Dig into Optimum-onnx

02/2026

Plan

- Main challenges when exporting a model for genai
- Technical Answers from optimum-onnx
- Easy way to export
- Proposals

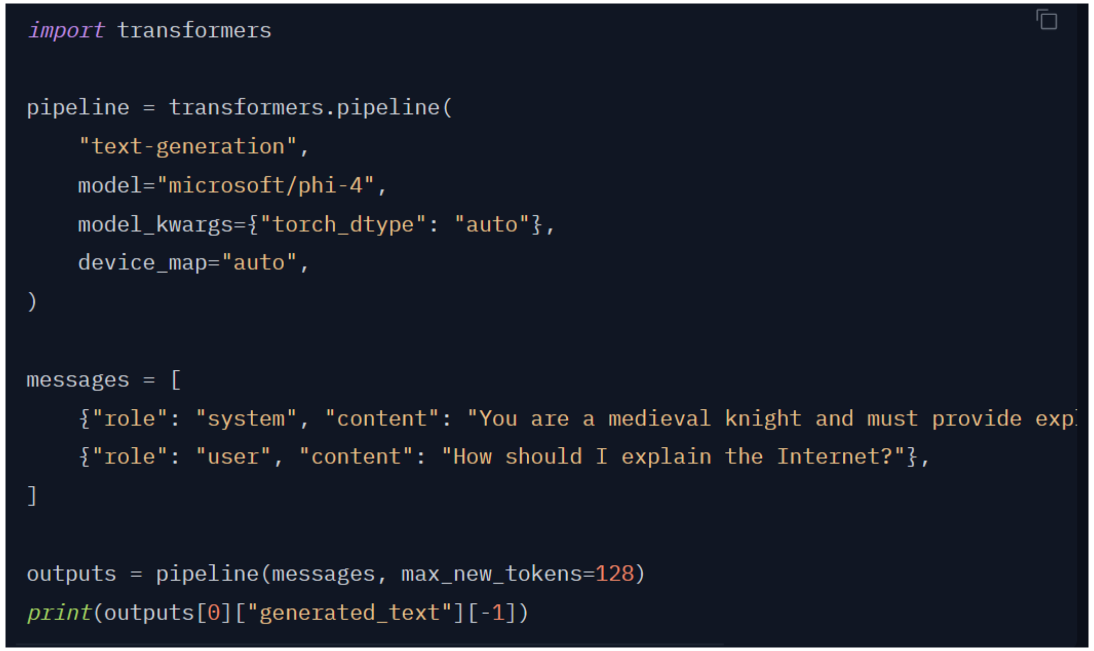
Challenge 1: generate to forward

What the users see is not what needs to be exported.

The user calls `pipeline` or `generate` but he needs to export method `forward`.

He has to guess:

- A set of inputs including a cache he usually never sees
- The corresponding dynamic shapes.



```python
import transformers

pipeline = transformers.pipeline(
    "text-generation",
    model="microsoft/phi-4",
    model_kwargs={"torch_dtype": "auto"},
    device_map="auto",
)

messages = [
    {"role": "system", "content": "You are a medieval knight and must provide exp
    {"role": "user", "content": "How should I explain the Internet?"},
]

outputs = pipeline(messages, max_new_tokens=128)
print(outputs[0]["generated_text"][-1])
```

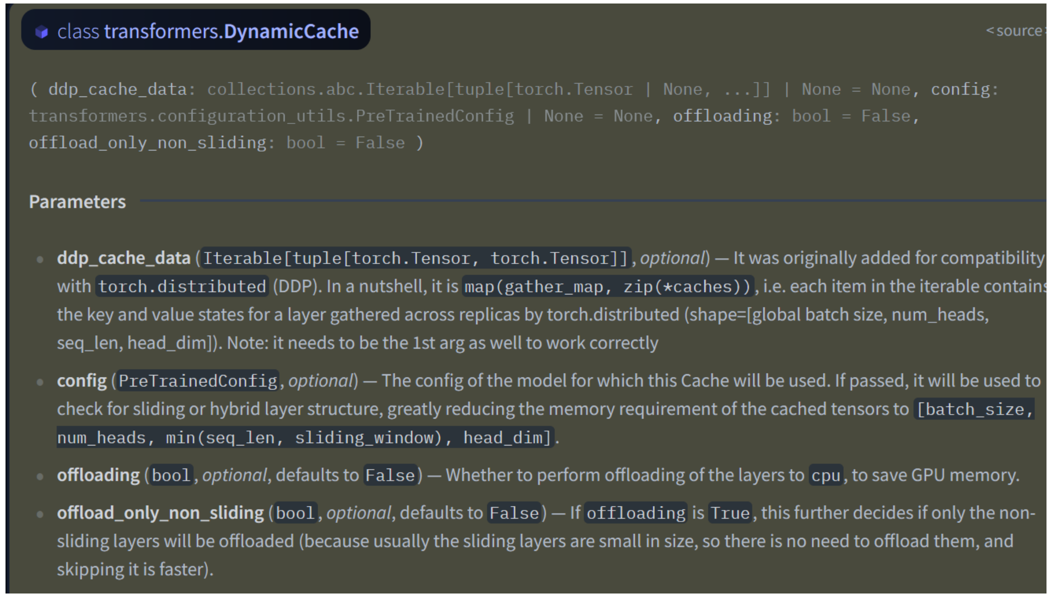
Challenge 2: flattening nested structure

The model takes a cache (DynamicCache, EncoderDecoderCache, ...) as input. The exporter needs a way to retrieve a flat list of tensors from any non native container.

onnxruntime only takes flat list of tensors as well.

This goes through flattening.

It is implemented in package optree.

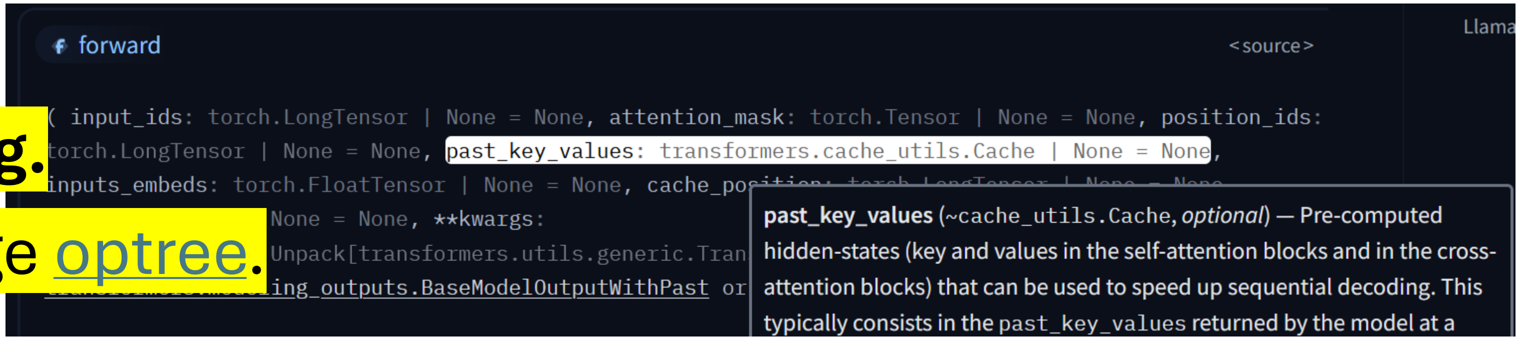


class transformers.DynamicCache <source

```
( ddp_cache_data: collections.abc.Iterable[tuple[torch.Tensor | None, ...]] | None = None, config:
transformers.configuration_utils.PreTrainedConfig | None = None, offloading: bool = False,
offload_only_non_sliding: bool = False )
```

