(supports-multimodal)=

Multi-Modal Support

This document walks you through the steps to extend a basic model so that it accepts [multi-modal inputs](#multimodal-inputs).

1. Update the base vLLM model

It is assumed that you have already implemented the model in vLLM according to [these steps](#new-model-basic).
Further update the model as follows:

- Reserve a keyword parameter in {meth}`~torch.nn.Module.forward` for each input tensor that corresponds to a multi-modal input, as shown in the following example:

  ```diff
    def forward(
        self,
        input_ids: torch.Tensor,
        positions: torch.Tensor,
        kv_caches: List[torch.Tensor],
        attn_metadata: AttentionMetadata,
  +     pixel_values: torch.Tensor,
    ) -> SamplerOutput:
  ```

More conveniently, you can simply pass `**kwargs` to the {meth}`~torch.nn.Module.forward` method and retrieve the keyword parameters for multimodal inputs from it.

- Implement {meth}`~vllm.model_executor.models.interfaces.SupportsMultiModal.get_multimodal_embeddings` that returns the embeddings from running the multimodal inputs through the multimodal tokenizer of the model. Below we provide a boilerplate of a typical implementation pattern, but feel free to adjust it to your own needs.

```python
class YourModelForImage2Seq(nn.Module):
    ...

    def _process_image_input(self, image_input: YourModelImageInputs) -> torch.Tensor:

        assert self.vision_encoder is not None
        image_features = self.vision_encoder(image_input)
        return self.multi_modal_projector(image_features)

    def get_multimodal_embeddings(self, **kwargs: object) -> Optional[NestedTensors]:

        # Validate the multimodal input keyword arguments
        image_input = self._parse_and_validate_image_input(**kwargs)
        if image_input is None:
            return None

        # Run multimodal inputs through encoder and projector
```

        vision_embeddings = self._process_image_input(image_input)

        return vision_embeddings

    ```

    :::{important}

    The returned `multimodal_embeddings` must be either a **3D {class}`torch.Tensor`** of shape `(num_items, feature_size, hidden_size)`, or a **list / tuple of 2D {class}`torch.Tensor`'s** of shape `(feature_size, hidden_size)`, so that `multimodal_embeddings[i]` retrieves the embeddings generated from the `i`-th multimodal data item (e.g, image) of the request.

    :::

- Implement {meth}`~vllm.model_executor.models.interfaces.SupportsMultiModal.get_input_embeddings` to merge `multimodal_embeddings` with text embeddings from the `input_ids`. If input processing for the model is implemented correctly (see sections below), then you can leverage the utility function we provide to easily merge the embeddings.

    ```python

    from .utils import merge_multimodal_embeddings

    class YourModelForImage2Seq(nn.Module):

        ...

        def get_input_embeddings(

            self,

            input_ids: torch.Tensor,

            multimodal_embeddings: Optional[NestedTensors] = None,

```python
    ) -> torch.Tensor:

        # `get_input_embeddings` should already be implemented for the language
        # model as one of the requirements of basic vLLM model implementation.
        inputs_embeds = self.language_model.get_input_embeddings(input_ids)

        if multimodal_embeddings is not None:
            inputs_embeds = merge_multimodal_embeddings(
                input_ids=input_ids,
                inputs_embeds=inputs_embeds,
                multimodal_embeddings=multimodal_embeddings,
                placeholder_token_id=self.config.image_token_index)

        return inputs_embeds
```

- Once the above steps are done, update the model class with the {class}`~vllm.model_executor.models.interfaces.SupportsMultiModal` interface.

```diff
+ from vllm.model_executor.models.interfaces import SupportsMultiModal

- class YourModelForImage2Seq(nn.Module):
+ class YourModelForImage2Seq(nn.Module, SupportsMultiModal):
```

:::{note}

The model class does not have to be named {code}`*ForCausalLM`.

Check out [the HuggingFace Transformers documentation](https://huggingface.co/docs/transformers/model_doc/auto#multimodal) for some examples.

:::

2. Specify processing information

Next, create a subclass of {class}`~vllm.multimodal.processing.BaseProcessingInfo` to provide basic information related to HF processing.

Maximum number of input items

You need to override the abstract method {meth}`~vllm.multimodal.processing.BaseProcessingInfo.get_supported_mm_limits` to return the maximum number of input items for each modality supported by the model.

