```
import logging
from unittest.mock import patch

import pytest
import torch

from swarm_models.huggingface import HuggingfaceLLM


# Fixture for the class instance
@pytest.fixture
def llm_instance():
    model_id = "NousResearch/Nous-Hermes-2-Vision-Alpha"
    instance = HuggingfaceLLM(model_id=model_id)
    return instance


# Test for instantiation and attributes
def test_llm_initialization(llm_instance):
    assert (
        llm_instance.model_id
        == "NousResearch/Nous-Hermes-2-Vision-Alpha"
    )
    assert llm_instance.max_length == 500
    # ... add more assertions for all default attributes
# Parameterized test for setting devices
@pytest.mark.parametrize("device", ["cpu", "cuda"])
def test_llm_set_device(llm_instance, device):
    llm_instance.set_device(device)
    assert llm_instance.device == device


# Test exception during initialization with a bad model_id
def test_llm_bad_model_initialization():
    with pytest.raises(Exception):
        HuggingfaceLLM(model_id="unknown-model")


# # Mocking the tokenizer and model to test run method
# @patch("swarms.models.huggingface.AutoTokenizer.from_pretrained")
# @patch(
#     "swarms.models.huggingface.AutoModelForCausalLM.from_pretrained"
# )
# def test_llm_run(mock_model, mock_tokenizer, llm_instance):
#     mock_model.return_value.generate.return_value = "mocked output"
#     mock_tokenizer.return_value.encode.return_value = "mocked input"
#     result = llm_instance.run("test task")
#     assert result == "mocked output"
# Async test (requires pytest-asyncio plugin)
@pytest.mark.asyncio
async def test_llm_run_async(llm_instance):
    result = await llm_instance.run_async("test task")
    assert isinstance(result, str)


# Test for checking GPU availability
def test_llm_gpu_availability(llm_instance):
    # Assuming the test is running on a machine where the GPU availability is known
    expected_result = torch.cuda.is_available()
    assert llm_instance.gpu_available() == expected_result


# Test for memory consumption reporting
def test_llm_memory_consumption(llm_instance):
    # Mocking torch.cuda functions for consistent results
    with patch("torch.cuda.memory_allocated", return_value=1024):
        with patch("torch.cuda.memory_reserved", return_value=2048):
            memory = llm_instance.memory_consumption()
    assert memory == {"allocated": 1024, "reserved": 2048}


# Test different initialization parameters
@pytest.mark.parametrize(
    "model_id, max_length",
    [
        ("NousResearch/Nous-Hermes-2-Vision-Alpha", 100),
        ("microsoft/Orca-2-13b", 200),
        (
            "berkeley-nest/Starling-LM-7B-alpha",
            None,
        ),  # None to check default behavior
    ],
)
def test_llm_initialization_params(model_id, max_length):
    if max_length:
        instance = HuggingfaceLLM(
            model_id=model_id, max_length=max_length
        )
        assert instance.max_length == max_length
    else:
        instance = HuggingfaceLLM(model_id=model_id)
        assert (
            instance.max_length == 500
        )  # Assuming 500 is the default max_length


# Test for setting an invalid device
def test_llm_set_invalid_device(llm_instance):
    with pytest.raises(ValueError):
        llm_instance.set_device("quantum_processor")
```

```
# Mocking external API call to test run method without network
@patch("swarms.models.huggingface.HuggingfaceLLM.run")
def test_llm_run_without_network(mock_run, llm_instance):
    mock_run.return_value = "mocked output"
    result = llm_instance.run("test task without network")
    assert result == "mocked output"



# Test handling of empty input for the run method
def test_llm_run_empty_input(llm_instance):
    with pytest.raises(ValueError):
        llm_instance.run("")



# Test the generation with a provided seed for reproducibility
@patch("swarms.models.huggingface.HuggingfaceLLM.run")
def test_llm_run_with_seed(mock_run, llm_instance):
    seed = 42
    llm_instance.set_seed(seed)
    # Assuming set_seed method affects the randomness in the model
    # You would typically ensure that setting the seed gives reproducible results
    mock_run.return_value = "mocked deterministic output"
    result = llm_instance.run("test task", seed=seed)
    assert result == "mocked deterministic output"
```

```
# Test the output length is as expected
@patch("swarms.models.huggingface.HuggingfaceLLM.run")
def test_llm_run_output_length(mock_run, llm_instance):
    input_text = "test task"
    llm_instance.max_length = 50  # set a max_length for the output
    mock_run.return_value = "mocked output" * 10  # some long text
    result = llm_instance.run(input_text)
    assert len(result.split()) <= llm_instance.max_length


# Test the tokenizer handling special tokens correctly
@patch("swarms.models.huggingface.HuggingfaceLLM._tokenizer.encode")
@patch("swarms.models.huggingface.HuggingfaceLLM._tokenizer.decode")
def test_llm_tokenizer_special_tokens(
    mock_decode, mock_encode, llm_instance
):
