On this page

Enhanced Inference

`autogen.OpenAIWrapper` provides enhanced LLM inference for `openai>=1`. `autogen.Completion` is a drop-in replacement of `openai.Completion` and `openai.ChatCompletion` for enhanced LLM inference using `openai<1`. There are a number of benefits of using `autogen` to perform inference: performance tuning, API unification, caching, error handling, multi-config inference, result filtering, templating and so on.

Tune Inference Parameters (for openai<1)

Find a list of examples in this page: Tune Inference Parameters Examples

Choices to optimize

The cost of using foundation models for text generation is typically measured in terms of the number of tokens in the input and output combined. From the perspective of an application builder using foundation models, the use case is to maximize the utility of the generated text under an inference budget constraint (e.g., measured by the average dollar cost needed to solve a coding problem). This can be achieved by optimizing the hyperparameters of the inference, which can significantly affect both the utility and the cost of the generated text.

The tunable hyperparameters include:

1. model - this is a required input, specifying the model ID to use.
2. prompt/messages - the input prompt/messages to the model, which provides the context for the text generation task.
3. max_tokens - the maximum number of tokens (words or word pieces) to generate in the output.
4. temperature - a value between 0 and 1 that controls the randomness of the generated text. A higher temperature will result in more random and diverse text, while a lower temperature will result in more predictable text.
5. top_p - a value between 0 and 1 that controls the sampling probability mass for each token generation. A lower top_p value will make it more likely to generate text based on the most likely tokens, while a higher value will allow the model to explore a wider range of possible tokens.
6. n - the number of responses to generate for a given prompt. Generating multiple responses can provide more diverse and potentially more useful output, but it also increases the cost of the request.
7. stop - a list of strings that, when encountered in the generated text, will cause the generation to stop. This can be used to control the length or the validity of the output.
8. presence_penalty, frequency_penalty - values that control the relative importance of the presence and frequency of certain words or phrases in the generated text.
9. best_of - the number of responses to generate server-side when selecting the "best" (the one with the highest log probability per token) response for a given prompt.

The cost and utility of text generation are intertwined with the joint effect of these hyperparameters. There are also complex interactions among subsets of the hyperparameters. For example, the temperature and top_p are not recommended to be altered from their default values together because they both control the randomness of the generated text, and changing both at the same time can result in conflicting effects; n and best_of are rarely tuned together because if the application can process multiple outputs, filtering on the server side causes unnecessary information loss; both n and max_tokens will affect the total number of tokens generated, which in turn will affect the cost of the request. These interactions and trade-offs make it difficult to manually determine the optimal hyperparameter settings for a given text generation task.

Do the choices matter? Check this blogpost to find example tuning results about gpt-3.5-turbo and gpt-4.

With AutoGen, the tuning can be performed with the following information:

1. Validation data.
2. Evaluation function.
3. Metric to optimize.
4. Search space.
5. Budgets: inference and optimization respectively.

Validation data

Collect a diverse set of instances. They can be stored in an iterable of dicts. For example, each instance dict can contain "problem" as a key and the description str of a math problem as the value; and "solution" as a key and the solution str as the value.

Evaluation function

The evaluation function should take a list of responses, and other keyword arguments corresponding to the keys in each validation data instance as input, and output a dict of metrics. For example,

```
def eval_math_responses(responses: List[str], solution: str, **args) -> Dict:
    # select a response from the list of responses
    answer = voted_answer(responses)
    # check whether the answer is correct
    return {"success": is_equivalent(answer, solution)}
```

`autogen.code_utils` and `autogen.math_utils` offer some example evaluation functions for code generation and math problem solving.

Metric to optimize

The metric to optimize is usually an aggregated metric over all the tuning data instances. For example, users can specify "success" as the metric and "max" as the optimization mode. By default, the aggregation function is taking the average. Users can provide a customized aggregation function if needed.

Search space

Users can specify the (optional) search range for each hyperparameter.