For example, if the model supports any number of images but only one video per prompt:

```python
def get_supported_mm_limits(self) -> Mapping[str, Optional[int]]:
    return {"image": None, "video": 1}
```

Maximum number of placeholder feature tokens

Also, override the abstract method

{meth}`~vllm.multimodal.processing.BaseProcessingInfo.get_mm_max_tokens_per_item`

to return the maximum number of placeholder feature tokens per input item for each modality.

When calling the model, the output embeddings from the visual encoder are assigned to the input positions

containing placeholder feature tokens. Therefore, the number of placeholder feature tokens should be equal

to the size of the output embeddings.

:::::{tab-set}

::::{tab-item} Basic example: LLaVA

:sync: llava

Looking at the code of HF's `LlavaForConditionalGeneration`:

```python
#
https://github.com/huggingface/transformers/blob/v4.47.1/src/transformers/models/llava/modeling_llava.py#L530-L544
n_image_tokens = (input_ids == self.config.image_token_index).sum().item()
n_image_features = image_features.shape[0] * image_features.shape[1]

if n_image_tokens != n_image_features:
    raise ValueError(
            f"Image features and image tokens do not match: tokens: {n_image_tokens}, features {n_image_features}"
    )
```

```
special_image_mask = (
    (input_ids == self.config.image_token_index)
    .unsqueeze(-1)
    .expand_as(inputs_embeds)
    .to(inputs_embeds.device)
)
image_features = image_features.to(inputs_embeds.device, inputs_embeds.dtype)
inputs_embeds = inputs_embeds.masked_scatter(special_image_mask, image_features)
```

The number of placeholder feature tokens per image is `image_features.shape[1]`.

`image_features` is calculated inside the `get_image_features` method:

```python
#
https://github.com/huggingface/transformers/blob/v4.47.1/src/transformers/models/llava/modeling_llava.py#L290-L300
image_outputs = self.vision_tower(pixel_values, output_hidden_states=True)

selected_image_feature = image_outputs.hidden_states[vision_feature_layer]
if vision_feature_select_strategy == "default":
    selected_image_feature = selected_image_feature[:, 1:]
elif vision_feature_select_strategy == "full":
    selected_image_feature = selected_image_feature
else:
    raise ValueError(f"Unexpected select feature strategy: {self.config.vision_feature_select_strategy}")
```

image_features = self.multi_modal_projector(selected_image_feature)

return image_features

```

We can infer that `image_features.shape[1]` is based on `image_outputs.hidden_states.shape[1]` from the vision tower (`CLIPVisionModel` for the [`llava-hf/llava-1.5-7b-hf`](https://huggingface.co/llava-hf/llava-1.5-7b-hf) model).

Moreover, we only need the sequence length (the second dimension of the tensor) to get `image_features.shape[1]`.

The sequence length is determined by the initial hidden states in `CLIPVisionTransformer` since the attention mechanism doesn't change the sequence length of the output hidden states.

```python
# https://github.com/huggingface/transformers/blob/v4.47.1/src/transformers/models/clip/modeling_clip.py#L1094-L1102
hidden_states = self.embeddings(pixel_values, interpolate_pos_encoding=interpolate_pos_encoding)
hidden_states = self.pre_layrnorm(hidden_states)

encoder_outputs = self.encoder(
    inputs_embeds=hidden_states,
    output_attentions=output_attentions,
    output_hidden_states=output_hidden_states,
    return_dict=return_dict,

)
```

To find the sequence length, we turn to the code of `CLIPVisionEmbeddings`:

```python
# https://github.com/huggingface/transformers/blob/v4.47.1/src/transformers/models/clip/modeling_clip.py#L247-L257
target_dtype = self.patch_embedding.weight.dtype
patch_embeds = self.patch_embedding(pixel_values.to(dtype=target_dtype))  # shape = [*, width, grid, grid]
patch_embeds = patch_embeds.flatten(2).transpose(1, 2)

class_embeds = self.class_embedding.expand(batch_size, 1, -1)
embeddings = torch.cat([class_embeds, patch_embeds], dim=1)
if interpolate_pos_encoding:
    embeddings = embeddings + self.interpolate_pos_encoding(embeddings, height, width)
else:
    embeddings = embeddings + self.position_embedding(self.position_ids)
return embeddings
```

We can infer that `embeddings.shape[1] == self.num_positions`, where

