mosec

Release latest

mosec maintainers

Jan 01, 2024

USER GUIDE

CHAPTER

ONE

INTRODUCTION

Mosec is a high-performance and flexible model serving framework for building ML model-enabled backend and microservices. It bridges the gap between any machine learning models you just trained and the efficient online service API.

- **Highly performant**: web layer and task coordination built with Rust , which offers blazing speed in addition to efficient CPU utilization powered by async I/O
- **Ease of use**: user interface purely in Python , by which users can serve their models in an ML framework-agnostic manner using the same code as they do for offline testing
- **Dynamic batching**: aggregate requests from different users for batched inference and distribute results back
- **Pipelined stages**: spawn multiple processes for pipelined stages to handle CPU/GPU/IO mixed workloads
- **Cloud friendly**: designed to run in the cloud, with the model warmup, graceful shutdown, and Prometheus monitoring metrics, easily managed by Kubernetes or any container orchestration systems
- **Do one thing well**: focus on the online serving part, users can pay attention to the model optimization and business logic

CHAPTER TWO

INSTALLATION

Mosec requires Python 3.7 or above. Install the latest PyPI package for Linux x86_64 or macOS x86_64 with:

```
pip install -U mosec
```

To build from the source code, install Rust and run the following command:

```
make package
```

You will get a mosec wheel file in the `dist` folder.

CHAPTER

THREE

USAGE

We demonstrate how Mosec can help you easily host a pre-trained stable diffusion model as a service. You need to install diffusers and transformers as prerequisites:

```
pip install --upgrade diffusers[torch] transformers
```

3.1 Write the server

Firstly, we import the libraries and set up a basic logger to better observe what happens.

```python
from io import BytesIO
from typing import List

import torch  # type: ignore
from diffusers import StableDiffusionPipeline  # type: ignore

from mosec import Server, Worker, get_logger
from mosec.mixin import MsgpackMixin

logger = get_logger()
```

Then, we **build an API** for clients to query a text prompt and obtain an image based on the stable-diffusion-v1-5 model in just 3 steps.

1) Define your service as a class which inherits `mosec.Worker`. Here we also inherit `MsgpackMixin` to employ the msgpack serialization format(a).
2) Inside the `__init__` method, initialize your model and put it onto the corresponding device. Optionally you can assign `self.example` with some data to warm up(b) the model. Note that the data should be compatible with your handler's input format, which we detail next.
3) Override the `forward` method to write your service handler(c), with the signature `forward(self, data: Any | List[Any]) -> Any | List[Any]`. Receiving/returning a single item or a tuple depends on whether *dynamic batching*(d) is configured.

```python
class StableDiffusion(MsgpackMixin, Worker):
    def __init__(self):
        self.pipe = StableDiffusionPipeline.from_pretrained(
            "runwayml/stable-diffusion-v1-5", torch_dtype=torch.float16
        )
        device = "cuda" if torch.cuda.is_available() else "cpu"
```

(continues on next page)

(continued from previous page)

```
        self.pipe = self.pipe.to(device)
        self.example = ["useless example prompt"] * 4  # warmup (batch_size=4)

    def forward(self, data: List[str]) -> List[memoryview]:
        logger.debug("generate images for %s", data)
        res = self.pipe(data)
        logger.debug("NSFW: %s", res[1])
        images = []
        for img in res[0]:
            dummy_file = BytesIO()
            img.save(dummy_file, format="JPEG")
            images.append(dummy_file.getbuffer())
        return images
```

[!NOTE]

(a) In this example we return an image in the binary format, which JSON does not support (unless encoded with base64 that makes the payload larger). Hence, msgpack suits our need better. If we do not inherit `MsgpackMixin`, JSON will be used by default. In other words, the protocol of the service request/response can be either msgpack, JSON, or any other format (check our mixins).

(b) Warm-up usually helps to allocate GPU memory in advance. If the warm-up example is specified, the service will only be ready after the example is forwarded through the handler. However, if no example is given, the first request's latency is expected to be longer. The `example` should be set as a single item or a tuple depending on what `forward` expects to receive. Moreover, in the case where you want to warm up with multiple different examples, you may set `multi_examples` (demo here).

(c) This example shows a single-stage service, where the `StableDiffusion` worker directly takes in client's prompt request and responds the image. Thus the `forward` can be considered as a complete service handler. However, we can also design a multi-stage service with workers doing different jobs (e.g., downloading images, model inference, post-processing) in a pipeline. In this case, the whole pipeline is considered as the service handler, with the first worker taking in the request and the last worker sending out the response. The data flow between workers is done by inter-process communication.

(d) Since dynamic batching is enabled in this example, the `forward` method will wishfully receive a *list* of string, e.g., `['a cute cat playing with a red ball', 'a man sitting in front of a computer', ...]`, aggregated from different clients for *batch inference*, improving the system throughput.

Finally, we append the worker to the server to construct a *single-stage* workflow (multiple stages can be pipelined to further boost the throughput, see this example), and specify the number of processes we want it to run in parallel (`num=1`), and the maximum batch size (`max_batch_size=4`, the maximum number of requests dynamic batching will accumulate before timeout; timeout is defined with the `max_wait_time=10` in milliseconds, meaning the longest time Mosec waits until sending the batch to the Worker).

```
if __name__ == "__main__":
    server = Server()
    # 1) `num` specifies the number of processes that will be spawned to run in parallel.
    # 2) By configuring the `max_batch_size` with the value > 1, the input data in your
    # `forward` function will be a list (batch); otherwise, it's a single item.
    server.append_worker(StableDiffusion, num=1, max_batch_size=4, max_wait_time=10)
    server.run()
```

3.2 Run the server

The above snippets are merged in our example file. You may directly run at the project root level. We first have a look at the *command line arguments* (explanations here):

```
python examples/stable_diffusion/server.py --help
```

Then let's start the server with debug logs:

```
python examples/stable_diffusion/server.py --log-level debug --timeout 30000
```

Open `http://127.0.0.1:8000/openapi/swagger/` in your browser to get the OpenAPI doc.

And in another terminal, test it:

```
python examples/stable_diffusion/client.py --prompt "a cute cat playing with a red ball"␣
↪--output cat.jpg --port 8000
```

You will get an image named "cat.jpg" in the current directory.

You can check the metrics:

```
curl http://127.0.0.1:8000/metrics
```

That's it! You have just hosted your ***stable-diffusion model*** as a service!

CHAPTER

FOUR

EXAMPLES

More ready-to-use examples can be found in the Example section. It includes:

- Pipeline: a simple echo demo even without any ML model.
- Request validation: validate the request with type annotation.
- Multiple route: serve multiple models in one service
- Embedding service: OpenAI compatible embedding service
- Shared memory IPC: inter-process communication with shared memory.
- Customized GPU allocation: deploy multiple replicas, each using different GPUs.
- Customized metrics: record your own metrics for monitoring.
- Jax jitted inference: just-in-time compilation speeds up the inference.
- PyTorch deep learning models:
  - sentiment analysis: infer the sentiment of a sentence.
  - image recognition: categorize a given image.
  - stable diffusion: generate images based on texts, with msgpack serialization.

CHAPTER

FIVE

CONFIGURATION

- Dynamic batching
  - `max_batch_size` and `max_wait_time (millisecond)` are configured when you call `append_worker`.
  - Make sure inference with the `max_batch_size` value won't cause the out-of-memory in GPU.
  - Normally, `max_wait_time` should be less than the batch inference time.
  - If enabled, it will collect a batch either when the number of accumulated requests reaches `max_batch_size` or when `max_wait_time` has elapsed. The service will benefit from this feature when the traffic is high.
- Check the arguments doc for other configurations.

CHAPTER

SIX

DEPLOYMENT

- If you're looking for a GPU base image with `mosec` installed, you can check the official image `mosecorg/mosec`. For the complex use case, check out envd.
- This service doesn't need Gunicorn or NGINX, but you can certainly use the ingress controller when necessary.
- This service should be the PID 1 process in the container since it controls multiple processes. If you need to run multiple processes in one container, you will need a supervisor. You may choose Supervisor or Horust.
- Remember to collect the **metrics**.
  - `mosec_service_batch_size_bucket` shows the batch size distribution.
  - `mosec_service_batch_duration_second_bucket` shows the duration of dynamic batching for each connection in each stage (starts from receiving the first task).
  - `mosec_service_process_duration_second_bucket` shows the duration of processing for each connection in each stage (including the IPC time but excluding the `mosec_service_batch_duration_second_bucket`).
  - `mosec_service_remaining_task` shows the number of currently processing tasks.
  - `mosec_service_throughput` shows the service throughput.
- Stop the service with `SIGINT` (`CTRL+C`) or `SIGTERM` (`kill {PID}`) since it has the graceful shutdown logic.