Parameters

- **ddp_cache_data** (`Iterable[tuple[torch.Tensor, torch.Tensor]]`, *optional*) — It was originally added for compatibility with `torch.distributed` (DDP). In a nutshell, it is `map(gather_map, zip(*caches))`, i.e. each item in the iterable contains the key and value states for a layer gathered across replicas by torch.distributed (shape=[global batch size, num_heads, seq_len, head_dim]). Note: it needs to be the 1st arg as well to work correctly
- **config** (`PreTrainedConfig`, *optional*) — The config of the model for which this Cache will be used. If passed, it will be used to check for sliding or hybrid layer structure, greatly reducing the memory requirement of the cached tensors to `[batch_size, num_heads, min(seq_len, sliding_window), head_dim]`.
- **offloading** (`bool`, *optional*, defaults to `False`) — Whether to perform offloading of the layers to `cpu`, to save GPU memory.
- **offload_only_non_sliding** (`bool`, *optional*, defaults to `False`) — If `offloading` is `True`, this further decides if only the non-sliding layers will be offloaded (because usually the sliding layers are small in size, so there is no need to offload them, and skipping it is faster).



forward <source>

Llama

```
( input_ids: torch.LongTensor | None = None, attention_mask: torch.Tensor | None = None, position_ids:
torch.LongTensor | None = None, past_key_values: transformers.cache_utils.Cache | None = None,
inputs_embeds: torch.FloatTensor | None = None, cache_po
None = None, **kwargs:
Unpack[transformers.utils.generic.Tran
ing_outputs.BaseModelOutputWithPast or
```

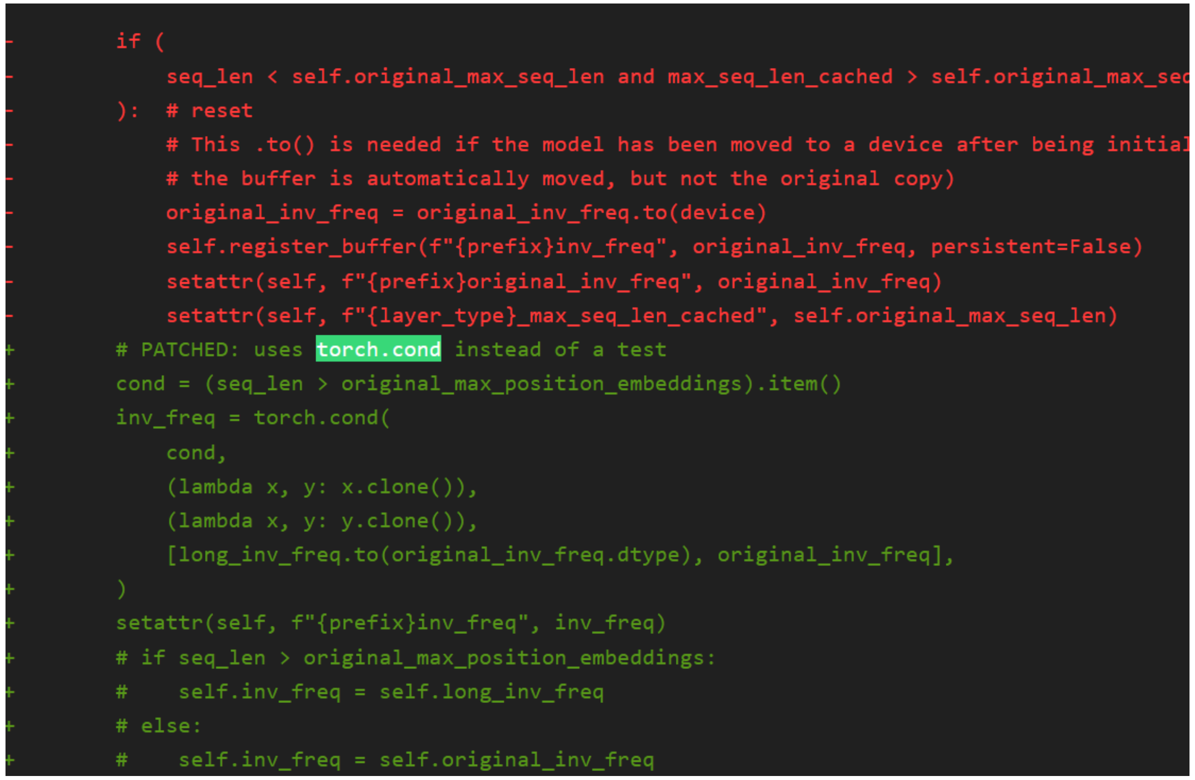
past_key_values (~`cache_utils.Cache`, *optional*) — Pre-computed hidden-states (key and values in the self-attention blocks and in the cross-attention blocks) that can be used to speed up sequential decoding. This typically consists in the `past_key_values` returned by the model at a

Challenge 3: rewriting for control flow

Tests and loops can be exported if the condition or the number of iteration is known before the execution (independant from the data).

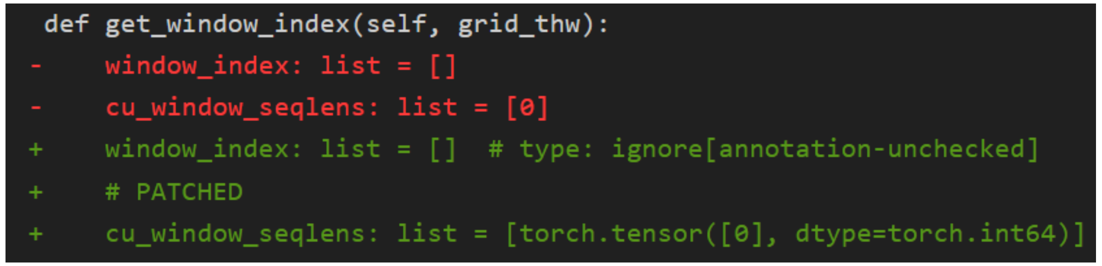
Otherwise, they need to be rewritten.