```python
#

```
https://github.com/huggingface/transformers/blob/v4.47.1/src/transformers/models/clip/modeling_clip.py#L195-L196
self.num_patches = (self.image_size // self.patch_size) ** 2
self.num_positions = self.num_patches + 1
```

Overall, the number of placeholder feature tokens for an image can be calculated as:

```python
def get_num_image_tokens(
    self,
    *,
    image_width: int,
    image_height: int,
) -> int:
    hf_config = self.get_hf_config()
    hf_processor = self.get_hf_processor()

    image_size = hf_config.vision_config.image_size
    patch_size = hf_config.vision_config.patch_size

    num_image_tokens = (image_size // patch_size) ** 2 + 1
    if hf_processor.vision_feature_select_strategy == "default":
        num_image_tokens -= 1

    return num_image_tokens
```

Notice that the number of image tokens doesn't depend on the image width and height.

So, we can calculate the maximum number of image tokens using any image size:

```python
def get_image_size_with_most_features(self) -> ImageSize:
    hf_config = self.get_hf_config()
    width = height = hf_config.image_size
    return ImageSize(width=width, height=height)

def get_max_image_tokens(self) -> int:
    target_width, target_height = self.get_image_size_with_most_features()

    return self.get_num_image_tokens(
        image_width=target_width,
        image_height=target_height,
    )
```

And thus, we can override the method as:

```python
def get_mm_max_tokens_per_item(
    self,
    seq_len: int,
    mm_counts: Mapping[str, int],
) -> Mapping[str, int]:
```

    return {"image": self.get_max_image_tokens()}
```

:::{note}
Our [actual code](gh-file:vllm/model_executor/models/llava.py) is more abstracted to support vision encoders other than CLIP.
:::

::::
:::::

## 3. Specify dummy inputs

Then, inherit {class}`~vllm.multimodal.profiling.BaseDummyInputsBuilder` to construct dummy inputs for
HF processing as well as memory profiling.

### For memory profiling

Override the abstract method {meth}`~vllm.multimodal.profiling.BaseDummyInputsBuilder.get_dummy_processor_inputs`
to construct dummy inputs for memory profiling. This dummy input should result in the worst-case memory usage of
the model so that vLLM can reserve the correct amount of memory for it.

Assuming that the memory usage increases with the number of tokens, the dummy input can be constructed based
```

on the code for {meth}`~vllm.multimodal.processing.BaseProcessingInfo.get_mm_max_tokens_per_item`.

::::{tab-set}

:::{tab-item} Basic example: LLaVA

:sync: llava

Making use of the `get_image_size_with_most_features` method implemented in the previous section:

```python
def get_dummy_processor_inputs(
    self,
    seq_len: int,
    mm_counts: Mapping[str, int],
) -> ProcessorInputs:
    num_images = mm_counts.get("image", 0)

    processor = self.info.get_hf_processor()
    image_token = processor.image_token

    hf_config = self.get_hf_config()
    target_width, target_height = self.info.get_image_size_with_most_features()

    mm_data = {
        "image":
        self._get_dummy_images(width=target_width,
                               height=target_height,
```

```
                            num_images=num_images)
    }

    return ProcessorInputs(
        prompt_text=image_token * num_images,
        mm_data=mm_data,
    )
```

:::

::::

4. Specify processing details

Afterwards, create a subclass of {class}`~vllm.multimodal.processing.BaseMultiModalProcessor` to fill in the missing details about HF processing.

:::{seealso}
[Multi-Modal Data Processing](#mm-processing)
:::

Multi-modal fields

Override {class}`~vllm.multimodal.processing.BaseMultiModalProcessor._get_mm_fields_config` to return a schema of the tensors outputted by the HF processor that are related to the input multi-modal items.