1. model. Either a constant str, or multiple choices specified by `flaml.tune.choice`.
2. prompt/messages. Prompt is either a str or a list of strs, of the prompt templates. messages is a list of dicts or a list of lists, of the message templates. Each prompt/message template will be formatted with each data instance. For example, the prompt template can be: "{problem} Solve the problem carefully. Simplify your answer as much as possible. Put the final answer in \boxed{{}}." And `{problem}` will be replaced by the "problem" field of each data instance.
3. max_tokens, n, best_of. They can be constants, or specified by `flaml.tune.randint`, `flaml.tune.qrandint`, `flaml.tune.lograndint` or `flaml.qlograndint`. By default, max_tokens is searched in [50, 1000); n is searched in [1, 100); and best_of is fixed to 1.
4. stop. It can be a str or a list of strs, or a list of lists of strs or None. Default is None.
5. temperature or top_p. One of them can be specified as a constant or by `flaml.tune.uniform` or `flaml.tune.loguniform` etc. Please don't provide both. By default, each configuration will choose either a temperature or a top_p in [0, 1] uniformly.
6. presence_penalty, frequency_penalty. They can be constants or specified by `flaml.tune.uniform` etc. Not tuned by default.

Budgets

One can specify an inference budget and an optimization budget. The inference budget refers to the average inference cost per data instance. The optimization budget refers to the total budget allowed in the tuning process. Both are measured by dollars and follow the price per 1000 tokens.

Perform tuning

Now, you can use `autogen.Completion.tune` for tuning. For example,

```
import autogen

config, analysis = autogen.Completion.tune(
    data=tune_data,
    metric="success",
    mode="max",
    eval_func=eval_func,
    inference_budget=0.05,
    optimization_budget=3,
    num_samples=-1,
)
```

`num_samples` is the number of configurations to sample. -1 means unlimited (until optimization budget is exhausted). The returned `config` contains the optimized configuration and `analysis` contains an ExperimentAnalysis object for all the tried configurations and results.

The tuned config can be used to perform inference.

API unification

`autogen.OpenAIWrapper.create()` can be used to create completions for both chat and non-chat models, and both OpenAI API and Azure OpenAI API.

```python
from autogen import OpenAIWrapper
# OpenAI endpoint
client = OpenAIWrapper()
# ChatCompletion
response = client.create(messages=[{"role": "user", "content": "2+2="}], model="gpt-3.5-turbo")
# extract the response text
print(client.extract_text_or_completion_object(response))
# get cost of this completion
print(response.cost)
# Azure OpenAI endpoint
client = OpenAIWrapper(api_key=..., base_url=..., api_version=..., api_type="azure")
# Completion
response = client.create(prompt="2+2=", model="gpt-3.5-turbo-instruct")
# extract the response text
print(client.extract_text_or_completion_object(response))
```

For local LLMs, one can spin up an endpoint using a package like FastChat, and then use the same API to send a request. See here for examples on how to make inference with local LLMs.

Usage Summary

The `OpenAIWrapper` from `autogen` tracks token counts and costs of your API calls. Use the `create()` method to initiate requests and `print_usage_summary()` to retrieve a detailed usage report, including total cost and token usage for both cached and actual requests.

- `mode=["actual", "total"]` (default): print usage summary for all completions and non-caching completions.
- `mode='actual'`: only print non-cached usage.
- `mode='total'`: only print all usage (including cache).

Reset your session's usage data with `clear_usage_summary()` when needed. View Notebook

Example usage:

```python
from autogen import OpenAIWrapper

client = OpenAIWrapper()
client.create(messages=[{"role": "user", "content": "Python learning tips."}], model="gpt-3.5-turbo")
client.print_usage_summary()  # Display usage
client.clear_usage_summary()  # Reset usage data
```

Sample output:

```
Usage summary excluding cached usage:
Total cost: 0.00015
* Model 'gpt-3.5-turbo': cost: 0.00015, prompt_tokens: 25, completion_tokens: 58, total_tokens: 83

Usage summary including cached usage:
Total cost: 0.00027
* Model 'gpt-3.5-turbo': cost: 0.00027, prompt_tokens: 50, completion_tokens: 100, total_tokens: 150
```

Caching

API call results are cached locally and reused when the same request is issued. This is useful when repeating or continuing experiments for reproducibility and cost saving. It still allows controlled randomness by setting the "cache_seed" specified in `OpenAIWrapper.create()` or the constructor of `OpenAIWrapper`.

```python
client = OpenAIWrapper(cache_seed=...)
client.create(...)
client = OpenAIWrapper()
client.create(cache_seed=..., ...)
```

Caching is enabled by default with cache_seed 41. To disable it please set `cache_seed` to None.