But the rewriting can never be commited to the source code as it is not trivial and incompatible with training.



```
-        if (
-            seq_len < self.original_max_seq_len and max_seq_len_cached > self.original_max_sec
-        ):  # reset
-            # This .to() is needed if the model has been moved to a device after being initial
-            # the buffer is automatically moved, but not the original copy)
-            original_inv_freq = original_inv_freq.to(device)
-            self.register_buffer(f"{prefix}inv_freq", original_inv_freq, persistent=False)
-            setattr(self, f"{prefix}original_inv_freq", original_inv_freq)
-            setattr(self, f"{layer_type}_max_seq_len_cached", self.original_max_seq_len)
+        # PATCHED: uses torch.cond instead of a test
+        cond = (seq_len > original_max_position_embeddings).item()
+        inv_freq = torch.cond(
+            cond,
+            (lambda x, y: x.clone()),
+            (lambda x, y: y.clone()),
+            [long_inv_freq.to(original_inv_freq.dtype), original_inv_freq],
+        )
+        setattr(self, f"{prefix}inv_freq", inv_freq)
+        # if seq_len > original_max_position_embeddings:
+        #     self.inv_freq = self.long_inv_freq
+        # else:
+        #     self.inv_freq = self.original_inv_freq
```

A line patched for Qwen 2.5-VL: replace an int by a tensor

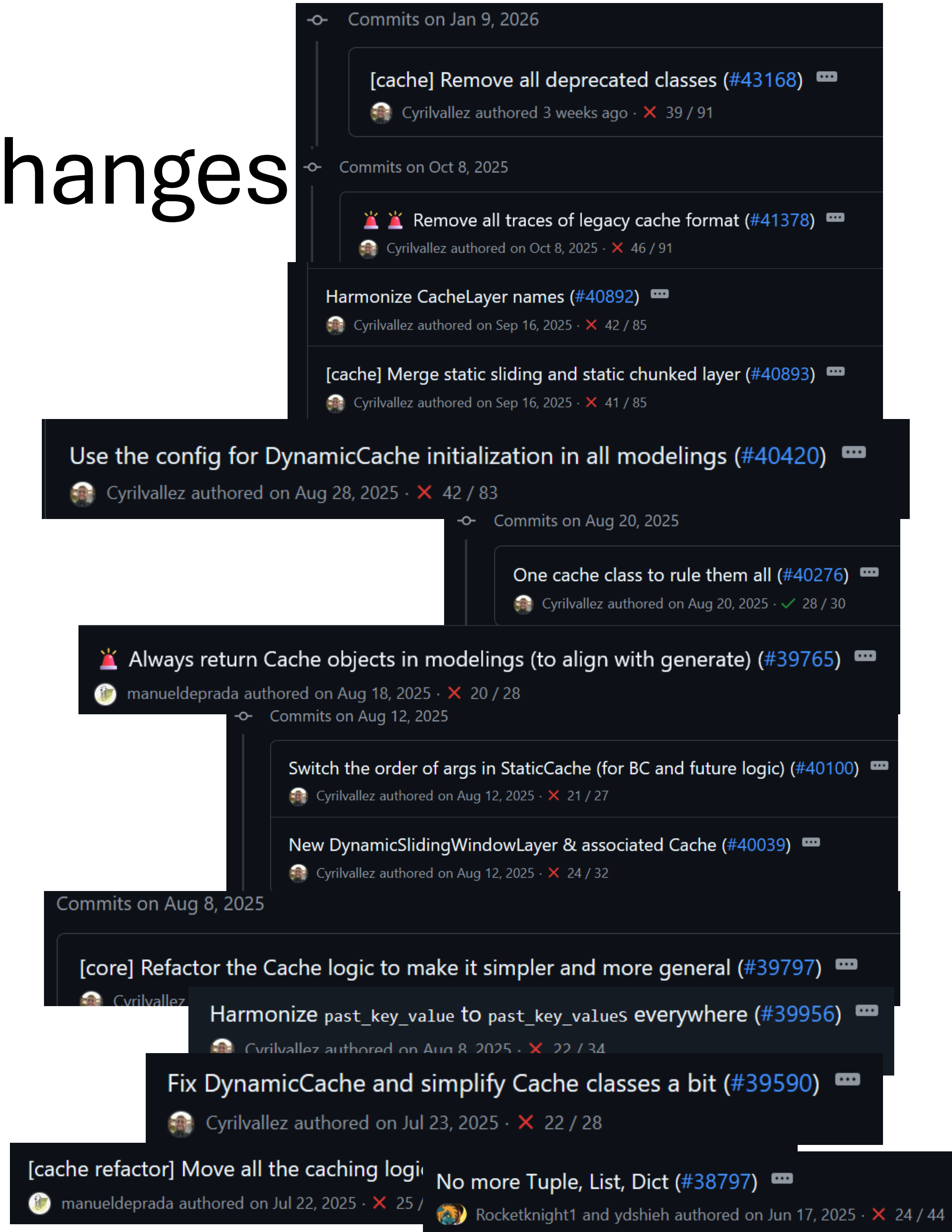


```
  def get_window_index(self, grid_thw):
-     window_index: list = []
-     cu_window_seqlens: list = [0]
+     window_index: list = []  # type: ignore[annotation-unchecked]
+     # PATCHED
+     cu_window_seqlens: list = [torch.tensor([0], dtype=torch.int64)]
```

Challenge 4: a year of BC changes

In 2025, transformers never stopped refactoring the caches breaking many times the export patches we wrote.

Pytorch fixed many bugs we reported related to dynamic shapes. In 2025H1, it was impossible to export a model with a dynamic batch size if the batch size was 1. It is now and some code needs refactoring to be simplified.



