mixed_precision=args.mixed_precision)
    # Sample hyper-parameters for learning rate, batch size, seed and a few other HPs
    lr = config["lr"]
    num_epochs = int(config["num_epochs"])
    seed = int(config["seed"])
    batch_size = int(config["batch_size"])

    # New Code #
    # Parse out whether we are saving every epoch or after a certain number of batches
    if hasattr(args.checkpointing_steps, "isdigit"):
        if args.checkpointing_steps == "epoch":
            checkpointing_steps = args.checkpointing_steps
        elif args.checkpointing_steps.isdigit():
            checkpointing_steps = int(args.checkpointing_steps)
        else:
            raise ValueError(
                f"Argument `checkpointing_steps` must be either a number or `epoch`. `{args.checkpointin
            )
    else:
        checkpointing_steps = None

    set_seed(seed)

    train_dataloader, eval_dataloader = get_dataloaders(accelerator, batch_size)
    metric = evaluate.load("glue", "mrpc")

    # If the batch size is too big we use gradient accumulation
    gradient_accumulation_steps = 1
    if batch_size > MAX_GPU_BATCH_SIZE and accelerator.distributed_type != DistributedType.TPU:
        gradient_accumulation_steps = batch_size // MAX_GPU_BATCH_SIZE
        batch_size = MAX_GPU_BATCH_SIZE

    # Instantiate the model (we build the model here so that the seed also control new weights initializ
    model = AutoModelForSequenceClassification.from_pretrained("bert-base-cased", return_dict=True)

    # We could avoid this line since the accelerator is set with `device_placement=True` (default value)
    # Note that if you are placing tensors on devices manually, this line absolutely needs to be before
    # creation otherwise training will not work on TPU (`accelerate` will kindly throw an error to make
    model = model.to(accelerator.device)

    # Instantiate optimizer
    optimizer = AdamW(params=model.parameters(), lr=lr)

    # Instantiate scheduler
    lr_scheduler = get_linear_schedule_with_warmup(
        optimizer=optimizer,
        num_warmup_steps=100,
        num_training_steps=(len(train_dataloader) * num_epochs) // gradient_accumulation_steps,
    )

    # Prepare everything
    # There is no specific order to remember, we just need to unpack the objects in the same order we ga
    # prepare method.
    model, optimizer, train_dataloader, eval_dataloader, lr_scheduler = accelerator.prepare(
        model, optimizer, train_dataloader, eval_dataloader, lr_scheduler
    )

    # New Code #
    # We need to keep track of how many total steps we have iterated over
    overall_step = 0
    # We also need to keep track of the stating epoch so files are named properly
    starting_epoch = 0

    # We need to load the checkpoint back in before training here with `load_state`
    # The total number of epochs is adjusted based on where the state is being loaded from,
    # as we assume continuation of the same training script
    if args.resume_from_checkpoint:
        if args.resume_from_checkpoint is not None or args.resume_from_checkpoint != "":
            accelerator.print(f"Resumed from checkpoint: {args.resume_from_checkpoint}")
            accelerator.load_state(args.resume_from_checkpoint)
            path = os.path.basename(args.resume_from_checkpoint)
        else:
            # Get the most recent checkpoint
            dirs = [f.name for f in os.scandir(os.getcwd()) if f.is_dir()]
            dirs.sort(key=os.path.getctime)
            path = dirs[-1]  # Sorts folders by date modified, most recent checkpoint is the last
        # Extract `epoch_{i}` or `step_{i}`
        training_difference = os.path.splitext(path)[0]

        if "epoch" in training_difference:
            starting_epoch = int(training_difference.replace("epoch_", "")) + 1
            resume_step = None
        else:
            resume_step = int(training_difference.replace("step_", ""))
            starting_epoch = resume_step // len(train_dataloader)
```

```
                resume_step -= starting_epoch * len(train_dataloader)

        # Now we train the model
        for epoch in range(starting_epoch, num_epochs):
            model.train()
            # New Code #
            if args.resume_from_checkpoint and epoch == starting_epoch and resume_step is not None:
                # We need to skip steps until we reach the resumed step
                train_dataloader = accelerator.skip_first_batches(train_dataloader, resume_step)
                overall_step += resume_step
            for step, batch in enumerate(train_dataloader):
                # We could avoid this line since we set the accelerator with `device_placement=True`.
                batch.to(accelerator.device)
                outputs = model(**batch)
                loss = outputs.loss
                loss = loss / gradient_accumulation_steps
                accelerator.backward(loss)
                if step % gradient_accumulation_steps == 0:
                    optimizer.step()
                    lr_scheduler.step()
                    optimizer.zero_grad()
                # New Code #
                overall_step += 1

                # New Code #
                # We save the model, optimizer, lr_scheduler, and seed states by calling `save_state`
                # These are saved to folders named `step_{overall_step}`
                # Will contain files: "pytorch_model.bin", "optimizer.bin", "scheduler.bin", and "random_sta
                # If mixed precision was used, will also save a "scalar.bin" file
                if isinstance(checkpointing_steps, int):
                    output_dir = f"step_{overall_step}"
                    if overall_step % checkpointing_steps == 0:
                        if args.output_dir is not None:
                            output_dir = os.path.join(args.output_dir, output_dir)
                        accelerator.save_state(output_dir)

            model.eval()
            for step, batch in enumerate(eval_dataloader):
                # We could avoid this line since we set the accelerator with `device_placement=True` (the de
                batch.to(accelerator.device)
                with torch.no_grad():
                    outputs = model(**batch)
                predictions = outputs.logits.argmax(dim=-1)
                predictions, references = accelerator.gather_for_metrics((predictions, batch["labels"]))
                metric.add_batch(
                    predictions=predictions,
                    references=references,
                )

            eval_metric = metric.compute()
            # Use accelerator.print to print only on the main process.
            accelerator.print(f"epoch {epoch}:", eval_metric)

            # New Code #
            # We save the model, optimizer, lr_scheduler, and seed states by calling `save_state`
            # These are saved to folders named `epoch_{epoch}`
            # Will contain files: "pytorch_model.bin", "optimizer.bin", "scheduler.bin", and "random_states.
            # If mixed precision was used, will also save a "scalar.bin" file
            if checkpointing_steps == "epoch":
                output_dir = f"epoch_{epoch}"
                if args.output_dir is not None:
                    output_dir = os.path.join(args.output_dir, output_dir)
                accelerator.save_state(output_dir)


    def main():
        parser = argparse.ArgumentParser(description="Simple example of training script.")
        parser.add_argument(
            "--mixed_precision",
            type=str,
            default=None,
            choices=["no", "fp16", "bf16", "fp8"],
            help="Whether to use mixed precision. Choose"
            "between fp16 and bf16 (bfloat16). Bf16 requires PyTorch >= 1.10."
            "and an Nvidia Ampere GPU.",
        )
        parser.add_argument("--cpu", action="store_true", help="If passed, will train on the CPU.")
        parser.add_argument(
            "--checkpointing_steps",
            type=str,
            default=None,
            help="Whether the various states should be saved at the end of every n steps, or 'epoch' for eac
        )
        parser.add_argument(
            "--output_dir",
            type=str,
            default=".",
            help="Optional save directory where all checkpoint folders will be stored. Default is the curren
        )
        parser.add_argument(
            "--resume_from_checkpoint",
            type=str,
            default=None,
            help="If the training should continue from a checkpoint folder.",
        )
        args = parser.parse_args()
        config = {"lr": 2e-5, "num_epochs": 3, "seed": 42, "batch_size": 16}
        training_function(config, args)


    if __name__ == "__main__":
        main()
```

accelerate-main/examples/by_feature/memory.py

```
# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
import argparse
import os

# New Code #
import evaluate
import torch
from datasets import load_dataset
from torch.optim import AdamW
from torch.utils.data import DataLoader
from transformers import AutoModelForSequenceClassification, AutoTokenizer, get_linear_schedule_with_war

from accelerate import Accelerator, DistributedType
from accelerate.utils import find_executable_batch_size


########################################################################
# This is a fully working simple example to use Accelerate,
# specifically showcasing how to ensure out-of-memory errors never
# interrupt training, and builds off the `nlp_example.py` script.
#
# This example trains a Bert base model on GLUE MRPC
# in any of the following settings (with the same script):
#   - single CPU or single GPU
#   - multi GPUS (using PyTorch distributed mode)
#   - (multi) TPUs
#   - fp16 (mixed-precision) or fp32 (normal precision)
#
# New additions from the base script can be found quickly by
# looking for the # New Code # tags
#
# To run it in each of these various modes, follow the instructions
# in the readme for examples:
# https://github.com/huggingface/accelerate/tree/main/examples
#
########################################################################


MAX_GPU_BATCH_SIZE = 16
EVAL_BATCH_SIZE = 32


def get_dataloaders(accelerator: Accelerator, batch_size: int = 16):
    """
    Creates a set of `DataLoader`s for the `glue` dataset,
    using "bert-base-cased" as the tokenizer.