CHAPTER SEVEN

PERFORMANCE TUNING

- Find out the best `max_batch_size` and `max_wait_time` for your inference service. The metrics will show the histograms of the real batch size and batch duration. Those are the key information to adjust these two parameters.
- Try to split the whole inference process into separate CPU and GPU stages (ref DistilBERT). Different stages will be run in a data pipeline, which will keep the GPU busy.
- You can also adjust the number of workers in each stage. For example, if your pipeline consists of a CPU stage for preprocessing and a GPU stage for model inference, increasing the number of CPU-stage workers can help to produce more data to be batched for model inference at the GPU stage; increasing the GPU-stage workers can fully utilize the GPU memory and computation power. Both ways may contribute to higher GPU utilization, which consequently results in higher service throughput.
- For multi-stage services, note that the data passing through different stages will be serialized/deserialized by the `serialize_ipc/deserialize_ipc` methods, so extremely large data might make the whole pipeline slow. The serialized data is passed to the next stage through rust by default, you could enable shared memory to potentially reduce the latency (ref RedisShmIPCMixin).
- You should choose appropriate `serialize/deserialize` methods, which are used to decode the user request and encode the response. By default, both are using JSON. However, images and embeddings are not well supported by JSON. You can choose msgpack which is faster and binary compatible (ref Stable Diffusion).
- Configure the threads for OpenBLAS or MKL. It might not be able to choose the most suitable CPUs used by the current Python process. You can configure it for each worker by using the env (ref custom GPU allocation).

CHAPTER EIGHT

ADOPTERS

Here are some of the companies and individual users that are using Mosec:

- Modelz: Serverless platform for ML inference.
- MOSS: An open sourced conversational language model like ChatGPT.
- TencentCloud: Tencent Cloud Machine Learning Platform, using Mosec as the core inference server framework.
- TensorChord: Cloud native AI infrastructure company.

CHAPTER NINE

CITATION

If you find this software useful for your research, please consider citing

```
@software{yang2021mosec,
  title = {{MOSEC: Model Serving made Efficient in the Cloud}},
  author = {Yang, Keming and Liu, Zichen and Cheng, Philip},
  url = {https://github.com/mosecorg/mosec},
  year = {2021}
}
```

CHAPTER

TEN

CONTRIBUTING

We welcome any kind of contribution. Please give us feedback by raising issues or discussing on Discord. You could also directly contribute your code and pull request!

To start develop, you can use envd to create an isolated and clean Python & Rust environment. Check the envd-docs or build.envd for more information.

10.1 Reference

10.1.1 CLI Arguments

```
python echo.py --help
```

```
usage: echo.py [-h] [--path PATH] [--capacity CAPACITY] [--timeout TIMEOUT]
               [--wait WAIT] [--address ADDRESS] [--port PORT]
               [--namespace NAMESPACE] [--debug]
               [--log-level {debug,info,warning,error}] [--dry-run]

Mosec Server Configurations

options:
  -h, --help            show this help message and exit
  --path PATH           Unix Domain Socket address for internal Inter-Process
                        Communication.If not set, a random path will be
                        created under the temporary dir. (default:
                        /tmp/mosec_718f8161)
  --capacity CAPACITY   Capacity of the request queue, beyond which new
                        requests will be rejected with status 429 (default:
                        1024)
  --timeout TIMEOUT     Service timeout for one request (milliseconds)
                        (default: 3000)
  --wait WAIT           [deprecated] Wait time for the batcher to batch
                        (milliseconds) (default: 10)
  --address ADDRESS     Address of the HTTP service (default: 0.0.0.0)
  --port PORT           Port of the HTTP service (default: 8000)
  --namespace NAMESPACE
                        Namespace for prometheus metrics (default:
                        mosec_service)
  --debug               Enable the service debug log (default: False)
```

(continues on next page)

(continued from previous page)

```
  --log-level {debug,info,warning,error}
                        Configure the service log level (default: info)
  --dry-run             Dry run the service with provided warmup examples (if
                        any). This will omit the worker number for each stage.
                        (default: False)

The following arguments can be set through environment variables: (path,
capacity, timeout, address, port, namespace, debug, log_level, dry_run). Note
that the environment variable should start with `MOSEC_` with upper case. For
example: `MOSEC_PORT=8080 MOSEC_TIMEOUT=5000 python main.py`.
```

10.1.2 Interface

Server

MOSEC server interface.

This module provides a way to define the service components for machine learning model serving.

Dynamic Batching

The user may enable the dynamic batching feature for any stage when the corresponding worker is appended, by setting the *append_worker(max_batch_size)*.

Multiprocessing

The user may spawn multiple processes for any stage when the corresponding worker is appended, by setting the *append_worker(num)*.

class mosec.server.**Server**

MOSEC server interface.

It allows users to sequentially append workers they implemented, builds the workflow pipeline automatically and starts up the server.

__init__()

Initialize a MOSEC Server.

register_daemon(*name*, *proc*)

Register a daemon to be monitored.

Parameters

- **name** (str) – the name of this daemon
- **proc** (Popen) – the process handle of the daemon

append_worker(*worker*, *num=1*, *max_batch_size=1*, *max_wait_time=0*, *start_method='spawn'*, *env=None*, *timeout=0*, *route='/inference'*)

Sequentially appends workers to the workflow pipeline.

Parameters

- **worker** (Type[*Worker*]) – the class you inherit from *Worker* which implements the *forward*
- **num** (int) – the number of processes for parallel computing (>=1)
- **max_batch_size** (int) – the maximum batch size allowed (>=1), will enable the dynamic batching if it > 1
- **max_wait_time** (int) – the maximum wait time (millisecond) for dynamic batching, needs to be used with *max_batch_size* to enable the feature. If not configure, will use the CLI argument *–wait* (default=10ms)
- **start_method** (str) – the process starting method ("spawn" or "fork"). (DO NOT change this unless you understand the difference between them)
- **env** (Optional[List[Dict[str, str]]]) – the environment variables to set before starting the process
- **timeout** (int) – the timeout (second) for each worker forward processing (>=1)
- **route** (Union[str, List[str]]) – the route path for this worker. If not configured, will use the default route path */inference*. If a list is provided, different route paths will share the same worker.

register_runtime(*routes*)

Register the runtime to the routes.

run()

Start the mosec model server.

mosec.server.**generate_openapi**(*workers*)

Generate the OpenAPI specification for one pipeline.

Worker

MOSEC worker interface.

This module provides the interface to define a worker with such behaviors:

1. initialize
2. serialize/deserialize data to/from another worker
3. serialize/deserialize data to/from the client side
4. data processing

class mosec.worker.**Worker**

MOSEC worker interface.

It provides default IPC (de)serialization methods, stores the worker meta data including its stage and maximum batch size, and leaves the `forward` method to be implemented by the users.

By default, we use JSON encoding. But users are free to customize via simply overriding the `deserialize` method in the **first** stage (we term it as *ingress* stage) and/or the `serialize` method in the **last** stage (we term it as *egress* stage).

For the encoding customization, there are many choices including MessagePack, Protocol Buffer and many other out-of-the-box protocols. Users can even define their own protocol and use it to manipulate the raw bytes! A naive customization can be found in this *PyTorch example*.

`__init__`()

Initialize the worker.

This method doesn't require the child class to override.

`serialize_ipc`(*data*)

Define IPC serialization method.

Parameters
`data` (`Any`) – returned data from *`forward()`*

Return type
`bytes`

`deserialize_ipc`(*data*)

Define IPC deserialization method.

Parameters
`data` (`bytes`) – input data for *`forward()`*

Return type
`Any`

`property stage: str`

Return the stage name.

`property max_batch_size: int`

Return the maximum batch size.

`property worker_id: int`

Return the ID of this worker instance.

This property returns the worker ID in the range of [1, … , `num`] (`num` as configured in *`append_worker(num)`*) to differentiate workers in the same stage.

`serialize`(*data*)

Serialize the last stage (egress).

Default response serialization method: JSON.

Check *`mosec.mixin`* for more information.

Parameters
`data` (`Any`) – the same type as the output of the *`forward()`*

Return type
`bytes`

Returns
the bytes you want to put into the response body

Raises
`EncodingError` – if the data cannot be serialized with JSON

`deserialize`(*data*)

Deserialize the first stage (ingress).