Commits on Jan 9, 2026

[cache] Remove all deprecated classes (#43168)
Cyrilvallez authored 3 weeks ago · ✕ 39 / 91

Commits on Oct 8, 2025

🚨 🚨 Remove all traces of legacy cache format (#41378)
Cyrilvallez authored on Oct 8, 2025 · ✕ 46 / 91

Harmonize CacheLayer names (#40892)
Cyrilvallez authored on Sep 16, 2025 · ✕ 42 / 85

[cache] Merge static sliding and static chunked layer (#40893)
Cyrilvallez authored on Sep 16, 2025 · ✕ 41 / 85

Use the config for DynamicCache initialization in all modelings (#40420)
Cyrilvallez authored on Aug 28, 2025 · ✕ 42 / 83

Commits on Aug 20, 2025

One cache class to rule them all (#40276)
Cyrilvallez authored on Aug 20, 2025 · ✓ 28 / 30

🚨 Always return Cache objects in modelings (to align with generate) (#39765)
manueldeprada authored on Aug 18, 2025 · ✕ 20 / 28

Commits on Aug 12, 2025

Switch the order of args in StaticCache (for BC and future logic) (#40100)
Cyrilvallez authored on Aug 12, 2025 · ✕ 21 / 27

New DynamicSlidingWindowLayer & associated Cache (#40039)
Cyrilvallez authored on Aug 12, 2025 · ✕ 24 / 32

Commits on Aug 8, 2025

[core] Refactor the Cache logic to make it simpler and more general (#39797)
Cyrilvallez

Harmonize `past_key_value` to `past_key_values` everywhere (#39956)
Cyrilvallez authored on Aug 8, 2025 · ✕ 22 / 34

Fix DynamicCache and simplify Cache classes a bit (#39590)
Cyrilvallez authored on Jul 23, 2025 · ✕ 22 / 28

[cache refactor] Move all the caching logi
manueldeprada authored on Jul 22, 2025 · ✕ 25 /

No more Tuple, List, Dict (#38797)
Rocketknight1 and ydshieh authored on Jun 17, 2025 · ✕ 24 / 44

Challenge 5: Input generation for multimodal models

LLMs handling text only support random inputs.

Models taking images, audio and text as inputs do not. Special tokens associated to the image/audio cannot be randomly inserted.

Models have specific constraints. It is difficult to build generic inputs for every model.

Technical Answers from Optimum-onnx

Optimum-onnx makes it easier to export many model by:

- Providing dynamic axes and good set of inputs to the exporter
- Flattening Caches
- Patching some pieces in the model (it replaces methods)
- Optimizing the exported model (onnxsim, onnxruntime patterns)

Out of scope but still implemented by optimum-onnx

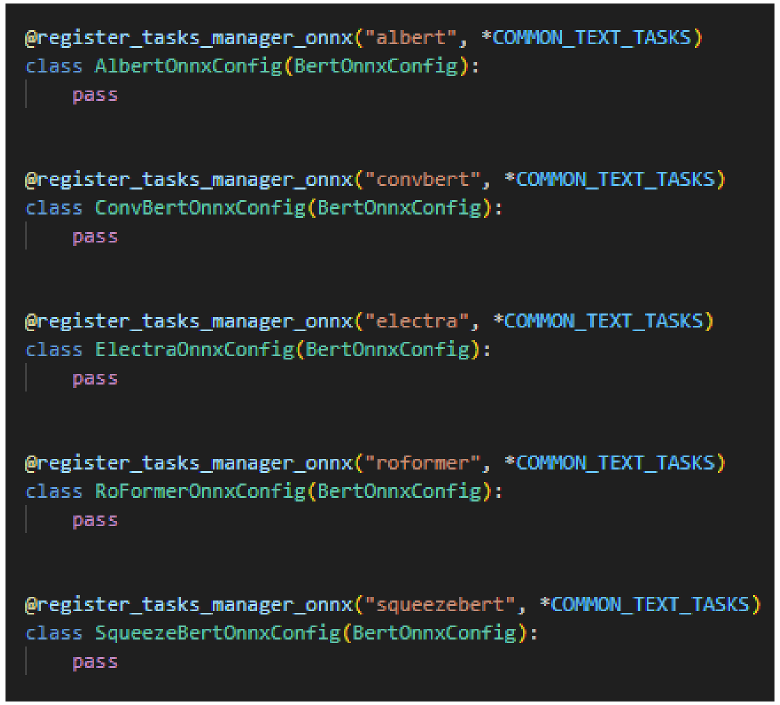
- Model optimization with onnxruntime.transformers.optimizer.optimize_model
- Model quantization with QDQQuantizer or ONNXQuantizer

Optimum-onnx: inputs

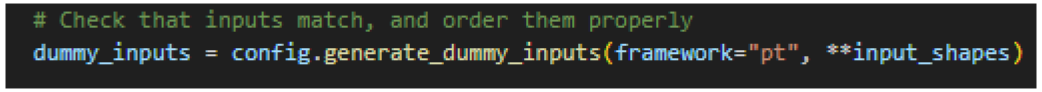
A model is mapped to a task.

The task is mapped to a set of inputs.

optimum/exporters/onnx/model_configs.py



```
@register_tasks_manager_onnx("albert", *COMMON_TEXT_TASKS)
class AlbertOnnxConfig(BertOnnxConfig):
    pass


@register_tasks_manager_onnx("convbert", *COMMON_TEXT_TASKS)
class ConvBertOnnxConfig(BertOnnxConfig):
    pass


@register_tasks_manager_onnx("electra", *COMMON_TEXT_TASKS)
class ElectraOnnxConfig(BertOnnxConfig):
    pass


@register_tasks_manager_onnx("roformer", *COMMON_TEXT_TASKS)
class RoFormerOnnxConfig(BertOnnxConfig):
    pass


@register_tasks_manager_onnx("squeezebert", *COMMON_TEXT_TASKS)
class SqueezeBertOnnxConfig(BertOnnxConfig):
    pass
```