    Args:
        accelerator (`Accelerator`):
            An `Accelerator` object
        batch_size (`int`, *optional*):
            The batch size for the train and validation DataLoaders.
    """
    tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
    datasets = load_dataset("glue", "mrpc")
    def tokenize_function(examples):
        # max_length=None => use the model max length (it's actually the default)
        outputs = tokenizer(examples["sentence1"], examples["sentence2"], truncation=True, max_length=No
        return outputs

    # Apply the method we just defined to all the examples in all the splits of the dataset
    # starting with the main process first:
    with accelerator.main_process_first():
        tokenized_datasets = datasets.map(
            tokenize_function,
            batched=True,
            remove_columns=["idx", "sentence1", "sentence2"],
        )

    # We also rename the 'label' column to 'labels' which is the expected name for labels by the models
    # transformers library
    tokenized_datasets = tokenized_datasets.rename_column("label", "labels")

    def collate_fn(examples):
        # On TPU it's best to pad everything to the same length or training will be very slow.
        max_length = 128 if accelerator.distributed_type == DistributedType.TPU else None
        # When using mixed precision we want round multiples of 8/16
        if accelerator.mixed_precision == "fp8":
            pad_to_multiple_of = 16
        elif accelerator.mixed_precision != "no":
            pad_to_multiple_of = 8
        else:
            pad_to_multiple_of = None

        return tokenizer.pad(
            examples,
            padding="longest",
            max_length=max_length,
            pad_to_multiple_of=pad_to_multiple_of,
            return_tensors="pt",
        )

    # Instantiate dataloaders.
    train_dataloader = DataLoader(
        tokenized_datasets["train"], shuffle=True, collate_fn=collate_fn, batch_size=batch_size
    )
    eval_dataloader = DataLoader(
        tokenized_datasets["validation"], shuffle=False, collate_fn=collate_fn, batch_size=EVAL_BATCH_SI
    )

    return train_dataloader, eval_dataloader


# For testing only
if os.environ.get("TESTING_MOCKED_DATALOADERS", None) == "1":
    from accelerate.test_utils.training import mocked_dataloaders

    get_dataloaders = mocked_dataloaders  # noqa: F811


def training_function(config, args):
    # For testing only
    if os.environ.get("TESTING_MOCKED_DATALOADERS", None) == "1":
        config["num_epochs"] = 2
    # Initialize accelerator
    accelerator = Accelerator(cpu=args.cpu, mixed_precision=args.mixed_precision)
    # Sample hyper-parameters for learning rate, batch size, seed and a few other HPs
    lr = config["lr"]
    num_epochs = int(config["num_epochs"])
    seed = int(config["seed"])
    batch_size = int(config["batch_size"])

    metric = evaluate.load("glue", "mrpc")

    # New Code #
    # We now can define an inner training loop function. It should take a batch size as the only paramet
    # and build the dataloaders in there.
    # It also gets our decorator
    @find_executable_batch_size(starting_batch_size=batch_size)
    def inner_training_loop(batch_size):
        # And now just move everything below under this function
        # We need to bring in the Accelerator object from earlier
        nonlocal accelerator
        # And reset all of its attributes that could hold onto any memory:
        accelerator.free_memory()

        # Then we can declare the model, optimizer, and everything else:
        set_seed(seed)

        # Instantiate the model (we build the model here so that the seed also control new weights initi
        model = AutoModelForSequenceClassification.from_pretrained("bert-base-cased", return_dict=True)

        # We could avoid this line since the accelerator is set with `device_placement=True` (default va
        # Note that if you are placing tensors on devices manually, this line absolutely needs to be bef
        # creation otherwise training will not work on TPU (`accelerate` will kindly throw an error to m
        model = model.to(accelerator.device)

        # Instantiate optimizer
        optimizer = AdamW(params=model.parameters(), lr=lr)
        train_dataloader, eval_dataloader = get_dataloaders(accelerator, batch_size)

        # Instantiate scheduler
        lr_scheduler = get_linear_schedule_with_warmup(
            optimizer=optimizer,
            num_warmup_steps=100,
            num_training_steps=(len(train_dataloader) * num_epochs),
        )

        # Prepare everything
        # There is no specific order to remember, we just need to unpack the objects in the same order w
        # prepare method.
        model, optimizer, train_dataloader, eval_dataloader, lr_scheduler = accelerator.prepare(
            model, optimizer, train_dataloader, eval_dataloader, lr_scheduler
        )

        # Now we train the model
        for epoch in range(num_epochs):
            model.train()
            for step, batch in enumerate(train_dataloader):
                # We could avoid this line since we set the accelerator with `device_placement=True`.
                batch.to(accelerator.device)
                outputs = model(**batch)
                loss = outputs.loss
                accelerator.backward(loss)
                optimizer.step()
                lr_scheduler.step()
                optimizer.zero_grad()

            model.eval()
            for step, batch in enumerate(eval_dataloader):
                # We could avoid this line since we set the accelerator with `device_placement=True`.
                batch.to(accelerator.device)
                with torch.no_grad():
                    outputs = model(**batch)
                predictions = outputs.logits.argmax(dim=-1)
                predictions, references = accelerator.gather_for_metrics((predictions, batch["labels"]))
                metric.add_batch(
                    predictions=predictions,
                    references=references,
                )

            eval_metric = metric.compute()
            # Use accelerator.print to print only on the main process.
            accelerator.print(f"epoch {epoch}:", eval_metric)

    # New Code #
    # And call it at the end with no arguments
    # Note: You could also refactor this outside of your training loop function
    inner_training_loop()
def main():
    parser = argparse.ArgumentParser(description="Simple example of training script.")
    parser.add_argument(
        "--mixed_precision",
        type=str,
        default=None,
        choices=["no", "fp16", "bf16", "fp8"],
        help="Whether to use mixed precision. Choose"
        "between fp16 and bf16 (bfloat16). Bf16 requires PyTorch >= 1.10."
        "and an Nvidia Ampere GPU.",
    )
    parser.add_argument("--cpu", action="store_true", help="If passed, will train on the CPU.")
    args = parser.parse_args()
    config = {"lr": 2e-5, "num_epochs": 3, "seed": 42, "batch_size": 16}
    training_function(config, args)


if __name__ == "__main__":
    main()
```

accelerate-main/examples/by_feature/cross_validation.py

```
# coding=utf-8
# Copyright 2022 The HuggingFace Inc. team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
import argparse
from typing import List

import evaluate
import numpy as np
import torch
from datasets import DatasetDict, load_dataset

# New Code #
# We'll be using StratifiedKFold for this example
from sklearn.model_selection import StratifiedKFold
from torch.optim import AdamW
from torch.utils.data import DataLoader
from transformers import AutoModelForSequenceClassification, AutoTokenizer, get_linear_schedule_with_war

from accelerate import Accelerator, DistributedType


########################################################################
# This is a fully working simple example to use Accelerate,
# specifically showcasing how to perform Cross Validation,
# and builds off the `nlp_example.py` script.
#
# This example trains a Bert base model on GLUE MRPC
# in any of the following settings (with the same script):
#   - single CPU or single GPU
#   - multi GPUS (using PyTorch distributed mode)
#   - (multi) TPUs
#   - fp16 (mixed-precision) or fp32 (normal precision)
#
# To help focus on the differences in the code, building `DataLoaders`
# was refactored into its own function.
# New additions from the base script can be found quickly by
# looking for the # New Code # tags
#
# To run it in each of these various modes, follow the instructions
# in the readme for examples:
# https://github.com/huggingface/accelerate/tree/main/examples
#
########################################################################


MAX_GPU_BATCH_SIZE = 16
EVAL_BATCH_SIZE = 32

# New Code #
# We need a different `get_dataloaders` function that will build dataloaders by index


def get_fold_dataloaders(
    accelerator: Accelerator, dataset: DatasetDict, train_idxs: List[int], valid_idxs: List[int], batch_
):
    """
    Gets a set of train, valid, and test dataloaders for a particular fold
    Args:
        accelerator (`Accelerator`):
            The main `Accelerator` object
        train_idxs (list of `int`):
            The split indices for the training dataset
        valid_idxs (list of `int`):
            The split indices for the validation dataset
        batch_size (`int`):
            The size of the minibatch. Default is 16
    """
    tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
    datasets = DatasetDict(
        {
            "train": dataset["train"].select(train_idxs),
            "validation": dataset["train"].select(valid_idxs),
            "test": dataset["validation"],
        }
    )

    def tokenize_function(examples):
        # max_length=None => use the model max length (it's actually the default)
        outputs = tokenizer(examples["sentence1"], examples["sentence2"], truncation=True, max_length=No
        return outputs

    # Apply the method we just defined to all the examples in all the splits of the dataset
    # starting with the main process first:
    with accelerator.main_process_first():
        tokenized_datasets = datasets.map(
            tokenize_function,
            batched=True,
            remove_columns=["idx", "sentence1", "sentence2"],
        )

    # We also rename the 'label' column to 'labels' which is the expected name for labels by the models
    # transformers library
    tokenized_datasets = tokenized_datasets.rename_column("label", "labels")

    def collate_fn(examples):
        # On TPU it's best to pad everything to the same length or training will be very slow.
        max_length = 128 if accelerator.distributed_type == DistributedType.TPU else None
        # When using mixed precision we want round multiples of 8/16
        if accelerator.mixed_precision == "fp8":
            pad_to_multiple_of = 16
        elif accelerator.mixed_precision != "no":
            pad_to_multiple_of = 8
        else:
            pad_to_multiple_of = None

        return tokenizer.pad(
            examples,
            padding="longest",
            max_length=max_length,
            pad_to_multiple_of=pad_to_multiple_of,
            return_tensors="pt",
        )

    # Instantiate dataloaders.
    train_dataloader = DataLoader(
        tokenized_datasets["train"], shuffle=True, collate_fn=collate_fn, batch_size=batch_size
    )
    eval_dataloader = DataLoader(
        tokenized_datasets["validation"], shuffle=False, collate_fn=collate_fn, batch_size=EVAL_BATCH_SI
    )

    test_dataloader = DataLoader(
        tokenized_datasets["test"], shuffle=False, collate_fn=collate_fn, batch_size=EVAL_BATCH_SIZE
    )

    return train_dataloader, eval_dataloader, test_dataloader


def training_function(config, args):
```

```
# New Code #
test_predictions = []
# Download the dataset
datasets = load_dataset("glue", "mrpc")
# Create our splits
kfold = StratifiedKFold(n_splits=int(args.num_folds))
# Initialize accelerator
accelerator = Accelerator(cpu=args.cpu, mixed_precision=args.mixed_precision)
# Sample hyper-parameters for learning rate, batch size, seed and a few other HPs
lr = config["lr"]
num_epochs = int(config["num_epochs"])
seed = int(config["seed"])
batch_size = int(config["batch_size"])

metric = evaluate.load("glue", "mrpc")

# If the batch size is too big we use gradient accumulation
gradient_accumulation_steps = 1
if batch_size > MAX_GPU_BATCH_SIZE and accelerator.distributed_type != DistributedType.TPU:
    gradient_accumulation_steps = batch_size // MAX_GPU_BATCH_SIZE
    batch_size = MAX_GPU_BATCH_SIZE

set_seed(seed)

# New Code #
# Create our folds:
folds = kfold.split(np.zeros(datasets["train"].num_rows), datasets["train"]["label"])
test_references = []
# Iterate over them
for i, (train_idxs, valid_idxs) in enumerate(folds):
    train_dataloader, eval_dataloader, test_dataloader = get_fold_dataloaders(
        accelerator,
        datasets,
        train_idxs,
        valid_idxs,
    )
    # Instantiate the model (we build the model here so that the seed also control new weights initi
    model = AutoModelForSequenceClassification.from_pretrained("bert-base-cased", return_dict=True)

    # We could avoid this line since the accelerator is set with `device_placement=True` (default va
    # Note that if you are placing tensors on devices manually, this line absolutely needs to be bef
    # creation otherwise training will not work on TPU (`accelerate` will kindly throw an error to m
    model = model.to(accelerator.device)