Default request deserialization method: JSON.

Check *`mosec.mixin`* for more information.

Parameters
`data` (`bytes`) – the raw bytes extracted from the request body

Return type
`Any`

Returns
the same type as the input of the *forward()*

Raises
DecodingError – if the data cannot be deserialized with JSON

abstract forward(*data*)

Model inference, data processing or computation logic.

Parameters
data (`Any`) – input data to be processed

Return type
`Any`

Must be overridden by the subclass.

If any code in this *forward()* needs to access other resources (e.g. a model, a memory cache, etc.), the user should initialize these resources as attributes of the class in the *__init__*.

Note: For a stage that enables dynamic batching, please return the results that have the same length and the same order of the input data.

Note:

- **for a single-stage worker, data will go through**
  `<deserialize> -> <forward> -> <serialize>`
- **for a multi-stage worker that is neither *ingress* not *egress*, data**
  will go through `<deserialize_ipc> -> <forward> -> <serialize_ipc>`

classmethod get_forward_json_schema(*target*, *ref_template*)

Retrieve the JSON schema for the *forward* method of the class.

Parameters

- **cls** – The class object.
- **target** (`ParseTarget`) – The target variable to parse the schema for.
- **ref_template** (`str`) – A template to use when generating `"$ref"` fields.

Return type
`Tuple[Dict[str, Any], Dict[str, Any]]`

Returns
A tuple containing the schema and the component schemas.

The *get_forward_json_schema()* method is a class method that returns the JSON schema for the *forward()* method of the `cls` class. It takes a `target` param specifying the target to parse the schema for.

The returned value is a tuple containing the schema and the component schema.

Note: Developer must implement this function to retrieve the JSON schema to enable openapi spec.

Note: The `MOSEC_REF_TEMPLATE` constant should be used as a reference template according to openapi standards.

`class mosec.worker.`**`SSEWorker`**

MOSEC worker with Server-Sent Events (SSE) support.

`send_stream_event``(`*text*, *index=0*`)`

Send a stream event to the client.

Parameters

- **`text`** (`str`) – the text to be sent, needs to be UTF-8 compatible
- **`index`** (`int`) – the index of the stream event. For the single request, this will always be 0. For dynamic batch request, this should be the index of the request in this batch.

Runtime

Managers to control Coordinator and Mosec process.

`class mosec.runtime.`**`Runtime`**`(`*worker*, *num=1*, *max_batch_size=1*, *max_wait_time=10*, *timeout=3*, *start_method='spawn'*, *env=None*`)`

The wrapper with one worker and its arguments.

`__init__``(`*worker*, *num=1*, *max_batch_size=1*, *max_wait_time=10*, *timeout=3*, *start_method='spawn'*, *env=None*`)`

Initialize the mosec coordinator.

Parameters

- **`worker`** (`Worker`) – subclass of *mosec.Worker* implemented by users.
- **`num`** (*`int`*) – number of workers
- **`max_batch_size`** (`int`) – the maximum batch size allowed (>=1), will enable the dynamic batching if it > 1
- **`max_wait_time`** (*`int`*) – the maximum wait time (millisecond) for dynamic batching, needs to be used with *max_batch_size* to enable the feature. If not configure, will use the CLI argument *–wait* (default=10ms)
- **`timeout`** (*`int`*) – timeout (second) for the *forward* function.
- **`start_method`** (`str`) – the process starting method (“spawn” or “fork”)
- **`env`** (`Optional[List[Dict[str, str]]]`) – the environment variables to set before starting the process

Errors

Exceptions used in the Worker.

Suppose the input dataflow of our model server is as follows:

bytes -> *deserialize* -> **data** -> *parse* -> **valid data**

If the raw bytes cannot be successfully deserialized, the *DecodingError* is raised; if the decoded data cannot pass the validation check (usually implemented by users), the *ValidationError* should be raised.

exception `mosec.errors.`**`MosecError`**

Bases: `Exception`

Mosec basic exception.

exception `mosec.errors.`**`ClientError`**

Bases: *`MosecError`*

Client side error.

This error indicates that the server cannot or will not process the request due to something that is perceived to be a client error. It will return the details to the client side with HTTP 400.

exception `mosec.errors.`**`ServerError`**

Bases: *`MosecError`*

Server side error.

This error indicates that the server encountered an unexpected condition that prevented it from fulfilling the request. It will return the details to the client side with HTTP 500.

Attention: be careful about the returned message since it may contain some sensitive information. If you don't want to return the details, just raise an exception that is not inherited from *mosec.errors.MosecError*.

exception `mosec.errors.`**`EncodingError`**

Bases: *`ServerError`*

Serialization error.

The *EncodingError* should be raised in user-implemented codes when the serialization for the response bytes fails. This error will set to status code to HTTP 500 and show the details in the response.

exception `mosec.errors.`**`DecodingError`**

Bases: *`ClientError`*

De-serialization error.

The *DecodingError* should be raised in user-implemented codes when the de-serialization for the request bytes fails. This error will set the status code to HTTP 400 in the response.

exception `mosec.errors.`**`ValidationError`**

Bases: *`MosecError`*

Request data validation error.

The *ValidationError* should be raised in user-implemented codes, where the validation for the input data fails. Usually, it should be put after the data de-serialization, which converts the raw bytes into structured data. This error will set the status code to HTTP 422 in the response.

exception mosec.errors.**MosecTimeoutError**

Bases: `BaseException`

Exception raised when a MOSEC worker operation times out.

If a bug in the forward code causes the worker to hang indefinitely, a timeout can be used to ensure that the worker eventually returns control to the main thread program. When a timeout occurs, the *MosecTimeout* exception is raised. This exception can be caught and handled appropriately to perform any necessary cleanup tasks or return a response indicating that the operation timed out.

Note that *MosecTimeout* is a subclass of *BaseException*, not *Exception*. This is because timeouts should not be caught and handled in the same way as other exceptions. Instead, they should be handled in a separate *except* block which isn't designed to break the working loop.

Mixins

Provide useful mixin to extend MOSEC.

class mosec.mixin.**MsgpackMixin**

Bases: `object`

Msgpack worker mixin interface.

serialize(*data*)

Serialize with msgpack for the last stage (egress).

Parameters
data (`Any`) – the **same type** as returned by *Worker.forward*

Return type
`bytes`

Returns
the bytes you want to put into the response body

Raises
EncodingError – if the data cannot be serialized with msgpack

deserialize(*data*)

Deserialize method for the first stage (ingress).

Parameters
data (`bytes`) – the raw bytes extracted from the request body

Return type
`Any`

Returns
the **same type** as the input of *Worker.forward*

Raises
DecodingError – if the data cannot be deserialized with msgpack

class mosec.mixin.**NumBinIPCMixin**

Bases: `object`

NumBin IPC worker mixin interface.

serialize_ipc(*data*)

Serialize with NumBin for the IPC.

Return type
`bytes`

`deserialize_ipc`(*data*)

Deserialize with NumBin for the IPC.

Return type
`Any`

`class mosec.mixin.PlasmaShmIPCMixin`

Bases: `Worker`

Plasma shared memory worker mixin interface.

`classmethod set_plasma_path`(*path*)

Set the plasma service path.

`serialize_ipc`(*data*)

Save the data to the plasma server and return the id.

Return type
`bytes`

`deserialize_ipc`(*data*)

Get the data from the plasma server and delete it.

Return type
`Any`

`class mosec.mixin.TypedMsgPackMixin`

Bases: `Worker`

Enable request type validation with *msgspec* and serde with *msgpack*.

`deserialize`(*data*)

Deserialize and validate request with msgspec.

Return type
`Any`

`serialize`(*data*)

Serialize with *msgpack*.

Return type
`bytes`

`classmethod get_forward_json_schema`(*target*, *ref_template*)

Get the JSON schema of the forward function.

Return type
`Tuple[Dict[str, Any], Dict[str, Any]]`

`class mosec.mixin.RedisShmIPCMixin`

Bases: `Worker`

Redis shared memory worker mixin interface.

`classmethod set_redis_url`(*url*)

Set the redis service url.

`serialize_ipc`(*data*)

Save the data to the redis server and return the id.

Return type
`bytes`

`deserialize_ipc`(*data*)

Get the data from the redis server and delete it.

Return type
`Any`

10.1.3 Concept and FAQs

There are a few terms used in `mosec`.