:::::{tab-set}

::::{tab-item} Basic example: LLaVA

:sync: llava

Looking at the model's `forward` method:

```python
#
https://github.com/huggingface/transformers/blob/v4.47.1/src/transformers/models/llava/modeling_ll
ava.py#L387-L404
def forward(
    self,
    input_ids: torch.LongTensor = None,
    pixel_values: torch.FloatTensor = None,
    attention_mask: Optional[torch.Tensor] = None,
    position_ids: Optional[torch.LongTensor] = None,
    past_key_values: Optional[List[torch.FloatTensor]] = None,
    inputs_embeds: Optional[torch.FloatTensor] = None,
    vision_feature_layer: Optional[int] = None,
    vision_feature_select_strategy: Optional[str] = None,
    labels: Optional[torch.LongTensor] = None,
    use_cache: Optional[bool] = None,
    output_attentions: Optional[bool] = None,
    output_hidden_states: Optional[bool] = None,
    return_dict: Optional[bool] = None,
    cache_position: Optional[torch.LongTensor] = None,
    num_logits_to_keep: int = 0,
```

) -> Union[Tuple, LlavaCausalLMOutputWithPast]:

```

The only related keyword argument is `pixel_values` which directly corresponds to input images.

The shape of `pixel_values` is `(N, C, H, W)` where `N` is the number of images.

So, we override the method as follows:

```python
def _get_mm_fields_config(
    self,
    hf_inputs: BatchFeature,
    hf_processor_mm_kwargs: Mapping[str, object],
) -> Mapping[str, MultiModalFieldConfig]:
    return dict(
        pixel_values=MultiModalFieldConfig.batched("image"),
    )
```

:::{note}

Our [actual code](gh-file:vllm/model_executor/models/llava.py) additionally supports pre-computed image embeddings, which can be passed to be model via the `image_embeds` argument.

:::

::::

:::::
```

Prompt replacements

Override
{class}`~vllm.multimodal.processing.BaseMultiModalProcessor._get_prompt_replacements` to
return a list of {class}`~vllm.multimodal.processing.PromptReplacement` instances.

Each {class}`~vllm.multimodal.processing.PromptReplacement` instance specifies a find-and-replace
operation performed by the HF processor.

::::{tab-set}
:::{tab-item} Basic example: LLaVA
:sync: llava

Looking at HF's `LlavaProcessor`:

```python
#
https://github.com/huggingface/transformers/blob/v4.47.1/src/transformers/models/llava/processing_
llava.py#L167-L170
prompt_strings = []
for sample in text:
    sample = sample.replace(self.image_token, self.image_token * num_image_tokens)
    prompt_strings.append(sample)
```

It simply repeats each input `image_token` a number of times equal to the number of placeholder feature tokens (`num_image_tokens`).

Based on this, we override the method as follows:

```python
def _get_prompt_replacements(
    self,
    mm_items: MultiModalDataItems,
    hf_processor_mm_kwargs: Mapping[str, object],
    out_mm_kwargs: MultiModalKwargs,
) -> list[PromptReplacement]:
    hf_config = self.info.get_hf_config()
    image_token_id = hf_config.image_token_index

    def get_replacement(item_idx: int):
        images = mm_items.get_items("image", ImageProcessorItems)

        image_size = images.get_image_size(item_idx)
        num_image_tokens = self.info.get_num_image_tokens(
            image_width=image_size.width,
            image_height=image_size.height,
        )

        return [image_token_id] * num_image_tokens

    return [
        PromptReplacement(
            modality="image",
```

            target=[image_token_id],
            replacement=get_replacement,
        ),
    ]
```

:::
::::

## 5. Register processor-related classes

After you have defined {class}`~vllm.multimodal.processing.BaseProcessingInfo` (Step 2),
{class}`~vllm.multimodal.profiling.BaseDummyInputsBuilder` (Step 3),
and {class}`~vllm.multimodal.processing.BaseMultiModalProcessor` (Step 4),
decorate the model class with {meth}`MULTIMODAL_REGISTRY.register_processor
&lt;vllm.multimodal.registry.MultiModalRegistry.register_processor&gt;`
to register them to the multi-modal registry:

```diff
  from vllm.model_executor.models.interfaces import SupportsMultiModal
+ from vllm.multimodal import MULTIMODAL_REGISTRY

+ @MULTIMODAL_REGISTRY.register_processor(YourMultiModalProcessor,
+                                         info=YourProcessingInfo,
+                                         dummy_inputs=YourDummyInputsBuilder)
  class YourModelForImage2Seq(nn.Module, SupportsMultiModal):
```
```