    # Instantiate optimizer
    optimizer = AdamW(params=model.parameters(), lr=lr)

    # Instantiate scheduler
    lr_scheduler = get_linear_schedule_with_warmup(
        optimizer=optimizer,
        num_warmup_steps=100,
        num_training_steps=(len(train_dataloader) * num_epochs) // gradient_accumulation_steps,
    )

    # Prepare everything
    # There is no specific order to remember, we just need to unpack the objects in the same order w
    # prepare method.
    model, optimizer, train_dataloader, eval_dataloader, lr_scheduler = accelerator.prepare(
        model, optimizer, train_dataloader, eval_dataloader, lr_scheduler
    )

    # Now we train the model
    for epoch in range(num_epochs):
        model.train()
        for step, batch in enumerate(train_dataloader):
            # We could avoid this line since we set the accelerator with `device_placement=True`.
            batch.to(accelerator.device)
            outputs = model(**batch)
            loss = outputs.loss
            loss = loss / gradient_accumulation_steps
            accelerator.backward(loss)
            if step % gradient_accumulation_steps == 0:
                    optimizer.step()
                    lr_scheduler.step()
                    optimizer.zero_grad()

            model.eval()
            for step, batch in enumerate(eval_dataloader):
                # We could avoid this line since we set the accelerator with `device_placement=True`.
                batch.to(accelerator.device)
                with torch.no_grad():
                    outputs = model(**batch)
                predictions = outputs.logits.argmax(dim=-1)
                predictions, references = accelerator.gather_for_metrics((predictions, batch["labels"]))
                metric.add_batch(
                    predictions=predictions,
                    references=references,
                )

            eval_metric = metric.compute()
            # Use accelerator.print to print only on the main process.
            accelerator.print(f"epoch {epoch}:", eval_metric)

        # New Code #
        # We also run predictions on the test set at the very end
        fold_predictions = []
        for step, batch in enumerate(test_dataloader):
            # We could avoid this line since we set the accelerator with `device_placement=True`.
            batch.to(accelerator.device)
            with torch.no_grad():
                outputs = model(**batch)
            predictions = outputs.logits
            predictions, references = accelerator.gather_for_metrics((predictions, batch["labels"]))
            fold_predictions.append(predictions.cpu())
            if i == 0:
                # We need all of the test predictions
                test_references.append(references.cpu())
        # Use accelerator.print to print only on the main process.
        test_predictions.append(torch.cat(fold_predictions, dim=0))
        # We now need to release all our memory and get rid of the current model, optimizer, etc
        accelerator.free_memory()
    # New Code #
    # Finally we check the accuracy of our folded results:
    test_references = torch.cat(test_references, dim=0)
    preds = torch.stack(test_predictions, dim=0).sum(dim=0).div(int(args.num_folds)).argmax(dim=-1)
    test_metric = metric.compute(predictions=preds, references=test_references)
    accelerator.print("Average test metrics from all folds:", test_metric)


def main():
    parser = argparse.ArgumentParser(description="Simple example of training script.")
    parser.add_argument(
        "--mixed_precision",
        type=str,
        default=None,
        choices=["no", "fp16", "bf16", "fp8"],
        help="Whether to use mixed precision. Choose"
        "between fp16 and bf16 (bfloat16). Bf16 requires PyTorch >= 1.10."
        "and an Nvidia Ampere GPU.",
    )
    parser.add_argument("--cpu", action="store_true", help="If passed, will train on the CPU.")
    # New Code #
    parser.add_argument("--num_folds", type=int, default=3, help="The number of splits to perform across
    args = parser.parse_args()
    config = {"lr": 2e-5, "num_epochs": 3, "seed": 42, "batch_size": 16}
    training_function(config, args)


if __name__ == "__main__":
    main()
```

accelerate-main/examples/by_feature/fsdp_with_peak_mem_tracking.py

```python
# coding=utf-8
# Copyright 2021 The HuggingFace Inc. team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
import argparse
import gc
import os

import evaluate
import torch
from datasets import load_dataset
from torch.utils.data import DataLoader
from transformers import AutoModelForSequenceClassification, AutoTokenizer, get_linear_schedule_with_war

from accelerate import Accelerator, DistributedType


########################################################################
# This is a fully working simple example to use Accelerate
#
# This example trains a Bert base model on GLUE MRPC
# in any of the following settings (with the same script):
#   - single CPU or single GPU
#   - multi GPUS (using PyTorch distributed mode)
#   - (multi) TPUs
#   - fp16 (mixed-precision) or fp32 (normal precision)
#   - FSDP
#
# This example also demonstrates the checkpointing and sharding capabilities
#
# To run it in each of these various modes, follow the instructions
# in the readme for examples:
# https://github.com/huggingface/accelerate/tree/main/examples
#
########################################################################


MAX_GPU_BATCH_SIZE = 16
EVAL_BATCH_SIZE = 32


# New Code #
# Converting Bytes to Megabytes
def b2mb(x):
    return int(x / 2**20)


# New Code #
# This context manager is used to track the peak memory usage of the process
class TorchTracemalloc:
    def __enter__(self):
        gc.collect()
        torch.cuda.empty_cache()
        torch.cuda.reset_max_memory_allocated()  # reset the peak gauge to zero
        self.begin = torch.cuda.memory_allocated()
        return self
    def __exit__(self, *exc):
        gc.collect()
        torch.cuda.empty_cache()
        self.end = torch.cuda.memory_allocated()
        self.peak = torch.cuda.max_memory_allocated()
        self.used = b2mb(self.end - self.begin)
        self.peaked = b2mb(self.peak - self.begin)
        # print(f"delta used/peak {self.used:4d}/{self.peaked:4d}")


# For testing only
if os.environ.get("TESTING_MOCKED_DATALOADERS", None) == "1":
    from accelerate.test_utils.training import mocked_dataloaders

    get_dataloaders = mocked_dataloaders  # noqa: F811


def training_function(config, args):
    # For testing only
    if os.environ.get("TESTING_MOCKED_DATALOADERS", None) == "1":
        config["num_epochs"] = 2
    # Initialize accelerator
    if args.with_tracking:
        accelerator = Accelerator(
            cpu=args.cpu, mixed_precision=args.mixed_precision, log_with="wandb", logging_dir=args.loggi
        )
    else:
        accelerator = Accelerator()
    accelerator.print(accelerator.distributed_type)

    if hasattr(args.checkpointing_steps, "isdigit"):
        if args.checkpointing_steps == "epoch":
            checkpointing_steps = args.checkpointing_steps
        elif args.checkpointing_steps.isdigit():
            checkpointing_steps = int(args.checkpointing_steps)
        else:
            raise ValueError(
                f"Argument `checkpointing_steps` must be either a number or `epoch`. `{args.checkpointin
            )
    else:
        checkpointing_steps = None
    # Sample hyper-parameters for learning rate, batch size, seed and a few other HPs
    lr = config["lr"]
    num_epochs = int(config["num_epochs"])
    seed = int(config["seed"])
    batch_size = int(config["batch_size"])

    # We need to initialize the trackers we use, and also store our configuration
    if args.with_tracking:
        experiment_config = vars(args)
        accelerator.init_trackers("fsdp_glue_no_trainer", experiment_config)

    tokenizer = AutoTokenizer.from_pretrained(args.model_name_or_path)
    datasets = load_dataset("glue", "mrpc")
    metric = evaluate.load("glue", "mrpc")

    def tokenize_function(examples):
        # max_length=None => use the model max length (it's actually the default)
        outputs = tokenizer(examples["sentence1"], examples["sentence2"], truncation=True, max_length=No
        return outputs

    # Apply the method we just defined to all the examples in all the splits of the dataset
    # starting with the main process first:
    with accelerator.main_process_first():
        tokenized_datasets = datasets.map(
            tokenize_function,
            batched=True,
            remove_columns=["idx", "sentence1", "sentence2"],
        )

    # We also rename the 'label' column to 'labels' which is the expected name for labels by the models
    # transformers library
```

```
tokenized_datasets = tokenized_datasets.rename_column("label", "labels")

# If the batch size is too big we use gradient accumulation
gradient_accumulation_steps = 1
if batch_size > MAX_GPU_BATCH_SIZE and accelerator.distributed_type != DistributedType.TPU:
    gradient_accumulation_steps = batch_size // MAX_GPU_BATCH_SIZE
    batch_size = MAX_GPU_BATCH_SIZE

def collate_fn(examples):
    # On TPU it's best to pad everything to the same length or training will be very slow.
    max_length = 128 if accelerator.distributed_type == DistributedType.TPU else None
    # When using mixed precision we want round multiples of 8/16
    if accelerator.mixed_precision == "fp8":
        pad_to_multiple_of = 16
    elif accelerator.mixed_precision != "no":
        pad_to_multiple_of = 8
    else:
        pad_to_multiple_of = None

    return tokenizer.pad(
        examples,
        padding="longest",
        max_length=max_length,
        pad_to_multiple_of=pad_to_multiple_of,
        return_tensors="pt",
    )

# Instantiate dataloaders.
train_dataloader = DataLoader(
    tokenized_datasets["train"], shuffle=True, collate_fn=collate_fn, batch_size=batch_size
)
eval_dataloader = DataLoader(
    tokenized_datasets["validation"], shuffle=False, collate_fn=collate_fn, batch_size=EVAL_BATCH_SI
)

set_seed(seed)

# Instantiate the model (we build the model here so that the seed also control new weights initializ
model = AutoModelForSequenceClassification.from_pretrained(args.model_name_or_path, return_dict=True
# New Code #
# For FSDP feature, it is highly recommended and efficient to prepare the model before creating opti
model = accelerator.prepare(model)
accelerator.print(model)

# Instantiate optimizer
# New Code #
# For FSDP feature, at present it doesn't support multiple parameter groups,
# so we need to create a single parameter group for the whole model
optimizer = torch.optim.AdamW(params=model.parameters(), lr=lr, weight_decay=2e-4)

# Instantiate scheduler
lr_scheduler = get_linear_schedule_with_warmup(
    optimizer=optimizer,
    num_warmup_steps=10,
    num_training_steps=(len(train_dataloader) * num_epochs) // gradient_accumulation_steps,
)

# New Code #
# For FSDP feature, prepare everything except the model as we have already prepared the model
# before creating the optimizer
# There is no specific order to remember, we just need to unpack the objects in the same order we ga
# prepare method.
optimizer, train_dataloader, eval_dataloader, lr_scheduler = accelerator.prepare(
    optimizer, train_dataloader, eval_dataloader, lr_scheduler
)

overall_step = 0

# Potentially load in the weights and states from a previous save
if args.resume_from_checkpoint:
    if args.resume_from_checkpoint is not None or args.resume_from_checkpoint != "":
        accelerator.print(f"Resumed from checkpoint: {args.resume_from_checkpoint}")
```