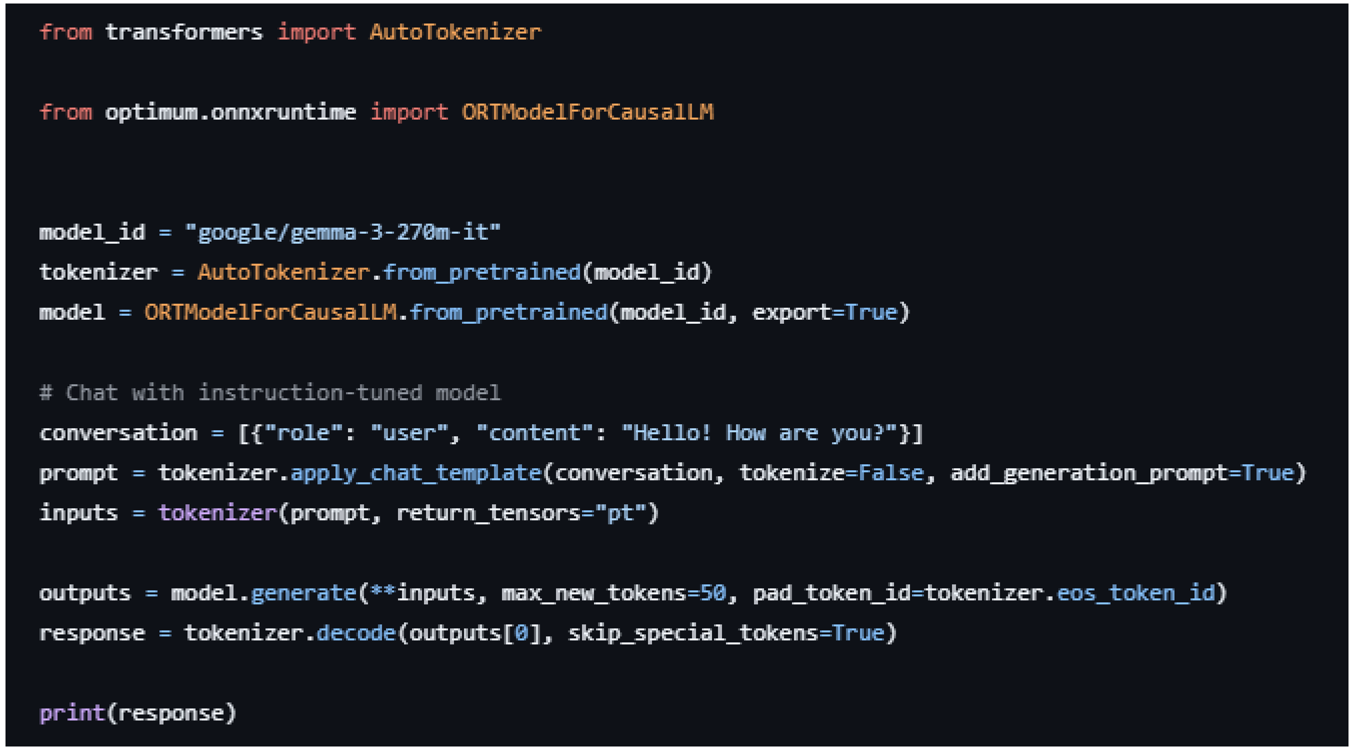
```
# Check that inputs match, and order them properly
dummy_inputs = config.generate_dummy_inputs(framework="pt", **input_shapes)
```

optimum-onnx: flattening

This is handled through the dummy inputs.

optimum-onnx implements a class ORTModelForCausalLM which overwrites forward method to flatten/unflatten caches before calling transformers.

optimum/onnxruntime/modeling_decoder.py

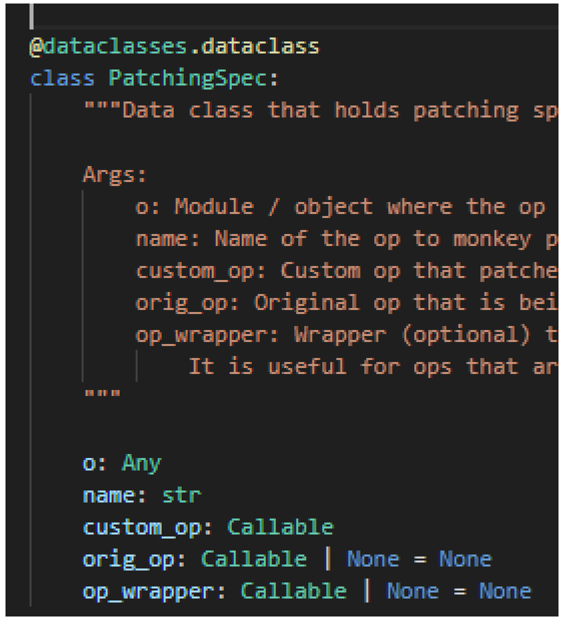


```python
from transformers import AutoTokenizer

from optimum.onnxruntime import ORTModelForCausalLM


model_id = "google/gemma-3-270m-it"
tokenizer = AutoTokenizer.from_pretrained(model_id)
model = ORTModelForCausalLM.from_pretrained(model_id, export=True)

# Chat with instruction-tuned model
conversation = [{"role": "user", "content": "Hello! How are you?"}]
prompt = tokenizer.apply_chat_template(conversation, tokenize=False, add_generation_prompt=True)
inputs = tokenizer(prompt, return_tensors="pt")

outputs = model.generate(**inputs, max_new_tokens=50, pad_token_id=tokenizer.eos_token_id)
response = tokenizer.decode(outputs[0], skip_special_tokens=True)

print(response)
```

Optimum-onnx: patches for torchscript

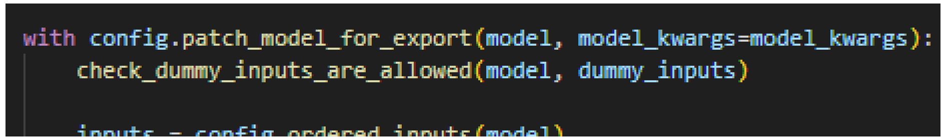
- Built on top of:
  - AutoModel from transformers
  - Dummy Input Generator from optimum
  - Implements patches for torch and transformers

optimum-onnx/optimum/exporters/onnx/model_patcher.py at main · huggingface/optimum-onnx



```
@dataclasses.dataclass
class PatchingSpec:
    """Data class that holds patching sp

    Args:
        o: Module / object where the op
        name: Name of the op to monkey p
        custom_op: Custom op that patche
        orig_op: Original op that is bei
        op_wrapper: Wrapper (optional) t
            It is useful for ops that ar
    """

    o: Any
    name: str
    custom_op: Callable
    orig_op: Callable | None = None
    op_wrapper: Callable | None = None
```

Many methods are replaced before exporting with patches.



```
with config.patch_model_for_export(model, model_kwargs=model_kwargs):
    check_dummy_inputs_are_allowed(model, dummy_inputs)

    inputs = config.ordered_inputs(model)
```

Optimum-onnx: limitations

- Only DynamicCache and EncoderDecoderCache are supported.
- Only onnx exporter based on torchscript is supported, patches are written for this exporter
- Code split into 3 repositories: transformers, optimum, optimum-onnx
- Rampup time is significant to support a new model, a new task, a new cache class, a new patch.
- Unnecessary code to maintain such as ORTModelForCausalLM.

Optimum-onnx: does not scale well

To export a new model:

- We need a set of dummy inputs.
- We need good dynamic shapes.
- We may need new flattening.
- We may need new patches.

Usually, it always misses something.

This approach is not scalable.

To scale, we need:

- A way to quickly support new patches.
- We need flattening/unflattening implemented in one place and not in many places in the code.
- We need to infer dummy inputs and dynamic scales.

Optimum-onnx: summary

KEEP

Testing code and ideas.

- It goes over many models.
- It goes over multiple versions of transformers.

Command line

- Many users use it.

Patches: we need them

Quantization?