- `worker`: a Python process that executes the `forward` method (inherit from `mosec.Worker`)
- `stage`: one processing unit in the pipeline, each stage contains several `worker` replicas
  - also known as `Runtime` in the code
  - each stage retrieves the data from the previous stage and passes the result to the next stage
  - retrieved data will be deserialized by the `Worker.deserialize_ipc` method
  - data to be passed will be serialized by the `Worker.serialize_ipc` method
- `ingress/egress`: the first/last stage in the pipeline
  - ingress gets data from the client, while egress sends data to the client
  - data will be deserialized by the ingress `Worker.serialize` method and serialized by the egress `Worker.deserialize` method
- `pipeline`: a chain of processing stages, will be registered to an endpoint (default: `/inference`)
  - a server can have multiple pipelines, check the *multi-route* example
- `dynamic batching`: batch requests until either the max batch size or the max wait time is reached
- `controller`: a Rust tokio thread that works on:
  - read from the previous queue to get new tasks
  - send tasks to the ready-to-process worker via the Unix domain socket
  - receive results from the worker
  - send the tasks to the next queue

FAQs

How to raise an exception?

Use the `raise` keyword with *mosec.errors*. Raising other exceptions will be treated as an “500 Internal Server Error”.

If a request raises any exception, the error will be returned to the client directly without going through the rest stages.

How to change the serialization/deserialization methods?

Just let the ingress/egress worker inherit a suitable mixin like *MsgpackMixin*.

Note: The inheritance order matters in Python. Check multiple inheritance for more information.

You can also implement the `serialize/deserialize` method to your `ingress/egress` worker directly.

How to share configurations among different workers?

If the configuration structure is initialized globally, all the workers should be able to use it directly.

If you want to assign different workers with different configurations, the best way is to use the `env` (ref *append_worker*).

10.1.4 Migration Guide

This guide will help you migrate from other frameworks to `mosec`.

From the `Triton Inference Server`

Both `PyTriton` and `Triton Python Backend` are using `Triton Inference Server`.

- `mosec` doesn't require a specific client, you can use any HTTP client library
- dynamic batching is configured when calling the *append_worker*
- `mosec` doesn't need to declare the `inputs` and `outputs`. If you want to validate the request, you can use the *TypedMsgPackMixin* (ref Validate Request)

`Triton Python Backend`

- change the `TritonPythonModel` class to a worker class that inherits *mosec.Worker*
- move the `initialize` method to the `__init__` method in the new class
- move the `execute` method to the `forward` method in the new class
- if you still prefer to use the `auto_complete_config` method, you can merge it into the `__init__` method
- `mosec` doesn't have the corresponding `finalize` method as an unloading handler
- `mosec` doesn't require any special model directories or configurations
- to run multiple replicas, configure the `num` in *append_worker*

PyTriton

- move the model loading logic to the `__init__` method, since this happens in a different process
- move the `infer_func` function to the `forward` method
- *CLI Arguments*
- *Interface*
- *Concept and FAQs*
- *Migration Guide*

10.2 Examples

10.2.1 Echo Example

An echo server is usually the very first server you wanna implement to get familiar with the framework.

This server sleeps for a given period and return. It is a simple illustration of how **multi-stage workload** is implemented. It also shows how to write a simple **validation** for input data.

The default JSON protocol will be used since the (de)serialization methods are not overridden in this demo. In particular, the input `data` of `Preprocess`'s `forward` is a dictionary decoded by JSON from the request body's bytes; and the output dictionary of `Postprocess`'s `forward` will be JSON-encoded as a mirrored process.

echo.py

```python
# Copyright 2022 MOSEC Authors
#
# 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.
"""Example: Sample structures for using mosec server."""

import time
from typing import List

from mosec import Server, ValidationError, Worker, get_logger

logger = get_logger()


class Preprocess(Worker):
```

(continues on next page)

(continued from previous page)

```
    """Sample Class."""

    example = {"time": 0}

    def forward(self, data: dict) -> float:
        logger.debug("pre received %s", data)
        # Customized, simple input validation
        try:
            count_time = float(data["time"])
        except KeyError as err:
            raise ValidationError(f"cannot find key {err}") from err
        return count_time


class Inference(Worker):
    """Sample Class."""

    example = [0, 1e-5, 2e-4]

    def forward(self, data: List[float]) -> List[float]:
        logger.info("sleeping for %s seconds", max(data))
        time.sleep(max(data))
        return data


class Postprocess(Worker):
    """Sample Class."""

    def forward(self, data: float) -> dict:
        logger.debug("post received %f", data)
        return {"msg": f"sleep {data} seconds"}


if __name__ == "__main__":
    server = Server()
    server.append_worker(Preprocess)
    server.append_worker(Inference, max_batch_size=32)
    server.append_worker(Postprocess)
    server.run()
```

Start

```
python echo.py
```

Test

```
http :8000/inference time=1.5
```

10.2.2 OpenAI compatible embedding service

This example shows how to create an embedding service that is compatible with the OpenAI API.

In this example, we use the embedding model from HuggingFace LeaderBoard.

Server

```
EMB_MODEL=thenlper/gte-base python examples/embedding/server.py
```

```
# Copyright 2023 MOSEC Authors
#
# 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.

"""OpenAI compatible embedding server."""

import base64
import os
from typing import List, Union

import numpy as np
import torch  # type: ignore
import torch.nn.functional as F  # type: ignore
import transformers  # type: ignore
from llmspec import EmbeddingData, EmbeddingRequest, EmbeddingResponse, TokenUsage

from mosec import ClientError, Runtime, Server, Worker

DEFAULT_MODEL = "thenlper/gte-base"


class Embedding(Worker):
    def __init__(self):
        self.model_name = os.environ.get("EMB_MODEL", DEFAULT_MODEL)
        self.tokenizer = transformers.AutoTokenizer.from_pretrained(self.model_name)
        self.model = transformers.AutoModel.from_pretrained(self.model_name)
        self.device = (
```

(continues on next page)

(continued from previous page)

```
            torch.cuda.current_device() if torch.cuda.is_available() else "cpu"
        )

        self.model = self.model.to(self.device)
        self.model.eval()

    def get_embedding_with_token_count(
        self, sentences: Union[str, List[Union[str, List[int]]]]
    ):
        # Mean Pooling - Take attention mask into account for correct averaging
        def mean_pooling(model_output, attention_mask):
            # First element of model_output contains all token embeddings
            token_embeddings = model_output[0]
            input_mask_expanded = (
                attention_mask.unsqueeze(-1).expand(token_embeddings.size()).float()
            )
            return torch.sum(token_embeddings * input_mask_expanded, 1) / torch.clamp(
                input_mask_expanded.sum(1), min=1e-9
            )

        # Tokenize sentences
        # TODO: support `List[List[int]]` input
        encoded_input = self.tokenizer(
            sentences, padding=True, truncation=True, return_tensors="pt"
        )
        inputs = encoded_input.to(self.device)
        token_count = inputs["attention_mask"].sum(dim=1).tolist()[0]
        # Compute token embeddings
        model_output = self.model(**inputs)
        # Perform pooling
        sentence_embeddings = mean_pooling(model_output, inputs["attention_mask"])
        # Normalize embeddings
        sentence_embeddings = F.normalize(sentence_embeddings, p=2, dim=1)

        return token_count, sentence_embeddings

    def deserialize(self, data: bytes) -> EmbeddingRequest:
        return EmbeddingRequest.from_bytes(data)

    def serialize(self, data: EmbeddingResponse) -> bytes:
        return data.to_json()

    def forward(self, data: EmbeddingRequest) -> EmbeddingResponse:
        if data.model != self.model_name:
            raise ClientError(
                f"the requested model {data.model} is not supported by "
                f"this worker {self.model_name}"
            )
        token_count, embeddings = self.get_embedding_with_token_count(data.input)
        embeddings = embeddings.detach()
        if self.device != "cpu":
            embeddings = embeddings.cpu()
```

(continues on next page)

(continued from previous page)

```
        embeddings = embeddings.numpy()
        if data.encoding_format == "base64":
            embeddings = [
                base64.b64encode(emb.astype(np.float32).tobytes()).decode("utf-8")
                for emb in embeddings
            ]
        else:
            embeddings = [emb.tolist() for emb in embeddings]

        resp = EmbeddingResponse(
            data=[
                EmbeddingData(embedding=emb, index=i)
                for i, emb in enumerate(embeddings)
            ],
            model=self.model_name,
            usage=TokenUsage(
                prompt_tokens=token_count,
                # No completions performed, only embeddings generated.
                completion_tokens=0,
                total_tokens=token_count,
            ),
        )
        return resp


if __name__ == "__main__":
    server = Server()
    emb = Runtime(Embedding)
    server.register_runtime(
        {
            "/v1/embeddings": [emb],
            "/embeddings": [emb],
        }
    )
    server.run()
```

Client

```
EMB_MODEL=thenlper/gte-base python examples/embedding/client.py
```

```
# Copyright 2023 MOSEC Authors
#
# 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,
```

(continues on next page)

(continued from previous page)

```python
# 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.

"""OpenAI embedding client example."""

import os

from openai import Client

DEFAULT_MODEL = "thenlper/gte-base"

client = Client(api_key="fake", base_url="http://127.0.0.1:8000/")
emb = client.embeddings.create(
    model=os.environ.get("EMB_MODEL", DEFAULT_MODEL),
    input="Hello world!",
)
print(emb.data[0].embedding)  # type: ignore
```

10.2.3 Customized GPU Allocation

This is an example demonstrating how to give different worker processes customized environment variables to control things like GPU device allocation, etc.