```
            accelerator.load_state(args.resume_from_checkpoint)
            path = os.path.basename(args.resume_from_checkpoint)
        else:
            # Get the most recent checkpoint
            dirs = [f.name for f in os.scandir(os.getcwd()) if f.is_dir()]
            dirs.sort(key=os.path.getctime)
            path = dirs[-1]  # Sorts folders by date modified, most recent checkpoint is the last
        # Extract `epoch_{i}` or `step_{i}`
        training_difference = os.path.splitext(path)[0]

        if "epoch" in training_difference:
            num_epochs -= int(training_difference.replace("epoch_", ""))
            resume_step = None
        else:
            resume_step = int(training_difference.replace("step_", ""))
            num_epochs -= resume_step // len(train_dataloader)
            # If resuming by step, we also need to know exactly how far into the DataLoader we went
            resume_step = (num_epochs * len(train_dataloader)) - resume_step

    # Now we train the model
    for epoch in range(num_epochs):
        # New Code #
        # context manager to track the peak memory usage during the training epoch
        with TorchTracemalloc() as tracemalloc:
            model.train()
            if args.with_tracking:
                total_loss = 0
            for step, batch in enumerate(train_dataloader):
                # We need to skip steps until we reach the resumed step
                if args.resume_from_checkpoint and epoch == 0:
                    if resume_step is not None and step < resume_step:
                        pass
                # We could avoid this line since we set the accelerator with `device_placement=True`.
                batch.to(accelerator.device)
                outputs = model(**batch)
                loss = outputs.loss
                loss = loss / gradient_accumulation_steps
                # We keep track of the loss at each epoch
                if args.with_tracking:
                    total_loss += loss.detach().float()
                accelerator.backward(loss)
                if step % gradient_accumulation_steps == 0:
                    optimizer.step()
                    lr_scheduler.step()
                    optimizer.zero_grad()
                    # accelerator.print(lr_scheduler.get_lr())

                overall_step += 1

                if isinstance(checkpointing_steps, int):
                    output_dir = f"step_{overall_step}"
                    if overall_step % checkpointing_steps == 0:
                        if args.output_dir is not None:
                            output_dir = os.path.join(args.output_dir, output_dir)
                        accelerator.save_state(output_dir)
        # New Code #
        # Printing the GPU memory usage details such as allocated memory, peak memory, and total memory
        accelerator.print("Memory before entering the train : {}".format(b2mb(tracemalloc.begin)))
        accelerator.print("Memory consumed at the end of the train (end-begin): {}".format(tracemalloc.u
        accelerator.print("Peak Memory consumed during the train (max-begin): {}".format(tracemalloc.pea
        accelerator.print(
            "Total Peak Memory consumed during the train (max): {}".format(
                tracemalloc.peaked + b2mb(tracemalloc.begin)
            )
        )
        # Logging the peak memory usage of the GPU to the tracker
        if args.with_tracking:
            accelerator.log(
                {
                    "train_total_peak_memory": tracemalloc.peaked + b2mb(tracemalloc.begin),
                },
                step=epoch,
            )

        # New Code #
        # context manager to track the peak memory usage during the evaluation
        with TorchTracemalloc() as tracemalloc:
            model.eval()
            for step, batch in enumerate(eval_dataloader):
                # We could avoid this line since we set the accelerator with `device_placement=True`.
                batch.to(accelerator.device)
                with torch.no_grad():
                    outputs = model(**batch)
                predictions = outputs.logits.argmax(dim=-1)
                predictions, references = accelerator.gather_for_metrics((predictions, batch["labels"]))
                metric.add_batch(
                    predictions=predictions,
                    references=references,
                )

            eval_metric = metric.compute()
            # Use accelerator.print to print only on the main process.
            accelerator.print(f"epoch {epoch}:", eval_metric)
            if args.with_tracking:
                accelerator.log(
                    {
                        "accuracy": eval_metric["accuracy"],
                        "f1": eval_metric["f1"],
                        "train_loss": total_loss.item() / len(train_dataloader),
                    },
                    step=epoch,
                )

            if checkpointing_steps == "epoch":
                output_dir = f"epoch_{epoch}"
                if args.output_dir is not None:
                    output_dir = os.path.join(args.output_dir, output_dir)
                accelerator.save_state(output_dir)
        # New Code #
        # Printing the GPU memory usage details such as allocated memory, peak memory, and total memory
        accelerator.print("Memory before entering the eval : {}".format(b2mb(tracemalloc.begin)))
        accelerator.print("Memory consumed at the end of the eval (end-begin): {}".format(tracemalloc.us
        accelerator.print("Peak Memory consumed during the eval (max-begin): {}".format(tracemalloc.peak
        accelerator.print(
            "Total Peak Memory consumed during the eval (max): {}".format(tracemalloc.peaked + b2mb(trac
        )
        # Logging the peak memory usage of the GPU to the tracker
        if args.with_tracking:
            accelerator.log(
                {
                    "eval_total_peak_memory": tracemalloc.peaked + b2mb(tracemalloc.begin),
                },
                step=epoch,
            )

    if args.with_tracking:
        accelerator.end_training()


def main():
    parser = argparse.ArgumentParser(description="Simple example of training script.")
    parser.add_argument(
        "--mixed_precision",
        type=str,
        default=None,
        choices=["no", "fp16", "bf16", "fp8"],
        help="Whether to use mixed precision. Choose"
        "between fp16 and bf16 (bfloat16). Bf16 requires PyTorch >= 1.10."
        "and an Nvidia Ampere GPU.",
    )
    parser.add_argument("--cpu", action="store_true", help="If passed, will train on the CPU.")
    parser.add_argument(
        "--checkpointing_steps",
        type=str,
        default=None,
        help="Whether the various states should be saved at the end of every n steps, or 'epoch' for eac
    )
    parser.add_argument(
        "--resume_from_checkpoint",
        type=str,
        default=None,
        help="If the training should continue from a checkpoint folder.",
    )
    parser.add_argument(
        "--with_tracking",
        action="store_true",
        help="Whether to load in all available experiment trackers from the environment and use them for
    )
    parser.add_argument(
        "--output_dir",
        type=str,
        default=".",
        help="Optional save directory where all checkpoint folders will be stored. Default is the curren
    )
    parser.add_argument(
        "--logging_dir",
        type=str,
        default="logs",
        help="Location on where to store experiment tracking logs`",
    )
    parser.add_argument(
        "--model_name_or_path",
        type=str,
        help="Path to pretrained model or model identifier from huggingface.co/models.",
        required=True,
    )
    args = parser.parse_args()
    config = {"lr": 2e-5, "num_epochs": 3, "seed": 1, "batch_size": 16}
    training_function(config, args)


if __name__ == "__main__":
    main()
```

accelerate-main/examples/by_feature/multi_process_metrics.py

```python
# coding=utf-8
# Copyright 2022 The HuggingFace Inc. team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
import argparse
import os

import evaluate
import torch
from datasets import load_dataset
from torch.optim import AdamW
from torch.utils.data import DataLoader
from transformers import AutoModelForSequenceClassification, AutoTokenizer, get_linear_schedule_with_war

from accelerate import Accelerator, DistributedType


########################################################################
# This is a fully working simple example to use Accelerate,
# specifically showcasing how to properly calculate the metrics on the
# validation dataset when in a distributed system, and builds off the
# `nlp_example.py` script.
#
# This example trains a Bert base model on GLUE MRPC
# in any of the following settings (with the same script):
#   - single CPU or single GPU
#   - multi GPUS (using PyTorch distributed mode)
#   - (multi) TPUs
#   - fp16 (mixed-precision) or fp32 (normal precision)
#
# To help focus on the differences in the code, building `DataLoaders`
# was refactored into its own function.
# New additions from the base script can be found quickly by
# looking for the # New Code # tags
#
# To run it in each of these various modes, follow the instructions
# in the readme for examples:
# https://github.com/huggingface/accelerate/tree/main/examples
#
########################################################################


MAX_GPU_BATCH_SIZE = 16
EVAL_BATCH_SIZE = 32


def get_dataloaders(accelerator: Accelerator, batch_size: int = 16):
    """
    Creates a set of `DataLoader`s for the `glue` dataset,
    using "bert-base-cased" as the tokenizer.

    Args:
        accelerator (`Accelerator`):
            An `Accelerator` object
        batch_size (`int`, *optional*):
            The batch size for the train and validation DataLoaders.
    """
    tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
    datasets = load_dataset("glue", "mrpc")

    def tokenize_function(examples):
        # max_length=None => use the model max length (it's actually the default)
        outputs = tokenizer(examples["sentence1"], examples["sentence2"], truncation=True, max_length=No
        return outputs

    # Apply the method we just defined to all the examples in all the splits of the dataset
    # starting with the main process first:
    with accelerator.main_process_first():
        tokenized_datasets = datasets.map(
            tokenize_function,
            batched=True,
            remove_columns=["idx", "sentence1", "sentence2"],
        )

    # We also rename the 'label' column to 'labels' which is the expected name for labels by the models
    # transformers library
    tokenized_datasets = tokenized_datasets.rename_column("label", "labels")

    def collate_fn(examples):
        # On TPU it's best to pad everything to the same length or training will be very slow.
        max_length = 128 if accelerator.distributed_type == DistributedType.TPU else None
        # When using mixed precision we want round multiples of 8/16
        if accelerator.mixed_precision == "fp8":
            pad_to_multiple_of = 16
        elif accelerator.mixed_precision != "no":
            pad_to_multiple_of = 8
        else:
            pad_to_multiple_of = None

        return tokenizer.pad(
            examples,
            padding="longest",
            max_length=max_length,
            pad_to_multiple_of=pad_to_multiple_of,
            return_tensors="pt",
        )