REMOVE

Code for dummies and dynamic shapes

- Not scalable
- Not centralized in one place

Cache

- Flatten Cache Classes is all over the place

Optimization

- Already done by the new exporter

pytorch 2.11

[torch.export-based ONNX Exporter — PyTorch main documentation]

Easy way to export

Only change for a new model: the code snippet

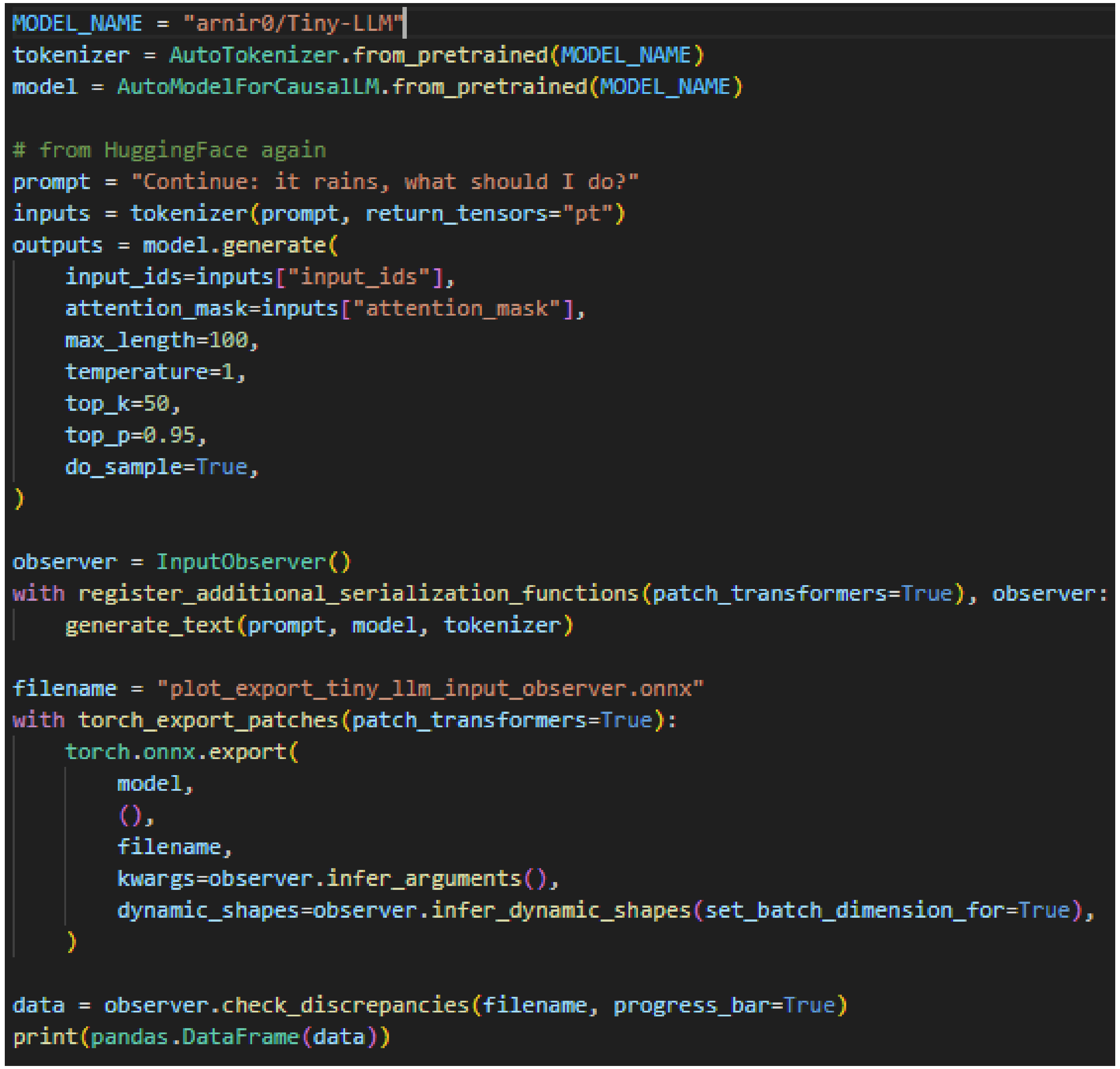
Snippet coming from HuggingFace Hub

Forward Inputs/Ouputs captured

Export code does not change any more, supports any custom cache if it can be flattened.

Export, usually with patches

Checks discrepancies

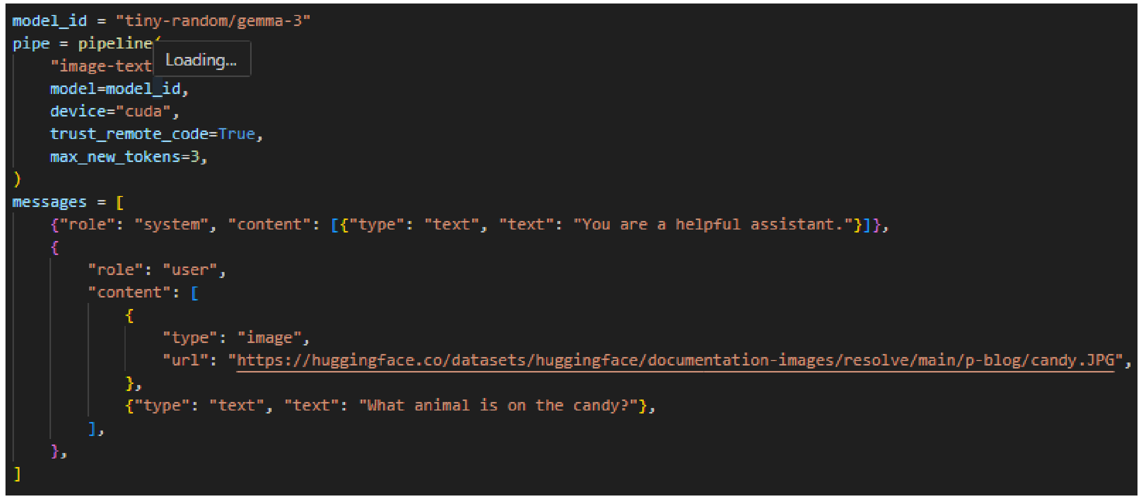


```python
MODEL_NAME = "arnir0/Tiny-LLM"
tokenizer = AutoTokenizer.from_pretrained(MODEL_NAME)
model = AutoModelForCausalLM.from_pretrained(MODEL_NAME)

# from HuggingFace again
prompt = "Continue: it rains, what should I do?"
inputs = tokenizer(prompt, return_tensors="pt")
outputs = model.generate(
    input_ids=inputs["input_ids"],
    attention_mask=inputs["attention_mask"],
    max_length=100,
    temperature=1,
    top_k=50,
    top_p=0.95,
    do_sample=True,
)

observer = InputObserver()
with register_additional_serialization_functions(patch_transformers=True), observer:
    generate_text(prompt, model, tokenizer)

filename = "plot_export_tiny_llm_input_observer.onnx"
with torch_export_patches(patch_transformers=True):
    torch.onnx.export(
        model,
        (),
        filename,
        kwargs=observer.infer_arguments(),
        dynamic_shapes=observer.infer_dynamic_shapes(set_batch_dimension_for=True),
    )

data = observer.check_discrepancies(filename, progress_bar=True)
print(pandas.DataFrame(data))
```