NOTE. openai v1.1 introduces a new param `seed`. The difference between autogen's `cache_seed` and openai's `seed` is that:

- autogen uses local disk cache to guarantee the exactly same output is produced for the same input and when cache is hit, no openai api call will be made.
- openai's `seed` is a best-effort deterministic sampling with no guarantee of determinism. When using openai's `seed` with `cache_seed` set to None, even for the same input, an openai api call will be made and there is no guarantee for getting exactly the same output.

Error handling

Runtime error

One can pass a list of configurations of different models/endpoints to mitigate the rate limits and other runtime error. For example,

```
client = OpenAIWrapper(
    config_list=[
        {
            "model": "gpt-4",
            "api_key": os.environ.get("AZURE_OPENAI_API_KEY"),
            "api_type": "azure",
            "base_url": os.environ.get("AZURE_OPENAI_API_BASE"),
            "api_version": "2023-08-01-preview",
        },
        {
            "model": "gpt-3.5-turbo",
            "api_key": os.environ.get("OPENAI_API_KEY"),
            "base_url": "https://api.openai.com/v1",
        },
        {
            "model": "llama2-chat-7B",
            "base_url": "http://127.0.0.1:8080",
        }
    ],
)
```

`client.create()` will try querying Azure OpenAI gpt-4, OpenAI gpt-3.5-turbo, and a locally hosted llama2-chat-7B one by one, until a valid result is returned. This can speed up the development process where the rate limit is a bottleneck. An error will be raised if the last choice fails. So make sure the last choice in the list has the best availability.

For convenience, we provide a number of utility functions to load config lists.

- `get_config_list`: Generates configurations for API calls, primarily from provided API keys.
- `config_list_openai_aoai`: Constructs a list of configurations using both Azure OpenAI and OpenAI endpoints, sourcing API keys from environment variables or local files.
- `config_list_from_json`: Loads configurations from a JSON structure, either from an environment variable or a local JSON file, with the flexibility of filtering configurations based on given criteria.
- `config_list_from_models`: Creates configurations based on a provided list of models, useful when targeting specific models without manually specifying each configuration.
- `config_list_from_dotenv`: Constructs a configuration list from a `.env` file, offering a consolidated way to manage multiple API configurations and keys from a single file.

We suggest that you take a look at this notebook for full code examples of the different methods to configure your model endpoints.

Logic error

Another type of error is that the returned response does not satisfy a requirement. For example, if the response is required to be a valid json string, one would like to filter the responses that are not. This can be achieved by providing a list of configurations and a filter function. For example,

```
def valid_json_filter(response, **_):
    for text in OpenAIWrapper.extract_text_or_completion_object(response):
        try:
            json.loads(text)
            return True
        except ValueError:
            pass
    return False

client = OpenAIWrapper(
    config_list=[{"model": "text-ada-001"}, {"model": "gpt-3.5-turbo-instruct"}, {"model": "text-davinci-003"}],
)
response = client.create(
    prompt="How to construct a json request to Bing API to search for 'latest AI news'? Return the JSON request.",
    filter_func=valid_json_filter,
)
```

The example above will try to use text-ada-001, gpt-3.5-turbo-instruct, and text-davinci-003 iteratively, until a valid json string is returned or the last config is used. One can also repeat the same model in the list for multiple times (with different seeds) to try one model multiple times for increasing the robustness of the final response.

Advanced use case: Check this blogpost to find how to improve GPT-4's coding performance from 68% to 90% while reducing the inference cost.