    # Instantiate dataloaders.
    train_dataloader = DataLoader(
        tokenized_datasets["train"], shuffle=True, collate_fn=collate_fn, batch_size=batch_size
    )
    eval_dataloader = DataLoader(
        tokenized_datasets["validation"], shuffle=False, collate_fn=collate_fn, batch_size=EVAL_BATCH_SI
    )

    return train_dataloader, eval_dataloader


# For testing only
if os.environ.get("TESTING_MOCKED_DATALOADERS", None) == "1":
    from accelerate.test_utils.training import mocked_dataloaders

    get_dataloaders = mocked_dataloaders  # noqa: F811


def training_function(config, args):
    # For testing only
    if os.environ.get("TESTING_MOCKED_DATALOADERS", None) == "1":
        config["num_epochs"] = 2
    # Initialize accelerator
    accelerator = Accelerator(cpu=args.cpu, mixed_precision=args.mixed_precision)
    # Sample hyper-parameters for learning rate, batch size, seed and a few other HPs
    lr = config["lr"]
    num_epochs = int(config["num_epochs"])
    seed = int(config["seed"])
    batch_size = int(config["batch_size"])

    metric = evaluate.load("glue", "mrpc")

    # If the batch size is too big we use gradient accumulation
```

```
gradient_accumulation_steps = 1
if batch_size > MAX_GPU_BATCH_SIZE and accelerator.distributed_type != DistributedType.TPU:
    gradient_accumulation_steps = batch_size // MAX_GPU_BATCH_SIZE
    batch_size = MAX_GPU_BATCH_SIZE

set_seed(seed)
train_dataloader, eval_dataloader = get_dataloaders(accelerator, batch_size)
# Instantiate the model (we build the model here so that the seed also control new weights initializ
model = AutoModelForSequenceClassification.from_pretrained("bert-base-cased", return_dict=True)

# We could avoid this line since the accelerator is set with `device_placement=True` (default value)
# Note that if you are placing tensors on devices manually, this line absolutely needs to be before
# creation otherwise training will not work on TPU (`accelerate` will kindly throw an error to make
model = model.to(accelerator.device)

# Instantiate optimizer
optimizer = AdamW(params=model.parameters(), lr=lr)

# Instantiate scheduler
lr_scheduler = get_linear_schedule_with_warmup(
    optimizer=optimizer,
    num_warmup_steps=100,
    num_training_steps=(len(train_dataloader) * num_epochs) // gradient_accumulation_steps,
)

# Prepare everything
# There is no specific order to remember, we just need to unpack the objects in the same order we ga
# prepare method.
model, optimizer, train_dataloader, eval_dataloader, lr_scheduler = accelerator.prepare(
    model, optimizer, train_dataloader, eval_dataloader, lr_scheduler
)

# Now we train the model
for epoch in range(num_epochs):
    model.train()
    for step, batch in enumerate(train_dataloader):
        # We could avoid this line since we set the accelerator with `device_placement=True`.
        batch.to(accelerator.device)
        outputs = model(**batch)
        loss = outputs.loss
        loss = loss / gradient_accumulation_steps
        accelerator.backward(loss)
        if step % gradient_accumulation_steps == 0:
            optimizer.step()
            lr_scheduler.step()
            optimizer.zero_grad()

    model.eval()
    samples_seen = 0
    for step, batch in enumerate(eval_dataloader):
        # We could avoid this line since we set the accelerator with `device_placement=True`.
        batch.to(accelerator.device)
        with torch.no_grad():
            outputs = model(**batch)
        predictions = outputs.logits.argmax(dim=-1)
        predictions, references = accelerator.gather((predictions, batch["labels"]))
        # New Code #
        # First we check if it's a distributed system
        if accelerator.use_distributed:
            # Then see if we're on the last batch of our eval dataloader
            if step == len(eval_dataloader) - 1:
                # Last batch needs to be truncated on distributed systems as it contains additional
                predictions = predictions[: len(eval_dataloader.dataset) - samples_seen]
                references = references[: len(eval_dataloader.dataset) - samples_seen]
            else:
                # Otherwise we add the number of samples seen
                samples_seen += references.shape[0]
        # All of this can be avoided if you use `Accelerator.gather_for_metrics` instead of `Acceler
        # accelerator.gather_for_metrics((predictions, batch["labels"]))
        metric.add_batch(
            predictions=predictions,
            references=references,
            )

        eval_metric = metric.compute()
        # Use accelerator.print to print only on the main process.
        accelerator.print(f"epoch {epoch}:", eval_metric)


def main():
    parser = argparse.ArgumentParser(description="Simple example of training script.")
    parser.add_argument(
        "--mixed_precision",
        type=str,
        default=None,
        choices=["no", "fp16", "bf16", "fp8"],
        help="Whether to use mixed precision. Choose"
        "between fp16 and bf16 (bfloat16). Bf16 requires PyTorch >= 1.10."
        "and an Nvidia Ampere GPU.",
    )
    parser.add_argument("--cpu", action="store_true", help="If passed, will train on the CPU.")
    args = parser.parse_args()
    config = {"lr": 2e-5, "num_epochs": 3, "seed": 42, "batch_size": 16}
    training_function(config, args)


if __name__ == "__main__":
    main()
```

accelerate-main/examples/by_feature/automatic_gradient_accumulation.py

```python
# Copyright 2022 The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
import argparse
import os

# New Code #
import evaluate
import torch
from datasets import load_dataset
from torch.optim import AdamW
from torch.utils.data import DataLoader
from transformers import AutoModelForSequenceClassification, AutoTokenizer, get_linear_schedule_with_war

from accelerate import Accelerator
from accelerate.utils import find_executable_batch_size


########################################################################
# This is a fully working simple example to use Accelerate,
# specifically showcasing how to combine both the gradient accumulation
# and automatic batch size finder utilities of Accelerate to perfrom
# automatic gradient accumulation
#
# This example trains a Bert base model on GLUE MRPC
# in any of the following settings (with the same script):
#   - single CPU or single GPU
#   - multi GPUS (using PyTorch distributed mode)
#   - (multi) TPUs
#   - fp16 (mixed-precision) or fp32 (normal precision)
#
# New additions from the base script can be found quickly by
# looking for the # New Code # tags
#
# To run it in each of these various modes, follow the instructions
# in the readme for examples:
# https://github.com/huggingface/accelerate/tree/main/examples
#
########################################################################

EVAL_BATCH_SIZE = 32


def get_dataloaders(accelerator: Accelerator, batch_size: int = 16):
    """
    Creates a set of `DataLoader`s for the `glue` dataset,
    using "bert-base-cased" as the tokenizer.

    Args:
        accelerator (`Accelerator`):
            An `Accelerator` object
        batch_size (`int`, *optional*):
            The batch size for the train and validation DataLoaders.
    """
    tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
    datasets = load_dataset("glue", "mrpc")
    def tokenize_function(examples):
        # max_length=None => use the model max length (it's actually the default)
        outputs = tokenizer(examples["sentence1"], examples["sentence2"], truncation=True, max_length=No
        return outputs

    # Apply the method we just defined to all the examples in all the splits of the dataset
    # starting with the main process first:
    with accelerator.main_process_first():
        tokenized_datasets = datasets.map(
            tokenize_function,
            batched=True,
            remove_columns=["idx", "sentence1", "sentence2"],
        )

    # We also rename the 'label' column to 'labels' which is the expected name for labels by the models
    # transformers library
    tokenized_datasets = tokenized_datasets.rename_column("label", "labels")

    def collate_fn(examples):
        # When using mixed precision we want round multiples of 8/16
        if accelerator.mixed_precision == "fp8":
            pad_to_multiple_of = 16
        elif accelerator.mixed_precision != "no":
            pad_to_multiple_of = 8
        else:
            pad_to_multiple_of = None

        return tokenizer.pad(
            examples,
            padding="longest",
            pad_to_multiple_of=pad_to_multiple_of,
            return_tensors="pt",
        )

    # Instantiate dataloaders.
    train_dataloader = DataLoader(
        tokenized_datasets["train"], shuffle=True, collate_fn=collate_fn, batch_size=batch_size
    )
    eval_dataloader = DataLoader(
        tokenized_datasets["validation"], shuffle=False, collate_fn=collate_fn, batch_size=EVAL_BATCH_SI
    )

    return train_dataloader, eval_dataloader


# For testing only
if os.environ.get("TESTING_MOCKED_DATALOADERS", None) == "1":
    from accelerate.test_utils.training import mocked_dataloaders

    get_dataloaders = mocked_dataloaders  # noqa: F811


def training_function(config, args):
    # For testing only
    if os.environ.get("TESTING_MOCKED_DATALOADERS", None) == "1":
        config["num_epochs"] = 2
    # Initialize accelerator
    accelerator = Accelerator(cpu=args.cpu, mixed_precision=args.mixed_precision)
    # Sample hyper-parameters for learning rate, batch size, seed and a few other HPs
    lr = config["lr"]
    num_epochs = int(config["num_epochs"])
    seed = int(config["seed"])
    observed_batch_size = int(config["batch_size"])

    metric = evaluate.load("glue", "mrpc")

    # New Code #
    # We use the `find_executable_batch_size` decorator, passing in the desired observed batch size
    # to train on. If a CUDA OOM error occurs, it will retry this loop cutting the batch size in
    # half each time. From this, we can calculate the number of gradient accumulation steps needed
    # and modify the Accelerator object as a result
    @find_executable_batch_size(starting_batch_size=int(observed_batch_size))
```

```
def inner_training_loop(batch_size):
    # Since we need to modify the outside accelerator object, we need to bring it
    # to the local scope
    nonlocal accelerator

    # We can calculate the number of gradient accumulation steps based on the current
    # batch size vs the starting batch size
    num_gradient_accumulation_steps = observed_batch_size // batch_size

    # And then set it in the Accelerator directly:
    accelerator.gradient_accumulation_steps = num_gradient_accumulation_steps

    # Next we need to free all of the stored model references in the Accelerator each time
    accelerator.free_memory()

    # And set the seed so our results are reproducable each reset
    set_seed(seed)

    # Instantiate the model (we build the model here so that the seed also control new weights initi
    model = AutoModelForSequenceClassification.from_pretrained("bert-base-cased", return_dict=True)

    # We could avoid this line since the accelerator is set with `device_placement=True` (default va
    # Note that if you are placing tensors on devices manually, this line absolutely needs to be bef
    # creation otherwise training will not work on TPU (`accelerate` will kindly throw an error to m
    model = model.to(accelerator.device)

    # Instantiate optimizer
    optimizer = AdamW(params=model.parameters(), lr=lr)
    train_dataloader, eval_dataloader = get_dataloaders(accelerator, batch_size)

    # Instantiate scheduler
    lr_scheduler = get_linear_schedule_with_warmup(
        optimizer=optimizer,
        num_warmup_steps=100,
        num_training_steps=(len(train_dataloader) * num_epochs),
    )