Assume your machine has 4 GPUs, and you hope to deploy your model to all of them to handle inference requests in parallel, maximizing your service's throughput. With MOSEC, we provide parallel workers with customized environment variables to satisfy the needs.

As shown in the codes below, we can define our inference worker together with a list of environment variable dictionaries, each of which will be passed to the corresponding worker process. For example, if we set `CUDA_VISIBLE_DEVICES` to `0-3`, (the same copy of) our model will be deployed on 4 different GPUs and be queried in parallel, largely improving the system's throughput. You could verify this either from the server logs or the client response.

custom_env.py

```python
# Copyright 2022 MOSEC Authors
#
# 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.
"""Example: Custom Environment setup"""

import os
```

(continues on next page)

(continued from previous page)

```python
from mosec import Server, Worker, get_logger

logger = get_logger()


class Inference(Worker):
    """Customisable inference class."""

    def __init__(self):
        super().__init__()
        # initialize your models here and allocate dedicated device to it
        device = os.environ["CUDA_VISIBLE_DEVICES"]
        logger.info("initializing model on device=%s", device)

    def forward(self, data: dict) -> dict:
        device = os.environ["CUDA_VISIBLE_DEVICES"]
        # NOTE self.worker_id is 1-indexed
        logger.info("worker=%d on device=%s is processing...", self.worker_id, device)
        return {"device": device}


if __name__ == "__main__":
    NUM_DEVICE = 4

    def _get_cuda_device(cid: int) -> dict:
        return {"CUDA_VISIBLE_DEVICES": str(cid)}

    server = Server()

    server.append_worker(
        Inference, num=NUM_DEVICE, env=[_get_cuda_device(x) for x in range(NUM_DEVICE)]
    )
    server.run()
```

Start

```
python custom_env.py
```

Test

```
http :8000/inference dummy=0
```

10.2.4 Jax jitted inference

This example shows how to utilize the Jax framework to build a just-in-time (JIT) compiled inference server. You could install Jax following their official guide and you also need `chex` to run this example (`pip install -U chex`).

We use a single layer neural network for this minimal example. You could also experiment the speedup of JIT by setting the environment variable `USE_JIT=true` and observe the latency difference. Note that in the `__init__` of the worker we set the `self.multi_examples` as a list of example inputs to warmup, because different batch sizes will trigger re-jitting when they are traced for the first time.

Server

```
USE_JIT=true python examples/jax_single_layer/server.py
```

```python
# Copyright 2023 MOSEC Authors
#
# 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.
"""Example: Simple jax jitted inference with a single layer classifier."""

import os
import time
from typing import List

import chex  # type: ignore
import jax  # type: ignore
import jax.numpy as jnp  # type: ignore

from mosec import Server, ValidationError, Worker, get_logger

logger = get_logger()

INPUT_SIZE = 3
LATENT_SIZE = 16
OUTPUT_SIZE = 2

MAX_BATCH_SIZE = 8
USE_JIT = os.environ.get("USE_JIT", "false")
```

(continues on next page)

(continued from previous page)

```python
class JittedInference(Worker):
    """Sample Class."""

    def __init__(self):
        super().__init__()
        key = jax.random.PRNGKey(42)
        k_1, k_2 = jax.random.split(key)
        self._layer1_w = jax.random.normal(k_1, (INPUT_SIZE, LATENT_SIZE))
        self._layer1_b = jnp.zeros(LATENT_SIZE)
        self._layer2_w = jax.random.normal(k_2, (LATENT_SIZE, OUTPUT_SIZE))
        self._layer2_b = jnp.zeros(OUTPUT_SIZE)

        # Enumerate all batch sizes for caching.
        self.multi_examples = []
        dummy_array = list(range(INPUT_SIZE))
        for i in range(MAX_BATCH_SIZE):
            self.multi_examples.append([{"array": dummy_array}] * (i + 1))

        if USE_JIT == "true":
            self.batch_forward = jax.jit(self._batch_forward)
        else:
            self.batch_forward = self._batch_forward

    def _forward(self, x_single: jnp.ndarray) -> jnp.ndarray:  # type: ignore
        chex.assert_rank([x_single], [1])
        h_1 = jnp.dot(self._layer1_w.T, x_single) + self._layer1_b
        a_1 = jax.nn.relu(h_1)
        h_2 = jnp.dot(self._layer2_w.T, a_1) + self._layer2_b
        o_2 = jax.nn.softmax(h_2)
        return jnp.argmax(o_2, axis=-1)

    def _batch_forward(self, x_batch: jnp.ndarray) -> jnp.ndarray:  # type: ignore
        chex.assert_rank([x_batch], [2])
        return jax.vmap(self._forward)(x_batch)

    def forward(self, data: List[dict]) -> List[dict]:
        time_start = time.perf_counter()
        try:
            input_array_raw = [ele["array"] for ele in data]
        except KeyError as err:
            raise ValidationError(f"cannot find key {err}") from err
        input_array = jnp.array(input_array_raw)
        output_array = self.batch_forward(input_array)
        output_category = output_array.tolist()
        elapse = time.perf_counter() - time_start
        return [{"category": c, "elapse": elapse} for c in output_category]


if __name__ == "__main__":
    server = Server()
```

(continues on next page)

(continued from previous page)

```
    server.append_worker(JittedInference, max_batch_size=MAX_BATCH_SIZE)
    server.run()
```

Client

```
python examples/jax_single_layer/client.py
```

```
# Copyright 2023 MOSEC Authors
#
# 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.
"""Example: Client of the Jax server."""

import random
from http import HTTPStatus

import httpx

input_data = [random.randint(-99, 99), random.randint(-99, 99), random.randint(-99, 99)]
print("Client : sending data : ", input_data)

prediction = httpx.post(
    "http://127.0.0.1:8000/inference",
    json={"array": input_data},
)
if prediction.status_code == HTTPStatus.OK:
    print(prediction.json())
else:
    print(prediction.status_code, prediction.json())
```

10.2.5 Shared Memory IPC

This is an example demonstrating how you can enable the plasma shared memory store or customize your own IPC wrapper.

Mosec's multi-stage pipeline requires the output data from the previous stage to be transferred to the next stage across python processes. This is coordinated via Unix domain socket between every Python worker process from all stages and the Rust controller process.

By default, we serialize the data and directly transfer the bytes over the socket. However, users may find wrapping this IPC useful or more efficient for specific use cases. Therefore, we provide an example implementation

PlasmaShmIPCMixin based on pyarrow.plasma and RedisShmIPCMixin based on redis. We recommend using RedisShmWrapper for better performance and longer-lasting updates.

Warning: plasma is deprecated. Please use Redis instead.