Gemma3

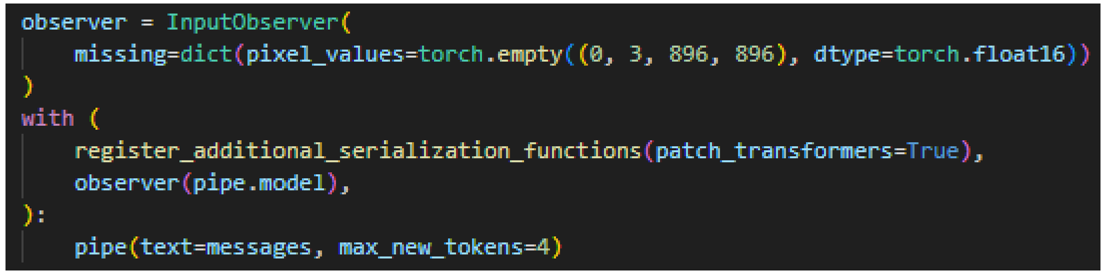
pixel_values only appears on the first call to foward method. Then it is removed and a cache is added in the second call. We tell the observer to add an empty tensor if missing. This is one unsolved case today (unsolved = not fully automated).

setup



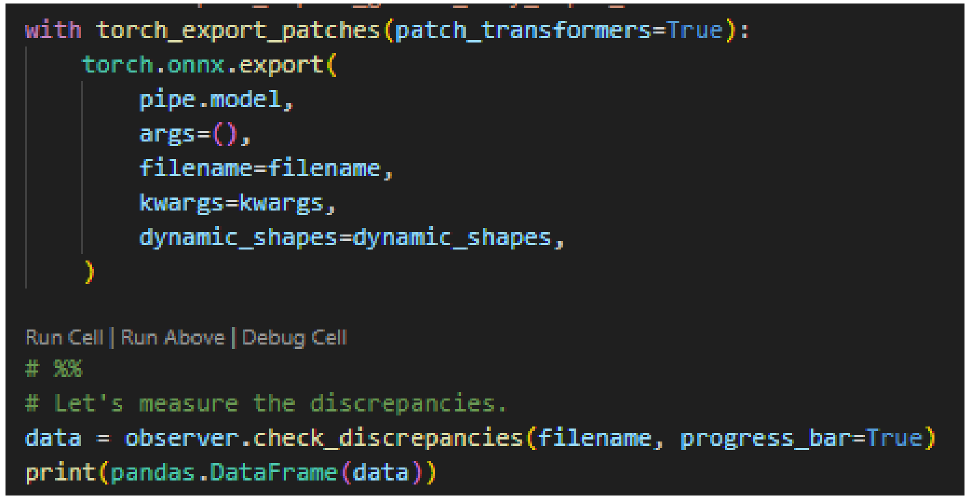
```python
model_id = "tiny-random/gemma-3"
pipe = pipeline(
    "image-text
    model=model_id,
    device="cuda",
    trust_remote_code=True,
    max_new_tokens=3,
)
messages = [
    {"role": "system", "content": [{"type": "text", "text": "You are a helpful assistant."}]},
    {
        "role": "user",
        "content": [
            {
                "type": "image",
                "url": "https://huggingface.co/datasets/huggingface/documentation-images/resolve/main/p-blog/candy.JPG",
            },
            {"type": "text", "text": "What animal is on the candy?"},
        ],
    },
]
```

observe



```python
observer = InputObserver(
    missing=dict(pixel_values=torch.empty((0, 3, 896, 896), dtype=torch.float16))
)
with (
    register_additional_serialization_functions(patch_transformers=True),
    observer(pipe.model),
):
    pipe(text=messages, max_new_tokens=4)
```

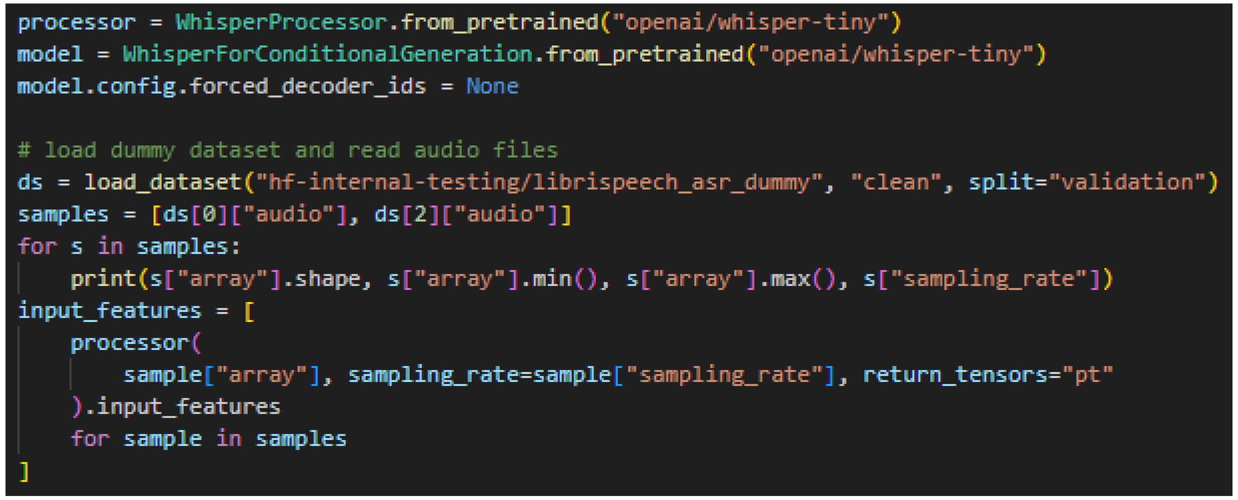
export



```python
with torch_export_patches(patch_transformers=True):
    torch.onnx.export(
        pipe.model,
        args=(),
        filename=filename,
        kwargs=kwargs,
        dynamic_shapes=dynamic_shapes,
    )

# %%
# Let's measure the discrepancies.
data = observer.check_discrepancies(filename, progress_bar=True)
print(pandas.DataFrame(data))
```

