(new-model-basic)=

Implementing a Basic Model

This guide walks you through the steps to implement a basic vLLM model.

1. Bring your model code

First, clone the PyTorch model code from the source repository.
For instance, vLLM's [OPT model](gh-file:vllm/model_executor/models/opt.py) was adapted from HuggingFace's [modeling_opt.py](https://github.com/huggingface/transformers/blob/main/src/transformers/models/opt/modeling_opt.py) file.

:::{warning}
Make sure to review and adhere to the original code's copyright and licensing terms!
:::

2. Make your code compatible with vLLM

To ensure compatibility with vLLM, your model must meet the following requirements:

Initialization Code

All vLLM modules within the model must include a `prefix` argument in their constructor. This `prefix` is typically the full name of the module in the model's state dictionary and is crucial for:

- Runtime support: vLLM's attention operators are registered in a model's state by their full names. Each attention operator must have a unique prefix as its layer name to avoid conflicts.
- Non-uniform quantization support: A quantized checkpoint can selectively quantize certain layers while keeping others in full precision. By providing the `prefix` during initialization, vLLM can match the current layer's `prefix` with the quantization configuration to determine if the layer should be initialized in quantized mode.

The initialization code should look like this:

```python
from torch import nn
from vllm.config import VllmConfig
from vllm.attention import Attention

class MyAttention(nn.Module):
    def __init__(self, vllm_config: VllmConfig, prefix: str):
        super().__init__()
        self.attn = Attention(prefix=f"{prefix}.attn")

class MyDecoderLayer(nn.Module):
    def __init__(self, vllm_config: VllmConfig, prefix: str):
        super().__init__()
        self.self_attn = MyAttention(prefix=f"{prefix}.self_attn")

class MyModel(nn.Module):
    def __init__(self, vllm_config: VllmConfig, prefix: str):
        super().__init__()
        self.layers = nn.ModuleList(
            [MyDecoderLayer(vllm_config, prefix=f"{prefix}.layers.{i}") for i in range(vllm_config.model_config.hf_config.num_hidden_layers)]
        )

class MyModelForCausalLM(nn.Module):
    def __init__(self, vllm_config: VllmConfig, prefix: str = ""):
        super().__init__()
        self.model = MyModel(vllm_config, prefix=f"{prefix}.model")
```

Computation Code

- Add a `get_input_embeddings` method inside `MyModel` module that returns the text embeddings given `input_ids`. This is equivalent to directly calling the text embedding layer, but provides a unified interface in case `MyModel` is used within a composite multimodal model.

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

    def get_input_embeddings(self, input_ids: torch.Tensor) -> torch.Tensor:
        ...
```

- Rewrite the {meth}`~torch.nn.Module.forward` method of your model to remove any unnecessary code, such as training-specific code. Modify the input parameters to treat `input_ids` and `positions` as flattened tensors with a single batch size dimension, without a max-sequence length dimension.

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

:::{note}
Currently, vLLM supports the basic multi-head attention mechanism and its variant with rotary positional embeddings.
If your model employs a different attention mechanism, you will need to implement a new attention layer in vLLM.
:::

For reference, check out our [Llama implementation](gh-file:vllm/model_executor/models/llama.py). vLLM already supports a large number of models. It is recommended to find a model similar to yours and adapt it to your model's architecture. Check out <gh-dir:vllm/model_executor/models> for more examples.

3. (Optional) Implement tensor parallelism and quantization support

If your model is too large to fit into a single GPU, you can use tensor parallelism to manage it.

To do this, substitute your model's linear and embedding layers with their tensor-parallel versions.

For the embedding layer, you can simply replace {class}`torch.nn.Embedding` with `VocabParallelEmbedding`. For the output LM head, you can use `ParallelLMHead`.

When it comes to the linear layers, we provide the following options to parallelize them:

- `ReplicatedLinear`: Replicates the inputs and weights across multiple GPUs. No memory saving.
- `RowParallelLinear`: The input tensor is partitioned along the hidden dimension. The weight matrix is partitioned along the rows (input dimension). An *all-reduce* operation is performed after the matrix multiplication to reduce the results. Typically used for the second FFN layer and the output linear transformation of the attention layer.
- `ColumnParallelLinear`: The input tensor is replicated. The weight matrix is partitioned along the columns (output dimension). The result is partitioned along the column dimension. Typically used for the first FFN layer and the separated QKV transformation of the attention layer in the original Transformer.
- `MergedColumnParallelLinear`: Column-parallel linear that merges multiple `ColumnParallelLinear` operators. Typically used for the first FFN layer with weighted activation functions (e.g., SiLU). This class handles the sharded weight loading logic of multiple weight matrices.
- `QKVParallelLinear`: Parallel linear layer for the query, key, and value projections of the multi-head and grouped-query attention mechanisms. When number of key/value heads are less than the world size, this class replicates the key/value heads properly. This class handles the weight loading and replication of the weight matrices.

Note that all the linear layers above take `linear_method` as an input. vLLM will set this parameter according to different quantization schemes to support weight quantization.

4. Implement the weight loading logic

You now need to implement the `load_weights` method in your `*ForCausalLM` class.
This method should load the weights from the HuggingFace's checkpoint file and assign them to the corresponding layers in your model. Specifically, for `MergedColumnParallelLinear` and `QKVParallelLinear` layers, if the original model has separated weight matrices, you need to load the different parts separately.

5. Register your model

See [this page](#new-model-registration) for instructions on how to register your new model to be used by vLLM.

Frequently Asked Questions

How to support models with interleaving sliding windows?

For models with interleaving sliding windows (e.g. `google/gemma-2-2b-it` and `mistralai/Ministral-8B-Instruct-2410`), the scheduler will treat the model as a full-attention model, i.e., kv-cache of all tokens will not be dropped. This is to make sure prefix caching works with these models. Sliding window only appears as a parameter to the attention kernel computation.

To support a model with interleaving sliding windows, we need to take care of the following details:

- Make sure [this line](https://github.com/vllm-project/vllm/blob/996357e4808ca5eab97d4c97c7d25b3073f46aab/vllm/config.py#L308) evaluates `has_interleaved_attention` to `True` for this model, and set `self.hf_text_config.interleaved_sliding_window` to the format of interleaving sliding windows the model can understand. Then, `self.hf_text_config.sliding_window` will be deleted, and the model will be treated as a full-attention model.

- In the modeling code, parse the correct sliding window value for every layer, and pass it to the attention layer's `per_layer_sliding_window` argument. For reference, check [this line](https://github.com/vllm-project/vllm/blob/996357e4808ca5eab97d4c97c7d25b3073f46aab/vllm/model_executor/models/llama.py#L171).

With these two steps, interleave sliding windows should work with the model.