The additional subprocess can be registered as a daemon thus it will be checked by mosec regularly and trigger graceful shutdown when the daemon exits.

plasma_legacy.py

```python
# Copyright 2022 MOSEC Authors
#
# 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.

"""Example: Using Plasma store with mosec mixin PlasmaShmIPCMixin.

We start a subprocess for the plasma server, and pass the path
to the plasma client which serves as the shm mixin.
We also register the plasma server process as a daemon, so
that when it exits the service is able to gracefully shutdown
and restarted by the orchestrator.
"""

import numpy as np
from pyarrow import plasma  # type: ignore

from mosec import Server, ValidationError, Worker
from mosec.mixin import PlasmaShmIPCMixin


class DataProducer(PlasmaShmIPCMixin, Worker):
    """Sample Data Producer."""

    def forward(self, data: dict) -> np.ndarray:
        # pylint: disable=duplicate-code
        try:
            nums = np.random.rand(int(data["size"]))
        except KeyError as err:
            raise ValidationError(err) from err
        return nums
```

(continues on next page)

(continued from previous page)

```python

class DataConsumer(PlasmaShmIPCMixin, Worker):
    """Sample Data Consumer."""

    def forward(self, data: np.ndarray) -> dict:
        return {"ipc test data": data.tolist()}


if __name__ == "__main__":
    # 200 Mb store, adjust the size according to your requirement
    with plasma.start_plasma_store(plasma_store_memory=200 * 1000 * 1000) as (
        shm_path,
        shm_process,
    ):
        # configure the plasma service path
        PlasmaShmIPCMixin.set_plasma_path(shm_path)

        server = Server()
        # register this process to be monitored
        server.register_daemon("plasma_server", shm_process)
        server.append_worker(DataProducer, num=2)
        server.append_worker(DataConsumer, num=2)
        server.run()
```

redis.py

```python
# Copyright 2023 MOSEC Authors
#
# 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.

"""Example: Using Redis store with mosec mixin RedisShmIPCMixin.

We start a subprocess for the Redis server, and pass the url
to the redis client which serves as the shm mixin.
We also register the redis server process as a daemon, so
that when it exits the service is able to gracefully shut down
and be restarted by the orchestrator.
"""

import subprocess
```

(continues on next page)

(continued from previous page)

```python
import numpy as np

from mosec import Server, ValidationError, Worker
from mosec.mixin import RedisShmIPCMixin


class DataProducer(RedisShmIPCMixin, Worker):
    """Sample Data Producer."""

    def forward(self, data: dict) -> np.ndarray:
        # pylint: disable=duplicate-code
        try:
            nums = np.random.rand(int(data["size"]))
        except KeyError as err:
            raise ValidationError(err) from err
        return nums


class DataConsumer(RedisShmIPCMixin, Worker):
    """Sample Data Consumer."""

    def forward(self, data: np.ndarray) -> dict:
        return {"ipc test data": data.tolist()}


if __name__ == "__main__":
    with subprocess.Popen(["redis-server"]) as p:  # start the redis server
        # configure the redis url
        RedisShmIPCMixin.set_redis_url("redis://localhost:6379/0")

        server = Server()
        # register this process to be monitored
        server.register_daemon("redis-server", p)
        server.append_worker(DataProducer, num=2)
        server.append_worker(DataConsumer, num=2)
        server.run()
```

Start

```
python examples/shm_ipc/plasma_legacy.py
```

or

```
python examples/shm_ipc/redis.py
```

Test

```
http :8000/inference size=100
```

10.2.6 Customized Metrics

This is an example demonstrating how to add your customized Python side Prometheus metrics.

Mosec already has the Rust side metrics, including:

- throughput for the inference endpoint
- duration for each stage (including the IPC time)
- batch size (only for the `max_batch_size > 1` workers)
- number of remaining tasks to be processed

If you need to monitor more details about the inference process, you can add some Python side metrics. E.g., the inference result distribution, the duration of some CPU-bound or GPU-bound processing, the IPC time (get from `rust_step_duration - python_step_duration`).

This example has a simple WSGI app as the monitoring metrics service. In each worker process, the `Counter` will collect the inference results and export them to the metrics service. For the inference part, it parses the batch data and compares them with the average value.

For more information about the multiprocess mode for the metrics, check the Prometheus doc.

python_side_metrics.py

```python
# Copyright 2022 MOSEC Authors
#
# 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.
"""Example: Adding metrics service."""

import os
import pathlib
import tempfile
from typing import List

from prometheus_client import (  # type: ignore
    CollectorRegistry,
    Counter,
    multiprocess,
```

(continues on next page)

(continued from previous page)

```
    start_http_server,
)

from mosec import Server, ValidationError, Worker, get_logger

logger = get_logger()


# check the PROMETHEUS_MULTIPROC_DIR environment variable before import Prometheus
if not os.environ.get("PROMETHEUS_MULTIPROC_DIR"):
    metric_dir_path = os.path.join(tempfile.gettempdir(), "prometheus_multiproc_dir")
    pathlib.Path(metric_dir_path).mkdir(parents=True, exist_ok=True)
    os.environ["PROMETHEUS_MULTIPROC_DIR"] = metric_dir_path


metric_registry = CollectorRegistry()
multiprocess.MultiProcessCollector(metric_registry)
counter = Counter(
    "inference_result",
    "statistic of result",
    ("status", "worker_id"),
    registry=metric_registry,
)


class Inference(Worker):
    """Sample Inference Worker."""

    def __init__(self):
        super().__init__()
        self.worker_id = str(self.worker_id)

    def deserialize(self, data: bytes) -> int:
        json_data = super().deserialize(data)
        try:
            res = int(json_data.get("num"))
        except Exception as err:
            raise ValidationError(err) from err
        return res

    def forward(self, data: List[int]) -> List[bool]:
        avg = sum(data) / len(data)
        ans = [x >= avg for x in data]
        counter.labels(status="true", worker_id=self.worker_id).inc(sum(ans))
        counter.labels(status="false", worker_id=self.worker_id).inc(
            len(ans) - sum(ans)
        )
        return ans


if __name__ == "__main__":
    # Run the metrics server in another thread.
```

(continues on next page)

(continued from previous page)

```
    start_http_server(5000, registry=metric_registry)

    # Run the inference server
    server = Server()
    server.append_worker(Inference, num=2, max_batch_size=8)
    server.run()
```

Start

```
python python_side_metrics.py
```

Test

```
http POST :8000/inference num=1
```

Check the Python side metrics

```
http :8080
```

Check the Rust side metrics

```
http :8000/metrics
```

How to build monitoring system for Mosec

In this tutorial, we will explain how to build monitoring system for Mosec, which includes Prometheus and Grafana.

Prerequisites

Before starting, you need to have Docker and Docker Compose installed on your machine. If you don't have them installed, you can follow the instructions get-docker and compose to install them.

Starting the monitoring system

Clone the repository containing the docker-compose.yaml file:

```
git clone https://github.com/mosecorg/mosec.git
```

Navigate to the directory containing the docker-compose.yaml file:

```
cd mosec/examples/monitor
```

Start the monitoring system by running the following command:

```
docker-compose up -d
```

This command will start three containers: Mosec, Prometheus, and Grafana.

Test

Run test and feed metrics to Prometheus.

```
http POST :8000/inference num=1
```

Accessing Prometheus

Prometheus is a monitoring and alerting system that collects metrics from Mosec. You can access the Prometheus UI by visiting http://127.0.0.1:9090 in your web browser.

Accessing Grafana

Grafana is a visualization tool for monitoring and analyzing metrics. You can access the Grafana UI by visiting http://127.0.0.1:3000 in your web browser. The default username and password are both admin.

Stopping the monitoring system

To stop the monitoring system, run the following command:

```
docker-compose down
```

This command will stop and remove the containers created by Docker Compose.

10.2.7 Multi-Route

This example shows how to use the multi-route feature.

You will need this feature if you want to:

- Serve multiple models in one service on different endpoints.
  - i.e. register `/embedding` & `/classify` with different models
- Serve one model to multiple different endpoints in one service.
  - i.e. register LLaMA with `/inference` and `/v1/chat/completions` to make it compatible with the OpenAI API
- Share a worker in different routes
  - The shared worker will collect the dynamic batch from multiple previous stages.
  - If you want to have multiple runtimes with sharing, you can declare multiple runtime instances with the same worker class.

The worker definition part is the same as for a single route. The only difference is how you register the worker with the server.

Here we expose a new *concept* called `Runtime`.

You can create the `Runtime` and register on the server with a `{endpoint:  [Runtime]}` dictionary.

See the complete demo code below. This will run a service with two endpoints:

- `/inference` with `Preprocess` and `Inference`
- `/v1/inference` with `TypedProcess`, `Inference` and `TypedPostprocess`

And the `Inference` worker is shared between the two routes.