Templating

If the provided prompt or message is a template, it will be automatically materialized with a given context. For example,

```
response = client.create(
    context={"problem": "How many positive integers, not exceeding 100, are multiples of 2 or 3 but not 4?"},
    prompt="{problem} Solve the problem carefully.",
    allow_format_str_template=True,
    **config
)
```

A template is either a format str, like the example above, or a function which produces a str from several input fields, like the example below.

```
def content(turn, context):
    return "\n".join(
        [
            context[f"user_message_{turn}"],
            context[f"external_info_{turn}"]
        ]
    )

messages = [
    {
        "role": "system",
        "content": "You are a teaching assistant of math.",
    },
    {
        "role": "user",
        "content": partial(content, turn=0),
    },
]
context = {
    "user_message_0": "Could you explain the solution to Problem 1?",
    "external_info_0": "Problem 1: ...",
}

response = client.create(context=context, messages=messages, **config)
messages.append(
    {
        "role": "assistant",
        "content": client.extract_text(response)[0]
    }
)
messages.append(
    {
        "role": "user",
        "content": partial(content, turn=1),
    },
)
context.append(
    {
        "user_message_1": "Why can't we apply Theorem 1 to Equation (2)?",
        "external_info_1": "Theorem 1: ...",
    }
)
response = client.create(context=context, messages=messages, **config)
```

Logging (for openai<1)

When debugging or diagnosing an LLM-based system, it is often convenient to log the API calls and analyze them. `autogen.Completion` and `autogen.ChatCompletion` offer an easy way to collect the API call histories. For example, to log the chat histories, simply run:

```
autogen.ChatCompletion.start_logging()
```

The API calls made after this will be automatically logged. They can be retrieved at any time by:

```
autogen.ChatCompletion.logged_history
```

There is a function that can be used to print usage summary (total cost, and token count usage from each model):

```
autogen.ChatCompletion.print_usage_summary()
```

To stop logging, use

```
autogen.ChatCompletion.stop_logging()
```

If one would like to append the history to an existing dict, pass the dict like:

```
autogen.ChatCompletion.start_logging(history_dict=existing_history_dict)
```

By default, the counter of API calls will be reset at `start_logging()`. If no reset is desired, set `reset_counter=False`.

There are two types of logging formats: compact logging and individual API call logging. The default format is compact. Set `compact=False` in `start_logging()` to switch.

- Example of a history dict with compact logging.

```
{
    """
    [
        {
            'role': 'system',
            'content': system_message,
        },
        {
            'role': 'user',
            'content': user_message_1,
        },
        {
            'role': 'assistant',
            'content': assistant_message_1,
        },
        {
            'role': 'user',
            'content': user_message_2,
        },
        {
            'role': 'assistant',
            'content': assistant_message_2,
        },
    ]""": {
        "created_at": [0, 1],
        "cost": [0.1, 0.2],
    }
}
```

- Example of a history dict with individual API call logging.

```
{
    0: {
        "request": {
            "messages": [
                {
                    "role": "system",
                    "content": system_message,
                },
                {
                    "role": "user",
                    "content": user_message_1,
                }
            ],
            ... # other parameters in the request
        },
        "response": {
            "choices": [
                "messages": {
                    "role": "assistant",
                    "content": assistant_message_1,
                },
            ],
            ... # other fields in the response
        }
    },
    1: {
        "request": {
            "messages": [
                {
                    "role": "system",
                    "content": system_message,
                },
                {
                    "role": "user",
                    "content": user_message_1,
                },
                {
                    "role": "assistant",
                    "content": assistant_message_1,
                },
                {
                    "role": "user",
                    "content": user_message_2,
                },
            ],
            ... # other parameters in the request
        },
        "response": {
            "choices": [
                "messages": {
                    "role": "assistant",
                    "content": assistant_message_2,
                },
            ],
            ... # other fields in the response
        }
    },
}
```

- Example of printing for usage summary

```
Total cost: <cost>
Token count summary for model <model>: prompt_tokens: <count 1>, completion_tokens: <count 2>, total_tokens: <count
```

It can be seen that the individual API call history contains redundant information of the conversation. For a long conversation the degree of redundancy is high. The compact history is more efficient and the individual API call history contains more details.

Edit this page

Previous
« Multi-agent Conversation Framework

Next
Contributing »

Twitter