    # Prepare everything
    # There is no specific order to remember, we just need to unpack the objects in the same order w
    # prepare method.
    model, optimizer, train_dataloader, eval_dataloader, lr_scheduler = accelerator.prepare(
        model, optimizer, train_dataloader, eval_dataloader, lr_scheduler
    )

    # Now we train the model
    for epoch in range(num_epochs):
        model.train()
        for step, batch in enumerate(train_dataloader):
            # And perform gradient accumulation
            with accelerator.accumulate(model):
                # We could avoid this line since we set the accelerator with `device_placement=True`
                batch.to(accelerator.device)
                outputs = model(**batch)
                loss = outputs.loss
                accelerator.backward(loss)
                optimizer.step()
                lr_scheduler.step()
                optimizer.zero_grad()

        model.eval()
        for step, batch in enumerate(eval_dataloader):
            # We could avoid this line since we set the accelerator with `device_placement=True`.
            batch.to(accelerator.device)
            with torch.no_grad():
                outputs = model(**batch)
            predictions = outputs.logits.argmax(dim=-1)
            predictions, references = accelerator.gather_for_metrics((predictions, batch["labels"]))
            metric.add_batch(
                predictions=predictions,
                references=references,
            )
        eval_metric = metric.compute()
```

```
            # Use accelerator.print to print only on the main process.
            accelerator.print(f"epoch {epoch}:", eval_metric)

    # New Code #
    # And call it at the end with no arguments
    # Note: You could also refactor this outside of your training loop function
    inner_training_loop()


def main():
    parser = argparse.ArgumentParser(description="Simple example of training script.")
    parser.add_argument(
        "--mixed_precision",
        type=str,
        default=None,
        choices=["no", "fp16", "bf16", "fp8"],
        help="Whether to use mixed precision. Choose"
        "between fp16 and bf16 (bfloat16). Bf16 requires PyTorch >= 1.10."
        "and an Nvidia Ampere GPU.",
    )
    parser.add_argument("--cpu", action="store_true", help="If passed, will train on the CPU.")
    args = parser.parse_args()
    # New Code #
    # We modify the starting batch size to be an observed batch size of 256, to guarentee an initial CUD
    config = {"lr": 2e-5, "num_epochs": 3, "seed": 42, "batch_size": 256}
    training_function(config, args)


if __name__ == "__main__":
    main()
```

accelerate-main/examples/by_feature/README.md

~~~
# What are these scripts?

All scripts in this folder originate from the `nlp_example.py` file, as it is a very simplistic NLP trai

From there, each further script adds in just **one** feature of Accelerate, showing how you can quickly

A full example with all of these parts integrated together can be found in the `complete_nlp_example.py`

Adjustments to each script from the base `nlp_example.py` file can be found quickly by searching for "#

## Example Scripts by Feature and their Arguments

### Base Example (`../nlp_example.py`)

- Shows how to use `Accelerator` in an extremely simplistic PyTorch training loop
- Arguments available:
  - `mixed_precision`, whether to use mixed precision. ("no", "fp16", or "bf16")
  - `cpu`, whether to train using only the CPU. (yes/no/1/0)

All following scripts also accept these arguments in addition to their added ones.

These arguments should be added at the end of any method for starting the python script (such as `python

```bash
accelerate launch ../nlp_example.py --mixed_precision fp16 --cpu 0
~~~

### Checkpointing and Resuming Training (`checkpointing.py`)

- Shows how to use `Accelerator.save_state` and `Accelerator.load_state` to save or continue training
- **It is assumed you are continuing off the same training script**
- Arguments available:
  - `checkpointing_steps`, after how many steps the various states should be saved. ("epoch", 1, 2, ...)
  - `output_dir`, where saved state folders should be saved to, default is current working directory
  - `resume_from_checkpoint`, what checkpoint folder to resume from. ("epoch_0", "step_22", ...)

These arguments should be added at the end of any method for starting the python script (such as `python

(Note, `resume_from_checkpoint` assumes that we've ran the script for one epoch with the `--checkpointin

```bash
accelerate launch ./checkpointing.py --checkpointing_steps epoch output_dir "checkpointing_tutorial" --r
```

### Cross Validation (`cross_validation.py`)

- Shows how to use `Accelerator.free_memory` and run cross validation efficiently with `datasets`.
- Arguments available:
  - `num_folds`, the number of folds the training dataset should be split into.

These arguments should be added at the end of any method for starting the python script (such as `python

```bash
accelerate launch ./cross_validation.py --num_folds 2
```

### Experiment Tracking (`tracking.py`)

- Shows how to use `Accelerate.init_trackers` and `Accelerator.log`
- Can be used with Weights and Biases, TensorBoard, or CometML.
- Arguments available:
  - `with_tracking`, whether to load in all available experiment trackers from the environment.

These arguments should be added at the end of any method for starting the python script (such as `python

```bash
accelerate launch ./tracking.py --with_tracking
```
```
```

### Gradient Accumulation (`gradient_accumulation.py`)

- Shows how to use `Accelerator.no_sync` to prevent gradient averaging in a distributed setup.
- Arguments available:
  - `gradient_accumulation_steps`, the number of steps to perform before the gradients are accumulated a

These arguments should be added at the end of any method for starting the python script (such as `python

```bash
accelerate launch ./gradient_accumulation.py --gradient_accumulation_steps 5
```