Server

```python
# Copyright 2023 MOSEC Authors
#
# 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 typing import Any

from msgspec import Struct

from mosec import Runtime, Server, Worker
from mosec.mixin import TypedMsgPackMixin


class Request(Struct):
    """User request struct."""

    # pylint: disable=too-few-public-methods

    bin: bytes
    name: str = "test"


class TypedPreprocess(TypedMsgPackMixin, Worker):
    """Dummy preprocess to exit early if the validation failed."""

    def forward(self, data: Request) -> Any:
        """Input will be parse as the `Request`."""
        print(f"received from {data.name} with {data.bin!r}")
        return data.bin


class Preprocess(Worker):
    """Dummy preprocess worker."""
```

(continues on next page)

(continued from previous page)

```python
    def deserialize(self, data: bytes) -> Any:
        return data

    def forward(self, data: Any) -> Any:
        return data


class Inference(Worker):
    """Dummy inference worker."""

    def forward(self, data: Any) -> Any:
        return [{"length": len(datum)} for datum in data]


class TypedPostprocess(TypedMsgPackMixin, Worker):
    """Dummy postprocess with msgpack."""

    def forward(self, data: Any) -> Any:
        return data


if __name__ == "__main__":
    server = Server()
    typed_pre = Runtime(TypedPreprocess)
    pre = Runtime(Preprocess)
    inf = Runtime(Inference, max_batch_size=16)
    typed_post = Runtime(TypedPostprocess)
    server.register_runtime(
        {
            "/v1/inference": [typed_pre, inf, typed_post],
            "/inference": [pre, inf],
        }
    )
    server.run()
```

Client

```python
# Copyright 2023 MOSEC Authors
#
# 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
```

(continues on next page)

(continued from previous page)

```
# limitations under the License.

import json
from http import HTTPStatus

import httpx
import msgpack  # type: ignore

typed_req = {
    "bin": b"hello mosec with type check",
    "name": "type check",
}

print(">> requesting for the typed route with msgpack serde")
resp = httpx.post(
    "http://127.0.0.1:8000/v1/inference", content=msgpack.packb(typed_req)
)
if resp.status_code == HTTPStatus.OK:
    print(f"OK: {msgpack.unpackb(resp.content)}")
else:
    print(f"err[{resp.status_code}] {resp.text}")

print(">> requesting for the untyped route with json serde")
resp = httpx.post("http://127.0.0.1:8000/inference", content=b"hello mosec")
if resp.status_code == HTTPStatus.OK:
    print(f"OK: {json.loads(resp.content)}")
else:
    print(f"err[{resp.status_code}] {resp.text}")
```

10.2.8 PyTorch Examples

Here are some out-of-the-box model servers powered by mosec for PyTorch users. We use the version 1.9.0 in the following examples.

Natural Language Processing

Natural language processing model servers usually receive text data and make predictions ranging from text classification, question answering to translation and text generation.

Sentiment Analysis

This server receives a string and predicts how positive its content is. We build the model server based on Transformers of version 4.11.0.

We show how to customize the `deserialize` method of the ingress stage (`Preprocess`) and the `serialize` method of the egress stage (`Inference`). In this way, we can enjoy the high flexibility, directly reading data bytes from request body and writing the results into response body.

Note that in a stage that enables batching (e.g. `Inference` in this example), its worker's `forward` method deals with a list of data, while its `serialize` and `deserialize` methods only need to manipulate individual datum.

Server

```
python distil_bert_server_pytorch.py
```

```
# Copyright 2022 MOSEC Authors
#
# 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.
"""Example: Mosec with Pytorch Distil BERT."""

from typing import Any, List

import torch  # type: ignore
from transformers import (  # type: ignore
    AutoModelForSequenceClassification,
    AutoTokenizer,
)

from mosec import Server, Worker, get_logger

logger = get_logger()

# type alias
Returns = Any

INFERENCE_BATCH_SIZE = 32
INFERENCE_WORKER_NUM = 1


class Preprocess(Worker):
    """Preprocess BERT on current setup."""

    def __init__(self):
        super().__init__()
        self.tokenizer = AutoTokenizer.from_pretrained(
            "distilbert-base-uncased-finetuned-sst-2-english"
        )

    def deserialize(self, data: bytes) -> str:
        # Override `deserialize` for the *first* stage;
        # `data` is the raw bytes from the request body
        return data.decode()
```

(continues on next page)

(continued from previous page)

```python
    def forward(self, data: str) -> Returns:
        tokens = self.tokenizer.encode(data, add_special_tokens=True)
        return tokens


class Inference(Worker):
    """Pytorch Inference class"""

    resp_mime_type = "text/plain"

    def __init__(self):
        super().__init__()
        self.device = (
            torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")
        )
        logger.info("using computing device: %s", self.device)
        self.model = AutoModelForSequenceClassification.from_pretrained(
            "distilbert-base-uncased-finetuned-sst-2-english"
        )
        self.model.eval()
        self.model.to(self.device)

        # Overwrite self.example for warmup
        self.example = [
            [101, 2023, 2003, 1037, 8403, 4937, 999, 102] * 5  # make sentence longer
        ] * INFERENCE_BATCH_SIZE

    def forward(self, data: List[Returns]) -> List[str]:
        tensors = [torch.tensor(token) for token in data]
        with torch.no_grad():
            result = self.model(
                torch.nn.utils.rnn.pad_sequence(tensors, batch_first=True).to(
                    self.device
                )
            )[0]
        scores = result.softmax(dim=1).cpu().tolist()
        return [f"positive={p}" for (_, p) in scores]

    def serialize(self, data: str) -> bytes:
        # Override `serialize` for the *last* stage;
        # `data` is the string from the `forward` output
        return data.encode()


if __name__ == "__main__":
    server = Server()
    server.append_worker(Preprocess, num=2 * INFERENCE_WORKER_NUM)
    server.append_worker(
        Inference, max_batch_size=INFERENCE_BATCH_SIZE, num=INFERENCE_WORKER_NUM
    )
    server.run()
```

Client

```
echo 'i bought this product for many times, highly recommend' | http POST :8000/inference
```

Computer Vision

Computer vision model servers usually receive images or links to the images (downloading from the link becomes an I/O workload then), feed the preprocessed image data into the model and extract information like categories, bounding boxes and pixel labels as results.

Image Recognition

This server receives an image and classify it according to the ImageNet categorization. We specifically use ResNet as an image classifier and build a model service based on it. Nevertheless, this file serves as the starter code for any kind of image recognition model server.

We enable multiprocessing for `Preprocess` stage, so that it can produce enough tasks for `Inference` stage to do **batch inference**, which better exploits the GPU computing power. More interestingly, we also started multiple model by setting the number of worker for `Inference` stage to 2. This is because a single model hardly fully occupy the GPU memory or utilization. Multiple models running on the same device in parallel can further increase our service throughput.

When instantiating the `Server`, we enable `plasma_shm`, which utilizes the `pyarrow.plasma` as a shared memory data store for IPC. This could benefit the data transfer, especially when the data is large (preprocessed image data in this case). Note that you need to use `pip install -U pyarrow==11` to install necessary dependencies.

We also demonstrate how to customized **validation** on the data content through this example. In the `forward` method of the `Preprocess` worker, we firstly check the key of the input, then try to decode the str and load it into array. If any of these steps fails, we raise the `ValidationError`. The status will be finally returned to our clients as HTTP 422.

Server

```
python examples/resnet50_msgpack/server.py
```

```
# Copyright 2022 MOSEC Authors
#
# 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.
"""Example: Sample Resnet server."""

from io import BytesIO
```

(continues on next page)

(continued from previous page)

```python
from typing import List
from urllib.request import urlretrieve

import numpy as np  # type: ignore
import torch  # type: ignore
import torchvision  # type: ignore
from PIL import Image  # type: ignore
from torchvision import transforms  # type: ignore

from mosec import Server, ValidationError, Worker, get_logger
from mosec.mixin import MsgpackMixin

logger = get_logger()

INFERENCE_BATCH_SIZE = 16


class Preprocess(MsgpackMixin, Worker):
    """Sample Preprocess worker"""

    def __init__(self) -> None:
        super().__init__()
        trans = torch.nn.Sequential(
            transforms.Resize((256, 256)),
            transforms.CenterCrop(224),
            transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225)),
        )
        self.transform = torch.jit.script(trans)  # type: ignore

    def forward(self, data: dict):
        # Customized validation for input key and field content; raise
        # ValidationError so that the client can get 422 as http status
        try:
            image = Image.open(BytesIO(data["image"]))
        except KeyError as err:
            raise ValidationError(f"cannot find key {err}") from err
        except Exception as err:
            raise ValidationError(f"cannot decode as image data: {err}") from err

        tensor = transforms.ToTensor()(image)
        data = self.transform(tensor)  # type: ignore
        return data


class Inference(Worker):
    """Sample Inference worker"""

    def __init__(self):
        super().__init__()
        self.device = (
            torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")
        )
```