    mock_encode.return_value = "encoded input with special tokens"
    mock_decode.return_value = "decoded output with special tokens"
    result = llm_instance.run("test task with special tokens")
    mock_encode.assert_called_once()
    mock_decode.assert_called_once()
    assert "special tokens" in result
```

```
# Test for correct handling of timeouts
@patch("swarms.models.huggingface.HuggingfaceLLM.run")
def test_llm_timeout_handling(mock_run, llm_instance):
    mock_run.side_effect = TimeoutError
    with pytest.raises(TimeoutError):
        llm_instance.run("test task with timeout")


# Test for response time within a threshold (performance test)
@patch("swarms.models.huggingface.HuggingfaceLLM.run")
def test_llm_response_time(mock_run, llm_instance):
    import time

    mock_run.return_value = "mocked output"
    start_time = time.time()
    llm_instance.run("test task for response time")
    end_time = time.time()
    assert (
        end_time - start_time < 1
    )  # Assuming the response should be faster than 1 second


# Test the logging of a warning for long inputs
@patch("swarms.models.huggingface.logging.warning")
def test_llm_long_input_warning(mock_warning, llm_instance):
    long_input = "x" * 10000  # input longer than the typical limit
    llm_instance.run(long_input)
    mock_warning.assert_called_once()



# Test for run method behavior when model raises an exception
@patch(
    "swarms.models.huggingface.HuggingfaceLLM._model.generate",
    side_effect=RuntimeError,
)
def test_llm_run_model_exception(mock_generate, llm_instance):
    with pytest.raises(RuntimeError):
        llm_instance.run("test task when model fails")



# Test the behavior when GPU is forced but not available
@patch("torch.cuda.is_available", return_value=False)
def test_llm_force_gpu_when_unavailable(
    mock_is_available, llm_instance
):
    with pytest.raises(EnvironmentError):
        llm_instance.set_device(
            "cuda"
        )  # Attempt to set CUDA when it's not available



# Test for proper cleanup after model use (releasing resources)
```

```
@patch("swarms.models.huggingface.HuggingfaceLLM._model")

def test_llm_cleanup(mock_model, mock_tokenizer, llm_instance):

    llm_instance.cleanup()

    # Assuming cleanup method is meant to free resources

    mock_model.delete.assert_called_once()

    mock_tokenizer.delete.assert_called_once()




# Test model's ability to handle multilingual input

@patch("swarms.models.huggingface.HuggingfaceLLM.run")

def test_llm_multilingual_input(mock_run, llm_instance):

    mock_run.return_value = "mocked multilingual output"

    multilingual_input = "Bonjour, ceci est un test multilingue."

    result = llm_instance.run(multilingual_input)

    assert isinstance(

        result, str

    )  # Simple check to ensure output is string type




# Test caching mechanism to prevent re-running the same inputs

@patch("swarms.models.huggingface.HuggingfaceLLM.run")

def test_llm_caching_mechanism(mock_run, llm_instance):

    input_text = "test caching mechanism"

    mock_run.return_value = "cached output"

    # Run the input twice

    first_run_result = llm_instance.run(input_text)
    second_run_result = llm_instance.run(input_text)
    mock_run.assert_called_once()  # Should only be called once due to caching
    assert first_run_result == second_run_result


# These tests are provided as examples. In real-world scenarios, you will need to adapt these tests to the actual logic of your `HuggingfaceLLM` class.
# For instance, "mock_model.delete.assert_called_once()" and similar lines are based on hypothetical methods and behaviors that you need to replace with actual implementations.


# Mock some functions and objects for testing
@pytest.fixture
def mock_huggingface_llm(monkeypatch):
    # Mock the model and tokenizer creation
    def mock_init(
        self,
        model_id,
        device="cpu",
        max_length=500,
        quantize=False,
        quantization_config=None,
        verbose=False,
        distributed=False,
        decoding=False,
        max_workers=5,
        repitition_penalty=1.3,
        no_repeat_ngram_size=5,
        temperature=0.7,
        top_k=40,
        top_p=0.8,
    ):
        pass

    # Mock the model loading
    def mock_load_model(self):
        pass

    # Mock the model generation
    def mock_run(self, task):
        pass

    monkeypatch.setattr(HuggingfaceLLM, "__init__", mock_init)
    monkeypatch.setattr(HuggingfaceLLM, "load_model", mock_load_model)
    monkeypatch.setattr(HuggingfaceLLM, "run", mock_run)


# Basic tests for initialization and attribute settings
def test_init_huggingface_llm():
    llm = HuggingfaceLLM(
        model_id="test_model",
        device="cuda",
        max_length=1000,
        quantize=True,
        quantization_config={"config_key": "config_value"},
        verbose=True,
        distributed=True,
        decoding=True,
        max_workers=3,
        repitition_penalty=1.5,
        no_repeat_ngram_size=4,
        temperature=0.8,