accelerate-main/examples/by_feature/deepspeed_with_config_support.py

```
#!/usr/bin/env python
# coding=utf-8
# Copyright 2022 The HuggingFace Inc. team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
"""
Fine-tuning the library models for causal language modeling (GPT, GPT-2, CTRL, ...)
on a text file or a dataset without using HuggingFace Trainer.

Here is the full list of checkpoints on the hub that can be fine-tuned by this script:
https://huggingface.co/models?filter=text-generation
"""
# You can also adapt this script on your own causal language modeling task. Pointers for this are left a

import argparse
import json
import logging
import math
import os
import random
from itertools import chain
from pathlib import Path

import datasets
import torch
import transformers
from datasets import load_dataset
from huggingface_hub import Repository
from torch.utils.data import DataLoader
from tqdm.auto import tqdm
from transformers import (
    CONFIG_MAPPING,
    MODEL_MAPPING,
    AutoConfig,
    AutoModelForCausalLM,
    AutoTokenizer,
    SchedulerType,
    default_data_collator,
    get_scheduler,
)
from transformers.utils import get_full_repo_name
from transformers.utils.versions import require_version

from accelerate import Accelerator, DistributedType
from accelerate.logging import get_logger
from accelerate.utils import DummyOptim, DummyScheduler, set_seed


logger = get_logger(__name__)

require_version("datasets>=1.8.0", "To fix: pip install -r examples/pytorch/language-modeling/requiremen

MODEL_CONFIG_CLASSES = list(MODEL_MAPPING.keys())
MODEL_TYPES = tuple(conf.model_type for conf in MODEL_CONFIG_CLASSES)
def parse_args():
    parser = argparse.ArgumentParser(description="Finetune a transformers model on a causal language mod
    parser.add_argument(
        "--dataset_name",
        type=str,
        default=None,
        help="The name of the dataset to use (via the datasets library).",
    )
    parser.add_argument(
        "--dataset_config_name",
        type=str,
        default=None,
        help="The configuration name of the dataset to use (via the datasets library).",
    )
    parser.add_argument(
        "--train_file", type=str, default=None, help="A csv or a json file containing the training data.
    )
    parser.add_argument(
        "--validation_file", type=str, default=None, help="A csv or a json file containing the validatio
    )
    parser.add_argument(
        "--validation_split_percentage",
        default=5,
        help="The percentage of the train set used as validation set in case there's no validation split
    )
    parser.add_argument(
        "--model_name_or_path",
        type=str,
        help="Path to pretrained model or model identifier from huggingface.co/models.",
        required=False,
    )
    parser.add_argument(
        "--config_name",
        type=str,
        default=None,
        help="Pretrained config name or path if not the same as model_name",
    )
    parser.add_argument(
        "--tokenizer_name",
        type=str,
        default=None,
        help="Pretrained tokenizer name or path if not the same as model_name",
    )
    parser.add_argument(
        "--use_slow_tokenizer",
        action="store_true",
        help="If passed, will use a slow tokenizer (not backed by the ■ Tokenizers library).",
    )
    parser.add_argument(
        "--per_device_train_batch_size",
        type=int,
        default=8,
        help="Batch size (per device) for the training dataloader.",
    )
    parser.add_argument(
        "--per_device_eval_batch_size",
        type=int,
        default=8,
        help="Batch size (per device) for the evaluation dataloader.",
    )
    parser.add_argument(
        "--learning_rate",
        type=float,
        default=5e-5,
        help="Initial learning rate (after the potential warmup period) to use.",
    )
    parser.add_argument("--weight_decay", type=float, default=0.0, help="Weight decay to use.")
    parser.add_argument("--num_train_epochs", type=int, default=3, help="Total number of training epochs
    parser.add_argument(
        "--max_train_steps",
        type=int,
        default=None,
        help="Total number of training steps to perform. If provided, overrides num_train_epochs.",
    )
```

```
parser.add_argument(
    "--gradient_accumulation_steps",
    type=int,
    default=1,
    help="Number of updates steps to accumulate before performing a backward/update pass.",
)
parser.add_argument(
    "--lr_scheduler_type",
    type=SchedulerType,
    default="linear",
    help="The scheduler type to use.",
    choices=["linear", "cosine", "cosine_with_restarts", "polynomial", "constant", "constant_with_wa
)
parser.add_argument(
    "--num_warmup_steps", type=int, default=0, help="Number of steps for the warmup in the lr schedu
)
parser.add_argument("--output_dir", type=str, default=None, help="Where to store the final model.")
parser.add_argument("--seed", type=int, default=None, help="A seed for reproducible training.")
parser.add_argument(
    "--model_type",
    type=str,
    default=None,
    help="Model type to use if training from scratch.",
    choices=MODEL_TYPES,
)
parser.add_argument(
    "--block_size",
    type=int,
    default=None,
    help=(
        "Optional input sequence length after tokenization. The training dataset will be truncated i
        " this size for training. Default to the model max input length for single sentence inputs (
        " account special tokens)."
    ),
)
parser.add_argument(
    "--preprocessing_num_workers",
    type=int,
    default=None,
    help="The number of processes to use for the preprocessing.",
)
parser.add_argument(
    "--overwrite_cache", type=bool, default=False, help="Overwrite the cached training and evaluatio
)
parser.add_argument(
    "--no_keep_linebreaks", action="store_true", help="Do not keep line breaks when using TXT files.
)
parser.add_argument("--push_to_hub", action="store_true", help="Whether or not to push the model to
parser.add_argument(
    "--hub_model_id", type=str, help="The name of the repository to keep in sync with the local `out
)
parser.add_argument("--hub_token", type=str, help="The token to use to push to the Model Hub.")
parser.add_argument(
    "--checkpointing_steps",
    type=str,
    default=None,
    help="Whether the various states should be saved at the end of every n steps, or 'epoch' for eac
)
parser.add_argument(
    "--resume_from_checkpoint",
    type=str,
    default=None,
    help="If the training should continue from a checkpoint folder.",
)
# New Code #
# Whether to load the best model at the end of training
parser.add_argument(
    "--load_best_model",
    action="store_true",
    help="Whether to load the best model at the end of training",
)
parser.add_argument(
        "--with_tracking",
        action="store_true",
        help="Whether to enable experiment trackers for logging.",
    )
    parser.add_argument(
        "--report_to",
        type=str,
        default="all",
        help=(
            'The integration to report the results and logs to. Supported platforms are `"tensorboard"`,
            ' `"wandb"` and `"comet_ml"`. Use `"all"` (default) to report to all integrations.'
            "Only applicable when `--with_tracking` is passed."
        ),
    )
    args = parser.parse_args()

    # Sanity checks
    if args.dataset_name is None and args.train_file is None and args.validation_file is None:
        raise ValueError("Need either a dataset name or a training/validation file.")
    else:
        if args.train_file is not None:
            extension = args.train_file.split(".")[-1]
            assert extension in ["csv", "json", "txt"], "`train_file` should be a csv, json or txt file.
        if args.validation_file is not None:
            extension = args.validation_file.split(".")[-1]
            assert extension in ["csv", "json", "txt"], "`validation_file` should be a csv, json or txt

    if args.push_to_hub:
        assert args.output_dir is not None, "Need an `output_dir` to create a repo when `--push_to_hub`

    return args


# New Code #
def checkpoint_model(checkpoint_folder, ckpt_id, model, epoch, last_global_step, **kwargs):
    """Utility function for checkpointing model + optimizer dictionaries
    The main purpose for this is to be able to resume training from that instant again
    """
    checkpoint_state_dict = {
        "epoch": epoch,
        "last_global_step": last_global_step,
    }
    # Add extra kwargs too
    checkpoint_state_dict.update(kwargs)

    success = model.save_checkpoint(checkpoint_folder, ckpt_id, checkpoint_state_dict)
    status_msg = f"checkpointing: checkpoint_folder={checkpoint_folder}, ckpt_id={ckpt_id}"
    if success:
        logging.info(f"Success {status_msg}")
    else:
        logging.warning(f"Failure {status_msg}")
    return


# New Code #
def load_training_checkpoint(model, load_dir, tag=None, **kwargs):
    """Utility function for checkpointing model + optimizer dictionaries
    The main purpose for this is to be able to resume training from that instant again
    """
    _, checkpoint_state_dict = model.load_checkpoint(load_dir, tag=tag, **kwargs)
    epoch = checkpoint_state_dict["epoch"]
    last_global_step = checkpoint_state_dict["last_global_step"]
    del checkpoint_state_dict
    return (epoch, last_global_step)


# New Code #
def evaluate(args, model, eval_dataloader, accelerator, eval_dataset):
    model.eval()
    losses = []
    for step, batch in enumerate(eval_dataloader):
        with torch.no_grad():
            outputs = model(**batch)

        loss = outputs.loss
        losses.append(accelerator.gather_for_metrics(loss.repeat(args.per_device_eval_batch_size)))

    losses = torch.cat(losses)
    try:
        eval_loss = torch.mean(losses)
        perplexity = math.exp(eval_loss)
    except OverflowError:
        perplexity = float("inf")
    return perplexity, eval_loss


def main():
    args = parse_args()

    # Initialize the accelerator. We will let the accelerator handle device placement for us in this exa
    # If we're using tracking, we also need to initialize it here and it will by default pick up all sup
    # in the environment
    accelerator = (
        Accelerator(log_with=args.report_to, logging_dir=args.output_dir) if args.with_tracking else Acc
    )
    # Make one log on every process with the configuration for debugging.
    logging.basicConfig(
        format="%(asctime)s - %(levelname)s - %(name)s - %(message)s",
        datefmt="%m/%d/%Y %H:%M:%S",
        level=logging.INFO,
    )
    logger.info(accelerator.state, main_process_only=False)
    if accelerator.is_local_main_process:
        datasets.utils.logging.set_verbosity_warning()
        transformers.utils.logging.set_verbosity_info()
    else:
        datasets.utils.logging.set_verbosity_error()
        transformers.utils.logging.set_verbosity_error()

    # If passed along, set the training seed now.
    if args.seed is not None:
        set_seed(args.seed)

    # Handle the repository creation
    if accelerator.is_main_process:
        if args.push_to_hub:
            if args.hub_model_id is None:
                repo_name = get_full_repo_name(Path(args.output_dir).name, token=args.hub_token)
            else:
                repo_name = args.hub_model_id
            repo = Repository(args.output_dir, clone_from=repo_name)

            with open(os.path.join(args.output_dir, ".gitignore"), "w+") as gitignore:
                if "step_*" not in gitignore:
                    gitignore.write("step_*\n")
                if "epoch_*" not in gitignore:
                    gitignore.write("epoch_*\n")
        elif args.output_dir is not None:
            os.makedirs(args.output_dir, exist_ok=True)
    accelerator.wait_for_everyone()

    # Get the datasets: you can either provide your own CSV/JSON/TXT training and evaluation files (see
    # or just provide the name of one of the public datasets available on the hub at https://huggingface
    # (the dataset will be downloaded automatically from the datasets Hub).
    #
    # For CSV/JSON files, this script will use the column called 'text' or the first column if no column
    # 'text' is found. You can easily tweak this behavior (see below).
    #
    # In distributed training, the load_dataset function guarantee that only one local process can concu
    # download the dataset.
    if args.dataset_name is not None:
        # Downloading and loading a dataset from the hub.
        raw_datasets = load_dataset(args.dataset_name, args.dataset_config_name)
        if "validation" not in raw_datasets.keys():
```

```
        raw_datasets["validation"] = load_dataset(
            args.dataset_name,
            args.dataset_config_name,
            split=f"train[:{args.validation_split_percentage}%]",
        )
        raw_datasets["train"] = load_dataset(
            args.dataset_name,
            args.dataset_config_name,
            split=f"train[{args.validation_split_percentage}%:]",
        )
else:
    data_files = {}
    dataset_args = {}
    if args.train_file is not None:
        data_files["train"] = args.train_file
    if args.validation_file is not None:
        data_files["validation"] = args.validation_file
    extension = args.train_file.split(".")[-1]
    if extension == "txt":
        extension = "text"
        dataset_args["keep_linebreaks"] = not args.no_keep_linebreaks
    raw_datasets = load_dataset(extension, data_files=data_files, **dataset_args)
    # If no validation data is there, validation_split_percentage will be used to divide the dataset
    if "validation" not in raw_datasets.keys():
        raw_datasets["validation"] = load_dataset(
            extension,
            data_files=data_files,
            split=f"train[:{args.validation_split_percentage}%]",
            **dataset_args,
        )
        raw_datasets["train"] = load_dataset(
            extension,
            data_files=data_files,
            split=f"train[{args.validation_split_percentage}%:]",
            **dataset_args,
        )