Whisper: 2 observers

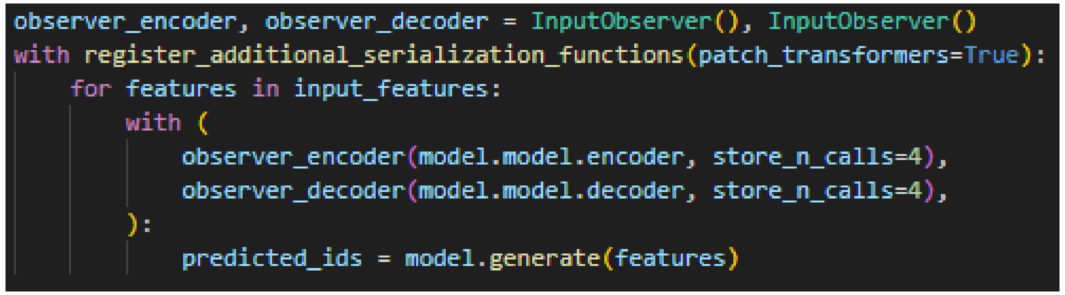
setup



```python
processor = WhisperProcessor.from_pretrained("openai/whisper-tiny")
model = WhisperForConditionalGeneration.from_pretrained("openai/whisper-tiny")
model.config.forced_decoder_ids = None

# load dummy dataset and read audio files
ds = load_dataset("hf-internal-testing/librispeech_asr_dummy", "clean", split="validation")
samples = [ds[0]["audio"], ds[2]["audio"]]
for s in samples:
    print(s["array"].shape, s["array"].min(), s["array"].max(), s["sampling_rate"])
input_features = [
    processor(
        sample["array"], sampling_rate=sample["sampling_rate"], return_tensors="pt"
    ).input_features
    for sample in samples
]
```

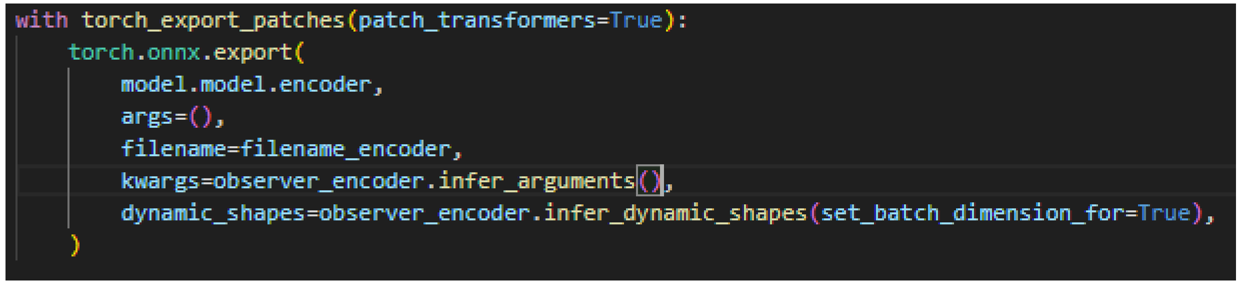
Any part of the model can be exported. More than one observer can be set up at the same time.

observe



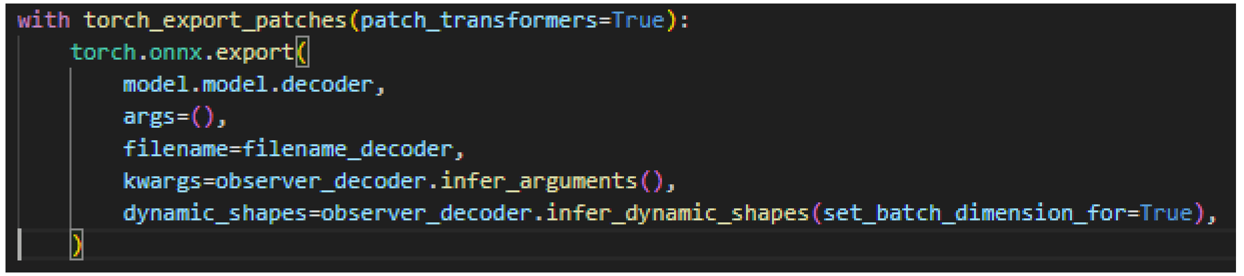
```python
observer_encoder, observer_decoder = InputObserver(), InputObserver()
with register_additional_serialization_functions(patch_transformers=True):
    for features in input_features:
        with (
            observer_encoder(model.model.encoder, store_n_calls=4),
            observer_decoder(model.model.decoder, store_n_calls=4),
        ):
            predicted_ids = model.generate(features)
```

export encoder



```python
with torch_export_patches(patch_transformers=True):
    torch.onnx.export(
        model.model.encoder,
        args=(),
        filename=filename_encoder,
        kwargs=observer_encoder.infer_arguments(),
        dynamic_shapes=observer_encoder.infer_dynamic_shapes(set_batch_dimension_for=True),
    )
```

export decoder



```python
with torch_export_patches(patch_transformers=True):
    torch.onnx.export(
        model.model.decoder,
        args=(),
        filename=filename_decoder,
        kwargs=observer_decoder.infer_arguments(),
        dynamic_shapes=observer_decoder.infer_dynamic_shapes(set_batch_dimension_for=True),
    )
```

Proposals

Why now? Why optimum-onnx?

Why now?

- API in pytorch and transformers have recently stabilized (transformers 5.0, pytorch 2.10). There was almost a breaking change every 2 weeks before that.
- We can now infer dynamic shapes and arguments to reduce the code to write.
- onnxscript supports many optimizations related to onnxruntime (so called contrib-ops)

Why optimum-onnx?

- Optimum-onnx is widely used to convert model into ONNX. Let’s not change something which works.
- Some models only work with a specific version of transformers. The solution needs to work with many versions of transformers.
- HuggingFace team makes changes to optimum-onnx.

Proposal 1: keeps everything as is

Trying:

- Enable dynamo export by xadupre · Pull Request #113 · huggingface/optimum-onnx
- Issues with dynamic axes, the logic is different now

Proposal 2: keep optimum-onnx API

Preserve until deprecation

- Current stack for old exporter
- New stack for new models

Keep

- API: the command line + the export function
- Patches
- Testing

Add

- If --dynamo is added, switch to a new stack based on InputObserver
- Inputs flattening using torch.utils._pytree / optree

Proposal 3: do not keep optimum-onnx

- Optimum-onnx remains only for torchscript
- Everybody handles patches on their own.
  - Onnxruntime
    - onnxruntime/onnxruntime/python/tools/transformers/models/torch_export_patches at main · microsoft/onnxruntime
  - Olive
    - Olive/olive/passes/onnx/conversion.py at main · microsoft/Olive
  - Onnx-diagnostic (research project)
    - onnx-diagnostic 0.9.1 documentation
    - Patches Explained - onnx-diagnostic 0.9.1 documentation