        top_k=50,
        top_p=0.7,
    )

    assert llm.model_id == "test_model"
    assert llm.device == "cuda"
    assert llm.max_length == 1000
    assert llm.quantize is True
    assert llm.quantization_config == {"config_key": "config_value"}
    assert llm.verbose is True
    assert llm.distributed is True
    assert llm.decoding is True
    assert llm.max_workers == 3
    assert llm.repitition_penalty == 1.5
    assert llm.no_repeat_ngram_size == 4
    assert llm.temperature == 0.8
```

```
    assert llm.top_k == 50

    assert llm.top_p == 0.7




# Test loading the model

def test_load_model(mock_huggingface_llm):

    llm = HuggingfaceLLM(model_id="test_model")

    llm.load_model()




# Test running the model

def test_run(mock_huggingface_llm):

    llm = HuggingfaceLLM(model_id="test_model")

    llm.run("Test prompt")




# Test for setting max_length

def test_llm_set_max_length(llm_instance):

    new_max_length = 1000

    llm_instance.set_max_length(new_max_length)

    assert llm_instance.max_length == new_max_length




# Test for setting verbose

def test_llm_set_verbose(llm_instance):

    llm_instance.set_verbose(True)
    assert llm_instance.verbose is True


# Test for setting distributed
def test_llm_set_distributed(llm_instance):
    llm_instance.set_distributed(True)
    assert llm_instance.distributed is True


# Test for setting decoding
def test_llm_set_decoding(llm_instance):
    llm_instance.set_decoding(True)
    assert llm_instance.decoding is True


# Test for setting max_workers
def test_llm_set_max_workers(llm_instance):
    new_max_workers = 10
    llm_instance.set_max_workers(new_max_workers)
    assert llm_instance.max_workers == new_max_workers


# Test for setting repitition_penalty
def test_llm_set_repitition_penalty(llm_instance):
    new_repitition_penalty = 1.5
    llm_instance.set_repitition_penalty(new_repitition_penalty)
    assert llm_instance.repitition_penalty == new_repitition_penalty



# Test for setting no_repeat_ngram_size
def test_llm_set_no_repeat_ngram_size(llm_instance):
    new_no_repeat_ngram_size = 6
    llm_instance.set_no_repeat_ngram_size(new_no_repeat_ngram_size)
    assert (
        llm_instance.no_repeat_ngram_size == new_no_repeat_ngram_size
    )



# Test for setting temperature
def test_llm_set_temperature(llm_instance):
    new_temperature = 0.8
    llm_instance.set_temperature(new_temperature)
    assert llm_instance.temperature == new_temperature



# Test for setting top_k
def test_llm_set_top_k(llm_instance):
    new_top_k = 50
    llm_instance.set_top_k(new_top_k)
    assert llm_instance.top_k == new_top_k
```

```
# Test for setting top_p
def test_llm_set_top_p(llm_instance):
    new_top_p = 0.9
    llm_instance.set_top_p(new_top_p)
    assert llm_instance.top_p == new_top_p


# Test for setting quantize
def test_llm_set_quantize(llm_instance):
    llm_instance.set_quantize(True)
    assert llm_instance.quantize is True


# Test for setting quantization_config
def test_llm_set_quantization_config(llm_instance):
    new_quantization_config = {
        "load_in_4bit": False,
        "bnb_4bit_use_double_quant": False,
        "bnb_4bit_quant_type": "nf4",
        "bnb_4bit_compute_dtype": torch.bfloat16,
    }
    llm_instance.set_quantization_config(new_quantization_config)
    assert llm_instance.quantization_config == new_quantization_config


# Test for setting model_id
```

```
def test_llm_set_model_id(llm_instance):
    new_model_id = "EleutherAI/gpt-neo-2.7B"
    llm_instance.set_model_id(new_model_id)
    assert llm_instance.model_id == new_model_id


# Test for setting model
@patch(
    "swarms.models.huggingface.AutoModelForCausalLM.from_pretrained"
)
def test_llm_set_model(mock_model, llm_instance):
    mock_model.return_value = "mocked model"
    llm_instance.set_model(mock_model)
    assert llm_instance.model == "mocked model"


# Test for setting tokenizer
@patch("swarms.models.huggingface.AutoTokenizer.from_pretrained")
def test_llm_set_tokenizer(mock_tokenizer, llm_instance):
    mock_tokenizer.return_value = "mocked tokenizer"
    llm_instance.set_tokenizer(mock_tokenizer)
    assert llm_instance.tokenizer == "mocked tokenizer"


# Test for setting logger
def test_llm_set_logger(llm_instance):
    new_logger = logging.getLogger("test_logger")

    llm_instance.set_logger(new_logger)

    assert llm_instance.logger == new_logger




# Test for saving model

@patch("torch.save")

def test_llm_save_model(mock_save, llm_instance):

    llm_instance.save_model("path/to/save")

    mock_save.assert_called_once()




# Test for print_dashboard

@patch("builtins.print")

def test_llm_print_dashboard(mock_print, llm_instance):

    llm_instance.print_dashboard("test task")

    mock_print.assert_called()




# Test for __call__ method

@patch("swarms.models.huggingface.HuggingfaceLLM.run")

def test_llm_call(mock_run, llm_instance):

    mock_run.return_value = "mocked output"

    result = llm_instance("test task")

    assert result == "mocked output"
```