# See more about loading any type of standard or custom dataset (from files, python dict, pandas Dat
# https://huggingface.co/docs/datasets/loading_datasets.html.

# Load pretrained model and tokenizer
#
# In distributed training, the .from_pretrained methods guarantee that only one local process can co
# download model & vocab.
if args.config_name:
    config = AutoConfig.from_pretrained(args.config_name)
elif args.model_name_or_path:
    config = AutoConfig.from_pretrained(args.model_name_or_path)
else:
    config = CONFIG_MAPPING[args.model_type]()
    logger.warning("You are instantiating a new config instance from scratch.")

if args.tokenizer_name:
    tokenizer = AutoTokenizer.from_pretrained(args.tokenizer_name, use_fast=not args.use_slow_tokeni
elif args.model_name_or_path:
    tokenizer = AutoTokenizer.from_pretrained(args.model_name_or_path, use_fast=not args.use_slow_to
else:
    raise ValueError(
        "You are instantiating a new tokenizer from scratch. This is not supported by this script."
        "You can do it from another script, save it, and load it from here, using --tokenizer_name."
    )

if args.model_name_or_path:
    model = AutoModelForCausalLM.from_pretrained(
        args.model_name_or_path,
        from_tf=bool(".ckpt" in args.model_name_or_path),
        config=config,
    )
else:
    logger.info("Training new model from scratch")
    model = AutoModelForCausalLM.from_config(config)
model.resize_token_embeddings(len(tokenizer))
```

```
# Preprocessing the datasets.
# First we tokenize all the texts.
column_names = raw_datasets["train"].column_names
text_column_name = "text" if "text" in column_names else column_names[0]

def tokenize_function(examples):
    return tokenizer(examples[text_column_name])

with accelerator.main_process_first():
    tokenized_datasets = raw_datasets.map(
        tokenize_function,
        batched=True,
        num_proc=args.preprocessing_num_workers,
        remove_columns=column_names,
        load_from_cache_file=not args.overwrite_cache,
        desc="Running tokenizer on dataset",
    )

if args.block_size is None:
    block_size = tokenizer.model_max_length
    if block_size > 1024:
        logger.warning(
            f"The tokenizer picked seems to have a very large `model_max_length` ({tokenizer.model_m
            "Picking 1024 instead. You can change that default value by passing --block_size xxx."
        )
    block_size = 1024
else:
    if args.block_size > tokenizer.model_max_length:
        logger.warning(
            f"The block_size passed ({args.block_size}) is larger than the maximum length for the mo
            f"({tokenizer.model_max_length}). Using block_size={tokenizer.model_max_length}."
        )
    block_size = min(args.block_size, tokenizer.model_max_length)

# Main data processing function that will concatenate all texts from our dataset and generate chunks
def group_texts(examples):
    # Concatenate all texts.
    concatenated_examples = {k: list(chain(*examples[k])) for k in examples.keys()}
    total_length = len(concatenated_examples[list(examples.keys())[0]])
    # We drop the small remainder, we could add padding if the model supported it instead of this dr
    # customize this part to your needs.
    if total_length >= block_size:
        total_length = (total_length // block_size) * block_size
    # Split by chunks of max_len.
    result = {
        k: [t[i : i + block_size] for i in range(0, total_length, block_size)]
        for k, t in concatenated_examples.items()
    }
    result["labels"] = result["input_ids"].copy()
    return result

# Note that with `batched=True`, this map processes 1,000 texts together, so group_texts throws away
# for each of those groups of 1,000 texts. You can adjust that batch_size here but a higher value mi
# to preprocess.
#
# To speed up this part, we use multiprocessing. See the documentation of the map method for more in
# https://huggingface.co/docs/datasets/package_reference/main_classes.html#datasets.Dataset.map

with accelerator.main_process_first():
    lm_datasets = tokenized_datasets.map(
        group_texts,
        batched=True,
        num_proc=args.preprocessing_num_workers,
        load_from_cache_file=not args.overwrite_cache,
        desc=f"Grouping texts in chunks of {block_size}",
    )

train_dataset = lm_datasets["train"]
eval_dataset = lm_datasets["validation"]

# Log a few random samples from the training set:
for index in random.sample(range(len(train_dataset)), 3):
    logger.info(f"Sample {index} of the training set: {train_dataset[index]}.")

# DataLoaders creation:
train_dataloader = DataLoader(
    train_dataset, shuffle=True, collate_fn=default_data_collator, batch_size=args.per_device_train_
)
eval_dataloader = DataLoader(
    eval_dataset, collate_fn=default_data_collator, batch_size=args.per_device_eval_batch_size
)

# Optimizer
# Split weights in two groups, one with weight decay and the other not.
no_decay = ["bias", "LayerNorm.weight"]
optimizer_grouped_parameters = [
    {
        "params": [p for n, p in model.named_parameters() if not any(nd in n for nd in no_decay)],
        "weight_decay": args.weight_decay,
    },
    {
        "params": [p for n, p in model.named_parameters() if any(nd in n for nd in no_decay)],
        "weight_decay": 0.0,
    },
]
# New Code #
# Creates Dummy Optimizer if `optimizer` was specified in the config file else creates Adam Optimize
optimizer_cls = (
    torch.optim.AdamW
    if accelerator.state.deepspeed_plugin is None
    or "optimizer" not in accelerator.state.deepspeed_plugin.deepspeed_config
    else DummyOptim
)
optimizer = optimizer_cls(optimizer_grouped_parameters, lr=args.learning_rate)

# On TPU, the tie weights in our model have been disconnected, so we need to restore the ties.
if accelerator.distributed_type == DistributedType.TPU:
    model.tie_weights()

# Scheduler and math around the number of training steps.

# New Code
# Get gradient accumulation steps from deepspeed config if available
if accelerator.state.deepspeed_plugin is not None:
    args.gradient_accumulation_steps = accelerator.state.deepspeed_plugin.deepspeed_config[
        "gradient_accumulation_steps"
    ]

num_update_steps_per_epoch = math.ceil(len(train_dataloader) / args.gradient_accumulation_steps)
if args.max_train_steps is None:
    args.max_train_steps = args.num_train_epochs * num_update_steps_per_epoch
else:
    args.num_train_epochs = math.ceil(args.max_train_steps / num_update_steps_per_epoch)

# New Code #
# Creates Dummy Scheduler if `scheduler` was specified in the config file else creates `args.lr_sche
if (
    accelerator.state.deepspeed_plugin is None
    or "scheduler" not in accelerator.state.deepspeed_plugin.deepspeed_config
):
    lr_scheduler = get_scheduler(
        name=args.lr_scheduler_type,
        optimizer=optimizer,
        num_warmup_steps=args.num_warmup_steps,
        num_training_steps=args.max_train_steps,
    )
else:
    lr_scheduler = DummyScheduler(
        optimizer, total_num_steps=args.max_train_steps, warmup_num_steps=args.num_warmup_steps
    )

# Prepare everything with our `accelerator`.
model, optimizer, train_dataloader, eval_dataloader, lr_scheduler = accelerator.prepare(
    model, optimizer, train_dataloader, eval_dataloader, lr_scheduler
```

```
)

# We need to recalculate our total training steps as the size of the training dataloader may have ch
num_update_steps_per_epoch = math.ceil(len(train_dataloader) / args.gradient_accumulation_steps)
args.max_train_steps = args.num_train_epochs * num_update_steps_per_epoch

# Figure out how many steps we should save the Accelerator states
if hasattr(args.checkpointing_steps, "isdigit"):
    checkpointing_steps = args.checkpointing_steps
    if args.checkpointing_steps.isdigit():
        checkpointing_steps = int(args.checkpointing_steps)
else:
    checkpointing_steps = None

# We need to initialize the trackers we use, and also store our configuration.
# The trackers initializes automatically on the main process.
if args.with_tracking:
    experiment_config = vars(args)
    # TensorBoard cannot log Enums, need the raw value
    experiment_config["lr_scheduler_type"] = experiment_config["lr_scheduler_type"].value
    accelerator.init_trackers("clm_no_trainer", experiment_config)

# Train!
total_batch_size = args.per_device_train_batch_size * accelerator.num_processes * args.gradient_accu

logger.info("***** Running training *****")
logger.info(f"  Num examples = {len(train_dataset)}")
logger.info(f"  Num Epochs = {args.num_train_epochs}")
logger.info(f"  Instantaneous batch size per device = {args.per_device_train_batch_size}")
logger.info(f"  Total train batch size (w. parallel, distributed & accumulation) = {total_batch_size
logger.info(f"  Gradient Accumulation steps = {args.gradient_accumulation_steps}")
logger.info(f"  Total optimization steps = {args.max_train_steps}")
# Only show the progress bar once on each machine.
progress_bar = tqdm(range(args.max_train_steps), disable=not accelerator.is_local_main_process)
completed_steps = 0
starting_epoch = 0
best_metric = None
best_metric_checkpoint = None

# Potentially load in the weights and states from a previous save
if args.resume_from_checkpoint:
    # New Code #
    # Loads the DeepSpeed checkpoint from the specified path
    _, last_global_step = load_training_checkpoint(
        model,
        args.resume_from_checkpoint,
        **{"load_optimizer_states": True, "load_lr_scheduler_states": True},
    )
    accelerator.print(f"Resumed from checkpoint: {args.resume_from_checkpoint}")
    resume_step = last_global_step
    starting_epoch = resume_step // len(train_dataloader)
    resume_step -= starting_epoch * len(train_dataloader)

for epoch in range(starting_epoch, args.num_train_epochs):
    model.train()
    if args.with_tracking:
        total_loss = 0
    for step, batch in enumerate(train_dataloader):
        # We need to skip steps until we reach the resumed step
        if args.resume_from_checkpoint and epoch == starting_epoch:
            if resume_step is not None and step < resume_step:
                completed_steps += 1
                continue
        outputs = model(**batch)
        loss = outputs.loss
        # We keep track of the loss at each epoch
        if args.with_tracking:
            total_loss += loss.detach().float()
        loss = loss / args.gradient_accumulation_steps
        accelerator.backward(loss)
        if (step + 1) % args.gradient_accumulation_steps == 0 or step == len(train_dataloader) - 1:
            optimizer.step()
```

```
                lr_scheduler.step()
                optimizer.zero_grad()
                progress_bar.update(1)
                completed_steps += 1

            if isinstance(checkpointing_steps, int):
                if completed_steps % checkpointing_steps == 0:
                    output_dir = f"step_{completed_steps }"
                    if args.output_dir is not None:
                        output_dir = os.path.join(args.output_dir, output_dir)
                    accelerator.save_state(output_dir)
            if completed_steps >= args.max_train_steps:
                break

        perplexity, eval_loss = evaluate(args, model, eval_dataloader, accelerator, eval_dataset)
        logger.info(f"epoch {epoch}: perplexity: {perplexity} eval_loss: {eval_loss}")

        if args.with_tracking:
            accelerator.log(
                {
                    "perplexity": perplexity,
                    "eval_loss": eval_loss,
                    "train_loss": total_loss.item() / len(train_dataloader),
                    "epoch": epoch,
                    "step": completed_steps,
                },
                step=completed_steps,
            )

        # New Code #
        # Save the DeepSpeed checkpoint to the specified path
        checkpoint_model(args.output_dir, epoch, model, epoch, completed_steps)

        # New Code #
        # Tracks the best checkpoint and best metric
        if best_metric is None or best_metric > perplexity:
            best_metric = perplexity
            best_metric_checkpoint = os.path.join(args.output_dir, str(epoch))
            accelerator.print(f"New best metric: {best_metric} at epoch {epoch}")
            accelerator.print(f"best_metric_checkpoint: {best_metric_checkpoint}")

    # New Code #
    # Loads the best checkpoint after the training is finished
    if args.load_best_model:
        _, last_global_step = load_training_checkpoint(
            model,
            "/".join(best_metric_checkpoint.split("/")[:-1]),
            tag=best_metric_checkpoint.split("/")[-1],
            **{"load_optimizer_states": True, "load_lr_scheduler_states": True},
        )

    # New Code #
    # Evaluates using the best checkpoint
    perplexity, eval_loss = evaluate(args, model, eval_dataloader, accelerator, eval_dataset)
    logger.info(f"Best model metrics: perplexity: {perplexity} eval_loss: {eval_loss}")
    if perplexity != best_metric:
        raise AssertionError(
            f"Best metric {best_metric} does not match the metric {perplexity} of the loaded best model.
        )

    if args.output_dir is not None:
        accelerator.wait_for_everyone()
        unwrapped_model = accelerator.unwrap_model(model)

        # New Code #
        # Saves the whole/unpartitioned fp16 model when in ZeRO Stage-3 to the output directory if
        # `stage3_gather_16bit_weights_on_model_save` is True in DeepSpeed Config file or
        # `zero3_save_16bit_model` is True in DeepSpeed Plugin.
        # For Zero Stages 1 and 2, models are saved as usual in the output directory.
        # The model name saved is `pytorch_model.bin`
        unwrapped_model.save_pretrained(
            args.output_dir,
            is_main_process=accelerator.is_main_process,
            save_function=accelerator.save,
            state_dict=accelerator.get_state_dict(model),
        )
        if accelerator.is_main_process:
            tokenizer.save_pretrained(args.output_dir)
            if args.push_to_hub:
                repo.push_to_hub(commit_message="End of training", auto_lfs_prune=True)

        with open(os.path.join(args.output_dir, "all_results.json"), "w") as f:
            json.dump({"perplexity": perplexity, "eval_loss": eval_loss.item()}, f)


if __name__ == "__main__":
    main()
```