(continues on next page)

(continued from previous page)

```python
        logger.info("using computing device: %s", self.device)
        self.model = torchvision.models.resnet50(pretrained=True)
        self.model.eval()
        self.model.to(self.device)

        # Overwrite self.example for warmup
        self.example = [
            np.zeros((3, 244, 244), dtype=np.float32)
        ] * INFERENCE_BATCH_SIZE

    def forward(self, data: List[np.ndarray]) -> List[int]:
        logger.info("processing batch with size: %d", len(data))
        with torch.no_grad():
            batch = torch.stack([torch.tensor(arr, device=self.device) for arr in data])
            output = self.model(batch)
            top1 = torch.argmax(output, dim=1)
        return top1.cpu().tolist()


class Postprocess(MsgpackMixin, Worker):
    """Sample Postprocess worker"""

    def __init__(self):
        super().__init__()
        logger.info("loading categories file...")
        local_filename, _ = urlretrieve(
            "https://raw.githubusercontent.com/pytorch/hub/master/imagenet_classes.txt"
        )

        with open(local_filename, encoding="utf8") as file:
            self.categories = list(map(lambda x: x.strip(), file.readlines()))

    def forward(self, data: int) -> dict:
        return {"category": self.categories[data]}


if __name__ == "__main__":
    server = Server()
    server.append_worker(Preprocess, num=4)
    server.append_worker(Inference, num=2, max_batch_size=INFERENCE_BATCH_SIZE)
    server.append_worker(Postprocess, num=1)
    server.run()
```

Client

```
python examples/resnet50_msgpack/client.py
```

```python
# Copyright 2022 MOSEC Authors
#
# 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.
"""Example: Sample Resnet client."""

from http import HTTPStatus

import httpx
import msgpack  # type: ignore

dog_bytes = httpx.get(
    "https://raw.githubusercontent.com/pytorch/hub/master/images/dog.jpg"
).content


prediction = httpx.post(
    "http://127.0.0.1:8000/inference",
    content=msgpack.packb({"image": dog_bytes}),
)
if prediction.status_code == HTTPStatus.OK:
    print(msgpack.unpackb(prediction.content))
else:
    print(prediction.status_code, prediction.content)
```

10.2.9 Stable Diffusion

This example provides a demo service for stable diffusion. You can develop this in the container environment by using envd: `envd up -p examples/stable_diffusion`.

You should be able to try this demo under the `mosec/examples/stable_diffusion/` directory.

Server

```
envd build -t sd:serving
docker run --rm --gpus all -p 8000:8000 sd:serving
```

```python
# Copyright 2023 MOSEC Authors
#
# 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 io import BytesIO
from typing import List

import torch  # type: ignore
from diffusers import StableDiffusionPipeline  # type: ignore

from mosec import Server, Worker, get_logger
from mosec.mixin import MsgpackMixin

logger = get_logger()


class StableDiffusion(MsgpackMixin, Worker):
    def __init__(self):
        self.pipe = StableDiffusionPipeline.from_pretrained(
            "runwayml/stable-diffusion-v1-5", torch_dtype=torch.float16
        )
        device = "cuda" if torch.cuda.is_available() else "cpu"
        self.pipe = self.pipe.to(device)  # type: ignore
        self.example = ["useless example prompt"] * 4  # warmup (bs=4)

    def forward(self, data: List[str]) -> List[memoryview]:
        logger.debug("generate images for %s", data)
        res = self.pipe(data)  # type: ignore
        logger.debug("NSFW: %s", res[1])
        images = []
        for img in res[0]:  # type: ignore
            dummy_file = BytesIO()
            img.save(dummy_file, format="JPEG")  # type: ignore
            images.append(dummy_file.getbuffer())
        return images


if __name__ == "__main__":
```

(continues on next page)

(continued from previous page)

```
    server = Server()
    server.append_worker(StableDiffusion, num=1, max_batch_size=4, max_wait_time=10)
    server.run()
```

```
python server.py --timeout 30000
```

Client

```
python client.py --prompt "a cute cat site on the basketball"
```

```
# Copyright 2023 MOSEC Authors
#
# 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 http import HTTPStatus

import httpx
import msgpack  # type: ignore

parser = argparse.ArgumentParser(
    prog="stable diffusion client demo",
)
parser.add_argument(
    "-p", "--prompt", default="a photo of an astronaut riding a horse on mars"
)
parser.add_argument(
    "-o", "--output", default="stable_diffusion_result.jpg", help="output filename"
)
parser.add_argument(
    "--port",
    default=8000,
    type=int,
    help="service port",
)


args = parser.parse_args()
resp = httpx.post(
    f"http://127.0.0.1:{args.port}/inference",
```

(continues on next page)

(continued from previous page)

```python
    content=msgpack.packb(args.prompt),
    timeout=httpx.Timeout(20),
)
if resp.status_code == HTTPStatus.OK:
    data = msgpack.unpackb(resp.content)
    with open(args.output, "wb") as f:
        f.write(data)
else:
    print(f"ERROR: <{resp.status_code}> {resp.text}")
```

10.2.10 Validate Request

This example shows how to use the `TypedMsgPackMixin` to validate the request with the help of `msgspec`.

Request validation can provide the following benefits:

- The client can know the exact expected data schema from the type definition.
- Validation failure will return the details of the failure reason to help the client debug.
- Ensure that the service is working on the correct data without fear.

First of all, define the request type with `msgspec.Struct` like:

```python
class Request(msgspec.Struct):
    media: str
    binary: bytes
```

Then, apply the `TypedMsgPackMixin` mixin and add the type you defined to the annotation of `forward(self, data)`:

```python
class Inference(TypedMsgPackMixin, Worker):
    def forward(self, data: Request):
        pass
```

Note: If you are using dynamic **batch** inference as the first stage, just use the `List[Request]` as the annotation.

You can check the full demo code below.

Server

```python
# Copyright 2023 MOSEC Authors
#
# 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,
```

(continues on next page)

(continued from previous page)

```python
# 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.

"""Request validation example."""

from typing import Any, List

from msgspec import Struct

from mosec import Server, Worker
from mosec.mixin import TypedMsgPackMixin


class Request(Struct):
    """User request struct."""

    # pylint: disable=too-few-public-methods

    bin: bytes
    name: str = "test"


class Preprocess(TypedMsgPackMixin, Worker):
    """Dummy preprocess to exit early if the validation failed."""

    def forward(self, data: Request) -> Any:
        """Input will be parse as the `Request`."""
        print(f"received {data}")
        return data.bin


class Inference(TypedMsgPackMixin, Worker):
    """Dummy batch inference."""

    def forward(self, data: List[bytes]) -> List[int]:
        return [len(buf) for buf in data]


if __name__ == "__main__":
    server = Server()
    server.append_worker(Preprocess)
    server.append_worker(Inference, max_batch_size=16)
    server.run()
```

Client

```python
# Copyright 2023 MOSEC Authors
#
# 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 http import HTTPStatus

import httpx
import msgpack  # type: ignore

req = {
    "bin": b"hello mosec",
    "name": "type check",
}

resp = httpx.post("http://127.0.0.1:8000/inference", content=msgpack.packb(req))
if resp.status_code == HTTPStatus.OK:
    print(f"OK: {msgpack.unpackb(resp.content)}")
else:
    print(f"err[{resp.status_code}] {resp.text}")
```

Test

```
python client.py
```

We provide examples across different ML frameworks and for various tasks in this section.

10.2.11 Requirements

All the examples in this section are self-contained and tested. Feel free to grab one and run:

```
python model_server.py
```

To test the server, we use `httpie` and `httpx` by default. You can have other choices but if you want to install them:

```
pip install httpie httpx
```

10.3 Development

10.3.1 Contributing to `Mosec`

Before contributing to this repository, please first discuss the change you wish to make via issue, email, or any other method with the owners of this repository before making a change.

Pull Request Process

1. After you have forked this repository, you could use `make install` for *the first time* to install the local development dependencies; afterward, you may use `make dev` to build the library when you have made any code changes.
2. Before committing your changes, you can use `make format && make lint` to ensure the codes follow our style standards.
3. Please add corresponding tests to your change if that's related to new feature or API, and ensure `make test` can pass.
4. Submit your pull request.

Contacts

- Keming
- zclzc
- *Contributing to Mosec*

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex

PYTHON MODULE INDEX

INDEX